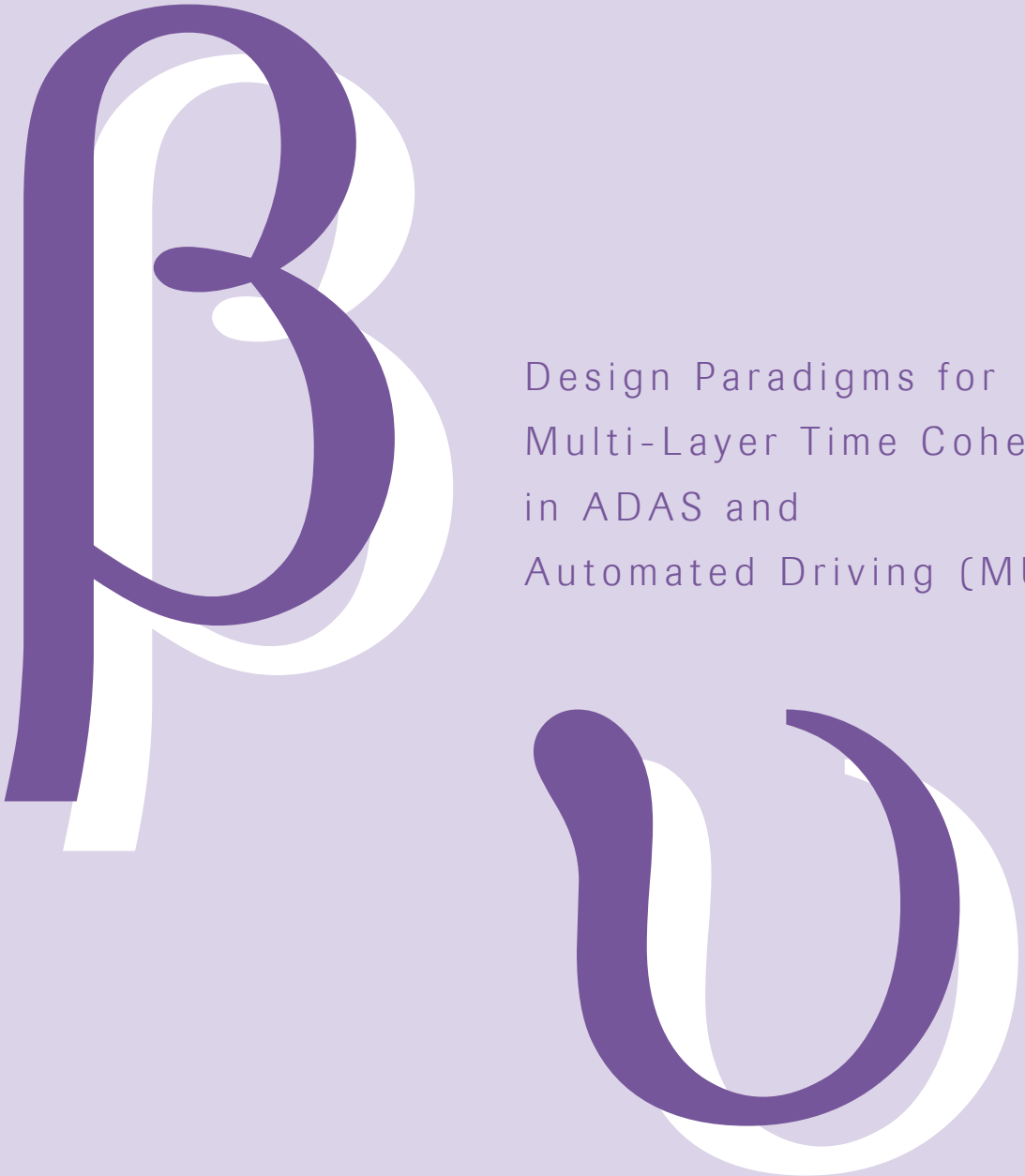
FORSCHUNGSVEREINIGUNG AUTOMOBILTECHNIK E.V.

FAT-SCHRIFTENREIHE

FAT 302



Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC)

VDA Verband der Automobilindustrie

Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC)

Forschungsstelle
OFFIS e.V.

Autoren
Eckard Böde (Project Lead)
Matthias Büker
Werner Damm (Scientific Lead)
Günter Ehmen (WP3 Lead)
Martin Fränzle (Scientific Co-Lead)
Sebastian Gerwinn
Thomas Goodfellow
Kim Grüttner (WP2 Lead)
Bernhard Josko
Björn Koopmann
Thomas Peikenkamp
Frank Poppen
Philipp Reinkemeier
Michael Siegel
Ingo Stierand (WP1 Lead)

Das Forschungsprojekt wurde mit Mitteln der Forschungsvereinigung Automobiltechnik e. V. (FAT) gefördert.

Contents

Preface

Executive Summary

The application of digital control in the automotive domain clearly follows an evolution with increasing complexity of both covered functions and their interaction. Advanced Driver Assistance Systems (ADAS) and Automated Driving (AD) comprise modular interacting software components that typically build upon a layered architecture as shown in Figure 0.1. As these components are typically developed by different teams, using different tools for different functional purposes and building upon different models of computation, an integration of all components guaranteeing the satisfaction of all requirements and a coherent handling of timing properties is a major challenge within the process of developing such systems.

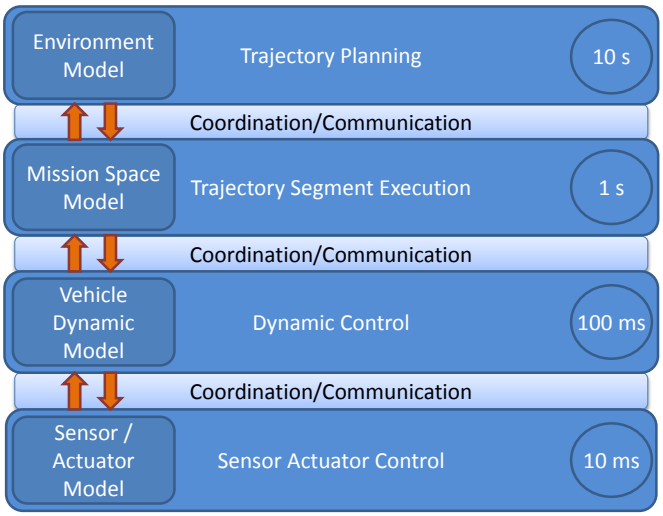


Figure 0.1.: MULTIC Layered architecture example for ADAS

For a continuous treatment of time on all four levels of the layered architecture, such design and programming paradigms and interfaces must be integrated into a common semantic framework. In particular, the consistent description of the layer transitions with regard to their time behavior must be achieved by means of adequate combinations of specification, modeling and programming approaches, as well as suitable analysis mechanisms. MULTIC addressed these topics in three main phases:

Phase 1 *"Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving"*

Phase 2 *"Design Approach for Multi-Layer Time Coherency in ADAS and Automated Driving"*

Phase 3 *"Demonstrator for Multi-Layer Time Coherency in ADAS and Automated Driving"*

Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving

System behavior typically subsumes different aspects, such as the functional aspect, timing, safety, and others. Here, we focus on the timing aspect, and a main objective is to elaborate on approaches that allow for capturing all relevant timing phenomena and effects for such systems in a consistent and coherent way across all system layers and functional domains, as well as ensuring traceability along the design process. Particularly the heterogeneity of Models of Computation involved in ADAS/AD design and their interaction plays a key role in this effort. Moreover, the nature and complexity of information sources and their processing in such systems introduces complex data paths that are hard to track down and has large potential to cause consistency issues.

The project has identified four important design paradigms to support the development of future ADAS/AD (see Figure 0.2).

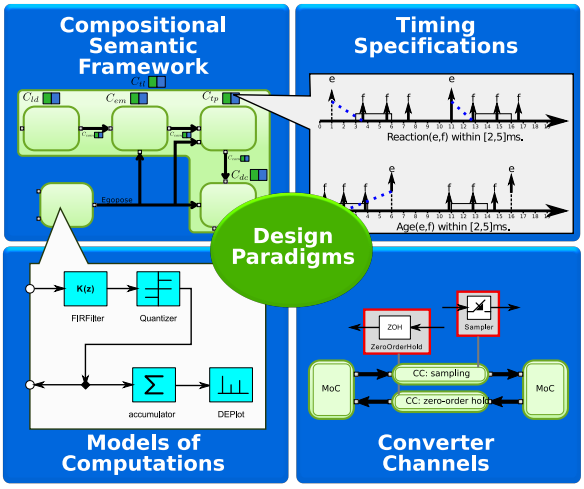


Figure 0.2.: MULTIC Design Paradigms

The first design paradigm is the **"Compositional Semantic Framework"**, which provides an architectural basis for system design by introducing a generic hierarchical component model. The model is intended to be instantiated with existing modeling languages (such as SysML) and tools by defining how to cast the individual modeling artifacts into artifacts of the conceptual model.

The component model serves as a carrier for the other three design paradigms:

- It defines a notion of contracts as a particular kind of (assume-guarantee style) specifications. Contracts give modeling entities and their interaction formal semantics, and enables one to reason about verification of the individual design steps, such as decomposition, refinement and realization.
- It supports the integration of different Models of Computation (MoC) for different parts within one system design.
- And it supports the integration of heterogeneous MoC using different abstraction of time through Converter Channel (CC).

The Compositional Semantic Framework enables to set up design processes where systems are

incrementally refined. It provides concepts allowing to relate different viewpoints like functional modeling and the technical realization as well as different abstraction levels.

The second design paradigm **"Timing Specifications"** instantiates contract based design for the timing aspect of the system design. It inherits timing specifications from well established frameworks such as AUTOSAR, and defines extensions where needed in order to enable coherent reasoning about timing within complex scenarios.

The third design paradigm **"Models of Computation" (MoC)** provide the formal basis for implementing components with well-defined execution semantics. MoCs support integration into different time domains (untimed, continuous time, discrete time and synchronous time) and are instantiated by concrete implementation languages such as Matlab/Simulink and C/C++, as well as by domain-specific languages such as various data flow, control flow and automaton-based languages. Given a particular MoC, its integration into the semantic framework requires two ingredients. First, a mapping of the interfaces of the MoC onto the (conceptual) component model defines how models are embedded into the component model. Second, a mapping of the potentially different notions of time must be specified, which is supported by the next paradigm.

The fourth design paradigm **"Converter Channels" (CC)** concerns the interaction between components. This is particularly important when components with different MoCs shall interact with each other, such as components implemented in C++ with component implemented using Matlab/Simulink. Various Converter Channels for "state of practice" interaction semantics are discussed, such as used for discrete-continuous signal coupling and vice versa.

Design Approach for Multi-Layer Time Coherency in ADAS and Automated Driving

We exemplify the application of the four proposed design paradigms along a simplified design process for a generic ADAS case study. A coherent treatment of time across different abstraction and development layers is demonstrated.

The simplified design process covers the main steps from a high-level functional model with top-level timing specifications down to an AUTOSAR implementation model and consists of four design phases (depicted in Figure 0.3). The first two phases (Functional Level A and B) cover the functional system design. The latter two phases (Technical Level B and C) consider the technical realization of the system. The distinction between functional and technical perspective complies with standard design frameworks such as the envisioned V-model exploited in ISO 26262 for the automotive domain. Also model refinement along different abstraction levels has been proved to be useful, for example to reflect organizational boundaries between OEM and suppliers.

The first design phase (Functional Level A) deals with the initial functional decomposition of the top-level functional architecture which is often considered as a typical entry point into the functional design, where engineers and other stakeholders agree on the initial architecture, consisting of the top-level functional components and their interaction. This entry point also complies with safety relevant standard approaches as defined for example in the ISO 26262, where the initial safety design leads to a "functional safety architecture".

The discussion of this first design phase (Functional Level A) concentrates on the two design paradigms "Compositional Semantic Framework" and "Timing Specifications". The application of "component based design" is exemplified by using the SysML modeling language for the design. The functional decomposition is complemented by a discussion of the corresponding decomposition of timing specifications. We start with the initial specifications given for the top-level system and show the application of contract based design methods in order to safeguard the decomposition, which ensures that the top-level specifications are fulfilled by the composition of the specifications

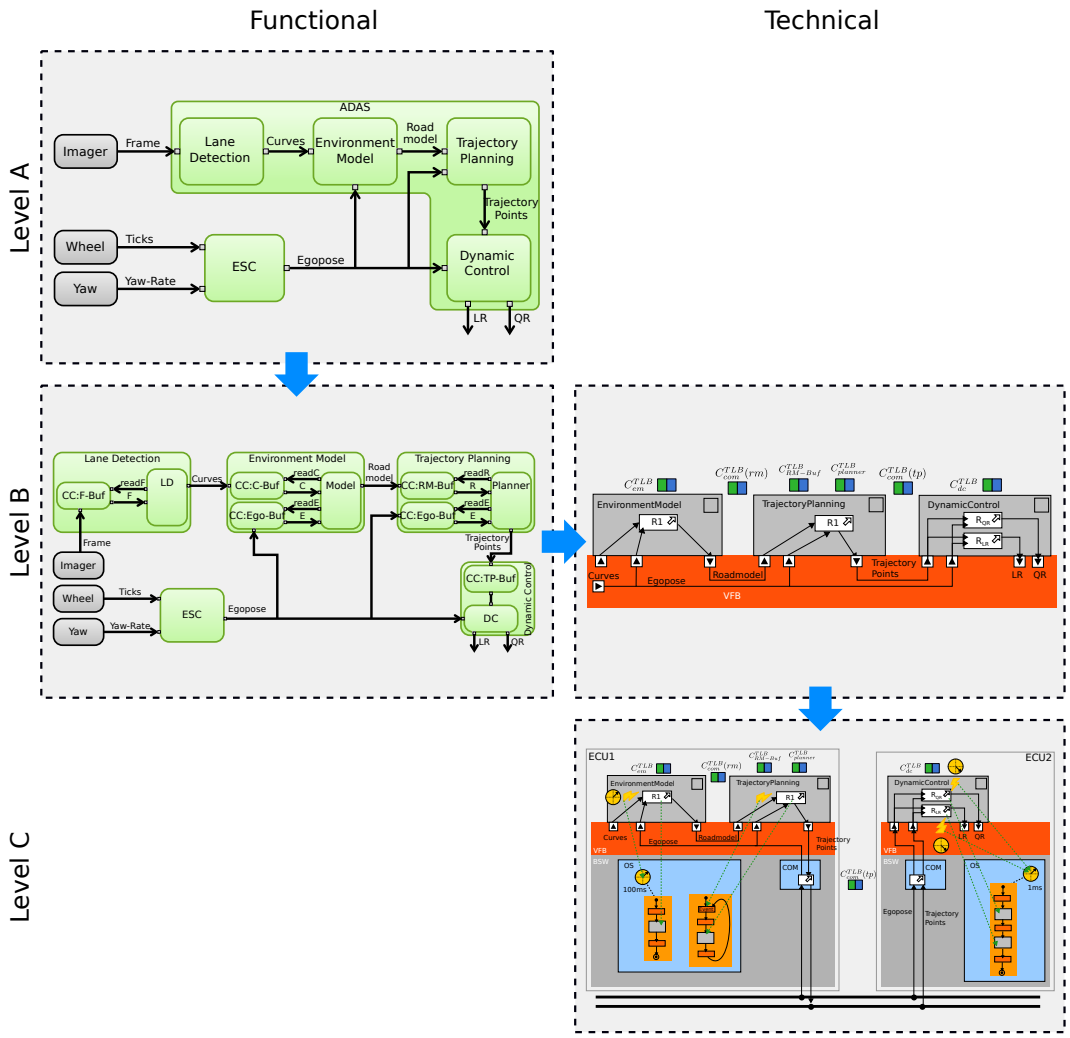


Figure 0.3.: Design Process Overview

of its subsystem components. This is instantiated by a set of well-defined natural language patterns helping engineers to lower the effort for expressing (timing) specifications. The Timing Specifications represent a conservative extension of existing specification languages, for example such as TADL. The context in which they are used, which is an instance of contract based design, however represents a non-conservative extension with respect to the state of practice.

The second design phase (Functional Level B) concentrates on the design paradigms "Models of Computation" and "Converter Channels". At this phase, initial versions of the functional models are developed. Within this report, we omit details on the implementation of the models themselves and it is considered sufficient to discuss the application of the design paradigms on a rather flat model with its component interaction. Hence, we assume that the identified functional units are already atomic in that they contain a single MoC and are written with a single modeling/programming language. This already enables us to introduce the application of Converter Channels.

The goal of the first technical design model (Technical Level B) is to map the functional design to

software and hardware entities. We assume that AUTOSAR, and particularly a AUTOSAR Virtual Function Bus (VFB) model, is used to model such a technical view. AUTOSAR focuses entirely on the software architecture of a system and only touches topics related to the underlying hardware where necessary. Therefore, functions are mapped to software components, respectively to runnables, which are the smallest executable code-fragments provided by a software component. Contrary to the previous functional designs, where mainly data flow and Converter Channels realizing the buffering of input data have been addressed, an AUTOSAR model also defines control flow aspects. For specifying timing contracts in an AUTOSAR model, the AUTOSAR modeling language already provides some elements, which are defined in the AUTOSAR Timing Extension and which are used her as well.

Within the next design phase (Technical Level C) the AUTOSAR VFB model will be refined resulting in *ECU Configuration Description* and *System Configuration Description*. These descriptions contain a mapping of SWCs to ECUs, thereby inducing a mapping of VFB communication to either ECU-local communication or to network-technology specific communication mechanisms such as CAN frames. Finally, the ECU configuration is created by defining configurations for each needed module of the basic software stack. From a timing point of view, we focus on the communication stack and how data items which are sent and received by the SWCs propagate through the communication stack. Contract-based timing specifications on the application part above the AUTOSAR RTE are considered as well as the satisfaction of contracts by an implementations of the runnables of a SWC are discussed.

We close this part with an identification of missing expressiveness in the AUTOSAR timing specification language, and a brief overview on standardized information exchange between different tools to realize contract traceability.

Demonstrator for Multi-Layer Time Coherency in ADAS and Automated Driving

In the third phase the proposed modeling, specification and programming concepts are demonstrated by the example of a prototypical and partially realized ADAS. This finally demonstrates how the concepts and approaches developed in phase 1 and phase 2 can be used in industrially relevant languages and test environments. We are focusing on the first two design phases (Functional Levels A and B) and apply the proposed paradigms on the ADAS, which is a simplified Emergency Stopping System (ESS) (see Figure 0.4).

First a SysML model of the functional architecture of the ADAS case study as described in phase 2 has been modeled using the freely available Papyrus tool. For each component of the ADAS a corresponding block has been modeled together with their decomposition and interconnection. To model the contracts linked to them the SysML Requirement concept has been used. Using a Requirement Diagram each block is linked by means of a Satisfy Dependency to the Requirement representing the contract that the behavior of the block shall satisfy. The text attribute of a Requirement has been used to capture the textual specification of the assumption and the guarantee of a contract (see Figure 0.4). Corresponding to the Functional Design on Level B, a decomposition of the different blocks has been modeled in the SysML model.

After setting up the SysML model three different experiments where accomplished in the scope of phase 3. While the first experiment implements timed events at the Functional Level A, the next two are positioned at Functional Level B integrating Converter Channels and finally the co-simulation with a driving simulator.

Experiment 1 Within this experiment we chose SystemC as a simulation base of particular industrial relevance and describe how simulation based virtual integration testing (VIT) can be accomplished by use of SystemC and the proposed paradigms. The SysML case study of an ADAS was put into

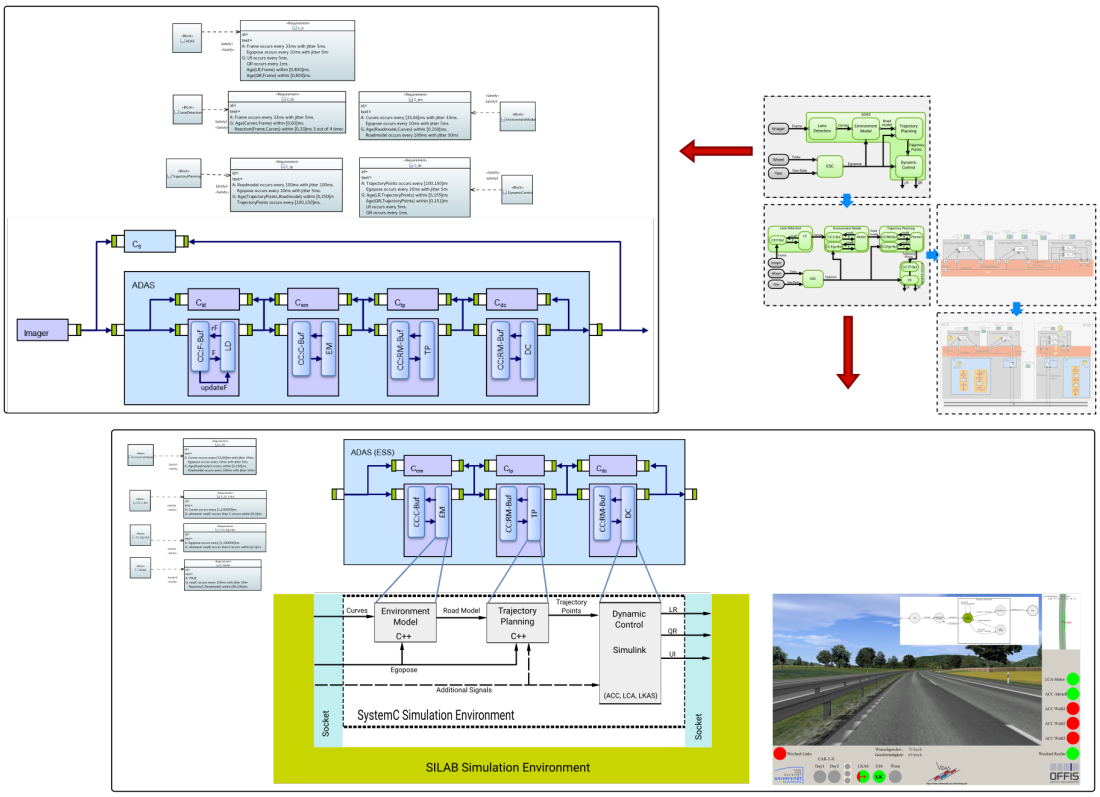


Figure 0.4.: Overview of the Demonstrator

C++ code in a somewhat tedious manual effort. The code is not restricted to a dedicated operating system. The first experiment implements timed events at the Functional Level A. It must be stated, that for sure it has to be the focus of future work to develop tools that automate the synthesis of VIT simulation models from specified contracts directly. The experiment shows manually derived event-generator-classes together with manually derived monitors that together form a VIT environment. The experiment also shows how issues in the specification are found.

Experiment 2 This is the continuation of the first experiment. By proceeding from Level A to Level B further details are refined into the simulation based VIT. Converter channels with channel semantics are instantiated in the model. The experiment demonstrates how the refinement process adds further local constraints that in the end lead to the violation of the overall specification. The simulation of manually derived event-generator-classes together with monitors again discloses these issues. Additionally, the experiment shows how functionality can be added to the timing model. So far, the VIT is about timing requirements only, without aspects of functionality. The experiment explains how functionality is added by a model of computation. For this example we assumed the approach of defining functionality with Mathwork's Matlab/Simulink tool suite that is able to automatically generate production C/C++ (embedded) source code from its executable models.

Experiment 3 In the third experiment the functionality of the ESS was added into the SystemC model and executed in a co-simulation environment together with a driving simulator (shown in Figure 0.4

at the bottom). The ESS monitors the health status of the driver and takes over control of the ego vehicle in case of an emergency. After the ego vehicles driving function has been adopted the ESS tries to reach a safe state to call the ambulance and to minimize the risk for other traffic participants. The safe state is defined as stopping the car on the road shoulder. The driving simulation software provides the ego-vehicle, the environment model and the sensors. The functional model of the ADAS is executed in the SystemC simulation environment and the necessary data exchange between the two simulation platforms is realized through a network socket connection. This experiment demonstrated the feasibility to integrate functional behavior into our timed simulation model. Furthermore, the feasibility to use the model in a complex co-simulation environment to conduct real driving experiments was shown.

Conclusion

With this report we have identified suitable design paradigms for the handling of time and timing effects in ADAS/AD development, explained how they should be employed in an integrated design framework and demonstrated their application in a realistic case study. Hence, we have shown that coherent treatment of time and timing effects in ADAS/AD design is indeed possible and can be integrated in typical industrial processes.

Scope and Structure of this Document

The MULTIC project identified design and programming paradigms that allow the coherent handling of (real) time properties in ADAS/AD design. In particular, these allow the transition between layers, as marked by red arrows in Figure 0.1, to be made accessible to a continuous development and analysis process. The process is supported by adequate combinations of specification, modelling and programming approaches, as well as suitable analysis mechanisms.

MULTIC addressed these topics in three main phases as depicted in Figure 0.5.

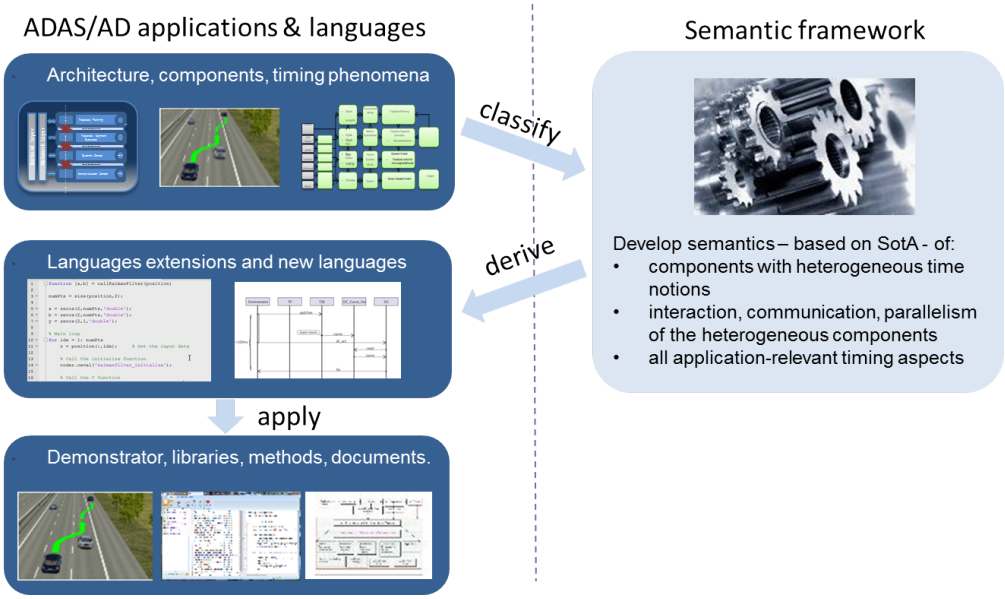


Figure 0.5.: MULTIC Approach

This document is organized in three different parts. Each part represents one of the MULTIC phases:

Part I represents Phase 1 *"Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving"*:

This phase starts with the analysis and classification of ADAS/AD architectures, components and timing phenomena. Based on this initial analysis, a semantic framework for the representation of ADAS/AS components with the ability to express interaction, communication and parallelism plus time notations is proposed. The timing annotation are capable to capture all of the identified timing phenomena. Furthermore, the proposed systematical framework will be accompanied with the development of a holistic design paradigm (including layer transitions) for the continuous treatment of time in ADAS/AD. This includes the aspects of specification, modeling, programming and target platform mapping on the individual levels as well as between layers of the layered architecture from Figure 0.1.

Part II represents Phase 2 *"Design Approach for Multi-Layer Time Coherency in ADAS and Automated Driving"*:

This phase applies the semantic framework and design paradigm from phase 1 on a running ADAS/AD example. This phase concludes with a derivation of research and development needs from the design paradigm for the integration, use and further development of existing specification, modeling and programming languages with a view to a continuous time treatment in ADAS ("conservative" extensions of existing languages). An evolutionary approach is being explored to arrive at an end-to-end time treatment without breaking existing design procedures/languages and using existing SW components in ADAS/AD (legacy code). As a result of an analysis of the limits and possibilities of conservative extensions, additional requirements for new languages ("non-conservative extensions") are derived from the development paradigm.

Part III represents Phase 3 *"Demonstrator for Multi-Layer Time Coherency in ADAS and Automated Driving"*:

This phase performs the prototypical construction of the running ADAS/AD demonstrator (emergency stop assistant), which illustrates an exemplary implementation of a continuous cross-layer treatment of selected (real) time properties. The ad-hoc enhancements necessary for the implementation in today's languages (e.g., through additional library elements for C/C ++, Matlab Simulink and Stateflow) are being implemented to show and evaluate the improvements in time management in today's ADAS design processes.

It is highly recommended to read the document sequentially. Each of the parts has an introductory section and references to previous parts to link the theoretical and practical contributions of this work.

Part I.

Design Paradigms for Multi-Layer Time Coherency

1. Introduction

The complexity in terms of interacting components in current vehicles is constantly increasing. Advanced Driver Assistance Systems (ADAS) comprise modular interacting software components that typically build upon a layered architecture. Figure 1.1 shows an example of such an architecture, annotated with the different reaction times characteristic to each layer.

As these components are typically developed by different teams, using different tools for different functional purposes and building upon different models of computation, an integration of all components guaranteeing the satisfaction of all requirements and a coherent handling of timing properties is a major challenge within the process of developing such systems.

For example, when connecting a trajectory planning component with a segment execution component, we have to ensure that the planned duration for the trajectory can be realized by the execution of the individual segments. On a technical level, further synchronization challenges need to be addressed. For instance, the frequency with which new segments are provided or polled have to match. In order to address those challenges, a common approach in the literature is to embed the different components into a common semantic framework, thereby enabling analysis as well as co-simulation of the components using the same notion of time. For sensor-actuator control, for example, it is possible to specify as well as verify real-time properties of Matlab/Simulink models on a specific platform using AUTOSAR. However, such an approach is not directly transferable to other layers, as the methods are specific to the time phenomena on these control layers.

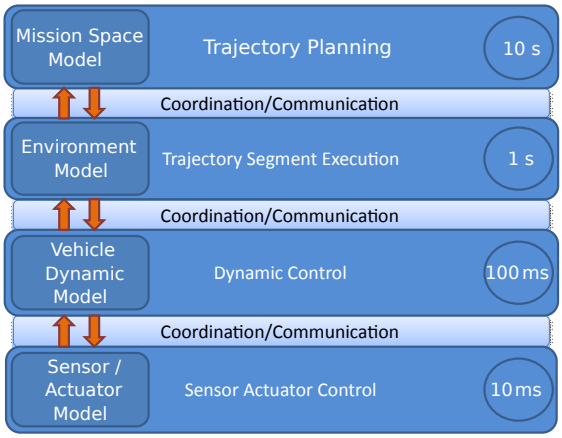


Figure 1.1.: Generic Functional Architecture for Automotive ADAS/AD

Therefore, in order to guarantee time properties on higher abstraction layers, similar interaction mechanisms need to be developed. In particular, specifications of the communication interfaces (including statements about the synchronization mechanism over these interfaces) are required. To this end, the first part of this document is organized as follows:

Chapter 2 provides an overview of the design paradigms presented in the report and their role in the development process of future ADAS/AD. To illustrate the different aspects which need to be developed for a coherent treatment of timing aspects across different abstraction layers, it also

depicts a simple example featuring a generic functional architecture of the considered application domain. Throughout the course of this partial report, parts of this generic architecture will be further instantiated and refined to exemplify particular concepts and ideas.

To arrive at a minimum set of extensions needed to achieve the desired coherent treatment, we first need an overview of the relevant timing phenomena and effects (see Chapter 3), where we reveal commonalities and discuss potential solutions in terms of suitable timing specifications. To this end, a simple semantic model is introduced based on the concept of components - in anticipation of Chapter 4, which enables expressing timing specifications in a common way.

To formalize this model further, an Architecture Modeling Framework (AMF) for Component-based Design is introduced in Chapter 4. This also includes contracts and Contract-based Design as a means to reason about timing properties. Using the example from Chapter 2, the key design steps in Contract-based Design are visualized. As a first result, the definition of the proposed timing contracts (i.e., how to specify required timing properties of the system such as those defined in Chapter 3) is presented here.

While Chapter 4 provides the semantic framework for system development, the instantiation of the particular components and their interfaces asks for suitable modeling and programming languages, or models of computation, allowing engineers to express the system's functionality in an efficient and effective way. For this purpose, Chapter 5 introduces the concept of Models of Computation (MoC) and their relationship to different relevant time models. Furthermore, relevant MoCs for the development of ADAS/AD in the functional and technical perspective are identified and introduced based on the example in Chapter 2. For the integration of different MoCs and different time models a concept for MoC interaction is introduced.

Chapter 6 finally characterizes a design paradigm where systems are composed of models from a heterogeneous set of MoCs. This chapter aims at giving an overview on how these design paradigms could be instantiated and applied in the context of ADAS/AD design.

While this first part focuses on the general framework of how timing properties can be specified and how necessary information about interacting components can be integrated, these concepts need to be illustrated by applying them on existing specification, modeling, and programming languages. To this end, Chapter 7 presents an outlook on Part II, where necessary steps are highlighted.

2. Quick Tour

The application of digital control in the automotive domain clearly follows an evolution with increasing complexity of both covered functions and their interactions. Despite, or because of, the complexity it is key to be able to reason about the behavior of such systems all along the design process. Defects that are detected late in the design induce additional costs in terms of development time and money.

System behavior typically subsumes different aspects, such as the functional aspect, timing, safety, and others. This report focusses on the timing aspect, and a main objective is to elaborate on approaches that allow for capturing all relevant timing phenomena and effects for such systems in a consistent and coherent way across all system layers and functional domains, as well as ensuring traceability along the design process. Particularly the heterogeneity of Models of Computation involved in ADAS/AD design and their interaction plays a key role in this effort. Moreover, the nature and complexity of information sources and their processing in such systems introduces complex data paths that are hard to track down and has large potential to cause consistency issues.

The present report proposes four design paradigms to support the development of future ADAS/AD, which are shown in Figure 2.1, and discusses their interplay in continuous design flows.

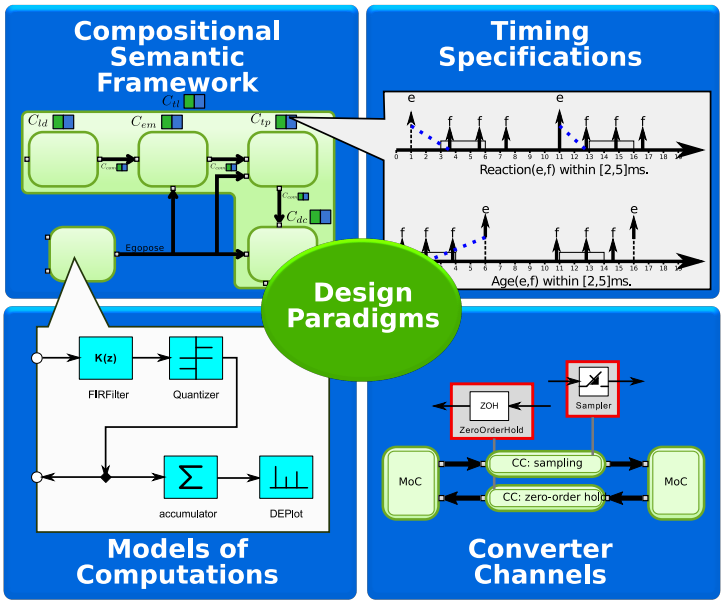


Figure 2.1.: Design Paradigms

2.1. Compositional Semantic Framework

The core design paradigm is a common semantic framework based on the "Component-based Design" paradigm. It consists of a generic component model, which provides all relevant concepts to capture

structural design models. The model defines component interfaces in terms of ports, on which the behavior of the component becomes visible, and over which components can interact. Component interaction is explicated by connectors, which connect individual component ports. The component model also defines a concept of composition, allowing to decompose components into (interacting) sub-components (e.g., component "Trajectory Planning" at the top-left part of Figure 2.2). Decomposition reflects an incremental refinement during the design process.

The framework also consists of a formal notion of behavior. It is based on the definition of data types for the component ports, enabling to specify which values can occur at a port. Together with a common notion of time, this provides for trace based semantics expressive enough to define all relevant kinds of behavior visible at the component ports.

The common notion of behavior forms the basis of the third ingredient of the framework, namely the concept of contracts, which allows expressing requirements and specifications for components. Contracts exploit assume-guarantee style specifications, where assumptions express the context in which the component is expected to work. If and only if these expectations are fulfilled by the environment of the component, it must behave as specified by the guarantee-part of the specification.

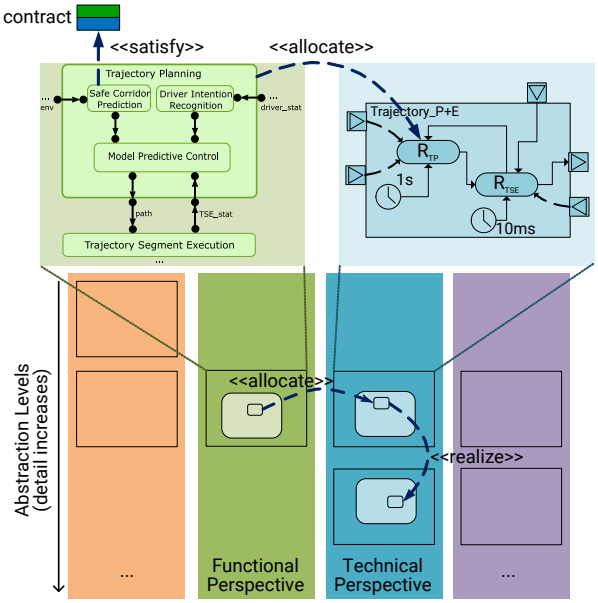


Figure 2.2.: Semantic Framework Overview

It is important to stress that the component model is typically not directly used in a design process. Being a *conceptual* model, it can (and will) be instantiated in many different forms, such as SysML, EAST-ADL and AUTOSAR, depending on the design phase and needs. However, every design model that is cast into the generic component model inherits its semantic foundation allowing to specify component behavior in terms of contracts. Since the Contract-based Design approach defines formal proof obligations for all relevant design steps, such as for the decomposition of design artifacts, the semantic framework allows to derive suitable V&V activities that safeguard the individual design process.

The framework is complemented by an organizational structure, where the component models are embedded into a two-dimensional matrix. The matrix provides a frame for structuring the design process along organizational and regulatory constraints, such as along supply chains and internal

organization boundaries, and along standard procedures as defined for example in the ISO26262. The semantic framework provides formally well defined concepts allowing to reason about consistency between the individual models in the structure (indicated by the `allocation` and `realize` links in Figure 2.2), enabling establishing continuous traceability of requirements along the design process.

2.2. Timing Specifications

Timing Specifications are put on top of the semantic framework. As an instantiation of Contract-based Design, the proposed specifications inherit all desirable properties such as well-defined composition operations. As the Timing Specifications are based on the common semantic framework they also share the paradigm of a single global (non-relativistic) time domain. The reason is simple. In complex system designs, particularly where information sharing with the environment is key, different parts of the overall system rely on local clocks which typically deviate from "real" time. A global reference clock allows to specify and to reason about such deviations. The Timing Specifications contain a concept for specifying local time bases in order to simplify expressing timing specifications for such cases.

The approach for Timing Specifications is to exploit existing well-established specification languages where possible, such as AUTOSAR Timing Extension, although the presentation uses a natural language based syntax for the sake of readability. Extensions to existing specification approaches focus on those aspects that are needed to capture expected timing phenomena in the considered application domain. This subsumes probabilistic timing phenomena that depend on data and operation modes, and (virtual) time stamping. Almost all proposed extensions are generalizations of existing specification patterns. We show how those extensions contribute to enable specification of relevant timing phenomena, particularly for expressing complex interaction between individual (sub-) systems.

2.3. Models of Computation

While the semantic framework provides for structural design models, the internal behavior of components is typically modeled in terms of languages that are based on different Models of Computation. The third design paradigm concerns the integration of component implementations into the semantic framework. Two aspects are particularly important. First, different models of computation may rely on different notions of time. The paradigm proposes an approach for establishing a common time domain for such heterogeneous Models of Computations. Second, the paradigm stresses on establishing an agreement on the component interfaces and the interfaces visible in the component implementation. With respect to such agreement, the Contract-based Design approach smoothly integrates the paradigms by defining if an implementation conforms to its specification, closing the remaining gap in the design flow.

2.4. Converter Channels

A major obstacle in the design of ADAS/AD is the interaction and interplay of the different control layers as well as between different sub-systems and systems in collaborative environments (cf. Figure 1.1). The report proposes a design paradigm, where such complex interaction is explicated by particular interaction components. Such components realize (complex) interaction protocols, and will typically be constructed from specialized design templates in order to ease the design process. These templates come with suitable sets of parametrised specifications to relieve the engineers from manual specification.

As a particular instance of such interaction components, we introduce the concept of Converter Channels. They comply to well-established patterns for the interaction of components with different interface semantics, which is often induced from the underlying model of computation. Other well-known design patterns are over- and under-sampling situations.

The report also exemplifies the design flow for the instantiation of interaction components and Converter Channels on the technical architecture level, focussing on systems modeled with AUTOSAR. It also shows - though not exhaustively - how traceability can be established between the specifications of early design models and the technical architecture, i.e., whether the technical architecture satisfies those early specifications.

2.5. Running Example

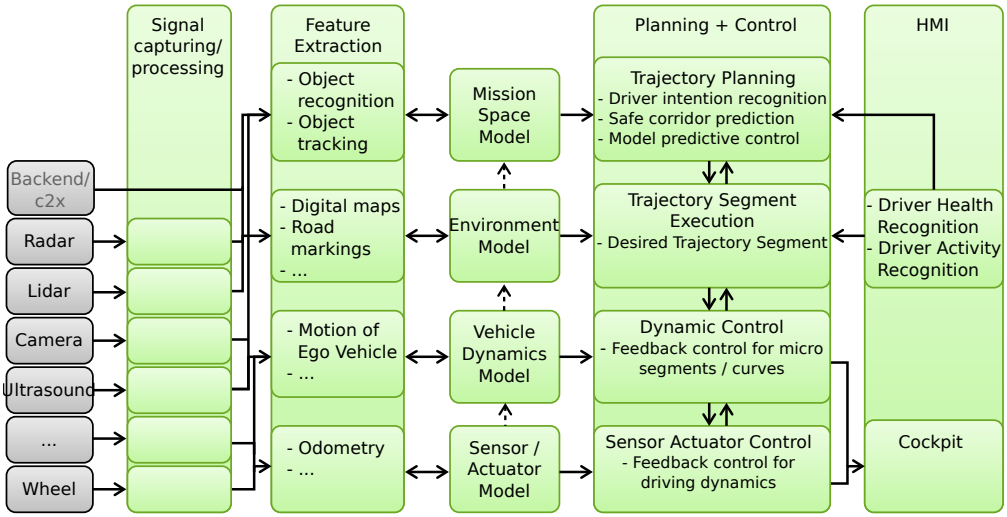


Figure 2.3.: Generic Functional Architecture for Automotive ADAS/AD

The report exploits (where possible) a running example to illustrate the relevant timing phenomena occurring within ADAS and to demonstrate the developed concepts for coherent treatment of time aspects across different abstraction layers. For this purpose we have selected a simplified emergency stopping system based on the description in [53]. The system can take over control of a vehicle in case the driver becomes incapacitated and execute a minimum risk maneuver to prevent accidents. Depending on the current situation the vehicle will either be stopped in the lane or steered safely to the hard shoulder. Since the sole purpose of the example is to illustrate the approach and not to develop a fully functional assistant system, we will use a simplified version of such an ADAS and adopt several idealized assumptions in order to focus on the aspects most relevant for this study:

- We focus on the driving function, driver monitoring and transfer of control is out of scope.
- We assume optimal environment conditions (e.g., no rain, fog, other limitations of perception)
- We assume good lane marking, continuous hard shoulder, no construction sites, etc.
- Scenarios subsume only light traffic density, speed limit 130 km/h.
- All traffic participants are “well behaved” (i.e., everyone follows regulations, no speeding, etc.)

The main development of the example into a fully functional demonstrator is subject of Part III. For now we focus on the functional architecture shown in Figure 2.3. Throughout the course of this document, parts of this architecture will be further instantiated and refined to exemplify particular concepts and ideas.

3. Coherent Timing Specifications

This chapter aims at collecting and characterizing timing phenomena and effects that are relevant in the considered domain of ADAS/AD design. It contributes to the goals of this report by revealing commonalities among these phenomena, and by discussing existing approaches for timing specifications as well as required extensions that enable a continuous and coherent treatment of time along the design of such systems. To this end, the following section provides - in anticipation of Chapter 4 - some basic semantic concepts that lay common ground for expressing time phenomena and effects. Although these concepts will be detailed in subsequent parts of this document, some are needed in order to talk about timing phenomena in a concise way, which follows in Section 3.2. Section 3.3 summarizes the chapter.

3.1. Semantic Design Dimensions

System design is a highly complex process where the system is decomposed and refined from early functional designs towards the final realization, resulting in a multitude of interacting functional components, which are represented at different levels of abstraction depending on the actual design phases. The V&V activities typically follow the design processes, checking whether timing and other properties are satisfied by the system in order to reason about its correctness. Particular V&V activities are performed at the different aggregation levels, starting from tests of single parts, whole components and sub systems towards final integration tests. V&V activities are also highly driven by separation of concern. Component testing often focuses on the functional aspect, with certain assumptions on the timing characteristics of, e.g., sensor inputs. The actual timing is often validated in later integration phases.

Such processes however require a coherent concept for characterizing the relevant aspects among all parts and participants of the system design. Incoherent or inconsistent interpretations will inevitably result in misinterpretations when the involved components are integrated[1]. This particularly includes the interpretation of time.

In the effort of providing a coherent concept of time for the considered application domain, we are facing a conglomerate of components, models and languages with different - also semantic - properties. Figure 3.1 provides an overview of relevant semantic dimensions along which the considered application domain can be characterized. In the automotive domain, almost all systems are open, as cars typically interact with their environment. This results in the fact that reasoning about system behavior is not possible without taking the environment into account - with all its consequences such as uncertainties of perception, incomplete knowledge about the environment, and so on. From the timing aspect, we are facing increasingly complex sensor processing chains in order to combine the perceived world into a consistent environment model. Maintaining consistency requires, among others, to reason about the age of data. Camera and radar sensor inputs for example are processed along different processing chains. The processed data cannot be combined correctly without information about the time point at which the data have been captured.

[1] The Mars Climate Orbiter Disaster is a good example (https://en.wikipedia.org/wiki/Mars_Climate_Orbiter) - resulting from an inconsistent interpretation of distances.

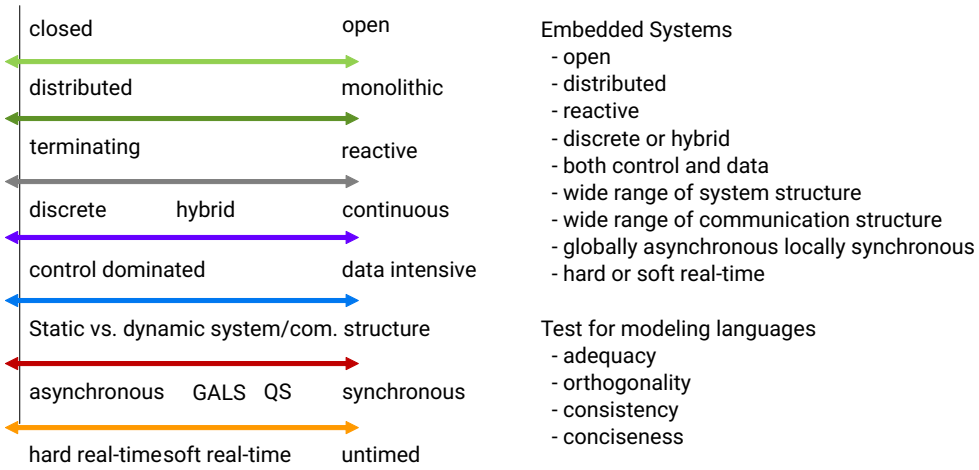


Figure 3.1.: System Design Dimensions

Automotive systems are also inherently distributed systems, which are highly influenced by communication between the individual sub systems. This not only introduces additional timing phenomena in terms of latencies but also additional issues such as bandwidth limitation of the underlying communication infrastructure. Also timing phenomena have to be considered which arise due to the complex nature of communication infrastructure such as the OSEK and AUTOSAR communication stacks as well as complex bus topologies imposing multi-hop communication.

While control applications in the automotive domain are dominated by reactive systems today, which is the standard paradigm for - continuous, discrete as well as hybrid - control, and typically modeled using "standard" tools like Matlab/Simulink, we will also see functions with terminating (aka. run-to-completion) semantics in the future. This change goes along the introduction of increasingly data intensive function, such as image processing on the sensor side. At higher control levels one will find functions for updates of (complex) environment models and planning tasks, for which reactive semantics not always fit well.

The increasing diversity of functions from different domains, as well as the distributed nature of the underlying architectures, comes along with the need to leave the synchronous world towards asynchronous paradigms that allow to specify systems in a more loosely coupled way. This indeed requires suitable concepts for modeling interaction between individual sub-systems and components - as well as timing specifications that are able to capture emerging timing phenomena.

ADAS/AD will interact with their environment also via communication in order to perform tasks collaboratively. Projects like simTD[2] and standardization activities for car-2-x communication (such as IEEE801.11p and the European C-ITS platform[3]) are witnesses of this trend. From the timing perspective, this requires a shift from strictly hard real-time systems towards soft real-time or even probabilistic control. For example, relying on the communication of positions and intentions of other cars in the neighborhood to safeguard the trajectory along a road crossing becomes questionable if the communication media are not reliable. Timing specifications for such systems must be able to cover this aspect.

Another aspect that is relevant in the context of future ADAS/AD design are dynamic computation and communication structures. This subsumes at least two different system scales. From the

[2] http://www.simtd.de
[3] http://ec.europa.eu/transport/themes/its/c-its_en.htm

perspective of car-2-x communication, the individual cars can be considered as agents among others in a dynamically changing system of systems. At the system design level, this may become visible by the need to differentiate communication with individual peers as well as different load situations with variable latencies.

Furthermore, dynamic system structures are highly important for fail-operational systems, where functionality of one component can be took over by another (probably degraded) one in case of failures. Some concepts will be considered in this report, such as the possibility to specify operation degradation and design-time task migration. Concepts with uncharted safety effects, like online migration of tasks, are not covered.

Not covered in this chapter is the fact that systems design in the considered domain typically exploit heterogeneous design models with potentially different interpretations of time. This aspect will be discussed in Chapter 5.

One To Rule Them All ... Considering timing phenomena in a way allowing covering such complex design dimensions calls for a common semantic domain that is as generic as needed to capture all relevant types of functions as well as interaction semantics.

Following the *Component-based Design* approach as a suitably generic framework, we assume systems to be build from *components* with *interfaces* as depicted in Figure 3.2. Components may represent software functions, hardware elements or any other part of a system, depending on the design context. Components may be implemented with arbitrary languages and arbitrary underlying Models of Computations. Components interact with their environment (such as other components) via interfaces. A component interface consists of a set of typed *ports*, where the type of a port defines which values can be observed at the port. Ports represent certain entities in the underlying (implementation) model of the component. For Matlab/Simulink models, for example, ports may represent input and output signals.

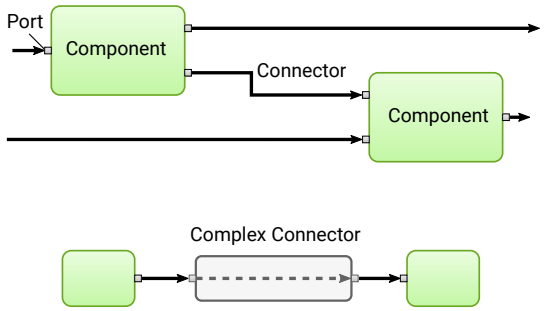


Figure 3.2.: General Component Model

Components interact via *connectors* between ports. We distinguish two types of connectors. A *simple* connector does not take time for transporting values between the connected ports. A *complex* connector, on the other hand, represents a physical transmission medium and imposes latencies as well as other complex behavior. Complex connectors are in fact also components as indicated at the bottom part of Figure 3.2. If not stated otherwise in the remainder of this chapter, connectors are complex. Connectors are always directed such that data flows only in one direction. Service interfaces are modeled by sets of (combined) ports, which together realize the involved protocols.

Key for establishing a coherent semantic domain is also a common notion of behavior and interaction of components. Such notion consists of two parts. A type system for port values creates a common understanding of the values that can be observed at the ports. The other part is a notion of time,

which defines how values on ports can evolve. System design in the considered domain will typically consist of heterogeneous models with different notions of time. Coherent design however calls for a common notion of time on which the individual entities can agree. Hence we assume a common notion of time among all component interfaces. How different time domains of individual (implementation) models can be cast into such common time domain is subject of Chapter 5.

A dense metric time domain based on the set of real numbers $\mathbb{R}^{\geq 0}$ is a natural choice, and allows for various commonly used interaction specifications as depicted in Figure 3.3. An important question is whether such framework shall exploit a globally defined time or individual local clocks. Local clocks have some advantages such as simple modeling of clock drifts. On the other hand, specification of the interaction between different components can easily become unintuitive and clumsy. More important, dealing with local clocks is counter-productive with respect to establishing coherent timing specifications. Neglecting relativistic effects, a global time domain allows for explicating (local) *believes* about time. Suppose a component that maintains a model about surrounding traffic participants. It relies on information about the positions of other vehicles, and the time instants of these data. Global time not only allows to reason precisely about the age of position data, but also to specify deviation of a (local) clock representing the believe of the component about the actual age. Such representation is however out of scope of this report and will be covered elsewhere.

A definition of time that is sufficient for our purpose is provided in [17]. The authors exploit so-called *superdense* time, which is represented by the set $\mathbb{T} = \mathbb{R}^{\geq 0} \times \mathbb{N}$. The first component $\mathbb{R}^{\geq 0}$ represents the Newtonian time, and $\mathbb{N}$ denotes so-called *microsteps*. The introduction of "two dimensional" time has strictly formal reasons in order to avoid issues with compositional reasoning, and has little, if any, practical relevance. All timing specifications discussed in this report employ the time domain $\mathbb{R}^{\geq 0}$, and we assume implicit casting into $\mathbb{T}$ whenever needed.

A *signal* is a function $s_x : \mathbb{T} \to \mathbb{V} \cup \{\epsilon\}$, where $\mathbb{V}$ is the value domain (type) of some port x and ϵ represents the absence of a value. We denote $S_x = \{s_x : \mathbb{T} \to \mathbb{V} \cup \{\epsilon\}\}$ the set of signals over port x. For a set X of ports, we also define s_X and S_X as tuples of signals and sets of signal tuples, respectively. The definition allows for many different kind of system behavior such as *continuous time* and *discrete event* semantics, as depicted in Figure 3.3. A *discrete-event* signal has non-absent values only for $\tau \in D \subset \mathbb{T}$, where D is some discrete set (i.e., is order isomorphic to the natural numbers). If, to the contrary, the signal is non-absent for all time points ($s(\tau) \neq \epsilon$ for all $\tau \in \mathbb{T}$) then we distinguish two further cases. *Discrete-time* (or more precisely discrete evolution) signals change their value only at time points as for discrete-event signals, i.e., for $\tau \in D \subset \mathbb{T}$, where D is a discrete set. Otherwise, finally, it is a *continuous-time* (continuous evolution) signal.

Figure 3.3.: Signal Types

These few definitions allow us to cover a large range of system behavior. The behavior of some component M with set X of input ports and Y of output ports can be characterized as a set $M \subseteq S_X \times S_Y$, where S_X is the set of signals over X, and S_Y is the set of signals over Y, respectively. Moreover, we can formally state specifications over M of the form $P \subseteq S_X \times S_Y$. We can also state verification tasks in terms of proof obligations such as whether M complies to P. This will be further elaborated in Section 4.2.

3.2. Time Phenomena

Based on the definitions from above, we can refine the example from Figure 2.3. Boxes in the figure are interpreted as components, and lines as connectors. While the lines in the figure indicate particular flows of information, they would probably be represented by two distinct connectors in cases where the communication between components involves a protocol (such as between the model components and the respective planning functions). Not shown in the figure are ports, and we want to assume that all end points of connectors are ports.

This refinement already allows us to reason about timing by expressing specifications of the behavior of the components at their component ports. behavior however can in general be arbitrarily complex, and so can specifications. On the other hand, there are well-established frameworks for timing specifications such as AUTOSAR Timing Extension and those specified in the TIMMO and TIMMO-2-USE projects [6, 66, 67]. In these frameworks, timing behavior is specified via instantiating pre-defined parametrized constraint templates. Each instantiated constraint template defines a timed language over some set of observable events referenced by the constraint instance. An observable event is understood as an identifiable state change. For example, assigning a value to a variable is an event, sending data at an output port of a component is an event and activation of an operating system task is an event. Consider again the different types of signals illustrated in Figure 3.3. The interpretation of events for a discrete-event signal is obvious. However, for a discrete-or continuous-time signal it is less obvious what an event is supposed to represent. Hardware description languages and corresponding simulators define events for discrete-time signals that denote a value change on such signals. On the other hand in [6, 66, 67] an event for such signals is considered to represent an update of the value of the signal. This is because usually timing analysis regarding schedulability and performance often abstract from the actual values of signals. We adopt the latter semantics, unless explicitly stated otherwise. For continuous-time signals we expect that it is made explicit in specifications how such a signal is sampled. This considerably simplifies the approach to reason about time in a coherent and consistent way. Please note that this does not prevent taking continuous-time behavioral models into account. We just insist that a reasoning about such models taking continuous evolution into account in combination with discrete control parts of the system would be too complex.

It is important to note that in our context events are solely observable at ports. In other contexts, this may not be necessary, and "state changes" may become visible by other entities. Hence, in order to observe for example an internal "state change" of a component, it must be "wired" to a port of the component.

The remainder of this chapter follows the line of specification patterns in that existing timing specifications are examined and put into the context of ADAS/AD development. Throughout this course, gaps will be identified that come from the specific needs of the considered domain, and extensions will be proposed to fill these gaps.

3.2.1. Event Specifications

As shown in Figure 3.2, components have ports and interact with other components via ports that are connected. The most basic concept needed to reasoning about time in this general component model is to characterize behavior observable at these ports. For instance, if a component provides data at an output port, it is of interest how frequently components consuming this data can expect new data at the output port. This knowledge is especially necessary in case the data is buffered on the receiver side, in order to reason about necessary buffer sizes if data shall not be lost. In [6, 67] different constraint types are provided to characterize such behavior in terms of *occurrences of events*.

Event occurrence constraints: The occurrences of events over time are specified by constraints defining a timed language for a single event. Usually, these constraints and languages refer to single event occurrences and characterize their distances in time. However, some constraint types also allow to not only refer to subsequent event occurrences, but also to a sequence of n occurrences of the same event. Thereby such constraints can also restrict the distance in time between the first and the last event occurrence of any sequence of n subsequent event occurrences.

PeriodicEventTriggering[6] is an example of such a constraint characterized by parameters *period*, *jitter* and a *minimum inter-arrival time*. *RepeatConstraint[67]* is an example of a constraint bounding the time interval for n subsequent occurrences of the same event. Figure 3.4 shows an excerpt of Figure 2.3 and contains a *PeriodicEventTriggering* constraint on the event named *distance*, which is specified to occur with a period of $100ms$. *Jitter* and *minimum inter-arrival time* are both 0. This is to specify that the function computing the motion vector of the car assume input data every $100ms$.

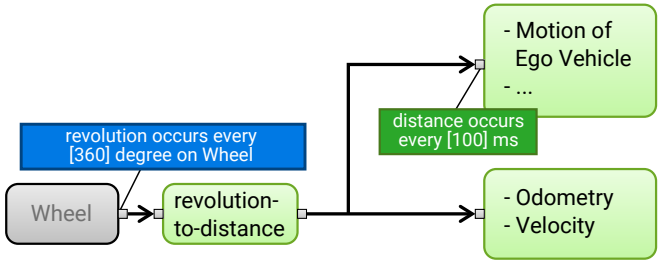


Figure 3.4.: Excerpt of the case study exemplifying specification of event occurrences

The behavior defined by these constraint types is typically time-invariant and may only change at well-defined points in time like mode changes. However, as pointed out in [67], there are use-cases for dynamically changing behavior of events. For example the occurrence of some events is better described depending on the evolution of a mechanical part like the rotation of a wheel. Like indicated in Figure 3.4, a wheel revolution sensor emits a signal on each complete rotation of a wheel. So a specification utilizing an angular "time base" expressed in terms of degrees is closer to the physical process and explicitly expresses the intention. For this purpose, in [67] means are proposed to specify alternative *dimensions* collecting a set of *units*, which in turn are used to express durations measured on a specified *time base*.

Obviously, for achieving a coherent treatment of time, it must always be possible to relate values expressed on different time bases. In [67], a pre-defined time base *Universal* with units *sec*, *ns*, *micros* and *ms* serves as a reference, and values expressed on different time bases must be ultimately convertible into this reference time base. In the example shown in Figure 3.4, the distance in time between occurrences of the event *revolution* are expressed in terms of the unit *degree* on the time base *Wheel*. In order to be able to interpret this specification it must be defined what is the equivalent value on the time base *Universal*. As shown below, the units *degree* and *rotation* are collected by a dimension *angle*. It is specified that values on the time base *Wheel* are expressed in terms of the units of dimension *angle*. The relation to the time base *Universal* depends on at least two parameters (this is simplified): The diameter $wheel\_dia\_in\_m$ of the wheels of the car and the current speed $speed\_in\_m\_per\_sec$ of the car. Assuming a diameter of $0.65m$, then at a speed of $1\frac{m}{s}$ the wheel rotates by $\frac{1\frac{m}{s}}{0.65m*\pi} * 360 \approx 180$ degrees, which results in the specification of the time base *Wheel* as shown below.

Dimension $angle\{degree : 1, rotation : 360\}$

TimeBase $Wheel : angle\{1\ rotation$ **on** $Wheel = wheel\_dia\_in\_m * \pi / speed\_in\_m\_per\_sec\ s\}$

While this style of using custom time bases (related to the *Universal* time base) is helpful when writing specifications, it poses a major problem on timing verification and validation activities. Parametrized specification such as the above can only be lifted to the *Universal* time base with respect to the values or value ranges of the free variables. While a variable like $wheel\_dia\_in\_m$ might be assigned a constant value at design time, obviously the current speed of the car is not constant. For verification purposes, it must always be possible to determine (at least) bounds for the values of the free variables, like $[0, 160]\frac{m}{s}$ for the variable $speed\_in\_m\_per\_sec$. Further, bounds on the derivative of variables could be considered in timing analyses, in order to better mirror physical behavior where the slope of the change of event rates is bounded. Note, however, that this kind of timing specifications considerably increases the verification effort.

3.2.2. Event Relations

Virtually every time phenomenon concerns the relation between events. With respect to time, one is not only interested in the ordering of such events but also in the time span that lays inbetween. This section discusses some well-established patterns for the definition of event relations.

Consider again the general component model sketched in Figure 3.2. Besides being able to reason about the timely behavior of single events, obviously it is also important to be able to reason about relation of events. For instance for a component the information about causal relations between input and output ports must be expressible in order to reason about chains of effects. This can be captured by specifications of orders on occurrences of different events. Further, timing constraints over sequences of such related events allow to reason about proper timing behavior of end-to-end effect chains. Some use cases also require proper synchronization of different effect chains with a certain tolerance on the timely occurrence of the eventual effect. For example a brake-by-wire system needs the brakes at the wheels to be actuated synchronously within a small time window.

A few words about causality: Some of the timing specification concepts in [6, 67] constrain the timing of a sequence of "causally related" event occurrences. This term deserves some explanations: If a source event e causes a target event f, this does not mean that e is a *necessary* cause of f. So the occurrence of f does *not necessarily* imply the occurrence of e before. Neither, a sufficient cause is required. Thus, the occurrence of e does *not necessarily* imply the occurrence of f.

This kind of interpretation of causality is useful for appropriately constraining the time delay for an effect chain that involves over- or under-sampling along the sequence of events. Assume that e denotes the assignment of a new value to a shared variable by a task executing a control algorithm, and f denotes reading a value from the same variable by an actuator control task. Now the rates at which e and f occur can differ, leading to over- or under-sampling of the shared variable by the actuator control task. Further, both tasks might have different sufficient causes of activation. Still e and f can be considered "causal" in a sense that the values read by the actuator control task are determined by the task executing the control algorithm. For instance, in the case of over-sampling, multiple occurrences of f can be seen as causal to the same occurrence of e. Questions about causality and influence of values of variables thus clearly involve the semantics of the underlying model of computation. The approach in [6, 67] merely assumes that causal relationships can be inferred from the underlying model when verifying/validating specifications. We will come back to this matter below.

Ordering constraints: A partial order on the occurrences of a set of events can be specified via constraints defining an untimed language over the event set. In [6] a special variant of such an ordering constraint is defined: *ExecutionOrderConstraint*. This constraint is tuned towards specification of a partial order on the execution of *RunnableEntities*, *BswModuleEntities* respectively. It also allows specifying orders of execution of a set of *Runnables* that rotate with each occurrence of a given event.

