



Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Ensuring Real-Time Behavior of Cooperating Cyber-Physical Systems

Von der Fakultät für Informatik, Wirtschafts- und
Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg
zur Erlangung des Grades und Titels eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

angenommene Dissertation von

Herrn Björn Philipp Koopmann
geboren am 29. August 1992 in Bremerhaven

To my parents and grandparents, who enabled me to pursue my studies and made me what I am today with their upbringing

— and —

to my wonderful wife Michelle and our dearly loved children Fiona and Jano, who accompanied me through the journey of writing this thesis and fill my life with love and meaning.

Submitted: June 26, 2025

First Reviewer: *Prof. Dr. Martin Fränzle*

Second Reviewer: *Prof. Dr. Rolf Drechsler*

Defended: December 18, 2025

Abstract

The introduction of highly automated driving and intelligent transportation infrastructure in road traffic will yield new types of demand-driven mobility and value-added services. To provide these services, future transportation systems will most likely consist of cooperative ensembles of highly automated and connected systems. The dynamic nature of these systems poses new challenges for system design and safety assurance. Since safety-critical driving decisions are no longer made exclusively on the basis of the vehicles' own situational overviews, but are instead based on cross-system interaction processes, enhanced specification and verification means are required to address cooperative system behavior.

This thesis is devoted to the timing aspect and presents concepts, methods, and tools for ensuring real-time behavior of (systems of) cooperating cyber-physical systems. It proposes a correct-by-design approach for the consistent formulation and refinement of top-level timing requirements. A formalism called time-constrained activity diagrams supports modeling the behavior of cooperative systems in a scenario-based way. The resulting activity models are partitioned to assign activities and actions to individual subsystems, enabling the derivation of system-level specifications. This serves as a starting point for parallel development of the subsystems involved based on consistent top-level specifications.

The approach builds upon a well-established contract-based design method and is backed up by a rich set of timing specifications that enable specifying the timing behavior of cooperating cyber-physical systems. It provides specification means for handling operating modes, mode transitions, and mode-dependent timing behavior to reason about behavioral changes and dynamic reconfiguration. A generic communication model with example parameter sets for domain-specific communication standards allows early consideration of temporal properties of unreliable communication. Moreover, the mapping of specifications to imprecise local clocks is formally defined to support distributed run-time monitoring. The careful use of under- and overapproximations enables the construction of meaningful and traceable relations between global and local clocks.

In order to provide tool support for incremental design approaches, verification methods for single systems are adapted to support analyzing real-time requirements for cooperative systems. The extensions allow to prove the correct refinement of higher-level timing specifications into specifications on lower hierarchy levels. The concepts are implemented as part of a modular and extensible toolchain for performing real-time analyses, such as virtual integration tests, to support engineers in developing provably safe cooperative systems.

Zusammenfassung

Die Einführung des hochautomatisierten Fahrens und intelligenter Verkehrsinfrastruktur im Straßenverkehr begünstigt neue Arten der bedarfsgerechten Mobilität und eröffnet Chancen für die Entstehung neuer Geschäftsmodelle. Zukünftige Verkehrssysteme werden mit hoher Wahrscheinlichkeit aus kooperativen Verbünden hochautomatisierter und vernetzter Systeme bestehen. Die dynamische Natur dieser Systeme stellt dabei eine neue Herausforderung für den Systementwurf und die Absicherung entsprechender Systeme dar. Da sicherheitsrelevante Fahrentscheidungen in der Regel nicht mehr ausschließlich auf der Grundlage des fahrzeugeigenen Situationsbewusstseins getroffen werden, sondern auf systemübergreifenden Interaktionsprozessen beruhen, sind geeignete Spezifikations- und Verifikationsmittel zur Handhabung kooperativer Funktionen erforderlich.

Die vorliegende Dissertation widmet sich den zeitlichen Aspekten entsprechender Funktionen und präsentiert Konzepte, Methoden und Werkzeuge zur Sicherstellung des korrekten zeitlichen Verhaltens von kooperativen Systemen. Der entwickelte Correct-by-Design-Entwurfsansatz unterstützt die konsistente Formulierung und Verfeinerung von zeitlichen Anforderungen. Hierfür wird das gewünschte Systemverhalten in einem szenarienbasierten Ansatz mithilfe von zeitbehafteten Aktivitätsdiagrammen modelliert. Die so entstehenden Aktivitätsmodelle werden anschließend systematisch in zeitliche Spezifikationen auf Systemebene übersetzt, die als Ausgangspunkt für den parallelen, hersteller-spezifischen Entwurf und die Implementierung der Teilsysteme dienen.

Der gewählte Ansatz basiert auf dem bewährten Prinzip des vertragsbasierten Entwurfs und erweitert eine bestehende Spezifikationssprache für Echtzeiteigenschaften. So werden geeignete Beschreibungsmittel für die Spezifikation von Betriebsmodi, Modusübergängen und modusabhängigem zeitlichen Verhalten entwickelt, um das dynamische Verhalten sowie die Rekonfiguration kooperativer Systeme zu beschreiben. Ein generisches Kommunikationsmodell mit beispielhaften Parametersätzen für domänenspezifische Kommunikationsstandards ermöglicht die Berücksichtigung zeitlicher Eigenschaften unzuverlässiger Kommunikation. Darüber hinaus wird die Übersetzung globaler Spezifikationen in den Zeitbereich ungenauer lokalen Uhren formal definiert, um eine Laufzeitüberwachung zur Verifikation des Echtzeitverhaltens zu ermöglichen.

Zur werkzeugseitigen Unterstützung inkrementeller Entwurfsansätze werden bestehende Verifikationsmethoden angepasst, um die Analyse des zeitlichen Verhaltens kooperativer Systeme zu unterstützen. Die Erweiterungen zielen hierbei insbesondere auf den Nachweis der korrekten Verfeinerung zeitlicher Spezifikationen ab. Sie werden als Teil einer modularen und erweiterbaren Werkzeugkette für Echtzeitanalysen implementiert, die an industriell genutzte Modellierungsumgebungen angebunden ist. Die Ergebnisse unterstützen Ingenieure bei der systematischen und effizienten Entwicklung sicherer kooperativer Systeme.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Cooperative Cyber-Physical Systems	5
1.2.1	Cooperative Driving: A Vision of Cooperative Systems	6
1.2.2	Characteristics of Cooperative Cyber-Physical Systems	17
1.2.3	Challenges in Ensuring Real-Time Behavior	22
1.3	Research Questions	24
1.4	Outline	25
2	Methods and Tools for Ensuring Real-Time Behavior	27
2.1	Existing Approaches and Technologies	28
2.1.1	Frameworks for Timing Specification	29
2.1.2	Modeling and Specification Languages	37
2.1.3	Contract-Based Design	41
2.1.4	Consistency Analysis	44
2.1.5	Real-Time and Scheduling Analysis	45
2.1.6	Run-Time Monitoring	46
2.2	Comparison of Existing Frameworks	47
2.3	Multi-Layer Time Coherency Methodology	50
2.3.1	Design Paradigms	51
2.3.2	Design Approach	53
2.3.3	Tool Support	56
2.4	Contribution	58
3	Consistent Specification of Top-Level Timing Requirements	63
3.1	Modeling and Specification of Cooperative Behavior	63
3.1.1	Classical Modeling Approaches	65
3.1.2	Business Process Modeling	65
3.1.3	Temporal Constraint Networks	66
3.1.4	Modeling Using Sequence Diagrams	67
3.1.5	Activity Modeling	69
3.2	Process Overview	70
3.2.1	Pre-Competitive Design of Cooperative System Behavior	70
3.2.2	Manufacturer-Specific Design and Implementation	72
3.3	Scenario-Based Activity Modeling	73
3.3.1	Time-Constrained Activity Diagrams	74
3.3.2	Application Example	85