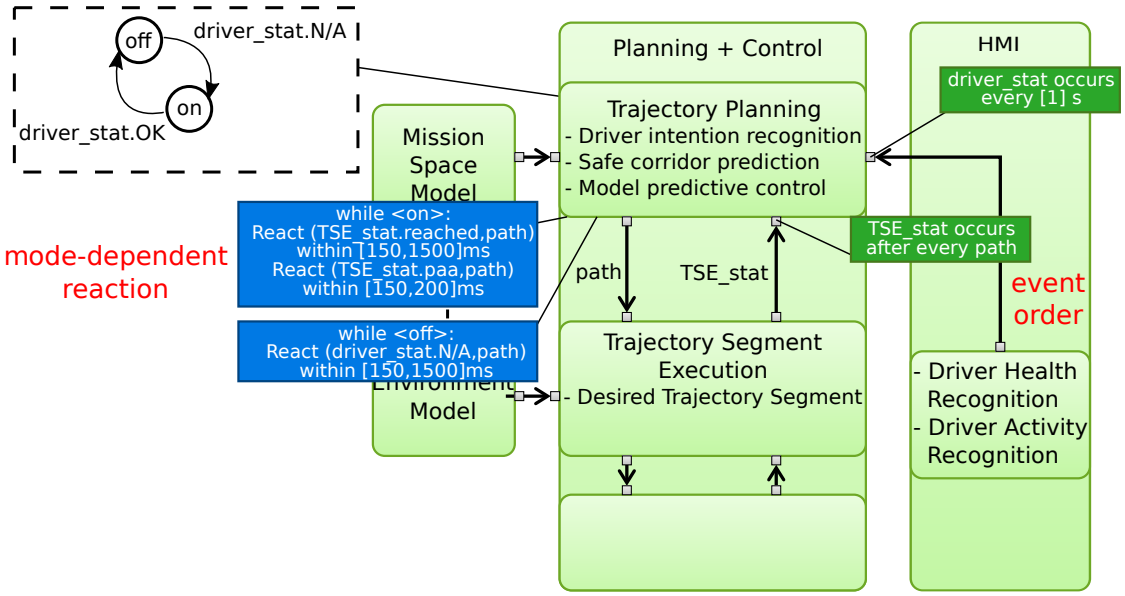


Figure 3.5.: Excerpt of the case study exemplifying specification of event relations

In Figure 3.5 an example is shown specifying an order on events. The specification states that an event *path* sent by the trajectory planning is always followed by an occurrence of event *TSE_stat* sent by the trajectory segment execution. Due to this specification it becomes clear that the trajectory planning module will only need to cope with events from the trajectory segment execution that it has caused on its own.

Delay constraints: Delay constraints define a timed language for occurrences of a pair of different events or a pair of disjoint sets of events. These kinds of constraints can be classified according to the causal relationship between the different events:

- Delay constraints can be specified for (sets of) events without requiring a causal relationship between them. It should be noted that just the existence of a target event occurrence within the constrained time interval is required. More target events can occur outside and inside the time interval without violating such a constraint. A stronger notion assigns increasing indices to event occurrences and requires a target event occurrence within the time interval with an index equal to the source event occurrence.
- Delay constraints can be specified for (sets of) events assuming a causal relationship between source and target event(s). These constraints can further be classified into *reaction* and *age* constraints. The former constrains the delay from a stimulus event occurrence to a response event occurrence. A stimulus event occurrence is seen as being causal to a response event occurrence. The first causally related occurrence of a response event must lie within a time interval relative to the occurrence of a stimulus event. The latter kind of constraints look at

response event occurrences and constrain the time interval to causally related stimulus event occurrences at some point in the history. The most recent causally related occurrence of a stimulus event must lie within the time interval.

- In [6] a special variant of an age delay constraint is defined, which is intended to be used when the age of data is important, but the sender is not known yet. In this case the age of data is constrained to lie within a specified bound.

In Figure 3.5 an example is shown specifying a reaction constraint on causally related events *TSE_stat* and *path*. This constraint makes use of two concepts extending the expressiveness of the catalog of constraints in [6, 67]: Event occurrence expressions [6] and mode dependencies of constraints [67]. The former is a concept allowing to precise an already defined event or to derive new events from already defined events. Usually, as an event is an identifiable state change, its representation needs to encode information about what state change is meant. The approach in [6, 67] is to make an event an explicitly modeled entity that is linked with the objects of an underlying system model. For example, if an event represents the arrival of new data at the input port of a particular component, then the event representation points to that component and the corresponding input port. However, that only expresses that the event occurs each time there is an assignment of an *arbitrary* value to the input port. While from a timing point of view this is sufficient for a large number of use-cases, it is sometimes important to refer to the actual value received at the referred input port. To this end, in [6] an expression language is proposed that semantically acts like a "filter" on an event. The event pointing at the input port of a component is defined to only occur if the received value fulfills a specified boolean expression. The very same expression language can also be used to construct *complex* events out of other event definitions. For example one can define an event that occurs whenever at least one of a specified set of events occur.

In [67] an extension is proposed that allows restricting the modes during which a given constraint must hold. The rationale is that some constraints are simply not valid for a given mode or the constraints are stricter in a given mode, less strict respectively.

In the example depicted in Figure 3.5, a reaction by means of an occurrence of event *path* within $[150, 1500]ms$ is required, if the event *TSE_stat* occurs, the value received on the referred input port equals *reached* and the state of the trajectory planning module is *on*. The idea here is to differentiate between following cases:

- The trajectory segment execution (TSE) signals that it has successfully reached the end-point of the path, which the trajectory planning module (TP) has sent before. In this case, the TP shall provide the next path to TSE within $[150, 1500]ms$.
- The TSE signals to TP that it has detected a critical situation (e.g., another car overtaking, while not maintaining a safety distance to the ego vehicle). In this case the TP module shall provide a new path to TSE within $[150, 200]ms$.
- Both of the reactions above are only required in case the TP module is in mode / state *on*. Initially, the mode is assumed to be *off*. When being in mode *off* and the value *N/A* is received on the input port of TP referenced by the event *driver_stat*, TP shall provide the first path of the maneuver within $[150, 1500]ms$. Further, in this case TP switches to state *on*.

Whenever TP receives the value *OK*, denoting that the driver is now being able to operate the vehicle again, it stops its operation returning to state *off* and signals to TSE to abort execution of any path segments it has sent to it before (not shown in Figure 3.5).

Consider again the specification of the order of occurrences of events *TSE_stat* and *path*. In conjunction with the mode- and value-dependent specification of parts of the behavior of TP, an

interaction protocol between TP and TSE is fixed. TSE only needs to be able to buffer one path. A new path is only requested from TP in the "initial" state, or if the previous path has been executed or if TSE detected a critical situation.

Execution Time Constraint: This type of constraint can be seen as a specialization of a delay constraint between causally related stimulus and response events, where both events belong to some software executable entity. The occurrence of a response event must lie within a specified time interval relative to the corresponding occurrence of the stimulus event, while not counting subintervals where the underlying executable entity was preempted. In [67], the definition of such a constraint requires to explicitly identify events that trigger the software executable entity, as well as events representing its termination, preemption and resuming the software executable entity if it has been preempted before. In contrast the approach taken in [6] seems to be more appropriate, where just the software executable entity is identified whose execution time shall be specified.

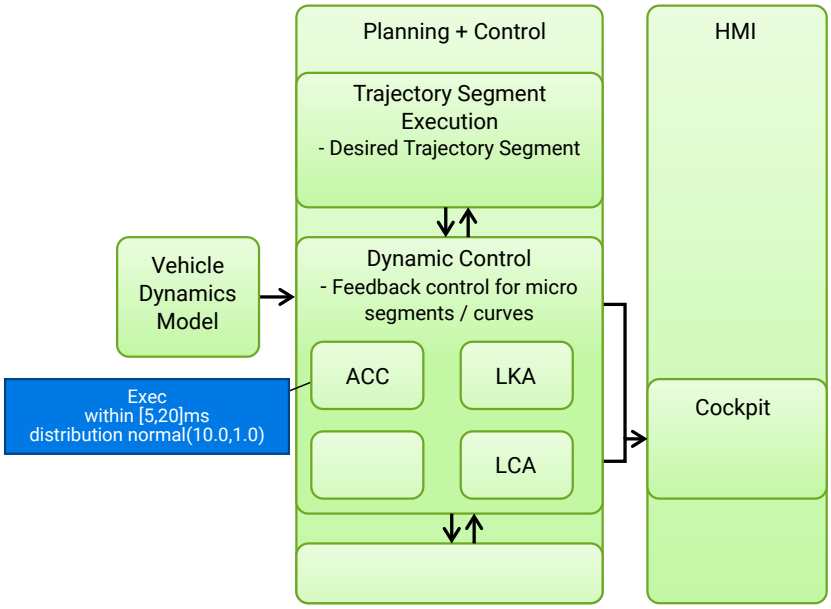


Figure 3.6.: Excerpt of the case study exemplifying specification of the execution time of a function

In Figure 3.6 an example is shown specifying the execution time of an Adaptive Cruise Control (ACC) function being part of the dynamic control component. This constraint makes use of a concept proposed in [67] allowing to express a probabilistic distribution of values in a given time interval. The rationale of such extension is that precise bounds for some timing behaviors are often very hard to determine. So distribution functions can be used instead, whose parameters are determined from, e.g., measurements. Corresponding analysis methods can then derive probabilistic information about, e.g., latencies of end-to-end effect chains. This in turn allows assessing whether occasional violations of deadlines can be tolerated. Usually, the normal distribution with location $\mu = 10.0$ and scale $\sigma = 1.0$ has an infinite support. However, the lower and upper bounds as shown in Figure 3.6 give rise to a truncated normal distribution, which is more appropriate if such a distribution is obtained from measurements.

Another mean to weaken the strictness of such constraints is provided by the concept of so called *weakly hard* timing constraints [15], which is also considered in [67]. The basic idea behind weakly

hard timing constraints is to allow occasional violations of a constraint, while precisely bounding the distribution of satisfied and violated instances of the constraint within any window w of time. Contrary to probabilistic approaches, weakly hard constraints allow arguing about for example a maximum number of constraint violations of out a number of say n instances of the constraint that can be tolerated. This style of weakening a constraint fits well with control engineering, if for instance a control algorithm is robust enough to compensate that 2 out of 20 control cycles are skipped. This prevents over-dimensioning the system, thereby reducing costs.

Synchronization constraints: These types of constraints define a timed language for occurrences of a set of different events. These kinds of constraints can be classified as follows:

- Synchronization constraints for a set of unrelated events. Such a constraint defines a sliding time window of length $tolerance$. The constraint requires that for each system behavior the referred events occur within such a sliding time window, meaning each referred event occurs at least once within a time window. The time windows may overlap and even "share" event occurrences. A stronger notion assigns increasing indices to occurrences of each event and requires event occurrences with the same index to lie within a time window of length $tolerance$.
- Synchronization constraints for a set of causally related events. These constraints can be further classified into *output* and *input* synchronization constraints. The former constrains that for a given occurrence of a stimulus event, the first causally related occurrences of the referred response events must lie within a time window of length $tolerance$. The latter kind of constraints look at response event occurrences and require that the most recent occurrences of causally related stimulus events must lie within a time window of length $tolerance$.

3.2.3. Data and Mode Dependencies

In contrast to simple control application, data flow as well as control flow in complex systems such as ADAS/AD is not static, but highly depends on the input data. For example, an ACC system acts depending on the existence and the distance of vehicles in front. Data dependencies may be reactive or persistent. The ACC control laws for example distinguish between *free flow* and *follow* modes.

The example in Figure 3.5 shows the use of mode dependent timing specifications. While modes have already been introduced (AUTOSAR for example has concepts for modes and mode management), they are not integrated well with the rest of the specification concepts. In [67], a mode is just an object with a name and it is assumed that some event activates a mode and another event deactivates a mode. The existing approaches leave open questions. For example:

- Assume an event e defined to activate mode A and event f defined to activate mode B. Neither e, nor f deactivate any mode. What happens if first e then f occur? Are two modes active at that point in time?
- What happens if a timing constraint requires that a causal I/O behavior needs to happen within a specified time interval $[lb, ub]$ when a mode A is active, but after the occurrence of a stimulating event the mode is switched to another mode during the time interval $[0, lb[$? Is the implementation still bound to show a reaction during $[lb, ub]$, or does the mode switch "cancel" all pending reactions?
- Usually, mode switches are to be implemented somehow by the underlying system. There exists an entity managing the modes of multiple components. Mode switches can be requested from that entity, which then distributes the mode switch by, e.g., switching the communication configuration, switching to another schedule table, deactivating certain functions in a mode.

This process takes time, which may depend on the mode being switched from, as well as the mode being switched to. How can one specify timing behavior for these mode switches?

Another closely related concept are data dependencies. Work like [79] (among other) developed initial ideas on integrating data dependencies into real-time analysis, but has not reached industrial practice.

To deal with mode- and data dependencies in timing constraints, we propose to add a notion of variables to timing specifications and conditions defined over a set of variables. Thereby it can be made explicit in specifications when signals are sampled, and especially when mode switches shall be evaluated and have an effect on the behavior.

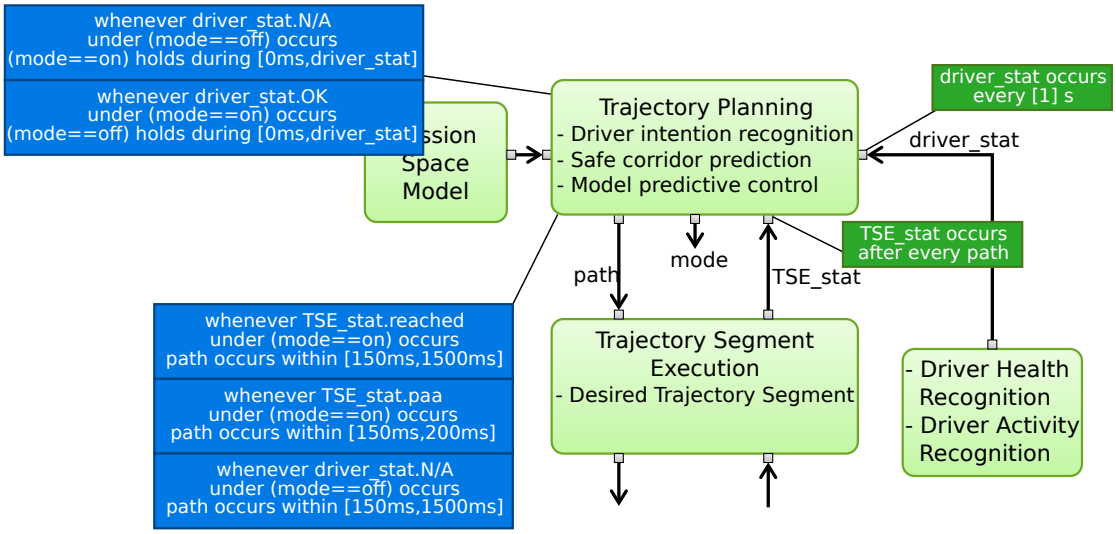


Figure 3.7.: Excerpt of the case study exemplifying mode- and data-dependent timing specifications

With this idea in mind, we reconsider the example shown in Figure 3.5. In Figure 3.7 an improved version is shown addressing the concerns mentioned above. The required reactions of the TP function are still dependent on a mode, which is either *on* or *off*. *mode* is considered to be a variable in the constraints. Further, additional constraints specify when this variable *mode* is updated. Following the idea in [6], the specifications do not just refer to events denoting a value change at some port (e.g., the input port on which signals from the TSE function are received), but to the values that are received. The constraints shown at the top-left of Figure 3.7 express that the value of the variable *mode* is updated according to the value received from the HMI module and remains stable until the next triggering event from the HMI module. Instead of just assigning a constraint to a mode, for which it shall be satisfied, the occurrence of an event *TSE_stat* is bound to a condition on the mode. Thus, it is precisely defined when the mode is being evaluated. Also the ambiguity regarding mode switches is resolved. When in mode *on* the event *TSE_stat.reached* or *TSE_stat.paa* occurs and the mode is switched to *off* before the corresponding occurrence of the event *path*, an implementation of TP is still required to produce a reaction by sending a path to TSE.

3.2.4. Causal Event Correlations

The need of data and mode depend timing specifications already shows that the functional and timing aspect become increasingly entangled when talking about specifications for complex systems. We already discussed notions of causality with respect to event correlations.

The question of integrating causality into timing specification is, at least from our point of view, a highly important though unresolved issue. Real-time reasoning is almost always based on causality assumptions, as mentioned earlier. That is, for example, we expect to talk about reactions in a way that assumes that a reaction must have some cause. Timing specifications typically express weak causal properties. For situations such as under- and over-sampling, this is reasonable and hence sufficient. For other cases we however need to express strong causal dependencies. This would be for example needed in order to reason about a buffer being filled with elements and a function being executed as many times as there are elements in the buffer. Consider the trajectory segment execution module. Assume that the module gets activated upon reception of a path sent by trajectory planning. For each received path it first computes a sequence of control points / curves to be handed over to the dynamic control, one by one. As along as it has control points left, it will keep executing. After the last control point has been sent, it will send a notification to trajectory planning and switches to an idle state waiting for the next path to be received. Obviously, timing behavior and schedulability depends on the number of control points a path is split into. The execution scheme of trajectory segment execution would be bursty, corresponding to the number of control points.

Some effort has been spent to add strong causality to specifications. In [72], we added particular attributes to specifications patterns to change the resulting timed language such that it mimics some aspects of the intended causal interpretation. In the context of the TIMMO projects, also different interpretations for effect chains have been suggested ([26]). Another approach developed in the project was to attach colors to events to identify causal dependencies.

We suggest an extension of timing specifications that generalizes the coloring approach in a way allowing to cover a large range of relevant use-cases. Some of them will be exemplified later in the document. The basic idea is simply to *pretend* that the observability in the system can be extended such that event correlations can be identified[4]. We leave the question open of how this kind of observability can be achieved. In some cases it would be possible to attach serial numbers to events. In other cases, the underlying model of computation could provide the required information.

We extend the specification of event occurrences by two functions, namely $\blacktriangleright(.,.)$ and $\blacktriangleleft(.,.)$, with following semantics: The function $\blacktriangleright(e,f)$ references the event occurrence(s) f that have causal dependencies on the event occurrence e. The function $\blacktriangleleft(f,e)$ references the event occurrence(s) e for which the event occurrence f has causal dependencies. Recall that events are attached to ports. Event occurrences are the individual instances of an event observed at the ports.

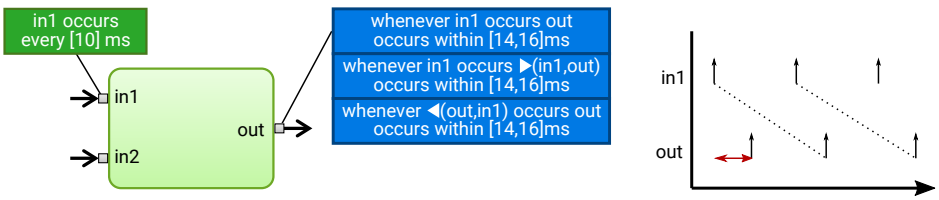


Figure 3.8.: Functional Timing Specification

An example of how these functions are used is depicted in Figure 3.8. Suppose that only the events $in1$ and $in2$, respectively, are observed at the input ports, and that only event out is observed at the output port. The three delay specifications at the output port are identical, except that for second and third one the explicitly named events are replaced by $\blacktriangleright(in1,out)$ and $\blacktriangleleft(out,in1)$, respectively. However, with the first specification we run into an interpretation problem, as it is not clear which output event correlates with an input event (which is the core reason for the consideration in [26]).

[4]Note that many real-time analysis approaches (implicitly) assume this kind of observability.

For the causal interpretation, this is perfectly clear, and both specifications are equivalent if exactly one output event occurs for every input event. A detailed discussion about causal event relations can be found in Appendix D.

Causal event correlations are useful in many contexts. An important application in ADAS/AD design is to enable specification of data consistency both, along the chain of complex sensor measurement and processing, and across different control layers. The following discusses some relevant aspects.

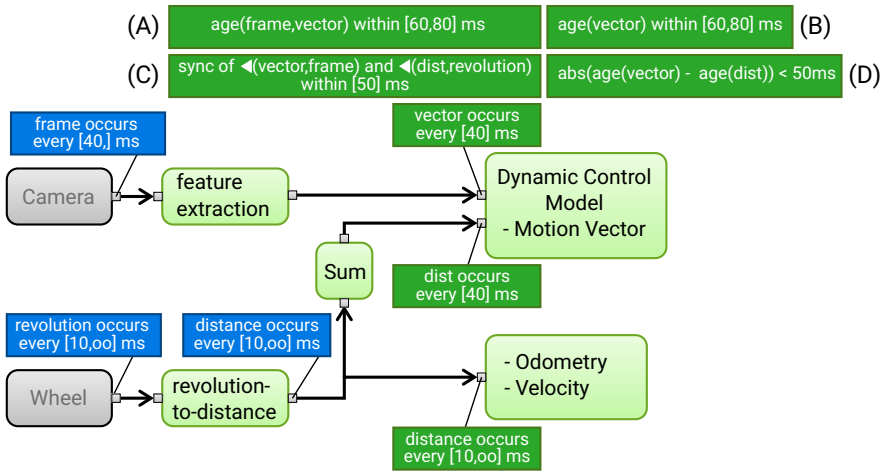


Figure 3.9.: Examples for the Application of Event Correlation

Refined Delay Constraints Figure 3.9 shows an excerpt of Figure 2.3. The input of a wheel revolution sensor is used to calculate distances and speed of the car for low level control. The data is also used to maintain the dynamic control model of the vehicle, together with motion vector data obtained from some feature extraction task for Camera data. We assume that the delay between providing a video frame by the camera and the reception of the respective motion vector by model component is constrained, and should be between $40ms$ and $80ms$. As we already observed, simple delay constraints do not cover causality as concise as probably needed. Hence, we suggest a refined specification as shown with the specification (A) "`age(frame,vector) ...`", which is an abbreviation of a delay constraint of the form "`Delay between` ◀(`vector`,`frame`) `and frame ...`".

Age Constraints The concept of specifying age constraints can be taken even a step further. Causality correlations enable to reason about the origin of events. In the example shown in Figure 3.9, the events that occur at the sensors have no predecessors and thus are *source* events. In cases where the original source of an event occurrence, say e, can be determined, we could state specifications such as "`age(e,origin(e)) ...`", or even shorter "`age(e) ...`". This is shown with specification (B) in the figure, which is an abbreviation of specification (A).

We will discuss below further applications of this powerful specification concept. Note, however, that the concept is also dangerous as it imposes issues in cases where no such origin exists or can not be determined. Design guidelines should ensure that such specifications are considered invalid if the origin cannot be determined by an analysis.

Data Consistency Constraints The component "Dynamic Control Model" receives data from multiple inputs. In order to maintain a consistent model of the car dynamics, it must be ensured that

the sensor data represent coincident states with respect to time. Causal relation enables expressing such constraints. Specification (C) states a synchronization constraint on the inputs of the dynamic control component. It refers to the original sensor events via event correlation functions. The specification states that for any two occurrences of the event `vector` and `dist`, respectively, all corresponding occurrences of the source event `frame` and `revolution`, respectively, must have a maximal distance of $50ms$. Specification (D) rephrases specification (C) based on `age` constraints.

Note that the data flow from sensing Wheel revolutions toward the taken distance involves asynchronous communication, and is neither under- nor over-sampling. Without event correlations, it would not be possible to specify the respective constraint without semantic issues.

3.2.5. Interaction Specifications

One of the main objectives of this report are timing related specifications to reason about (complex) interaction between various subsystems. This includes both synchronous as well as asynchronous interaction as well as complex interaction protocols. The previous sections provide the essential ingredients enabling expressing also complex interaction patterns. The present and the following section discuss concrete application scenarios.

Figure 3.9 shows a typical situation where frequency of data provision and reception differ. Here, the model component periodically polls distance values from the Wheel sensors, which are provided sporadically. This causes the specification (D) to become violated, which is due to the fact that the Wheel sensor provides updates with minimal frequency of $10ms$, but no maximal frequency (infinity). Hence, also the delay between a `dist` event at the model component and its correlated revolution sensor event might grow infinitely large. As the latency between video frames and motion vectors is bounded (as discussed above), also the distance between frame events and revolution events in unbounded.

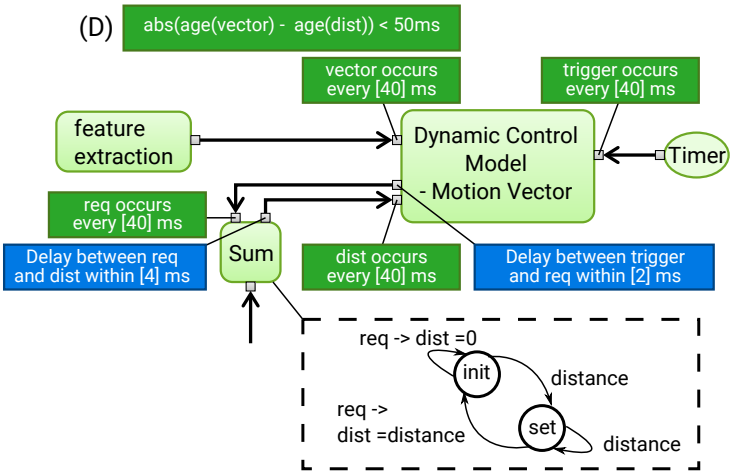


Figure 3.10.: Application of Event Correlation for Interaction Specifications

The issues could be mitigated by further refining the model as shown in Figure 3.10. The model component is now activated periodically by a timer event. The blue specification states that whenever the component is triggered, it sends a request to component `Sum`. The component has two modes. One mode represents the situation where no distance updates have been obtained since the last request.

With the knowledge of the mode transitions in component Sum, specification (D) can now be discharged. The specification "dist occurs every [40] ms" is satisfied due to the chain of specifications form the trigger event, over the request event, up to the reply of component Sum. Furthermore, the origin of ever occurrence of the event dist at the model component is known. In the initial state (init), the event comes originally from component Sum itself. Its age is therefore 0. Otherwise, the origin of the event is the last occurrence of the event distance, and we assume to be able to determine its age.

The example not only shows the interplay of different proposed extensions of timing specifications, but also non-trivial interaction between components. With these extensions we will also be able to cover other relevant interaction paradigms, which are desirable in the context of ADAS/AD design. This particularly includes "standard" interaction semantics for multi-rate systems. We will come back to this matter in Section 5.6 when discussing interaction between components with different underlying models of time.

3.2.6. Exception Handling

Exception handling is a well-established design pattern. In the automotive domain for example watchdogs are used to enforce timeout conditions for task executions.

Figure 3.11 shows two situations where exceptions are relevant in the considered application domain. The top left shows the communication of the vehicle system with a backend layer, where up-to-date information of the digital map around the trajectory of the car is requested. Communication with the backend must take unreliable communication media into account. Hence, this function must not rely on worst-case response times, but offer a robust fall-back strategy (such as using pre-cached digital maps) when a timeout indicates some failure in the communication path. This is realized by a watchdog component that raises a timeout exception if no reply to a request is received within a defined time bound.

The right part depicts the interaction between the trajectory planning and the segment execution control. The constraint at the right top of the figure states that the trajectory planning component delivers a path whenever it receives an update from the mission space model. Note that the interaction between two components is simplified. We rather assume that the trajectory planning is executed depending on the current scenario and requests updates from the mission space model by some service interface similar to the one modeled for the backend communication. However, the specification also states that the calculation of a new path might be terminated due to the occurrence of an obstacle detected by the trajectory segment execution control.

Exception handling is a first class citizen of timing specifications because of semantic subtleties. The specification "whenever request occurs ... except timeout occurs within 42ms" at the watchdog component is very similar to the (more general) disjunction of (possible) behavior. The pattern defines that either an event reply occurs within 40 and $42ms$ following a request, or the event timeout occurs. However, the timeout must not occur if a reply event occurs before the timeout interval elapsed.

Another reason for introducing a distinct concept for exceptions is a methodological one. Later in the design process toward the technical realization, components may be implemented synchronously as well as asynchronously. A synchronous realization will remain active while waiting for a reply to a request. In such cases, the components execution (busy waiting) is typically terminated. The except pattern supports the design process by indicating the use of such terminating execution. Similar observations hold for asynchronous realizations where the component may be put into a waiting state, or is (re-)activated by the incoming reply event.

Note that the specification comes in two different roles, the watchdog component as the "provider" of the timeout event, and the digital map component as "consumer".

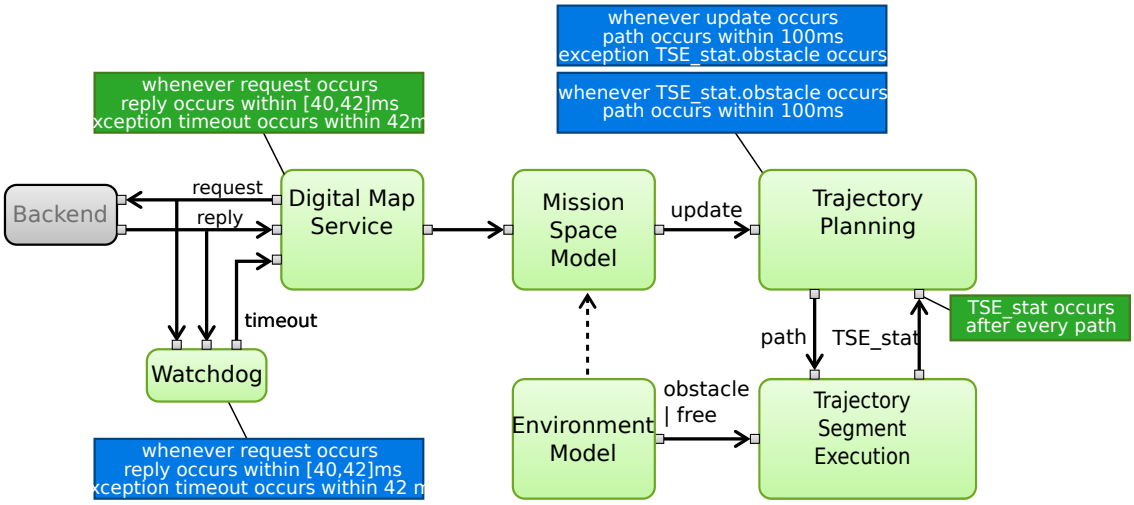


Figure 3.11.: Timing Specifications For Exception Handling

3.2.7. Probabilistic Timing Phenomena

Technically, the extension of simple interval-based timing specification towards probabilistic ones has been mentioned above. Probabilistic specifications have (at least) two use cases. First, precise bounds for timing behavior are often very hard to determine, and probability distributions allow capturing uncertainties. Second, probabilistic considerations of system behavior sometime fit better the intended engineering approach. For example, processing of camera streams may be allowed to drop individual frames in order to reduce processing utilization in certain situations.

In the project TIMMO-2-USE, this kind of extensions has been instantiated by the introduction of probability distributions [67]. Uniform, normal, Gumbel, Fréchet and Weibull distribution functions are considered for timing constraints with a time interval $[lb, ub]$. Except for a uniform distribution, the support of the other distribution functions is infinite. In these cases the time interval $[lb, ub]$ gives rise to truncate the distributions at these bounds. Probability distributions is particularly useful in combination with weakly hard specifications as discussed at Page 33.

Another possible extension would be to specify probabilities for whole constraints. One could for example specify that a synchronization constrain must hold with probability 0.9999. This opens a whole new area of specifications (and analysis challenges), including safety considerations.

These two types of extensions can be exploited for example for probabilistic timing analysis approaches, such as suggested in [36]. Here, task execution times are defined in terms of probability distributions for an (abstract) automotive engine control system. Due to potential overload situations, some end-to-end latencies could be violated. A probabilistic timing analysis would be able to calculate the probability that this happens. This way, it could be verified whether a probabilistic timing specification is satisfied.

Another interesting aspect in the model given by [36] is the fact, that in overload situations, task activations are simply omitted. The OSEK standard explicitly allows to define a maximum number of activations for individual tasks, and that activations above the maximum shall be ignored. In practice one is often interested in the question how often this happens within a given time frame (see discussion on the concept of weakly hard constraints in Section 3.2.2). In [67] only one kind of weakly hard constraint is considered, where a constraint must be met at least m out of k instances of the constrained behavior. However, the publication [15] cited by this TIMMO-2-USE deliverable also

indicates further constructs of weakly hard constraints and indicates possible areas where they can be applied. These variants are also expressed using two numbers with a different semantic interpretation: A constraint must be met at least in m *consecutive* instances out of k instances of the constrained behavior. This interpretation is obviously different from requiring that the constraint is met *any* m out of k consecutive instances of the constrained behavior. A slight variant of the constraint swaps the role of met and violated constraints by requiring that for k instances of the constrained behavior at most m consecutive instances may violate the constraint. For example, this style of weakly hard timing constraints is useful for systems processing audio or video streams, where the quality of the output produced by such systems is sensitive to the number of consecutive misses of deadlines. On the other hand, if deadline misses occur non-consecutively the impact on the quality of the audio or video signal is lower. Especially in the context of ADAS/AD development, one can expect to find functions where such constraints are applicable, for instance when processing frames sent by a camera.

3.2.8. Uncertainty Impacts

The last aspect considered here concerns the impact of timing on the functional behavior. Reconsidering for example Figure 3.11, trajectory planning highly depends on the age of information obtained from the mission space model. The older data about surrounding vehicles, the less reliable they are. If these data also come from the backend layer, timing specifications that reason about the age of event occurrences as above is not sufficient any more. Here, age of data must be provided by the source of the data, for example in terms of time stamps. This however imposes additional uncertainties such as due to the use of different local clocks.

As discussed in Section 3.2.4, causal event correlations enable reasoning about the age of event occurrences with respect to their sources. This has been specified with the pattern "`age(<event>) within <timespec>`", where `age(<event>)` refers to the origin of the event occurrence, i.e., to `origin(<event>)`. The age of origin events has been assumed to be 0. Figure 3.12 shows an example of how this concept could be further extended. Here, the backend component gets a specification about the age of replied data, which is between 200 and $1000ms$. In fact, the specification could be considered as "virtual" time stamps for data from the backend. Uncertainties about these time stamps are expressed as intervals (in this particular case). Also probabilistic specifications would be possible, but are omitted to keep simplicity.

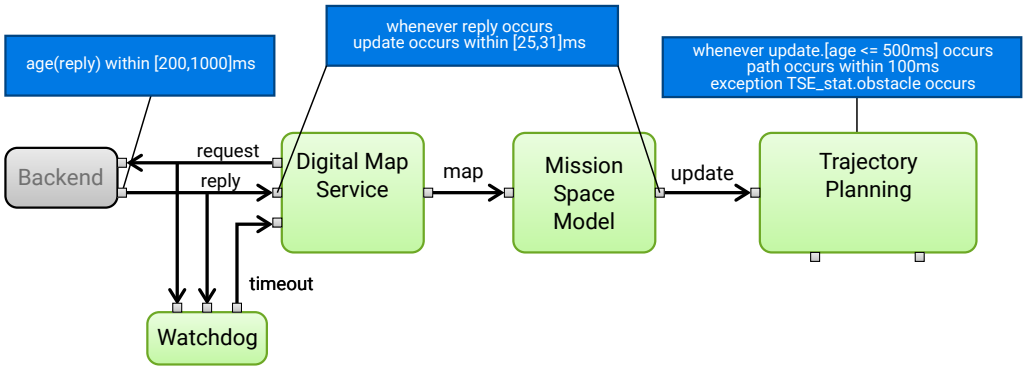


Figure 3.12.: Timing Specifications For Time Stamping and Uncertainties

The extended specifications to reason about the age of events can further be combined with the concept for data dependencies discussed in Section 3.2.3. To this end, we assume that every event occurrence has a value `age`, which is consistent to the age specification as discussed in Section 3.2.4.

This is shown with the specification for the trajectory planning component. The part `update.[age <= 500ms]` specifies the case where the value `age` of the `update` event is less or equal $500ms$.

The specification at the top center shall give an example on how the age for the `update` event could be calculated. The specification `age(update,origin(update))` refers to the occurrence of the event `reply` with which the individual occurrence of the `update` event is correlated. The age of every `reply` event is between 200 and $1000ms$, according to the specification at the backend component. If the specification at the top center is satisfied, then the age of the `update` event is equal to the age of the `reply` event plus between 25 and $31ms$, i.e., in the interval $[225, 1031]ms$.

3.2.9. Architecture Specific Timing Aspects

Up to now, timing phenomena have been discussed mainly at the functional design level, where the technical realization on hardware architecture is not considered. The timing specifications considered so far are however applicable to the technical architecture as well. This is evident due to the fact that the core part of the considered timing specification are consistent with, e.g., AUTOSAR Timing Extension, where event recurrence patterns and end-to-end latencies, among others, are relevant. In summary, the timing specifications considered here are both relevant and applicable at all design phases, including the functional as well as the technical architecture.

Not considered so far are timing effects that are specific to the technical realization on the execution platform. This includes effects due to inferences by concurrent activities, such as computations on processing units, memory and cache accesses and communication. These effects are typically captured by resource usage constraints like worst-case execution times (WCET), and recently by resource provision models [77, 80] that allowing discharging such constraints by exploiting characteristics of the underlying scheduling policies of the technical architecture.

This report proposes a design strategy where these aspects are handled in particular design steps, where we speak about the technical *realizations* of systems. The underlying fundamental design paradigms will be discussed in Chapter 4. Chapter 5 then discusses Models of Computations allowing to design systems at the technical design level, such as provided by the AUTOSAR design methodology. Together with the concepts introduced in Chapter 4, this provides for continuous design processes as proposed in Chapter 6, where all such timing aspects can be tackled.

3.3. Summary

This chapter collected relevant timing phenomena and discussed existing approaches for their specification as well as desired extensions to fill potential gaps. The approach was to specify these extensions in a consistent way allowing as smooth evolution as possible. However, the suggested extensions are only tentative. Some of them are rather obvious and have foreseeable consequences with respect to analysis. Other extensions might impose greater effort, including to derive design guidelines to guarantee decidability of such extended timing specifications. Investigation potential issues and how to resolve them would be a matter of further work.

The relation between the identified time phenomena and particular specification patterns as discussed above is summarized in Table 3.1. The crossed cells show which specifications patterns contribute to the specification of the corresponding phenomena. In other words, a cross means that the pattern may be involved in the specification of the time phenomenon. Note that the relation sometimes goes the other direction. Mode dependent phenomena require the involved specification patterns to provide respective support. This particularly holds for the specification of probabilistic effects. A somehow different position holds the specification of architecture specific aspects. It has been argued that no particular specification patterns are required here. On the other hand may all patterns be desired for covering those aspects.

The two right most columns do not specify timing phenomena. The column 'Reference' provides pointers to formal definitions of the corresponding specification patterns, which can be found in the appendix of this document. The column 'Use Case' anticipates Part II of the document, and shows of which specifications it makes use. The use case has been developed in consultation and together with the members of the VDA working group in order to illustrate the application of the design approach for a relevant scenario. It is necessarily kept simple to limit the development effort to reasonable measures, and thus does not subsume all identified phenomena.

	Timing Phenomena											
Specification Patterns	Event Occurrence	Event Relations	Causal Event Relations	Data Dependency	Mode Dependency	Complex Interaction	Exception Handling	Probabilistic Effects	Uncertainties	Architecture Specific	Reference	Use Case
Periodic Events	x			x				x		x	D.2	✓
Event Repetition	x			x				x	x	x	D.2	✓
Ordering		x	x	x	x	x		x	(x)	x	D.3, D.4	
Delay		x	x	x	x	x		x	x	x	D.3 - D.6	✓
Synchronization		x	x	x	x	x		x	x	x		
Execution Time		x		x	x			x		x		
Exception						x	x		x	x		

Table 3.1.: Timing Phenomena and Specifcations

We like to point out that, though we exclusively used textual patterns in this chapter, all extensions could also be integrated into existing model-based timing specification languages like the AUTOSAR Timing Extension [6] and the Timing Augemented Description Language (TADL) [67]. An initial elaboration on how the different kinds of specifications relate can for example be found in [72]. Further investigation of formal descriptions and possible extensions is subject of Part II.

4. Compositional Semantic Framework for Handling of Timing Aspects

Model based development is today generally accepted as a key enabler for coping with complex system design due to its capabilities to support early requirement validation and virtual system integration. While initially the overhead introduced by additional modeling activities for specification and design models and the costs of maintaining coherency of such models and their implementation slowed down the introduction of model-based development, the benefits of such frontloading of processes in achieving high-quality design and avoiding deep iteration cycles is increasingly seen as key benefit by systems industries, with sector- and application-specific penetration rates reaching close to 100% such as for primary and secondary flight control in aerospace, and engine control or dynamic stability control in automotive. Methods used depend on design layer and application class, such as the use of SysML or AADL for complete system modeling, Matlab/Simulink for control-law design, and UML, Scade and TargetLink for detailed design. Today's state-of-the-art in model based design includes automatic code-generation, simulation coupled with requirement monitoring, co-simulation of heterogeneous models such as UML and Matlab/Simulink, model-based analysis including verification of compliance of requirements and specification models, model-based test-generation, rapid prototyping, and virtual integration testing. We also like to point out the additional role of model-based design in enabling design-space exploration, architecture evaluation and platform-based design to the subsection addressing the challenge of overall optimization of the system under development. We refer to the projects SPES2020 and SPES_XT[1], MBAT[2], ARAMiS[3] and AMALTHEA4public[4], to name only a few where respective methods and techniques have been studied.

While thus model-based design is already today instrumental in improving product quality and boosting productivity in complex embedded system design, it is largely focusing on architecture and function. Non-functional aspects such performance-, timing-, power-, or safety are typically addressed in dedicated specialized tools using tool-specific models, with the entailed risk of incoherency with models actually driving design and implementation, and models used to assess such non-functional characteristics of designs. To counteract these risks, meta-models encompassing multiple views of design entities, enabling co-modeling and co-analysis of typically heterogeneous viewpoint specific models have been developed. Examples include the MARTE UML profile for real-time system analysis [1] and the Metropolis meta-model [22] [5]. Along the same lines, the need to enable integration of point-tools for multiple viewpoints with industry-standard development tools has been the driving force in providing the SPEEDS[6] meta-model building on and extending SysML, which has been demonstrated to support co-simulation and co-analysis of system models for transportation applications allowing co-assessment of functional, real-time and safety requirements, and forms an integral part of the meta-model-based interoperability concepts of the CESAR reference technology platform[7].

[1] http://spes2020.informatik.tu-muenchen.de
[2] http://www.mbat-artemis.eu/home/
[3] https://www.projekt-aramis.de
[4] http://www.amalthea-project.org
[5] The term meta-model is intended here in the semantic domain, i.e., a sort of abstract semantics, while in the traditional use of the term, meta-model is a structural concept and corresponds to abstract syntax.
[6] http://www.offis.de/f_e_bereiche/verkehr/projekt/projekte/speeds.html
[7] https://artemis-ia.eu/project/1-cesar.html

4.1. Architecture Modeling Framework (AMF) for Component-based Design

Component-based Design subsumes different strategies to cope with complex systems. First, it incorporates layered design, which focuses on those aspects of the system pertinent to support the design activities at the corresponding level of the V diagram. This approach is particularly powerful if the details of a lower level of abstraction are encapsulated when the design is carried out at the higher level. Layered approaches are well understood and standard in many application domains. As an example, AUTOSAR defines several abstraction levels.

The benefits of using layered design are manifold. Using AUTOSAR's layer structure as example, the complete separation of the logical architecture of an application (as represented by a set of components interconnected using the so-called virtual function bus) and target hardware is a key design objective of AUTOSAR, in that it allows complete decoupling of the number of automotive functions from the number of hardware components. In particular, it is flexible enough to mix components from different applications on one and the same ECU. This illustrates the double role of abstraction levels, in allowing to focus completely in this case on the logic of the application and abstracting from the underlying hardware, while at the same time imposing a minimal (or even no) constraint on the design space of possible hardware architectures. In particular, this allows re-using the application design across multiple platforms, varying in number of bus-systems and/or number and class of ECUs. Such design layers can, in addition, be used to match the boundaries of either organizational units within a company, or to define interfaces between different organizations in the supply chain. The challenge, then, rests in providing the proper abstractions of lower-level design entities, which must meet the double criteria of on the one hand being sufficiently detailed so as to support – among others – virtual integration testing even with respect to non-functional viewpoints on the next higher level, while at the same time not overly restricting the space of possible lower-level implementations.

Whereas layered designs decompose complexity of systems "vertically", by splitting the design into multiple design layers - denoted *abstraction levels* in this framework - component-based approaches additionally reduce complexity "horizontally" whereby designs are obtained by assembling (composing) strongly encapsulated design entities called "components" equipped with concise and rigorous interface specifications. While these interface specifications are key and relevant for any system, the "quality attribute" of perceiving a subsystem as a component is typically related to two orthogonal criteria, that of "small interfaces", and that of minimally constraining the deployment context, so as to maximize the potential for re-use. "Small interfaces", i.e., interfaces which are both small in terms of number of interface variables or ports, as well as "logically small", in that protocols governing the invocation of component services have compact specifications not requiring deep levels of synchronization, constitute evidence of the success of encapsulation. The second quality attribute is naturally expressible in terms of contract-based interface specifications, where re-use can be maximized by finding the weakest assumptions on the environment sufficient to establish the guarantees on a given component implementation.

In this section, we introduce the *Architecture Modeling Framework* (AMF) building the semantic base for Component- and Contract-based Design enabling to formally represent the key design steps in model based development.

4.1.1. Abstraction Levels and Perspectives

One keystone of the Architecture Modeling Framework is the concept of abstraction levels and perspectives, which is inspired by partitions of EAST-ADL abstraction levels [24]. While abstraction levels allow the support of layered design, perspectives (as certain kinds of viewpoints) enable to regard the system from *structurally* different points of view. With growing system complexity as induced,

e.g., by automated driving functions such concepts are essential to be able to manage the system complexity at all.

First, this is due to the necessity to represent organizational issues also in the design process and in particular in the system models. For example, if a model is handed over from one organizational unit to another (or even to several different units) this should be explicitly visible in the representation of the design process as a kind of intermediate result. This is a prime example where a transition from one abstraction level to another may be carried out. Second, regulatory issues induce the need to thoroughly document the design process and design decisions for a successful homologation and type approval, respectively. The combination of abstraction levels and perspectives (resulting in a matrix structure) allows to represent relevant intermediate results of each executed design step as well as the relations to previous results. Third, the more complex a system is, the more a separation of concerns is inevitable to be able to cope with that complexity. Here, in particular the concept of perspectives helps to focus on the system characteristics that are currently relevant (and to hide those that are not) while still being able to unambiguously link model elements of different perspectives to keep the models consistent.

Abstraction Levels

A system under development or a constituting design item (e.g., a component) can be modeled on different abstraction levels. Less abstract models belonging to a lower abstraction level may increase the degree of detail of the description of the design item at hand. A car for example may be modeled at the top-level as a single component where only the interface of the car to the environment is visible. At a lower level, certain components of the car may become visible (chassis,engine, ...), and so on. This means that the interface description may be refined as well as the demanded behavior. Furthermore, the component hierarchy modeled within a certain abstraction level may be re-arranged within the next lower abstraction level. Thus, a component-based architecture can not only be refined by decomposition of components (see Section 4.3.1), but also by means of a different component hierarchy. So refinement subsumes but is not restricted to simple decomposition of components.

Therefore increasing the level of detail, thus at the same time lowering the level of abstraction, adds knowledge about the design item. Often reducing the level of abstraction is accompanied by a refined granularity. That is, the design items under consideration are decomposed into multiple finer grained items in order to keep them at a manageable size. Refining the granularity is however not a necessary condition, when introducing a lower level of abstraction. For example one may describe the same design item on two different levels of abstraction with the same granularity, but render more precise the specification of a certain aspect. Despite an increasing level of detail, another motivation for introducing a dedicated abstraction level can be the handover of an initial system specification to another organizational unit. Thus the initial specification can remain unchanged and on a new abstraction level the model can be refined as well as the interface specifications of the components. Realization links between component parts allow to trace those refinements. While models on lower abstraction levels provide more detail, the components still have to respect the aspects specified for their higher level counterparts.

In Section 4.3 we discuss the key design steps of switching abstraction levels by means of Realize-links. In discussions with industrial partners during several projects it became clear, that design processes are very diverse for different domains like, e.g., automotive and avionics. It depends on the design process and the role of a company in the supplier chain, how many levels of abstraction a system under development will undergo. In addition the domain specific terms used for designs at a given level of abstraction vary. Therefore it is not useful to define a unique fixed set of abstraction levels. Instead we consider a generic abstraction level concept, which can be tailored to the specific needs of a company or an application domain and provide semantics allowing to reason about realization

between different abstraction levels.

Viewpoints, Perspectives and Aspects

Beside the distinction of different user-defined abstraction levels, a model of a system under development may be associated with certain *Viewpoints*, such as defined in [48]. The Architecture Modeling Framework differentiates between viewpoints that establish structurally different *Perspectives* on the system, and viewpoints that are orthogonal to the structure of the system and describe different *Aspects* of functional and/or non-functional properties of model elements. Concerning aspects, in the context of this work, we focus on the *Timing Aspect* and where necessary we also take the *Functional Aspect* into account.

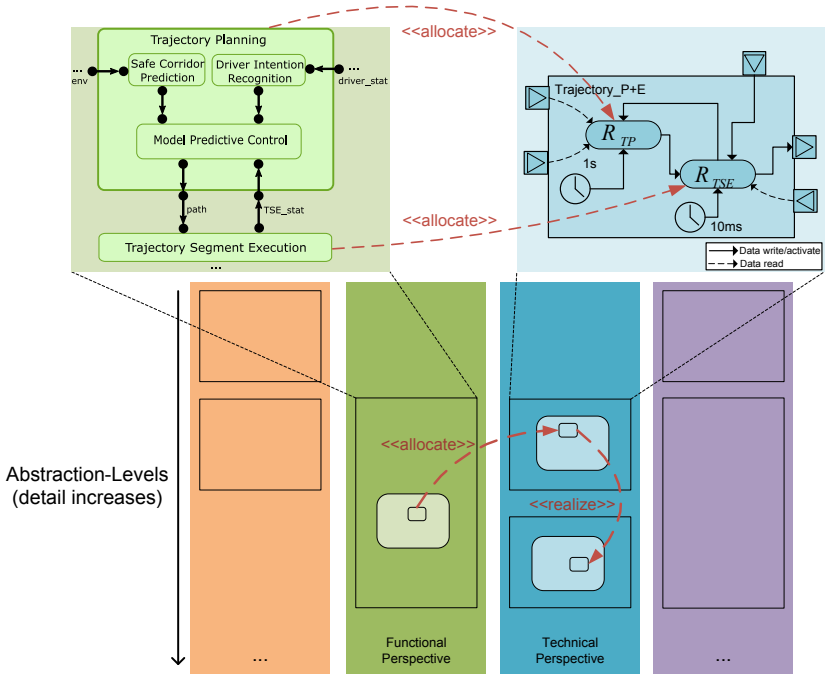


Figure 4.1.: The concept of perspectives and abstraction levels at the example of the functional and technical perspective

In general the choice of the set of perspectives to be used for a system design depends on the application domain and the individual design process and use case. In cooperations with industrial partners from different domains (in particular automotive, avionics and automation engineering) within several projects (such as SPES2020, SPES_XT and CESAR) we identified typical perspectives that are relevant for system design in these areas, which are the operational, functional, logical, technical and geometrical perspective.

In the *Operational Perspective* the context of the system is considered. This means the embedding of the system under development in its environment is modeled and analyzed including the interactions between system and environment. As a result the system boundary is determined, i.e., its interface, as well as requirements on the interface behavior. Another purpose of this perspective is to identify the customer needs as one of the first steps towards a concrete functional system design that fulfills

these needs. In the *Functional Perspective* the functional needs of the system are identified and their breakdown into sub-functions is modeled that together realize the top-level system functions. Ideally, functions are completely independent of any architecture. Among other things this perspective allows to represent most results of the ISO26262 process activities.

In the *Logical Perspective* the system under development is decomposed by means of hierarchical components that implement the functions the system needs to perform. Thus, the logical perspective serves as a partitioning of functions driven by concerns like performance, reusability and management of the supply chain, i.e., to consider the interfaces between organizations in the supply chain. However, any kind of resources or properties of a target platform are still neglected here.

In the *Technical Perspective* the electrical and electronic architecture, the software and mechanical parts of the system under development are described. This encompasses mechanical, hydraulic parts, and electronic control units, buses, and software tasks, where the functions and logical components, respectively, shall be allocated to. While in the functional and logical perspective it is usually assumed that each component can always be executed, in the technical perspective resource limitations come into play. Some resources, e.g., electronic control units, will be shared by multiple components, whose behavior will therefore depend on the properties of the resource as well as other components allocated to the very same resource. In the automotive domain the technical perspective, for example, may contain AUTOSAR models.

In the *Geometrical Perspective* the physical layout of the system is considered. While in the technical perspective some resource is modeled in terms of its properties, its interfaces and its behavior, in the geometrical perspective its spatial dimensions are modeled. The geometrical perspective deals for example with cable lengths and space limitations.

In this report we restrict to the perspectives that we consider as being most relevant in automotive ADAS/AD design, which are the functional and the technical perspective.

The Matrix

The combination of the concepts of abstraction levels and perspectives results in a two-dimensional organization (*Matrix*) of component models as shown in Figure 4.1 where as examples excerpts of the functional (colored green) and the technical perspective (colored blue) of the running example system are depicted. While the functional part is a refinement of Figure 2.3, the technical part is an example for a representation of these functions in an AUTOSAR model. Here, again we see a surrounding component with ports (drawn as squares with an arrow indicating the port direction), which is decomposed into two AUTOSAR runnables R_{TP} and R_{TSE}. Additionally, the component contains two triggers (clocks), one triggering R_{TP} with a period of 1 second and another one triggering R_{TSE} with a period of 10 milliseconds. According to the legend, in the technical perspective dotted arrows mean data read accesses and solid arrows mean data write accesses or activations. Between the models of the different perspectives *allocate*-links are drawn indicating how the components of the functional perspective are mapped to the technical perspective. In this case the functional component `Trajectory Planning` constituting of three subcomponents is allocated to the runnable R_{TP} of the technical perspective and the functional component `Trajectory Segment Execution` is allocated to the runnable R_{TSE}. More details involving the mapping of ports between perspectives and abstraction levels can be found in Section 4.3.2.

In general, the relationship between models of different perspectives is not necessarily an isomorphism, i.e., the components do not need to be in a one-to-one relation. For example, three interacting functions defined at the functional perspective may be modeled as two components in the technical perspective. Of course the models of the very same system from different perspectives need to be consistent and it must be ensured that the technical realizations indeed implement the specifications of the identified functions of the system. The *realization* and *allocation* links discussed in Section 4.3.2

provide support for this activity by establishing a traceability of requirements along the different abstraction levels and perspectives, while at the same time inducing proof obligations for demonstrating their consistency.

4.1.2. Components

We already briefly introduced the concept of components in Section 3.1 and will go in more detail here. The paradigm of Component-based Design is well established and used in many different application domains. In standards like EAST-ADL [24] and AUTOSAR [7], a component is the basic design-entity structuring any model. A component has a well defined interfaces, which allow to abstract from the internals of a component and to reuse it in different design contexts. The notion of components can be established independently of any design domain. That is, a component can model a reusable piece of software, a specific sensor measuring temperature or a computing device (e.g., ECU or IMA module) hosting software applications.

Ports

Ports are the interaction points of a component. Together with their types they define the syntactical interface of the component. A port can either define a data-oriented interaction point or service-oriented interaction point. Further, they have a direction and are either input port (sometimes called required ports) or output ports (provided ports).

Hierarchy and Composition

The AMF component model adopts the type/part (sometimes also denoted prototype or property) concept found in many other modeling languages, e.g., UML [64], SysML [63], EAST-ADL [24] and AUTOSAR [7]. Figure 4.2 shows an illustration of using that concept for building compositions by reusing other components. Here the component `Trajectory Planning` is modeled with its ports and its constituting components `Safe Corridor Prediction`, `Driver Intention Recognition` and `Model Predictive Control`. These parts are connected via connectors and ports with each other and with the ports of the surrounding component. The type/part concept results in a possibly deeply nested hierarchy of components ultimately leading to some top-level component specification. Such a hierarchy of components can be constructed for every combination of abstraction level and perspective (i.e., every "cell" of the matrix), representing a view of the system under development.

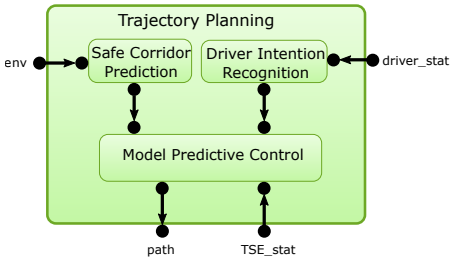


Figure 4.2.: Hierarchy of Components at the example of Trajectory Planning

Obviously, the type/part concept allows for top-down and bottom-up design approaches, as well as mixing them. To decide whether a component can be used in a given context and connected with other components, the ports that are connected must be compatible. On the one hand, there exists a notion of syntactical compatibility, which is also addressed in the languages mentioned above. On the

other hand compatibility can also be regarded from a behavioral point of view. To this end, AMF combines Component-based Design with the Contract-based Design paradigm as described next.

4.2. Contract-based Design

System companies such as automotive and avionics are facing a rapid increase in complexity of their products. Combined with shorter time-to-market, this poses a great challenge on product design and development. VDC research [8] shows that system integration is a major factor of complexity and is often the cause of tasks being delayed in the development of a product. Indeed, a recurrent reason for failures causing deep iterations across supply chain boundaries rests in incomplete characterizations of the conditions for use and environment of the system to be developed by the supplier. This highlights the need to explicate assumptions on the design context of components and subsystems in OEM-supplier chains. This explication of assumptions helps assigning responsibilities to stakeholders for each design entity. The paradigm of *Contract-based Design* emphasizes this important role of assumptions in specifications.

Contract-based Design is a paradigm, where specifications of design components not only express what a component is supposed to do, but also what the legal context for a component is (see Appendix C for a brief overview of contract semantics and corresponding literature). That means a contract expresses assumptions about the environment of a component. Further, it states a required behavior that must be guaranteed by the implementation of a component, provided it is used in a context that is compliant with the assumptions about the environment specified by the contract. This split of a specification into assumptions and guarantees enables compositional reasoning schemes, where implementations of contracts can be developed *independently*. Proper integration of the final implementations of components can then be checked based only on their contracts. This reduces complexity of the verification and also enables a frontloading of the verification of proper system integration. Thus, integration issues are detected early in the design process, before the actual implementations are developed. Note that a contract deliberately does not impose any restrictions on the behavior of a component if it is used in a context that is not compliant with the assumption. This reflects that components are not designed to work in an arbitrary context. To exemplify this, consider the following example taken from [13]:

A component M computes the division between two inputs x and y and returns the results on its output z. The underlying assumption is that M will be used within a design context that never submits $y = 0$. Of course M cannot prevent its environment of violating this assumption, since x and y are inputs not under its own control. So the idea is that in this case M can behave in an arbitrary manner if given an input $y = 0$. For example, it might always return the value $z = 0$ in this case. This is in fact quite similar to the concept of *undefined behavior* known to programming languages like C/C++. For example, accessing an array out of its bounds could result in a machine exception if an illegal address is accessed or the program might just continue execution with completely unpredictable results. Further, if any step in the execution of a program has undefined behavior, then the entire execution is without meaning. In general, this is also true for contracts. If at any point in time the environment behaves in a way not compliant with the assumption of a contract, the implementation of a contract is free to do anything. Thus, a contract can be seen as an underspecified description of a component. Obviously, one needs to ensure that a component with an associated contract is only used in contexts compliant with its assumption.

[8] VDC research, Track 3: Embedded Systems Market Statistics Exhibit II-13 from volumes on automotive/industrial automation/medical, 2008

Specifying Contracts The theory of contracts as defined in [14, 13] is deliberately abstract and imposes neither restrictions on the language with which assumptions and guarantees are expressed, nor on the kind of behavioral aspects of a component that are specified. This abstract denotational framework can be used for timing aspects, as well as for safety or functional behavior aspects. In this work, the focus lies on timing aspects and phenomena. To this end, we consider contracts whose assumptions and guarantees are expressed using the concepts discussed in Section 3.2. Indeed in all of the examples contained in those sections, we already distinguished between behavioral constraints taking the role of an assumption and constraints playing the role of a guarantee. For instance, the example shown in Figure 3.8 can be considered as a timing contract, whose assumption is given by the constraint shown in the green box and whose guarantee is given by the constraints shown in the blue boxes. The previous example about a component implementing a division already illustrated that it is quite natural to restrict functional specifications with regard to an assumed environment. Indeed the same is true for timing related specifications. Taking a closer look at the specifications shown in Figure 3.8, we see that the contract expresses an assumption about the rate at which new data is submitted at the input port $in1$. The guarantee part expresses the requirement that in such a context the component shall provide for each occurrence of an event at port $in1$ a corresponding value at its output port out within a time interval of $[14ms, 16ms]$. Such assumptions are very typical for timing specifications and are in fact even necessary if input values are buffered and must not get lost. In such cases assumptions about the arrival rate at inputs allows to reason about the sizes of buffers storing input values until they are processed and a corresponding output is produced.

4.3. Design Process

The AMF intends to support the development of complex systems involving a large number of different engineers. In such a process, it is essential to specify the roles and responsibilities of each engineer: who is responsible for ensuring what, and under which conditions. Only if this is ensured, integration problems in later design phases can successfully be identified and fixed.

In AMF, first class citizens are components and their interfaces. Components may interact with each other and with the environment by means of their interfaces, thus forming a model of the intended system. The AMF supports the development process in that it allows to represent roles and responsibilities of engineers by corresponding concepts in the design architecture. For example, engineers are usually responsible for distinct parts of the design. Design artifacts have to be constructed, refined, and implemented in different perspectives and at different abstraction levels. The AMF allows to (formally) express responsibilities for components in form of contracts as introduced in Section 4.2. If different engineers, responsible for certain design artifacts, have to integrate the respective parts of the design, contracts allow to define under which conditions this integration will be successful. The same holds for example, if different engineers are responsible for specification and implementation of certain components. AMF provides means to reason about under which conditions an implementation satisfies the corresponding specification.

The underlying concepts of the AMF particularly allow to define proof obligations that have to be resolved in order to ensure whether responsibilities defined for model and component interfaces are satisfied. In an early design phase for example, functionality of the system may be identified, together with a set of requirements to be satisfied by those functions. When in a subsequent design phase decomposition of the system into logical components is performed, it should be explicated which logical components are responsible to guarantee which system functionality. If this is done "the right way", the AMF allows the application of tools that, if not resolving proof obligations automatically, explicate which proof obligations have to be resolved in order to ensure satisfaction of the identified function requirements. This section intends to provide an overview of those concepts related to establish a

continuous design process, and to provide traceability of requirements and the corresponding relations between design artifacts.

It should be noted that the AMF does not define a concrete design flow. However, we have identified a set of *Key Design Steps* that are repeatedly applied during all design processes. Thus, each design step that is performed in a certain process can typically be broken into a sequence of the key steps *Decomposition*, *Realization*, *Allocation* and *Implementation*. Each key design step introduces a set of proof obligations clearly defining under which conditions all responsibilities of the involved design artifacts are satisfied. This in advance allows to define clear interfaces between the responsibilities of all engineers involved in a design process. The proof obligations are *Virtual Integration Test* (VIT) (for checking (de-)composition, allocation and realization) and *Satisfaction* (for checking implementation). Appendix C contains a formalization of these two proof obligations. In the following, the key design steps and their proof obligations are discussed in more detail. Not mentioned are, however, concrete analysis techniques and methods for these proof obligations, such as code reviews, static analysis, simulation and formal verification techniques like model-checking.

4.3.1. (De-)Composition

Technically, a component may be a (composed) system, which is constructed from a set of (sub-) components. In more detail, a system is given by a set of components, an interface and a set of named (inter-) connections. An interface is a set of directed ports, denoted inports and outports, respectively. Each port specifies a set of typed (data) flows or services. In a design process, composed systems usually do not exist a-priori (except in case of re-use), but are the result of a concrete design step. For example, due to a design decision, it may be found that the functional component `Trajectory Planning` should be realized by a composition of three subcomponents as shown in Figure 4.3. Contract-based Design helps in such situation not only to allocate responsibilities for the respective sub-components, but also to trace back whether the requirements of the component are satisfied when all of its sub-components satisfy their "local" requirements. Hence, in Figure 4.3 concrete timing contracts in terms of assumptions (green colored boxes) and guarantees (blue colored boxes) are annotated at the (sub-) components. The relations between components and contracts are denoted *satisfy*-links meaning that the implementation of a component needs to satisfy the linked contracts. More details concerning implementation and satisfaction can be found in Section 4.3.3 and in Appendix C.

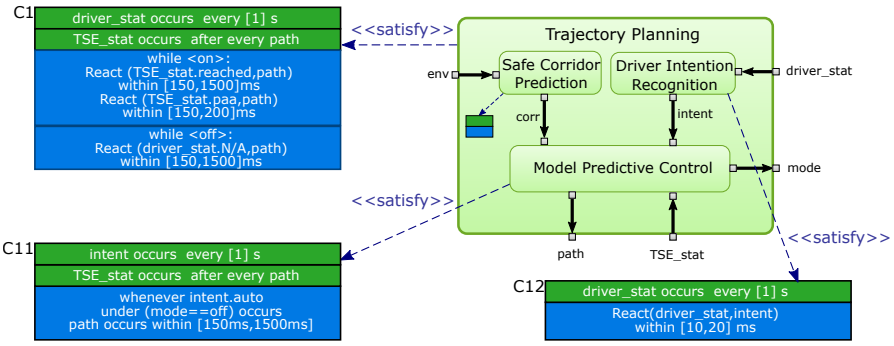


Figure 4.3.: Composition of components with contracts at the example of `Trajectory Planning`

The proof obligations resulting from a decomposition of components and their respective contracts are subsumed by virtual integration testing. Please note that the virtual integration we consider here has not the meaning of X-in-the-loop tests, which are often summarized as virtual integration tests as

well. The term "virtual" here is due to the fact, that these proof obligations are independent from the implementation of the respective components. That is, if a VIT has been performed successfully, any set of implementations satisfying the responsibilities of the respective sub-components are inherently covered by the VIT.

Proof Obligation: Virtual Integration Testing (VIT)

VIT defines the conditions under which the contracts of all sub-components, when put together, satisfy the contract(s) of the parent component, and which verification steps have to be performed in order to establish this property. The relevant property is denoted *Refinement*. To verify that two (or more) components can be put together reasonably at all - denoted *Compatibility* - VIT also defines necessary proof obligations (see condition C.1 in the appendix) . Due to the fact that components are characterized by their specifications, compatibility is defined for contracts. Compatibility reflects the distinction of responsibilities in that assumptions about the environment of a component must not be violated by another component that is connected to this component. In the example from Figure 4.3 the specifications of the components `Driver Intention Recognition` and `Model Predictive Control` are not compatible. This is because `Model Predictive Control` expects updates on the port `intent` every $1\,s$ as shown with the first assumption of contract $C11$. Contract $C12$ of component `Driver Intention Recognition` states that every update of the `driver_stat` port is answered by an update of the `intent` port within an interval of $10 - 20\ ms$. Hence, updates of the `intent` port do not occur every $1\,s$, but due to a jitter of $10\ ms$ in intervals between 990 and 1010 ms, and thus it would be problematic to put these components together.

If compatibility is given, the second proof obligation of VIT is to ensure that the composition of sub-components satisfies the responsibilities of the respective parent component. That means the guaranteed behavior of the sub-contracts must *refine* the behavior guaranteed by the parent contract, within a design context compliant with the assumption of the parent contract (see condition C.2). For the example in Figure 4.3 this means that the guaranteed behavior of the three subcomponents refines the guarantee of the surrounding component `Trajectory Planning` (the constraints shown in blue boxes), under the assumptions of the surrounding component (the constraints shown in green boxes). Note that the composition of $C11$ and $C12$ does not satisfy $C1$. The second guarantee of $C1$ states that for every update of the `driver_stat` port (and if autonomous driving is switched off) a new path is delivered within 150 and $1500\ ms$. This latency is reflected in the guarantee of contract $C11$. There is however an additional delay of $10 - 20\ ms$ before updates of the `driver_stat` port reach component `Model Predictive Control`, and thus the overall delay is $160 - 1520\ ms$.

4.3.2. Realization and Allocation

While (de-)composition reasons about components of a particular model, two further design steps are important with respect to different perspectives and abstraction levels, that is, for the interfaces between different models. These design steps are denoted realization and allocation.

Allocation represents the design of a system "from left to right", i.e., from one perspective to the other. We say the system of a certain perspective is allocated to the system in the next perspective to the right (of the same abstraction level). Typically, such a design process starts with the definition of the system boundaries, interfaces and top-level requirements (often called goals). Then the functions of the system are identified that are needed to achieve these goals. In another perspective design components are identified that perform the identified functions, and so forth. Each of such design steps usually involves an assignment of requirements to the newly identified components. In general, relations between components of different perspectives are not always one-to-one but may be in general n-to-m. For example, logical components may perform several functions (sequentially, parallel,

or in any other order), or several logical components may be involved performing a single functionality. Thus, tracing of requirements along perspectives and checking for consistency of the models of the system from different perspectives is key for assuring the correctness of individual design steps with regard to formulated requirements.

The same approach of reasoning about contracts across different perspectives can also be used for the transition between abstraction levels, which is denoted *Realization*. When a system is designed incrementally, the same system is represented at different levels of granularity. We say, the system at a certain abstraction level (and the same perspective) realizes the system at the higher levels. Although various refinement steps can be established by decomposition of components, there are other cases where changing the abstraction level is more appropriate. This may be simply due to organizational boundaries, where different departments in a company are responsible for certain subsystems. Another reason is the fact that refinement not always follows a simple decomposition pattern, but demands for more complex m-to-n relations.

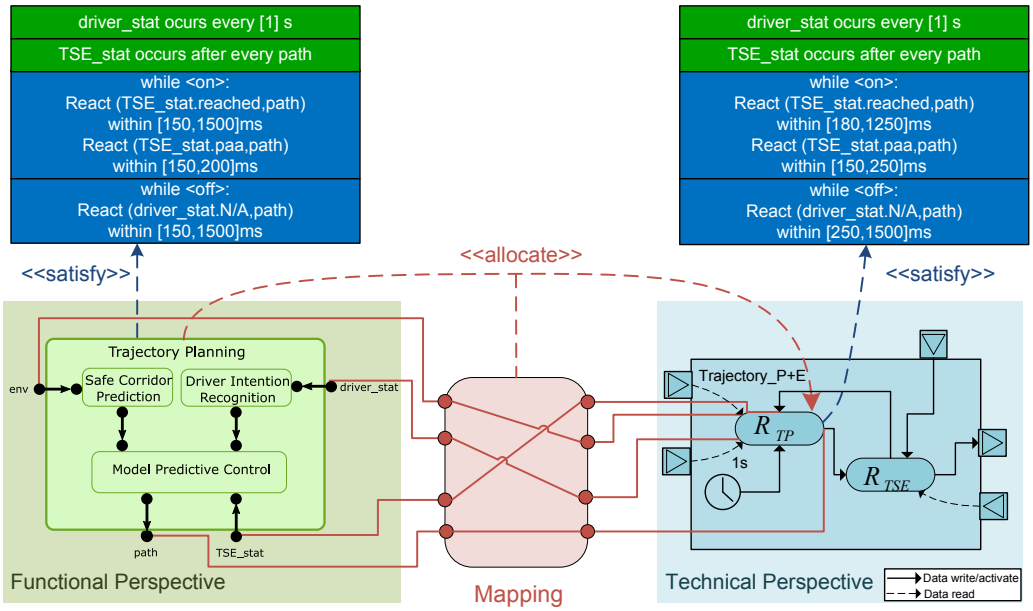


Figure 4.4.: Mapping example for an allocation with real-time contracts

Proof Obligation: Realization / Allocation and VIT

Since the requirements for establishing traceability between perspectives and between abstraction levels are very similar, the concepts of realization and allocation share the same semantics. The corresponding theoretical concept is that of realization (see [13], Section 3.7). Hence, allocation is a technical term used to denote design phases where, for example, logical components are allocated to a technical architecture.

In order to instantiate realization and allocation relations, the AMF introduces the common concept of *Mapping*. Technically, a mapping is similar to a component. It has interfaces consisting of ports. Its interfaces are however not used for interaction with other components but to relate different models.

As Figure 4.4 shows, a mapping connects (arbitrary) ports of two different models of a design, in this example a model of the functional perspective with one of the technical perspective. Mappings are asymmetric in that the roles of connected model artifacts are different.

A mapping can be considered as an observer of one of the participating models. In Figure 4.4 for example, the depicted mapping "observes" the behavior of the AUTOSAR model at the right. Due to its specification, a mapping provides a well defined (possibly modified) view to the behavior of the "observed" model. More important, the observer concept gives means to the fact that models of a design are closely related to each other. Model artifacts within a perspective at a certain abstraction level are allocated to artifacts of other perspectives, and artifacts at a lower level are considered to realize those of a higher abstraction level. In this sense, a mapping provides the foundation to reason about refinement of the specification of an observing model by the specification of another (observed) model.

Technically, a mapping can be used to "replace" components within the observing model. While replacing, the mapping gets associated with the set of contracts that have been associated to the replaced components. This gives means to the proof obligation corresponding to the question whether the observed model satisfies the contracts defined for the observing model. In fact, the situation is very similar to the one for virtual integration testing. If the resulting system, which contains the mapping and the observed model, is considered as a single component, the same proof obligations apply as for composition. The result is that, whenever the contracts of the components of the observed model are satisfied, then the contracts of the observing models are also satisfied with respect to the specification of the mapping. Thus, considering the example shown in Figure 4.4, a successful VIT of the allocation step means that the specification of the technical realization in terms of the runnable R_{TP} is consistent with the specification of the function `Trajectory Planning`. This means that the contracts of the AUTOSAR runnable need to be valid refinements of the contracts of the functional component meaning that the behavior specified by the technical contracts is completely contained in the behavior specified by the functional contracts. In this example this is not the case because the contract of the AUTOSAR runnable specifies a reaction interval of $[150, 250]$ ms between the occurrences of $TSE\_stat.paa$ and $path$ while the contract of the functional specification claims an interval with a smaller upper bound ($[150, 200]$) for the respective reaction. The other modified intervals of the technical contract are however valid refinements of the functional one.

If a technical specification is a valid refinement of its functional counterpart, the same holds for its *implementation* as long as it satisfies the contracts of the technical model (more details about implementation can be found in Section 4.3.3). This means that the VIT proof obligations allow for transitive reasoning along the trace links between different models.

As outlined above mappings provide well-defined interfaces between artifacts of models within different perspectives and abstraction levels. However, the corresponding proof obligations are valid only with respect to the specification of the mapping. Hence, confidence of the result of a refinement check also depends on the faithfulness of the mapping. Allocate- and realize-links are high-level constructs in that for each link an underlying mapping has to be created. Each such link thus requires further specification with respect to the underlying mapping. In many cases however, simple mapping functions are sufficient to define allocations and realizations. Often these just consist of one-to-one relations between ports of components of the mapped models as it is also the case in Figure 4.4.

4.3.3. Implementation

Of course the goal of any design process is to eventually have implementations of components. Depending on the perspective and abstraction level, implementations may be models like Matlab/Simulink from which code can be generated or even hand-written C-code. AMF does not assume any particular form of implementations. Instead it is left to the needs of the respective design process to reason

about satisfaction of contract specifications, and to define sufficient and necessary conditions based on the needs, e.g., for certification. Thus, there is further verification / validation effort necessary in order to demonstrate that implementations of components indeed satisfy their associated contract(s). The benefit of the approach is that these steps are isolated and can be carried out independently from each other. Various forms of implementations in terms of behavioral models are discussed in Chapter 5.

AMF provides for concepts to specify implementations for components employing the same underlying formal semantics as for contracts. This is however convenient only for some special cases and thus AMF does not rely on it. It rather provides an open and extensible approach to integrate any kind of implementations such as Real-Time StateCharts, Matlab models and even C-Code. That way, it is left to the needs of the respective design processes to reason about satisfaction, and to define sufficient and necessary conditions based on the needs, e.g., for certification. Various forms of implementations will be discussed in Chapter 5.

Proof Obligation: Satisfaction

Formally, the contracts of a component characterize allowed behavior of the component within an assumed design context (see Appendix C). Thus, any implementation of a component must *satisfy* the contracts of the component specified by a satisfy-link as can be found in Figure 4.3 and Figure 4.4. An implementation of a component satisfies its specification if its behavior does not violate the behavior specified by the component's contracts. At least two different approaches can be distinguished to establish timing specifications: formal verification and testing.

Although formal verification of timing behavior is a well-established research area and various formalisms and model-checking techniques have been developed (such as AutoFocus, FUJABA Real-Time tool suites, Uppaal), their application to problems and systems of the industrial practice is still challenging. Amongst others, this is in particular the case for the technical and low-level implementation layers and those system parts where stochastic theory plays a major role. And also exhaustive verification is still a challenging task due to complexity reasons. Beside such general techniques, specialized analytic approaches such as real-time scheduling theory can be used as well. Here, implementations of the involved components, usually called tasks, are taken in order to calculate execution times for invocations of the respective components. Due to a characterization of the underlying (computational) resource, scheduling analysis calculates worst-case scenarios in order to verify whether the requirements about maximal delays are satisfied. Since timing contracts usually define activation scenarios for components, and maximal delays, real-time scheduling theory fits well to establish timing contracts. For more complex timing contracts, and at higher abstraction levels where no detailed characterization of the underlying resources exist, more involved analysis techniques can be applied such as combination of scheduling analysis and model-checking.

Another way to establish timing contracts is due to testing. To this end, the implementation is executed on a prototyping platform, and execution traces are captured while running the system. In a subsequent hosted simulation, the traces are checked against (executable) contract specifications. Indeed also other ways exist for testing, such as providing an executable implementation of the specification in terms of a monitor which runs in parallel to the implementation code. Also often more abstract implementations are considered, which is the domain of simulation.

5. Models of Computation

Chapter 4 provides a common conceptual framework for system development. It discusses key design steps from which the individual development processes can be instantiated, and how these steps can be supported by V&V activities. This framework must still be instantiated by modeling and programming, i.e., implementing, the particular components and their interfaces.

The main outcome of the predecessor study on the "Future Programming Paradigms in the Automotive Industry" [60] was that *"there is no silver bullet"* for such implementations. The study concluded: *"In no way there is one single programming language able to cover all the needs of the automotive industry. Also, there will be no single language for the automotive industry in the future, instead languages are to be tailored on per sub-domain basis and interaction is to be researched in the future."* [60].

Obviously, suitable modeling and programming languages allow engineers to express system functionality in an efficient and effective way. Instead of discussing particular languages in detail, we focus on *Models of Computation* (MoC), which characterize the underlying concepts of individual modeling and programming languages. The chapter considers Models of Computation that are relevant for the development of complex ADAS/AD in the automotive domain. Where appropriate and available, tools and concrete examples are mentioned, which can be considered as instances of theses MoC. Finally, the interaction and integration of different MoCs is addressed, thus directly addressing one of the main identified needs in [60].

5.1. MoC Introduction

In the academic world, the term *Model of Computation* (MoC) was introduced to focus on the specification of concurrency and time. In literature, different definitions can be found [50, 56, 68, 75, 83]. In this document, a MoC defines the time representation and the semantics of communication and synchronization between processes in a process network. Thus, a MoC defines how computation takes place in a structure of concurrent processes, hence giving a semantics to such a structure [25, 49]. This semantics can be used to formulate an abstract machine that is able to execute a model.

As stated in [50] a model of computation should support the following properties:

Implementation independence An abstract model should not expose details of a possible implementation, e.g., which kind of processor used, how much parallel resources available, what kind of hardware implementation technology used, details of the memory architecture, etc. Since a model of computation is a machine abstraction, it should by definition avoid unnecessary machine details. The benefits of an abstract model include, that analysis and processing is faster and more efficient, that analysis results are relevant for a larger set of implementations, and that the same abstract model can be directed to different architectures and implementations.

Composability Since many parts and components are typically developed independently and integrated into a system, it is important to avoid unexpected interferences. Thus, some kind of composability property is desirable.

Analyzability Generally speaking, there is a trade-off between expressiveness and analysability: The more freedom and flexibility a MoC allows to the designer, the less properties (like schedulability,

deadlock freedom, or memory boundedness) can be guaranteed. Depending on the application (data flow or control dominated) and the requirements on the overall system (e.g., hard real-time system, no real-time constraints) the MoC is chosen. By restricting models in clever ways, one can apply powerful and efficient analysis and synthesis methods.

5.2. MoC Time Abstraction

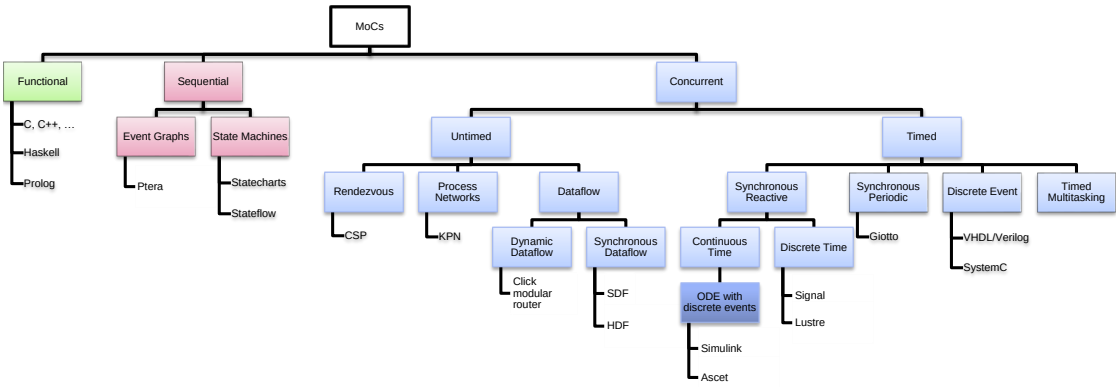


Figure 5.1.: Classification of MoCs (Ptera = Ptolemy Event Relationship Actor, KPN = Kahn Process Network, CSP = communicating sequential processes, SDF = Synchronous Data Flow, HDF = Heterogeneous Data Flow, SA-SDF = Scenario Aware Data Flow)

Following [49, 50] MoCs can be organized according to their time abstraction. Figure 5.1 shows a classification of different MoCs, following the Ptolemy classification of semantic domains. In this work, the following MoCs will be considered:

- **continuous time (CT) models:** When time is represented by a continuous set (usually real numbers), we talk about continuous time. Languages and modeling tools that support continuous time MoCs are Matlab/Simulink [59], Modelica [44] and SystemC-AMS [43]. Behavior in continuous time MoCs is usually expressed by equations over real numbers. Differential equations are used to express arbitrary internal feedback-loops. For the simulation of these MoCs differential equations solvers are used.

- **discrete time (DT) models:** When time is represented by a discrete set (usually integer or natural numbers), we talk about discrete time. In a discrete event model, events are associated with a discrete time instant. Discrete time (and especially discrete event) models are often used for hardware simulation. Examples for discrete event languages and associated simulators are VHDL [85], Verilog [86] and SystemC [42].

- **synchronous (ST) models:** Synchronous models are an abstraction of discrete time models. Time is represented by a discrete set of integers or natural numbers, but the time unit is not a physical time unit but an abstract interpretation of time. In synchronous models the following abstraction techniques are employed:
  - The occurrence of events is not precisely defined, but constrained by the beginning and the end of time slots (or time steps).

  - The timing of events that are not visible at the end of the time slots (or time steps) is not relevant and can be ignored.

  Languages and modeling tools that support synchronous discrete time MoCs are Matlab/Simulink [59] and SystemC-AMS [43].

- **untimed (UT) models:**
  - **untimed sequential**
    * State Machines, with hierarchy and concurrency extensions, expressed in dedicated languages/tools (e.g., MathWorks Stateflow, ETAS ASCET, IBM Rational Rhapsody (Statemate)) can be used to express control dominated subsystems.
    * Arbitrary sequential programming languages (e.g., Java, C/C++) can be used to describe complex functionality.
  - **data flow process networks:** For explicitly expressing concurrency or explicitly specifying potential concurrency at specification level, untimed sequential blocks can be composed in an untimed process network. Classical examples for untimed process networks are Kahn Process Networks (KPN) [52] and Synchronous Data Flow (SDF) [55].

5.3. Integration of MoCs into the Component Model

As shown in Figure 5.2, a MoC is used to describe the implementation of a component. The MoC defines the time representation and the semantics of communication and synchronization between processes in the implementation. A MoC can be realized in different programming languages. Every MoC requires precise definitions of how its interfaces map to the interface concept of the component model. For example, in order to use a Matlab/Simulink model with this approach, a mapping of source and sink signals to component ports is required.

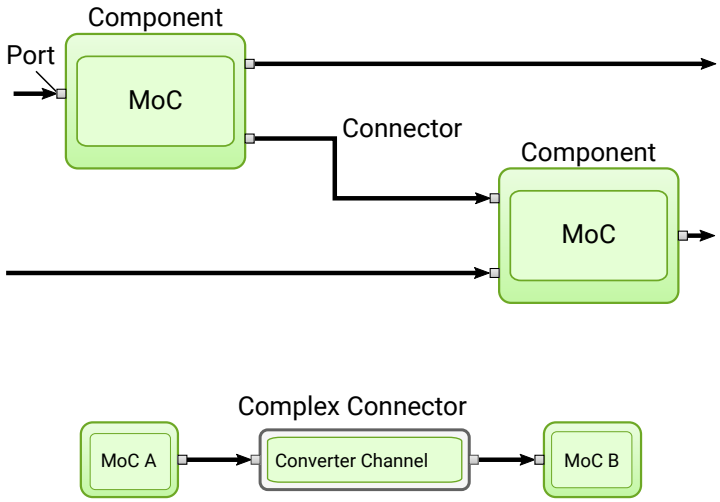


Figure 5.2.: Relation between Components and MoCs

MoCs of the same type can be connected through ports and connectors of the component model, i.e., they can directly interact with each other without introducing consistency issues. When two MoCs of different types should be connected, a complex connector is required. This complex connector is

realized by a Converter Channel. A Converter Channel is responsible for (1) data type conversion (e.g., float to fixed-point conversion), and (2) time conversion. This will be detailed in Section 5.6.

The approach also allows for hierarchical models where the individual sub-models exploit different MoCs, as indicated in Figure 5.3. However, with respect to the Component-based Design paradigm, hierarchical components must conform to component composition (cf. Section 4.1.2). This is exemplified with the state-based model in Figure 5.3, where the individual states contain models of a different MoC. Hence, hierarchical models can be exploited only in the case where not only whole models are mapped to the component framework, but a more fine grained mapping of individual model entities exists.

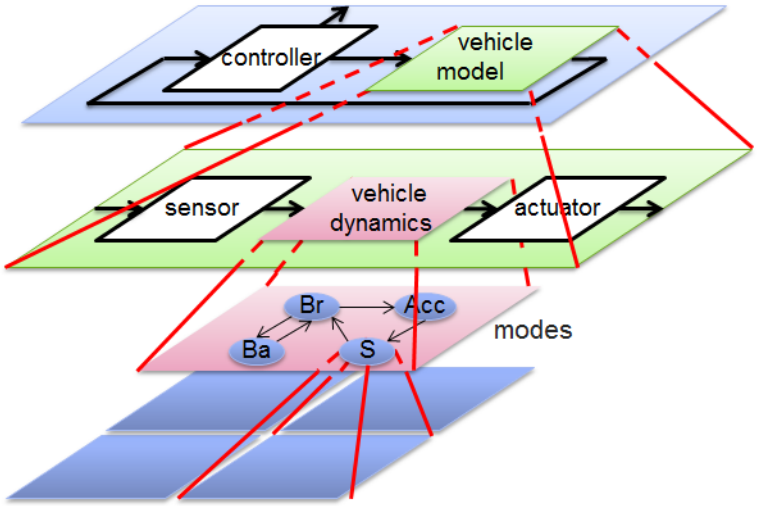


Figure 5.3.: Heterogenuous Hierarchical MoCs

With this, MoCs may serve as structure-providing elements as well as leaf blocks for modeling the actual functionality. This also includes arbitrary programming languages, although their usage should be constrained in order to avoid unwanted side-effects such as traps, interrupts, infinite loops, etc. Moreover, we require compatibility to the enclosing MoC. These things will typically be ensured in terms of programming rules and suitable application programming interfaces (API).

The next two sections give an overview on relevant MoC for ADAS/AD development. We explicitly consider different development phases where the individual MoCs are typically used. For functional modeling, we will find Models of Computation suitable to design systems without focusing on the technical aspects. Here indeed effects such as control discretization might be of interest. On the other hand, issues like OS-specific program execution and interferences due to concurrent processing are of no particular concern. Though, during realization towards the technical architecture, one will be faced with different models of computation, such as multitasking process models. The question how theses models are related has been discussed in Chapter 4 in general. How this could be instantiated in concrete design processes is a matter of Chapter 6.

5.4. Modelling The Functional Perspective

Models of Computation for the Functional Perspective focus on the specification of functional dependencies (i.e., execution ordering) and thus potential parallelism. These MoCs do not explicitly

specify time, but execution rates or ratios can be provided. In the Functional Perspective, different functional domains of an ADAS/AD can be expressed by their supporting MoCs.

5.4.1. Closed-Loop Control

Closed-Loop or Feedback Control systems can be described as **Synchronous Data Flow** (SDF) ([55], see Section B.2) or **Synchronous Reactive** Systems. These systems can be described as untimed SDF or (discrete) **Timed Data Flow** (TDF) [70]. Synchronous reactive formulations can be either in continuous time (differential equation) or in discrete time.

Untimed SDF can be transformed into TDF by assigning timing properties (i.e., execution times) to the actors (computational nodes/containers in a data flow graph). All data flow graph formulations permit a deadlock-free static execution sequence, that can be repeated periodically (the repeated schedule is also called repetition vector). Based on the static execution sequence (also called static schedule), the sizes of the FIFO buffers between actors is bounded. A synchronous reactive continuous time formulation (i.e., system of differential equations) can be expressed as an SDF that evaluates or numerically solves a differential equation during actor activation.

5.4.2. Data Aggregation and Evaluation

Data aggregation and data evaluation (e.g., as part of sensor data fusion) can be described as static or dynamic data flow. This can either be described by a **Kahn Process Network** (KPN) (see Section B.1) that consists of Actors implementing the data processing and edges that represent FIFO buffers, or a more constrained **Synchronous Data Flow** (SDF) graph. A KPN formulation allowed more flexibility, as it allows dynamic data production and consumption rates. This property requires a dynamic schedule and dynamic deadlock resolution. No memory consumption upper bound for arbitrary KPNs can be guaranteed. An SDF restricts the data production and consumption rates as constants per edge. This allows static scheduling and upper bounded memory consumption. More flexible dynamic data rates can be expressed by **Scenario Aware Synchronous Data Flow** (SASDF) graphs that combine a scenario finite-state machine with an SDF. The state machine represents different data production and consumption rates of the SDF.

5.4.3. State Representation and State-based Decisions/Open-Loop Controllers

A **Finite-State Machine** (FSM) (see Section B.3) is the most popular model for the description of State-based decision or open-loop controllers. An FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with these states or transitions.

An extension to eliminate the problem of the state and arc explosion is the introduction of concurrency and hierarchy. This model is called **Hierarchical Concurrent Finite-State Machine** (HCFSM) [33] and is implemented in the widely used StateCharts [38, 10]. States of a HCFSM can be decomposed into an FSM. Hierarchy, also known as OR-decomposition, can be decomposed into a flat FSM with the same number of states but more transitions. Concurrent states, also known as AND-composition, can be decomposed into a sequential FSM with more states and more transitions (cross product automaton construction). **Finite-State Machines with Datapath** (FSMD) combine the features of the FSM and the DFG models. Merging the FSMD model with the concept of programming languages (see next section) leads to the so-called **Superstate FSMD** (SFSMD). Such a superstate can be specified by constructs of programming languages. Replacing the FSM model in HCFSMs by a SFSMD model leads to a HCSFSMD, or much shorter **Program-State Machine** (PSM) [29].

A PSM consists of a hierarchy of program-states with each of them specifying a single mode of computation. At each point in time only a subset of program-states is active, and thus perform their computations. A composite program-state can be decomposed into either sequential or concurrent program-substates.

5.4.4. Complex Computation

"Classical" programming languages will remain an important development tool in the foreseeable future. At the technical architecture level, the vast amount of operating systems is realized with programming languages like C and C++. Also application programming, though increasingly supported by automatic code generation, remains a domain of programming languages, such as for high performance tasks like image processing as well as legacy support.

Drawing a landscape in the context of future ADAS/AD development was subject of a predecessor study [60], where a number of existing languages has been analyzed and compared based on their features, and some new traits not available in the current wide-spread programming languages are proposed.

As a complementary comment to the features considered in the study, we like to stress the need for language interfaces that provide for the integration of MoCs with different interaction semantics. This will be further elaborated in Section 5.6.

5.5. Modelling The Technical Perspective

Models of Computations for the technical perspective focus on the execution platform and thus are particularly designed for modeling program executions. This is the domain of MoCs like process networks and time multitasking. Many different MoCs have been developed also in the real-time scheduling domain, e.g., [35, 20]. The Giotto framework [41] provides a MoC conforming to the Ptolemy framework [25], which is based on a strict time triggered paradigm.

While such academic approaches provide for clear unambiguous semantics, we strongly believe that the technical perspective should exploit modeling languages that are well-established in the industry or are developed with strong industrial emphasis. The AUTOSAR standard defines two abstraction levels for the technical perspective (the SW Component Description and the System/ECU Configuration Description), which reflect an agreed design methodology. Particularly the SW Component Description defines a number of different interaction paradigms for the communication between individual software component, which are widely consistent with the interaction paradigms discussed in the following section. These interaction paradigms are realized in the System/ECU Configuration Description level in terms of configuration parameters such as, e.g., for runnables and the communication stack, providing for (at least basic) traceability of timing specifications in the functional perspective.

The specification language developed in the AMALTHEA4public project aims at a clean methodological approach for system design process. Models are divided into individual parts that are instantiated in different design steps. The specification is largely consistent with the AUTOSAR standard, but defines additional concepts (e.g., probabilistic timing specifications and memory models) that allow for more concise and refined models, and also raises expressiveness.

In the context of this report, both AUTOSAR and AMALTHEA4public still lack expressiveness. Data dependent executions, modes of operations and causal event correlations come into mind. For some of these concepts potential extensions for the AUTOSAR standard have already been discussed.

5.6. Heterogeneous MoC Integration and Interaction

As indicated in Section 5.3, the role of MoCs is to provide semantic foundation for modeling and programming languages that allow engineers to "implement" component behavior. As the individual MoC are defined on different time domains, this may cause integration issues not only if components with MoCs of different time domains are connected, but also if timing specifications are used for components with time domains other than the common time domain used for such specifications (as introduced in Chapter 3).

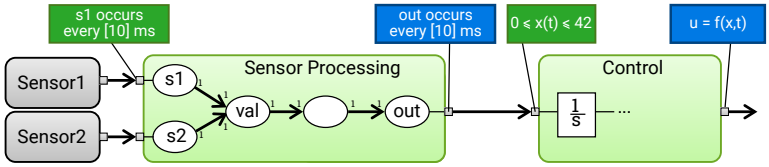


Figure 5.4.: Simple Interaction Example

A simple example is shown in Figure 5.4, where a sensor processing chain is modeled as a synchronous dataflow graph (SDFG) that feeds a continuous time control algorithm. While SDFG are inherently untimed, timed semantics is "attached" to the Sensor Processing component by specifying event occurrence patterns (cf. Section 3.2.1) to the respective input and output ports of the component. While being untimed, SDFG have well defined execution semantics as discussed for example in [55], which is based on finding execution orders that satisfy balance equations for token production vs. consumption.

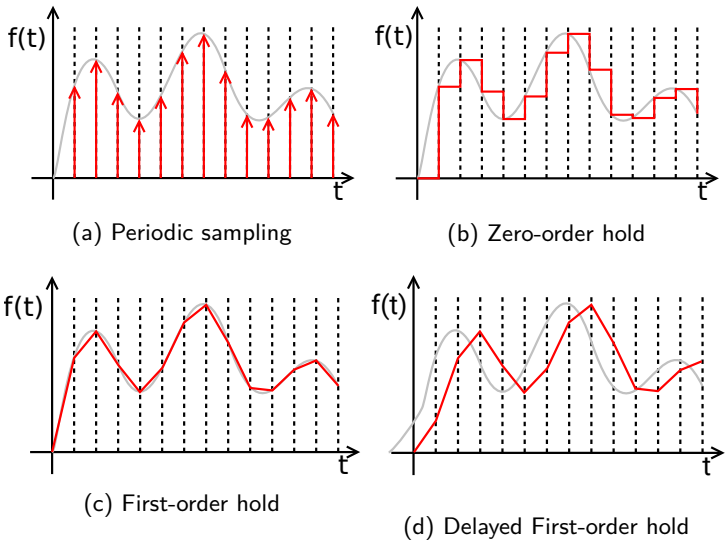
from \ to	**CT**	**DT**	**ST**	**UT**
CT	id	periodic sampling	periodic sampling	transform to DT, ST
DT	n-th-order, hold	id	periodic sampling	transform to DT, ST
ST	n-th-order hold	id	id	transform to DT, ST
UT	transform to DT, ST	transform to DT, ST	transform to DT, ST	id

Table 5.1.: MoC Converter Channel Time Conversion (CT = Continuous Time, DT = Discrete Time, ST = Synchronous Time, UT = Un-Timed, id = identity, no time conversion necessary)

According to the Contract-based Design paradigm, the question whether or not the model complies to the (timing) specification is a distinct verification step (cf. Section 4.3.3). During the design process, the actors of the SDFG model will be implemented in some programming language (either manually or by code generation). The concrete implementation will be executed on a hardware/software architecture where processing takes time. Hence the satisfaction verification could be delayed until such concrete technical implementation exists. For early validation and verification, however, the untimed semantics of SFDG must be cast into the timed domain. This is achieved by *refining* the model in which it is translated into another time domain (cf. [70, 32]). Formal background of such

translations is introduced in Appendix A. More practically, e.g., for simulation purposes, one would select an appropriate simulation kernel that provides a suitable scheduling scheme conforming to the balance equations, and provide additional timing annotations to the actors.

The figure also exemplifies integration issues that may occur due to the connection of components with MoCs of different time domains. In this concrete example, the output port of the Sensor Processing is of type discrete-event signal, while the input port of the Control is a continuously evolving signal. There are well-established concepts in order to cope with such situations, as shown in Table 5.1. For conversion from continuous to discrete event signals, periodic sampling (see Figure 5.5a) is often used. The opposite direction is often tackled with an n^{th}-order hold approach. Figure 5.5b show the example of a zero-order hold conversion, while Figure 5.5c shows the example of a first-order hold, i.e., piecewise linear approximation. For the connection of timed (CT, DT and ST) to un-timed (UT) models, the UT models need to be transformed or integrated into DT or ST models first.



(a) Periodic sampling (b) Zero-order hold

(c) First-order hold (d) Delayed First-order hold

Figure 5.5.: Examples for CT $\leftrightarrow$ DT/ST time conversion

We propose to explicate necessary type conversions conforming to the Component-based Design paradigm by the introduction of *Converter Channels*. Conceptually, Converter Channels are themselves components that provide for potentially complex interaction semantics between components. This approach fits well into the design process, as it allows for (complex) allocation patterns where Converter Channels can be realized also on distributed target architectures including, e.g., communication stacks in without loosing traceability of given (timing) requirements. The basic concept is depicted in Figure 5.6, where a Converter Channel explicates an n^{th}-order hold conversion between the two components.

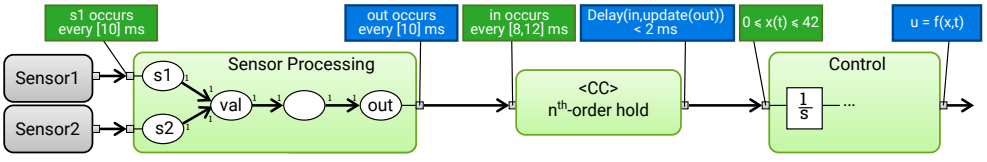


Figure 5.6.: Simple Interaction Example with Converter Channel

The concept could be instantiated in concrete design flows by providing libraries of parametrized component patterns for Converter Channels, which can be instantiated for the individual use-case (as it is standard in Matlab/Simulink). The pattern would include a set of also parametrized contracts providing basic specifications such as sampling occurrences and delays. Parametrized timing specifications would greatly increase the flexibility of the concept, as this would allow to define the range of contexts in which a Converter Channel could be used. In our example, the Converter Channel accepts inputs in period range between 8 to $12ms$, and thus is not restricted to strictly periodic sampling. If this is not possible or desired, one would define specification parameters that do not allow to specify such period ranges.

The concept of Converter Channels can be further extended to other types of interaction components, such as FIFO and stack buffers, shared variables, and even more complex ones, which are particularly useful in the context of multi-rate systems. Figure 5.7 depicts a system where a shared variable (SV) is used to provide interaction semantics for under- and over-sampling, respectively. Note that the specification of SV is over-simplified as it does not specify what happens if the consumer reads data before the producer has written data (and whether this is allowed). A more elaborated example is shown in Section 3.2.5, which uses mode dependent specifications useful also to specify, e.g., FIFO buffers. Again, also for interaction components the use of libraries of parametrized component patterns would be highly recommended in order to ease the design process.

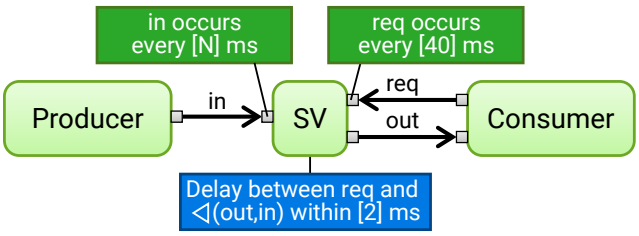


Figure 5.7.: Interaction Component Examples

Explicating (complex) component interaction in the design has large potential to increase observance of potential integration issues. For example, side effects due to component interaction get higher visibility, and explicit interaction components enable enforcing execution rules (such as sampling periods). Using explicit interaction components also improves traceability during the design process. As indicated earlier, realization of component interaction may involve complex communication paths on the technical architecture, which becomes visible by using interaction components that are decomposed and allocated along the design process, and for which corresponding V&V activities allow for satisfaction of the initial specifications.

Implementing Converter Channels Interaction components have been used in various contexts. In the signal processing domain, under- and over-sampling are well-known concepts. As we have seen in Chapter 3, the real-time domain provides respective specification patterns allowing to reason about under- and over-sampling effects. Matlab/Simulink provides various modeling concepts for interaction specifications, such as Rate-Transition- and delays-blocks as well as nth-order hold blocks. For the technical perspective, AUTOSAR defines different types of ports for software components, which are consistent with the concept of Converter Channels, though realizing only a sub set.

Figure 5.8 depicts an example where conversion is performed in a Matlab/Simulink model in order to provide a consistent interface with a surrounding discrete-event MoC. The same functionality is obtained with an explicit conversion channel as shown at the bottom part of the figure. Beside

having the benefit of making the conversion explicit, the approach also increases re-usability, as the Matlab/Simulink function block can be exploited in different contexts.

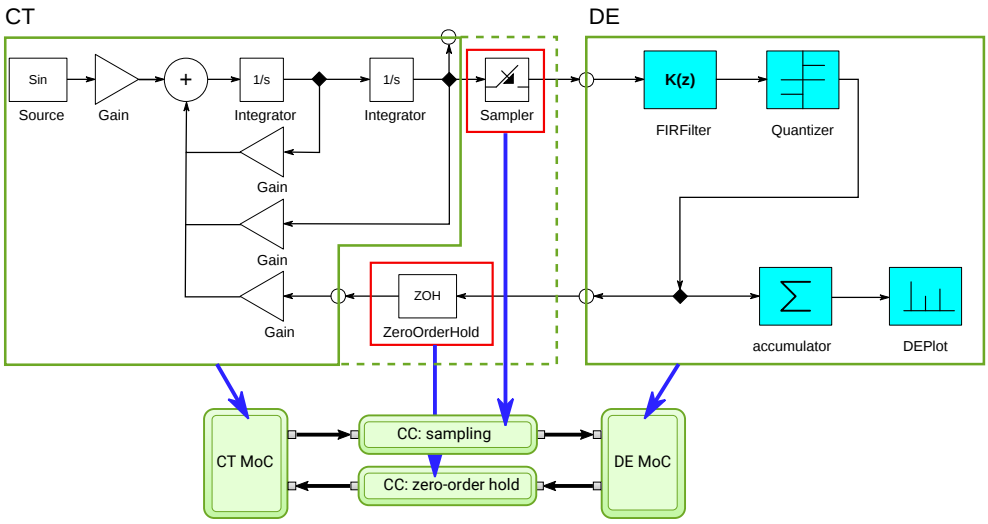


Figure 5.8.: Interaction Example in Simulink

Converter Channels can be implemented in the same way as other components. This includes, indeed, the use of Matlab/Simulink as in the figure above, or arbitrary programming languages. Also here, programming of Converter Channels and other interaction components should be complemented by suitable libraries.

It is worth to note that co-simulation is another domain where component interaction is of key interest. Frameworks like the Functional Mock-up Interface (FMI), and here particularly the project ACOSAR[1], investigate on solutions for coupling and integration of real-time systems into simulation environments. The work in [17] puts this effort into the context of signal semantics, which is used in this report as a formal foundation.

[1] http://acosar.eu/

6. Design Paradigms for Time Coherency

This chapter gives an overview on how the concepts introduced in the previous chapters can be instantiated and applied in the context of ADAS/AD design. The basic idea for the combination of these concepts is simple, and consists of two main parts. First, it exploits a component model that provides all ingredients needed to characterize system designs in a concise and unambiguous way. Together with an also concise and unambiguous concept of (timing) specifications as an instance of Contract-based Design, it provides means for performing development processes in a continuous and traceable way. As already noted, this does not mean to define a particular development process but a set of common design steps from which the individual development process is assembled, and proof obligations for V&V activities to support these design steps. Secondly, it is based on the integration of heterogeneous MoCs, which are used to model and implement the individual components of a system.

6.1. Component Model and Models of Computation

The main concepts of the heterogeneous system design paradigm are depicted in Figure 6.1. The lower part shows the component model of the intended system, which is assumed to conform to the semantic framework discussed in Chapter 4. Component modeling is typically based on some framework providing a formal notion of the artifacts and their relations in terms of a meta-model. A number of frameworks exist such as EAST-ADL[1], MARTE[2], SysML[3], and those defined in projects like MBAT and CRYSTAL. These component frameworks typically provide UML profiles, and there are tools available that can be used for developing such component models (e.g., IBM Rhapsody, Enterprise Architect, Eclipse Papyrus). It should be noted that many existing frameworks focus on particular modeling perspectives, such as MARTE and AUTOSAR on the technical perspective, which would be an obstacle with respect to continuous design processes. A sufficient component model must support the design of all relevant perspectives.

Concerning (timing) specifications, two requirements can be identified for the envisioned component model. First, it must enable defining component interfaces that conform to the semantic definitions discussed in this report. This includes support for the relevant data types for ports, and for the relevant interaction semantics such as discrete-event and discrete as well as continuous evolutions. Second, the framework must either allow for the definition of timing specifications in terms of contracts or for linking specifications that are specified using specialized external tools (e.g., IBM DOORS and BTC EmbeddedSpecifier).

Figure 6.1 indicates that models will typically contain hierarchies, where components are decomposed into (interacting) sub-components during the design process. Given that systems are comprehensively specified, this already gives means for V&V activities that allow to check whether the specification of a component is satisfied by its sub-components and their interaction. This is a huge advantage in the design process, considering the fact that this step does not require any knowledge about concrete component realizations. Verification tools "only" need to understand the specification languages used for expressing the involved contracts, and the same tools can be used in all design phases

[1] http://www.east-adl.info
[2] http://www.omg.org/spec/MARTE
[3] http://sysml.org

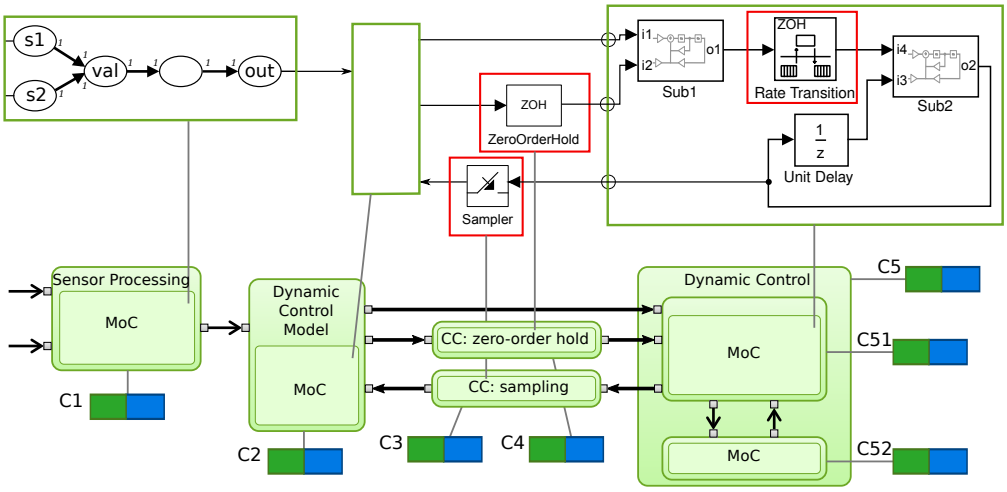


Figure 6.1.: MoC Integration and Timing Specifications in the Component Model

and independently from the concrete design perspective in order to reason about decompositions of components by means of their contract based specifications. The underlying concepts have been discussed and exemplified in Section 4.3. Preliminary support for this kind of verification has been developed, e.g., in [31].

As discussed in Chapter 5, components with inconsistent interface semantics interact via Converter Channels. Though not required, it would be highly desirable to have libraries of component templates at hand that allow for easy instantiation of such components in the modeling tools. It would also be desirable if these templates would come with pre-defined timing specifications that could be tailored by parameters (such as delay bounds). Further note that inconsistencies between interfaces can also be detected by respective V&V tools as indicated in Section 4.3.

The last piece of the paradigm concerns the design of individual models by specifying internal behavior of the components in terms of MoCs. Various MoCs have been discussed, ranging from those exploiting continuous time to untimed models like finite state machines. For each of these MoCs implementations exist that are supported by tools employing particular languages. Examples for such tools are Uppaal, which models systems in terms of timed automata, and the ubiquitous Matlab/Simulink to only mention a few. Also programming languages such as C and C++, among other, fall into this category. It should be noted that the distinction between application components and Converter Channels (and other interaction components) is only a methodological one. Both types of components are modeled and programmed, respectively, using the same concepts.

With respect to the design paradigm it is key that modeling and programming of the individual components is compliant with the defined component interfaces. This includes (1) the involved data types, and (2) the time domains. For the top-right component in Figure 6.1 for example, we expect that the interfaces of the component specify continuous time (evolution) ports, as this conforms to the semantics of the Matlab/Simulink model that implements the component.

Selection and matching of interfaces is sometimes a complex design decision, and depends on various factors. While the underlying MoC of the individual programming language specifies a certain time domain, there is still room for interpretations. In the example above, one also could argue that the final implementation of the Matlab/Simulink model exploits a semantics where continuous blocks are translated into a discrete-event semantics with reasonable small sampling periods. Here, a continuous time semantics of the corresponding component ports would not be the correct choice. So, interface

design depends on the concrete use-case and design phase. It also depends on the tools supposed for verification. For this purpose, the respective models are often implicitly transformed into another time domain. This has to be taken into account, i.e., for checking the satisfaction of component contracts by simulation, where some kind of models (like SDFG) are augmented with timing characteristics as a preparation step for the simulation run. The design paradigm also supports these design steps by providing the foundation to specify mappings between the individual models, which leads to a more formal notion of traceability. That is, it formally defines V&V activities in terms of proof obligations allowing to check whether requirements are still satisfied by the transformed model.

6.2. Abstraction Levels and Perspectives

While most examples given in this report so far concentrate on the functional design, the paradigm applies to all design phases and perspectives. Which perspectives are used in the individual design process depends on the application domain and particular use case. In the avionic domain often functional, logical and technical architectures are modeled, where the logical perspective is used to group functions that relate to the same (logical) subsystem and component, respectively. These perspectives are refined in several abstraction levels, where the refinement often goes along organizational boundaries.

Concerning automotive ADAS/AD design, we do not restrict nor propose particular perspectives and abstraction levels to be used. The functional perspective though fits well with existing standards such as ISO26262, where the functional safety architecture is an outcome of the initial safety process. Also the AUTOSAR standard fits well with the notion of perspectives, in that it proposes a Virtual Functional Bus (VFB) view that focuses on an architecture in terms of software components (SWCs) which is then refined towards a view where SWCs are allocated to a network of ECUs and the basic software of each ECU is configured.

We propose that the particularities of each MoC with regard to data and control flow are explicated by means of specifications, as well as the relationship of their model of time with a metric time base (cf. Section 5.6). As discussed in Section 6.1, this enables reasoning about composition of different MoCs by means of Converter Channels connecting them, which in turn implement the communications between the MoCs and synchronize computations. The same approach also applies when relating different perspectives of the same system. Here we focus on the relation between a functional design and its technical realization on a given target platform.

6.2.1. From the Functional to the Technical Perspective

As an example consider a functional design modeled in Matlab/Simulink as shown in the top of Figure 6.2. When considering the technical realization of this functional design by an AUTOSAR ECU configuration, it is critical to ensure that this realization indeed preserves the semantics of the MoC, which involves the timing aspect. As pointed out in Section 5.1, a MoC defines a notion of time, semantics of communication and synchronization between processes and how computations take place. For example Matlab/Simulink is basically a data-flow synchronous language, similarly to LUSTRE [34] and SIGNAL [54] (at least if a fixed-step discrete solver is used, which is mandatory for code generation). As the name suggests, the primary focus of such languages is on data-flow. Computations are assumed to be executed in a synchronous manner, meaning the behavior is described by computations executed during a synchronous step. This ensures functional determinism, contrary to asynchronous MoCs where computations are interleaved. The *synchronous hypothesis* [11, 71] allows for a conceptual abstraction from the actual duration of computations on a metric time base. So using that abstraction, the reactions of a system at each step are instantaneously. Of course this is not true in practice. Thus, the approach is to ensure that computations are fast enough to be completed within one step.

This can be compared to clock-synchronous circuits modeled in a hardware description language like Verilog/VHDL, where the delay of the clocked processes must not exceed the clock period (critical path analysis). In these languages time is also abstract and related to metric time by introducing clock signals that change their value following some periodic pattern. The advantage of synchronous languages in terms of their functional determinism should of course not be lost when being realized on a technical platform. To ensure this, it must be checked whether the synchronous hypothesis still holds for the technical realization. Otherwise, the functional behavior observed during simulations, e.g., in Matlab/Simulink, might be different from the implementation. Note that in some cases, exactly preserving the simulation semantics is not necessary. Small discrepancies between semantics and implementation can sometimes be tolerated, for example when designing robust controllers. On the other hand, as pointed out in [19], controllers also contain more and more *discrete logic*, which is not robust (a single bit-flip may change the course of an if-then-else statement). Preserving the deterministic functional behavior is then important. In any case, the necessary consistency conditions with regard to the functional behavior should be made explicit in specifications, be it the preservation of the synchronous behavior or a relaxation requiring values to be as *fresh* as possible. The simplest strategy for a technical realization of a synchronous program is to translate it to sequential single loop code, with a single task executing this code. Each execution of the single loop corresponds to the computations of one synchronous step, updating the state of the synchronous program and computing output values depending on the current input and state [11]. However, this fully sequential computation is suboptimal and ignores possibilities for parallelization of computations.

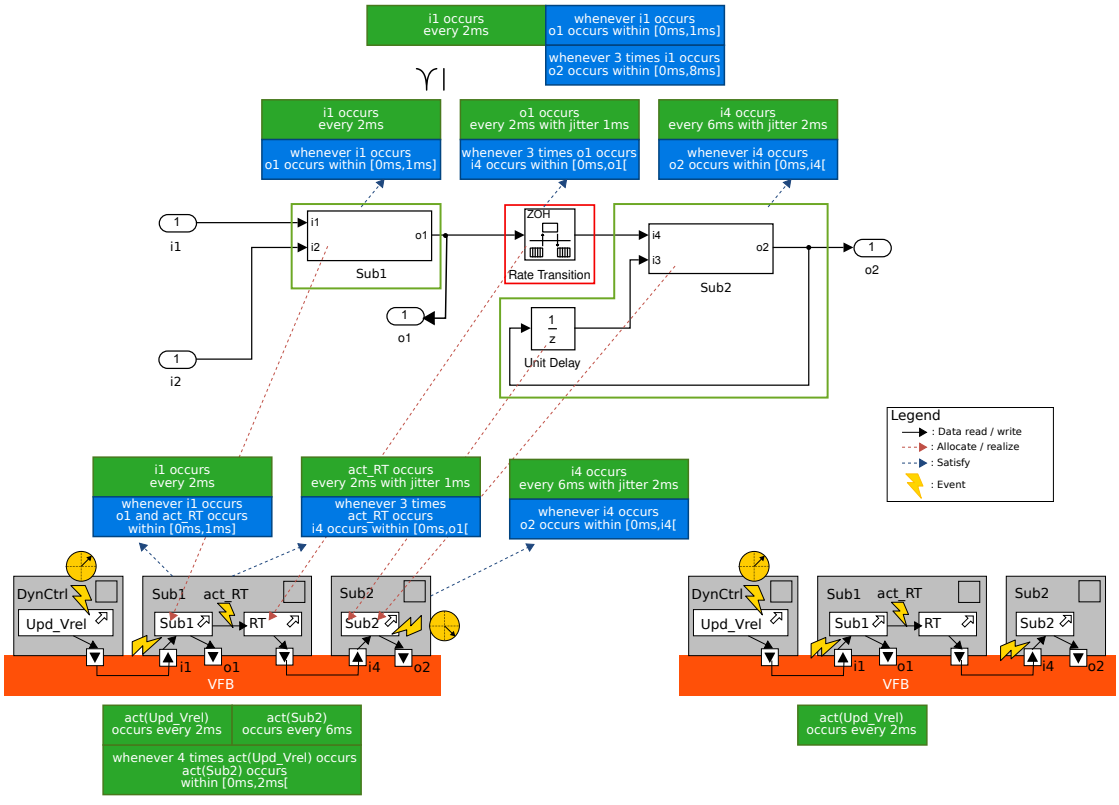


Figure 6.2.: Moving from Matlab/Simulink functional perspective to AUTOSAR logical VFB architecture

Consider again the simple Matlab/Simulink model shown in Figure 6.2. The model contains two subsystems with different sample rates. Sub1 is executed and samples its inputs with a rate of 2. Sub2 is executed and samples its inputs with a rate of 6. So we have a multi-rate system. Since Sub1 produces data that is consumed by Sub2, the model must express how the data transfer is supposed to be done. For each execution step of Sub2, Sub1 is executed three times. Which of the three values for output o1 computed by Sub1 is supposed to be read and used by Sub2? The usual strategy is to under-sample the data-flow of Sub1 in Sub2, taking one out of three subsequent values and computing the step of Sub2 using that value. To achieve an overall functional determinism, the sample should of course also be drawn in a deterministic manner, meaning that for instance every third value shall be used. This is encoded by the rate transition block between Sub1 and Sub2, which applies a certain strategy for determining which values are sampled depending on whether over- or under-sampling occurs. For under-sampling, a rate transition basically behaves like a zero-order hold block. For over-sampling, it behaves like a unit delay block. For each synchronous execution step m of Sub2, both variants precisely define (in a deterministic way) which is the synchronous step n of Sub1 whose output values are used during the m'th step of Sub2.

When aiming at a technical realization, it must be ensured that this deterministic communication is preserved. As indicated before, a simple approach is to have a single task executing with the base-rate of the model, in this case $2ms$. First the task executes the code of Sub1. Upon each third invocation, the tasks also executes the code of Sub2 directly after having executed Sub1, taking the outputs of Sub1 computed during the same invocation of the task. Obviously, this realizes the same functional determinism as the initial Matlab/Simulink model and achieves consistent rates of execution of Sub1 and Sub2. In that case one needs to verify that the single task executing the code periodically each $2ms$ always finishes its computation before the next execution cycle begins.

On the other hand, it is more likely that the blocks of a Matlab/Simulink model are partitioned and assigned to different tasks, as this allows for parallel execution, making more efficient use of the available processing resources. In this case it is not as obvious as in case of a single task how deterministic data transfer can be ensured. Code generators like the Real-Time Workshop Embedded Coder framework can generate code for single processor systems where deterministic data transfer is guaranteed. Buffers are inserted as necessary and it is assumed that tasks are scheduled according to the rate monotonic principle.

Suppose the two subsystems Sub1 and Sub2 are developed independently, and later on are integrated on the same ECU. In this case, using a code generator on the integrated model might not be an option, e.g., due to intellectual property reasons. Still specifications are available for both subsystems regarding sample rates of inputs and expected latencies like shown in the top of Figure 6.2. The rate transition block plays the role of a *Converter Channel*, as introduced in Section 5.6, bridging Matlab/Simulink models operating at different rates. Its required behavior is specified by a contract as well. Every third output value of Sub1 is passed to Sub2 within a delay such that no further buffering of input values is needed. At the bottom of Figure 6.2 a virtual functional bus (VFB) view of the AUTOSAR model is depicted. According to the concepts introduced in Section 4.3.2, refinement of the contracts associated with the Matlab/Simulink model by the contracts of the VFB model can be checked, which then implies a correct realization from a timing point of view of these contracts by the AUTOSAR model, provided that it satisfies its contracts. The rate transition block is mapped to a dedicated runnable entity called RT that is triggered by the runnable Sub1, which in turn corresponds to the equally named block in the Matlab/Simulink model. At the bottom right of Figure 6.2, an alternative AUTOSAR VFB model is shown, which slightly differs in the interaction of the runnable RT and Sub2. In the first variant Sub2 receives the output value of RT, but is *not* triggered upon arrival of data at input port i4. Instead the activating trigger of Sub2 is a so-called TimingEvent [3, 7.3 RTEEvent]. In the alternative model, Sub2 is triggered upon reception of a data value at the input port i4, which is encoded in the AUTOSAR model by means of a DataReceivedEvent. Both

models could indeed be correct realizations of the functional Matlab/Simulink model, provided that the contracts are satisfied. The second alternative however introduces a precedence between `RT` and `Sub2`, whereas in the first one just a data dependency exists. It is typical that depending on the VFB model some assumptions must be stated with regard to the later ECU configuration phase. At the bottom of Figure 6.2 examples for such assumptions are shown for each VFB model. For the variant depicted on the left side, assumptions are stated about the phase of different timers that cyclically activate runnables `Upd_Vrel` and `Sub2`.

6.2.2. Refining the Technical Perspective

After having mapped the Matlab/Simulink model to an AUTOSAR VFB model, the next step would be to assign the software components to ECUs in a system topology and to define mappings of the runnables of these software components to tasks scheduled by an operating system. In addition, task parameters like priority and preemptability need to be determined. Figure 6.3 illustrates the outcome of this design step for the two alternative VFB models shown in Figure 6.2. On the left hand side the case of a data dependency between `RT` and `Sub2` is shown. In the following we denote this model *alternative 1*. On the right hand side a possible task mapping is sketched for the case of a precedence between `RT` and `Sub2`. In the following we call this model *alternative 2*.

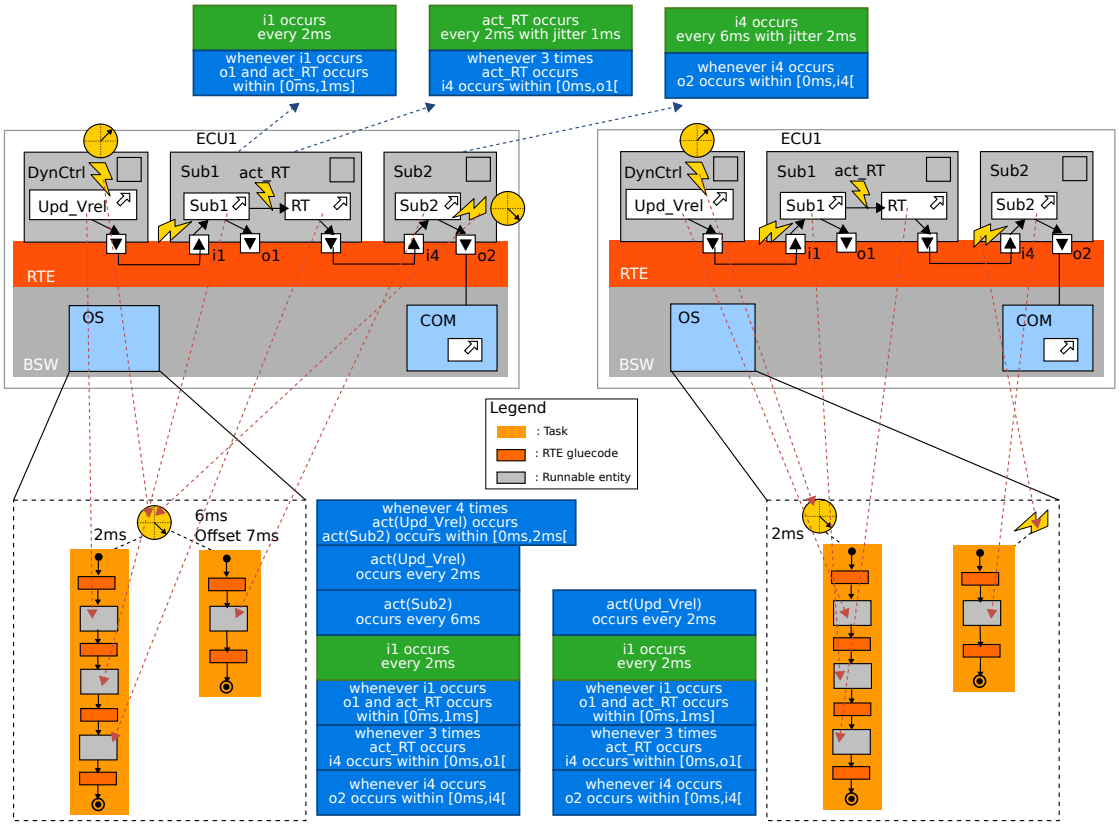


Figure 6.3.: Moving from AUTOSAR logical VFB architecture to an AUTOSAR ECU configuration

Typically, once the step of mapping a functional design to a technical model has been carried out successfully, i.e. refinement of contracts has been checked, the components are implemented. As

already pointed out in Section 4.3.3, in the proposed development paradigm we do not assume a particular form of implementation. An implementation is just considered to be some description of behavior, which - depending on the design process - could be C-code or models with a denotational and/or operational semantics. Once a representation of an implementation is obtained, it needs to be checked whether it satisfies its associated contracts. In this particular case, we consider the AUTOSAR ECU configuration model as an implementation. Assumptions about this ECU configuration made on the VFB level can be discharged at this point. For instance, assumptions about the frequency of timers activating `Upd_Vrel` and `Sub2` are established by a corresponding configuration of software timers. The assumption about their synchronization can be discharged by utilizing the same hardware timer for them and configuring an initial offset for the $6ms$ timer. As we focus on the timing aspect, we abstract from the actual code execution and assume that execution time intervals for runnables are known. Then, proving satisfaction of the remaining contract guarantees shown in Figure 6.3 can be done via a timing/scheduling analysis. For *alternative 1*, such an analysis can be done using classical schedulability and response time analyses (see [76] for an overview of techniques). Note that a synchronization of the time triggers of both tasks is required. Having arbitrary offsets (phasing) between them would violate the right-most contract that requires an output at `o2` to be sent before the next input at `i4` arrives. For *alternative 2*, checking satisfaction of the contracts requires a timing/scheduling analysis that can cope with task precedences. Analytical approaches like SymTA/S [40] or the Real-Time Calculus [84] provide such capabilities by considering offsets. Approaches based on model-checking like [39, 2, 82] also can cope with arbitrary complex precedence constraints among tasks. Of course, the different levels of abstraction from the actual system behavior by the different analysis approaches have an impact on the results. In [69] the influence of such different abstractions on the analysis results is studied. On the one hand, choosing an analysis approach that abstracts from complex interaction behavior will exhibit a good scalability. On the other hand, over-approximation of inferred timing properties might wrongly deem contracts to be violated.

We like to point out that the approach we discussed in this section is just to be understood as an example of how the concepts introduced in Chapter 3, Chapter 4 and Chapter 5 can be used in conjunction in order to achieve a coherent treatment of time along a design process with heterogeneous models, and where corresponding implementations are finally integrated on a common technical platform. So, even though the example just shows how to map a Matlab/Simulink model to an AUTOSAR architecture, the principles of the approach also apply if models conforming to different MoCs are mapped to a technical realization by means of an AUTOSAR architecture.

The characterized design flow involves many complex design decisions and requires much expertise. Some of these tasks may be supported by tools, helping to find suitable solutions in such complex design space, such as discussed in [23] and [81].

7. Outlook

According to the overall approach of MULTIC (cf. Figure 7.1), the first step is to characterize a semantic framework - based on existing approaches - for the development of ADAS/AD applications. This is the content of Part I. The next step is the identification and prototypical development of respective extensions for existing specification, modeling, and programming languages with respect to such framework, and the identification and description of new programming paradigms and languages. This is the content of Part II. The aim of this chapter is to give an outlook to this next step.

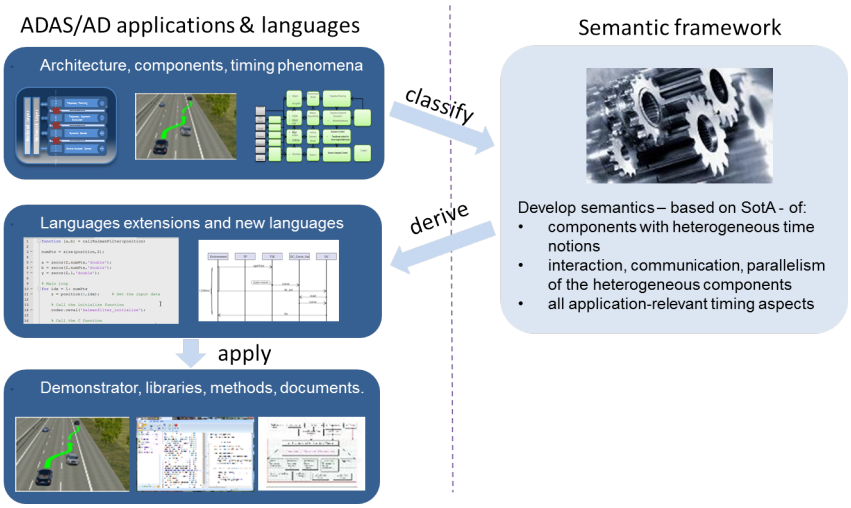


Figure 7.1.: MULTIC approach

More specifically, in Part II, we are going to target:

- A heterogeneous component model with contracts and Converter Channels in SysML (see Section 7.1).
- A textual specification of timing contracts (see Section 7.2).
- The expression of time phenomena in an executable system model in SystemC (see Section 7.3).
- An analysis of C/C++ and Matlab/Simulink regarding their expressiveness to exploit different MoCs and to observe and control time (see Section 7.3).
- An analysis of the described timing phenomena against contracts (see Section 7.4).
- A discussion of the necessary tool support to handle the proposed design paradigm (see Section 7.5).