3.4	Derivation of System-Level Timing Specifications	92
3.4.1	Partitioning of Time-Constrained Activity Diagrams	93
3.4.2	Mapping of Events and Signals to Ports	101
3.4.3	Formulation of System-Level Specifications	106
3.5	Development of Constituent Systems	120
4	Handling of Dynamic Architectures	125
4.1	Modeling and Specification of Dynamic Architectures	126
4.1.1	System of Systems Engineering	126
4.1.2	Formal Modeling of Dynamic Systems	127
4.1.3	Modeling of Dynamic Architectures	128
4.2	Specification of Mode-Dependent Timing Behavior	129
4.2.1	Existing Approaches	131
4.2.2	Basic Concepts	133
4.2.3	Operating Modes	136
4.2.4	Mode-Dependent Timing Specifications	139
4.2.5	Application Example	147
4.2.6	Discussion	153
4.3	Modeling of Dynamic Reconfiguration	155
5	Modeling of Unreliable Communication	159
5.1	Modeling and Analysis of Unreliable Communication	160
5.1.1	Communication Models	161
5.1.2	Timing Analysis of Communication Systems	162
5.2	Communication Standards	163
5.2.1	IEEE 802.11p	163
5.2.2	Cellular V2X	164
5.3	Relevant Time Phenomena	165
5.4	Generic Communication Model	174
5.4.1	Specification Templates	174
5.4.2	Example Parameter Sets	178
6	Mapping of Specifications to Imprecise Local Clocks	181
6.1	Handling of Imprecise Local Clocks	182
6.1.1	Clock Synchronization	182
6.1.2	Globally Asynchronous, Locally Synchronous Systems	184
6.1.3	Detection of Global State Predicates	184
6.1.4	Deterministic Coordination Approaches	185
6.1.5	Modeling of Clock Characteristics	185
6.2	Problem Description	186
6.3	Clock Relations	188
6.3.1	Monotonicity and Progress	191
6.3.2	Clocks with Finite Resolution	192
6.3.3	Clocks with Drift	195

6.3.4	Clocks with Skew	197
6.3.5	Clocks with Difference Constraints	198
6.3.6	Multiple Clocks	200
6.4	Weaker Notions of Consistency	201
6.4.1	Overapproximations	201
6.4.2	Underapproximations	201
6.4.3	Particularities in Contract-Based Design	204
6.5	Timing Specifications	204
6.5.1	Repetition Pattern	205
6.5.2	Reaction Pattern	210
6.5.3	Synchronization Pattern	211
7	Tool Support	213
7.1	Requirements	214
7.2	UACMbox Toolchain	216
7.2.1	Modeling Environments and Data Sources	217
7.2.2	Intermediate Representation and Model Transformations	217
7.2.3	Analysis Tools and Monitoring Frameworks	225
7.3	Tool Architectures and Components	225
7.3.1	MULTIC-Tooling Industrial	226
7.3.2	UACM Toolbox	233
7.3.3	RTana _{2sim}	242
7.4	Evaluation	245
7.4.1	Specification Language Support	246
7.4.2	Performance	247
8	Conclusion	253
8.1	Review of Goals	253
8.1.1	Consistent Formulation of Real-Time Requirements	253
8.1.2	Specification Support for Cooperative Systems	254
8.1.3	Automated Verification of Timing Specifications	256
8.1.4	Modular and Extensible Tool Support	257
8.2	Limitations and Future Work	257
A	Timing Specification Language	261
A.1	Common Definitions	262
A.1.1	Timing Contracts	262
A.1.2	Intervals and Time Expressions	263
A.1.3	Factor Expressions	263
A.1.4	Basic Expressions	263
A.1.5	Data Types	264
A.2	MTSL 1.5+MC	264
A.2.1	Specification Patterns	264
A.2.2	Clock References	265

Contents

A.3	MTSL 1.5+Modes	265
A.3.1	Specification Patterns	266
A.3.2	Event and Variable Expressions	266
A.3.3	Mode Conditions	267
A.3.4	Mode Configurations	267
A.4	MTSL 1.5	268
A.4.1	Specification Patterns	268
A.4.2	Event Specifications	268
A.4.3	Event Occurrence Modalities	270
A.4.4	Timed Relation Modalities	270
A.5	MTSL 1.0	271
A.5.1	Specification Patterns	271
A.5.2	Event Expressions, Events, and Ports	272
A.5.3	Repetition Options	273
A.5.4	Causal Relations	273
B	Modeling Guidelines	275
B.1	Time-Constrained Activity Diagrams	275
B.2	Functional Architectures with Timing Specifications	277
	List of Figures	279
	List of Tables	281
	List of Listings	283
	Bibliography	285

1 Introduction

This thesis deals with the consistent handling of real-time properties in the development of connected and highly automated systems. It presents concepts, methods, and tools for ensuring the real-time behavior of *systems of cyber-physical systems* (SoCPSs), ranging from cooperative driving functions that enable the interaction of individual vehicles to large-scale, cooperative ensembles of connected and highly automated systems. Although all concepts, methods, and tools are developed and demonstrated using application examples from the automotive domain, they can in principle also be applied to systems from other application domains, such as railway, maritime, and aerospace.

The intended target audience is not only the scientific community, but also requirements, development, test, and safety engineers from companies that are faced with challenges in designing, verifying, validating, and approving increasingly complex, connected, highly automated, and cooperative systems. In addition, the results can be used as inputs for certification bodies and standardization committees from the transportation domain.

The first chapter of this thesis begins with the motivation. It then introduces key concepts of cooperative SoCPSs and discusses their typical characteristics as well as particular challenges in ensuring real-time behavior of such systems in Section 1.2. Subsequently, the research questions addressed in this thesis are presented in Section 1.3. Finally, Section 1.4 provides an overview of the structure of this document.

1.1 Motivation

The development of *advanced driver assistance systems* (ADASs) and *automated driving* (AD) functions has been in the media and in the headlines for many years. As a result, the release of the Mercedes Benz DRIVE PILOT [417] and the BMW Personal Pilot L3 [48] as Germany’s first level 3 systems approved by the German *Federal Motor Transport Authority* (KBA) [220] was met with great public interest and also attracted a great deal of attention in the international press. The colloquial description of the approval of these level 3 systems as “*moon landing*” [245] shows the enormous emotional significance and the importance of achieving this milestone for the automotive industry.

Today’s technical capabilities are based on years of evolution from comparatively simple driver assistance systems to *automated driving systems* (ADSs). The capabilities of systems to assist human drivers, up to and including a complete takeover of the driving task, can be characterized by using the six levels of driving automation defined by the *Society of Automotive Engineers* (SAE) [513], which are depicted in Figure 1.1. The included levels are generally accepted and represent a widely used classification of driving functions, ranging from level 0 (“*No Driving Automation*”) to level 5 (“*Full*

1 Introduction

Driving Automation”). As explained earlier, today’s top-of-the-range ADSs, like the two systems mentioned above, meet the requirements of SAE level 3 (“*Conditional Driving Automation*”). In this case, the AD functions are capable of performing the driving task under limited conditions and can request the human driver to take over control if the requisite conditions may no longer be met within the predictable time horizon.



SAE J3016™ LEVELS OF DRIVING AUTOMATION™

Learn more here: [sae.org/standards/content/j3016_202104](https://www.sae.org/standards/content/j3016_202104)

Copyright © 2021 SAE International. The summary table may be freely copied and distributed AS-IS provided that SAE International is acknowledged as the source of the content.