For the outlook on Part II, we are referring to the example from Section 6, as shown in Figure 6.1. This figure will be used in the following sections to visualize the goals of Part II.

7.1. Integration of Heterogeneous Models

The component framework discussed in Chapter 4 is a key element of the proposed design paradigm. Despite the fact that such component framework should be carefully designed and implemented in terms of a common meta-model, the main "success" factor is not to provide tools that cover all aspects of the component model, but the ability to establish a common view on the system among a diversity of design tools. This holds particularly for heterogeneous systems as considered in this report.

In the scope of Part II we are going to propose the integration of timing contracts and Converter Channels in a standard SysML model.

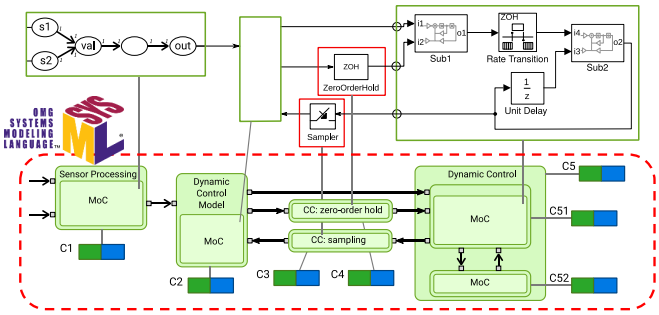


Figure 7.2.: Representative of the SysML component model with contract and Converter Channel extensions (surrounded by the red dashed line).

Figure 7.2 depicts a representative of the SysML component model using standard SysML components, ports and connectors. Converter Channels will be expressed as dedicated SysML components (e.g., as extension through UML stereotype). Contracts will be textually annotated for each component. A link between contracts and ports of a component will by supported by a simple name unification scheme.

7.2. Extension of Timing Specifications

While Chapter 3 discusses relevant concepts for timing specifications, it does not explain how they can be applied in the concrete design process. For the technical perspectives, respective extensions of existing specification languages might be the right way. In other contexts with other MoCs and programming languages involved, pattern or even automata based specifications may be more appropriate.

In Chapter 3 and throughout the running example (see Figure 7.3) of this report, different timing specification patterns have been used. The following group of timing patterns serve as a foundation for the proposed Timing Specification Language to be used in the Timing Contract Extension.

The following paragraphs provide our initially used patterns. They serve as a foundation for the proposed timing specification extensions in Part II.

Event Occurrences Our considered timing specifications rely on the specification of events, or more precisely event occurrences. While for discrete-event signals a direct relation to event occurrences exist, other signal types require a notion of *derived* events. To this end, the pattern `change(.)` defines that an event occurs whenever the value of the signal changes. Note that this does not apply to updates of the signal if no change of the corresponding signal value occurs. More precisely, it captures exactly the time points where piecewise-continuous signals have steps. The derived events

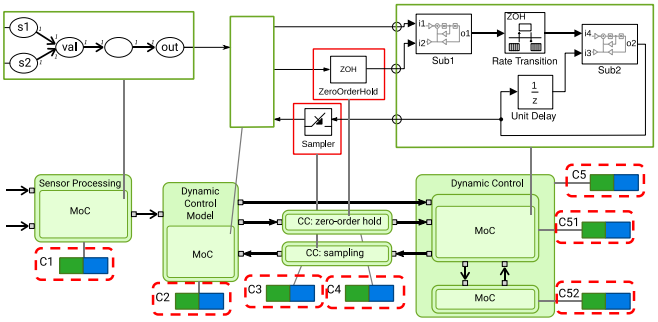


Figure 7.3.: Representative of timing contracts (surrounded by red dashed lines) consisting out of timing assumptions (green) and timing guarantees (blue). Timing guarantees and assumptions as expressed using the same timing specification patterns/language.

`<](<event>)` and so on specify causally related events as discussed earlier. Further types of derived events could be defined (such as value updates). It must, however, be possible to clearly specify the observable behavior of such events.

Additionally, we allow for (ordered) sequences and (unordered) sets of events as event specifications.

```
<event>     ::= <port> ["." <value> ]?
              | "change(" <port> [ <value> "," <value> ]? ")"
              | { "<]","<=],"[>","[=>" }  (" <event> [ "," <port> ]? ")"
<eventlist>  ::= <event> [ "," <event> ]*
<eventset>   ::= "{" <event> [ "," <event> ]* "}"
<eventspec> ::= <eventlist>
              | <eventset>
```

Time Specifications

```
<timespec> ::= <value> <unit> | "[" <value> "," <value> "]" <unit>
             | <probability distribution>
```

Event Occurrence Constraints For recurring event occurrences only a single pattern exist:

```
<eventspec> "occurs every" <timespec> [ "with jitter" <timespec> ]?
```

Event Relations The following defines a very versatile pattern for specifying event relations:

```
<delay> ::= "whenever" <eventspec> "occurs then"
            <eventspec> "occurs" ["within" <timespec>]
            [ "or" <eventspec> occurs ["within" <timespec>] ]*
            [ "exception" <eventspec> occurs "within" <timespec> ]?
<age>   ::= "age(" <event> ["," <port> ] ")" "within" <timespec>
```

Alternatives (`or` and `exception`) are bound by the time interval that results from the union of all specified time intervals in the pattern. For example, the specification

```
whenever request occurs then reply occurs within [40,42] ms
or timeout occurs
```

is violated when an event `timeout` occurs $43ms$ after an event `request` has occurred.

The `or` and the `exception` alternatives differ in that the event specification is allowed to happen only if any of the other alternatives did not happen in the specified time interval. Therefore, time specifications for exceptions are mandatory.

7.3. Extension of MoCs and Programming Languages

To support the implementation phase an analysis of a selected set of programming languages will be performed within Part II. We will focus on the following representative and industrially used modeling and implementation languages:

- Matlab/Simulink: with explicit support for continuous (CT) and synchronous time (ST) models.
- Stateflow: with explicit support for hierarchical and parallel state machine models (UT).
- C/C++: with no explicit MoC support (UT).
- AUTOSAR: with explicit support time and event driven task models (e.g., as indicated in Chapter 3).

The analysis will be performed along the following list of criteria:

- Analysis of the implicit and explicit support for the expression or implementation of
  - different MoCs,
  - Converter Channels,
  - time observability,
  - time manipulation.
- Identification of applicable timing analysis techniques.
- Discussion of the applicability within the running case-study example.

In order to conduct this analysis we are targeting a two-step approach. In a first step, we are identifying relevant time phenomena and will capture the set of most important time phenomena in executable models. The executable model will be structured into modules and ports that are in a structural 1:1 relationship to the corresponding SysML model. The targeted simulation model implements the observable event occurrence without the need of realizing the complete functionality (see Figure 7.4).

For the implementation of the executable time phenomena model, we have chosen SystemC and SystemC-AMS as the most appropriate language, since:

- SystemC is a freely available C++ library that implements a discrete event simulation, capable to express functionality and timing.
- SystemC offers built-in support for Discrete Time (DT), Synchronous Time (ST) and Untimed (UT) models.
- With the SystemC-AMS extension, support for Continuous Time (CT) models is supported.
- SystemC and SystemC-AMS are active IEEE standards (IEEE 1666-2011 and IEEE 1666.1-2016).

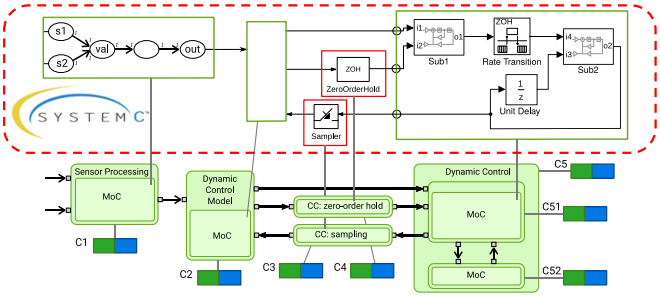


Figure 7.4.: Representative of the functional implementation of the SysML model components in different programming languages and exploiting different MoCs (surrounded by the red dashed line).

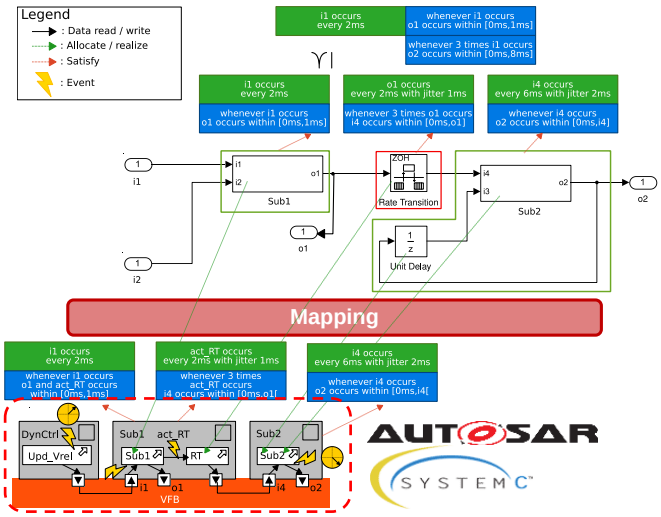


Figure 7.5.: Representative of the mapping of a functional implementation to an AUTOSAR software component architecture (surrounded by the red dashed line).

Due to its flexibility and MoC coverage, all relevant programming languages and their MoCs can be represented in the SystemC-based time phenomena simulator.

To address the mapping of a functional implementation in C/C++, Matlab/Simulink and Stateflow to an AUTOSAR software component model, consisting out of runnable and tasks, selected AUTOSAR computation and communication services will be represented in SystemC as well (see Figure 7.5).

In a second step, after a representation of the chosen timing phenomena in an executable SystemC model, a check against the specified timing contracts can be performed. This will be briefly described in the following section.

And finally, based on this two step analysis, conservative and non-conservative timing extensions of the considered programming languages will be discussed and proposed.

7.4. Timing Analysis

As already described in the previous section, the observable timing phenomena shall be checked against the timing contracts of the SysML component model.

For this purpose, timing contracts will be realized as observers in the SystemC executable model (see Figure 7.6).

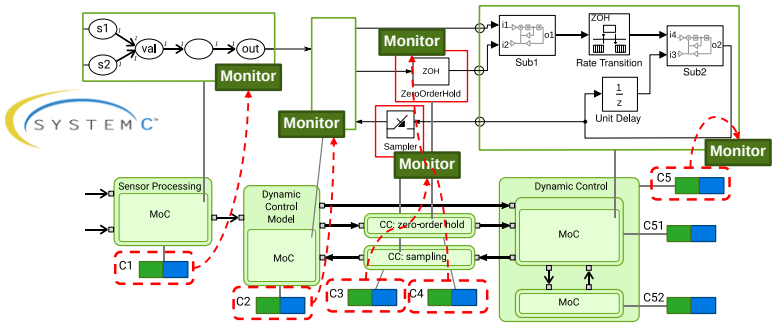


Figure 7.6.: Representative of the contract satisfaction monitoring in the executable SystemC model.

For this purpose, a selected set of contracts will be manually transformed into SystemC timing property observers. These observers act as assertions that stop the simulation and report the timing property violation to the user.

7.5. Tool Support

There is substantial ongoing work on establishing a unified view for heterogeneous design models as targeted in Section 7.1, both in the context of European projects as well as at the industry level (cf. AMALTHEA4public, SystemWeaver, IBM). A widely accepted approach is sketched in Figure 7.7. The unified view (depicted in the center of the figure) is established by an *Interoperability Specification* (IOS) that defines protocols and services allowing to retrieve and modify models in terms of a common meta-model. These protocols and services are implemented by the individual tools, which provide dedicated views to the systems this way (cf. Section 7.5). This indeed requires a well-defined mapping between the model representations that is internally used by the respective tools and the common meta-model.

The IOS must further provide for linking of individual parts of the design model. Timing specifications, for example, that are defined by one tool must be attached to model artifacts created by another tool. Last not least, interoperability also subsumes integration of the individual tools into the desired work flows, which is particularly important in order to design the services provided by the individual interfaces. This requires a careful balance between generality and usability.

An important ingredient for establishing interoperability requires tool support in order to implement protocols and services mentioned above. IOS interfaces can be constructed in two fundamentally different ways. First, they are implemented in the tool itself, requiring effort from the tool vendor. Another way is to create external adapters for mediating the different model semantics. This way has for example been taken with the Lyo adapter[1], which provides for an OSLC-based[2] interface for Matlab/Simulink models.

[1] https://wiki.eclipse.org/Lyo/Simulink
[2] http://open-services.net/

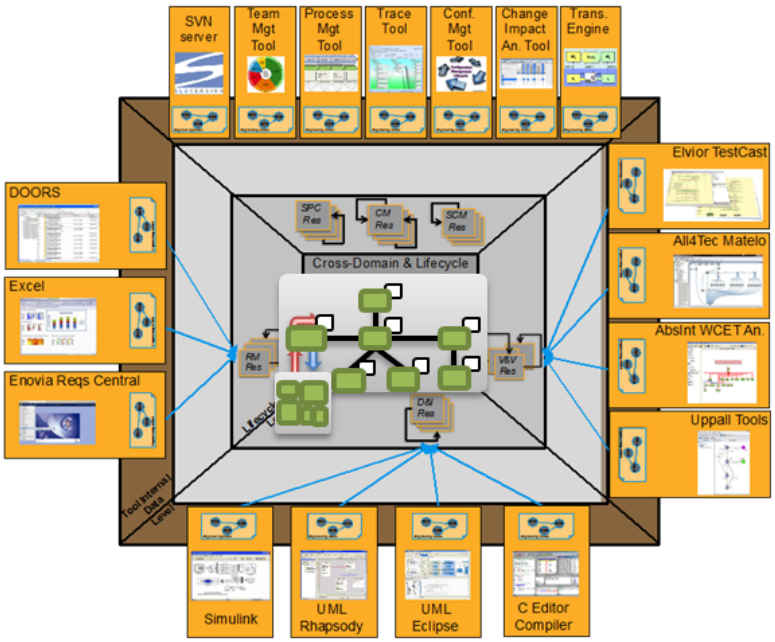


Figure 7.7.: IOS Approach

Part II will provide a brief overview and discussion on these integration issues and potential solutions for the future.

8. Summary

Within this first part of the report, we have laid out the foundations for a coherent treatment of time across different abstraction and development layers. In particular, this involved the following parts:

ADAS specific components In Chapter 2 we exemplified the typical components in an ADAS setting for which a characterization and analysis of time properties is desired.

Relevant timing properties for information provision In Chapter 3 common types of time phenomena arising as a result of the type of information provision over time (continuously, discrete event type, etc.) were collected and explained.

Information exchange over time To ensure to the desired coherent treatment of time, we analyzed how different types of information provision can be combined, i.e., which information is needed for a synchronization between multiple components.

Formalization of timing properties To enable the embedding of these timing properties in a coherent specification and analysis process, we proposed a formalization using contracts.

Within this framework, important concepts are "*Models of Computation*" and "*Converter Channels*", which encapsulate the necessary information with respect to timing properties of individual components and the composition of components respectively. Models of computation on the one hand specify how components provide values over time. The resulting timing scheme critically depends on the language / tool used for the development of the component. The heterogeneity of the tools used hence give rise to different Models of Computation. On the other hand, when connecting multiple components in a larger system (such as an ADAS), the second concept – the Converter Channel – uses this information and formalizes the way how different models of computation can synchronize their information over time thereby defining a model of computation of the combination.

To ensure traceability as well as compositional analysis throughout the development process, we propose the use of (timing-) contracts to specify timing properties. By carefully separating assumptions and guarantees about the timing behavior of the interacting components, contracts are ideally suited to track timing properties within subsystems by assuming a certain guarantee. By assuming a guarantee of a connected component, we can analyze the timing behavior of the overall system while the exact implementation of the connected component is not known and only a particular component is the focus of the development.

Additionally, contracts also provide a guideline for the necessary extensions of currently used tools and languages in order to enable the analysis of time-properties of the overall system. In fact, we need to first annotate the assumptions and guarantees for each software component. Then in a verification step, we need to show that the guarantees are fulfilled by the implemented component under the annotated assumptions. When combining several software components, we then only have to show that the assumptions and guarantees fit together without investigating the detailed behavior of the components anymore while still being able to guarantee the overall timing requirements.

Part II.

Design Approach for Multi-Layer Time Coherency

9. Introduction

Part I proposes and discusses four "building blocks" enabling engineers to consider the timing aspect along the design process of automotive ADAS/AD in a coherent and continuous way. These building blocks, or design paradigms, are depicted in Figure 9.1.

The 1st design paradigm is the **"Compositional Semantic Framework"**, which provides an architectural basis for system design by introducing a generic hierarchical component model. The model is intended to be instantiated with existing modeling languages (such as SysML) and tools by defining how to cast the individual modeling artifacts into artifacts of the conceptual model.

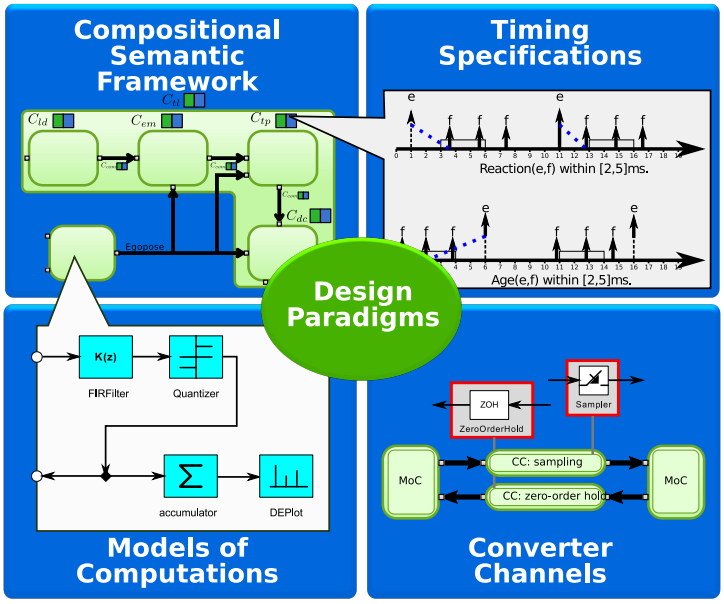


Figure 9.1.: Design Paradigms as introduced in Part I

The component model serves as a carrier for the other three design paradigms:

- It defines a notion of contracts[1] as a particular kind of (assume-guarantee style) specifications.
- It supports the integration of different Models of Computation (MoC) in the leaf component hierarchy.
- And it supports the integration of heterogeneous MoC through Converter Channel (CC).

The Compositional Semantic Framework finally enables to set up design processes where systems are incrementally refined. It provides concepts allowing to relate different viewpoints like functional modeling and the technical realization as well as different abstraction levels.

[1] Contracts give modeling entities and their interaction formal semantics, and enables one to reason about verification of the individual design steps, such as decomposition, refinement and realization.

The 2nd design paradigm **"Timing Specifications"** instantiates Contract-based Design for the timing aspect of the system design. It inherits timing specifications from well established frameworks such as AUTOSAR, and defines extensions where needed in order to enable coherent reasoning about timing within complex scenarios.

The 3rd design paradigm **"Models of Computation" (MoC)** provide the formal basis for implementing components with well-defined execution semantics. MoCs support integration into different time domains (untimed, continuous time, discrete time and synchronous time) and are instantiated by concrete implementation languages such as Matlab/Simulink and C/C++, as well as by domain-specific languages such as various data flow, control flow and automaton-based (i.e., discrete state-machine) languages. Given a particular MoC, its integration into the semantic framework requires two ingredients. First, a mapping of the interfaces of the MoC onto the (conceptual) component model defines how models are embedded into the component model. For example for an automaton-based MoC such mapping defines on which component ports the events of the automaton become visible. Second, a mapping of the potentially different notions of time must be specified. For detailed discussion of the latter we refer to Appendix A.

The 4th design paradigm **"Converter Channels" (CC)** concerns the interaction between components. This is particularly important when components with different MoCs shall interact with each other, such as components implemented in C++ with components implemented using Matlab/Simulink. Part I discusses Converter Channel for "state of practice" interaction semantics as they are used for discrete-continuous signal coupling and vice versa. The present report exploits more complex Converter Channels in order to implement different buffer semantics.

Part II aims at effectively applying these design paradigms in existing design processes. To this end, the report discusses a simplified but realistic design process along a case study in which an ADAS is developed. The case study is deliberately left generic, and does not specify the actual functionality. It has been developed together with domain expert from the FAT AK31 members in order to ensure that it covers relevant issues to be addressed by the application of the design paradigms. It shows where the design paradigms apply in the individual design steps, and particularly focusses on tools and languages that may be used. In doing so, spots are identified where extensions and also disruptive changes may be required when compared to the state of practice. The implementation of a demonstrator which exploits the case study is subject of Part III.

The second part of this document is structured as follows. Chapter 10 provides an overview on the case study. It sketches the overall design flow, and explains which paradigms become relevant in which design phases. Chapter 11 is structured along the design process in the case study. It discusses the individual design phases in detail, as well as the relations between them. The focus lies on practical instantiation of the various building blocks. Language and modeling extensions identified throughout the investigations on the case study are further discussed in Chapter 12. Chapter 13 concludes Part II and summarizes the proposed conservative and disruptive languages extensions and new languages.

10. Overview: Case Study and Design Process

This section exemplifies the application of the four proposed design paradigms along a simplified design process for a generic ADAS case study as depicted in Figure 10.1. The systems shall provide longitudinal (LR) and lateral (QR) actuation control by an automated driving function, depending on the observation of the environment via camera. Perception of the vehicle's environment by a single camera is no realistic scenario, which typically requires the fusion of multiple sensors such as cameras, radar and ultra-sound sensors. However, the system complexity has been carefully selected to show the application of the proposed design paradigms in a non-trivial setting, while being sufficiently simple to keep focus on the design paradigms and not on the system's function.

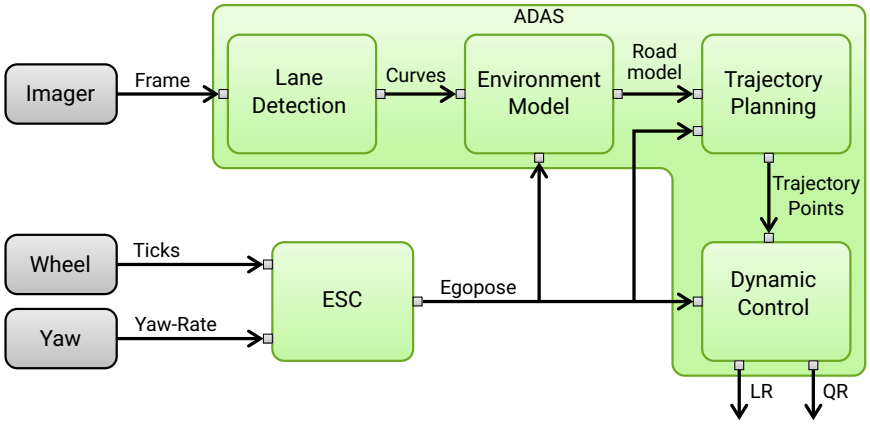


Figure 10.1.: Example: Top-Level Functional Architecture

Figure 10.1 shows that the ADAS function is integrated into an existing baseline system, where the electronic stability control (ESC) subsystem provides additional input for the new function. The ESC function calculates an ego pose (position, yaw and velocity) of the car, which is also needed by the ADAS.

The figure also shows the initial functional decomposition of the system. The "Lane Detection" function extracts curve segments representing the detected track markers from incoming video frames. These curve segments are integrated by the "Environment Model" function into road model that is hence continuously evolving. The "Trajectory Planning" function calculates safe paths along the road model with a certain time horizon, which are finally processed by the "Dynamic Control" function in order to derive corresponding actuator control parameters which allow the vehicle to follow the given paths.

It is important to note that the report does not cover the development of the various functions in detail. In fact, the report even leaves the question open, which kind of automated driving function exactly shall be developed. This might be a lane keeping system, an emergency stopping, or some other advanced driving assistant system. In any case, the development of such system requires expertise

and experience from many domains, such as image recognition, planning and control theory. There is a large body of work on this subject within the literature as well as much knowledge exists in industrial practice. It is stressed again that the report focuses on the general design steps, and how the proposed design paradigms are applied during these steps in order to support engineers to tackle the complexity of the system development. Nonetheless, the chosen example has been developed together with experts in this field, and contains enough details in order to reflect sufficiently realistic application scenarios.

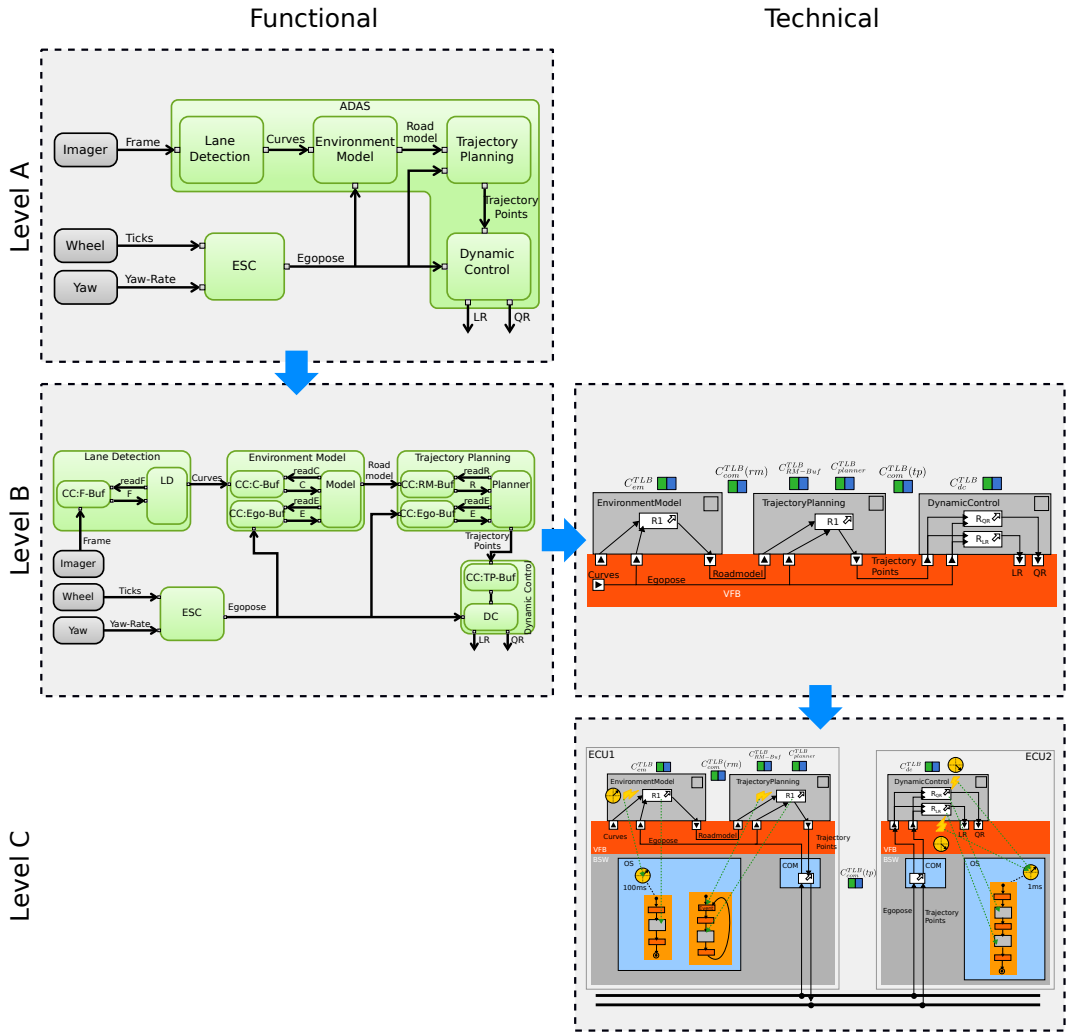


Figure 10.2.: Design Process Overview

Part II exemplifies a design process as depicted in Figure 10.2, which consists of four design phases. The first two phases (Functional Level A and B) cover the functional system design. As the name suggests, these design phases mainly concern system functionality. Hardware/software partitioning is considered only in the sense that general characteristics of sensors are typically known as well as special purpose hardware components for specific tasks, which may influence design decisions. The latter two phases (Technical Level B and C) consider the technical realization of the system. The technical

design exploits AUTOSAR. Although it is expected that future ADAS will involve heterogeneous distributed platforms with a variety of different OS platforms, we argue that a well established standard like AUTOSAR provides a good basis for showing the relation with the conceptual components. The technical architecture distinguishes between hardware and software components. These phases also subsumes, for example, code generation of the functional models as well as the definition of execution semantics of the functional elements (in our case runnables).

It should be stressed that the semantic framework neither restricts nor suggest the design phases as sketched above. The presented design flow has a solely demonstrative character. However, the distinction between functional and technical perspective complies with standard design frameworks such as the envisioned V-model exploited in ISO 26262 for the automotive domain. Also model refinement along different abstraction levels has been proved to be useful, for example to reflect organizational boundaries, for example between OEM and suppliers.

The first design phase (Functional Level A) is discussed in Section 11.1, and deals with the initial functional decomposition of the top-level functional architecture as shown in Figure 10.1. This is often considered as a typical entry point into the functional design, where engineers and other stakeholders agree on the initial architecture, consisting of the top-level functional components and their interaction. This entry point also complies with safety relevant standard approaches as defined, for example in the ISO 26262, where the initial safety design leads to a "functional safety architecture". Furthermore, the result of the decomposition reflects in many cases organizational aspects, where individual components are further designed, e.g., by different departments.

The discussion in Section 11.1 of this first design phase (Functional Level A) concentrates on the two design paradigms "Compositional Semantic Framework" and "Timing Specifications". The application of "Component-based Design" is exemplified by using the SysML modeling language for the design. The functional decomposition is complemented by a discussion of the corresponding decomposition of timing specifications. We start with the initial specifications given for the top-level system and show the application of Contract-based Design methods in order to safeguard the decomposition, which ensures that the top-level specifications are fulfilled by the composition of the specifications of its subsystem components. This is instantiated by a set of well-defined natural language patterns helping engineers to lower the effort for expressing (timing) specifications. The timing specifications represent a conservative extension of existing specification languages, such as TADL [6] and the TIMMO projects [66, 67]. The context in which they are used, which is an instance of Contract-based Design, however represents a non-conservative extension with respect to the state of practice. Section 11.1 hence aims at the graphic introduction of the concepts and, even more important, at the benefits of the approach.

The second design phase (Functional Level B), which is presented in Section 11.2, concentrates on the design paradigms "Models of Computation" and "Converter Channels". At this phase, initial versions of the functional models are developed. Within this report, we omit details on the implementation of the models themselves. Particularly further decomposition and modeling of the individual functions, which may involve different models of computation and languages, would be of interest with respect to the proposed design paradigms. Such decomposition would however call for expertise in the respective functional domains (e.g., trajectory planning or dynamic control), and thus is out of scope of the present document. Moreover, Component-based Design is effective within a hierarchy with arbitrary levels of abstraction (as indicated with the decomposition of the functional architecture in the first phase), and it is considered sufficient to discuss the application of the design paradigms on a rather flat model with its component interaction. Hence, we assume that the identified functional units are already atomic in that they contain a single MoC and are written with a single modeling/programming language. This already enables us to introduce the application of Converter Channels, which have been proposed in Part II to effectively design model interaction.

However, modeling activities will typically involve the refinement of component interfaces, such as the definition of concrete data types for the communicated data. This not only pushes the design towards a technical realization, but also enables early validation of the functional models, for example by setting up co-simulations of the developed models. It is further assumed that the model refinement again goes along the refinement of the corresponding (timing) specifications, which hence is considered in this report. With respect to a continuous and coherent consideration of time along the design process, it is also shown how Contract-based Design can be – effectively – applied in order to verify whether the initial requirements are fulfilled by the refined specifications in the considered functional architecture. While the example is deliberately kept simple in this respect, also more complex scenarios of such "realization" verification are mentioned.

The second (Functional Level B) to third (Technical Level B) design phase transition is the entry into the technical realization. It is also the design phase where the distinction between software and hardware becomes evident. In the present case study, modeling in this design phase is done within the AUTOSAR framework. Although it is expected that future ADAS will be developed for distributed platforms that include other platforms, such as AUTOSAR adaptive, ROS and others, we argue that the underlying design concepts are rather similar with respect to the proposed design paradigms. For example, the AUTOSAR modeling framework introduces concepts on top of the conceptual component model, such as *RTEvents* which are used to model control flow in the system (like the activation of runnables) and various different semantics of software component ports. However, all this is covered by combining the proposed design paradigms as discussed in Section 11.3. This claim also remains true for the final design phase, which concerns the refinement of the technical architecture including the definition of scheduling policies, communication stacks, and other RTE components. Although not explicitly covered in this report, the semantics framework also enables the definition of resource usage, such as the use of a processor core due to code execution, which is consistent to Contract-based Design. Further details about this can be found in [80, 73].

The third design phase (Technical Level B) in our case study focuses on software partitioning, where the functional units are combined into SW components. Along this goes the specification of the interaction of these units, such as whether execution is event triggered or time triggered. The hardware platform is abstracted in this design phase by the VFB layer, except for communication latencies that are relevant for the deployment. However, function partitioning aims at determining which executable units are allocated onto the same processing units. The design phase may or may not include the generation of executable code from the functional models developed earlier. In some scenarios the initial technical design may rather focus on re-partitioning the functional models. For example, engineers may want to restructure their models according to the envisioned software architecture. Whether executable code is generated in this phase or not also depends on the verification methods used to safeguard the design phase. According to the proposed design process, also the technical architecture is equipped with (timing) specifications, and it should be verified that the system fulfills these specifications. Possibilities to discharge this proof obligation are discussed in Section 11.3. In any case, another verification task is to ensure that the specifications of the technical architecture are a realization of the specification made for the functional one. This is also discussed in Section 11.3.

The final design phase (Technical Level C) in the case study aims at refining the technical architecture such that it can serve as the entry point for system integration. This includes the definition of the hardware platform consisting of electronic control units (ECU) and the communication network. Furthermore, the operating systems, the software execution, and communication stacks are defined. This design phase is the latest where code should be generated.

We again focus on the AUTOSAR modeling framework. The same proof obligations as before have to be discharged, including to check whether the refined system is a proper realization of

the abstract technical architecture, and whether the components implementations comply to their local specifications. The verification methods, however, will typically differ from the previous ones. Section 11.4 will discuss different methods such as (classical) scheduling analysis, HIL simulation and testing. Although it is not in the center of the discussion, the section also proposes some basic ideas on runtime observation, which is particularly useful in the context of fail-operational systems. It is one of the benefits of the proposed design approach that observer generation can directly be driven from the timing specifications developed along the design process.

Relation To Robotics Software Systems Design The design paradigms considered in this report are based on long standing research activities, also involving several projects with applications from different domains such as automotive and avionics. As such, particularly the Compositional Semantic Framework is quite generic, which sometimes makes it hard to recognize its applicability in specific application domains. Sketching the relation to the design paradigms for the robotic domain proposed in [58] and [57] might be instructive in order to assess their position in the development process.

The authors of [58, 57] argue that successful and efficient system engineering relies on a clear separation of concerns in order to establish efficient collaboration between the involved stakeholders. Particularly the separation between the actual functional implementations by robotics experts on one hand, and the integration by application experts on the other hand are mentioned. This separation is established in the context of the MULTIC project by the introduction of (1) the compositional framework, and (2) the MoC design paradigm, which reasons about component implementations.

The authors also emphasise that "precise control over the dynamic execution and interaction behavior of functional components, i.e., the computation and communication semantics, on model level without hidden (i.e., code-defined) parts is urgently needed". We strongly agree on that. Part I explicitly states that interaction between components is (and shall be) visible solely on component ports, and no hidden interaction should ever occur due to implementation artifacts.

The authors further argue that while generic models offer concepts for describing the execution and interaction behavior of components on system level, they suffer from a freedom-of-choice philosophy offering all kinds of alternative concepts. They hence define a domain specific language (DSL) in terms of a meta-model for the particular use case of robotic system design. We also agree on that. While the Compositional Semantic Framework is left generic, it is also stated that it should be understood as a conceptual one. It is assumed that engineers will exploit suitable DSLs for the individual design tasks. As long as there is a well-defined mapping to the semantic framework, all proposed design paradigms can be used as discussed in the present part of the report. It should be noted that such mapping is particularly exploited in the later sections where AUTOSAR is used as modeling language.

The remainder of this section considers the relation between the robotics DSL and the design paradigms discussed in the present report. As first observation it can be stated that the component model proposed in Section 4.1 is a core part of the DSL, consisting of components, ports and connections. The DSL further defines a concept of tasks, with the particular emphasis on execution semantics in terms of activation constraints. As it will be discussed later, those constraints are expressed in terms of (timing) specifications as one of the design paradigms. For the technical realization, AUTOSAR concepts will be used to instantiate the constraints. In the robotic DSL, they are modeled in terms of Timers. Similar holds for the specification of CauseEffectChains in the DSL. The Task concept in the robotics DSL would be considered as a particular kind of component.

The robotics DSL emphasizes on a client-server architecture, which is evident due to the special definition of connections. In the present report, this would be modeled as a particular kind of interaction component (Converter Channel). Converter Channels that model client-server communication will be presented later in this document.

To summarize, the proposed design paradigms could be consistently integrated into the models and methods developed for the robotics DSL.

11. Design Process

11.1. Functional Design - Level A

Figure 11.1 depicts the entry point of the system considered in the case study. The ADAS to be developed takes the images from an image sensor, and produces longitudinal and lateral control parameters for the actuation control of the vehicle. It has been identified that the function also requires the egopose of the vehicle as an additional input, which is provided by an already existing ESC component.

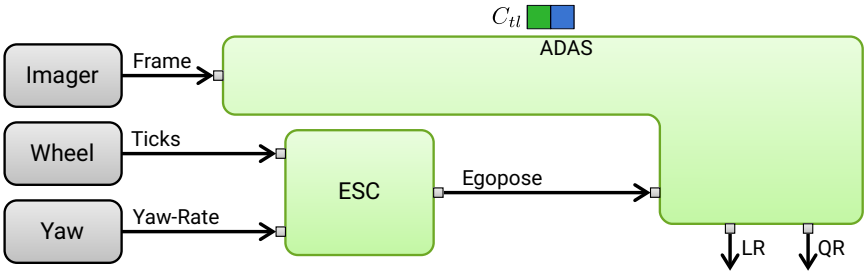


Figure 11.1.: Top-level Functional Architecture

The corresponding timing specification is depicted in Figure 11.2. The assumptions (shown in the upper part of C_{tl}[1] annotated with A) specify the context in which the ADAS function is expected to be executed. The assumption states that the imager provides `Frame` inputs with frequency of $30Hz$ and some jitter $5ms$. That this is sufficient is the result of initial considerations about the needs for observing the environment. With a maximum (relative) velocity of $180km/h$, which is reasonable for driving on an autobahn, objects can move $50m$ per second. Hence, taking snapshots every $33ms$ (possibly further delayed by $5ms$) results in a position delta of $1.9m$ of any object in the environment, including the ego vehicle. As the camera snapshots are expected to display a horizon which is much wider than $1.9m$, this is sufficient to extract a continuous road model. The assumption of getting an update of the `Egopose` every $10ms$ (and jitter $5ms$) is taken from the specification of the ESC system, which is expected as already been defined. Formal definitions of all specifications used in the course of this partial report are given in Appendix D.

Concerning the timing behavior, which has to be guaranteed by the ADAS component, we observe updates of the longitudinal control parameters `LR` every $5ms$, and of the lateral control parameters `QR` every $1ms$. Theses values come from control theoretical considerations as well as experience, stating that vehicle dynamics can be controlled in a comfort zone with such update rates.

The two specifications at the bottom of Figure 11.2 serve two purposes. First, the *Age* pattern specifies that every control parameter update refers to a concrete (and well defined) instance of the `Frame` input of the ADAS function. That means, every `QR` event, and `LR` event respectively, is a reaction to at least one (well defined) `Frame`. A detailed discussion about causal events is given in Section 3.2.4. Second, the maximum latency of the reaction is expected to be related to some

[1] C_{tl} is the top-level (tl) contract.

C_{tl}	A	Frame occurs every $33ms$ with jitter $5ms$. Egopose occurs every $10ms$ with jitter $5ms$.
	G	LR occurs every $5ms$. QR occurs every $1ms$. Age(LR,Frame) within $[0, 800]ms$. Age(QR,Frame) within $[0, 800]ms$.

Figure 11.2.: Top-level Requirement

safety assessment. When the ADAS is developed, the engineers assume that the vehicles environment perception is sufficiently reliable in order to guarantee that no unforeseeable changes will ever happen within a radius of $40m$. Assuming again a maximum speed (difference) of $180km/h$, vehicles can move $40m$ in $800ms$. This is hence the *safety corridor* of the ego vehicle. If anything unforeseen happens within a radius of $40m$, such as the sudden appearance of an obstacle, the vehicle will not be able to react properly. In other words, a reaction in terms of `LR` and `QR` control data should not be later than $800ms$ after the corresponding frame has been received.

It is not claimed that the specification in Figure 11.2 for the ADAS function is complete in any way. For example, the timing relation between the egopose and the actuator control values is not specified, which is omitted here for the sake of brevity. The question of completeness of specifications certainly is of key importance, and a large amount of work, both theoretically and practice oriented, exists in this domain. In industrial practice, (timing) specifications should be defined in a systematic and methodically structured processes in a similar way as for the safety aspect. This is however out of scope of the present report.

The remainder of this design phase concerns the functional decomposition of the top-level architecture. The result is depicted in Figure 11.3. The engineers decide to develop the system in four distinct functional blocks "Lane Detection", "Environment Model", "Trajectory Planning" and "Dynamic Control". The decomposition goes along a refinement of the interfaces. For example, the egopose of the vehicle is needed by all of the latter three functions. Also the interfaces between the internal functions are defined. The "Lane Detection" function extracts lane markings from the video frames received from the imager. The detected lane fragments are sent in form of curves together with the kind of the lane markings (such as shoulder or lane separation). The "Environment Model" function stitches the curves together to form a continuous model of the road ahead of the vehicle, which is relayed to the "Trajectory Planning" component. This function calculates a safe corridor along the road, and provides periodically vectors of trajectory points to the "Dynamic Control" function, which translates them into actuator control parameters suitable to drive the vehicle along the planned trajectory.

In the case study we assume that the model is created with the SysML modeling language.[2] More precisely, the model exploits simple block diagrams and requirements. The functions are defined in terms of the "Functional Block" stereotype. For the specifications, we use simple "Requirement" objects. The specification patterns defined in the course of this document can be written in terms of simple text. The distinction between assumptions and guarantees can be obtained by (textual) convention. We further assume that blocks and requirements are associated via "satisfy" links. Various modeling tools support SysML. Papyrus[3] is a plugin for the open Eclipse framework. Also Enterprise Architect provides for SysML modeling[4]. Using one or the other tool supporting SysML is sufficient to perform the design stage at hand. The data flow may or may not be defined in terms of concrete data

[2] http://www.omgsysml.org
[3] https://eclipse.org/papyrus/
[4] http://www.sparxsystems.de/enterprisearchitecttechnology/mdgtechn-sysml/

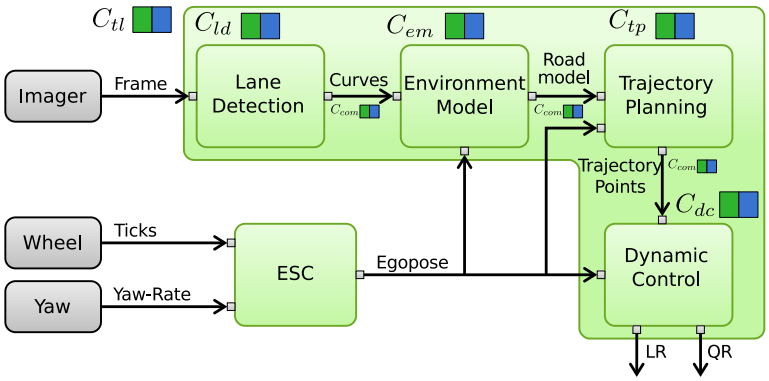


Figure 11.3.: Functional Break Down

types at this design phase, depending on when these design decisions become important. Concrete port times in conjunction with the timing specifications help to reason about the required communication resources, which may be important for the functional decomposition with a particular distributed platform in mind. In our case study this is done later in the second design phase, where also the development of the internal functional behavior takes place.

11.1.1. Lane Detection

The "Lane Detection" function takes the frames from the imager and extracts line markings from the images. Although engineers could argue, that it would be safer from a functional viewpoint to start the extraction only when at least a defined number of valid frames have been seen, from the timing aspect, however, this is of no relevance. Thus the designer specified the timing aspects as depicted in Figure 11.4. The assumption is simply taken from one of the top-level specifications. To properly state the guarantee, the engineers observe that lane detection is a complex task that may require more time than the inter-arrival time between two frames. As this induces an under-sampling situation, it is decided to express the latency by an Age pattern, stating that for every `Curves` data package exists a (causally related) `Frame`, which has been received at most $60ms$ before the output is provided. However, this under-sampling situation must not happen too often. Hence a Reaction pattern is specified, which states that for every received `Frame` indeed `Curves` data is sent no later than $28ms$ at least 3 out of 4 times. We chose $28ms$ instead of the Frame period because of the jitter, which imposes a minimal inter-arrival time between successive frames of $28ms$, ensuring that the function finishes timely.

C_{ld}	A	Frame occurs every $33ms$ with jitter $5ms$.
	G	Age(Curves,Frame) within $[0, 60]ms$. Reaction(Frame,Curves) within $[0, 28]ms$ 3 out of 4 times.

Figure 11.4.: Specification of Component "Lane Detection"

11.1.2. Environment Model

Due to potential under-sampling of the Lane Detection, the assumption of the Trajectory Planning about timing of incoming curves is more complex. Specifying a period interval for the input between

33 and $66ms$, together with an additional jitter of at most $33ms$ is a good approximation of what is seen at this input port. Indeed, this can (and should) be checked (formally), which belongs to the VIT discussed in Section 11.1.5. The assumption on occurrences of the egopose is again directly derived from the top-level specification.

The function will be developed as cyclic execution with a period of $100ms$, as a third of the update rate of the frames is considered to be sufficient. Due to expected execution time variations, the output is assumed to jitter around this period, but not more than $50ms$. The engineers further decided that whenever the function starts execution, it takes the most recent curves for the update of the road model, which arrived before the actual execution of the function. Due to cyclic execution and its jitter, curve updates stay in the buffers for $[0, 150[ms$. Adding the execution cycle of the function, we obtain (by calculation) a maximum age of $250ms$, as specified.

C_{em}	A	Curves occurs every $[33, 66]ms$ with jitter $33ms$. Egopose occurs every $10ms$ with jitter $5ms$.
	G	Age(Roadmodel,Curves) within $[0, 250[ms$. Roadmodel occurs every $100ms$ with jitter $50ms$.

Figure 11.5.: Specification of Environment Model

The Environment Model function also needs the actual egopose in order to update the road model. The curve segments delivered by the lane detection function reflect the time point of the frame from which they are extracted. This *age* is expressed in C_{ld} (Figure 11.4), and has a maximum value of $60ms$. Due to the dynamics of the ego vehicle, the curve segments represent a state in the past and the Environment Model function has to add a corresponding correction to calculation. This can be done only if the vehicle dynamics is known, for which the function gets regular updates of the egopose. For the sake of simplicity, the corresponding timing specifications are omitted. However, also the egopose events would be constrained by their age similar to the age of the curve segments.

11.1.3. Trajectory Planning

The Trajectory Planning function is intended to be developed in a similar way as the Environment Model function. It takes updates of the road model and calculates safe paths along the road with a certain goal "in mind". The paths delivered by the function are "lists" of trajectory points, which contain the vehicle's position, its direction and speed, together with a time point (in the future) when this point has to be reached.

In contrast to the Environment Model function, Trajectory Planning is expected to execute in cycles of variable length. This is expressed by the specification C_{tp} at the bottom of Figure 11.6, saying that the update of the trajectory points happens in intervals of length 100 to $150ms$. The argument of the age of the trajectory points is similar to the one for the calculation of the road model.

From the functional (and safety) viewpoint it is important to mention that trajectory points point to the future. The delivered paths are intended to guide the vehicle for the next, say, $200ms$. It hence would be dangerous if no updates would occur within this time span. This relation to the timing aspect is expressed by the guarantee that the Trajectory Planning function delivers updates with the specified inter-arrival times. Last but not least, also the Trajectory Planning needs the actual egopose of the vehicle, which again is not further considered here with respect to its specification.

11.1.4. Dynamic Control

The last considered component is the Dynamic Control function. It assumes, in correspondence to the guarantee given by the Trajectory Planning function, that updates of trajectory points are delivered

C_{tp}	A	Roadmodel occurs every $100ms$ with jitter $100ms$. Egopose occurs every $10ms$ with jitter $5ms$.
	G	Age(Trajectorypoints,Roadmodel) within $[0, 250[ms$. Trajectorypoints occurs every $[100, 150]ms$.

Figure 11.6.: Specification of Trajectory Planning

regularly every $100 - 150ms$. The two specifications at the bottom of the guarantee, reflect the needs discussed for the top-level system function from above.

The age patterns play a key role in the specification of the Dynamic Control behavior. The control function takes the trajectory points, one by one, and calculates a series of control parameters for the underlying actuation control function in order to approach the intended point as close as possible. When the time instant (and the trajectory point) has been reached, then the next point is taken for the calculations. As time passes by, "older" trajectory points disappear from the list. Hence, it would be a valid argument that the Dynamic Control function always operates on relatively fresh data.

From the viewpoint of the top-level requirements, this is however an invalid argument. It remains true that the information "stored" in the delivered path always reflects the state of the time point when the trajectory points have been delivered. Hence, the specification states that the age of the data on which the `LR` and `QR` events rely on is the maximum time between successive updates of the trajectory points, plus one execution period of the Dynamic Control function. Note that this implies that the function does not operate on old values as soon as updates are available and thus the buffer is overwritten (flushed) on updates. This will (as for the other functions) become relevant in the next design step.

C_{dc}	A	Trajectorypoints occurs every $[100, 150]ms$. Egopose occurs every $10ms$ with jitter $5ms$.
	G	Age(LR,Trajectorypoints) within $[0, 155]ms$. Age(QR,Trajectorypoints) within $[0, 151]ms$. LR occurs every $5ms$. QR occurs every $1ms$.

Figure 11.7.: Specification of Dynamic Control

11.1.5. Virtual Integration Testing

After the functional decomposition, where the internal top-level functional architecture is defined together with the corresponding (timing) specifications, Contract-based Design asks for discharging emerging proof obligations. In the present scenario, the proof obligation are the result of the decomposition of the system and states that the top-level requirement must be fulfilled by the composition of the specifications of its subcomponents and their interaction. According to Appendix C this is formally stated as

$$C_{tl} \succeq C_{ld} \otimes C_{em} \otimes C_{tp} \otimes C_{dc}$$

where $\succeq$ means that the specification of the parent component must be refined by the composition ($\otimes$) of the specifications of its respective subcomponents. The proof obligation is often divided into two conditions to be checked, which is also called *Virtual Integration Test*. The first condition asks whether the input assumptions of the individual components capture all the behavior which may be observed from all other interacting components. In our case study, the contracts are specified in a

way that makes this part rather easy. The first assumption of the component "Trajectory Planning", for example, is immediately discharged by the guarantees of component "Environment Model".[5] In other scenarios, such as in the case where components already exist, this is often not that obvious. This is because such components are typically designed in other contexts and state guarantees that do not match precisely the corresponding assumptions.

The second condition expresses the requirement that the guarantee of the top-level specification is fulfilled by the composition of the guarantees of its subcomponents, i.e., whether

$$A_{tl} \cap G_{ld} \cap G_{em} \cap G_{tp} \cap C_{dc} \subseteq G_{tl}$$

A short calculation of the specified latencies (ld: 60, em: 250, tp: 250, dc: 155) results in $715ms$, which is reasonable with respect to the top-level requirement. It might be an interesting fact that the ages of data can simply be summed up as one would expect for "simple" end-to-end delays. Formally, this is due to the fact that also causal relations are transitive as discussed in Section 3.2.4.

Note that the number $715ms$ does not include communication latencies between the components. This has been omitted in the discussion above for simplicity. In Figure 11.3, they are indicated by contracts C_{com}, which are assumed to specify (simple) reaction times. With three communication channels involved in the function chain, the overall communication delay must not be larger than $85ms$. Otherwise, the top-level specification would be violated.

However, we are typically seeking for tool support in performing the VIT. There are both tools from academia as well as industrially capable approaches. The authors of [31], for example, present an approach where VIT is performed on timing specification patterns (which are a subset of those presented in this document) using the Uppaal tool suite.

11.1.6. Simulation-based VIT

The previous section names VIT as the methodology to proof that the top-level requirements are fulfilled by the composition of the specifications of its subcomponents. It is also said that this could be achieved by formal methods. This is true for formal models of manageable complexity. Unfortunately, such formal methods don't scale very well and the process of model checking with tools like Uppaal can become cumbersome or even close to impossible.

We decided for an alternative, simulation based approach. We are aware that a Simulation-based VIT is not capable to guarantee completeness of the test[6]. Instead, specific scenarios can be tested and the simulation model can serve as an integration platform for functional testing. In the following section, an example for a Simulation-based VIT of the functional design at level A is provided. Simulation based approaches are of particular industrial relevance. In the context of this case study, we manually set up a simulation environment based on SystemC to demonstrate the feasibility. For sure it has to be the focus of future work to develop tools that automate the synthesis of VIT simulation models from specified contracts directly.

SystemC is an active IEEE standard [45] for system simulation and available as an open source C++ class library[7] which provides an event-driven simulation interface. Designed and widely applied as an event based simulation environment for chip design verification, SystemC is capable of handling parallel events and processes, as well as a notion of time. Furthermore, the SystemC Verification library (SCV) extends SystemC for functional verification. These features of SystemC support the implementation of an executable system model and contracts for a Simulation-based VIT.

[5] Indeed, a guarantee of the "ESC" component is needed to discharge the whole assumption.

[6] Formal methods (e.g., [31, 21]) guarantee completeness and should be used in a combination with simulation based testing.

[7] `http://accellera.org/downloads/standards/systemc/`

In this project, we assume that a SystemC simulation model can be generated from a SysML component model with contracts. SysML components can be transformed into SystemC modules, while SysML component ports are transformable into SystemC ports. The SysML contracts can be transformed into observer automata (e.g., implemented as an observer process in a SystemC module) or the assumption part of a contract can be transformed into an event generator to trigger the simulation model within the specified timing constraints (compare with Figure 11.8).

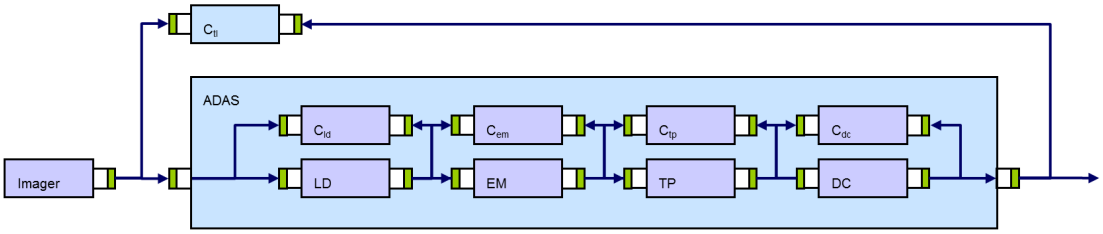


Figure 11.8.: Contracts SystemC Simulation Architecture

Figure 11.9 symbolizes how this is to be understood. In SystemC progression of time is explicitly modeled by a call of the SystemC wait-function. A periodic occurrence of an event is implemented by such a wait of the proper time inside, e.g.,a while-loop. Random aspects like a jitter are modeled by use of the SystemC add-on library SCV. So called "bags" filled with possible values are defined to randomly draw from. Writing to SystemC signals causes for the notification of the remainder simulation model for this event which triggers further responses.

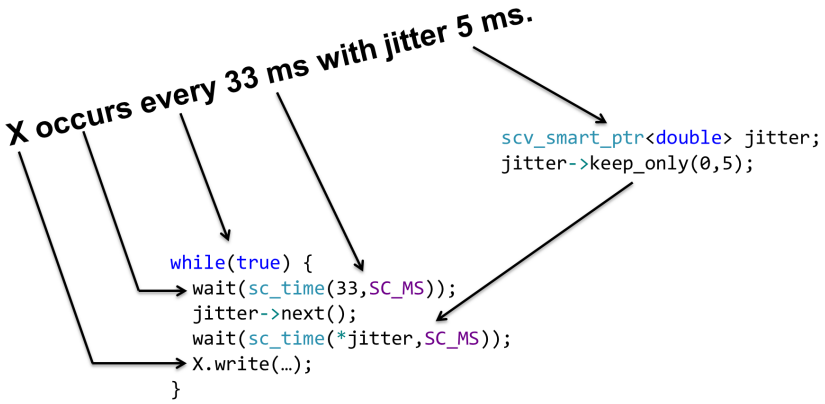


Figure 11.9.: Generating SystemC Simulation Models from Contracts

In the following example these transformation techniques have been manually applied on the case-study. For this report we decided to depict the outcome of a Simulation-based VIT first before we outline the implementation of contracts in SystemC in more detail. We believe a better understanding of what the simulation looks like facilitates the understanding of the code examples. For the sake of simplicity, we chose the contract C_{ld} of the Lane Detection component as an illustrative example instead of explaining every contract. In the following, we are referring to it as "the contract". Figure 11.10 depicts the VIT's Lane Detection simulation signal trace.

Executing the contract requires a stimulus for which we realized a SystemC dummy component of the $Imager$ that generates a new-frame-signal-event every $33ms$ added with a random jitter of

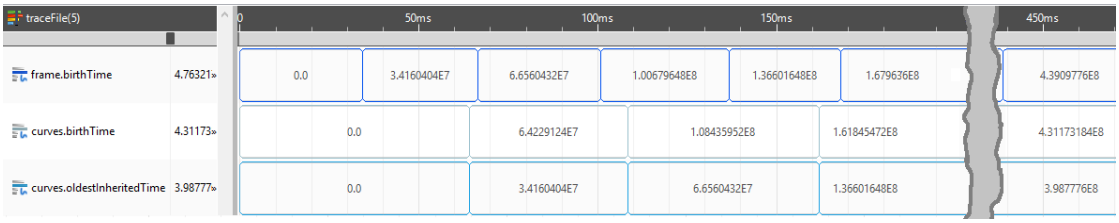


Figure 11.10.: Signal Trace of SystemC Simulation of Lane Detection Contract C_{ld}

up to $5ms$ according to the specification. We configured the stimulus and communication to be of structure-type signals with various value members (cf. Listing 1). One member is labelled $birthTime$ which is shown for the signal $Frame$ in the first line of Figure 11.10. The birth time defines the point in simulated time at which a signal's event was created. The time base is configurable, in this case for a resolution of $1ns$. The shown value of $3.416040E7ns$ therefore translates to roughly $34.2ms$.

```
template<typename T, T init>
struct multic_signalTypeStruct {
  // constructor
  multic_signalTypeStruct(
    tDataId birthId = 0,
    tTime birthTime = 0,
    tTime oldestInheritedTime = 0,
    tTime newestInheritedTime = 0,
    T payload = init)
  {
    m_birthId = birthId;
    m_birthTime = birthTime;
    m_oldestInheritedTime = oldestInheritedTime;
    m_newestInheritedTime = newestInheritedTime;
    m_payload = payload;  // Actuall data transported
  }
};
```

Listing 1: Signal of Type Struct

The second line of the same figure shows the birth time of the first $Curve$ to be at approx. $64ms$. The contract guarantees that $LaneDetection$ reacts to the frame event within $20ms$ to $60ms$. The dummy component of $LaneDetection$ draws a random delay value from this interval to mimic various computation delays. In the case shown this was $30ms$ for the first frame and approx. $42ms$ for the second frame. The last is too late to still handle the third frame-signal arriving at approx. $100ms$. This frame is dropped from computation. For at least the next three frame events the second guarantee of C_{ld} ensures, that from now on a narrower delay interval from $0ms$ to $33ms$ is guaranteed. The third curve event shows accordingly a random computation delay of approx. $25ms$.

$oldestInheritedTime$ is a copied time-stamp value from a causing signal event for a dependent, new signal event. In the example, the cause for the birth of the curves-signal at approx. $64ms$ was due to the frame-signal at approx. $34ms$. The frame-signal itself is not caused by another event in the past, so the frame's inherited time is the same as its birth time. The third $curves$ declares approx. $137ms$ as its starting cause time. The $frame$ at approx. $100ms$ was dropped because the random computation delay for $curves$ exceeded the arrival time of that frame. Hence, there is no curve in the signal trace which has this time as its oldest inherited time-stamp.

In Part III, the final demonstrator will show how this inherited time stamps travel all the way through the system to the output of Dynamic Control C_{dc}. A subtraction of the inherited time from the current simulation time defines the response delay of the system for an event in its past. The structure $signalTypeStruct$ can easily and should be expanded for further requirements on information tracking throughout VIT.

The dummy frame component fulfills the contract's assumption for a period of $400ms$ after which it (intentionally) fails to do so. The period deteriorates to $44ms$ after which the assumption of C_{ld} is invalidated. As a consequence, the contract terminates and is no longer operational. The curves-signal is no longer updated. The following output trace of Listing 2 documents more clearly what happens during a simulation run of 1 second time frame. Simulation of many hours can be executed within minutes likewise.

```
1         SystemC 2.3.1-Accellera --- Jan 26 2017 08:50:16
2         Copyright (c) 1996-2014 by all Contributors,
3         ALL RIGHTS RESERVED
4
5  Info: MAIN_INFO: Elaborating main simulation ...
6      <simulation length:   1 s
7
8  Info: ELABORATION_INFO: Elaborating component imager...
9  Info: ELABORATION_INFO:     Frame occurs every 33 ms with jitter 5 ms.
10 Info: ELABORATION_INFO: Elaborating component esc...
11 Info: ELABORATION_INFO:     Egopose occurs every 10 ms with jitter 5 ms.
12 Info: ELABORATION_INFO: Elaborating component laneDetectionClass_Inst...
13 Info: ELABORATION_INFO:     G: Age(Curves, Frame) within [20 ms,60 ms].
14 Info: ELABORATION_INFO:     G: Reaction(Frame, Curves) within [0 s,33 ms] 3 out of 4
   ↪  times.
15 Info: ELABORATION_INFO: Elaborating contract monitorcontractLd...
16 Info: ELABORATION_INFO:     A: Frame occurs every 33 ms with jitter 5 ms.
17 Info: ELABORATION_INFO:     G: Age(Curves, Frame) within [20 ms,60 ms].
18 Info: ELABORATION_INFO:     G: Reaction(Frame, Curves) within [0 s,33 ms] 3 out of 4
   ↪  times.
19 Info: ELABORATION_INFO: Elaborating component environmentModel...
20
21 ...
22
23 Info: MAIN_INFO: Starting main simulation ...
24 Info: (I702) default timescale unit used for tracing: 1 ns (traceFile.vcd)
25 Info: RUNTIME_INFO: 34160403 ns Assumption monitor of contractLd triggered by new frame
   ↪  event. OK.
26 Info: RUNTIME_INFO: 64229123 ns laneDetectionClass_Inst reacted to new frame event @
   ↪  34160403 ns. Processing delay was: 30068720 ns *** ON TIME ***
27 Info: RUNTIME_INFO: 64229123 ns Guarantee monitor of contractLd triggered by new frames
   ↪  event followed by curves event.
28 Info: RUNTIME_INFO: 66560431 ns Assumption monitor of contractLd triggered by new frame
   ↪  event. OK.
29 Info: RUNTIME_INFO: 100679647 ns Assumption monitor of contractLd triggered by new frame
   ↪  event. OK.
30 Info: RUNTIME_INFO: 108435948 ns laneDetectionClass_Inst reacted to new frame event @
   ↪  66560431 ns. Processing delay was: 41875517 ns *** LATE ***
31 Info: RUNTIME_INFO: 108435948 ns Guarantee monitor of contractLd triggered by new frames
   ↪  event followed by curves event.
32
```

```
...

Error: RUNTIME_INFO: 439097752 ns Assumption monitor of contractLd triggered by new
    ↪  frame event.
VIOLATION of:     A: Frame occurs every 33 ms with jitter 5 ms.
Jitter detected is: 10097752 ns
In file:
    ↪  p:\multic_-_fat_ak31_ausschreibung\work\frankp\modeling\projects\laneassist\src\cld.cpp:108
In process: adas.contractLd.cld_A @ 439097752 ns

...

Info: MAIN_INFO: Ending main simulation (1 s) ...
```

Listing 2: VIT Simulation Output Trace

The contract with its assumptions and guarantees is implemented as a SystemC module with methods and threads. Events are implemented through SystemC signals. The contract module is simulated in parallel to the component which it monitors (refer to Figure 11.8). The component's inputs signals connect to contract's assumption monitor while the component's outputs connect to the contract's guarantee monitor. For a centralized maintenance of all contracts' values it was decided to keep all constant values together in a single header file as listed in Listing 3.

```
// ** ASSUMPTIONS
// Frame occurs each 33 ms with jitter J_Frame
#define CLD_A1_FRAME_OCCURS (sc_time(33, SC_MS))
#define CLD_A1_FRAME_OCCURS_JITTER (sc_time( 5, SC_MS))
// ** GUARANTEES
// Age(Curves, Frame) within[20, 60]ms.
#define CLD_G1_AGE_CURVE_FRAME_INTERVAL_START  (sc_time(20, SC_MS))
#define CLD_G1_AGE_CURVE_FRAME_INTERVAL_END  (sc_time(60, SC_MS))
// Reaction(Frame, Curves) within[0, 33]ms 3 out of 4 times.
#define CLD_G2_REACTION_FRAME_CURVES_INTERVAL_START (sc_time(0, SC_MS))
#define CLD_G2_REACTION_FRAME_CURVES_INTERVAL_END (sc_time(33, SC_MS))
#define CLD_G2_REACTION_FRAME_CURVES_SET_ALL (4)
#define CLD_G2_REACTION_FRAME_CURVES_SET_OK (3)
```

Listing 3: Definition of Contract Parameters

The constants are the source of values for the contract's constructor which is declared as follows with Listing 4:

```
1  class contractLdClass : public sc_module {
2  public:
3    SC_HAS_PROCESS(contractLdClass);
4
5    // Constructor
6    contractLdClass(
7      sc_module_name moduleName,
8      sc_time cld_a1_frame_occurs,
9      sc_time cld_a1_frame_occurs_jitter,
10     sc_time cld_g1_age_curve_frame_interval_start,
11     sc_time cld_g1_age_curve_frame_interval_end,
12     sc_time cld_g2_reaction_frame_curves_interval_start,
13     sc_time cld_g2_reaction_frame_curves_interval_end,
14     unsigned char cld_g2_reaction_frame_curves_set_all,
15     unsigned char cld_g2_reaction_frame_curves_set_ok
16    );
17
18   // IO ports
19   sc_in<MULTIC::multic_signalTypeStruct> i_frame;
20   sc_in<MULTIC::multic_signalTypeStruct> i_curves;
21
22 private:
23   void cld_A(void);
24   void cld_G(void);
25
26   // Contract's relevant assumption values
27   const sc_time m_cld_a1_frame_occurs;
28   const sc_time m_cld_a1_frame_occurs_jitter;
29   // Contract's relevant guarantee values
30   const sc_time m_cld_g1_age_curve_frame_interval_start;
31   const sc_time m_cld_g1_age_curve_frame_interval_end;
32   const sc_time m_cld_g2_reaction_frame_curves_interval_start;
33   const sc_time m_cld_g2_reaction_frame_curves_interval_end;
34   const unsigned int m_cld_g2_reaction_frame_curves_set_all;
35   const unsigned int m_cld_g2_reaction_frame_curves_set_ok;
36
37   bool m_contractOK;
38   sc_time m_cld_a_last_frame_occured;
39   sc_time m_cld_g_last_frame_occured;
40   sc_time m_nextFrameEvent;                    // time for next frame
41   unsigned int m_shiftRegLate;
42   unsigned int m_countRegLate;
43 };
```

Listing 4: Declaration of Contract C_{ld} Class

The SystemC module implementation defines two input ports. The first named i_frame is connected to the input of the $LaneDetection$ to be able to monitor the arrival of $frame$ events. The second named i_curves connects to the output of $LaneDetection$ to monitor guarantees. The next two listings depict the implementation of assumption monitor (Listing 5) and guarantee monitor (Listing 6) which where manually implemented from the above defined C_{ld}.

Line 4 of Listing 5 enables the contract to disable itself once it monitored a trace that disqualifies the

contract. With any assumption violated it is no longer meaningful to continue and the initially $true$ boolean value $m\_contractOK$ becomes $false$ (lines 19 and 36). Right before the assumption monitor pauses its execution (line 40) it forecasts the next expected frame event by adding the frame-period time to the member $m\_nextFrameEvent$. Once the sc_method is awakened again by an event on i_fram, to which it is sensitive, it compares the current simulation time to the expected time for an event (line 11). Should it have been awakened before the planned next event, the period constraint was violated and an according message is generated using the SystemC report handler (line 17). The remainder of the code works accordingly.

```
1  void contractLdClass::cld_A(void) {
2    std::stringstream msg;
3    m_cld_a_last_frame_occured = sc_time_stamp();
4    if (m_contractOK) { //
5      msg << sc_time_stamp() << " Assumption monitor of "
6                     << basename() << " triggered by new frame event.";
7
8      // New frame event has arrived and triggered this method.
9      // Compare the current time with the planned arrival time.
10     // Arrival cannot be before scheduled next frame.
11     if (sc_time_stamp() < m_nextFrameEvent) {
12       // Assumption broken, contract no longer valid. Frame arrived too early
13       msg << "\nVIOLATION of: ";
14       msg << "    A: Frame occurs every " << m_cld_a1_frame_occurs
15           << " with jitter " << m_cld_a1_frame_occurs_jitter << ".";
16       msg << "\nFrame is early: " << (m_nextFrameEvent - sc_time_stamp());
17       sc_report_handler::report(SC_ERROR, "RUNTIME_INFO", msg.str().c_str(),
18                                 SC_DEBUG, __FILE__, __LINE__); msg.str("");
19       m_contractOK = false; //
20     }
21     // Arrival should be within the interval jitter after scheduled next frame.
22     if ((sc_time_stamp() - m_nextFrameEvent) <= m_cld_a1_frame_occurs_jitter) {
23       // Assumption OK
24       msg << " OK.";
25       sc_report_handler::report(SC_INFO, "RUNTIME_INFO", msg.str().c_str(),
26                                 SC_DEBUG, __FILE__, __LINE__); msg.str("");
27     }
28    else {
29       // Assumption broken, contract no longer valid
30       msg << "\nVIOLATION of: ";
31       msg << "    A: Frame occurs every " << m_cld_a1_frame_occurs
32           << " with jitter " << m_cld_a1_frame_occurs_jitter << ".";
33       msg << "\nJitter detected is: " << (sc_time_stamp() - m_nextFrameEvent);
34       sc_report_handler::report(SC_ERROR, "RUNTIME_INFO", msg.str().c_str(),
35                                 SC_DEBUG, __FILE__, __LINE__); msg.str("");
36       m_contractOK = false; //
37     }
38   }
39   // Expecting next frame event around ... (+/- jitter)
40   m_nextFrameEvent += m_cld_a1_frame_occurs; //
41 }
```

Listing 5: Definition of Assumption-Method

Definition of the guarantee method is part of Listing 6. The implemented observer waits until it is awakened by a change event on the frame-signal (line 15). The simulated time-stamp of this event is remembered (line 17) before the observer continues waiting for a response on the curves-signal (line 20). The expected delay between the two events should be small enough "3 out of 4 times". Tracking of this constraint is implemented using a shift-register behavior (lines 27 onward). The number of exceeded processing delays is counted in $m\_countRegLate$ (line 42) and should never increase above the "3 out of 4 times" contract. If so (line 47), an according error message is thrown.

```
1  void contractLdClass::cld_G(void) {
2  /*  The check here is if m out of n events arrive in time.
3      For our example this is 3 out of 4.
4      On a slow event we shift a 1, a zero else. We also count
5      how many 1s there are in our shift. This would look
6      like this, after one slow and 3 fast events:
7      shift = 1 0 0 0   |   count = 1
8      We need to look at bit n. If we shift in another error,
9      count goes to 2. But the old error is pushed out resulting in:
10     shift = 0 0 0 1   |   count = 1
11 */
12   std::stringstream msg;
13   while (m_contractOK) {
14     // Wait for a frame to arrive. This causes for simulated time to pass.
15     wait(i_frame.value_changed_event()); //
16     // Remember arrival time of this frame
17     m_cld_g_last_frame_occured = sc_time_stamp(); //
18
19     // Wait for reaction on this event
20     wait(i_curves.value_changed_event()); //
21     msg << sc_time_stamp() << " Guarantee monitor of "
22         << basename() << " triggered by new frames event followed by curves event.";
23     sc_report_handler::report(SC_INFO, "RUNTIME_INFO", msg.str().c_str(),
24                               SC_DEBUG, __FILE__, __LINE__); msg.str("");
25
26     // Shift reg by one to make room for the new event information
27     m_shiftRegLate = m_shiftRegLate << 1; //
28     // Check if an old error falls out the shift. If so reduce counter.
29     // using a bit mask of (1 << m_cld_g2_reaction_frame_curves_set_all)
30     if (m_shiftRegLate & (1 << m_cld_g2_reaction_frame_curves_set_all)) {
31       // Highest position is true. Forget outdated slow event from counter.
32       m_countRegLate--;
33     }
34     if ((m_cld_g_last_frame_occured + m_cld_g2_reaction_frame_curves_interval_end) >
35          sc_time_stamp()) {
36       // G2 is fullfiled => fast reaction on frame
37       // add zero error (good) to shift and count
38     } else {
39       // G2 not reached => slow reaction on frame
40       // add one error (bad) to shift and count
41       m_shiftRegLate++; // set lowest bit in shiftreg
42       m_countRegLate++; // count another bad into the shift //
43       if (m_countRegLate >
44           (m_cld_g2_reaction_frame_curves_set_all -
45                                       m_cld_g2_reaction_frame_curves_set_ok)) {
```

```
46          // Too many late frames! Guarantee is broken!
47          msg << sc_time_stamp() << " Guarantee monitor of "  //
48              << basename() << " for curves event.\nVIOLATION of: ";
49          msg << " Reaction(Frame, Curves) within ["
50              << m_cld_g2_reaction_frame_curves_interval_start << ","
51              << m_cld_g2_reaction_frame_curves_interval_end << "] "
52              << m_cld_g2_reaction_frame_curves_set_ok << " out of "
53              << m_cld_g2_reaction_frame_curves_set_all << " times.";
54          sc_report_handler::report(SC_ERROR, "RUNTIME_INFO", msg.str().c_str(),
55                                    SC_DEBUG, __FILE__, __LINE__); msg.str("");
56        }
57        if ((m_cld_g_last_frame_occured + m_cld_g1_age_curve_frame_interval_end) >
58            sc_time_stamp()) {
59          // G2 not reached but G1 is => slow reaction on frame good for m out n times
60        } else {
61          // G1 not reached => way too slow, ERROR
62          msg << sc_time_stamp() << " Guarantee monitor of "
63              << basename() << " for curves event.\nVIOLATION of: ";
64          msg << "    G: Age(Curves, Frame) within ["
65              << m_cld_g1_age_curve_frame_interval_start << ","
66              << m_cld_g1_age_curve_frame_interval_end << "].";
67          sc_report_handler::report(SC_ERROR, "RUNTIME_INFO", msg.str().c_str(),
68                                    SC_DEBUG, __FILE__, __LINE__); msg.str("");
69        }
70      }
71    }
72  }
```

Listing 6: Definition of Guarantee-Method

The practical experiment shows that SystemC VIT simulation could be a practical approach to verify proof obligations. The experiment was executed on a Windows 10 Notebook using Microsoft Visual Studio Professional 2015 C++ compiler. Any other C++ compiler and operating system are suitable alike. Other SystemC based projects at OFFIS make use of the GNU compiler chain under Debian Linux without any issues. Full selection of choices is given. We applied the Versions of 2.3.1 for SystemC and 2.0.0 for SCV. Figure 11.10 shows the Impulse Waveform Viewer plugin for the Neon Eclipse IDE. The discussed example is available as tar-ball. For a quick and easy "try it at home" OFFIS could also offer the image of a virtual machine containing a Linux system with installed SystemC, SCV and GNU-compiler.

11.2. Functional Design - Level B

The second design phase concentrates, in this case study, on the development of the actual functionality. To this end, the architecture is further decomposed until the leaf components have suitable granularity for implementation. The result of this refinement process is depicted in Figure 11.11. In this process, the engineers also decided on the (modeling) languages to be used for implementing the individual components. This is where the next two design paradigms come into play. First, the decision about the components implementation is also a decision about the underlying MoCs. Second, the interaction of components exploits Converter Channels.

A general discussion about interaction of component models with different notions of time and "standard" Converter Channels can be found in Part I. This case study focuses on Converter Channels

that are individually tailored for the considered use case. Furthermore, the case study does not cover the concrete implementation of functions. This will be the subject of Part III. Thus, as Figure 11.11 shows, decomposition covers only one step into the top-level functions. It assumes that the leaf components (LD, Model, Planner) are implemented with standard imperative languages such as C or C++. The function DC is implemented in Matlab/Simulink.

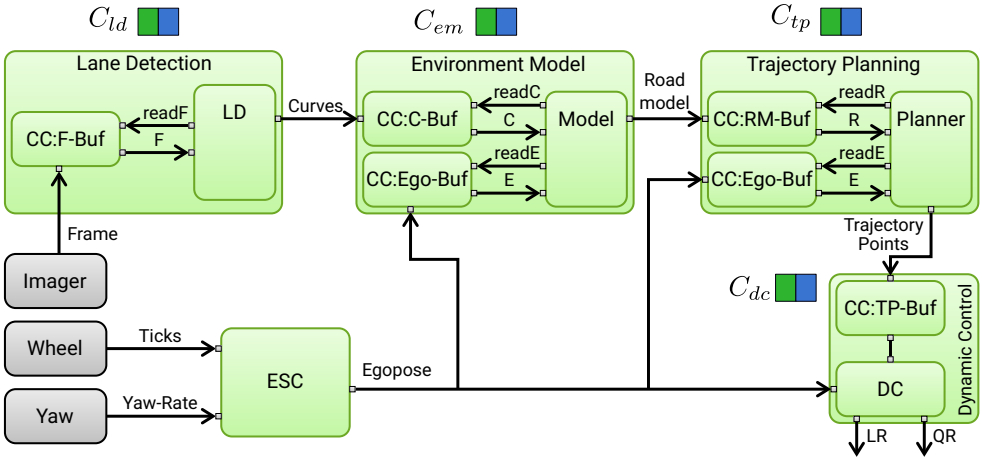


Figure 11.11.: Functional Architecture - Level B

On behalf of a continuous design process, the refinement of the architecture should be escorted by a corresponding refinement of the specifications. Moreover, the step from the functional architecture defined earlier to the actual one should be safeguarded. The semantic framework gives means to both tasks. The relation between the first and the actual architecture is so-called *realization*, which is closely related to the VIT discussed above. It aims on discharging the specifications of the top-level components of the actual architecture with those made in the previous design phase. That is, if the realization relation is satisfied then it is ensured that the requirements of the design level A are satisfied if the specifications of the actually considered architecture are satisfied.

Realization relations can be complex. In the considered case study, however, it is (deliberately) kept simple. In fact, the top-level components and their specifications are more or less a copy of the leaf components of the architecture of design level A (cf. Figure 11.3). The only difference is that the interfaces are refined in order to reflect the data types which shall be used for the actual implementation as well as for the technical realization of the system. This is however not visible in Figure 11.11. Hence, in our case, realization is a simple one-to-one relation, and no further analysis has to be performed in order to show satisfaction of the relation[8].

11.2.1. Architecture Refinement

Lane Detection Figure 11.12 shows the decomposition of component "Lane Detection" and the first application of the design paradigm "Converter Channels". The decomposition of the component into an interaction component (`CC:F-Buf`) that manages incoming input data, and a functional part (`LD`) that processes the input is a design pattern, which is continuously used along the case study.

[8]Recall that satisfaction of the top-level requirement by composing the component specifications has already been shown. Hence, given that realization is satisfied, the top-level specification of design level A is satisfied if the specifications of the top-level components of design level B are.

As said earlier, the Lane Detection function takes frames from the visual sensors to extract lane markings, which are delivered in form of curve segments to subsequent functions for further processing. It has also been stated that extracting the curves may take longer (up to $60ms$) than the time until another frame arrives at the input port of the component. A discussion among the engineers resulted in the decision that the function shall always take the most recent frame for calculations, though at least a certain number of frames shall be buffered in order to maintain some history information. This "semantics" conforms to a ring-buffer with size that corresponds to the intended history, and where all data can be accessed by the function.

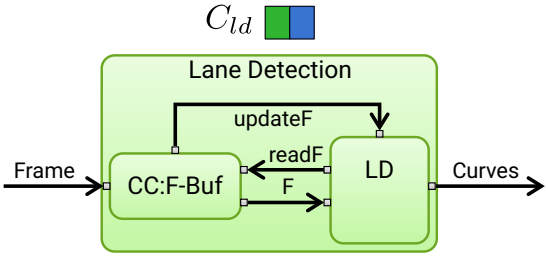


Figure 11.12.: Decomposition of Lane Detection

Figure 11.12 also shows that the interaction of the actual functional component `LD` with the Converter Channel is explicated as a *service call*. The function reads elements from the buffer, which is modeled as a `readF` event. The buffer delivers the corresponding data observable by an `F` event.

C_{ld}	A	Frame occurs every $33ms$ with jitter $5ms$.
	G	Age(Curves,Frame) within $[0, 60]ms$. Reaction(Frame,Curves) within $[0, 33]ms$ 3 out of 4 times.
$C_{CC:F-Buf}$	A	Frame occurs every $[1, 100000]ms$.
	G	whenever readF occurs then F occurs within $[0, 1]ms$. $<](F, Frame)(F_i, t_i) := (Frame_j, t_j) :$ $\nexists(Frame_k, t_k) \wedge t_j \leq t_k \leq t_i$. whenever Frame occurs then updateF occurs within $[0, 1]ms$.
C_{LD}	A	updateF occurs every $33ms$ with jitter $6ms$.
	G	Age(readF,updateF) within $[0, 2]ms$. Reaction(F,Curves) within $[0, 56]ms$. Reaction(F,Curves) within $[0, 24]ms$ 3 out of 4 times.

Figure 11.13.: Specification of Component "Lane Detection" and its Subcomponents

A key element of Converter Channels is the definition of *causal relations*. The top of Figure 11.13 rephrases the specification of the component `Lane Detection` (cf. Figure 11.4). According to Contract-based Design, the composition of its subcomponents must comply to this specification, as the design would otherwise be erroneous. The first specification of component `CC:F-Buf` says that the buffer delivers frame data (`F`) whenever it gets a `readF` request, which does not take longer than $1ms$. However, the delivered `F` is not yet related to the incoming `Frame` data events. This is particularly important if multiple `Frame` arrive before the next frame is requested, as shown at the top of Figure 11.14. Discharging the (causal) age specification of C_{ld} requires to know to which `Frame` the individual `Curve` belongs to. The underlying causal relation is specified in $C_{CC:F-Buf}$ as

$$<](F, Frame)(F_i, t_i) := (Frame_j, t_j) : \nexists(Frame_k, t_k) \wedge t_j \leq t_k \leq t_i$$

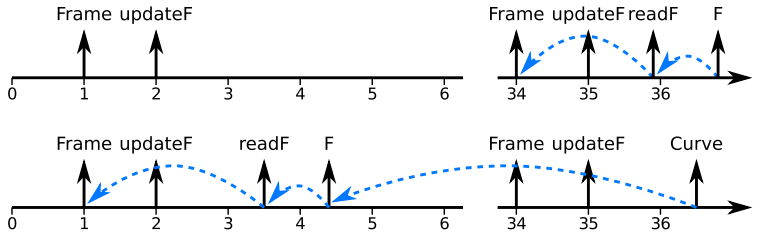


Figure 11.14.: Example Traces of Lane Detection

and defines that every occurrence (F_i, t_i) of F is causally related to the most recent incoming Frame. That is, to the occurrence of F at time instant t_i exists a causally related occurrence of Frame at time instant t_j such that no Frame occurrence exists in the time interval between t_i and t_j.

Specifying causal relations can be error prone when done by hand. It is hence strongly recommended to define Converter Channels in terms of library components, i.e., predefined components templates, that are instantiated for individual designs. This not only subsumes the specifications but also their implementation. The assumption of specification $C_{CC:F-Buf}$ indicates that the buffer can be instantiated in all contexts where input data arrive with minimum inter-arrival time of $1ms$. Concerning the functional aspect, Converter Channels can be verified a-priori using suitable functional verification tools such as simulation or formal verification. Concerning the timing aspect (such as the delay between read requests and replies), we propose parametric contracts that can be instantiated for the individual scenario.

The two Reaction specifications of component LD are the result of the considerations already made in design level A. The fact that the function requires most of the time specified in C_{ld} raises a synchronization issue. Every incoming frame that is processed by LD must be read within a very small amount of time ($2ms$), as otherwise the age constraint of $60ms$ would be violated. Hence, the design decision is made that the interaction between component CC:F-Buf and LD involves a synchronization mechanism, where the buffer informs the LD component about incoming frames, such that the component can react accordingly. This behavior is expressed by the two patterns at the bottom of $C_{CC:F-Buf}$ and the top of C_{ld}, respectively.

To summarize, the decomposition of the Lane Detection function into a Converter Channel and a functional part, which is later to be implemented by a suitable modeling language must again be discharged by performing a VIT. The following Listing 7 is an example of what the decomposition could look like in SystemC. Just like the functional design level A model, the model B is compatible and implements the same IO ports (lines 17 and 18). The same higher level contracts apply to this model (lines 8 to 13) and are enriched by the additional contracts of functional level B. The Listing 7 does not include the definitions of $C_{CC:F-Buf}$ yet, but will implement these for the final demonstrator. The two components CC:F-Buf and LD are instantiated in lines 27 and 28. Their example implementations are discussed in Section 11.2.2 and Section 11.2.3. The wiring between the two and their parent component $laneDetectionBClass$ is defined with lines 63 to 69. While the inputs and outputs are directly connected between parent and child (lines 64 and 69), the internal communication makes use of internal signals (lines 21, 65 and 67) and interfaces (line 68).

```
1 class laneDetectionBClass : public sc_module {
2 public:
3   SC_HAS_PROCESS(laneDetectionBClass);
4
5   laneDetectionBClass(
6     bool injectError,
```

```
7       sc_module_name moduleName,
8       sc_time cld_g1_age_curve_frame_interval_start,
9       sc_time cld_g1_age_curve_frame_interval_end,
10      sc_time cld_g2_reaction_frame_curves_interval_start,
11      sc_time cld_g2_reaction_frame_curves_interval_end,
12      unsigned char cld_g2_reaction_frame_curves_set_all,
13      unsigned char cld_g2_reaction_frame_curves_set_ok
14    );
15
16    // IO ports
17    sc_in<MULTIC::multic_signalTypeStruct<int,0>> i_frame;
18    sc_out<MULTIC::multic_signalTypeStruct<int,0>> o_curves;
19
20    // Signals
21    sc_signal<bool> s_updateF;
22
23  private:
24    // The lane detection component and functional level B consists
25    // of two sub-components, the converter channel plus the funtional processing
26    // component LD
27    laneDetectionCcClass m_laneDetectionCc;
28    laneDetectionLdClass m_laneDetectionLdClass;
29  };
30
31  laneDetectionBClass::laneDetectionBClass(
32    bool injectError,
33    sc_module_name moduleName,
34    sc_time cld_g1_age_curve_frame_interval_start,
35    sc_time cld_g1_age_curve_frame_interval_end,
36    sc_time cld_g2_reaction_frame_curves_interval_start,
37    sc_time cld_g2_reaction_frame_curves_interval_end,
38    unsigned char cld_g2_reaction_frame_curves_set_all,
39    unsigned char cld_g2_reaction_frame_curves_set_ok
40  ) :
41    sc_module(moduleName),
42    m_laneDetectionCc("laneDetectionConverterChannel"),
43    m_laneDetectionLdClass(
44      injectError,
45      "laneDetectionLD",
46      cld_g1_age_curve_frame_interval_start,
47      cld_g1_age_curve_frame_interval_end,
48      cld_g2_reaction_frame_curves_interval_start,
49      cld_g2_reaction_frame_curves_interval_end,
50      cld_g2_reaction_frame_curves_set_all,
51      cld_g2_reaction_frame_curves_set_ok
52    ),
53    // IO ports
54    i_frame("i_frame"),
55    o_curves("o_curves"),
56    // Signals
57    s_updateF("s_updateF")
58  {
59    std::stringstream msg;
```

```
60   msg << "Elaborating component " << basename() << "...";
61   sc_report_handler::report(SC_INFO, "ELABORATION_INFO", msg.str().c_str(),
62                             SC_MEDIUM, __FILE__, __LINE__); msg.str("");
63
64   // Wireing up the Converter Channel
65   m_laneDetectionCc.i_frame(i_frame);
66   m_laneDetectionCc.o_updateF(s_updateF);
67   // Wireing up the Lane Detection processing
68   m_laneDetectionLdClass.i_updateF(s_updateF);
69   m_laneDetectionLdClass.i_F(m_laneDetectionCc);
70   m_laneDetectionLdClass.o_curves(o_curves);
71 }
```

Listing 7: Definition of Lane Detection Class for Level B

Environment Model Decomposition of the Environment Model function, as well as the other functions, is similar to the decomposition of the Lane Detection function. In contrast to the Lane Detection function, however, the timing specification of the Environment Model does not require such tight synchronization, as discussed earlier.

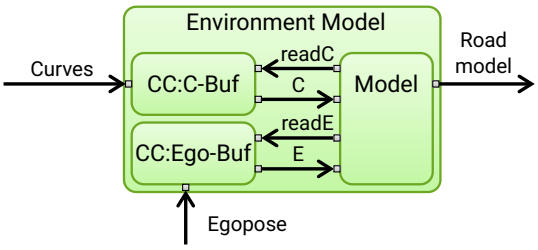


Figure 11.15.: Decomposition of Environment Model

The interaction between the curve buffer (CC:C-Buf) and the functional component EM is again modeled in terms of service calls, and hence is not repeated. The Converter Channel for the environment model though has slightly different semantics. The function calculates updates of the road model with respect to all curve segments that arrived since the last read request. That means, with every request all elements from the buffer are read. This must be reflected by the causal relation (and indeed the implementation of the buffer), as depicted in Figure 11.16.

In order to properly update the road model, the Environment Model function requires knowledge about the egopose that corresponds to the situation perceived by the camera frames. To this end, the function searches a history of egoposes for the value which time stamp matches best those of the actual curve. Hence, Converter Channel CC:Ego-Buf is a ring-buffer. The size of the ring buffer can be calculated from the knowledge about the maximum age of incoming curves (which is $60ms$ according to C_{ld}), and the age and frequency of incoming egopose data. Assuming a maximum age of, say, $20ms$ and updates every $10ms$, we need no more than 4 egopose updates in order to cover the time span of $60ms$. The corresponding causal relation is depicted in $C_{CC:Ego-Buf}$.

Trajectory Planning Refinement of the Trajectory Planning is again very similar to the refinement of the Environment Model.

C_{em}	A	Curves occurs every $[33, 66]ms$ with jitter $33ms$. Egopose occurs every $10ms$ with jitter $5ms$.
	G	Age(Roadmodel,Curves) within $[0, 250[ms$. Roadmodel occurs every $100ms$ with jitter $50ms$.
$C_{CC:C-Buf}$	A	Curves occurs every $[1, 100000]ms$.
	G	whenever readC occurs then C occurs within $[0, 1]ms$. $<](C, Curves)(C_i, t_i) := \{(Curves_j, t_j) \mid \exists(C_{i-1}, t_{i-1}) \wedge t_{i-1} < t_j \leq t_i\}$.
$C_{CC:Ego-Buf}$	A	Egopose occurs every $[1, 100000]ms$.
	G	whenever readE occurs then E occurs within $[0, 1]ms$. $<](E, Egopose)(E_i, t_i) := \{(Egopose_{j-3}, t_{j-3}), ..., (Egopose_j, t_j) \mid \nexists(Egopose_k, t_k) \wedge t_j < t_k \leq t_i\}$.
C_{Model}	G	readC occurs every $100ms$ with jitter $50ms$. Reaction(C,Roadmodel) within $[96, 196]ms$.

Figure 11.16.: Specification of Environment Model

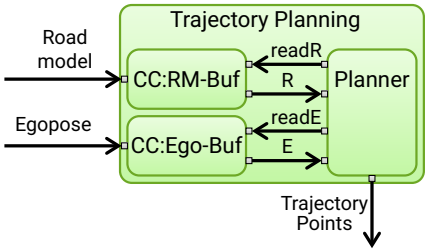


Figure 11.17.: Decomposition of Trajectory Planning

The timing specification does not require tight synchronization, and hence no further confinement of the execution of the function is required. The converter channel `CC:RM-Buf` is modeled as for the lane detection by a shared variable, which stores only the most recent updates. The specifications of the components are shown in Figure 11.18. The specification for the Converter Channel storing egopose updates and the interaction with the `Planner` component are omitted, as they are identical to those for the Environment Model function.

Dynamic Control The last component considered in this design phase is Dynamic Control. In contrast to the other components, this function shows an over-sampling behavior, where incoming trajectory points are processed with (much) higher frequency than it produces output data (cf. Figure 11.7). Also in contrast to the other functions, we do not assume that the functional part `DC` is an atomic functional unit. However, further decomposition of the function is out of scope of this report. Concerning the interaction of `DC` with the Trajectory Planning, it is assumed that the function assess the individual trajectory points according to their time stamp (which lays in the future) and the actual egopose. How this is modeled is deliberately left open in Figure 11.19, as this would require a deeper insight into the `DC` component. One possibility would be to model Converter Channel `CC:TP-Buf` as a simple shared variable, which is read with high frequency every $1ms$. An alternative is to model it as a FIFO buffer, which is overwritten when new trajectory points arrive.

C_{tp}	A	Roadmodel occurs every $100ms$ with jitter $100ms$. Egopose occurs every $10ms$ with jitter $5ms$.
	G	Age(Trajectorypoints,Roadmodel) within $[0, 250[ms$. Trajectorypoints occurs every $[100, 150]ms$.
C_{RM-Buf}	A	ReadR occurs every $[100, 150]ms$.
	G	whenever readR occurs then R occurs within $[0, 2]ms$. $<](R, Roadmodel)(R_i, t_i) := (Roadmodel_j, t_j) :$ $\nexists(Roadmodel_k, t_k) \wedge t_j \leq t_k \leq t_i$.
$C_{Planner}$	A	true.
	G	{ReadR,ReadE} occurs every $[100, 150]ms$. Reaction(R,Trajectorypoints) within $[100, 148]ms$. Reaction(R,ReadR) within $[100, 148]ms$.

Figure 11.18.: Specification of Trajectory Planning

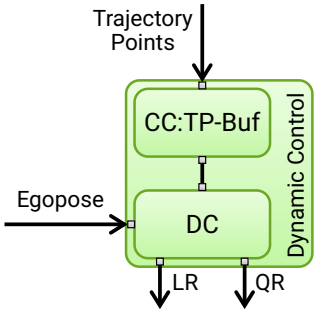


Figure 11.19.: Decomposition of Dynamic Control

11.2.2. Implementation

So far, the VIT is about timing requirements only, without aspects of functionality. These subsections explain how functionality could be added by a model of computation. For this example we assume the approach of defining functionality with Mathwork's Matlab/Simulink tool suite that is able to automatically generate production C/C++ (embedded) source code from its executable models. Listing 8 shows the instantiation of functional Matlab/Simulink code in lines 33, 34 and its usage in lines 61 to 69.

At the time of writing this partial report, the MULTIC working group did not have access to a model for the Lane Detection algorithm (e.g., Matlab/Simulink executable model). Developing the functionality of such a component is far beyond the scope of the planned effort in the MULTIC project. This is no obstacle though to explain in the following how in principle functionality can be included into the timing model. Due to the lack of access to a implementation for the Lane Detection algorithm, this report makes use of a simple constructed Matlab/Simulink model (see Figure 11.20) which consists of a simple data-path with one input named $i\_simulink$ (line 64) and one output $o\_simulink$ (line 69). After applying values to the input the functional model computes new values at its output by a call to its $step()$ method. Be aware that this functional code is not aware of any timing assumptions or delays. As part of the VIT the $step()$ method itself does not consume any simulated time. Progression of time in SystemC is achieved by a call to the $wait()$ function. Line 39 of the listing causes the model to proceed to that point in time at which a new frame value arrived in the

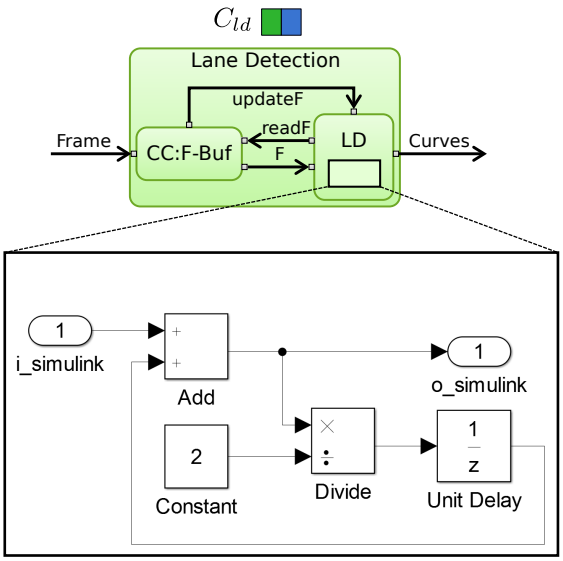


Figure 11.20.: Integration of Matlab/Simulink functional block model in Lane Detection component

Converter Channel. This is related to the contract guarantee definition "Reaction(Frame,Curves)" of C_{ld}. The wait of line 48 simulates the expected random execution delay of the functional code which is drawn randomly from either the interval $[0, 60]ms$ or $[0, 33]ms$ but in such a way to guarantee the use of the shorter period by at least 3 out of 4 times according to C_{ld}. The implementation achieves this by switching back and forth between a slow bag of random values (lines following 18) and a fast bag of random values (lines following 24). Once a delay exceeded $33ms$ (lines 84) the fast bag is selected (line 90) for the next three (line 102) executions.

Only such execution delays should be specified and therefor modeled, which an application expert in the domain considers realistic. Further on in the development, once functional code is executed and measured on target hardware of ,e.g., an ECU, it is advisable to back-annotate the measurements into the model to enhance its accuracy.

```
1  void laneDetectionLdClass::framesToCurvesProcessor(void) {
2    // For messages through the SystemC reporting handler
3    std::stringstream msg;
4
5    unsigned int cld_g2_reaction_frame_curves_ok = m_cld_g2_reaction_frame_curves_set_all;
6
7    // Reset output of curves at start
8    o_curves.write(MULTIC::multic_signalTypeStruct<int, 0>(
9      0, // birthId
10     0, // birthTime
11     0, // oldestInheritedTime
12     0, // newestInheritedTime
13     0  // payload
14     ));
15
16   // Remove from random bag all values that are not part of the allowed delay times
17   // keep [20, 60]
18   scv_smart_ptr<double> reactionDelaySlow; //
```

```
19   reactionDelaySlow->keep_only(
20     m_cld_g1_age_curve_frame_interval_start.to_default_time_units(),
21     m_cld_g1_age_curve_frame_interval_end.to_default_time_units()
22   );
23   // keep [ 0, 33]
24   scv_smart_ptr<double> reactionDelayFast; //
25   reactionDelayFast->keep_only(
26     m_cld_g2_reaction_frame_curves_interval_start.to_default_time_units(),
27     m_cld_g2_reaction_frame_curves_interval_end.to_default_time_units()
28   );
29   // Initial random values to choose from are the slower one.
30   scv_smart_ptr<double> *reactionDelay = &reactionDelaySlow;
31
32   // Instantiating Simulink generated functional code
33   dummyShowOffForConceptModelClass dummyShowOffForConceptModel; // instantiation
34   dummyShowOffForConceptModel.initialize(); // initialization
35
36   // Guarantee will only be working if assumptions are OK
37   while (true) {
38     // Wait for a frame to arrive. This causes for simulated time to pass.
39     wait(i_updateF.value_changed_event()); //
40
41     // Select a random delay time from the defined intervall
42     (*reactionDelay)->next();
43     msg << basename() << " reacted to new updateF event @ "
44         << sc_time_stamp() << ". Processing delay was: "
45         << sc_time(**reactionDelay, tTimeBASE);
46
47     // "Burn" time that it takes to process the arrived frame
48     wait(sc_time(**reactionDelay, tTimeBASE)); //
49
50     // BUILD IN FAULTY BEHAVIOR AFTER SOME TIME
51     // Trigger contract to react accordingly
52     if (m_injectError) {
53       if (sc_time_stamp() > sc_time(400, SC_MS)) {
54         wait(sc_time(40, SC_MS));
55       }
56     }
57
58     MULTIC::multic_signalTypeStruct<int, 0> tmp;
59     i_F->read(tmp); // read data from converter channel
60
61     // Put your functional code here:
62     // Write data from converter channel to input port of the
63     // generated Simulink model.
64     dummyShowOffForConceptModel.rtU.i_simulink = tmp.m_payload; //
65     // Execute a single step of the generated Simulink model.
66     dummyShowOffForConceptModel.step();
67     // Read data from the output port of the generated Simulink
68     // model and pass it to the output port of the lane detection block.
69     tmp.m_payload = dummyShowOffForConceptModel.rtY.o_simulink; //
70
71     std::stringstream msgSimulink;
```

```
72      msgSimulink << sc_time_stamp() << " Simulink model executed with input: "
73                  << dummyShowOffForConceptModel.rtU.i_simulink << " resulted in output: "
74                  << dummyShowOffForConceptModel.rtY.o_simulink;
75      sc_report_handler::report(SC_INFO, "RUNTIME_INFO", msgSimulink.str().c_str(),
76                                SC_DEBUG, __FILE__, __LINE__); msgSimulink.str("");
77
78      // Write the simulink functional values to output
79      o_curves.write(tmp);
80
81      // Check if we where in time for processing the last frame.
82      // We can only miss 1 out of 4 to be late!
83      if (sc_time(**reactionDelay, tTimeBASE) >
84                      m_cld_g2_reaction_frame_curves_interval_end) { //
85        msg << " *** LATE ***";
86        // We are late. This can happen only once out of four times
87        // Now we have to be "good" for the next three times. Adjust
88        // random interval to be "in time"
89        m_cld_g2_reaction_frame_curves_interval_end.to_default_time_units());
90        reactionDelay = &reactionDelayFast; //
91        cld_g2_reaction_frame_curves_ok = 0;
92      } else {
93        msg << " *** ON TIME ***";
94        // We are on time.
95        if (cld_g2_reaction_frame_curves_ok <
96            m_cld_g2_reaction_frame_curves_set_all) {
97          // count one good frame
98          cld_g2_reaction_frame_curves_ok++;
99        }
100
101       if (cld_g2_reaction_frame_curves_ok >=
102           m_cld_g2_reaction_frame_curves_set_ok) { //
103         // Ehough good frames delivered. Now we could be late again.
104         // Adjust random interval to be "evil"
105         m_cld_g1_age_curve_frame_interval_end.to_default_time_units());
106         reactionDelay = &reactionDelaySlow;
107       }
108     }
109     std::stringstream msgtmp;
110     msgtmp << sc_time_stamp() << " " << msg.str(); msg.str(msgtmp.str());
111     sc_report_handler::report(SC_INFO, "RUNTIME_INFO", msg.str().c_str(),
112                               SC_DEBUG, __FILE__, __LINE__); msg.str("");
113   }
114 }
```

Listing 8: Instantiating functionality from Matlab/Simulink

11.2.3. Converter Channels

The chosen example defines the Converter Channel of the Lane Detection component as a buffer of five frames. This is a more sophisticated refinement as compared to functional level A. A visualization of what this refinement involves can be seen from a comparison of Figure 11.10 and the Figure 11.21 below. The trace of the *curves* events demonstrates that the *curves* are depended on the last five

frame values. Comparing the two subsequent computations of two curves values marked with green and red arrows of Figure 11.21 demonstrates how the oldest of five frames is dropped (or rather ignored) in case of a late processing delay.

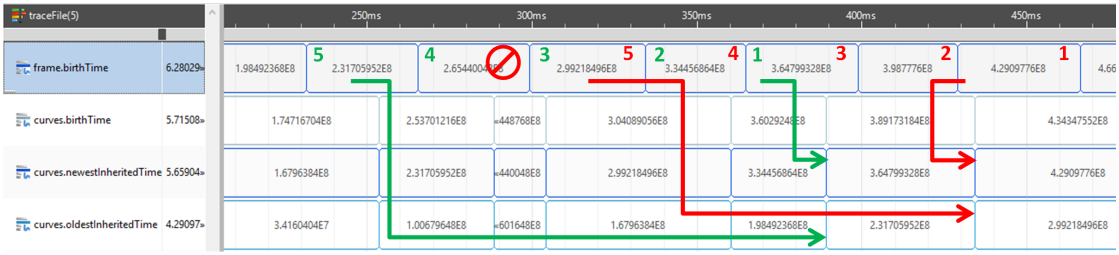


Figure 11.21.: Signal Trace of Lane Detection Contract C_{ld} Functional Level B

Listing 9 shows the SystemC implementation of the Converter Channel used to generate the above trace. It is connected to the $imager$ with its frame input port declared at line 9. A SystemC sc_thread method, which is defined from line 31 onward, reacts to arrivals of new frames (line 36) to store the last five frames in an array (declared in line 21) that is used as a ring-buffer. The event for a newly arrived frame is forwarded in line 49 to the processing component of the Lane Detection via the output $o\_updateF$.

The Converter Channel class is derived from $sc\_interface$ to implement a read interface called by the laneDetectionClass of Listing 8 in line 59. Listing 9 shows the dummy implementation of such a read interface from line 56 on. Usually we would expect here to see a datatype that is able to transport all required values from the channel to the processing component of the Lane Detection. For ease of code-writing and presentation we decided to just create a new $multic\_signalTypeStruct$ with a meaningless payload and oldest, as well as newest derived time taken from the array (lines 73 to 80).

```
1  class laneDetectionCcClass : public sc_module, public laneDetectionCc_read_if {
2  public:
3    SC_HAS_PROCESS(laneDetectionCcClass);
4    laneDetectionCcClass(
5      sc_module_name moduleName
6    );
7
8    // IO ports
9    sc_in< MULTIC::multic_signalTypeStruct<int,0> >  i_frame; //
10   sc_out<bool> o_updateF;
11
12   // Read Interface
13   void laneDetectionCc_read_if::read(MULTIC::multic_signalTypeStruct<int,0> &readout);
14
15 private:
16   // Thread to implement thr ring buffer behavior
17   void frameBufferRing(void);
18
19   // Storage buffer and iterator to current position in ring
20   std::array<MULTIC::multic_signalTypeStruct<int,0>,
21                    laneDetectionCcBufferSize>                m_frameBufferRing; //
22   std::array<MULTIC::multic_signalTypeStruct<int,0>,
23                    laneDetectionCcBufferSize>::iterator      m_frameBufferRingIter;
24
```

```
25   MULTIC::tDataId m_birthId;
26 };
27
28 /***********************************************
29  * SC_THREAD for lane detection ring buffer       *
30  ***********************************************/
31 void laneDetectionCcClass::frameBufferRing(void) {  //
32   std::stringstream msg;
33   while (true) {
34
35     // Wait for a frame to arrive. This causes for simulated time to pass.
36     wait(i_frame.value_changed_event()); //
37     msg << basename() << " reacted to new frame event @ " << sc_time_stamp();
38     sc_report_handler::report(SC_INFO, "RUNTIME_INFO", msg.str().c_str(),
39                               SC_DEBUG, __FILE__, __LINE__); msg.str("");
40
41     m_frameBufferRingIter++;
42     if (m_frameBufferRingIter == m_frameBufferRing.end()) {
43       // Completed circle in ring, back to the beginning
44       m_frameBufferRingIter = m_frameBufferRing.begin();
45     }
46     *m_frameBufferRingIter = i_frame.read();
47
48     // Sent updateF to LD
49     o_updateF.write(!o_updateF.read()); //
50   }
51 }
52
53 /***********************************************
54  * Converter Channel Read Interface               *
55  ***********************************************/
56 void laneDetectionCcClass::read(MULTIC::multic_signalTypeStruct<int,0> &readout) { //
57   // Acces to the converter channel. It stores a number of frames in a ring buffer to
58   // which the processing component requires parallel access. At this time we do not
59   // model any real data but we can process the modeled timing to check with
60   // the specification/contracts.
61
62   // The ring buffer iterator is pointing to the newest (last written) value inside the
63   // frame buffer. The next value in the ring is therefor the oldest.
64   std::array<MULTIC::multic_signalTypeStruct<int,0>,
65                   laneDetectionCcBufferSize>::iterator iter;
66   iter = m_frameBufferRingIter;
67   iter++;
68   if (iter == m_frameBufferRing.end()) {
69     // Circle completed, going back to beginning
70     iter = m_frameBufferRing.begin();
71   }
72
73   readout.m_birthId = m_birthId++; //
74   readout.m_birthTime = sc_time_stamp().to_double();
75   readout.m_oldestInheritedTime = iter->m_oldestInheritedTime;
76   readout.m_newestInheritedTime = m_frameBufferRingIter->m_newestInheritedTime;
77   // As this example does not contain meaningfull functional data, we just use
```

```
78   // the Id as data payload. For a real scenario, we would have to get real
79   // frame data here.
80   readout.m_payload = readout.m_birthId; //
81 }
```

Listing 9: Definition of Converter Channel Class

11.2.4. Checking Satisfaction

Similar to the VIT, which discharges the decomposition of specifications and, with some additional considerations, the realization across different design phases, the Contract-based Design paradigm calls for checking the *implementation* of components against their specifications.

Though functional verification is out of scope of this report, which concentrates on the timing aspect, this is an important and often the first considered aspect. As already mentioned in Part I, functional verification can done using formal verification, simulation, testing and code reviews.

11.3. Technical Design - Level B

The goal of the first technical design model is to map the functional design to software and hardware entities. We assume that AUTOSAR is used to model such a technical view. AUTOSAR focuses entirely on the software architecture of a system and only touches topics related to the underlying hardware where necessary. Therefore, functions are mapped to software components, respectively to runnables, which are the smallest executable code-fragments provided by a software component. Contrary to the previous functional designs, where mainly data flow and Converter Channels realizing the buffering of input data have been addressed, an AUTOSAR model also defines control flow aspects. This is done by defining activation conditions of runnables, by linking `RTEEvent`s to runnables, where the runnables shall be activated when the `RTEEvent` occurs. Different kinds of `RTEEvent`s are available in AUTOSAR. For instance, a `TimingEvent` can be modeled that occurs cyclically with a specified period, such that a referenced runnable is then cyclically executed. Other examples are `DataReceivedEvent`s, which occur when a new data item has been received at an input port of a software component. The semantics of these ports of software components is also configured at this design level. For example, input ports with a `SenderReceiverInterface` can be configured to implement a bounded FIFO buffer for storing received input values. Of course these configuration activities have an impact on the behavior of the system. Following the component concept of the Architecture Modelling Framework (AMF) proposed in Chapter 4, ports make certain events of the behavior of the owning component visible outside the component. These events are then referred to by specifications in terms of contracts. For specifying timing contracts in an AUTOSAR model, the AUTOSAR modeling language already provides an extensive list of such events, which is defined in the AUTOSAR Timing Extension [6]. Hence, the following discussion is based on them and it is assumed that the reader is to some degree familiar with the concepts of the timing extensions.

11.3.1. AUTOSAR Communication Specification as Converter Channels

The AUTOSAR modeling language provides various kinds of ports with different communication semantics, named $PortInterface$ in [3]. The most typical ones are: $SenderReceiverInterface$, $ClientServerInterface$ and $TriggerInterface$. Ports of a modeled component conform to one of these interfaces, either meaning that data items are sent or received, methods are provided or called, or triggers are sent or received through them. Besides this general communication semantics

defined by a $PortInterface$, AUTOSAR also provides means to further configure the communication semantics for each connected pair of ports. For example, for a receiving port conforming to a $SenderReceiverInterface$ the buffering of input values can be configured. Such a port can be setup to either maintain a queue of configured length of received data items or to maintain a buffer always holding the latest received element. Obviously, such buffering strategies have an impact on the age of data that is being processed by a runnable reading values from such a port. Even more, depending on the configured buffering, input values may get lost (overwritten), which may or may not be acceptable.

In other words, configuration of the communication semantics of AUTOSAR ports has an impact on satisfaction of contracts. This is because they have a certain behavior on their own. Following the component concept of the Architecture Modelling Framework (AMF) proposed in Chapter 4, ports just make certain events of the behavior of owning components visible outside the component. Connections between these ports denote the identity of their exposed behavior. For these reasons, AUTOSAR ports cannot be understood as ports in the sense of Section 4.1 but must be components with a behavior one might also attach contracts to. This statement is best explained by an example: Consider the AUTOSAR component depicted in Figure 11.22. The AUTOSAR Software Component SWC_A has a runnable R_1 that has read access to the input port inP of SWC_A. Now in case inP is configured as a bounded queue a read access of R_1 might not return the last received value but the value stored at the end of the FIFO queue of inP. If we now assume that a contract is specified for SWC_A whose guarantee requires a maximum reaction latency from inP to $outP$ then the contract might be satisfied only if the queue can store a certain amount of elements. A reduction of the maximum length of the queue may then result in lost input values because R_1 is not fast enough to process them. Fortunately, the AUTOSAR Timing Extension already provide classes of events that allow referring, e.g., to the observable point in time a new data item is received and is available in the communication buffer of the RTE. Furthermore, one can also refer to the observable event when a runnable accesses such a buffered data item. Hence, the AUTOSAR Timing Extension already provide an AMF compliant view, where an AUTOSAR port is an entity with behavior, meaning a component in the sense of the AMF. With regard to the terminology introduced in Part I, connected AUTOSAR ports correspond to Converter Channels realizing the communication between the corresponding SWCs. Thus, since AUTOSAR ports have a behavior on their own rather than just making behavior of the SWC visible to other SWCs, one can attach contracts to them. For instance, those contracts can be used to specify reaction latencies between the points in time a runnable reads data from such an input port until it actually receives the data using it. This, for example, allows specifying different behavior for the runnable depending on whether a read request times out or returns valid data.

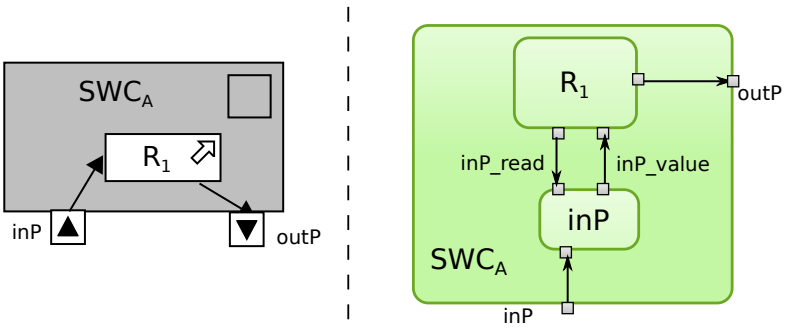


Figure 11.22.: AUTOSAR ports: AUTOSAR view VS AMF view

11.3.2. The AUTOSAR Software Component Architecture

We begin with the AUTOSAR VFB model, which is depicted at the bottom of Figure 11.23. For the sake of simplicity, we only consider an excerpt of it corresponding to the functions `EnvironmentModel`, `TrajectoryPlanning` and `DynamicControl` as discussed in Section 11.2. We assume that each of these functions is realized by a single software component (SWC) and each of these SWCs contains a single runnable whose code implements the desired function except for component `DynamicControl`, which has two runnables. Note, that these restrictions are only for the sake of simplicity, yet allow an illustration of the design paradigm. Realistic software architecture are likely to have software components, each consisting of a set of subcomponents, hosting dozens of runnables.

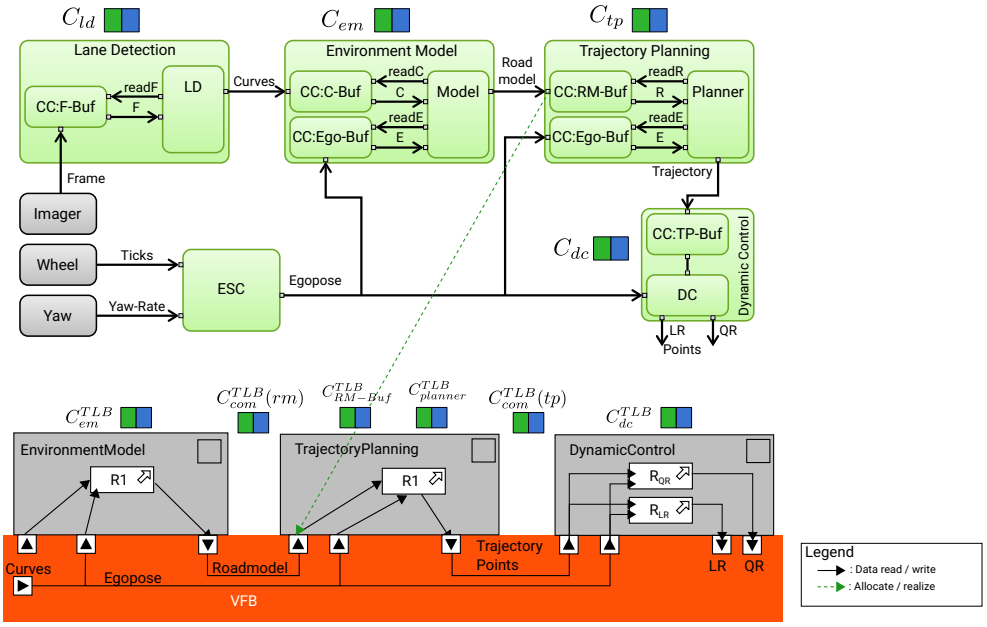


Figure 11.23.: Allocation of Functional Design to AUTOSAR VFB-level Architecture

Now the goal is to come up with timing specifications for the SWCs and containing runnables, such that the contracts of the functional architecture on level B are realied. In the sense of Contract-based Design this means the contracts of the AUTOSAR VFB model refine the contracts of the functional architecture, or more formally:

$$C^{TLB}_{em} \otimes C^{TLB}_{RM-Buf} \otimes C^{TLB}_{planner} \otimes C^{TLB}_{dc} \preceq C_{em} \otimes C_{RM-Buf} \otimes C_{Planner} \otimes C_{dc} \tag{11.1}$$

Given that the condition above is satisfied, we can conclude that a technical implementation of the SWCs `EnvironmentModel` and `TrajectoryPlanning` and `DynamicControl`, fulfilling the contracts C^{TLB}_{em} and C^{TLB}_{tp} and C^{TLB}_{dc} indeed also fulfills the contract specifications of the functional design according to the defined mapping. Then by transitivity of the relation $\preceq$ also the top-level functional specification C_{tl} is fulfilled. This reasoning across models of different perspectives is called *allocation* in Chapter 4 and requires a definition how observable events in the different models are related to each other. An example is indicated in Figure 11.23 by the dashed green arrow. As already discussed, we make use of classes of observable events specified in the AUTOSAR Timing Extension [6]. Thus, for each such event that is modeled in AUTOSAR, a corresponding event in the functional design on level B must be identified. Then the reasoning above is well-defined.

EnvironmentModel

The SWC implementing the `EnvironmentModel` is linked with the following contract C_{em}^{TLB} shown in Figure 11.24 specifying its assumed and guaranteed timing behavior. The engineers decided that the runnable `R1` of `EnvironmentModel` shall be executed cyclically every $100ms$, sampling the inputs `Curves` and `Egopose` on each invocation. Since the precise activation rate is not under the control of the developers responsible for implementing `EnvironmentModel`, the cyclic activation is part of the assumption of C_{em}^{TLB}. In the guarantee C_{em}^{TLB} a reaction latency of $[50, 55]ms$ is required from an activating event until the runnable produces its causally related output, the computed `Roadmodel`.

C_{em}^{TLB}	A	EvVFBRecv(Curves) occurs every $[33, 66]ms$ with jitter $33ms$. EvVFBRecv(Egopose) occurs every $10ms$ with jitter $5ms$. EvSWCAct(EnvironmentModel.R1) occurs every $100ms$ with jitter $5ms$.
	G	Age(EvVFBSnd(Roadmodel),EvVFBRecv(Curves)) within $[0, 250[ms$. Reaction(EvSWCAct(EnvironmentModel.R1),EvVFBSnd(Roadmodel)) within $[50, 55]ms$.

Figure 11.24.: Specification of EnvironmentModel (technical architecture level B)

Please note that the notations `EvVFBRecv(...)`, `EvVFBSnd(...)` and `EvSWCAct(...)` are just names for observable events that should be specified by means of the AUTOSAR Timing Extension: `EvVFBRecv(Curves)` shall represent a $TDEventVariableDataPrototype$ (defined within the AUTOSAR Timing Extension), whose type is set to denote the reception of a data item on the referenced required port named `Curves` with a $SenderReceiverInterface$. This event denotes the observable points in time, when a data item corresponding to the input port `Curves` has been received and is available in the related communication buffer of the AUTOSAR runtime environment (RTE) for the receiving SWC, in this case `EnvironmentModel`. The same kind of event class is used for `EvVFBRecv(Egopose)`. `EvVFBSnd(Roadmodel)` represents the same type of observable event $TDEventVariableDataPrototype$, but this time its attributes are set to denote the points in time a data item is sent on a referenced provided port with a $SenderReceiverInterface$ and the data item is available in the related communication buffer of the RTE, ready to be potentially handed over to the communication stack for being packed into a frame for remote transmission via a bus. Finally, `EvSWCAct(EnvironmentModel.R1)` shall represent a so called $TDEventSwcInternalBehavior$, whose type is set to denote a point in time a referenced runnable enters the state *to be started*. If the runnable is later on mapped to some task of the operating system, then this is the point in time that task gets activated and will be executing the runnable during its own execution.

Since we decided in the refined functional architecture on level B that the under-sampling of curves calculated by the LaneDetection function shall result in history of curves to be buffered, this buffering must be realized by the AUTOSAR model. As discussed before, this can be done by configuring the receiving input port as a bounded FIFO buffer. From the assumption of contract C_{em}^{TLB} we can infer that the size of the FIFO needs to be at least 5 in order to not lose any `Curves` sent by the `LaneDetection` function, provided that all Curves are read by `R1`, and hence removed from the FIFO, which have been present in the FIFO at the time `R1` starts its execution. This conclusion is only valid if and only if the jitter of the input `Curves` is less than $60ms$. Thus, the assumption of a maximum jitter of $33ms$ is sufficient here for drawing this conclusion. The calculation of the minimum size of the input FIFO is based on the observation that during a computation cycle of $105ms$ at most 4 `Curves` might be put into the FIFO, given the arrival rate and jitter of `Curves`. If we now assume that during such a computation cycle `R1` does not use any of them, because it already accessed the FIFO before then during the next execution cycle at most one additional `Curves` input can be written into the FIFO before the `Roadmodel` is guaranteed to be sent.

In fact, the age constraint in the guarantee already forces the implementation of R1 to read all Curves and remove them from the FIFO which have been present in the FIFO at the time R1 is started. If it would not do so and leave elements inside the FIFO the maximum age of Curves relative to the causal computation of the Roadmodel could be violated in the next execution cycle of R1.

TrajectoryPlanning

The SWC implementing the TrajectoryPlanning is linked with the pair of contracts C^{TLB}_{RM-Buf} and $C^{TLB}_{planner}$, which are shown in Figure 11.25 and specify the assumed and guaranteed timing behavior of that SWC. This time, the engineers decide that the runnable R1 of TrajectoryPlanning shall be executed continuously, meaning each time the computation of TrajectoryPoints has finished, a new execution cycle starts sampling the inputs Roadmodel and Egopose. Again, the specification of the activation should be stated as assumption because it is subject to later ECU configuration activities. As one assumption is that R1 will be re-activated once it finishes its computation we also need an assumption that it gets activated at all. This is covered by the first expression of the assumption of $C^{TLB}_{planner}$. The guarantee of $C^{TLB}_{planner}$ specifies a required reaction latency of $[0, 2]ms$ from an activating event until the runnable reads a Roadmodel from its input port. Further, a reaction latency of $[110, 140]ms$ is specified from the point in time the runnable has read a Roadmodel input until it produces its causally related output, the computed list of TrajectoryPoints.

C^{TLB}_{RM-Buf}	A	true.
	G	$<](EvSWCReadRet(Roadmodel), EvVFBRecv(Roadmodel))$ $(EvSWCReadRet(Roadmodel)_i, t_i) := (EvVFBRecv(Roadmodel)_j, t_j) :$ $\nexists(EvVFBRecv(Roadmodel)_k, t_k) \wedge t_j \leq t_k \leq t_i.$ whenever EvSWCRead(Roadmodel) occurs then EvSWCReadRet(Roadmodel) occurs within $[0, 2]ms$.
$C^{TLB}_{planner}$	A	EvSWCAct(TrajectoryPlanning.R1) occurs during $[0, 5]ms$. Reaction(EvVFBSnd(TrajectoryPoints),EvSWCAct(TrajectoryPlanning.R1)) within $[0, 5]ms$.
	G	Reaction(EvSWCAct(TrajectoryPlanning.R1),EvSWCRead(Roadmodel)) within $[0, 2]ms$. Reaction(EvSWCReadRet(Roadmodel),EvVFBSnd(TrajectoryPoints)) within $[110, 140]ms$.

Figure 11.25.: Specification of TrajectoryPlanning (technical architecture level B)

Since we decided in the refined functional architecture on level B that the under-sampling of roadmodels calculated by the EnvironmentModel function shall result in overwriting the last Roadmodel, this buffering must be realized by the AUTOSAR model. In this case, the receiving input port is configured as a shared variable. In order to be consistent with the refined contract of the TrajectoryPlanning function in Section 11.2, we also want to give an example of contracts of Converter Channels in the context of AUTOSAR. As already discussed in Section 11.3.1, AUTOSAR ports of SWCs and their communication configuration correspond to Converter Channels. The contract C^{TLB}_{RM-Buf} specifies the timing behavior of port Roadmodel when a read request of R1 occurs (EvSWCRead(Roadmodel)), until the corresponding API function of the AUTOSAR runtime environment (RTE) returns (EvSWCReadRet(Roadmodel)). Furthermore, a causality function is defined between event EvSWCReadRet(Roadmodel) and event EvVFBRecv(Roadmodel), the latter denoting the points in time a new data item for input port Roadmodel has been received and is available in the corresponding buffer of the RTE ready to be read by runnables of the SWC

`TrajectoryPlanning`.

Discussion: While our goal was to use the classes of observable events defined by the AUTOSAR Timing Extension [6] to the greatest possible extend, we have observed a gap (and possibly also a semantic issue). The timing extensions provide a class of events named $TDEventSwcInternalBehavior$, whose parameters can be set to indicate four different kinds of observable state changes in an AUTOSAR system. The following is quoted verbatim from [6]

- $runnableEntityActivated$: A point in time where the associated RunnableEntity has been activated, which means that it has entered the state "to be started".
- $runnableEntityStarted$: A point in time where the associated RunnableEntity has entered the state "started" after its activation.
- $runnableEntityTerminated$: A point in time where the associated RunnableEntity has terminated and entered the state "suspended".
- $runnableEntityVariableAccess$: A point in time where the associated variable is accessed.

Since we want to refer to the points in time when `R1` reads the input port `Roadmodel`, the kind $runnableEntityVariableAccess$ is the closest candidate. However, it is not precisely stated in [6] which point in time is meant by this event. Is it the point in time the corresponding API of the AUTOSAR RTE is called by a runnable or is it the point in time when such an API call returns? For the contract C^{TLB}_{RM-Buf} specified above we want to distinguish between these two events. While for some use-cases the distance in time might be considered to be negligible, e.g., an uninterrupted read- or write access to a variable, there definitely exist use-cases, where it makes sense to refer to both events. One such use-case is for example when the RTE API called $Rte\_Receive$ (see [5]) is used in blocking manner by a runnable. In this case the input port that is read from must be configured as a bounded FIFO. In case that the FIFO is empty, the execution of a runnable using the $Rte\_Receive$ API will be blocked until either a data item is received and put into the FIFO or a timeout occurs. In such a scenario it is very likely that different reactions would be required from the runnable which should thus be captured in its timing specifications.

Concluding this short excursion, we propose that semantics of $TDEventSwcInternalBehavior$ of kind $runnableEntityVariableAccess$ should be clarified in [6]. Further, a new class of events should be introduced in [6] denoting the missing class of observable events needed for use-cases like outlined above. In the context of this document, we just assume that the two kinds of events are well-defined in [6], and let `EvSWCRead(Roadmodel)` and `EvSWCReadRet(Roadmodel)` represent these events with the meaning discussed above.

DynamicControl

The SWC implementing the `DynamicControl` is linked with the following contract C^{TLB}_{dc} shown in Figure 11.26 specifying its assumed and guaranteed timing behavior. The engineers decided that one runnable R_{LR} is responsible for calculating longitudinal control parameters and another runnable R_{QR} shall compute parameters for lateral control. Both runnables sample the inputs `Egopose` and `TrajectoryPoints` based on which the control parameters are computed. Since parameters for longitudinal control are needed less frequently, different activation rates are specified for R_{LR} and R_{QR}. Again, the activation rates are specified as assumptions.