		SAE LEVEL 0™	SAE LEVEL 1™	SAE LEVEL 2™	SAE LEVEL 3™	SAE LEVEL 4™	SAE LEVEL 5™
What does the human in the driver's seat have to do?		You are driving whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering			You are not driving when these automated driving features are engaged – even if you are seated in “the driver’s seat”		
		You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
Copyright © 2021 SAE International.							
		These are driver support features			These are automated driving features		
What do these features do?		These features are limited to providing warnings and momentary assistance	These features provide steering OR brake/acceleration support to the driver	These features provide steering AND brake/acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met	This feature can drive the vehicle under all conditions	
Example Features		• automatic emergency braking • blind spot warning • lane departure warning	• lane centering OR • adaptive cruise control	• lane centering AND • adaptive cruise control at the same time	• traffic jam chauffeur	• local driverless taxi • pedals/steering wheel may or may not be installed	• same as level 4, but feature can drive everywhere in all conditions

Fig. 1.1: Levels of driving automation [511]

In the future, it is very likely that the trend towards increasing automation of driving functions will continue. Due to the increasing automation, also the consequences of failures will become more severe. This results in particular from the fact that—in contrast to lower degrees of automation—not only individual subtasks, but the entire driving task is (almost) completely taken over. In addition, the controllability by human drivers is more and more limited, since the possibilities for human intervention are gradually removed. This means that a driver is only available as a fallback in critical situations with a longer delay—or not at all—in automation levels starting at level 3. The absence of a human driver as a constantly available fallback option is leading to paradigm shift from the development of *fail-safe* to *fail-operational* systems [561]. This transition leads to a demand for additional system properties, such as increased redundancy of safety-critical system functions and availability of degraded system states. The measures taken are aimed at achieving a sufficient level of safety to maintain system operation after the occurrence of faults, while increasing the complexity of system design and safety argumentations.

Highly automated driving in urban areas is considered to be the top of the class. It reinforces the resulting challenges in the perception, evaluation, and handling of dynamic traffic situations as well as the anticipatory planning of suitable driving maneuvers. The operating domain in inner-city traffic is usually rather unrestricted, so that highly automated vehicles are confronted with, for example, hidden road users, poorly visible curves and intersections, limited self-localization due to attenuated *global navigation satellite system* (GNSS) signals, and the lack of a hard shoulder as a “safe harbor”. From a real-time perspective, highly automated driving in urban traffic requires a particularly strict consideration of timing behavior and end-to-end latencies such as, for example, from the detection of potential hazards until appropriate safety measures (e.g., evasive or emergency braking maneuvers) become effective. In this context, timeouts and deadline violations can have catastrophic effects, leading to serious consequences for the lives or health of people involved, lasting damage to the environment, and high costs.

The design of highly automated vehicles requires rigorous development processes and analysis methods that help engineers to create solid safety argumentations. In many application domains, standards exist that define process steps and artifacts required to achieve such argumentation. ISO 26262 [324], for example, is a standard that aims to ensure functional safety of automotive systems. It mandates the identification of potential hazards and risks imposed to users and their environment as part of a *Hazard Analysis and Risk Assessment* (HARA). Top-level safety requirements and *Automotive Safety Integrity Levels* (ASILs) are derived as input for the definition of the functional safety concept to mitigate the identified risks. This includes performing structured cause analyses such as *fault tree analyses* (FTAs) and *failure mode and effects analyses* (FMEAs). Subsequently, a technical safety concept is defined that realizes the required safety mechanisms. The standard mandates that all steps must be covered by analyses to ensure that the system actually adheres to the safety goals. As the initial requirements are iteratively refined along with the system design, analyses ensure that the refined requirements can be traced back to the top-level requirements in a verifiable way to allow statements about their compliance. By integrating the analysis results into a *safety assurance case* (SAC) [84, 470], they can be used as evidence to prove compliance with safety goals or to substantiate (sub-)claims. The purpose of the SAC is to demonstrate that all possible faults have been identified during system design and are effectively mitigated by the overall system.¹

Not least due to the high degree of automation and the associated necessity to take into account a multitude of different driving scenarios in highly dynamic environments, expert groups from a wide variety of disciplines must work together to ensure a safe, reliable, and secure realization of ADSs. Since expert knowledge is domain knowledge, each team usually has its own concepts and tools that are particularly suited to describe parts of the system under design. Nevertheless, only a few methodologies exist to take appropriate account of timing behavior and to support the integration of all building

¹An earlier version of this paragraph has been published in [58]. The corresponding passage in the original publication was written in close collaboration with Jan Steffen Becker and Ingo Stierand. It was subsequently revised by me to fit into the new context.

1 Introduction

blocks into a common system. As a result of the increasing complexity, the challenges in handling real-time properties are manifold. They can be divided into two problem areas.

The first problem area consists of the challenges in ensuring real-time behavior of ADAS and AD functions of individual vehicles. These systems are typically based on layered architectures [92]. While the lower-level layers belong to the “classical” reactive control domain, connected and highly automated driving functions also subsume higher-level control regimes, requiring complex sensor processing, feature extraction, and the maintenance of environment models [361]. Components in these layers are typically developed by different teams with different tools for different functional purposes. The resulting heterogeneity causes discontinuities between specification, implementation, and verification, entailing high design and debugging efforts for real-time properties.

Problem area 1 is already partially addressed by industrial tools that provide support for various engineering tasks [310, 589] as well as existing research on how to connect these tools in integrated, standard-compliant design processes [18, 579]. By combining state-of-the-art design methods and analysis tools, engineers are able to ensure real-time behavior of single systems. However, current design approaches lack support for some aspects that will become more important in the context of highly automated driving and software-defined vehicles. These include, for example, the explicit specification of operating modes and mode-dependent timing behavior at high abstraction levels. In addition, the mapping of specifications to imprecise, local clocks is typically not supported. Chapter 2 is dedicated to the state of the art and gives an overview of existing approaches. As we will see, they can further benefit from the results of this thesis.

Current research give reason to assume that cooperation of *connected and highly automated vehicles* (CHAVs) with each other, conventional vehicles, cyclists, and pedestrians can increase traffic efficiency and road safety [17, 242, 546, 617]. For this reason, it is likely that in the future not only assuring the correct and timely functioning of individual vehicles, but also the safe interaction of different road users will be an important aspect. However, there is currently a lack of suitable methods and tools to ensure real-time behavior of SoCPSs. In this context, the second problem area comprises challenges in the consistent specification and verification of real-time properties of cooperative systems.

Since safety-critical driving decisions of CHAVs are no longer made exclusively on the basis of the vehicles’ own situational overviews, but are instead based on cross-system interaction processes, enhanced specification and verification means are required to cover relevant aspects. Moreover, suitable extensions to state-of-the-art design processes are needed to account for cooperative system behavior. The planned enhancements aim to facilitate early collaboration between manufacturers and suppliers to develop consistent top-level specifications in early design phases that enable safe and reliable cooperation of automated systems. In order to achieve the desired SoCPS-wide timing coherence, a number of methodological and technical challenges need to be solved, which are detailed in Section 1.2. They form the basis for deriving the research questions in Section 1.3 and are addressed by the contribution of this thesis, which is presented in Section 2.4.

1.2 Cooperative Cyber-Physical Systems

From a system-theoretic perspective, highly automated vehicles can be considered as *cyber-physical systems* (CPSs), i.e., control systems that integrate “*computation and physical processes*” [384, p. 363], using “*embedded computers and networks [to] monitor and control [...] physical processes*” [384, p. 363] in real-time. In these systems, “*physical and software components are deeply intertwined, each operating on different spatial and temporal scales, exhibiting multiple and distinct behavioral modalities, and interacting with each other in a myriad of ways that change with context*” [440].

Consequently, cooperative ensembles of CHAVs can be regarded as SoCPSs, which are a special type of *systems of systems* (SoSs). According to ISO/IEC/IEEE 21839, SoSs are defined as “*set of systems or system elements that interact to provide a unique capability that none of the constituent systems [...] can accomplish on its own*” [325]. Note that this one of many commonly used definitions [328, 604]. However, it is a representative one and illustrates the basic idea. Interested readers are referred to [448, p. 18:34], which provides an overview of existing SoS definitions. We will take a closer look at some of these definitions to identify typical characteristics of such systems in Section 1.2.2.