From contract C^{TLB}_{dc} we can infer that both inputs, `Egopose` and `TrajectoryPoints`, are over-sampled by both runnables of the SWC `DynamicControl`. The input ports can hence be configured as shared variables. The notation $EvSWCStart(\cdots)$ shall represent an

C_{dc}^{TLB}	A	EvVFBRecv(TrajectoryPoints) occurs every $[100, 150]ms$. EvVFBRecv(Egopose) occurs every $10ms$ with jitter $5ms$. EvSWCStart(DynamicControl.R_{LR}) occurs every $5ms$. EvSWCStart(DynamicControl.R_{QR}) occurs every $1ms$.
	G	Reaction(EvSWCStart(DynamicControl.R_{LR}),EvVFBSnd(LR)) within $[500, 500]\mu s$. Reaction(EvSWCStart(DynamicControl.R_{QR}),EvVFBSnd(QR)) within $[500, 500]\mu s$.

Figure 11.26.: Specification of DynamicControl (technical architecture level B)

$TDEventSwcInternalBehavior$, whose type is set to denote a point in time a referenced runnable enters the state *started*. This denotes the point in time the $C - function$ modeled by the runnable is called.

11.3.3. Virtual Integration Testing

After we have designed an AUTOSAR software architecture on the VFB-level and allocated the elements of the functional design to it, the next step is to check whether the specification of the technical architecture refines the one of the functional architecture. As mentioned before, this means verifying that Condition (11.1) holds. Now applying the proof scheme discussed in Section 11.1.5, we observe that the condition does not hold. The problem at this point is that contracts C_{em}^{TLB}, $C_{planner}^{TLB}$ and C_{dc}^{TLB} intentionally state assumptions about the activation behavior of runnables, where no counterpart exists that would guarantee this. This is because the actual activation behavior of runnables is fixed at a later refinement of the AUTOSAR VFB model when configurations of ECUs are modeled. At that stage the runnables of SWCs are mapped to tasks of the operating system (OS) and so called $OSAlarm$s and $OSEvent$s (c.f. specification of AUTOSAR operating system [4]) are configured according to the `RTEEvent`s modeling the activation of runnables.

It should be noted that problems like this are not unusual: Sometimes assumptions need to be stated towards a refined model or a model from a different perspective, which this other model has to guarantee. According to the CBD paradigm we create another contract C_{alloc}^{TLB}, whose guarantee reflects the missing assumed parts. In our case C_{alloc}^{TLB} is as follows:

C_{alloc}^{TLB}	A	$True$.
	G	EvSWCAct(EnvironmentModel.R1) occurs every $100ms$. EvSWCAct(TrajectoryPlanning.R1) occurs during $[0, 5]ms$. Reaction(EvVFBSnd(TrajectoryPoints),EvSWCAct(TrajectoryPlanning.R1)) within $[0, 5]ms$. EvSWCStart(DynamicControl.R_{LR}) occurs every $5ms$. EvSWCStart(DynamicControl.R_{QR}) occurs during $1ms$.

Figure 11.27.: Specification of an allocation contract

Now we can split condition (11.1) into the following simpler conditions:

$$C_{em}^{TLB} \otimes C_{alloc}^{TLB} \preceq C_{em} \tag{11.2}$$

$$C_{dc}^{TLB} \otimes C_{alloc}^{TLB} \preceq C_{dc} \tag{11.3}$$

$$C_{RM-Buf}^{TLB} \preceq C_{RM-Buf} \tag{11.4}$$

$$C_{planner}^{TLB} \otimes C_{alloc}^{TLB} \preceq C_{Planner} \tag{11.5}$$

Now, given that the conditions above are satisfied, the following modification of condition (11.1), including contract C_{alloc}^{TLB}, is satisfied as well.

$$C_{em}^{TLB} \otimes C_{RM-Buf}^{TLB} \otimes C_{planner}^{TLB} \otimes C_{dc}^{TLB} \otimes C_{alloc}^{TLB} \preceq C_{em} \otimes C_{RM-Buf} \otimes C_{Planner} \otimes C_{dc} \tag{11.6}$$

Thus, correctness of this reasoning hinges on a later refinement of the technical architecture fulfilling contract C_{alloc}^{TLB}.

We start the reasoning by observing that the assumptions of the contracts of technical level B are identical to the assumptions of their functional counterparts, modulo the assumed activation behaviors that are however guaranteed by contract C_{alloc}^{TLB}.

Now focus on the guarantees of the contracts involved in condition (11.2). The age constraint of C_{em}^{TLB} is identical to the age constraint of C_{em}. A simple propagation of the assumed activation jitter and the required reaction interval implies that `EnvironmentModel.R1` sends its output every $100ms$ with a jitter of at most $10ms$ which refines the output rate of `Roadmodel` as required by C_{em}. Hence, condition (11.2) holds.

The arguments about the guarantees of the contracts involved in condition (11.3) are similar to the ones for condition (11.2). Again, a simple propagation of the activation rates of $DynamicControl.R_{LR}$ and $DynamicControl.R_{QR}$ implies output rates for $EvVFBSnd(LR)$ and $EvVFBSnd(QR)$ which are identical to the required output rates of LR and QR in contract C_{dc}. The constraints regarding the age of `TrajectoryPoints` based on which the control parameters LR and QR are computed are implied by the assumed activation rates and required reaction latencies of the runnables R_{LR} and R_{QR}. For both runnables the maximum age is less than or equal to $150.5ms$.

Now consider the contracts involved in condition (11.4). The required causality functions are consistent, provided that event `EvSWCRead(Roadmodel)` is mapped to event `readR` of the specification on Functional Level B, event `EvSWCReadRet(Roadmodel)` is mapped to event `R`, and that event `EvVFBRecv(Roadmodel)` is mapped to its functional counterpart `Roadmodel`. The specified timing requirements are identical. Now since - compared to C_{RM-Buf} - the assumption is weakened in C_{RM-Buf}^{TLB} condition (11.4) holds.

The last condition to be checked is (11.5). Again we focus on the guarantee parts, because the assumptions of $C_{planner}^{TLB}$ are discharged by C_{alloc}^{TLB} and $C_{planner}$ has no assumptions. Now we make the following observations:

- The expression Reaction(EvSWCReadRet(Roadmodel),EvVFBSnd(TrajectoryPoints)) within $[110, 140]ms$ implies the expression Reaction(R,Trajectorypoints) within $[100, 148]ms$.
- The expressions in the assumption and guarantee of $C_{planner}^{TLB}$ imply a occurrence of `EvSWCRead(Roadmodel)` every $[110, 147]ms$, which in turn implies the expression ReadR occurs every $[100, 150]ms$.
- The expressions in the assumption and guarantee of $C_{planner}^{TLB}$ imply a reaction from `EvSWCReadRet(Roadmodel)` to `EvSWCRead(Roadmodel)` within $[110, 147]ms$, which in turn implies the expression Reaction(R,ReadR) within $[100, 148]ms$.

Hence, all parts of the guarantee of C_{RM-Buf} are implied by the parts of the guarantee of C^{TLB}_{RM-Buf}, and condition (11.5) holds. Thus, also condition (11.6) must hold, which concludes the reasoning outlined above. Please note that similarly to the discussion in Section 11.1.5 we omitted reasoning about communication latencies between components, which are indicated by contracts $C^{TLB}_{com}(rm)$ and $C^{TLB}_{com}(tp)$ in Figure 11.23. Again, these would specify (simple) reaction times that have to be taken into account in the above reasoning as well.

11.4. Technical Design - Level C

The purpose of this section is to characterize a refinement of the AUTOSAR VFB model, which was discussed in the previous Section 11.3 and is called *ECU Configuration Description* in the AUTOSAR methodology [7]. This step is followed by another step where a so called *System Configuration Description* is created. This description contains a mapping of SWCs to ECUs, thereby inducing a mapping of VFB communication to either ECU-local communication or to network-technology specific communication mechanisms such as CAN frames. Finally, the ECU configuration is created by defining configurations for each needed module of the basic software stack like the operating system, the communication manager, the CAN driver, CAN state manager, CAN transport layer, ECU state manager, basic software mode manager, just to name a few. Generally, the same approach which has been presented in Section 11.3, can also be applied to the basic software modules mentioned above, albeit a very high specification effort. In the AUTOSAR Timing Extension also some classes of observable events are defined, which can be used at this level. However, the focus is clearly on the communication stack and how data items which are sent and received by the SWCs at their ports propagate through the communication stack. Thus, it is indeed very likely that contract-based timing specifications would focus on the application part above the AUTOSAR RTE as shown in Figure 11.28 and possibly also the communication stack and scheduling of runnables.

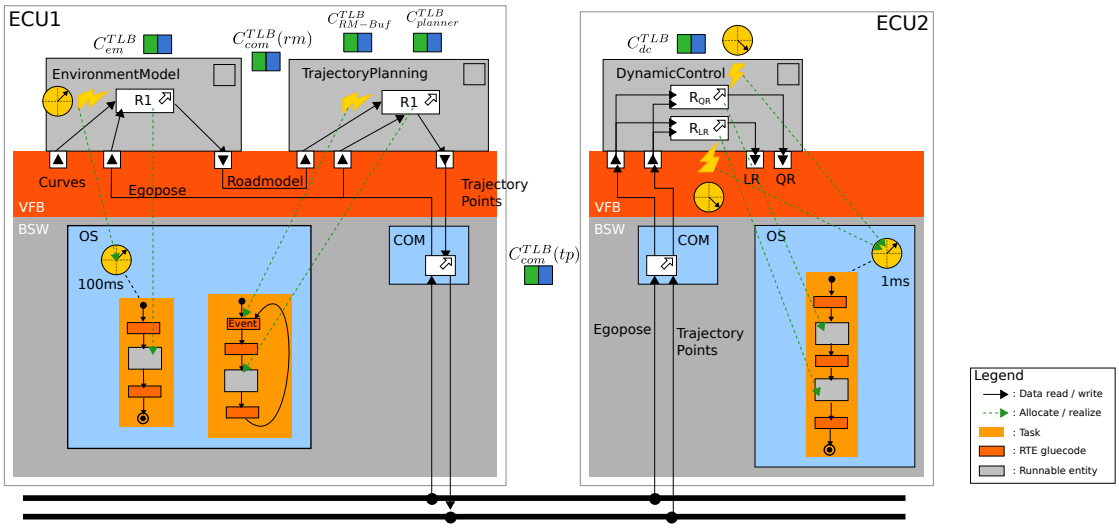


Figure 11.28.: AUTOSAR ECU configurations

In this section we want to focus on the mapping of runnables to tasks of the operating system and how satisfaction of contracts, like specified in the previous section, can be checked. We begin this discussion with the corresponding parts of the configuration of two ECUs `ECU1` and `ECU2`, which is illustrated in Figure 11.28.

The SWCs defined in the previous section contain runnables, which represent the smallest executable code-fragments provided by an SWC. It is now up to the ECU configurator to define a set of operating system tasks that will execute these runnables. During this configuration activity the activation conditions of runnables modeled by `RTEEvent`s have to be considered by defining $OSAlarm$s and $OSEvent$s [4] triggering the tasks in a way such that the activation models defined by the `RTEEvent`s for the runnable executed by a task are satisfied. In the configuration shown in Figure 11.28 two tasks have been configured for `ECU1` and one task for `ECU2`. One task is configured to execute runnable `R1` of SWC `EnvironmentModel`. For this task an $OSAlarm$ is configured that is fired each $100ms$ and has an offset of $0ms$. The other OS task on `ECU1` is configured to execute `R1` of SWC `TrajectoryPlanning` in an infinite loop. An $OSEvent$ is configured on which the execution of the task blocks until the event is sent, which is however done by the executed runnable itself leading to a self-activation and continuous execution, if not preempted by other tasks, of the runnable implementing the calculation of trajectory points. Obviously, this tasks needs to be assigned a lower priority than the first task because otherwise the first task would starve.

For `ECU2` the single OS task executes both runnables of SWC `DynamicControl` in a cyclic fashion. Again, this is achieved by configuring an $OSAlarm$ activating the task. This time, the period of that alarm is set to $1ms$. Such a mapping of time-triggered runnables with different activation frequencies is perfectly legal in AUTOSAR by choosing the greatest common divisor as activation period of the executing task. This will result in the generation of code for the body of the OS task which will call runnable R_{LR} on every $5'th$ activation of the task.

The allocation contract C_{alloc}^{TLB}: As discussed in the previous section, we have stated assumptions about the activation behavior of runnables in the corresponding contracts and added a contract C_{alloc}^{TLB} guaranteeing them. At this point, we can now already conclude that some guarantees of that contract are satisfied only by a simple analysis of the ECU configurations. These are summarized in Figure 11.29. Since the task executing runnable `EnvironmentModel.R1` has an $OSAlarm$ fired each $100ms$, the first expression is fulfilled. The second expression requires a re-activation of `TrajectoryPlanning.R1` once it has sent its output and terminates, which is satisfied by the infinite loop of the executing task.

EvSWCAct(EnvironmentModel.R1) occurs every $100ms$.
Reaction(EvVFBSnd(TrajectoryPoints),EvSWCAct(TrajectoryPlanning.R1)) within $[0, 5]ms$.

Figure 11.29.: Guarantees of C_{alloc}^{TLB} discharged by ECU configurations

11.4.1. Establishing Satisfaction of Contracts

For checking the satisfaction of the remaining parts of the guarantees of C_{alloc}^{TLB} and the other contracts stated in the previous section a lot of methods exist ranging from testing to HIL simulation over (classical) scheduling analysis to approaches based on model-checking. Common to all these methods is the strategy for establishing the satisfaction. As indicated in Appendix C a component (in the AMF sense) satisfies a contract C if, when put into an environment compliant with the assumption of C, it behaves as required by the guarantee of C. Thus, for a testing approach a testbench would be configured to stimulate the component according to the assumption of C while trying to achieve a certain coverage of that assumption. The same holds for simulation-based approaches like the one discussed in Section 11.1.6. Scheduling analyses and model-checking are complete methods testing the entire behavior of an assumption. Regarding scheduling analyses some possible candidates are SymTA/S [40], the Modeling and Analysis Suite for Real-Time Applications (MAST) [37] and the

Real-Time Calculus [84]. Regarding approaches based on model-checking we refer to approaches based on timed-automata analyzed using Uppaal [39], the TIMES tool [2] and RTana$_2$ [82].

In the present design depicted in Figure 11.28, the two ECUs ECU1 and ECU2 would be treated as components. So ECU1 is stimulated according to the assumption of C_{em}^{TLB} and ECU2 is stimulated according to the assumption of C_{dc}^{TLB}. Regarding the non-exhaustive methods like testing and simulation there always exists a risk that not the entire assumption has been covered. In such a case it is undefined (from a specification point of view) what the component does. Thus, also other components connected to it may be stimulated with a behavior not compliant to their assumption. So, monitoring the satisfaction of contracts complements the reasoning at design time as discussed next.

11.4.2. Runtime Observation

Another benefit of considering (timing) specifications consistently and continuously along the design process is that it gives means to runtime observation of systems. Timing specifications can be compiled into executable code as shown above with the SystemC simulation approach. The application of online monitoring specification in the context of safety relevant application has been shown, e.g., in [62, 28].

Online monitoring, however, requires solving several challenges. First, monitoring a system changes the system's behavior. Particularly timing impacts due to interference as well as to the safety architecture itself have to be taken into account. With respect to the proposed design paradigms this should be considered already in early design phases, which is indeed also mandatory in the context of fail-operational design.

Second, physical clocks are typically subject to drift effects and have limited resolution. Hence, observing timing specifications based on the definition of a global time domain may fail not because the specification is violated but because of imprecision the clocks used for observation. Because of such imprecision, relying on physical clocks also may suffers from synchronization issues, which becomes even harder to tackle in scenarios with distributed hardware platforms. While time stamping data helps to provide the information needed to reason locally about timing requirements, it remains open which physical clock is used to obtain the time stamps and how it is related to the clock used for monitoring.

A potential solution of such issues is to constrain allowed clock inaccuracies and synchronization errors between all involved physical clocks and to take measures ensuring that these constraints remain satisfied. Based on such constraints timing specification could be relaxed with respect to clock inaccuracies and synchronization issues. For example, a timing specification that characterizes a maximum delay of, say, $10ms$ would be relaxed to $10.1ms$ if the maximum allowed synchronization gap is $0.1ms$.

Assessing the impact of such a relaxation on the precision of the resulting observers with respect to the timing specification is subject of further work, and will be published elsewhere.

12. Proposed Language Extensions and Tool Support

Our proposed framework has been designed programming language independent. The entry point is a decomposable component model with timing contracts, capable to specify different component interactions through Converter Channel. The actual functional implementation of Converter Channels and components can be done in an arbitrary programming language (in the frame of this project we have used C++ and Matlab/Simulink). Nevertheless, the implementation needs to fulfill its contract. For this reason, we propose to extend programming languages with our timing specification language to assist the programmer to check the actual timing behavior on the targeted hardware platform against the contractually agreed timing specification. This timing check can be done through measurements on the hardware platform[1] and monitors that are derived or generated from the contracts.

In this report, we have not further addressed the observability of time on hardware platforms (i.e., ECUs). One of the identified challenges is to formally guarantee and reason about timing observations that have been made in independent clock domains (i.e., each ECU has its local clock that might not be synchronized with local clocks of other ECUs). Two obvious solutions are: (1) explicit clock synchronization to a globally agreed clock (2) characterization of the maximum clock drift[2] and clock skew[3] to establish an analytical relationship between different local clocks.

The remainder of this chapter is organized as follows. It starts with the presentation of our proposed timing specification language, used in the contract assumptions and guarantees. This is followed by an adoptable SysML extension for the expression of contracts in a hierarchical component model and an approach for the expression of contracts as assertions/monitors in arbitrary programming language. This chapter closes with an identification of missing expressiveness in the AUTOSAR timing specification language, and a brief overview on standardized information exchange between different tools to realize contract traceability.

12.1. Timing Specification Language

Timing specification languages exist in various flavors, such as with the modeling frameworks UML MARTE, AUTOSAR and AMATHEA4public. The present document nonetheless identified some desirable features, such as constraints over causally related events, proper exception handling and mode- as well as data-dependent specifications, in orderto tackle also complex scenarios as needed for ADAS design.

The report proposes a set of text-based natural language specifications enabling light-weight integration into various modeling and programming languages. While the language is deliberately kept simple it exhibits formally well-defined semantics as discussed in Appendix D. Note that the definitions capture only those features used throughout the course of the case study, and has to be further extended

[1] Other approaches like static timing analysis are also possible, but might not be applicable to complex hardware architecture.

[2] Clock drift refers to several related phenomena where a clock does not run at exactly the same rate as a reference clock.

[3] Clock skew (sometimes called timing skew) is a phenomenon in synchronous digital circuit systems (such as computer systems) in which the same sourced clock signal arrives at different components at different times.

in order to support other features mentioned above. The BNF of the proposed language is as follows:

TimeSpec :: *Repetition* | *SingleEvent* | *Reaction* | *Age* | *CausalReaction* | *CausalAge*
Repetition :: *Event* `occurs every` *Interval* [`with jitter` *TimeExpr*]? `'.'`
SingleEvent :: *Event* `occurs within` *Interval* `'.'`
Reaction :: `whenever` *EventExpr* `occurs then` *EventExpr* `occurs within` *Interval* [`once`]? [*Number* `out of` *Number* `times`]? `'.'`
Age :: `whenever` *EventExpr* `occurs then` *EventExpr* `has occurred within` *Interval* [`once`]? [*Number* `out of` *Number* `times`]? `'.'`
CausalReaction :: `Reaction(`*Event* `','` *Event* `) within` *Interval* `'.'`
CausalAge :: `Age(`*Event* `','` *Event* `) within` *Interval* `'.'`
EventExpr :: *Event* | `'('` *Event* [`','` *Event* `')'`]* | `'{'` *Event* [`','` *Event*]* `'}'` | `[>(`*Event* `','` *Event* `)` | `<](`*Event* `','` *Event* `)`
Event :: *Port* | *Port* `'.'` *EventValue*
Port :: *PortName* | *ComponentName* `'.'` *PortName*
Interval :: *TimeExpr* | *Boundary Value* `','` *Value Boundary Unit*
TimeExpr :: *Value Unit*
Boundary :: `'['` | `']'`
Value :: *Number* | *Number* `'.'` *Number*
Unit :: `s` | `ms` | `us` | ...
Number :: `0 .. 9` [`0 .. 9`]*

Based on this language, a desired next step would be automated support for validation and verification tools. This would include automatic observer generation in various languages like C-code for integration into simulations such as the SystemC-based approach discussed in this report, and automata as used for the various verification tools. Another desired target could be, e.g., AUTOSAR timing specifications for integration into other modeling tools, though this would be possible only for those fragments supported by the target languages.

On behalf of a coherent consideration of time, the proposed specification language is based on a single global (physical) time domain. Relation to the various Models of Computations is achieved by translation of the underlying timing models in conjunction with Converter Channels that handle necessary interaction, which together imposes the desired execution semantics to the individual components. Considerations about the relation between the physical time domain and local clocks as done with the UML MARTE framework are only sketched in Part I. This would be subject of further work, as it requires deep investigation into the impacts with respect to coherent consideration of time, as indicated in Section 11.4.2.

12.2. SysML Contract Extensions

SysML does not provide built-in support for contracts, but it supports the specification of constraints. To support the expression of contacts in SysML, CHESS [46] provides a UML profile for contracts. The proposed SysML language extension is shown in Figure 12.1 and the remaining part of this section reproduces the proposed CHESS approach for SysML contract integration.

«Contract» is a stereotype which extends the SysML *ConstraintBlock* entity. Contract allows to aggregate two special kind of UML Constraint, in particular it allows the modeling of the assumption and guarantee properties as Constraint owned by the *ConstraintBlock* itself.

The CHESS profile does not impose any particular language to be used for the specification of the assume and guarantee properties. In our case assume and guarantee properties shall be expressed in

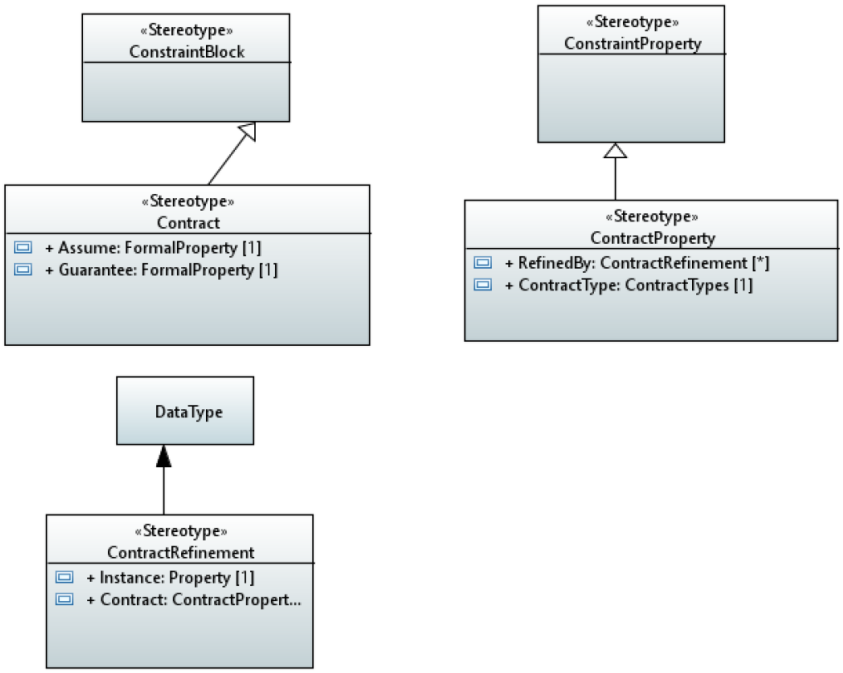


Figure 12.1.: UML profile diagram of the CHESS Contract profile entities [46]

the timing specification as defined in Section 12.1.

A contract specifies restriction on the attributes (i.e., their values) owned by a given component (e.g., component input and output ports): according to the SysML semantics defined for the *ConstraintBlock*, the attributes which are subject of the assume and guarantee constraints specification (i.e., subject of the contract) can also appear as parameters (i.e., attributes) of the Contract. This allows to model contracts in isolation, enabling their reuse, while giving the possibility to bind the contract and its constrained attributes to a given component and attributes at a later stage in the process. This is analogous to how *ConstraintBlocks* works in SysML (in particular by using parametric diagram).

The association between a *«Contract»* and a component is obtained by instantiating a *«Contract-Property»* into the component as required by SysML for the *ConstraintBlock* and the *Constraint-Property* entities.

The *«ContractProperty»* is a stereotype which is derived from the SysML *ConstraintProperty*. It allows to assign Contracts to components. The *ContractType* attribute allows to model contracts assignment.

The *«ContractProperty»* stereotype has a *RefinedBy* attribute that refers the set of *ContractProperties* that decompose it. The usage of the *ContractRefinement* data type allows the modeling of contracts decomposition at the level of the contracts instantiations, i.e., at the level of *ContractProperties* defined for the parent and child components.

Figure 12.2 depicts a CHESS SysML block diagram with the described contact extension. Contracts are associated to components and consist out of assume and guarantees blocks each. The *RefinedBy* property expresses explicitly that the monitoring contract is refined by the sensing contract of the sensor (*sensor.sensing*) and by the monitoring contract of the fdir (*fdir.monitor*)

Figure 12.3 shows the class diagram view of Figure 12.2.

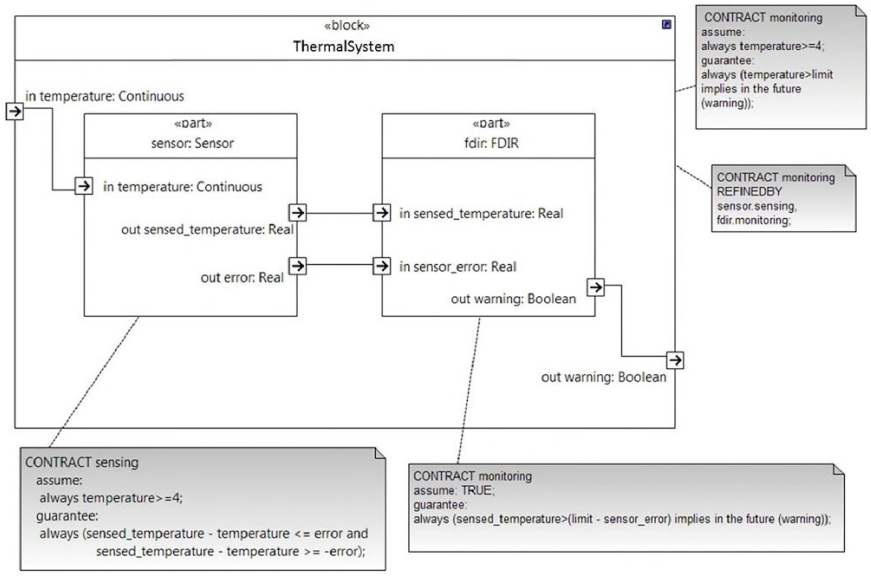


Figure 12.2.: CHESS SysML contract example: block diagram [8]

12.3. Programming Language Extensions for Contracts

The timing specification language introduced in Section 12.1 holds a proposal for languages extensions that can also be added into programming languages. It is the task of future work to integrate such extensions into relevant programming languages. As a first step, it is unlikely to enhance and thereby modify standardized languages or well established tool environments directly. It is more realistic, to target pre-compile/pre-interpretation tools in a first approach (very similar to the integration of assertions into the SystemVerilog language, now called SystemVerilog Assertions (SVA)). As an early assertion specification language it was specified separately from hardware design languages such as Verilog and VHDL. It started in the form of interpreted comments inside the code, but today is a native part of the SystemVerilog standard itself [18]. Listing 10 shows a possible initial approach of integrating our contracts and timing specification language into SystemC and thereby into the C++ language[4]. The shown C++ (SystemC) example is merely a placeholder for any other language of relevance. A future pre-processor extracts and interprets language extension inserted as comments (lines 20), which are framed and parseable by pragma's (lines 19 and 21) to generate additional observer code prior to compilation and then execution of the code.

This way, proposed language extensions become established and undergo a broader review that leads to evaluation of their practical relevance and triggers feedback for enhancements. Once the extensions are widely applied, the next step would be to target standardization. In a special case like SystemC it is an option to create a language extension C++ library. The SystemC Verification Standard (SCV) is an example for such an approach. Including and linking an extensions library to SystemC leave the original standard unchanged, but makes the new language features directly accessible at run time.

```
1 class laneDetectionBClass : public sc_module {
2 public:
3   SC_HAS_PROCESS(laneDetectionBClass);
```

[4]Please remember, that SystemC is not a language, but only a C++ class library

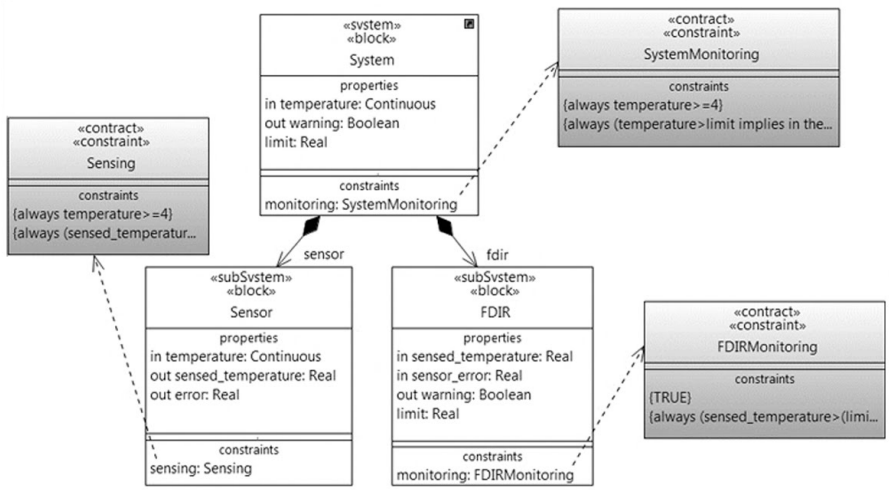


Figure 12.3.: CHESS SysML contract example: class diagram view of Figure 12.2 [8]

```
4
5   laneDetectionBClass(
6     sc_module_name moduleName
7   );
8
9   // Defining a "REACTION" constraint as comment in source code. It will be processed
10  // by a precompiler of future work according to the specified grammar in the Appendix,
11  // which is \eg{}
12  //   Reaction :: whenever EventExpr occurs then EventExpr occurs
13  //                 within Interval [once ]? [ Number out of Number times ]? '.'
14
15  // IO ports
16  sc_in<PROJECT::frameDataType> i_frame;
17  sc_out<PROJECT::curvesDataType> o_curves;
18  /*
19  #assert_pragma_begin //
20   whenever i_frame occurs then o_curves occurs within [0,33]ms 3 out of 4 times. //
21  #assert_pragma_end //
22  */
23
24  // Signals
25  sc_signal<bool> s_updateF;
26
27 private:
28  // The lane detection component and \funcLvl{} B consists
29  // of two sub-components, the converter channel plus the funtional processing
30  // component LD
31  laneDetectionCcClass m_laneDetectionCc;
32  laneDetectionLdClass m_laneDetectionLdClass;
33 };
```

Listing 10: Example of Language Extensions in SystemC

12.4. AUTOSAR Timing Specification Language Extensions

As discussed in Section 11.3, the component concept of the Architecture Modelling Framework (AMF) proposed in Chapter 4 comes with a notion of ports which make certain events of the behavior of the owning component visible outside the component. These events are then referred to by specifications in terms of contracts. For specifying timing contracts in an AUTOSAR model, the AUTOSAR modeling language already provides an extensive list of such events, which is defined in the AUTOSAR Timing Extensions [6]. However, during development of the case-study, we identified a class of observable events that is missing in [6], which was necessary in order to precisely specify the timing behavior at the level of runnable entities (see paragraph Discussion on page 121). Adding support for missing classes of observable events like the identified one to the AUTOSAR Timing Extension would hence be a useful extension of the AUTOSAR modeling language, since then specifications of contracts could refer to events whose semantics is part of the widely accepted AUTOSAR standard.

A further extension with regard to the AUTOSAR Timing Extension concerns causal relations as discussed in Section D. This allows the definition of complex causal dependencies between observable state changes in the system. In particular the specification of causal relations also remove ambiguity in the interpretation of timing requirements. As an example consider timing requirements like reaction or age latencies formulated using the concepts from [6]. In order to specify such a requirement, a so called *Timing Description Event Chain* has to be specified, which basically consists of a pair of observable events, one being the *stimulus* event, the other being the *response* event. A latency requirement would then refer to such an event chain constraining the distance in time between occurrences of its *stimulus* event and *response* event. Regarding the semantics of event chains, the following is quoted verbatim from [6]:

> An event chain describes the causal order for a set of functionally dependent timing events. Each event chain has a well defined stimulus and response, which describe its start and end point. Furthermore, it can be hierarchically decomposed into an arbitrary number of sub-chains, so called *event chain segments*.

However, it is *not* precisely stated which of the multiple *occurrences* of stimulus and response are causally related. In other words, it is *not* precisely stated which of the multiple occurrences of stimulus and response of an event chain are subject to a requirement referring to it. The problem is that inferring this causal relation on event occurrences from the underlying AUTOSAR model is not always possible, since occurrences of a lot of the classes of observable events actually depends on the implementation and is not fixed by the AUTOSAR model. Thus, we think they should be specified explicitly.

The last extension regarding the AUTOSAR Timing Extension directly addresses the distinction of responsibilities in contracts in form of assumptions and guarantees. Currently, [6] provides two roles for timing constraints, *timingGuarantee* and *timingRequirement*. Regarding their semantics, the following is quoted verbatim from [6]:

- $timingGuarantee$: The timing constraints that belong to a specific timing specification in the role of a timing guarantee.
- $timingRequirement$: The timing constraints that belong to a specific timing specification in the role of a timing requirement.

Certainly, none of these two roles for constraints fits the semantics of assumptions of contracts. These are specifications about the behavior of the *environment* of a component, not the behavior of the component itself. That means, based on timing contracts one can reason about whether such assumptions are fulfilled by guarantees of other contracts. Then the implementer of a component can

indeed rely on the behavior of the environment as specified by the assumption. In order to support such a reasoning scheme, adding support for a role *assumption* of timing constraints to the AUTOSAR Timing Extension is necessary.

12.5. Tool integration using IOS and OSLC

The proposed design paradigms and associated compositional framework described a disruptive change w.r.t. to the existing design process and tool landscape. Since we propose a conceptual component model (e.g., in SysML) with formal timing requirements as contracts, links to analysis tools for the VIT and used programming languages (e.g., Matlab/Simulink or C++) are necessary.

A hands-on approach would be to implement a dedicated adapter for the respective analysis tools (VIT, Consistency Analysis, etc, see Figure 12.4) enabling to asses the content of the component model in SysML and, in this particular case, the contract specifications. The drawback of this is the fact that such interfaces have to be implemented for every scenario separately. Alternatively to this point-to-point approach (see left panel in Figure 12.4), one could embed each of these tools into a common semantic framework (right panel of Figure 12.4), thereby reducing the number of necessary adapters to implement.

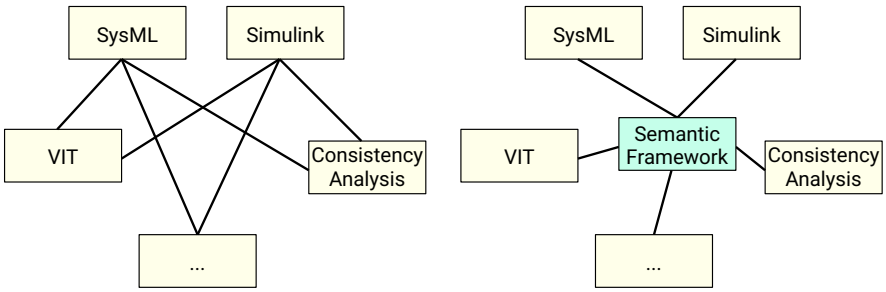


Figure 12.4.: Tool interoperability: Point-to-Point vs. IOS

This kind of interoperability concepts have been investigated in the European research projects CESAR[5], MBAT[6], CRYSTAL[7] and CP-SETIS[8]. Their aim was to set up a Reference Technology Platform (RTP) and Interoperability Specifications (IOS) that provide tools and services, based on OSLC[9] (Open Services for Lifecycle Collaboration). These can be used to build comprehensive engineering tool-chains integrating various design and verification and validation (V&V) tools where one application can seamlessly interact with engineering artifacts that are created and stored within another application. One could, for instance, link a SysML component with associated contracts to a Matlab/Simulink implementation model. An example that demonstrates how this approach has been used to build a tool-chain that can perform a model-based safety analysis is described in [51].

Currently, OFFIS is extending the IOS approach to the engineering of multi-core embedded systems in the AMALTHEA4public[10] project. Results of this work will be made available as an Eclipse [11] project and will be developed to such a degree that it can be integrated with the IBM Jazz[12] platform.

[5]https://artemis-ia.eu/project/1-cesar.html
[6]https://www.mbat-artemis.eu
[7]http://www.crystal-artemis.eu
[8]https://cp-setis.eu/
[9]https://open-services.net/
[10]http://www.amalthea-project.org/
[11]https://eclipse.org/
[12]https://jazz.net/ is a platform for collaborative lifecycle management

13. Summary

Within this part of the report, we have instantiated and exemplified Contract-based Design concepts for a coherent treatment of time across different abstraction and development layers. In particular, this involved the following parts:

- A simplified ADAS design example and design process, covering the main steps from a high-level functional model with top-level timing specifications down to an AUTOSAR implementation model.
- Application of our proposed compositional and hierarchical component framework with support for trace based timing specifications on component ports (using contracts).
- Application of the proposed formal timing specification language that allows the specification of reaction and age constrains, over- and under-sampling, based on causal event relationships.
- An example for a timed executable model (based on SystemC) of the component framework for the integration with existing programming environments and analysis of function dependent timing properties.
- Examples for the specification and implementation of Converter Channels.

For the application of our proposed design paradigms for multi-layer time coherency in the presented design process, we propose the following language and tool extensions and implementation of new languages:

Integration of contracts into SysML Assuming that SysML is the system specification language of choice, contracts should be integrated as dedicated modeling elements to specify timing requirements. A good starting point for this SysML+C (SysML with Contracts) extension is the CHESS[1] Contract profile.

Causal event relationship aware timing property specification language For the expression of the proposed timing specifications a dedicated language is required. This language shall be capable to express event occurrences, reaction and age constraints, restrictions for over- and under-sampling and causal event relationships. An initial proposal for the grammar of this language can be found in Appendix D. Based on this timing property specification language domain specific pattern for easy reuse can be specified.

Timing property specification language into SysML+C contract profile integration For the application of the timing contracts introduced in this project into a SysML environment, a parser for the causal event relationship aware timing property specification language (2) needs to be integrated into the tool environment (1).

Automatic compatibility and virtual integration testing To support an effective working environment for the specification of contracts, automatic compatibility checks (horizontal composition) and virtual integration testing (vertical decomposition) require sufficient tool support. The integration of a verification tool (e.g., OCRA[2] [21]) is necessary.

[1] https://www.polarsys.org/chess/index.html
[2] https://ocra.fbk.eu/

SysML+C to executable functional model transformation Since many timing properties in ADAS/AD are tightly coupled with the functional specification and implementation, an early integration of functional models into the SysML specification model is recommended. To realize this transformation from the specification to the functional implementation, we propose an automatic SysML to SystemC [45] model generation to support integration of functional models (C/C++, Matlab/Simulink) as native C/C++ models. This generation transforms the SysML components, ports and their interconnections as well as the contracts into appropriate SystemC modeling elements. In the case-study of this document only an integration of C/C++ code and automatically generated C/C++ code from a Matlab/Simulink block was outlined. A more generic and well adaptable integration concept for the integration of arbitrary functional models can be established through a well-defined co-simulation interface via FMI [16, 27].

Converter Channel component template library In the proposed approach, Converter Channels are introduced in the functional decomposition phase to bridge component interfaces with different data production rates, thus specifying over and under-sampling interfaces. As described in Part I, Converter Channel also support to bridge time domains (e.g., continuous to discrete time) and data type conversions (e.g., quantization and de-quantization, complex type conversions), similar to DA/AD converters. For the effective usage of Converter Channels in the proposed design methodology, Converter Channel SysML template components and associated C/C++ template library implementations should be supported for recurring conversions.

SysML+C to AUTOSAR model generation Similar to the SysML+C to executable functional model transformation, a SysML+C to AUTOSAR VFB model generation should be supported to enable a transformation into existing AUTOSAR tool environments. This transformation process needs to take care of the SysML+C component to AUTOSAR software component and the contract to AUTOSAR timing extension mapping.

Part III.

Demonstrator for Multi-Layer Time Coherency

14. Introduction

In this part, the proposed modeling, specification and programming concepts are demonstrated by the example of a prototypical and partially realized ADAS. The example is a simple emergency stop assistant. This partial report finally demonstrates how the theoretical approaches developed in Part I and applied in the proposed design processes in Part II can be used in industrially relevant languages and test environments.

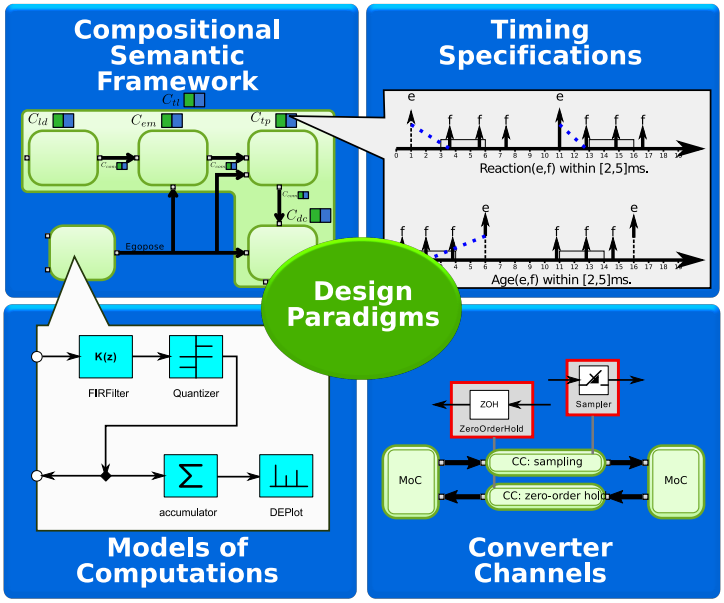


Figure 14.1.: Design Paradigms as introduced in Part I

Part I proposes and discusses four "building blocks" enabling engineers to consider the timing aspect along the design process of automotive ADAS/AD in a coherent and continuous way. These building blocks, or design paradigms (as shown in Figure 14.1) are:

1. **"Compositional Semantic Framework"**, which provides an architectural basis for system design by introducing a generic hierarchical component model with Converter Channels and contracts. The model is intended to be instantiated with existing modeling languages (such as SysML) and tools by defining how to cast the individual modeling artifacts into artifacts of the conceptual model.

2. **"Timing Specifications"** that instantiates Contract-based Design for the timing aspect of the system design. It inherits timing specifications from well established frameworks such as AUTOSAR, and defines extensions where needed in order to enable coherent reasoning about timing within complex scenarios.

3. **"Models of Computation" (MoC)** provide the formal basis for implementing components with well-defined execution semantics. MoCs support integration into different time domains (untimed, continuous time, discrete time and synchronous time) and are instantiated by concrete implementation languages such as Matlab/Simulink, Stateflow and C/C++.

4. **"Converter Channels" (CC)** concerns the interaction between components. This is particularly important when components with different MoCs shall interact with each other, such as components implemented in C/C++ with components implemented using Matlab/Simulink and Stateflow.

Part II aims at effectively applying these design paradigms in existing design processes. To this end, the report discusses a simplified but realistic design process along a case study in which an ADAS is developed. In Part II the case study is deliberately left generic, and does not specify the actual functionality. Instead it shows where the design paradigms apply in the individual design steps (see Figure 14.2), and particularly focuses on tools and languages that may be used.

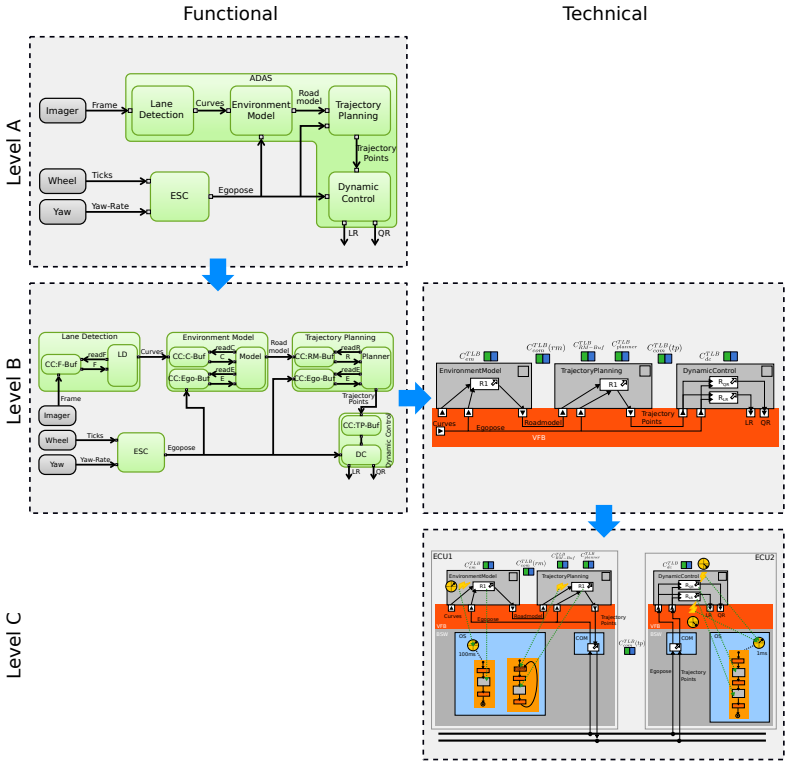


Figure 14.2.: Design Process applied to generic case study as introduced in Part II

The depicted design process consists of four design phases:

The first two phases (Functional Level A and B) cover the functional system design. As the name suggests, these design phases mainly concern system functionality. Hardware/software partitioning is considered only in the sense that general characteristics of sensors are typically known as well as special purpose hardware components for specific tasks, which may influence design decisions.

The latter two phases (Technical Level B and C) consider the technical realization of the system. The technical design exploits AUTOSAR. Although it is expected that future ADAS will involve heterogeneous distributed platforms with a variety of different OS platforms, we argue that a well established standard like AUTOSAR provides a good basis for showing the relation with the conceptual components.

In this part, we are focusing on the first two phases (Functional Level A and B) and apply the proposed paradigms on the simplified emergency stop demonstrator. While Part II already introduced the application of the proposed design paradigms, the main aim of this deliverable is the documentation of the conducted experiments and the actual delivery of the associated code and models.

Table 14.1.: Experiments and Demonstrations

No.	Design Phase	Paradigm and Covered Topics	Language(s)	Aspect	Section
0a	Func. A	Component model with contracts, timing specification	SysML, proposed timing specification language	Timing	15
0b	Func. B	Component model with contracts, timing specification, Converter Channels	SysML, proposed timing specification language	Timing	15
1	Func. A	Contract monitoring, VIT	C++, SystemC	Timing	16.1
2	Func. B	Converter Channels	C++, SystemC	Timing	16.2
3	Func. B	MoC integration, function integration	C/C++, SystemC, Matlab/Simulink, Stateflow	Timing and Function	16.3

Table 14.1 provides an overview of the conducted experiments and demonstrations. It contains information on the covered "design phase" (according to Figure 14.2), the "paradigm and covered topics" (according to Figure 14.1), the used "language(s)" as discussed in Part II, the covered "aspect", the "section" of this report describing the experiments and demonstrations, and where the "code" and supplementary material[1] can be found.

[1]The delivered material per experiment and demonstration is quite different. Where possible, the complete source code, enabling a full reproducibility, is delivered (i.e., for the experiments 0a, 0b, 1 and 2). For the functional demonstrator (experiment 3) of the emergency stop system, a demo video is delivered, since the simulator and the majority of the functional code has been applied from a different project with limited distribution right.

15. SysML Model of Func. Level A & B

In this section we present a SysML model of the functional architectures of the ADAS case study as described in Part II. The discussion focuses on how the ADAS and its contracts have been modeled using SysML. A description of the ADAS itself and its contracts can be found in Part II. Hence, it is assumed that the reader is familiar with it.

15.1. Installation and Configuration of Papyrus

In order to create the SysML model of the ADAS, the tool Papyrus was used. Papyrus was chosen because it is freely available. The first step is to download Papyrus, which is available under the following URL: `https://eclipse.org/papyrus/`. Papyrus Neon release (2.0.2) was used for creating the SysML models. The downloaded file is an archive that must be extracted. Executing the file `Papyrus/papyrus` starts the Papyrus modeling environment. The downloaded Papyrus version comes with UML modeling capabilities and the possibility to apply profiles like SysML on top of such models. For modeling the case-study we used an implementation of SysML as a domain specific modeling language, which is available as an extension of Papyrus. In order to install this extension, start the Papyrus modeling environment and navigate to the menu entry `Help` → `Install Papyrus Additional Components`. After the components have been loaded an entry `SysML` should be available for installation under the category `Languages`. Note that you have to make sure that the checkbox named `Experimental` is enabled. Otherwise the SysML component is not shown. Starting the installation launches a wizard, where all features of the SysML component must be selected. Installing the SysML component may take a couple of minutes. The installation should be finished by restarting the Papyrus modeling environment.

15.2. SysML Modeling Approach

In Part II, the functional view of the ADAS is described by means of a functional design with two levels of decomposition. For this functional design a SysML model has been created. The SysML model makes use of the following SysML concepts and diagrams:

- Blocks, Properties, PartAssociations, ProxyPorts, InterfaceBlocks, FlowProperties, BindingConnectors, Requirements and Satisfy Dependencies.
- Block Definition Diagrams (BDDs), Internal Block Diagrams (IBDs) and Requirement Diagrams.

The semantics of these SysML concepts is explained in detail in the SysML specification [63] and is hence not repeated here. SysML defines two kinds of ports: full ports and proxy ports. Both provide ways of defining the boundary of the owning block. While full ports provide their own behavior, proxy ports expose some of the behavior of the owning block. Hence their semantics conforms to the semantics of ports of the framework proposed in Section 4.1.2 and they are used exclusively in the SysML model. Without confusion we use the term port when we means a proxy port.

Since the description of the ADAS in Part II does not mention any data types of the values visible at the ports of the involved components, a single InterfaceBlock `Event` has been modeled containing

a single FlowProperty `EventFlow` whose direction attribute is set to `out`. Each port in the model is typed by this InterfaceBlock. In case an input port is modeled its attibute `isConjugated` is set to `true`. If enabled this attribute reverses the direction of all flows of that port.

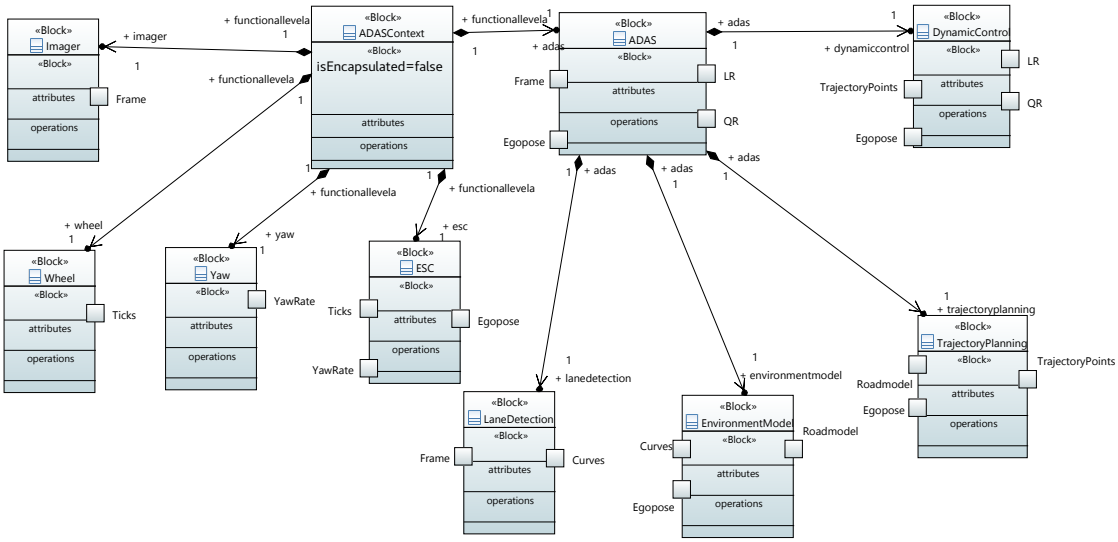


Figure 15.1.: Block Definition Diagram of Functional Design - Level A

The modeling approach was to make use of BDDs in order to model a block with its ports and parts. For each component of the ADAS a corresponding block has been modeled. Figure 15.1 shows the resulting BDD of the context of the ADAS and its breakdown into parts. The block `ADASContext` consists of an `ADAS` block and blocks providing inputs to the ADAS. The ADAS itself consists of a `LaneDetection`, an `EnvironmentModel`, a `TrajectoryPlanning` and a `DynamicControl`.

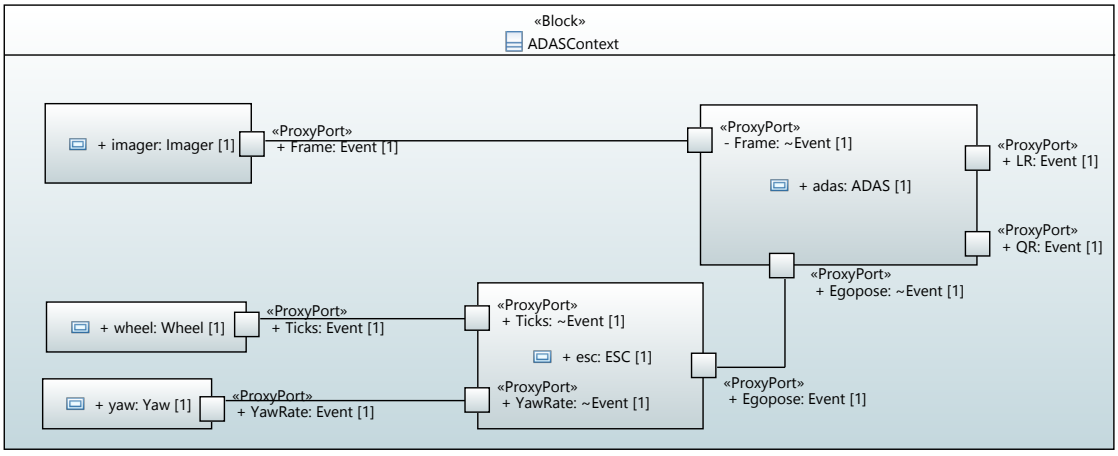


Figure 15.2.: Internal Block Diagram of Functional Design - Level A

After modeling the reuseable blocks, their ports and their part-relationships, IBDs have been created for each block containing a part. The ports are linked by means of BindingConnectors, corresponding to the interconnection of the ADAS components as presented in Part II. As an example Figure 15.2

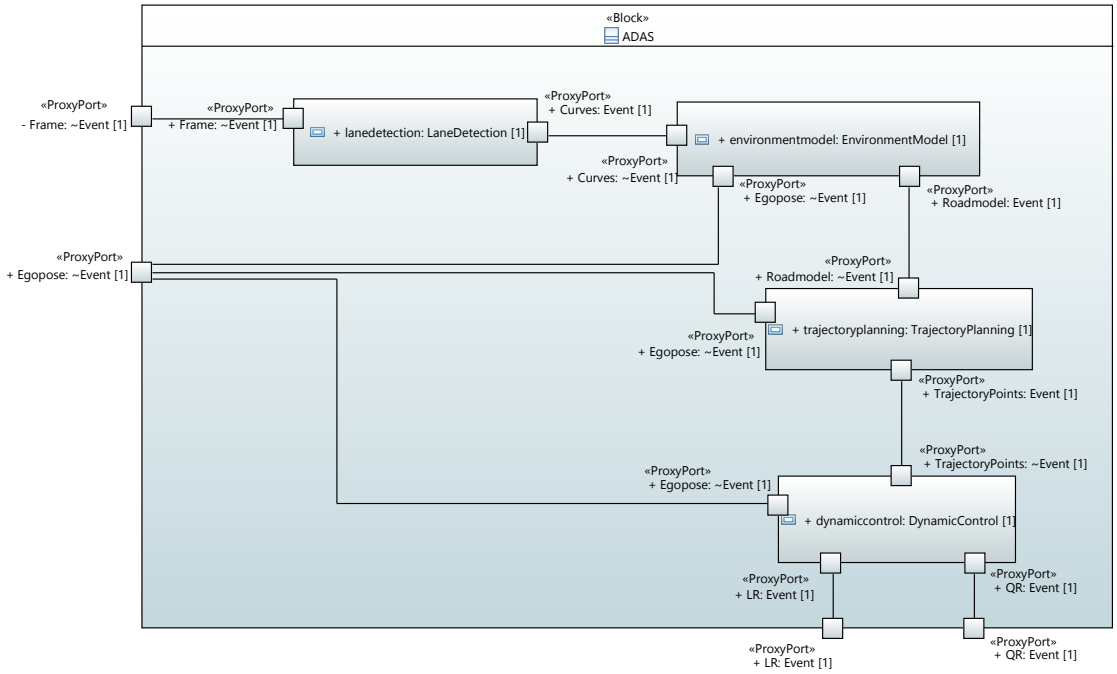


Figure 15.3.: Internal Block Diagram of ADAS in Functional Design - Level A

shows the IBD of the block `ADASContext`, where the ports of blocks providing inputs to the ADAS are connected with the ports of the `ADAS`. The internal structure of the `ADAS` block, in turn, is again modeled by means of an IBD as depicted in Figure 15.3.

Having modeled the components as blocks, with their decomposition and interconnection, the next step is to model the contracts linked to them. For this the SysML Requirement concept has been used. Using a Requirement Diagram each block is linked by means of a Satisfy Dependency to the Requirement representing the contract that the behavior of the block shall satisfy. The `text` attribute of a Requirement has been used to capture the textual specification of the assumption and the guarantee of a contract. Figure 15.4 depicts the requirement diagram showing the contract of the `ADAS` block and the contracts of the blocks typing its parts. Note that in SysML it would also be possible to specify a Satisfy Dependency between a requirement and a particular part of a block. However, the semantic framework from Chapter 4 emphasizes reuse of components and the assumptions of contracts already define the allowed usage context. So it makes sense to link the contracts modeled as Requirements to blocks, thereby requiring the behavior to satisfy the contract in all usage contexts.

Corresponding to the Functional Design on Level B from Part II, decompositions of the blocks `LaneDetection`, `EnvironmentModel` and `TrajectoryPlanning` have been modeled in the SysML model. In the following we pick the `LaneDetection` block to illustrate the principle. A Block Definition Diagram similar to the one shown in Figure 15.1 has been created, which models the decomposition of `LaneDetection` into its parts. One part is a buffer for the frames recevied by the Lane Detection, the other part is a block representing the computation of curves based on the buffered frames. The IBD of the `LaneDetection` block is depicted in Figure 15.5. The contract of the `LaneDetection` block is decomposed according to the decomposition of the block. The resulting requirement diagram is shown in Figure 15.6. Again the contracts are linked to the blocks typing the parts of the `LaneDetection` block.

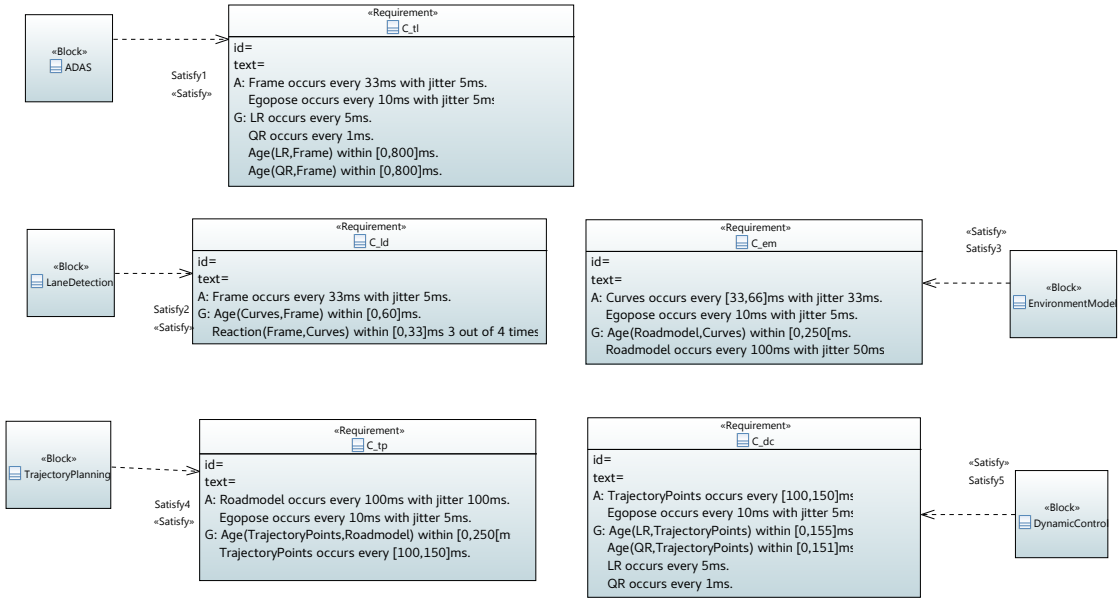


Figure 15.4.: Requirement Diagram of Functional Design - Level A

It should be noted that we intentionally did not make use of the SysML concept of refinement links. This concept is intended to express a refinement relation on requirements. In the SysML model this is however not necessary, because the decomposition and interconnection of a block already implies that its contract shall be refined by the set of contracts linked to the blocks typing the parts of the block.

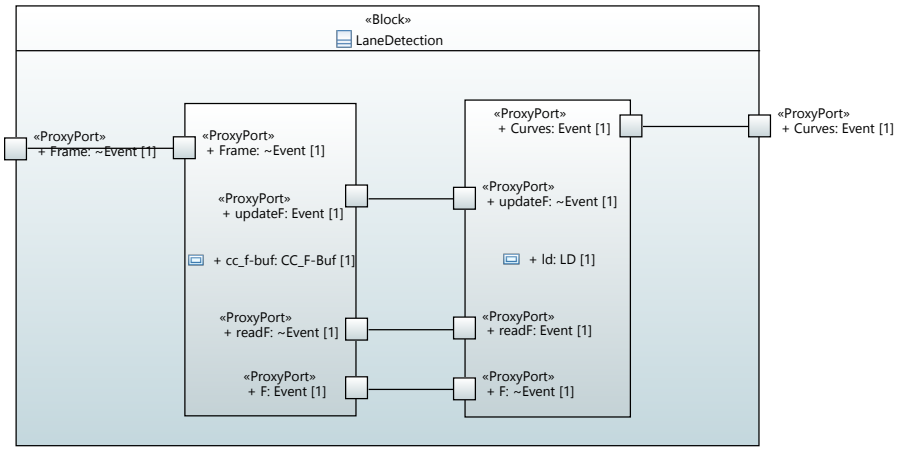


Figure 15.5.: Internal Block Diagram of LaneDetection in Functional Design - Level B

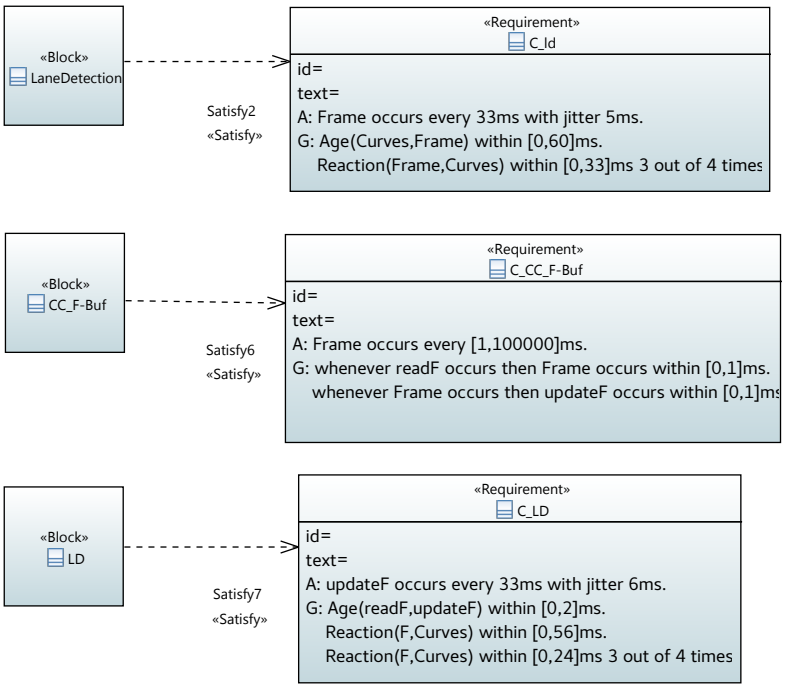


Figure 15.6.: Requirement Diagram of LaneDetection in Functional Design - Level B

16. SystemC Model of Funct. Level A & B

Part I exemplified the application of four proposed design paradigms along a simplified design process for a generic ADAS case study. The text discusses the use of a Compositional Semantic Framework together with the design paradigms Timing Specifications, Models of Computation and Converter Channels. Part II is thorough in it's depth of detail, but only started to demonstrate how to proceed practically. In Chapter 11.1.6 we reasoned and decided to chose a simulation based approach over formal methods which don't scale very well. We chose SystemC as a simulation base of particular industrial relevance and started to describe how Simulation-based VIT can be accomplished by use of SystemC and the proposed paradigms. In the scope of Part III the started work was continued and the case study of an ADAS was put into C++ code in a somewhat tedious manual effort. It must be stated again, that for sure it has to be the focus of future work to develop tools that automate the synthesis of VIT simulation models from specified contracts directly.

The following three subchapters refer to three experiments that where accomplished in the scope of Part III. While the first experiment implements timed events at the Functional Level A, the next two are positioned at Functional Level B integrating Converter Channels and finally a co-simulation with a driving simulator. Executable and source code are available for the MULTIC Consortium for the first two experiments. The third experiment bases on the commercial SILAB simulator offered by Würzburger Institut für Verkehrswissenschaften GmbH (wivw). The delivered code requires the download of SystemC 2.3.1 and SCV 2.0.0 which is freely available at http://accellera.org/downloads/standards/systemc. The MULTIC base directory contains a Microsoft Visual Studio 2015 solution (laneAssist.sln) and project (laneAssist.vcxproj) file (refer to Figure 16.1). Minor modifications to these files will be necessary to adapt include- and library-paths to the user's installation of SystemC and SCV. This is no obstacle for an average user of Microsoft Visual Studio.

The code is not restricted to a dedicated operating system. After spending minor effort it is easily possible to create according Makefiles to compile the experiments with e.g., gcc for Unix/Linux. It is an option to deliver such a Makefile or even an installation as virtual machine on request.

16.1. Level A: Only Timed Events

The following description counts for the Windows operating system. Prerequisite to the execution of this experiment is the download and installation of SystemC and SCV. Opening the MULTIC file structure and double-clicking the solution file laneAssist.sln will open Microsoft Visual Studio 2015 which should be installed on the machine (refer to Figure 16.1). The SystemC and SCV projects are linked inside the solution laneAssist.sln which will most likely cause for a project-not-found-message for SystemC and SCV since the installation path of these depend on the user's individual setup. This can quickly be fixed by deleting the unfound SystemC and SCV project from the solution. It is not necessary to add the real location of the projects to the solution again, but this is encouraged as navigating the code all the way into SystemC and SCV is enabled this way. It is a requirement though to correct the include and library path to the SystemC and SCV installation in the laneAssist.vcxproj as part of the solution to enable compile and linkage (compare with Figure 16.2 and Figure 16.3).

The project resembles the architecture as depicted in the upper part of Figure 16.4. The boxes left to right resemble the Imager, Lane Detection (LD), Environment Model (EM), Trajectory Planning

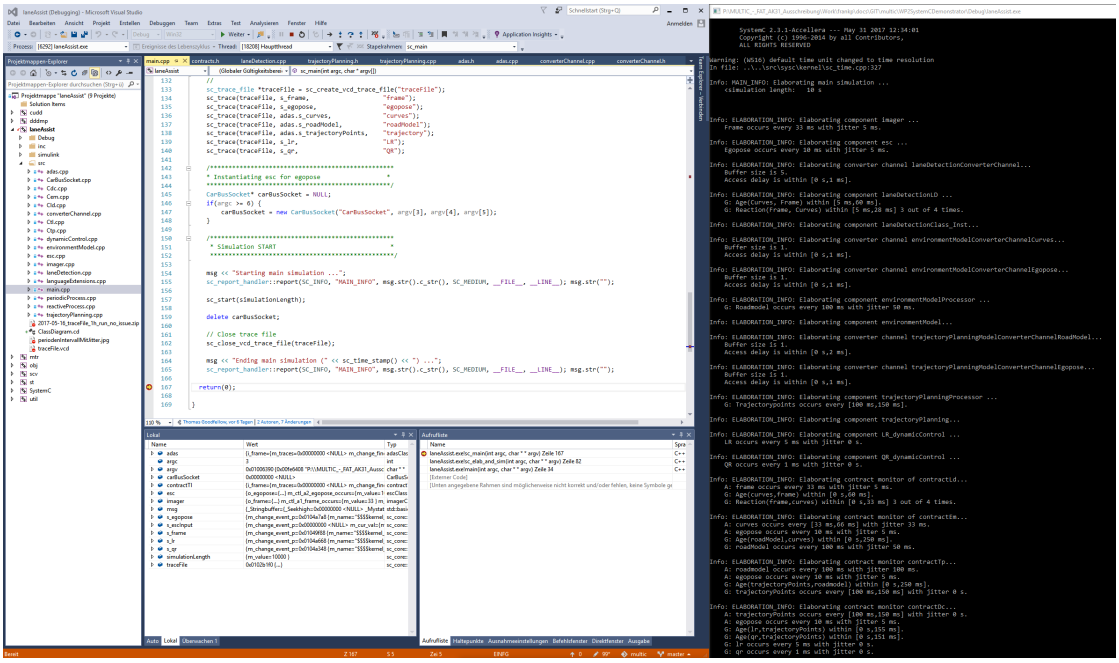


Figure 16.1.: Project of the experiments opened in Visual Studio 2015

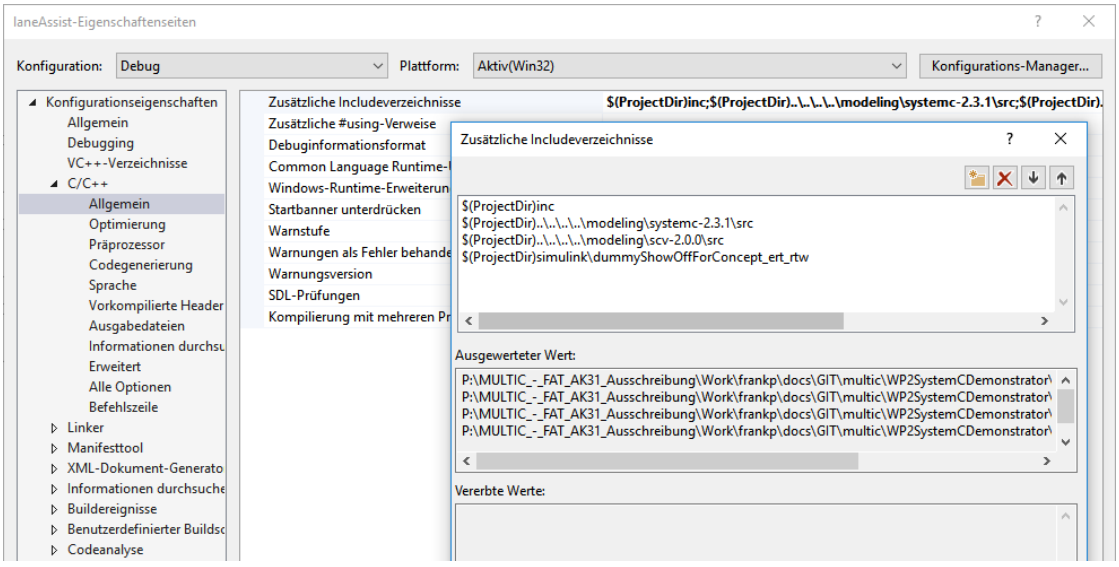


Figure 16.2.: Adjusting SystemC and SCV include path in project

(TP) and Dynamic Control (DC). The last four are encapsulated inside the hierarchy of the ADAS entity, together with their parallel instantiated individual monitors. The ADAS entity itself has its own monitor in parallel. The picture is intentionally left incomplete not to overload it. Next to the Imager generating new-frame-events at the specified rate, the implementation also contains an egopose-event-generator. The latter has been left out from the figure together with its connections to

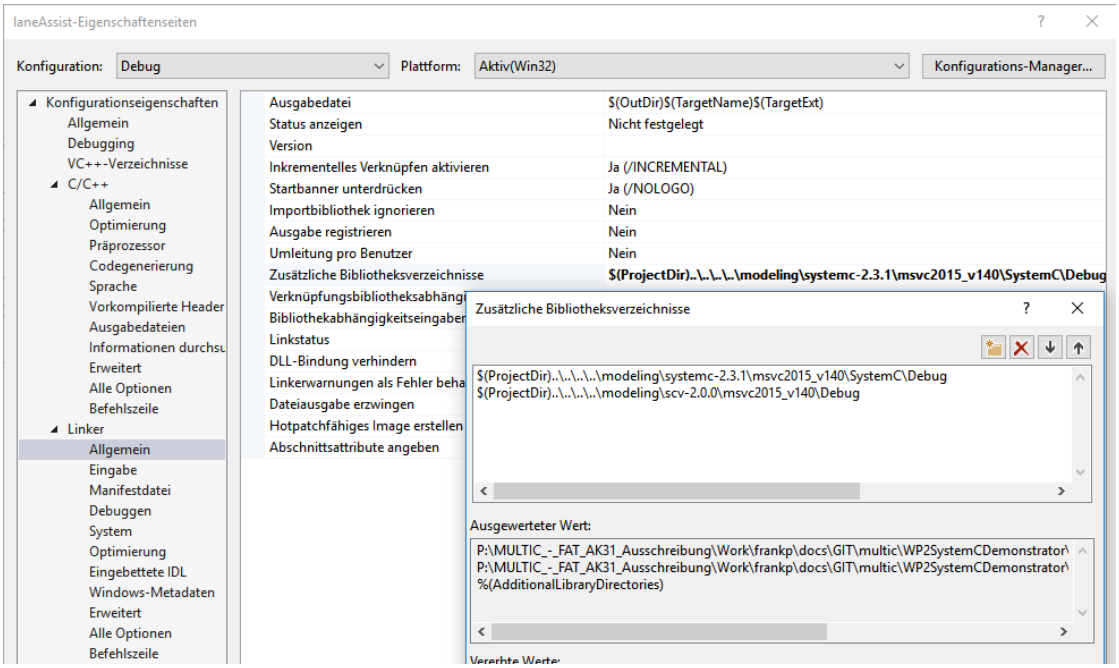


Figure 16.3.: Adjusting SystemC and SCV library path in project

LD, EM, TP, DC, C_{tl}, C_{em}, C_{tp} and C_{dc} to keep it readable. The lower part of Figure 16.4 shows a screen shot of a simulation run. The start up trace shows the elaboration messages of the implemented SystemC components. For some but not all of them the figure holds the linkage to the architecture depiction above. Again, this was done to not overload the figure. The missing correlations of messages to components should be clear.

The executable model accepts two parameters to control a simulation run (compare with Figure 16.5). Parameter one defines the duration of simulated time in seconds. The second parameter defines the verbosity of messages. The simulation model reuses the SystemC specified verbosity: (NONE=0,LOW=100,MEDIUM=200,HIGH=300,FULL=400,DEBUG=500).

Running a simulation creates a Value Change Dump (VCD) file in the MULTIC base directory named traceFile.vcd. The VCD can be inspected with a viewer of choice. Figure 16.6 shows a screenshot of the opened traceFile.vcd using the Eclipse plugin Impulse Waveform Viewer.

When the simulation is set up and run with the original specification of Part II the monitor of the Environment Model is triggered showing the message of Figure 16.7. During the simulation of the 13th second the EM-monitor detects an early arrival error of a curves-event from LD. The specification of Part II defines the first Assumption for EM as "Curves occurs every [33,66] ms with jitter 33 ms" which on first glance should not cause for an issue in the monitored case. With a minimum period of 33 ms and a jitter that can reach up to 33 ms it is possible to have two curves events in a row with 0 time in between (first event at 33 ms period plus 33 ms jitter and second event at 66 ms period and no jitter; both events occur at 66 ms with 0 time in between). This can happen for some event traces, but is not generally legal.

Have a look at Figure 16.8. To understand the specified assumption that contains a period defined as an interval plus an extra jitter, it needs to be understood that most of the time one cannot differentiate from a single event between the randomness of the period interval or the randomness of the jitter. But it is possible to receive information on the jitter, if two events at time t_0 and t_1 are

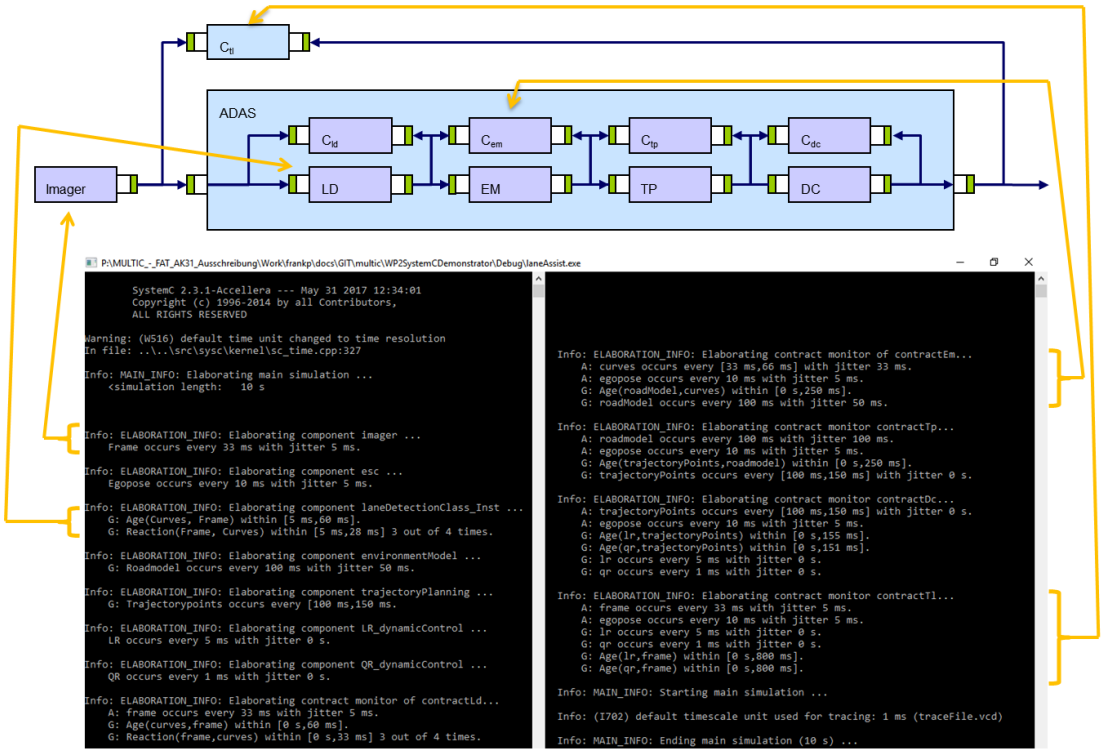


Figure 16.4.: SystemC Architecture of Experiment 1

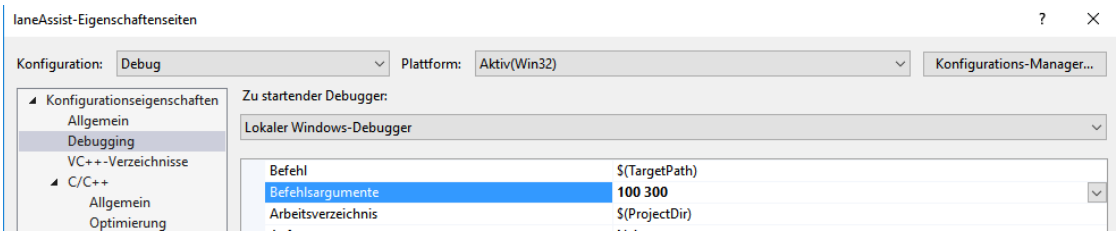


Figure 16.5.: Two Parameters to control VIT Simulation: Duration in Seconds and Message Versatility

examined together. This is what is depicted in Figure 16.8.

Looking at t_0 we cannot make any assumption on the jitter that this event contains exactly. It might be either none or maximum. Consequently we have to assume that the jitter of t_0 is somewhere in the interval of J_0=[0,33] ms. Ignoring this jitter t_1 would have to arrive in the period interval 33 ms to 66 ms later. In the figure we marked this interval as dotted lines P_{t1}^{-} and P_{t1}^{+}. If t_0 has a jitter included, then it would get closer to this interval, or looked at it from the point of view of the next event: P_{t1}^{-} and P_{t1}^{+} move to the left towards t_0. The larger t_0's jitter is the closer the interval moves towards t_0. Using the jitter uncertainty of the previous event we receive a new period interval for the next event which reaches from P_{t1}^{--} to P_{t1}^{++}.

With this we can observe five possible situations a) to e) depicted in Figure 16.8.

Case a) An event arriving before P_{t1}^{--} is too early to be legal. Even if the jitter of t_1 is zero, the

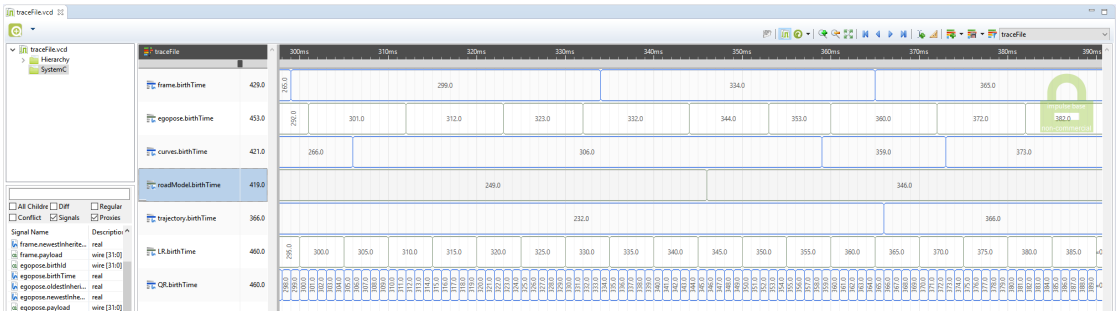


Figure 16.6.: View of VCD File with Impulse Plugin in Eclipse

```
Error: RUNTIME_INFO: 12543 ms Monitor of A1 triggered by new curves event.
VIOLATION of:     curves occurs every [33 ms,66 ms] with jitter 33 ms.
Event arrived early.
Expected window of arrival [12544 ms,12623 ms].
  last detected: 12524 ms
  m_P_now: [[12544 ms,12557 ms],[12577 ms,12590 ms]]
  estimated jitter was: [0 s,13 ms]
In file: p:\multic_-_fat_ak31_ausschreibung\work\frankp\docs\git\multic\wp2systemcdemonstrator\src\languageextensions.cpp:179
In process: adas.contractEm.A1.monitor @ 12543 ms
```

Figure 16.7.: Issue Detected by the Functional Model Level A

event would still be before its period interval. The assumption is violated with the event arriving too early.

Case b) If t_1 arrives exactly at time P_{t1}^{--}, then this is the earliest allowed time according to the periodic interval specification. Any extra jitter of t_1 would move it to the right. So for the border of this case we know exactly that t_1 has a jitter of 0 ms which is a valuable information again for the next, third event and so on.

Case c) Once we leave the bounds of P_{t1}^{--} plus the maximum jitter, we can no longer gain any extra information on t_1's jitter and have to assume the complete interval as for t_0.

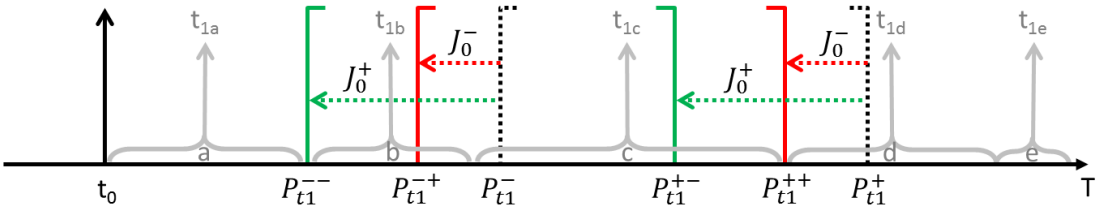
Case d) If the next event reaches beyond P_{t1}^{++}, then we are outside the periodic interval. To arrive here legally, t_1 requires extra jitter. From this we can conclude, that the jitter of t_1 cannot be 0 ms.

Case e) If t_1 gets beyond P_{t1}^{++} more than the allowed maximum jitter, then the assumption is violated with the event arriving too late.

The error message of Figure 16.7 is caused by a case a) situation. The derived jitter for the previous curves event at 12524 ms is in the interval [0,13] ms. The begin of the next period interval therefore is 12524 ms + 33 ms - 13 ms = 12544 ms. The end of the interval is computed accordingly. With 12543 ms the current curves event arrived 1 ms too early to be acceptable for the assumption monitor.

Examining this violation revealed the combination of two factors to cause this. Firstly, the jitter of 5 ms on frame events can cause for inter frame arrivals of less then 33 ms. If f_0 has a maximum jitter of 5 ms while the next frame f_1 has 0 ms jitter, then the distance between the two is only 28 ms. This is to be expected from a jitter. Another scenario is, where f_1 has a jitter of 4 ms. The distance between the two is 32 ms and thereby below 33 ms. If we look at the jitter of f_2 to be 3 ms, for f_3 to be 2 ms, and so on, we observe a sequence of frames with only 32 ms apart. Conclusion: several frames in a row can arrive faster then 33 ms. The second factor is the definition of the LD constraint "Reaction(Frame,Curves) within [0,33] ms 3 out of 4 times." This includes a possible computational

time of 0 ms. So curves alike can arrive in a sequence of less than 33 ms apart, and this is where the EM-monitor is triggered. The solution is, to no longer allow for a zero delay in the LD constraint to fix this.



$$\text{Period: } P = [P^-, P^+] = [33{,}66] \text{ ms}$$
$$\text{Jitter: } J = 33 \text{ ms}$$
$$J_0 = [J_0^-, J_0^+] = [0{,}33] \text{ ms}$$

$$P_{t1}^- = [P_{t1}^{--}, P_{t1}^{-+}] = [t_0 + P^- - J_0^+, t_0 + P^- - J_0^-]$$
$$P_{t1}^+ = [P_{t1}^{+-}, P_{t1}^{++}] = [t_0 + P^+ - J_0^+, t_0 + P^+ - J_0^-]$$

a) $P_{t1}^{--} > t_1 \Rightarrow$ „violation – event too early!“
b) $P_{t1}^{--} + J \geq t_1 \geq P_{t1}^{--} \Rightarrow J_1 = [0, t_1 - P_{t1}^{--}]$
c) $P_{t1}^{++} > t_1 \geq P_{t1}^{--} + J \Rightarrow J_1 = [0, J]$
d) $P_{t1}^{++} + J \geq t_1 \geq P_{t1}^{++} \Rightarrow J_1 = [t_1 - P_{t1}^{++}, J]$
e) $t_1 \geq P_{t1}^{++} + J \Rightarrow$ „violation – event too late!“

Figure 16.8.: Explaining Periodic Interval with Jitter

In the model this is fixed by adding/subtracting the frames jitter to/from the minimum/maximum processing time of Lane Detection. This way, curves will arrive 5 ms after a frame event at the earliest. To enable the fix the project needs to be recompiled with define “FIX1” declared (compare with Figure 16.9). Another run of the simulated VIT will now finish without any alarm of a monitor.

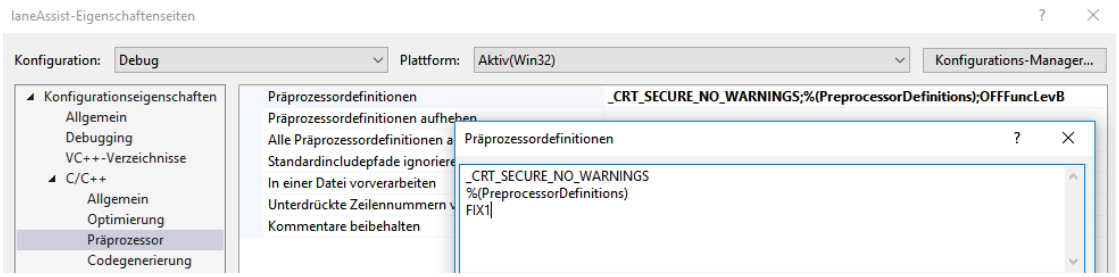


Figure 16.9.: Enabling FIX1

Concluding this chapter: Finding and fixing the above issue in the specification is a very good demonstration of what VIT can achieve. It needs to be remembered that the experiment bases on a simplified design process of a generic ADAS case study. Contracts where specified together with only one artificial MULTIC application in mind. As a result, the contracts of the individual components appear to the reader to be obviously compatible by design. Still, the VIT simulation detects trace combinations specified by the contracts that do not comply with the designer’s intend.

In the general industrial case, components and their contracts are reused from different contexts and different projects. A manual review of their compatibility would be extremely tedious to impossible. VIT and its monitors are a powerful tool to find incompatibilities before behavioral implementation and deployment to hardware begin. Future work will have to provide automatic synthesis of a VIT from its contracts.

16.2. Level B: Timed Events and Converter Channels with Channel Semantics

The previous chapter documents the Simulation-based VIT for the MULTIC generic ADAS case study at Functional Level A. The same source code project and usage accounts for simulation at Functional Level B. Please refer to Figure 16.10. With the preprocessor define "FuncLevB" and a recompile of the project, Converter Channels with channel semantics are instantiated in the model. The architecture changes accordingly from Figure 16.4 to the new architecture of Figure 16.11. Again, for a more simple to view layout of the figure, we decided to leave out components and communication for the Egopose event-paths.

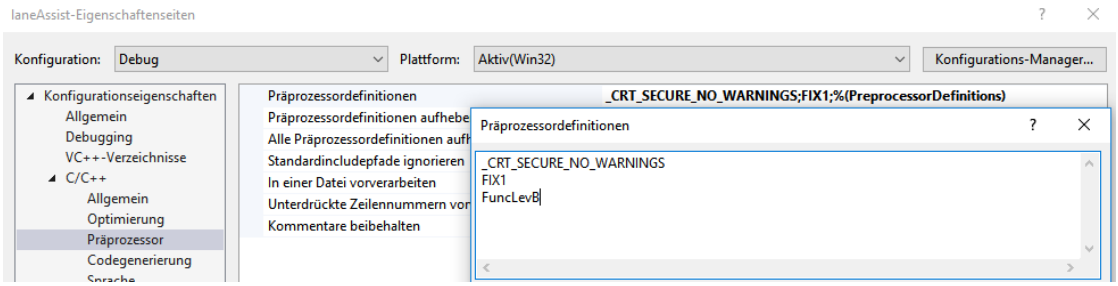


Figure 16.10.: Enabling Level B Simulation

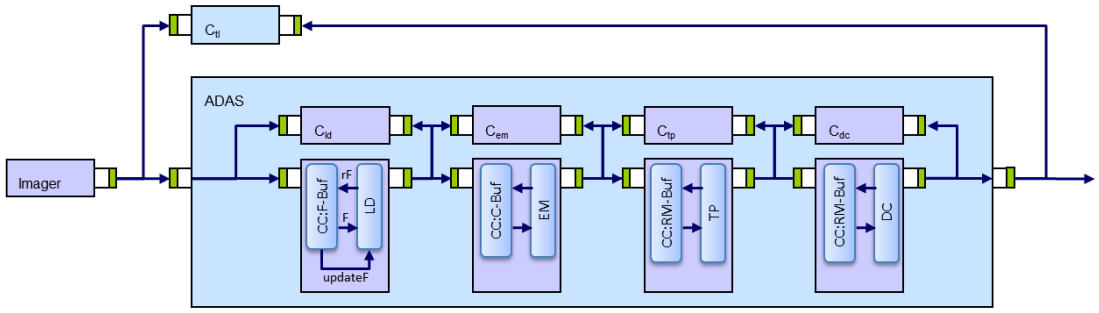


Figure 16.11.: SystemC Architecture of Experiment 2

The elaboration messages of the simulation now show additional information on further components e.g., "*Elaborating converter channel laneDetectionConverterChannel... Buffer size is 5. Access delay is within [0 s,1 ms]. Update delay is within [0 s,1 ms]*". This message is dedicated to the implementation of what is specified in Figure 11.12, Figure 11.13 and Figure 11.14. The contracts name a ring-buffer "CC:F-Buf" that "corresponds to the intended history, and where all data can be accessed by the

function". The contract defines an update delay of up to 1 ms on the arrival of a new frame at the input of the channel, and the same delay for a read access to the channel. The timed events of the Level B model show this specification behavior accordingly. Refer to Figure 16.12. It shows a screen-shot of the Eclipse Impulse Plugin to visualize the waveform traces created during a simulation run of the model.

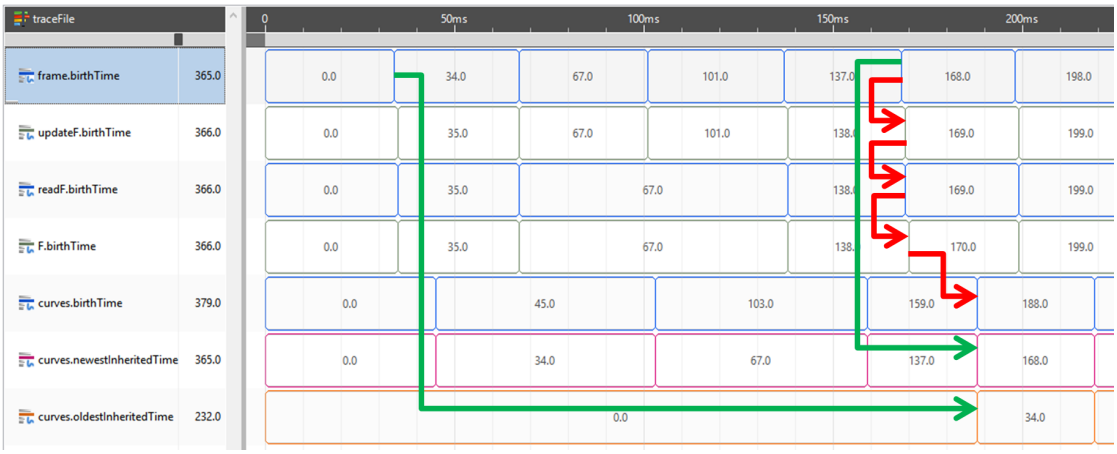


Figure 16.12.: Timed Events for the Lane Detection's Frame Buffer

The first line of Figure 16.12 holds the change events on the Imager component which periodically delivers new frames. According to the specification, every 33 ms arrival of new frames can be followed by some jitter: 33 ms (+1 ms jitter), 66 ms (+1 ms jitter), 99 ms (+2 ms jitter), 132 ms (+5 ms jitter), etc. Let's have a more detailed look at the new frame event that arrives at time 168 ms. Part II states in Figure 11.12: "whenever Frame occurs then udpateF occurs within [0, 1]ms." The model shows valid behavior in the second row of Figure 16.12. The frame arriving at 168 ms triggers the updateF event 1 ms later at 169 ms. The delay is not always 1 ms but specified as an interval from 0 ms to 1 ms, e.g., the updateF event at 101 ms occurred directly as a reaction to the new frame at the same 101 ms. Without any delay the updateF event at 169 ms triggers a Lane Detection's read access to the Converter Channel (third row of Figure 16.12). Again according to the Part II specification "whenever readF occurs then F occurs within [0, 1]ms." the read-request receives a response (fourth row) after 1 ms at time 170 ms.

From this point on we experience the same behavior as before at Functional Level A of the previous chapter. The frames F are available to be processed. The event model of the Lane Detection processor is specified as "*Reaction(Frame,Curves) within [0, 33]ms 3 out of 4 times.*" which we see to be fulfilled in row five. Figure 16.13 highlights the three out of four more behavior clearly. With a delay of 18 ms a new curve event is released at 188 ms. The inherited times (green arrows in Figure 16.12) show that this event bases on the newest frame event of time 168 ms (20 ms into the past) but also five frames in the past because of the buffer size of five of time 34 ms (154 ms into the past). For sure, the evaluation of such a short period of simulated time is no proof for correctness. It is a tool to find issues in the (executable) specification.

It needs to be pointed out, that all previously discussed contracts and monitors of the previous chapter on Functional Level A remained active throughout the entire second experiment at Functional Level B. Looking at Figure 16.12 it is apparent that the specification refinement causes for a variation in the timing behavior. Formally or even manually it would be hard to impossible to asses, if the refined system model is still within the specification. The simulation run shows that the proof obligation of

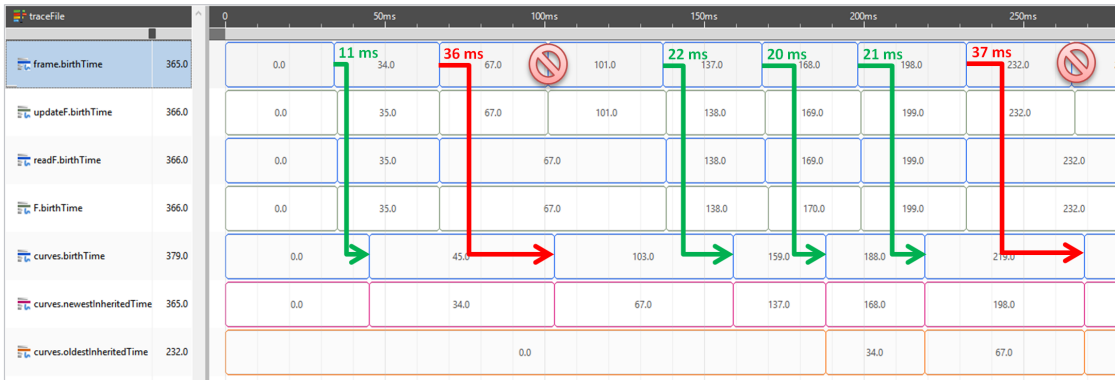


Figure 16.13.: Visualizing Timed Events for "3 out of 4" Contract

the further decomposed system fails (compare with Figure 16.14). The added delay of the Converter Channel is too much to guarantee the "3 out of 4" reaction constraint.

```
Error: RUNTIME_INFO:
VIOLATION of:
    Reaction(frame,curves) within [5 ms,28 ms] 3 out of 4 times..
    Out of 4 events 2 where late.
In file: p:\multic_-_fat_ak31_ausschreibung\work\frankp\docs\git\multic\wp2systemcdemonstrator\src\languageextensions.cpp:490
In process: adas.contractLd.G2.monitor @ 34771 ms
```

Figure 16.14.: Violation of Lane Detection's Contract and Trigger of Monitor

This issue needs to be fixed by further modifications to the decomposed system contracts or the decomposition itself, until an extensive VIT simulation run no longer fails on the proof obligation and until the confidence on fulfillment is sufficient. Also missing is the implementation of all Level B monitors. Figure 16.15 depicts the completed VIT architecture for the Functional Level B Environment Model as an example.

As stated above, automated tools are considered to be mandatory to efficiently generate monitors and simulated VIT obligation checks. Today, such tools are not available and they remain to be part of future work. For the MULTIC project, the suggested design process had to be implemented and applied manually - an effort consuming task, but a valuable assessment on the to be expected advantage of the MULTIC suggested design paradigms. The experiment of this chapter ends here, as fixing the pure artificial ADAS case study isn't very helpful to proof further benefits of the methodology.

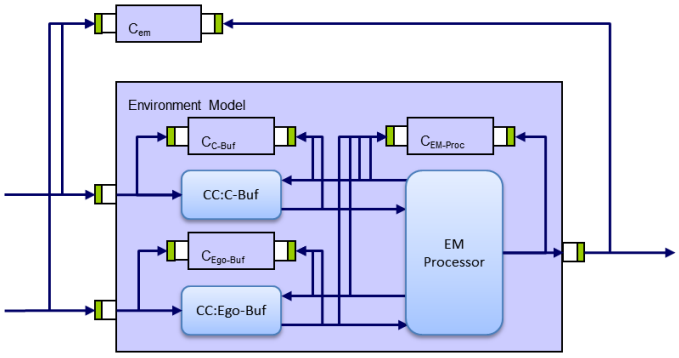


Figure 16.15.: Complete Architecture of Level B Environment Model

16.3. Level B: Timed Events with Converter Channels and Functional Models for Selected Components (Co-Simulation with Driving Simulator)

In the two experiments above, only timing specifications have be considered and checked in a VIT. Now a simplified functionality of an emergency stop assistant is added into the SystemC model from the previous section.

16.3.1. Co-Simulation Environment

The co-simulation environment is set up as depicted in Figure 16.16. The driving simulation software SILAB provides the ego-vehicle, the environment model and the sensors. The functional model of the ADAS is executed in the SystemC simulation environment. The necessary data exchange between the two simulation platforms is realized through a network socket connection.

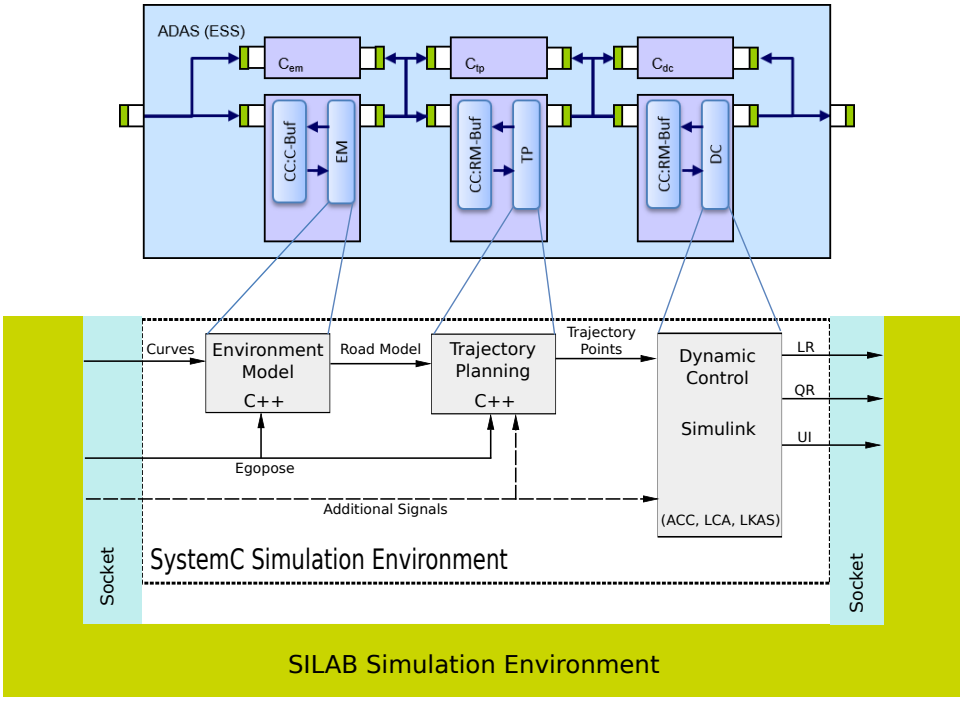
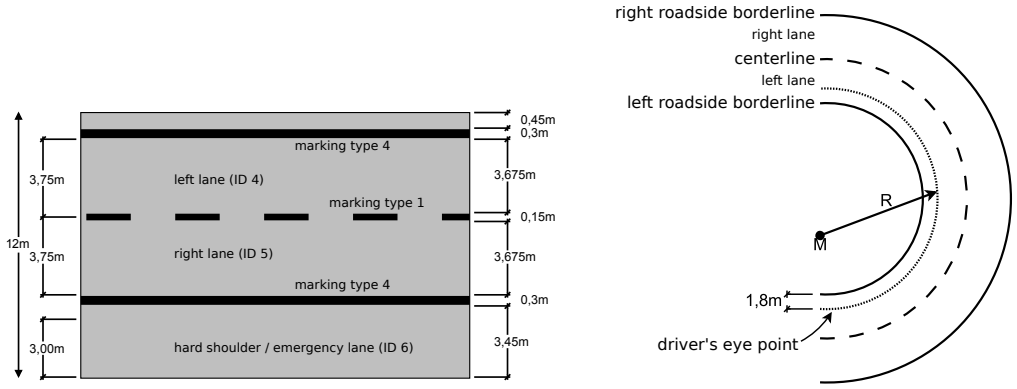


Figure 16.16.: Co-Simulation SILAB and SystemC

SILAB Simulation Environment Within the simulation environment the ego-vehicle is represented by a middle-class vehicle – a Skoda Octavia – as it provides defined dimensions and is available as a model for the simulator. The driving behavior of this vehicle has been experimentally determined by means of the driving simulator. In addition, the required sensors have been implemented within the simulator. These sensors are idealized so that no noisy sensor data exists.

The road model used is a German national four-lane motorway with hard shoulders. The dimensions are defined by the guidelines for constructing roadways [74] and are depicted in Figure 16.17a. Figure 16.17b shows the minimal curve radius which is about 900m. This minimum radius allows a driver as well as an ADAS sufficient forwards and backwards view for the early recognition of obstacles.



(a) Road dimensioning (b) Curve dimensioning

Figure 16.17.: Road model dimensioning

SystemC Environment The ADAS functional model runs in SystemC and is required to run faster than real time, so that it responds immediately to vehicle status updates and provides vehicle control inputs at the required intervals.

Socket Connection The vehicle simulator and SystemC simulation exchange status and control data via UDP datagrams over an Ethernet network connection. It is required that the complete network connection (including both network stacks and hardware) has no packet loss and delay and jitter of less than one millisecond; in practice a conventional desktop environment meets this requirement. UDP was chosen both because the data is inherently in discrete packets and time-critical (repeating a missed update is unhelpful).

Synchronization The vehicle simulator runs in real-time and provides the clock for the overall simulation. Both the receipt of vehicle status and the intervals for sending control inputs are used as synchronization points for the SystemC model, suspending the SystemC process until the elapsed real time matches the ADAS simulation time.

Signals Table 16.1 summarizes the different signals which are necessary for the ESS functionality and the co-simulation between SILAB and SystemC. Most of the signals, their range and unit are self explanatory or are depicted in Figure 16.17. The outputs for the longitudinal control are corresponding to the brake and throttle pedal positions of the ego vehicle. It should furthermore be noted that all signals are specific for the simulation environment used in this demonstrator.

16.3.2. Functional Models

The Emergency Stopping System (ESS) monitors the health status of the driver and takes over control of the ego vehicle in case of an emergency. After the ego vehicles driving function has been adopted

Signal	Description	Range	Unit
Curves	Lane ID	[0 .. 6]	
	Lane Width	[0 .. 3750]	cm
	Lane Marking Left/Right	[0 .. 6]	
	Curvature	[-2550 .. 2550]	$\frac{1}{r}$
	Distance to Curve	[0 .. 200]	m
Egopose	Velocity	[0 .. 160]	km/h
	xOff	[0 .. Lane Width]	cm
	Yaw	[0 .. 180]	degree
	Odometry	[0 .. ∞]	m
Road Model	Lane ID	[0 .. 6]	
	Lane Width	[0 .. 3750]	cm
	Curvature	[-2550 .. 2550]	$\frac{1}{r}$
Trajectory Points	Look Ahead	[0 .. 200]	m
	Desired Velocity	[0 .. 160]	km/h
	Desired xOff	[0 .. Lane Width]	cm
LR	Accelerator Pedal	[0.0 .. 1.0]	actuating value
	Brake Pedal	[0.0 .. 10.0]	actuating value
QR	Steeringwheel Angle	[-3.0 .. 3.0]	rad
UI	Different signals for the user interface		
Additional Signals	Signals necessary for operation but not relevant for the demonstrator		

Table 16.1.: Signal Overview

the ESS tries to reach a safe state to call the ambulance and to minimize the risk for other traffic participants. The safe state is defined as stopping the car on the road shoulder. In the following the realization of the ESS functionality will be explained.

Environment Model Due to the application of idealized sensors within the driving simulation software SILAB data processing in the Environment Model is not necessary. As a result of this the relevant sensor data is directly forwarded from the Environment Model to the Trajectory Planning.

Trajectory Planning The Trajectory Planning implements the ESS behavior. It receives the road model and the egopose, calculates trajectory points and sends them to the Dynamic Control. A trajectory point is defined as follows: **[Look Ahead, Desired Velocity, Desired xOff]** (see Table 16.1) and has the following meaning. Starting at the current position **curPos**, the ego vehicle should reach the the desired velocity and the desired xOff not later than the ego vehicle reaches the position **curPos + Look Ahead**.

The ESS behavior is depicted in Figure 16.18. After the simulation has started the trajectory planning stays initially in the **Idle** state. After activation the system switches to the **Autonomous Driving** state, where the ego vehicle keeps the lane and the current speed. This state is implemented for demonstration purposes only.

In the case of an emergency the trajectory planning switches to the **Lane Keeping** state. Now the ESS behavior is activated and the system tries to reach the safe state. Therefore the ESS first keeps the lane and then tries to do a lane change several times until the ego vehicle reaches the road shoulder. It should be noted that lane changes in curves are not permitted. This is depicted in Figure 16.18, states **Lane Keeping** and **Lane Change**. After the ego vehicle reaches the road

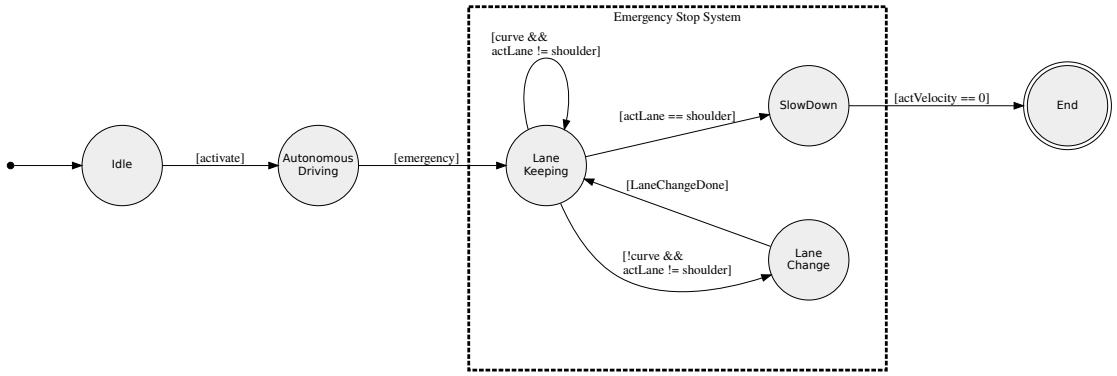


Figure 16.18.: Statemachine of the ESS behavior

shoulder the ESS starts to decrease the ego vehicles velocity until the ego vehicle stops (state **Slow Down**). Finally the ESS switches to the **End** state and the system deactivates itself. The functionality of the trajectory planning was implemented by hand in C++.

Dynamic Control In this demonstrator an existing implementation of ADAS is used to realize the Dynamic Control functionality. Supporting the driver in driving on highways, the driver assistance system ViDAs (Virtual Driver Assistance) provides functions of Adaptive Cruise Control (ACC), Lane Change Assistant (LCA), Lane Keeping Assistance (LKAS), Car2x Communication and traffic sign detection. Additionally visual feedback is given to the driver who can activate and deactivate the system by means of dedicated operation elements, and the driver is able to override the system at any time. The system's functionality is based upon state-of-the-art driver assistance systems established in the market. Technical details have been derived from the "Handbuch Fahrerassistenzsysteme" [87], the standards ISO 15622 for ACC systems [47] and ISO 11270 for LKAS systems [78]. The whole system is developed in Matlab/Simulink and Stateflow. Furthermore, design ideas of the ViDAs developers have influenced the implementation of the driver assistance system. Originally the ViDAs system was developed within a student project group as part of the computer science degree course at the Carl von Ossietzky University Oldenburg [9]. Continuous enhancements of the system are further accomplished by research projects and student theses. Figure 16.19 gives an overview of the system. The light gray subsystems are not used in this demonstrator to realize the ESS functionality.

To enable the trajectory planning to control and trigger the ViDAs System a wrapper has been implemented in Matlab/Simulink (Figure 16.19, gray box). The wrapper receives and interprets the trajectory points and forwards these to the different ViDAs subsystems to fullfill the desired longitudinal and lateral control.

16.3.3. Results

Figure 16.20a to Figure 16.20h illustrate different simulation scenarios with deactivated or activated ESS. In Figure 16.20a the ESS is deactivated and the system is in the **Idle** state. In Figure 16.20b the ESS is activated and the system is in the **Autonomous Driving** state. In Figure 16.20c an emergency has occurred and the system is now in the **Lane Keeping** state as the ego vehicle is in front of a curve. Once the ego vehicle has left the curve the ESS starts with a lane change maneuver (state **Lane Change**, Figure 16.20d). In Figure 16.20e the ego vehicle has reached the right lane and the system switches back to the **Lane Keeping** state. After that a further lane change

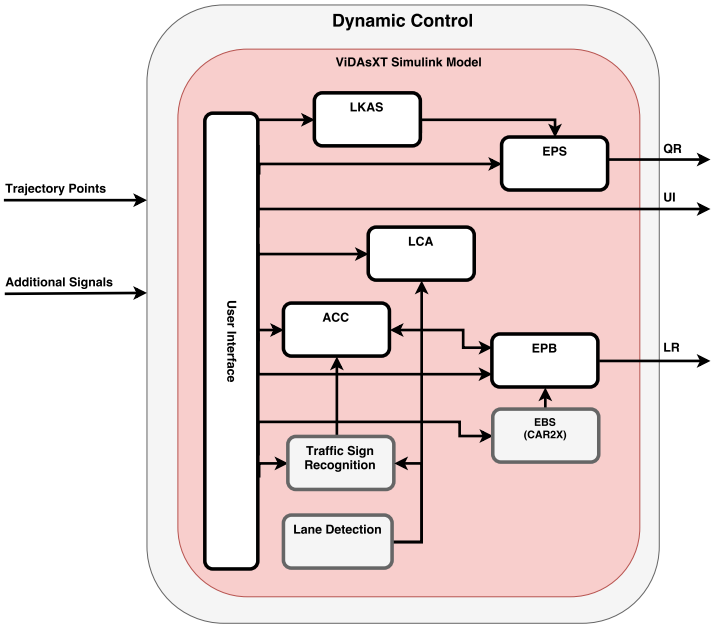


Figure 16.19.: Dynamic Control wrapper with ViDAs ADAS

is executed (see Figure 16.20f). After reaching the road shoulder the ESS changes again the state (**Slow Down**) and starts decreasing the ego vehicles velocity (Figure 16.20g). Finally the system terminates (Figure 16.20h, state **End**).

(a) Idle

(b) Autonomous Driving

(c) Lane Keeping Curve

(d) Lane Change Left Lane

(e) Lane Keeping Right Lane

(f) Lane Change Right Lane

(g) Slow Down

(h) Finished

Figure 16.20.: Different driving situations during simulation

17. Summary

The third part of this report presented the conducted experiments and demonstrations. It covers all design paradigms presented in Part I, and picks up the design approach and demonstrates its applicability on the ADAS case-study introduced in Part II. To this end, the case-study has been functionally extended to a simplified emergency stop assistant.

All demonstrations focus on the coverage of the main design activities w.r.t. timing specification and validation on Functional Level A and B. The technical levels that have been introduced in Part II have not been covered.

To support the design entry, the first part of the demonstration showed SysML model at Functional Level A & B with contracts included as SysML requirements. These requirements were textually expressed in the proposed timing specification language from Part II. For further automatic processing of such timing contracts a parser for the proposed timing specification language is required. Such parser would translate the textual timing specification into an internal representation that can be used for static contract compatibility & consistency checking and monitor/observer generation (e.g. to be used in a C++/SystemC model).

The second part of the demonstration picked up the SystemC model at Functional Level A & B, as introduced in Part II. Based on this executable model, the following three experiments have been performed:

Experiment 1 describes a simulation based virtual integration testing of the timing contracts at functional level A. The used SystemC simulation model was manually transformed from the SysML model at the same functional level. In contains (as the SysML model) no functionality or behavior but only timed events that have been derived from the timing contracts.

The experiment shows that a functional model can support and help designers in understanding the timing contracts. In a case of timing violation, the simulation provides useful information for repairing inconsistent or incompatible contracts in the SysML model.

Experiment 2 is similar to Experiment 1 but deals with functional level B. This level includes additional refinement of the top-level components and timing contracts. Furthermore, Converter Channel with channel execution semantics have been integrated and functionally tested.

Similar to Experiment 1, this experiment showed the successful application of a simulation model to check the refined timing contracts for consistency and compatibility.

Experiment 3 is an extension Experiment 2 with functional models in C++ Matlab/Simulink for selected components. Other components have been replaced by a components of a driving simulator. This experiment demonstrated the feasibility to integrate functional behavior into our timed simulation model. Furthermore, the feasibility to use the model in a complex co-simulation environment to conduct real driving experiments was shown.

For reproducibility of the experiments and results, the source code of Experiments 1 and 2 has been provided. For the complex functional simulation (which is an extension of Experiment 2), two demo videos have been produced. The first video shows the system under normal operation and the second video shows the injection of a timing violation that was detected by the timing contracts.

Conclusion

The overarching goal of the MULTIC project was to investigate on design paradigms helping engineers to develop advanced driver assistant systems and automated driving functions. In particular the focus was on enabling a coherent consideration of time and timing effects – along all design phases from the development of early functional models down to the technical realization including the hardware architecture.

We have identified four of those paradigms. A compositional semantic framework – based on a notion of components, their interfaces and their interaction – provides the common ground. Equipped with well-defined semantics allowing to express specifications in terms of contracts, and together with also well-defined operations (such as for decomposition, refinement and realization), the framework gives means to all typically design steps in the considered application domain. In fact, the framework can be instantiated and tailored for almost any design process in practice.

The second paradigm is a particular instance of the contract based design as part of the common framework. It consists of a carefully selected set of specification patterns enabling covering a multitude of time phenomena relevant in the considered application domain. The identified patterns are based on well-established specification languages while also providing substantial non-trivial extensions.

The third paradigm concerns the implementation phase of the design, where engineers use different programming and modeling languages in order to breath life into the system design. We argue that the problems induced by mixing the various languages are rooted in the underlying models of computation (MoC) – or more precisely the different models of time. Hence, we have identified most of the known such MoCs and showed how they can be brought together into a common semantic domain.

The fourth design paradigm provides for integrating different MoCs by the definition of converter channels – a particular kind of components. It also enables engineers to manage complexity when integrating different subsystems. Converter channels are particularly tailored *interaction components*, which are suggested to be available as library 'functions' in the respective design models.

All those paradigms are well-known either in academia, industrial practice or both. Although we have extended them where needed in order to fit the particular needs of ADAS/AD design (with a special focus on timing specifications and component interaction), it is foremost their interplay which makes them a sharp sword. This is exemplified by a case study, where most of the proposed building bricks are applied to an abstract automated driving function. This part also discusses some useful analysis methods helping engineers to recognize the validity of the particular design steps with respect to specified timing properties.

The third and final contribution of the report is a discussion of a practical hands-on demonstration of the case study. Here, the driving function has been instantiated by an emergency stopping system which brings the car into a save state on the hard shoulder in case of a medical emergency of the human driver. The demonstrator not only implements the function but also applies established modeling and analysis languages and tools in order to show the applicability of the approach.

The report indeed leaves open questions, which is no surprise considering the complexity of the matter. The most severe one might be to judge about its practical use, as the paradigms are introduced on a conceptual level and are only partially instantiated. On the other hand the report references a vast amount of publications, projects, standards and tools where these paradigms have been applied, exploited and tailored for various application domains. As such we consider the findings of the report a success in terms of supporting time coherency in ADAS and automated driving systems engineering. We have identified suitable design paradigms for the handling of time and timing effects and demonstrated how that can be embedded in an integrated design framework. Hence, we have shown that coherent treatment of time and timing effects in ADAS/AD design is indeed possible and can be integrated in typical industrial development processes.

Bibliography

[1] *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Version 1.0. Object Management Group. 2009.

[2] Tobias Amnell et al. "TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems". In: *Proceedings of the 1st International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'03)*. Lecture Notes in Computer Science 2791. Springer-Verlag, 2003, pp. 60–72.

[3] AUTOSAR GbR. *Software Component Template*. Version 4.2.2. 2015.

[4] AUTOSAR GbR. *Specification of Operating System*. Version 4.2.2. 2015.

[5] AUTOSAR GbR. *Specification of RTE*. Version 4.2.2. 2015.

[6] AUTOSAR GbR. *Specification of Timing Extensions*. Version 4.2.2. 2015.

[7] AUTOSAR GbR. *Virtual Functional Bus*. Version 4.2.2. 2015.

[8] Alessandra Bagnato et al. *Handbook of Research on Embedded Systems Design*. Hershey, PA, USA, 2014. DOI: 10.4018/978-1-4666-6194-3. URL: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-4666-6194-3.

[9] Ianyu Bao et al. *ViDAs - Projektgruppenendbericht*. Webseite. Online erhältlich unter http://vidas.informatik.uni-oldenburg.de/autobuild/doc/Endbericht.pdf; abgerufen am 14. Juni 2017. 2010.

[10] Michael von der Beeck. "A Comparison of StateCharts Variants". In: *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*. Berlin: Springer-Verlag, 1994, pp. 128–148.

[11] Albert Benveniste and Gerard Berry. "The Synchronous approach to Reactive and Real-Time Systems". In: *Proceedings of the IEEE*. 1991, pp. 1270–1282.

[12] Albert Benveniste et al. *Contracts for Systems Design: Methodology and Application Cases*. Research Report RR-8760. Inria Rennes Bretagne Atlantique ; INRIA, 2015, pp. 1–63. URL: https://hal.inria.fr/hal-01178469.

[13] Albert Benveniste et al. *Contracts for Systems Design: Theory*. Research Report RR-8759. Inria Rennes Bretagne Atlantique; INRIA, 2015, pp. 1–86. URL: https://hal.inria.fr/hal-01178467.

[14] Albert Benveniste et al. "Multiple Viewpoint Contract-Based Specification and Design". In: *Proceedings of the 6th International Symposium on Formal Methods for Components and Objects (FMCO'07)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 200–225. ISBN: 978-3-540-92188-2. DOI: 10.1007/978-3-540-92188-2_9. URL: http://dx.doi.org/10.1007/978-3-540-92188-2_9.

[15] Guillem Bernat, Alan Burns, and Albert Llamosi. "Weakly Hard Real-Time Systems". In: *IEEE Transactions on Computers* 50.4 (2001), pp. 308–321. ISSN: 0018-9340. DOI: 10.1109/12.919277. URL: http://dx.doi.org/10.1109/12.919277.

[16] Christian Bertsch, Elmar Ahle, and Ulrich Schulmeister. "The Functional Mockup Interface - seen from an Industrial Perspective". In: *Proceedings of the 10th International Modelica Conference*. Linköping University Electronic Press; Linköpings universitet, 2014, pp. 27–33.

[17] David Broman et al. "Requirements for Hybrid Cosimulation Standards". In: *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. HSCC '15. Seattle, Washington: ACM, 2015, pp. 179–188. ISBN: 978-1-4503-3433-4. DOI: 10.1145/2728606.2728629. URL: http://doi.acm.org/10.1145/2728606.2728629.

[18] Doron Bustan et al. "SystemVerilog Assertions: Past, Present, and Future SVA Standardization Experience". In: *IEEE Design & Test of Computers* 29.2 (2012), pp. 23–31. ISSN: 0740-7475. DOI: 10.1109/MDT.2012.2183336.

[19] Paul Caspi et al. "Semantics-Preserving Multitask Implementation of Synchronous Programs". In: *ACM Transactions on Embedded Computing Systems* 7.2 (2008), 15:1–15:40. ISSN: 1539-9087.

[20] Samarjit Chakraborty and Lothar Thiele. "A New Task Model for Streaming Applications and its Schedulability Analysis". In: *Design, Automation and Test (DATE'05)*. 2005.

[21] Alessandro Cimatti and Stefano Tonetta. "Contracts-Refinement Proof System for Component-Based Embedded Systems". In: *Science of Computer Programming* 97, Part 3 (2015). Object-Oriented Programming and Systems (OOPS 2010) Modeling and Analysis of Compositional Software (papers from EUROMICRO SEAA 12), pp. 333–348. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2014.06.011. URL: http://www.sciencedirect.com/science/ARTICLE/pii/S0167642314002901.

[22] Abhijit Davare et al. "A Next-Generation Design Framework for Platform-based Design". In: *Proceedings of the Design and Verification Conference (DVCon'07)*. 2007.

[23] Marco Di Natale et al. "Synthesis of Multitask Implementations of Simulink Models with Minimum Delays". In: *IEEE Transactions on Industrial Informatics* (2010). ISSN: 1551-3203. DOI: 10.1109/TII.2010.2072511.

[24] *EAST ADL 2.0 Specification*. The ATESST Consortium. 2008.

[25] Johan Eker et al. "Taming Heterogeneity - the Ptolemy Approach". In: *Proceedings of the IEEE* 91.1 (2003), pp. 127–144. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829.

[26] Nico Feiertag et al. "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics". In: *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*. IEEE Computer Society, 2008.

[27] Yishai A. Feldman, Lev Greenberg, and Eldad Palachi. "Simulating Rhapsody SysML Blocks in Hybrid Models with FMI". In: *Proceedings of the 10th International Modelica Conference*. Linköping University Electronic Press; Linköpings universitet, 2014, pp. 43–52.

[28] Orlando Ferrante et al. "Monitor-Based Run-Time Contract Verification of Distributed Systems". In: *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. 2014, pp. 1–4. DOI: 10.1109/SIES.2014.7087332.

[29] Daniel D. Gajski, Frank Vahid, and Sanjiv Narayan. "SpecCharts: A VHDL Front-End for Embedded Systems". In: *IEEE Transactions on Computer Aided Design* 14.6 (1995), pp. 694–706.

[30] Marc Geilen and Twan Basten. "Requirements on the Execution of Kahn Process Networks". In: *Proceedings of the 12th European Conference on Programming (ESOP'03)*. Warsaw, Poland: Springer-Verlag, 2003, pp. 319–334. ISBN: 3-540-00886-1. URL: http://dl.acm.org/citation.cfm?id=1765712.1765736.

[31] Tayfun Gezgin, Raphael Weber, and Markus Oertel. "Multi-aspect Virtual Integration approach for Real-Time and Safety Properties". In: *International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS'14)*. Lausanne: IEEE Computer Society, 2014.

[32] Amir H. Ghamarian et al. "Liveness and Boundedness of Synchronous Data Flow Graphs". In: *Proceedings of the 6th International Conference on Formal Methods in Computer Aided Design (FMCAD'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 68–75. ISBN: 0-769-52707-8. DOI: http://dx.doi.org/10.1109/FMCAD.2006.20.

[33] Alain Girault, Bilung Lee, and Edward A. Lee. *Hierarchical Finite State Machines with Multiple Concurrency Models*. Tech. rep. UCB/ERL M97/57. Berkeley, CA 94720: Electronics Research Laboratory, 1998.

[34] Nicholas Halbwachs et al. "The Synchronous Dataflow Programming Language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[35] Arne Hamann et al. "A Framework for Modular Analysis and Exploration of Heterogeneous Embedded Systems". In: *Real-Time Systems* 33.1–3 (2006), pp. 101–137.

[36] Arne Hamann et al. "Demonstration of the FMTV 2016 Timing Verification Challenge". In: *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'16) – Work-in-Progress and Demo Proceedings*. 2016, pp. 61–62.

[37] Michael G. Harbour et al. "MAST: Modeling and Analysis Suite for Real Time Applications". In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*. 2001, pp. 125–134.

[38] David Harel. "StateCharts: A Visual Formalism for Complex Systems". In: *Science of Programming* 8.3 (1987).

[39] Martijn Hendriks and Marcel Verhoef. "Timed Automata based Analysis of Embedded System Architectures". In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing Symposium (IPDPS'06)*. 2006. DOI: 10.1109/IPDPS.2006.1639422.

[40] Rafik Henia et al. "System Level Performance Analysis - the SymTA/S Approach". In: *IEE Proceedings – Computers and Digital Techniques* 152.2 (2005), pp. 148–166. ISSN: 1350-2387. DOI: 10.1049/ip-cdt:20045088.

[41] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. "Giotto: A Time-Triggered Language for Embedded Programming". In: *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT'01)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 166–184. ISBN: 978-3-540-45449-6. DOI: 10.1007/3-540-45449-7_12. URL: http://dx.doi.org/10.1007/3-540-45449-7_12.

[42] *Homepage of the Accellera Systems Initiative*. http://www.accellera.org. URL: http://www.accellera.org.

[43] *Homepage of the Accellera Systems Initiative SystemC Analog/Mixed-Signal (AMS) Working Group*. http://accellera.org/activities/working-groups/systemc-ams. URL: http://accellera.org/activities/working-groups/systemc-ams.

[44] *Homepage of the Modelica and the Modelica Association*. https://www.modelica.org/. URL: https://www.modelica.org/.

[45] Accellera Systems Initiative. "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[46] Intecs. *CHESS Modelling Language UML/MARTE/SysML profile*. Technical Report. PolarSys, 2012.

[47] *ISO 15622 Transport information and control systems - Adaptive Cruise Control Systems - Performance requirements and test procedures*. ANSI American National Standards Institute. 2002.

[48] ISO/IEC/(IEEE). *ISO/IEC 42010 (IEEE Std) 1471-2000 : Systems and Software engineering - Recomended practice for architectural description of software-intensive systems*. 2007.

[49] Axel Jantsch. *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann series in Systems on Silicon. Morgan Kaufmann, 2004. ISBN: 978-1-55860-925-9. URL: http://books.google.de/books?id=hgYhEhzI72IC.

[50] Axel Jantsch. "Models of Embedded Computation". In: *Embedded Systems Handbook*. CRC Press, 2005.