In the context of SoSs, individual (sub-)systems are usually referred to as *constituent systems* (CSs) [344, 432, 448, 547]. These CSs are generally independent, but “*can be part of one or more SoS*” [325]. Each CS is a “*useful system by itself, having its own development, management, utilization, goals, and resources, but interacts within the SoS to provide the unique capability of the SoS*” [325]. In the following, we consider CHAVs as CSs of a higher-level cooperative SoCPS, which may also include other CSs such as intelligent infrastructure systems that are involved in cooperative processes.

In general, the term *cooperation* is defined as *collaboration to achieve common goals* or, in the context of SoCPSs, as “*collaboration [...] to achieve a common goal through the mutual provision of functions that individual systems alone cannot achieve*” [94, p. 19]. In the context of this work, *cooperative systems* are understood as SoCPSs, whose CSs interact to achieve common high-level goals, e.g., improving road safety or increasing traffic efficiency, and individual lower-level goals, e.g., the execution of a certain driving maneuver that is required to achieve a specific higher-level goal. Note that we focus on the interaction of technical systems. Human-machine interactions between ADSs and human drivers (or passengers), which can also be understood as a form of cooperation, is abstracted below. We thus follow the idealized assumption that both driving automation and human participants will strive for consensus and reach it at the appropriate times.

To simplify the description of cooperative processes, we introduce the concept of *cooperative system groups* (CSGs). CSGs are defined as cooperative groups of CSs that are part of a higher-level SoCPS and have, for example, similar or complementary behavioral intentions (e.g., driving in the same direction with a zipper merge ahead, crossing an urban intersection in the opposite direction). Both SoCPSs and CSGs can generally be considered as *dynamic* systems. This means that both the number of individual subsystems as well as the composition of those subsystems can evolve over time, i.e., CSs may enter and leave CSGs and SoCPSs at defined, non-zero time points.

1 Introduction

In addition, we focus on *mobile* systems, which means that at least some of the CSs may change their position in the global coordinate system during interaction.

Cooperative systems as defined in this thesis are widely used in almost all application domains, such as collaborative transport robots in manufacturing, automated container terminals and fleet management systems in the maritime domain, the *European Train Control System* (ETCS) and cooperative train control in the rail domain as well as drone fleets and satellite control systems in the aerospace domain. As mentioned above, we will focus on cooperative driving systems from the automotive domain.

1.2.1 Cooperative Driving: A Vision of Cooperative Systems

Future mobility will be increasingly automated, connected, and cooperative [219]. While road traffic will presumably continue to play an important role, it will undergo a fundamental transformation, driven by research and industrial developments over the last decades aimed at developing CHAVs and intelligent transportation infrastructure [390]. Effects such systems are promising include an increased traffic efficiency and road safety, reduced noise pollution, environmental advantages, and the rise of new mobility services, such as *demand-responsive transport* (DRT). The emerging transportation systems will include a mixture of CHAVs, conventional vehicles with human drivers, potentially assisted by ADASs, as well as *vulnerable road users* (VRUs) such as cyclists and pedestrians.²

The SAE has introduced a taxonomy of cooperative driving automation functions [512] that can be used to categorize cooperative driving systems. It distinguishes between four classes of cooperation based on their purpose and their binding nature [512, pp. 5–7]:

Status-Sharing Cooperation Sharing of “[p]erception information about the traffic environment and information about the sending entity [...] for potential utilization by receiving entities” (Class A)

—“Here I am, and here is what I see.”—

Intent-Sharing Cooperation Exchange of “[i]nformation about planned future actions of the sending entity [...] for potential utilization by receiving entities” (Class B)

—“This is what I plan to do.”—

Agreement-Seeking Cooperation “[S]equence of collaborative messages [...] intended to influence local planning of specific [driving task]-related actions” (Class C)

—“Let’s do this together.”—

Prescriptive Cooperation “[D]irection of specific action(s) to specific traffic participants for imminent performance of the [driving task], provided by a prescribing [entity] and adhered to by a receiving [entity]” (Class D)

—“I will do as directed.”—

²An earlier version of this paragraph has been published in [362]. The corresponding passage in the original publication was written by me and was finalized with editorial changes by my co-authors.

In the further course, we will mainly deal with cooperative systems of classes C and D, as they are the only ones in which the behavior of the CSs involved is directly affected. Nevertheless, the concepts developed can also be applied to classes A and B. They are understood as a basis for establishing cross-system situational overviews and provide important information for the execution of cooperative driving maneuvers.

We will use the selected classes of cooperative driving applications as examples of cooperative systems, whose development is driven by the automotive industry. They have also been investigated in numerous research projects such as Ko-HAF [357], Digitaler Knoten 4.0 [185], IMAGinE [309], ViVre [591], and KoKoVI [358].

Use Cases and Applications

In recent years, almost all working groups and standardization committees active in this area have defined common use cases for cooperative driving [1, 123, 216]. In addition, key functionalities required for the realization of such use cases were derived [121]. The development of cooperative driving functions is highly dependent on the availability of appropriate communication means. Corresponding vehicle-to-everything (V2X) applications are generally divided into three generations called “*days*” [46, 122]:

Day 1 – Cooperative Awareness Exchange of vehicle information such as location, speed, and trajectory, as well as status information and warnings about unexpected events

Day 2 – Cooperative Perception Sharing of information from sensors such as cameras, radars, and lidars; infrastructure support for prioritizing special vehicles

Day 3+ – Cooperative Driving Exchange of driving intentions; cooperative planning and execution of driving tasks, including the coordination of maneuvers to avoid conflicts with other, non-connected and/or non-automated road users

The three generations can be thought of as deployment phases, as they build upon each other. They are related to the SAE J3216 classes. For example, Day 1 and Day 2 V2X applications are sufficient for the implementation of class A cooperative driving functions only. For all higher cooperation classes, Day 3+ generation V2X solutions are required.

Botte et al. provide an overview of the current state of implementation of V2X applications and cooperative driving in the *European Union* (EU) [99]. The implementation and rollout of such services depends on many factors, such as local political and organizational structures, the affinity of decision-makers for technology, and the regulatory context. Currently, corresponding services are only available in test fields and model regions. Nevertheless, the *5G Automotive Association* (5GAA), which is a global, cross-industry organization of automotive, technology, and telecommunications companies, expects mass deployment of safety-critical cooperative driving applications by the end of this decade [1]. In the following, we will take a closer look at use cases and applications of cooperative driving to gain a better understanding of the application domain.

Cooperative Driver Assistance Systems Simple cooperative driving applications of SAE J3216 classes A and B based on Day 1 V2X solutions have already reached series production maturity. The *Audi Green Light Optimized Speed Advisory* system [42], for example, is an enhanced version of the *Audi Traffic Light Information* system [43] and provides *Green Light Optimized Speed Advisory* (GLOSA) information. Another example is the *Volkswagen Traffic Hazard Alert* system [594] that issues and receives warnings about active safety systems in vehicles ahead, stationary or broken-down vehicles as well as accidents, traffic jams, or construction sites. It also warns drivers of approaching emergency vehicles. In both cases, traffic light information and warnings are provided to a human driver, who can then react (or not) to the traffic situation, as highly automated driving functions are not yet available on the market. Nevertheless, these systems represent a first step towards the introduction of cooperative driving functions on public roads.

In addition to the already mature systems of lower cooperation classes, experimental applications of classes C and D are being tested on test tracks and in real traffic. Existing prototypes include, *Cooperative Adaptive Cruise Control* (CACC) systems for CHAV platooning [17, 350, 481], highway lane reduction and urban intersection crossing scenarios [482] as well as active safety systems such as a *Cooperative Active Blind Spot Assistant* (CABSA) [524]. At the same time, the academic world is already investigating more sophisticated use cases such as cooperative vehicle-intersection systems [64, 80, 625].

Recent developments provide the basis for harmonizing and standardizing cooperative driving applications. In Europe, for example, numerous industry-driven standardization activities in the area of communication architectures for cooperative driving exist, such as in the context of the *European Telecommunications Standards Institute* (ETSI) and the *CAR 2 CAR Communication Consortium* (C2C-CC). Moreover, the standardization of system architectures across manufacturers through the definition of reference architectures is an important step in facilitating the use of cooperative driving functions [62]. In recent years, for example, reference architectures for connected and cooperative driving applications have been developed and made available to automotive stakeholders that set the stage for *cooperative intelligent transport systems* (C-ITSs) [237, 361, 364].

Cooperative Intelligent Transport Systems³ The use of CHAVs and intelligent infrastructure systems in urban areas can make an important contribution to solving social, economic, and environmental challenges by enabling the application of coordination and optimization techniques [390]. A first step in this direction is to support CHAVs with infrastructural sensors and communicating information systems to increase sensing ranges and reliability [119]. This enables connected road users to receive real-time information about the traffic situation in remote and poorly visible areas [238]. In addition, traffic light phases and future signal courses can be transmitted [596]. The digitization and virtualization of traffic infrastructure elements [578] could help to realize infrastructure-free traffic information systems [334] and to introduce virtual traffic signs to make the complex and error-prone process of vision-based traffic sign detection [515, 597] obsolete.

³An earlier version of this section has been published in [362]. The corresponding passage in the original publication was written by me and was finalized with editorial changes by my co-authors.

Equipped with V2X transceivers and environmental sensors, specialized systems like *traffic management systems* (TMSs) can ultimately generate behavior or cooperation recommendations to actively support CHAVs in their safe and efficient journey [363].

A common vision in transportation research is to lift local coordination approaches to higher levels. In this context, the ultimate goal is to establish multi-level C-ITSs that are capable of coordinating the interplay of large numbers of automated and non-automated actors in cities or regions. As part of the KoKoVI project [358], building blocks for such systems were designed, implemented, and tested along relevant use cases. Figure 1.2 shows a sequence of three example scenarios derived from these use cases. Here, we focus on a CHAV that is part of a DRT service and interacts with other road users.

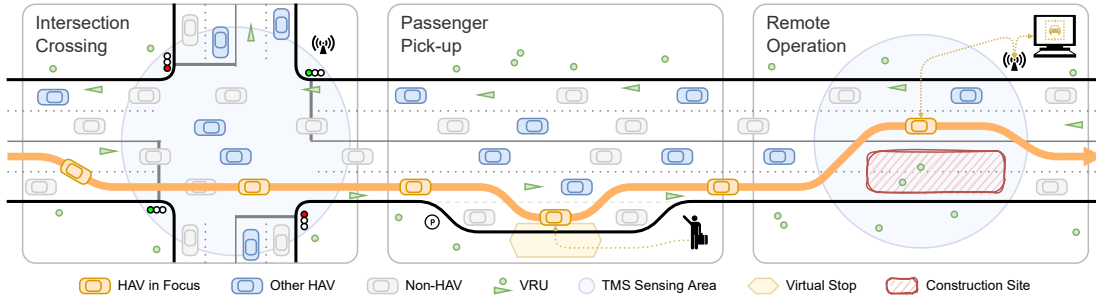


Fig. 1.2: Sequence of example driving scenarios

Intersection Crossing The first scenario is the automated crossing of an urban intersection. While it approaches the intersection, the CHAV in focus informs the TMS of its arrival via V2X messages to enable an early and reliable coordination. The reception of the subscription message is confirmed by sending a detailed response containing the current signal phase of the traffic lights, their future signal courses, a digital map of the intersection as well as information about static obstacles and dynamic objects detected by the infrastructure sensors. The TMS makes use of all available information to identify potential conflict situations between CHAVs. Based on this, cooperative driving maneuvers of CHAVs are initiated by sending out cooperation recommendations, e.g., for cooperative lane changes and left-turn maneuvers [363]. In addition, CHAVs and infrastructure systems interact via suitable *human-machine interfaces* (HMIs), i.e., internal/external HMIs, traffic lights, and variable message signs, with non-automated road users to minimize collision risks and to increase traffic efficiency.

Passenger Pick-up In the second scenario, the CHAV in focus picks up a passenger at a virtual stop located in a parking bay. The vehicle is part of a DRT service that supports the pick-up and drop-off of passengers at virtual stops, i.e., stops without physical infrastructure agreed upon as meeting points during booking [277, 578]. After sending a trip request via the user's mobile device, a disposition system handles the incoming request and disposes a fleet of specialized CHAVs that serve as shuttle vehicles. It coordinates the communication with the shuttle and initiates the user's navigation to

1 Introduction

the virtual stop [304]. During the trip, user guidance is required. This includes status information about the planned route, the progress of the trip, and unexpected events.

Remote Operation The third scenario focuses on the support of the CHAV by a human remote operator in unknown or critical situations that cannot be handled by the vehicle’s automation. Each CHAV continuously monitors its own condition, its environment, and the behavior of other road users to assess its capabilities in handling the traffic situation. If leaving the *operational design domain* (ODD) is likely in near future, countermeasures might be taken to ensure safe operation:

1. Adaptation of the vehicle’s route (if possible)
2. Execution of a minimal risk maneuver [342, 494]
3. Transfer of the driving task to (external) infrastructure [528]
4. Remote vehicle control by a human operator [348, 349]

In the example scenario, the need to pass through oncoming traffic to avoid a construction site is detected early by the CHAV. Assuming that this behavior is not supported by the vehicle’s automation, control is transferred to a human remote operator, who releases a pre-planned (or manually defined) trajectory and monitors the execution of the evasive maneuver. After the safe passage through the unmanageable area, control is returned to the CHAV in focus, which continues its journey through urban traffic.

Application Example: Cooperative Left-Turn Assistant⁴

Intersections are a key element of today’s transportation infrastructure and have a significant impact on inner-city traffic efficiency. At intersections, traffic flows from different directions meet. These flows comprise different road users—each with individual goals and driving styles. Depending on the active traffic light phase, conflict areas with other road users arise while crossing the intersection. Figure 1.3 shows an example of these areas resulting from a left-turn scenario. After passing through oncoming traffic ①, the orange-colored vehicle has to consider crossing cyclists and pedestrians ②.

In [363], we have presented a decentralized, game-theoretic approach for negotiating cooperative maneuvers of CHAVs at urban intersections. To support the initiation of such maneuvers, we assumed the existence of an intelligent TMS with a global view on the whole scenery in order to derive cooperation recommendations without directly controlling the CHAVs. This combination allows for mutually consistent, distributed decision making on cooperative driving maneuvers, taking into account higher-level and lower-level goals.

The developed cooperation approach addresses the *Intersection Crossing* use case presented in the previous section and is classified as *Agreement-Seeking Cooperation* (Class C) according to SAE J3216 [512]. It supports a wide range of application scenarios,

⁴An earlier version of this section has been published in [363]. The corresponding passage in the original publication was written by me and was finalized with editorial changes by my co-authors.

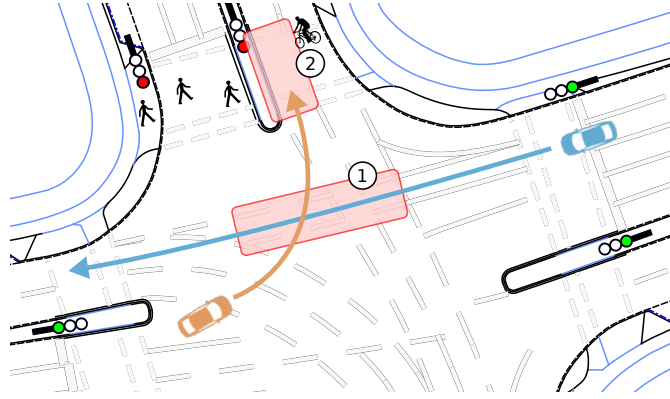


Fig. 1.3: Example conflict areas [363]

in which it can provide benefits to road users, cities, and municipalities. In the further course, a *Cooperative Left-Turn Assistant* (CLTA), whose basic principle is illustrated in Figure 1.4, is selected as an example to demonstrate the approach. It was extensively studied as part of the *Digitaler Knoten 4.0* project [185, 360] and will be used as a running example to apply the developed concepts, methods, and tools in practice.