[51] Omar Kacimi et al. "Creating a Reference Technology Platform - Performing Model-based Safety Analysis in a Heterogeneous Development Environment". In: *Modelsward*. SCITEPRESS – Science, and Technology Publications, 2014, pp. 645–652.

[52] Gilles Kahn. "The Semantics of Simple Language for Parallel Programming". In: *IFIP Congress*. 1974, pp. 471–475. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip74.html#Kahn74.

[53] Nico Kämpchen et al. "Umfelderfassung für den Nothalteassistenten – ein System zum automatischen Anhalten bei plötzlich reduzierter Fahrfähigkeit des Fahrers". In: *Braunschweiger Symposium Automatisierungs-, Assistenzsysteme und eingebettete Systeme für Transportmittel (AAET)*. 2010.

[54] Paul Le Guernic et al. "Programming Real-Time Applications with Signal". In: *Proceedings of the IEEE*. Another look at Real-time programming 79.9 (1991), pp. 1321–1336. DOI: 10.1109/5.97301.

[55] Edward A. Lee and David G. Messerschmitt. "Synchronous Data Flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[56] Edward A. Lee and Alberto Sangiovanni-Vincentelli. "A Framework for Comparing Models of Computation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17.12 (1998). ISSN: 0278-0070. DOI: 10.1109/43.736561.

[57] Alex Lotz et al. "Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis". In: *Proceedings of the IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR'16)*. 2016, pp. 170–176. DOI: 10.1109/SIMPAR.2016.7862392.

[58] Alex Lotz et al. "Modeling Non-Functional Application Domain Constraints for Component-Based Robotics Software Systems". In: *CoRR* abs/1601.02379 (2016). URL: http://arxiv.org/abs/1601.02379.

[59] *MathWorks Hompage*. http://www.mathworks.de/. URL: http://www.mathworks.de/.

[60] Zaur Molotnikov et al. *Future Programming Paradigms in the Automotive Industry*. Tech. rep. 287. VDA - FAT Schriftenreihe, 2016.

[61] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. "Implementation of End-to-End Latency Analysis for Component-Based Multi-Rate Real-Time Systems in Rubus-ICE". In: *Proceedings of the 9th IEEE International Workshop on Factory Communication Systems (WFCS'12)*. 2012, pp. 165–168. DOI: 10.1109/WFCS.2012.6242562.

[62] Moritz Neukirchner et al. "Monitoring of Workload Arrival Functions for Mixed-Criticality Systems". In: *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS'13)*. 2013, pp. 88–96. DOI: 10.1109/RTSS.2013.17.

[63] *OMG Systems Modeling Language™(OMG SysML)*. Version 1.4. Object Management Group. 2015.

[64] *OMG Unified Modeling Language™(OMG UML), Superstructure*. Version 2.3. Object Management Group. 2010.

[65] Thomas M. Parks. "Bounded Scheduling of Process Networks". PhD thesis. University of California at Berkeley, 1995.

[66] TIMMO Partners. *TADL: Timing Augmented Description Language Version 2*. TIMMO Deliverable D6. TIMMO Project, 2009.

[67] TIMMO-2-USE Partners. *Language Syntax, Semantics, Metamodel V2*. TIMMO-2-USE Deliverable D11. TIMMO-2-USE Project, 2012.

[68] JoAnn M. Paul and Donald E. Thomas. "Models of Computation for Systems-on-Chips". In: *Multiprocessor Systems-on-Chip*. Ed. by Ahmed A. Jerraya and Wayne Wolf. Morgan Kaufman Publishers, 2004. Chap. 15.

[69] Simon Perathoner et al. "Influence of Different System Abstractions on the Performance Analysis of Distributed Real-Time Systems". In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT'07)*. 2007. DOI: 10.1145/1289927.1289959.

[70] Jose L. Pino and Khalil Kalbasi. "Cosimulating Synchronous DSP Applications with Analog RF Circuits". In: *Proceedings of the 32nd Annual Asilomar Conference on Signals, Systems, and Computers (ACSSC'98)*. 1998.

[71] Dumitru Potop-Butucaru, Robert de Simone, and Jean-Pierre Talpin. "The Synchronous Hypothesis and Synchronous Languages". In: *Embedded Systems Handbook*. 2004.

[72] Philipp Reinkemeier et al. "A Pattern-based Requirement Specification Language: Mapping Automotive Specific Timing Requirements". In: *Software Engineering*. Lecture Notes in Informatics. Gesellschaft für Informatik, 2011.

[73] Philipp Reinkemeier et al. "Contracts for Schedulability Analysis". In: *Proceedings of the 13th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'15)*. 2015.

[74] *Richtlinien für die Anlage von Autobahnen*. Forschungsgesellschaft für Straßen- und Verkehrswesen, Arbeitsgruppe Straßenentwurf. 2008.

[75] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison Wesley, 1998. ISBN: 978-0-20189-539-1. URL: http://books.google.de/books?id=j2kZAQAAIAAJ.

[76] Lui Sha et al. "Real Time Scheduling Theory: A Historical Perspective". In: *Real-Time Systems* 28.2 (2004), pp. 101–155. ISSN: 1573-1383.

[77] Insik Shin and Insup Lee. "Periodic Resource Model for Compositional Real-Time Guarantees". In: *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS'03)*. IEEE Computer Society, 2003, pp. 2–13.

[78] International Organization for Standardization. *ISO 11270 - Intelligent transport systems - Lane keeping assistance systems (LKAS) - Performance requirements and test procedures*. Tech. rep. International Organization for Standardization, 2014.

[79] Jan Staschulat et al. "Context Sensitive Performance Analysis of Automotive Applications". In: *Design, Automation and Test in Europe (DATE'05)*. 2005.

[80] Ingo Stierand, Philipp Reinkemeier, and Purandar Bhaduri. "Virtual Integration of Real-Time Systems based on Resource Segregation Abstraction". In: *Proceedings of the 12th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'14)*. Lectures Notes in Computer Science. 2014, pp. 1–15.

[81] Ingo Stierand et al. "From Specification Models to Distributed Embedded Applications: A Holistic User-Guided Approach". In: *SAE International Journal of Passenger Cars – Electronic and Electrical Systems* 6 (2013), pp. 194–212. DOI: 10.4271/2013-01-0432. URL: http://dx.doi.org/10.4271/2013-01-0432.

[82] Ingo Stierand et al. "Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems". In: *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES'13)*. 2013.

[83] Ralph G. Taylor. *Models of Computation and Formal Languages*. Oxford University Press, New York, 1998.

[84] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. "Real-time Calculus for Scheduling Hard Real-Time Systems". In: *IEEE International Symposium on Circuits and Systems (ISCAS'00)*. Vol. 4. 2000, pp. 101–104. DOI: 10.1109/ISCAS.2000.858698.

[85] *Website of the IEEE P1076 Study Group - VHDL Analysis and Standardization Group (VASG)*. http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome. URL: http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome.

[86] *Website of the IEEE P1364-2005 Group - Verilog Hardware Description Language*. http://www.verilog.com/IEEEVerilog.html. URL: http://www.verilog.com/IEEEVerilog.html.

[87] Hermann Winner et al. *Handbuch Fahrerassistenzsysteme*. 3rd ed. ATZ/MTZ-Fachbuch. Wiesbaden: Springer Vieweg, 2015. ISBN: 978-3-658-05733-6. DOI: 10.1007/978-3-658-05734-3.

Appendix

A. Modeling of Time

This section presents the tagged signal, and time models as defined in *"A framework for comparing Models of Computation"* [56]. It defines a denotational framework (also called "meta model") for comparing certain properties of Models of Computation. Concurrent processes in general terms are described as sets of possible behaviors. The interaction between processes is through signals, which are collections of events. Each event is a value-tag pair, where the tags can come from a partially ordered or totally ordered set. In timed models the set of tags is totally ordered. Synchronous events share the same tag, and synchronous signals contain events with the same set of tags. Synchronous processes have only synchronous signals as behaviors.

A.1. Basic Definitions

Definition 1 (Event)**.** *An event is a tuple consisting of values $v \in V$ and tags $t \in T$, with event $e \in T \times V$.*

An event has a tag and a value. Tags are used to model time, precedence relationships, synchronization points, and other key properties of a model of computation. The values represent the operands and results of computation. Note that the definition differs from the use of the term "event" in the document, which refers to the value only. The terms are however consistent because we can say that an event *occurs* at some time (tag) as well as identify the occurrence and the time where it occurs as an event. Throughout the document, an event as defined above hence is also referred as *event occurrence*.

Definition 2 (Signal)**.** *A signal s is a set of events, with $s \in \wp(T \times V)$ (the set of all subsets of $T \times V$).*

A *functional signal* or *proper signal* is a (possibly partial) function from T to V. Partial function in this case means that it may be defined only for a subset of T. Function in this case means that if $e_1 = (t, v_1) \in s$ and $e_2 = (t, v_2) \in s$, then $v_1 = v_2$. $S = \wp(T \times V)$ is the set of all signals.

$\vec{s}$ is a *tuple of N signals*, where N is a natural number. The set of all such tuples is denoted S^N. Position in the tuple serves the purpose of naming. Reordering of the tuple serves the purposes of renaming.

Definition 3 (Process)**.** *A process P is a subset of S^N for some N.*

A particular $\vec{s} \in S^N$ is said be a *behavior* of the process if $\vec{s} \in P$. Thus, a process is a set of possible behaviors. For $N \geq 2$, process are a relation between the N signals in $\vec{s}$.

More information about the composition of processes and the modeling of inputs and outputs can be found in [56].

A.2. Time Models

A natural interpretation for the tags of the tagged signal model is that they mark time in a physical system. When neglecting relativistic effects, time is the same everywhere, so tagging events with the

time at which they occur puts them in a certain order (if two events are genuinely simultaneous, then they have the same tag).

For specifying systems, the global ordering of events in a timed system may be overly restrictive. A specification should not be constrained by one particular physical implementation, and therefore need not be based on the semantics of the physical world. Thus, for specification, often the tags should not mark time, but should instead reflect ordering. In a model of a physical system, by contrast, tagging the events with the time at which they occur is natural. They must occur at a particular time, and if we accept that time is uniform (i.e., again neglecting relativistic effects), then our model should reflect the ensuing ordering of events.

The central role of a tag system is to establish ordering among events. An *ordering relation* on the set T is a reflexive, transitive, antisymmetric relation on members of the set, denoted by "$\leq$". The properties of the relation $\leq$ are $\forall\, t, t', t'' \in T$:

$$\begin{aligned} t \leq t & \quad \text{(reflexive)} \\ t \leq t' \wedge t' \leq t'' \Rightarrow t \leq t'' & \quad \text{(transitive)} \\ t \leq t' \wedge t' \leq t \Rightarrow t = t' & \quad \text{(antisymmetric)} \end{aligned}$$

The related irreflexive relation "$<$" is $t < t'$ if $t \leq t'$ and $t \neq t'$.

Ordering of the tags induces an ordering of events. Given two events $e = (t, v)$ and $e' = (t', v')$, $e < e'$ if and only if $t < t'$. A set T with an ordering relationship is called *ordered set*. If the ordering relationship is partial (there exist $t, t' \in T$ such that neither $t < t'$ nor $t' < t$), then T is called a *partially ordered set* or *poset*.

Timed Model of Computation A timed model of computation has a tag system where T is a *totally ordered set*. That is, $\forall t, t' \in T$, either $t < t'$, $t' < t$ or $t' = t$. In timed systems, a tag is also called *time stamp*. In this report we distinguish between the following flavors of timed models:

Metric Time In a metric time system the tag system T has a *metric* which is a function $d\colon T \times T \to \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers, that satisfy the following conditions $\forall t, t', t'' \in T$:

$$\begin{aligned} d(t, t') &= d(t', t) \\ d(t, t') &= 0 \Leftrightarrow t = t' \\ d(t, t') &\geq 0 \\ d(t, t') + d(t', t'') &\geq d(t, t'') \end{aligned}$$

This is given if T is an *Abelian group* in addition to being totally ordered. This means that there is an operation $+\colon T \times T \to T$, called addition, under which T is closed. Moreover, there is an element, called *zero* and denoted "0", such that $\forall t \in T\colon t + 0 = t$. Finally, for every element $t \in T$, there is another element $-t \in T$ such that $t + (-t) = 0$. A consequence is that $t_2 - t_1$ is itself a tag for any $t_1, t_2 \in T$. A sufficient metric is given by $d(t, t') = |t - t'|$.

Continuous Time Let $T(s) \subseteq T$ denote the set of tags in a signal s. A *continuous-time system* is a metric timed system where T is a connected set, and $T(s) = T$ for each signal s in any tuple $\vec{s}$ that satisfies the system. T is a connected set if no nonempty disjoint open sets O_1 and O_2 exist such that $T = O_1 \cup O_2$.

Discrete Event Given a process P, and a tuple of signals $\vec{s} \in P$ that satisfies the process, let $T(\vec{s})$ denote the set of tags appearing in any signal in the tuple $\vec{s}$. $T(\vec{s}) \subseteq T$, and the ordering

relationship for members of T induces an ordering relationship for members of $T(\vec{s})$. A *discrete-event model of computation* has a timed tag system, and for all processes P and all $\vec{s} \in P, T(\vec{s})$ is order isomorphic to a subset of the integers.

A map $f\colon A \to B$ from one ordered set A to another B is *order preserving or monotonic* if $a < a' \Rightarrow f(a) < f(a')$ (where the ordering relations are the ones for the appropriate set). A map $f\colon A \to B$ is a *bijection* if $f(A) = B \wedge a \neq a' \Rightarrow f(a) \neq f(a')$. An order isomorphism is an order-preserving bijection. Two sets are order isomorphic if there exists an order isomorphism from one to the other.

Synchronous and Discrete-Time Two events are *synchronous* if they have the same tag. Two signals are synchronous if all events in one signal are synchronous with an event in the other signal and vice versa. A process is synchronous if every signal in any behavior of the process is synchronous with every other signal in the behavior. A discrete-time system is a synchronous discrete-event system. Cycle-based logic simulators are discrete-time systems.

Sequential A degenerate form of timed tag systems is a sequential system. The tagged signal model for a sequential process has a single signal s, and the tags $T(s)$ in the signal are totally ordered. For example, under the Von Neumann model of computation, the values $v \in V$ denote states of the system, and the signal denotes the sequence of states corresponding to the execution of a program. There exist several ways to construct untimed concurrent systems by composing sequential systems.

Untimed Model of Computation When tags are *partially ordered* rather than totally ordered, we say that the tag system is *untimed*.

A.3. Transformation between models

Central to the approach in [56] is the use a tag system T, which can be partially ordered or totally ordered, and captures temporal and causal properties of systems. These properties are distinct from the functional properties of a system, which relate only to values in the corresponding models. An important observation is that a model of a system can be transformed into a different model only by manipulating the tag system.

Suppose two tag systems T and T', an order-preserving mapping $f\colon T \to T'$, and a signal $s \in \wp(T \times V)$. The mapping induces a signal $s' \in \wp(T' \times V)$ by replacing each tag t in s by $f(t)$. This mapping is elevated to processes. Given a process $P \subseteq (\wp(T \times V))^N$ we can define a new process P' constructively by mapping the signals involved in P. With this mapping P and P' are closely related processes. If, for example, T is partially ordered, where the partial order represents data precedences, and T' is totally ordered, where the tags represent time, then P' describes an implementation in time of P. For example, P might represent a dataflow model of a system and P' might represent the evaluation of that dataflow model on a sequential computer.

Note that such mapping is a formally well-defined instantiation of the concept of *realization* defined in this report. Hence, the process P' can be considered as a realization of process P in this sense. With this, the tagged signal model can be used to formulate design refinement and verification.

In the context of this report, transformation into the time domain $\mathbb{T} = \mathbb{R}^{\geq 0} \times \mathbb{N}$ commonly used for specifications is of particular importance. A tag system T together with an appropriate mapping $f : T \to \mathbb{T}$ will typically result in signals s' that are only partial functions, i.e., where $s' : \mathbb{T}' \to \mathbb{V}$ such that $\mathbb{T}' \subset \mathbb{T}$. In these cases, we (implicitly) extend the mapping to a signal $s'' : \mathbb{T}' \to \mathbb{V} \cup \{\epsilon\}$ such that $s''(t) = \epsilon$ for all $t \in \mathbb{T} \setminus \mathbb{T}'$.

B. Selection of Relevant MoCs

B.1. Kahn Process Networks

Definition

In a Kahn Process Network (KPN), concurrent processes (not to be mixed up with Def. 3) communicate only through one-way FIFO channels with unbounded capacity. Each channel carries a possibly infinite *sequence* (a *stream*) denoted $X = [x_1, x_2, \dots]$, where each x_i is an atomic data object, or *token* drawn from some set. Each token is written (produced) exactly once, and read (consumed) exactly once. Writes to the channels are *non-blocking* (they always succeed immediately), but reads are *blocking*. This means that a process that attempts to read from an empty input channel stalls until the buffer has sufficient tokens to satisfy the read.

Definition 4 (KPN Graph)**.** *A KPN Graph* $KPN = (N, E, F)$ *is defined as:*

$$
\begin{aligned}
N \neq \{\} \quad &: \quad \textit{set of process nodes (actors)} \\
E \neq \{\} \wedge E \subset V \times V \quad &: \quad \textit{set of edges with FIFO semantics} \\
F \colon N \times N \to E \quad &: \quad \textit{is a function, which determines the edges as an ordered pair of nodes } (n_i, n_j),\ n_i, n_j \in N
\end{aligned}
$$

There are two special kinds of process nodes:

data source: *does not read any input or have any input channels*

data sink: *does not write any output or have any output channels*

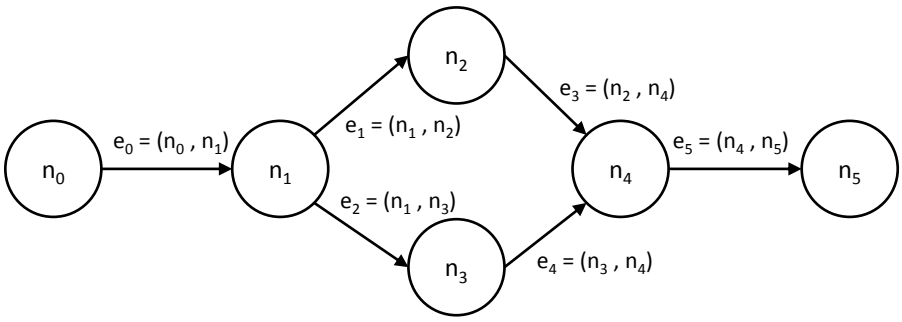


Figure B.1.: Example of a Kahn Process Network (KPN) graph. The nodes denote processes, with data source n_0 and data sink n_5. The arrows between the process notes denote FIFO channels. The direction of the arrows describe the write-read direction of data tokens.

Properties

Boundedness of channels A channel is *strictly bounded* by b if it has at most b unconsumed tokens for any possible execution. A KPN is strictly bounded by b if all channels are strictly bounded by b.

The number of unconsumed tokens depends on the execution order (scheduling) of processes. A spontaneous data source could produce arbitrarily many tokens into a channel if the scheduler would not execute processes consuming those tokens.

A real application can not have unbounded FIFOs and therefore scheduling and maximum capacity of FIFOs must be designed into a practical implementation. The maximum capacity of FIFOs can be handled in several ways:

- FIFO bounds can be mathematically derived in design to avoid FIFO overflows. This is however not possible for all KPNs. It is an undecidable problem to test whether a KPN is strictly bounded by b. Moreover, in practical situations, the bound may be data dependent.
- FIFO bounds can be grown on demand [65].
- Blocking writes can be used so that a process blocks if a FIFO is full. This approach may unfortunately lead to an artificial deadlock unless the designer properly derives safe bounds for FIFOs [65]. Local artificial deadlock detection at run-time may be necessary to guarantee the production of the correct output [30].

Determinism Processes of a KPN are *deterministic*. For the same input history they must always produce exactly the same output. Processes can be modeled as sequential programs that do reads and writes to ports in any order or quantity as long as determinism property is preserved. As a consequence, KPN model is deterministic so that following factors entirely determine outputs of the system: processes, the network, initial tokens. Hence, timing of the processes does not affect outputs of the system.

Monotonicity KPN processes are *monotonic*, which means that they only need partial information of the input stream in order to produce partial information of the output stream. Monotonicity allows parallelism. In a KPN there is a total order of events/emitted data per communication edge. However, there is no order relation between events/emitted data in different communication edges. Thus, KPNs are only partially ordered, which classifies them as untimed model.

B.2. Synchronous Data Flow

A Synchronous Data Flow model is a restriction of Kahn process networks. Nodes produce and consume a fixed number of atomic data objects (tokens) per firing.

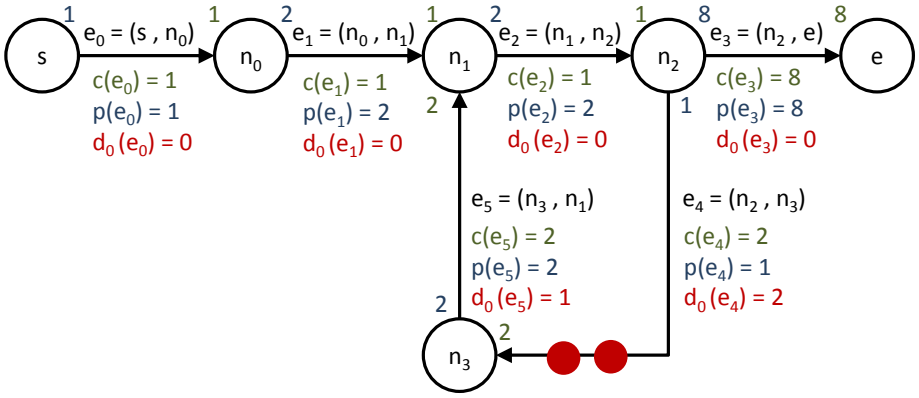


Figure B.2.: Example of a Synchronous Data Flow (SDF) graph. The nodes denote processes, with data source s and data sink e. The arrows between the process notes denote FIFO channels. The direction of the arrows describe the write-read direction of data tokens. A static schedule of this graph can be computed from the balance equation $4s = 4n_0 = 2n_1 = n_2 = e = 2n_3$ with the solution $s = 1, n_0 = 1, n_1 = 2, n_2 = 4, n_3 = 2, e = 4$ and a possible resulting schedule of $sn_0n_3n_1n_2n_2een_3n_1n_2n_2ee$.

Definition

Definition 5 (SDF Graph). *An SDF Graph $SDF = (N, E, F, c, p, d)$ is defined as:*

$$
\begin{aligned}
N \neq \{\} \quad &: \quad \textit{set of process nodes (actors)} \\
E \neq \{\} \wedge E \subset N \times N \quad &: \quad \textit{set of edges with FIFO semantics as KPN} \\
F\colon N \times N \to E \quad &: \quad \textit{is a function, which determines the edges as an ordered pair of nodes} \\
&\quad (n_i, n_j),\ n_i, n_j \in N \\
c\colon E \to \mathbb{N} \quad &: \quad \textit{number of tokens consumed (required for firing)} \\
p\colon E \to \mathbb{N} \quad &: \quad \textit{number of tokens produced after firing} \\
d\colon E \times \mathbb{N}_{\geq 0} \to \mathbb{N}_{\geq 0} \quad &: \quad \textit{time dependent number of tokens per edge (FIFO)} \\
&\quad d_0\colon E \to \mathbb{N}_{\geq 0} \textit{ initial tokens} \\
&\quad d_i \in \mathbb{N}_{\geq 0}^{|E|} \textit{ otherwise}
\end{aligned}
$$

An SDF graph with $c = p = 1$ for all edges E is a homogeneous SDF graph.

Definition 6 (Firing Rule for SDF). *A Node $u \in E$ can fire at time $i \in \mathbb{N}_{\geq 0}$, if for all incoming edges $[*, u] \in E$ at least as many tokens as required for consumption by u are available:*

$$v, u \in N \wedge [v, u] \in E \quad \forall v\colon d_i([v, u]) \geq c([u, v]) \tag{B.1}$$

Properties

Boundedness of channels The size of the FIFO channels in SDF graphs is strictly bound. The buffer size depends on the execution order (scheduling) of processes. Based on a cyclic and deadlock free schedule, the schedule with the overall minimum buffer size can be found.

Determinism As KPN also SDF graphs are *deterministic*. A static and deterministic schedule of an SDF graph can be computed. It is called *repetition vector* which can be computed by solving the so-called *balance equations*. Given a *topology matrix* $\mathbf{C}$ of the SDF graph, which contains the consumption per actor per channel (edges in rows and actors in columns), the repetition vector $\overline{r}$ can be computed by solving the linear (balance) equation system $\mathbf{C} \cdot \overline{r} = \overline{0}$ ($\overline{0}$ is the zero vector that contains only 0). Firing rates are called *consistent* if a repetition vector $\overline{r}$ exists.

Monotonicity As KPN also SDF graphs are monotonic. In addition to KPN the repeated firing of an actor can be captured as a recursive function that is *continuous*.

B.3. Finite-State Machine

A **Finite-State Machine** (FSM) is the most popular model for the description of State-based decision or open-loop controllers. An FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with these states or transitions.

Definition 7 (Finite-State Machine (FSM))**.** *A Finite-State Machine (FSM)* $FSM = [S, I, O, F, H, s_0]$ *consists of*

$$
\begin{aligned}
S = \{s_0, s_1, \ldots, s_l\} &: \textit{set of all states} \\
I = \{i_0, i_1, \ldots, i_m\} &: \textit{set of inputs} \\
O = \{o_0, o_1, \ldots, o_n\} &: \textit{set of outputs} \\
F \colon S \times I \to S &: \textit{next-state function} \\
H \colon S \to O &: \textit{output function} \\
s_0 \in S &: \textit{initial state}
\end{aligned}
$$

A moore-type FSM associates outputs with states $H \colon S \to O$*, as given above.*
A mealy-type FSM associates outputs with transitions $H \colon S \times I \to O$*.*

An extension to eliminate the problem of the state and arc explosion is the introduction of concurrency and hierarchy. This model is called **Hierarchical Concurrent Finite-State Machine** (HCFSM) [33] and is implemented in the widely used StateCharts [38, 10]. States of a HCFSM can be decomposed into an FSM. Hierarchy, also known as OR-decomposition, can be decomposed into a flat FSM with the same number of states but more transitions. Concurrent states, also known as AND-composition, can be decomposed into a sequential FSM with more states and more transitions (cross product automaton construction).

A **Dataflow Graph** (DFG) is used for describing computationally intensive systems. Terms that are used to describe computations can be easily represented by a DFG. It consists of nodes that represent operations or functions. Directed arcs between nodes define the execution order. In the context of PSMs we consider dataflow graphs which are *homogeneous SDF graphs*.

Finite-State Machines with Datapath (FSMD) combine the features of the FSM and the DFG models since most real world systems consist of both control and computation. The FSMD is well suited for modeling hardware: Each state transition appears at a single clock cycle and the operations executed in each state can be interpreted as a set of register-transfer operations.

Definition 8 (Finite-State Machine with Datapath (FSMD))**.** *A Finite-State Machine with Datapath*

(FSMD) $FSMD = [S, I, O, V, F, H, s_0]$ *consists of*

$$
\begin{aligned}
S = \{s_0, s_1, \ldots, s_l\} &: \textit{set of states} \\
I = \{i_0, i_1, \ldots, i_m\} &: \textit{set of inputs} \\
O = \{o_0, o_1, \ldots, o_n\} &: \textit{set of outputs} \\
V = \{v_0, v_1, \ldots, v_n\} &: \textit{set of variables} \\
F\colon S \times I \times V \to S &: \textit{next-state function} \\
H\colon S \to O \cup V &: \textit{action function} \\
s_0 \in S &: \textit{initial state}
\end{aligned}
$$

I, O, V may represent complex data types (i.e., integers, floating point, etc.). H is an action function, not just an output function and describes variable updates as well as outputs. Complete system state now consists of current state, s_i, and values of all variables V.

Merging the FSMD model with the concept of programming languages leads to the so-called **Superstate FSMD** (SFSMD). In this model a superstate does not represent exactly a single clock cycle as in the FSMD model, but any number of clock cycles which depend on the final implementation. Such a superstate can be specified by constructs of programming languages, as mentioned above.

Replacing the FSM model in HCFSMs by a SFSMD model leads to a HCSFSMD, or much shorter **Program-State Machine** (PSM) [29].

Hierarchical composition

A PSM consists of a hierarchy of program-states with each of them specifying a single mode of computation. At each point in time only a subset of program-states is active, and thus perform their computations. A composite program-state can be decomposed into either sequential or concurrent program-substates.

C. A Brief Summary of Contract-based Design

The following is a brief summary of contracts, corresponding definitions, relations and operations. In [14] a first version of Contract-based Design principles is provided, which has been developed in the european SPEEDS project. The more recent work [13] contains an exhaustive theoretical treatment of contracts, with [12] providing corresponding methodological discussions and application cases.

In addition to requirements, a contract expresses assumptions about the environment of a component. A simple contract theory can be constructed based on a notion of *assertions*, which is just seen as a set of traces/behaviors. Thus assertions are equipped with the algebra of sets. The behavior of an implementation of a component is identified by an assertion. We denote by M and E assertions representing the behavior of a component.

Based on this simple component model, which are just assertions, a contract is a pair of assertions as well, called *assumption* and *guarantee*:

Definition 9 (Contract)**.** *A contract is a pair $C = (A, G)$ of assertions, called assumption and guarantee.*

- *Each component $E \subseteq A$ is a legal environment of C.*
- *Each component M, for which $A \cap M \subseteq G$ holds, is an implementation of the contract C.*

Corresponding to established terminology in requirements engineering, we also say that a component M *satisfies* C, iff M is an implementation of C according to Definition 9.

Refinement is a partial order on contracts. Given two contracts C and C', C' refines contract C, written $C' \preceq C$, if and only if

$$A' \supseteq A \quad \text{and} \quad A \cap G' \subseteq G$$

Refinement thus weakens the assumption about the environment, and strengthens the guaranteed behavior under the assumed context. Consequently, an implementation of C' is also an implementation of C and is able to operate in any context defined by the assumption A of C.

In combination with a composition operator $\otimes$ on contracts, two conditions can be derived for checking whether a set of contracts, when being composed, refines another contract, i.e., $\bigotimes_{i\in\{1\ldots n\}} C_i \preceq C$. We refer to these conditions as *Virtual Integration Test* (VIT). When being satisfied, it holds that a composition of implementations of contracts $C_1 \ldots C_n$ implements C as well. This gives rise to a design paradigm supporting incremental design and independent implementability. Proper integration of components and subsystems can be checked early based on their contract specifications, while ensuring that implementations will fit together and fulfill the contract specifications of component compositions. The conditions to be checked by VIT are as follows:

$$A \bigcap_{i\in\{1\ldots n\}} (\neg A_i \cup G_i) \subseteq \bigcap_{i\in\{1\ldots n\}} A_i \tag{C.1}$$

$$A \bigcap_{i\in\{1\ldots n\}} G_i \subseteq G \tag{C.2}$$

Note that the theory of simple assume/guarantee style contracts is not based on models with an operational semantics. So methods are needed to effectively carry out the checks whether a component M implements/satisfies a contract or carrying out a VIT. For example in case of automata modeling A, G and M, this check can be carried out using an approach based on observer automata. Then the automaton for G becomes an observer of the product of A and M, transitioning into a *bad* state in case $A \times M$ produces a behavior not accepted by G. In other words, in the context defined by A, the component M does not behave as required by G, meaning M does not implement/satisfy the contract (A, G).

D. Timing Specifications

This chapter gives a brief overview of the formal semantics for timing specification patterns defined within the MULTIC project.

The definitions are based on those made in Chapter A. In the context of this report, the time domain $\mathbb{T} = \mathbb{R}^{\geq 0} \times \mathbb{N}$ is the common notion of time for timing specifications. Hence, we set $T := \mathbb{T}$ in the following. However, this "two-dimensional" time domain only serves capturing subtleties with composition operations, and we assume the second element to 0 if not stated otherwise, leaving us with the "usual" dense time domain.

Timing specifications are, as an instance of Contract-based Design, defined over component interfaces, namely the *ports* of component. Any behavior in the component model is solely observable at the component ports. Ports are typed. Hence, every behavior observable at a port is restricted to its *value domain* specified by the port type. We denote $\mathbb{V}_p$ the value domain of port p. We assume the special value ϵ to be member of every value domain, which represents the absence of a value.

Furthermore, timing specifications considered in this report focus on *discrete-event* signals, which have non-absent values only for $\tau \in D \subset \mathbb{T}$, where D is some discrete set (i.e., is order isomorphic to the natural numbers). This, and the fact that we consider functional signals allows us to represent semantics of port behavior in terms of *timed traces*, which sometimes are easier to recognize:

Definition 10 (Timed Trace)**.** *A timed trace over port p is defined as an (infinite) sequence of events $\omega_p = (t_i, v_i)_{i\in\mathbb{N}}$, where the sequence $(t_i)_{i\in\mathbb{N}}$ forms a monotonic sequence of time instances, and $v_i \in \mathbb{V}_p$. We require timed traces to be non-zeno, i.e., for each $t \in \mathbb{R}^{\geq 0}$ exists (t_i, v_i) such that $t_i = (r_i, n_i)$ and $r_i \geq t$. We denote $\Omega_p \subset \{\omega = (t_i, v_i)_{i\in\mathbb{N}}\}$ the set of timed traces observable at port p (semantics of p).*

For a set P of ports, we define timed traces $(t_i, \vec{v_i})_{i\in\mathbb{N}}$ over P, where $\vec{v_i} = (v_1, ..., v_n) \in \mathbb{V}_{p_1} \times ... \times \mathbb{V}_{p_n}$. Given a timed trace ω_P, we define its projection $\omega_P|_q$ to port $q \in P$ such that $\omega_P = (t_i, (v_1, ..., v, ..., v_n)_i)_{i\in\mathbb{N}} \Rightarrow \omega_P|_q = (t_i, v_i)_{i\in\mathbb{N}}$ We denote $\Omega_P = \{\omega_P \mid \forall q \in P : \omega_P|_q \in \Omega_q\}$ the semantics of the port set P. For a system S where P is the set of all ports in the system, we denote $\Omega_S = \Omega_P$ semantics of S.

The following sections contain material which is derived from [72]. It proposes a textual requirement specification language (RSL) based on patterns which have a formally defined semantics. The paper also presents a mapping of the Timing Augmented Description Language (TADL) to RSL patterns. Indeed, also the reverse mapping is possible in many cases, which gives rise to an automatic translation from RSL patterns to TADL constraints.

D.1. Basics

The following sections define patterns in BNF grammar. Herein, *parameters* are written in *slanted* fonts, and `keywords` are written in `typewriter` font. Sometimes, keywords are hard to recognize, in which cases they are additionally enclosed in quotation marks like in `'keyword'`. Optional parts are enclosed in brackets, followed by a question mark, like for example [optional part]?. Parts that may occur zero or more times are enclosed in brackets followed by a star, such as [repeated part]*. Grammar patterns are defined by a name (non-terminal) at the left side, followed by ::, followed by the definition. Alternatives in the definition are separated by | .

The fundamental concept for timing specifications are events. Events are as stated above, solely visible at ports, and are fixed to the corresponding value domains. Though specifications normally are attached to components, hence having a well defined context, ports are specified as follows:

Port :: *PortName* | *ComponentName* '.' *PortName*

All timing specifications refer to one or more events. The event value observed at a port may or may not be of importance. Events in the specifications comply to a common pattern:

Event :: *Port* | *Port* '.' *EventValue*

The parameter *EventValue* is deliberately left open. It may consist of labels as well as (complex) values. We introduce the following notion. Given an event (t_i, v_i), we say it satisfies the event specification *Event*, denoted $(t_i, v_i) \models$ *Event*, if either *Event* specifies a port and v_i belongs to the value domain of the port, or *Event* specifies an event value and v_i is equal to that value.

Some timing specifications refer to event sequences or sets of events:

EventExpr :: *Event* | '(' *Event* [',' *Event* ')']* | '{' *Event* [',' *Event*]* '}'

We extend the notion above to event expressions. For a sequence *Event*$_1$, ..., *Event*$_n$, we say $(t_i, v_i), ..., (t_j, v_j)$ satisfies the sequence if every (t_k, v_k),$i \leq k \leq j$, satisfies the corresponding event specification *Event*$_{1+k-i}$. For sets we say $(t_i, v_i), ..., (t_j, v_j)$ satisfies the event expression if there is a sequence of the specified events which is satisfied.

Time occurs in the specifications either as time point or as interval:

TimeExpr :: *Value Unit*
Boundary :: '[' | ']'
Interval :: *TimeExpr* | *Boundary Value* ',' *Value Boundary Unit*

Units may, as indicated in Part I, be derived from other basic units. In order to keep the definitions simple, we omit those specifications and stay with the usual time units:

Unit :: `s` | `ms` | `us` | ...

For time values, we restrict to simple numbers:

Number :: `0 .. 9` [`0 .. 9`]*
Value :: *Number* | *Number* '.' *Number*

D.2. Event Occurrence

For repetitive event occurrences on a particular port, we define a single simplified event pattern:

Repetition :: *Event* `occurs every` *Interval* [`with jitter` *TimeExpr*]? .

The parameter *Interval* defines minimal and maximal time periods between the occurrence of subsequent

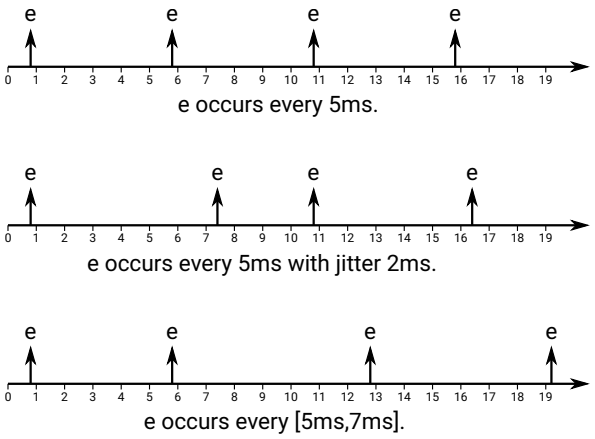


Figure D.1.: Event Occurrence Pattern Examples

events. The *jitter* defines an additional (non-deterministic) delay for the occurrence of an event.

Formally, the semantics of this pattern is defined as the set of $(t_i, v_i)_{i\in\mathbb{N}}$ such that $t_i = u_i + j_i \wedge u_0 \in [0, P^+] \wedge u_{i+1} - u_i \in I \wedge j_i \in [0, J]$ where $I = (P^-, P^+)$ $((,) \in \{[,]\})$ is the specified interval, and J is the jitter (which is 0 if omitted). This complies with the usual meaning of periodic pattern as well as patterns with minimal and maximal inter-arrival times. Figure D.1 depicts some examples.

Sometimes one wants to specify a single event occurrence. The corresponding pattern defines an interval, which is interpreted as relative to the startup of the system:

SingleEvent :: *Event* `occurs within` *Interval* .

The formal semantics of the pattern "E occurs within I." is the set of $(t_i, v_i)_{i\in\mathbb{N}}$ such that $t_0 \in I \wedge v_0 = E \wedge \forall i > 0 : v_i = \epsilon$.

D.3. Reaction Constraints

The reaction pattern provides for "classical" forward delay specifications:

Reaction :: `whenever` *EventExpr* `occurs then` *EventExpr* `occurs within` *Interval* [`once`]? .

The pattern also allows definition of reactions event sets and event sequences. For the formal definition, we use the following notion. Given a timed trace $\pi = (t_i, v_i)_{i\in\mathbb{N}}$ and an event sequence $es = e_1, ..., e_k$. Then we say the sequence $(t_i, v_i)...(t_{i+k-1}, v_{i+k-1}) \in \pi$ satisfies es, $(t_i, v_i)...(t_{i+k-1}, v_{i+k-1}) \models es$ if $v_{i+j-1} = e_j$ for all $j = 1, ..., k$. For an event set $es = \{e_1, ..., e_k\}$ we define $(t_i, v_i)...(t_{i+k-1}, v_{i+k-1}) \models es$ if $\{v_i, ..., v_{i+k-1}\} = es$.

Semantics of the pattern "whenever es_1 occurs then es_2 occurs within I", where es_1 contains k events and es_2 contains l events, respectively, is defined as the set of timed traces $(t_i, v_i)_{i\in\mathbb{N}}$ such that $\forall (t_i, v_i)...(t_{i+k-1}, v_{i+k-1}) \models es_1 : \exists j \geq i + k : (t_j, v_j)...(t_{j+l-1}, v_{j+l-1}) \models es_2 \wedge t_j - t_{i+k-1} \in I \wedge ... \wedge \wedge t_{j+l-1} - t_{i+k-1} \in I$.

The optional `once` keyword forces the pattern to fail if more than one reaction occurs within the

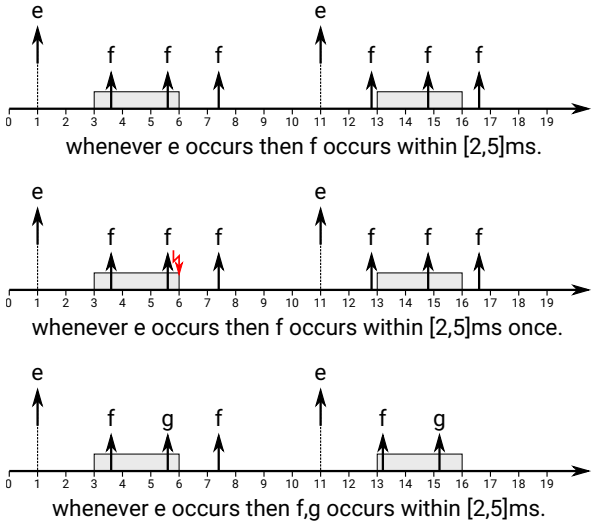


Figure D.2.: Reaction Pattern Examples

specified time window. That is, there is exactly one $j \geq i + k$ such that the corresponding sequence satisfies es_2. Figure D.2 depicts some examples.

D.4. Age Constraints

The age pattern provides for "classical" backward delay specifications:

Age :: `whenever` *EventExpr* `occurs then` *EventExpr* `has occurred within` *Interval* [`once`]? .

Formal semantics of the age pattern corresponds to the one for the reaction pattern, except that it points backward in time. The pattern "whenever es_1 occurs then es_2 has occurred within I", where es_1 contains k events and es_2 contains l events, respectively, is defined as the set of timed traces $(t_i, v_i)_{i \in \mathbb{N}}$ such that $\forall (t_i, v_i)...(t_{i+k-1}, v_{i+k-1}) \models es_1 : \exists j \geq i + k : (t_j, v_j)...(t_{j+l-1}, v_{j+l-1}) \models es_2 \wedge t_{i+k-1} - t_j \in I \wedge ... \wedge \wedge t_{i+k-1} - t_{j+l-1} \in I$.

As for reaction constraints, the optional `once` keyword forces the pattern to fail if more than one reaction occurs within the specified time window. That is, there is exactly one $j \geq i + k$ such that the corresponding sequence satisfies es_2. Figure D.3 depicts some examples.

D.5. Restricting Over- and Undersampling

Reaction and age pattern are often used in conjunction with over- or undersampling scenarios. If for example a read operation is expected to happen less often than the respective write event (undersampling), then an age pattern may be used to characterize such behavior (indeed w/o `once` extension). This kind of specification however does not allow for restricting the number of under- or oversampling situations, which can be achieved with the *restricted* flavor of the reaction and age patterns:

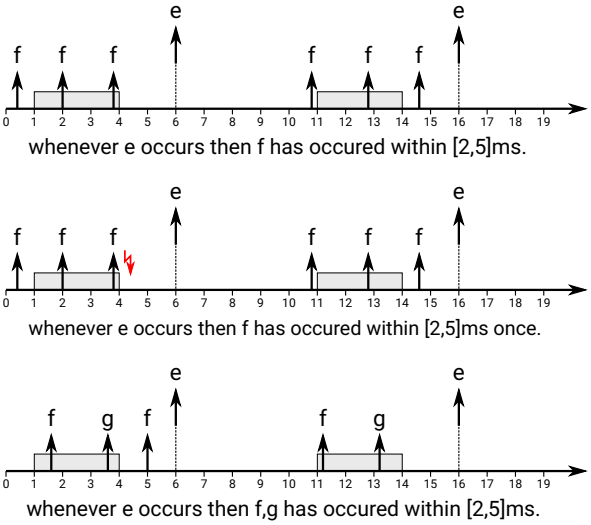


Figure D.3.: Age Pattern Examples

Reaction :: `whenever` *Event* `occurs then` *EventExpr* `occurs within` *Interval* [`once`]? [*Number* `out of` *Number* `times`]? .

Age :: `whenever` *Event* `occurs then` *EventExpr* `has occurred within` *Interval* [`once`]? [*Number* `out of` *Number* `times`]? .

The part *Number* `out of` *Number* `times` specifies that the condition defined above for the reaction (age) pattern may be violated for a fraction of occurrences. For example, the pattern "whenever e occurs then f occurs within $[10, 12]ms$ 3 out of 5 times" specifies that event f must follow e in the time window $[10, 12]ms$ at least 3 times for every 5 successive occurrences of event e. The formal definition is omitted here, as it can be easily derived from the specification of the reaction (age) pattern above. Figure D.4 depicts some examples.

D.6. Causal Event Relations

It is a well-known problem that timing specifications which are based on event observations have only limited expressiveness when it comes to specifying functional relations between events. Several approaches exists to mitigate the issue, such as [26, 61]. These approaches can be applied in rather deterministic scenarios, such as those depicted in Figure D.5. Cases with complex functional relations, over- and under-sampling situations, or where event ordering is non-deterministic, e.g., due to variable execution times in parallel execution (multi-core) platforms call for more expressive approaches. [67] proposes an approach where events can be distinguished by coloring.

The approach proposed in this report goes in the same direction. It is based on the definition of *causality function*. The advantage of this approach is that it enables the definition of (arbitrary) complex causal dependencies. This is achieved by a strictly distinguishing between the definition of the actual causal relations and their realization. On the other hand, this also means that it might be

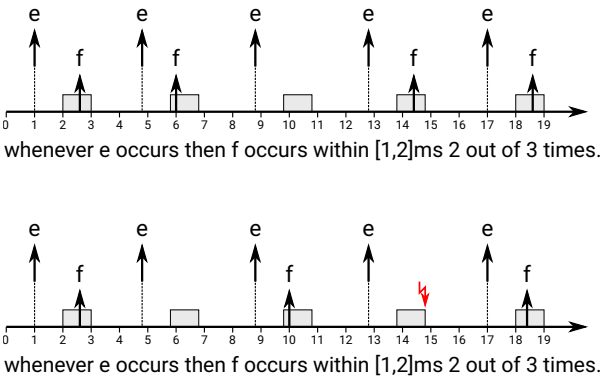


Figure D.4.: Reaction Pattern with Undersampling

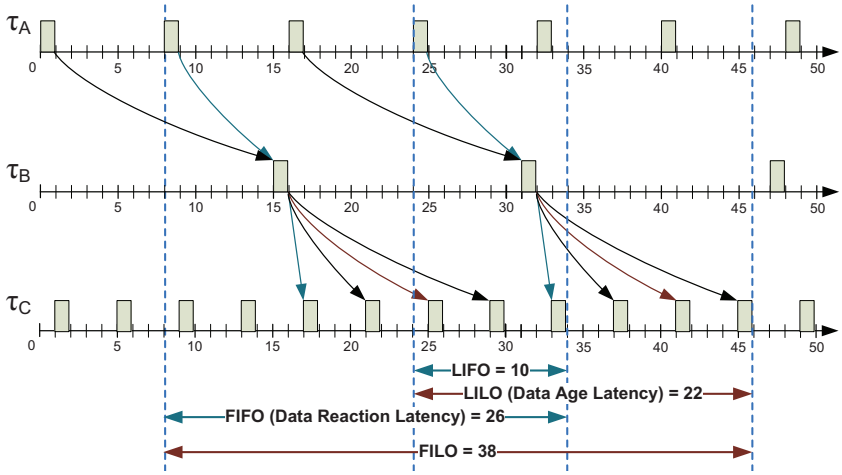


Figure D.5.: Event Causality Examples (Source: [61])

difficult to define how a given causal relation between events can be made observable.

The observability problem can be reduced to the question of how events can be uniquely identified. In the TIMMO projects, observability is established in terms of coloring of events. The approach assumes that related events can be colored in a way that enables identification of the actual relation. Another way is the introduction of time stamps. Annotating events with the time point at which they are created allows to distinguish events from each other.

Causal event relations are defined as follows:

Definition 11 (Causal Event Relation)**.** *Let p_1 and p_2 be ports, and let Ω_{p_1,p_2} be the semantics of p_1 and p_2. A* causal event relation *over p_1 and p_2 is a function*

$[> (p_1, p_2) : (\mathbb{T} \times \mathbb{V}_{p_1}) \rightarrow 2^{\mathbb{T} \times \mathbb{V}_{p_2}}$

where for all $(t_i, v_i)_{i \in \mathbb{N}} \in \Omega_{p_1,p_2}|_{p_1}$ and for all event occurrences $(t_i, v_i) \in (t_i, v_i)_{i \in \mathbb{N}}$ holds $[> (p_1, p_2)((t_i, v_i)) = \{(u_j, w_j), ..., (u_k, w_k)\} \neq \emptyset$ and $t_i \leq u_j, ..., u_k$.
We also define the backward direction of causal relations:

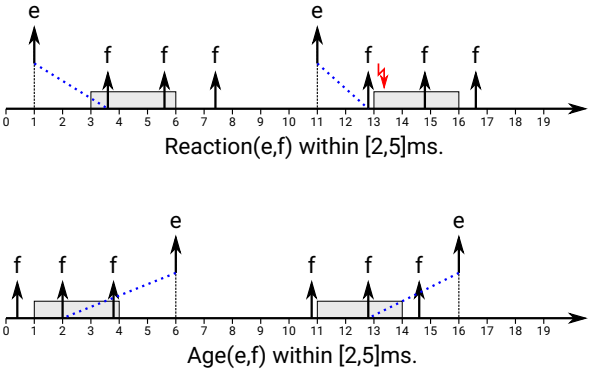


Figure D.6.: Causal Pattern Examples

$<](p_1,p_2) : (\mathbb{T} \times \mathbb{V}_{p_1}) \rightarrow 2^{\mathbb{T} \times \mathbb{V}_{p_2}}$
where for all $(t_i,v_i)_{i\in\mathbb{N}} \in \Omega_{p_1,p_2}|_{p_1}$ *and for all event occurrences* $(t_i,v_i) \in (t_i,v_i)_{i\in\mathbb{N}}$ *holds* $[> (p_1,p_2)((t_i,v_i)) = \{(u_j,w_j),...,(u_k,w_k)\} \neq \emptyset$ *and* $u_j,...,u_k \leq t_i$.

Causal event relations are transitive. Given three ports p_1, p_2, p_3, and causal event relations $[> (p_1,p_2)$ and $[> (p_1,p_2)$, then $[> (p_1,p_3)$ is defined as:

$$(u_k,w_k) \in [> (p_1,p_3)((t_i,v_i)) \Leftrightarrow \exists (t'_j,v'_j) \in [> (p_1,p_2)((t_i,v_i)) \wedge (u_k,w_k) \in [> (p_1,p_2)((t'_j,v'_j))$$

This property (though not surprisingly) gives means to the intuition of "additive" latencies, where we say that response times Xms and Yms sum up to $X+Yms$.

Event specifications are extended in order to support causal relations:

EventExpr :: `[>(`*Event* `,` *Event* `) | <](`*Event* `,` *Event* `)`

We also define a causal version of the reaction constraint:

CausalReaction :: `Reaction(`*Event* `,` *Event* `) within` *Interval* .

Semantics of the pattern "`Reaction(`e_1,e_2`) in` I." , where e_1 refers to port p_1, and e_2 refers to port p_2, respectively, is defined as follows:
For all $(t_i,v_i)_{i\in\mathbb{N}} \in \Omega_{p_1,p_2}|_{p_1}$, $(u_i,w_i)_{i\in\mathbb{N}} \in \Omega_{p_1,p_2}|_{p_2}$, and for all event occurrences $(t_i,v_i) \in (t_i,v_i)_{i\in\mathbb{N}}$, $v_i \models e_1$, holds $(u_j,w_j) \in [> (p_1,p_2)((t_i,v_i)) \wedge w_j \models e_2 \Rightarrow t_i - u_j \in I$.

Also a causal age pattern is defined:

CausalAge :: `Age(`*Event* `,` *Event* `) within` *Interval* .

Semantics of the pattern "`Age(`e_1,e_2`) in` I." is defined as follows:
For all $(t_i,v_i)_{i\in\mathbb{N}} \in \Omega_{p_1,p_2}|_{p_1}$, $(u_i,w_i)_{i\in\mathbb{N}} \in \Omega_{p_1,p_2}|_{p_2}$, and for all event occurrences $(t_i,v_i) \in (t_i,v_i)_{i\in\mathbb{N}}$ $v_i \models e_1$, holds $(u_j,w_j) \in <](p_1,p_2)((t_i,v_i)) \wedge w_j \models e_2 \Rightarrow t_i - u_j \in I$.

Figure D.6 depicts some examples.

All the above patterns require the existence of corresponding causal event relations. That is, every reference to a function $[>$ or $<]$ is assumed to be specified in the respective contracts. This calls

for a specification language for such relations. As this is however ongoing work, we specify causal relations in "classical" mathematical notations throughout the document.

D.7. BNF

TimeSpec	::	*Repetition* | *SingleEvent* | *Reaction* | *Age* | *CausalReaction* | *CausalAge*
Repetition	::	*Event* `occurs every` *Interval* [`with jitter` *TimeExpr*]? .
SingleEvent	::	*Event* `occurs within` *Interval* .
Reaction	::	`whenever` *EventExpr* `occurs then` *EventExpr* `occurs within` *Interval* [`once`]? [*Number* `out of` *Number* `times`]? .
Age	::	`whenever` *EventExpr* `occurs then` *EventExpr* `has occurred within` *Interval* [`once`]? [*Number* `out of` *Number* `times`]? .
CausalReaction	::	`Reaction(`*Event* `,` *Event* `) within` *Interval* .
CausalAge	::	`Age(`*Event* `,` *Event* `) within` *Interval* .
EventExpr	::	*Event* | `(` *Event* [`,` *Event* `)`]* | `{` *Event* [`,` *Event*]* `}` | `[>(`*Event* `,` *Event* `)` | `<](`*Event* `,` *Event* `)`
Event	::	*Port* | *Port* `.` *EventValue*
Port	::	*PortName* | *ComponentName* `.` *PortName*
Interval	::	*TimeExpr* | *Boundary Value* `,` *Value Boundary Unit*
TimeExpr	::	*Value Unit*
Boundary	::	`[` | `]`
Value	::	*Number* | *Number* `.` *Number*
Unit	::	`s` | `ms` | `us` | ...
Number	::	`0 .. 9` [`0 .. 9`]*

Bisher in der FAT-Schriftenreihe erschienen (ab 2010)

Nr.	Titel
227	Schwingfestigkeitsbewertung von Nahtenden MSG-geschweißter Dünnbleche aus Stahl, 2010
228	Systemmodellierung für Komponenten von Hybridfahrzeugen unter Berücksichtigung von Funktions- und EMV-Gesichtspunkten, 2010
229	Methodische und technische Aspekte einer Naturalistic Driving Study, 2010
230	Analyse der sekundären Gewichtseinsparung, 2010
231	Zuverlässigkeit von automotive embedded Systems, 2011
232	Erweiterung von Prozessgrenzen der Bonded Blank Technologie durch hydromechanische Umformung, 2011
233	Spezifische Anforderungen an das Heiz-Klimasystem elektromotorisch angetriebener Fahrzeuge, 2011
234	Konsistentes Materialmodell für Umwandlung und mechanische Eigenschaften beim Schweißen hochfester Mehrphasen-Stähle, 2011
235	Makrostrukturelle Änderungen des Straßenverkehrslärms, Auswirkung auf Lästigkeit und Leistung, 2011
236	Verbesserung der Crashsimulation von Kunststoffbauteilen durch Einbinden von Morphologiedaten aus der Spritzgießsimulation, 2011
237	Verbrauchsreduktion an Nutzfahrzeugkombinationen durch aerodynamische Maßnahmen, 2011
238	Wechselwirkungen zwischen Dieselmotortechnik und -emissionen mit dem Schwerpunkt auf Partikeln, 2012
239	Überlasten und ihre Auswirkungen auf die Betriebsfestigkeit widerstandspunktgeschweißter Feinblech-strukturen, 2012
240	Einsatz- und Marktpotenzial neuer verbrauchseffizienter Fahrzeugkonzepte, 2012
241	Aerodynamik von schweren Nutzfahrzeugen - Stand des Wissens, 2012
242	Nutzung des Leichtbaupotentials von höchstfesten Stahlfeinblechen durch die Berücksichtigung von Fertigungseinflüssen auf die Festigkeitseigenschaften, 2012
243	Aluminiumschaum für den Automobileinsatz, 2012
244	Beitrag zum Fortschritt im Automobilleichtbau durch belastungsgerechte Gestaltung und innovative Lösungen für lokale Verstärkungen von Fahrzeugstrukturen in Mischbauweise, 2012
245	Verkehrssicherheit von schwächeren Verkehrsteilnehmern im Zusammenhang mit dem geringen Geräuschniveau von Fahrzeugen mit alternativen Antrieben, 2012
246	Beitrag zum Fortschritt im Automobilleichtbau durch die Entwicklung von Crashabsorbern aus textil-verstärkten Kunststoffen auf Basis geflochtener Preforms und deren Abbildung in der Simulation, 2013
247	Zuverlässige Wiederverwendung und abgesicherte Integration von Softwarekomponenten im Automobil, 2013
248	Modellierung des dynamischen Verhaltens von Komponenten im Bordnetz unter Berücksichtigung des EMV-Verhaltens im Hochvoltbereich, 2013
249	Hochspannungsverkopplung in elektronischen Komponenten und Steuergeräten, 2013
250	Schwingfestigkeitsbewertung von Nahtenden MSG-geschweißter Feinbleche aus Stahl unter Schubbeanspruchung, 2013

251 Parametrischer Bauraum – synchronisierter Fahrzeugentwurf, 2013

252 Reifenentwicklung unter aerodynamischen Aspekten, 2013

253 Einsatz- und Marktpotenzial neuer verbrauchseffizienter Fahrzeugkonzepte – Phase 2, 2013

254 Qualifizierung von Aluminiumwerkstoffen für korrosiv beanspruchte Fahrwerksbauteile unter zyklischer Belastung (Salzkorrosion), 2013

255 Untersuchung des Rollwiderstands von Nutzfahrzeugreifen auf echten Fahrbahnen, 2013

256 Naturalistic Driving Data, Re-Analyse von Daten aus dem EU-Projekt euroFOT, 2013

257 Ableitung eines messbaren Klimasummenmaßes für den Vergleich des Fahrzeugklimas konventioneller und elektrischer Fahrzeuge, 2013

258 Sensitivitätsanalyse rollwiderstandsrelevanter Einflussgrößen bei Nutzfahrzeugen, Teile 1 und 2, 2013

259 Erweiterung des Kerbspannungskonzepts auf Nahtübergänge von Linienschweißnähten an dünnen Blechen, 2013

260 Numerische Untersuchungen zur Aerodynamik von Nutzfahrzeugkombinationen bei realitätsnahen Fahrbedingungen unter Seitenwindeinfluss, 2013

261 Rechnerische und probandengestützte Untersuchung des Einflusses der Kontaktwärmeübertragung in Fahrzeugsitzen auf die thermische Behaglichkeit, 2013

262 Modellierung der Auswirkungen verkehrsbedingter Partikelanzahl-Emissionen auf die Luftqualität für eine typische Hauptverkehrsstraße, 2013

263 Laserstrahlschweißen von Stahl an Aluminium mittels spektroskopischer Kontrolle der Einschweißtiefe und erhöhter Anbindungsbreite durch zweidimensional ausgeprägte Schweißnähte, 2014

264 Entwicklung von Methoden zur zuverlässigen Metamodellierung von CAE Simulations-Modellen, 2014

265 Auswirkungen alternativer Antriebskonzepte auf die Fahrdynamik von PKW, 2014

266 Entwicklung einer numerischen Methode zur Berücksichtigung stochastischer Effekte für die Crash-simulation von Punktschweißverbindungen, 2014

267 Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 1, 2014

268 Bewegungsverhalten von Fußgängern im Straßenverkehr - Teil 2, 2014

269 Schwingfestigkeitsbewertung von Schweißnahtenden MSG-geschweißter Feinblechstrukturen aus Aluminium, 2014

270 Physiologische Effekte bei PWM-gesteuerter LED-Beleuchtung im Automobil, 2015

271 Auskunft über verfügbare Parkplätze in Städten, 2015

272 Zusammenhang zwischen lokalem und globalem Behaglichkeitsempfinden: Untersuchung des Kombinationseffektes von Sitzheizung und Strahlungswärmeübertragung zur energieeffizienten Fahrzeugklimatisierung, 2015

273 UmCra - Werkstoffmodelle und Kennwertermittlung für die industrielle Anwendung der Umform- und Crash-Simulation unter Berücksichtigung der mechanischen und thermischen Vorgeschichte bei hochfesten Stählen, 2015

274 Exemplary development & validation of a practical specification language for semantic interfaces of automotive software components, 2015

275 Hochrechnung von GIDAS auf das Unfallgeschehen in Deutschland, 2015

276 Literaturanalyse und Methodenauswahl zur Gestaltung von Systemen zum hochautomatisierten Fahren, 2015

277 Modellierung der Einflüsse von Porenmorphologie auf das Versagensverhalten von Al-Druckgussteilen mit stochastischem Aspekt für durchgängige Simulation von Gießen bis Crash, 2015

278 Wahrnehmung und Bewertung von Fahrzeugaußengeräuschen durch Fußgänger in verschiedenen Verkehrssituationen und unterschiedlichen Betriebszuständen, 2015

279 Sensitivitätsanalyse rollwiderstandsrelevanter Einflussgrößen bei Nutzfahrzeugen – Teil 3, 2015

280 PCM from iGLAD database, 2015

281 Schwere Nutzfahrzeugkonfigurationen unter Einfluss realitätsnaher Anströmbedingungen, 2015

282 Studie zur Wirkung niederfrequenter magnetischer Felder in der Umwelt auf medizinische Implantate, 2015

283 Verformungs- und Versagensverhalten von Stählen für den Automobilbau unter crashartiger mehrachsiger Belastung, 2016

284 Entwicklung einer Methode zur Crashsimulation von langfaserverstärkten Thermoplast (LFT) Bauteilen auf Basis der Faserorientierung aus der Formfüllsimulation, 2016

285 Untersuchung des Rollwiderstands von Nutzfahrzeugreifen auf realer Fahrbahn, 2016

286 χMCF - A Standard for Describing Connections and Joints in the Automotive Industry, 2016

287 Future Programming Paradigms in the Automotive Industry, 2016

288 Laserstrahlschweißen von anwendungsnahen Stahl-Aluminium-Mischverbindungen für den automobilen Leichtbau, 2016

289 Untersuchung der Bewältigungsleistung des Fahrers von kurzfristig auftretenden Wiederübernahmesituationen nach teilautomatischem, freihändigem Fahren, 2016

290 Auslegung von geklebten Stahlblechstrukturen im Automobilbau für schwingende Last bei wechselnden Temperaturen unter Berücksichtigung des Versagensverhaltens, 2016

291 Analyse, Messung und Optimierung des Ventilationswiderstands von Pkw-Rädern, 2016

292 Innenhochdruckumformen laserstrahlgelöteter Tailored Hybrid Tubes aus Stahl-Aluminium-Mischverbindungen für den automobilen Leichtbau, 2017

293 Filterung an Stelle von Schirmung für Hochvolt-Komponenten in Elektrofahrzeugen, 2017

294 Schwingfestigkeitsbewertung von Nahtenden MSG-geschweißter Feinbleche aus Stahl unter kombinierter Beanspruchung, 2017

295 Wechselwirkungen zwischen zyklisch-mechanischen Beanspruchungen und Korrosion: Bewertung der Schädigungsäquivalenz von Kollektiv- und Signalformen unter mechanisch-korrosiven Beanspruchungsbedingungen, 2017

296 Auswirkungen des teil- und hochautomatisierten Fahrens auf die Kapazität der Fernstraßeninfrastruktur, 2017

297 Analyse zum Stand und Aufzeigen von Handlungsfeldern beim vernetzten und automatisierten Fahren von Nutzfahrzeugen, 2017

298 Bestimmung des Luftwiderstandsbeiwertes von realen Nutzfahrzeugen im Fahrversuch und Vergleich verschiedener Verfahren zur numerischen Simulation, 2017

299 Unfallvermeidung durch Reibwertprognosen, 2017

300 Thermisches Rollwiderstandsmodell für Nutzfahrzeugreifen zur Prognose fahrprofilspezifischer Energieverbräuche, 2017

301 The Contribution of Brake Wear Emissions to Particulate Matter in Ambient Air, 2017

302 Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC), 2017

Impressum

Herausgeber	FAT Forschungsvereinigung Automobiltechnik e.V. Behrenstraße 35 10117 Berlin Telefon +49 30 897842-0 Fax +49 30 897842-600 www.vda-fat.de
ISSN	2192-7863

VDA | Verband der Automobilindustrie

FAT | Forschungsvereinigung Automobiltechnik

Behrenstraße 35
10117 Berlin
www.vda.de
www.vda-fat.de