In the first scenario, a CHAV, referred to as CHAV 1 (orange), is approaching an urban intersection and is about to make a left turn. At the same time, a second vehicle, referred to as CHAV 2 (light blue), is approaching in oncoming traffic and wants to drive straight across the intersection. Both vehicles subscribe to updates from an intelligent TMS that is responsible for supporting CHAVs in a safe and efficient crossing (see Figure 1.4a). The TMS observes the situation and determines that traffic efficiency could be increased by CHAV 1 turning in front of CHAV 2. It thus recommends to perform a cooperative left turn. While approaching the conflict area, which is indicated by the red area, both CHAVs evaluate the recommendation based on their own situational overviews (see Figure 1.4b). Subsequently, they jointly decide whether or not to execute the proposed maneuver. After the vehicles have decided to perform the maneuver, the cooperative left turn is executed safely and efficiently by their ADSs (see Figure 1.4c), resulting in shorter waiting times and increased road safety. Finally, the two CHAVs unsubscribe from updates from the TMS while leaving the intersection area and continue their journey through urban traffic (see Figure 1.4d). Note that Figure 1.4 is only schematic, i.e., not all dimensions and distances are to scale. In real-world scenarios, both vehicles would interact with the TMS much earlier. This would allow enough time to detect potential resource conflicts and send and evaluate the cooperation recommendation.

Relevant Actors and Roles The CLTA application example involves four main actors, which can be considered as CSs of the cooperative SoCPS and contribute to the scenario in different roles. In addition to the V2X communication channel, which is seen as the basis for coordination and cooperation, two CHAVs and one TMS are involved:

1 Introduction

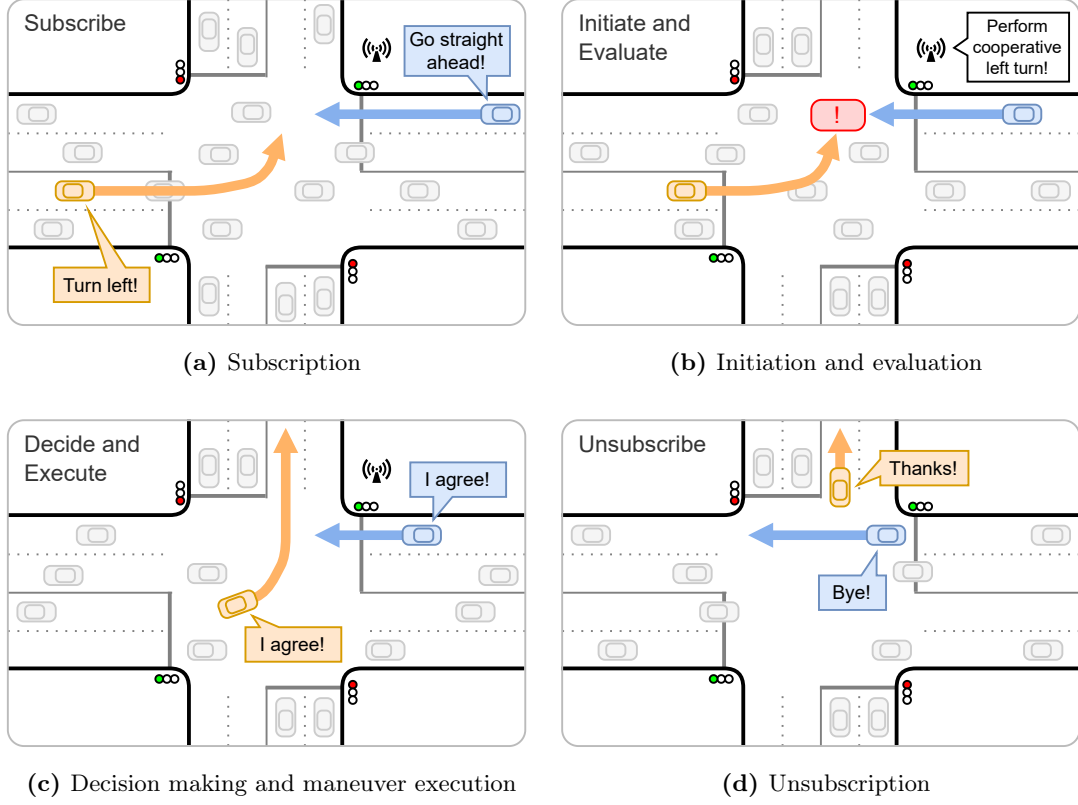


Fig. 1.4: Cooperative left-turn maneuver (based on [363])

1. A vehicle designated as *CHAV 1* that is about to make a left turn and takes on the role of the LEFT-TURNING VEHICLE. It is also referred to as *supported CHAV*.
2. A vehicle designated as *CHAV 2* that wants to drive straight ahead and takes on the role of the STRAIGHT-AHEAD VEHICLE. It is also referred to as *supporting CHAV*.
3. A *TMS* that initiates and supervises the cooperative maneuver.

The assumptions about the CHAVs and the TMS are explained below.

Connected and Highly Automated Vehicles The CHAVs in focus are assumed to have automation levels from 4 to 5 [513]. Thus, their individual driving tasks are (almost) completely taken over by their ADS. Each CHAV has the capability to send and receive V2X messages and is equipped with an appropriate set of environmental sensors. Moreover, a high-accuracy self-localization and state-of-the-art vehicle sensors are considered prerequisites for highly automated operation in urban traffic.

Each CHAV operates at a maximum speed of 50 km/h and complies with all traffic regulations. Due to the high degree of automation and the lack of human reaction time, a CHAV should adapt to external events in significantly less than 1 s. Since conservative

estimates of distances and time gaps are used, such vehicles drive very defensively and with foresight. All CHAVs behave prudently when interacting with other road users.

Traffic Management System As explained above, the use of an intelligent TMS is a key element of the presented approach. In our vision, such a system has the capability to send and receive V2X messages with a minimum reception range of 200 m. In addition, we assume that it is equipped with environmental sensors that detect, classify, and track static obstacles and dynamic objects within a sensing range of 150 m in order to maintain a real-time situational overview and predict the future behavior of traffic participants. As a third source of information, a TMS should be linked to the intersection's traffic light system, which provides access to signal phases and future signal courses.

Based on its situational overview, the TMS analyzes the current and anticipated traffic situation in the intersection area. A central decision making component brings intelligence to the TMS and takes all high-level control decisions related to the flow of mixed traffic. In addition, it transmits infrastructure-to-vehicle (I2V) messages, such as behavior recommendations and hazard warnings, controls the infrastructure actuators (e.g., variable message signs), and provides evaluation services for traffic analysis.

Time Subdivision Into Phases In order to structure the overall cooperation approach and to enable a detailed analysis of the resulting traffic situations, the journey of each CHAV through the intersection area is divided into three so-called cooperation phases, namely a *subscription phase*, an *execution phase*, and an *unsubscribe phase*.

Subscription Phase The aim of the first phase is to improve the situational awareness of the CHAV in focus and the TMS by communicating vehicle-specific data and exchange information about the traffic situation to set the stage for cooperative maneuvers. In a first step, a unique vehicle identifier, the driving intention (e.g., turn left, go straight ahead, turn right), and the intended destination lane are sent to the TMS. The reception of the message is confirmed to the CHAV by sending a detailed response containing the current signal phase of the traffic lights, their future signal courses, a digital map of the intersection as well as information about static obstacles and dynamic objects.

According to the assumptions made, the subscription phase must be completed at least 150 m before the stop line of the corresponding lane. If the subscription of a CHAV is not completed in time, the crossing must take place without benefiting from the additional services provided by the TMS, including support in cooperative maneuvers.

Execution Phase From the perspective of an individual CHAV, the primary aim of the execution phase is to cross the intersection safely and as efficiently as possible. During the rule-compliant approach of the CHAV to the stop line, the interaction with other road users takes place as it would do even without the existence of an intelligent TMS. Based on the information acquired in the subscription phase, however, an earlier decision can be made for or against passing the intersections traffic lights.

1 Introduction

At the same time, the TMS aims to increase the intersection's capacity, prevent congestion, and reduce emissions. It makes use of all available information about the current positions, velocities, driving intentions, and the planned destination lanes of the subscribed CHAVs and other participants, its knowledge about the intersection geometry, static obstacles as well as dynamic objects, and identifies potential conflict situations between CHAVs. In a second step, the TMS tries to derive feasible solutions to these conflicts that increase traffic efficiency, e.g., through behavioral predictions and the recognition of previously learned patterns within the traffic scene. Subsequently, promising solutions are proposed to the participating CHAVs in the form of cooperation recommendations. After receiving the recommendation, the CHAVs evaluate the traffic situation based on their own situational overviews and exchange their evaluations, which may or may not lead to a joint decision to execute the proposed cooperative maneuver.

Unsubscription Phase In the course of the unsubscription phase, the information collected during the crossing of the intersection is fed back to the TMS. By receiving the periodically sent position and velocity updates of all leaving CHAVs, further information can be obtained on traffic density and the likelihood of congestion in the individual exits. After leaving the reception area of the V2X transceiver, each CHAV is indirectly unsubscribed by its removal from the situational overview.

Cooperation Procedure We propose a decentralized cooperation approach for the negotiation of cooperative maneuvers. For this purpose, the TMS serves as a central source of information and generates recommendations that actively supports the approaching CHAVs as they cross the intersection. At the same time, it involves other traffic participants using suitable sensors and actuators. The interacting CHAVs agree decentrally on recommended driving maneuvers or decide against their execution.

Figure 1.5 provides an overview of the intended communication flow. Following the distribution of the *cooperation recommendation* to the CHAVs ①, an assessment of the proposed cooperative maneuver and the resulting strategy combinations is performed locally. Each vehicle then communicates the results of its assessment by means of an *evaluation* ②. The messages sent are also received by the TMS and can be used to monitor the cooperation process and improve behavioral predictions. As a final step, the decision for or against the execution of the cooperative driving maneuver is made ③.

Cooperation Recommendation As already described in Section 1.2.1, the TMS makes use of all available information about the current positions, velocities, driving intentions, and the planned destination lanes of traffic participants, its knowledge about the intersection geometry, static obstacles as well as dynamic objects, and identifies potential conflict situations between CHAVs to be solved. When a situation is identified in which an efficiency gain can be expected by executing a cooperative driving maneuver, the first step is to check whether the vehicles involved are CHAVs. If so, their unique identifiers are stored to prepare the message exchange. Subsequently, suitable strategies for both vehicles are identified or extracted from a strategy catalog based on expert knowledge

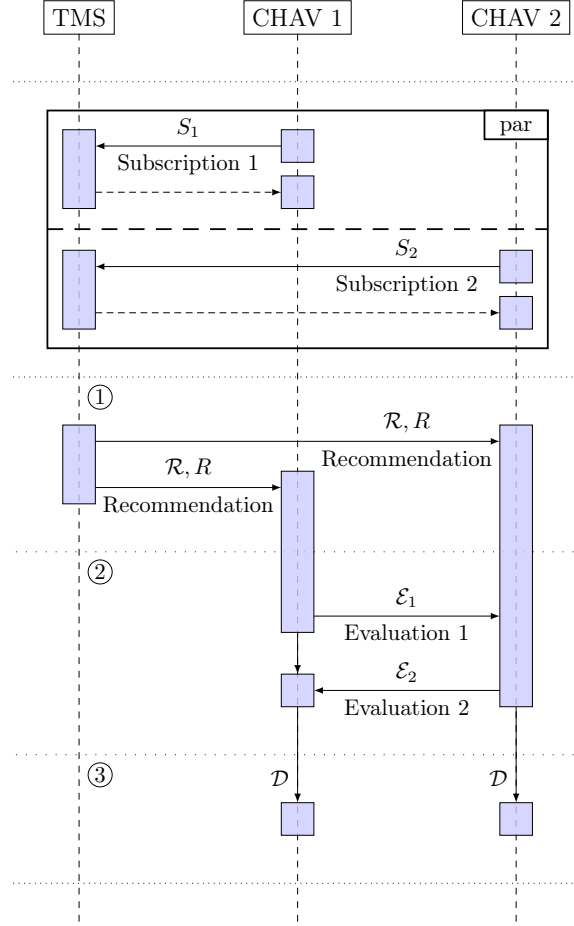


Fig. 1.5: Time sequence of communication

and feedback from vehicles that have already passed the intersection. A strategy consists of a pair of (abstract) longitudinal and lateral behavior descriptions, which can be derived directly from the cooperative maneuver and the given traffic situation.

With regard to the example scenario shown in Figure 1.4, CHAV 1 is recommended to perform a cooperative left turn with CHAV 2. Table 1.1 contains a list of example strategies proposed by the TMS that can be used to solve the conflict situation from the perspective of CHAV 1. On the other hand, CHAV 2 is recommended to support the intended left turn of CHAV 1. Again, different strategies are proposed by the TMS, which are listed in Table 1.2. Note that in real-world scenarios, additional strategies may be relevant, such as turning left after the supporting cooperation partner has received a red light, or performing a left turn with simultaneous acceleration of both CHAVs.

Following the derivation of suitable strategies for both CHAVs, an initial payoff matrix $\mathcal{R}$ is generated, which is shown in Table 1.3. Each of its cells represents a strategy

1 Introduction

Tab. 1.1: Proposed strategies for CHAV 1

<i>ID</i>	<i>Longitudinal Behavior</i>	<i>Lateral Behavior</i>
S1.1	Continue	Turn left
S1.2	Decelerate	Turn left

Tab. 1.2: Proposed strategies for CHAV 2

<i>ID</i>	<i>Longitudinal Behavior</i>	<i>Lateral Behavior</i>
S2.1	Decelerate	Continue
S2.2	Continue	Continue

combination that includes one strategy for each cooperation partner. At this point, all entries are empty because the strategy combinations have not yet been evaluated.

Tab. 1.3: Initial payoff matrix

$\mathcal{R}$		CHAV 2	
		S2.1	S2.2
CHAV 1	S1.1	(-, -)	(-, -)
	S1.2	(-, -)	(-, -)

As a final step, the initial payoff matrix $\mathcal{R}$ as well as the recommended strategy combination R , which represents the most valuable solution with respect to higher-level goals (i.e., increasing traffic efficiency and road safety), are transmitted to the CHAVs.

Evaluation and Decision Making After receiving the recommendation, an independent, in-vehicle evaluation of all strategy combinations proposed by the TMS is initiated by both CHAVs. For this purpose, each cooperation candidate evaluates the solutions based of its own situational overview. In addition to the individual velocities, the distances between the vehicles and their differential speeds, the predicted future behavior of dynamic objects as well as lower-level goals of the CHAV are considered to assess the suitability of strategy combinations. While the evaluation approach is manufacturer- and implementation-specific, the payoffs can be understood as a function of the results of two separate safety and efficiency evaluations. The calculated payoffs are then filled into the matrix entries as illustrated in Table 1.4, resulting in two half-filled payoff matrices.

Once all strategy combinations have been evaluated, the payoff matrices $\mathcal{E}_1$ and $\mathcal{E}_2$ are exchanged to merge them into two complete and consistent copies of matrix $\mathcal{D}$, one for each vehicle. In order to make the game fair and safe, communication must take place (almost) simultaneously. For the practical implementation of this approach, it is crucial

Tab. 1.4: Half-filled payoff matrices resulting from maneuver evaluation

(a) Payoff matrix for CHAV 1			(b) Payoff matrix for CHAV 2		
$\mathcal{E}_1$	CHAV 2		$\mathcal{E}_2$	CHAV 2	
	S2.1	S2.2		S2.1	S2.2
CHAV 1	S1.1	(4 , -)	CHAV 1	S1.1	(- , 2)
	S1.2	(-2 , -)		S1.2	(- , -2)
		(1 , -)			(- , 1)
					(- , 0)

that no prematurely received evaluations of other players must be used as a basis for one's own evaluation. In terms of game theory, it thus has to be a *static* game.

Tab. 1.5: Merged payoff matrix for decision making

$\mathcal{D}$		CHAV 2	
		S2.1	S2.2
CHAV 1	S1.1	(4 , 2)	(3 , -2)
	S1.2	(-2 , 1)	(1 , 0)

In Table 1.5, the determination of the Nash equilibrium [439] for the previously merged payoff matrix $\mathcal{D}$ is shown. For this purpose, the strategy combination with the highest payoffs for both cooperation candidates is selected. If a Nash equilibrium is found (condition 1) that corresponds to the original cooperation recommendation (condition 2), a clear, joint solution has been found. If condition 1 does not hold, the cooperation is rejected for safety and fairness reasons. If condition 1 is fulfilled, but condition 2 is not, the maneuver may still be executed to satisfy the lower-level goals, even though it may not contribute to achieving the higher-level goals as originally proposed by the TMS.

Execution of Cooperative Maneuvers The determination of a Nash equilibrium is defined as the signing of a binding contract for the execution of the agreed strategy combination. Following an update of the situational overview, the planned cooperative maneuver is executed using in-vehicle trajectory planning and control algorithms. In order to achieve a higher safety level during execution, time bounds or reserved lane areas, for example, could be agreed in advance. Further cooperative maneuvers may be recommended by the TMS until the participating CHAVs leave the intersection area.

1.2.2 Characteristics of Cooperative Cyber-Physical Systems

In order to derive the challenges in ensuring real-time behavior of cooperative SoCPSs, we will first investigate typical characteristics of such systems. As already explained in

1 Introduction

Section 1.2, cooperative SoCPSs are a special type of SoSs. Using the OMGEE acronym [402, 448], SoSs can be characterized by the following five features:

Operational Independence *“Any system that is part of an SoS is independent and is able to operate serviceably if the SoS is disassembled.”*

Managerial Independence *“Despite collaborating with the other members of the SoS, the individual systems are self-governing and individually managed so that they ‘not only can operate independently, they do operate independently’.”*

Geographic Distribution *“The parties collaborating in an SoS are distributed over a large geographic extent. Although the geographic extent is defined vaguely, it is stressed that the collaborating systems can only exchange information and not considerable quantities of mass or energy.”*

Evolutionary Development *“An SoS’s existence and development are evolutionary in the sense that objectives and functionality can be under constant change, as they can be added, modified, or removed with experience. Thus, an SoS never appears completely formed.”*

Emergent Behavior *“Through the collaboration between the systems in an SoS a synergism is reached in which the system behaviour fulfils a purpose that cannot be achieved by, or attributed to, any of the individual systems.”*

The acronym refines our definitions of SoSs and CSs. From our point of view, CSs can have independent development processes and cooperate to achieve common higher-level and individual lower-level goals, while having a limited ability to influence the behavior of other CSs. Other CSs may or may not be within the perception or reception ranges from the perspective of a particular CS, i.e., the set of other CSs that are “visible” to a CS in focus may develop over time. All CSs are constantly evolving and may change their behavior from time to time. In addition, existing CSs may be removed from a cooperative SoCPSs, or new CSs may be added. With respect to emergent behavior, it is trivially clear that cooperative behavior can only arise through the interaction of CSs.

Resulting from a comparison of individual systems with SoSs [90], the ABCDE acronym is another common mnemonic that summarizes five main SoS characteristics [448]:

Autonomy *“Each system is free and independent with its own purpose of operation.”*

Belonging *“Systems function collaboratively to meet a common higher purpose.”*

Connectivity *“Synergism is enabled by the highly dynamic distributed network.”*

Diversity *“The constituents are heterogeneous self-sufficient systems that are open for enhancement by evolution and adaptation.”*

Emerging *“The cumulative actions and interactions between the constituents of an SoS give rise to the behaviours that can be attributed to the SoS as a whole.”*

The ABCDE acronym goes along the same line as the OMGEE schema. However, it emphasizes the particular importance of a “*highly dynamic distributed network*” [448, p. 18:5], i.e., a shared communication infrastructure that enables interaction between interoperable CSs, as a vital basis for coordination and cooperation within the overall SoCPS. Both acronyms already provide a good overview of the high-level characteristics of SoSs. Nevertheless, the resulting properties are too coarse to derive concrete challenges in ensuring real-time behavior of such systems. We will therefore take a closer look at existing works that deal with typical characteristics of cooperative systems.

In the literature, desirable properties of SoSs—and thus indirectly of cooperative SoCPSs—have been extensively discussed [90, 345, 448]. *Integrability* [190], *interoperability* [190, 516], and *availability* [190], for example, are considered to be essential prerequisites for ensuring reliable interaction of CSs. On top of that, *consistency* [190] and *consensus* [191, 459, 497] are seen as the most important characteristics of cooperative SoCPSs. They are the basis for many higher-level properties, such as *safety* and *efficiency* [179, 625], and enable the achievement of common goals through cooperative behavior [363]. *Flexibility* [190], *evolvability* [190], and *adaptability* [479] allow adaptation to new environments and ensure long-term operation. In addition, other application-specific features required to perform certain tasks, such as optimal *coverage* [157] of a local area, e.g., for distributing transport robots or highly automated shuttle vehicles, are considered to be relevant.

For realistic systems, it may not be possible to achieve all desirable properties at the same time. Therefore, trade-offs must usually be made for or against the preservation of certain system properties. Possible reasons for limitations must also be considered during system design and safety assurance. In the following, we will discuss some of these reasons as a basis for deriving the challenges in ensuring real-time behavior.

From a computer science perspective, cooperative SoCPSs can also be considered as *distributed* systems, i.e., systems that “*consist of a number of physically distributed components that work independently [...], but [...] communicate [...] by explicit message passing*” [37, p. 373]. They usually “[*appear*] to [*their*] users as a single coherent system” [585, p. 2]. The desirable properties of such systems and the reasons for their impairment have been thoroughly investigated in the past and presented in the form of theorems.

The *Consistency, Availability, Partition Tolerance* (CAP) theorem, for example, states that “*any networked shared-data system can have at most two of three desirable properties*” [103, p. 23]. It thus characterizes the relation between *consistency* (*C*), *availability* (*A*), and *partition tolerance* (*P*). The theorem was proposed in a talk by Eric Brewer and was formalized and proved by Seth Gilbert and Nancy Lynch [244]. Ten years later, Brewer revised the original definition to address three common misconceptions [103, p. 24]:

“First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times [...] at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the [...] data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but

1 Introduction

there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.”

At almost the same time, Abadi introduced the PACELC theorem [13] as an extension of the CAP theorem. He stated that “*availability and latency are arguably the same thing*” [13, p. 38] and pursued the goal of including a case-by-case distinction between the presence and absence of network partitions in distributed systems [13, p. 41]:

“If there is a partition (P), how does the system trade off availability (A) and consistency (C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?”

The theorem addresses a similar problem as Brewer’s first adaptation. In the presence of network partitions, there is a trade-off between availability and consistency. In the absence of partitions, a choice must be made between low latency and high consistency.

With the same mindset, Lee et al. generalized and quantified the concepts of *consistency* and *availability* from the CAP theorem by replacing the *partitions* component with a “*numeric measure of the apparent latency*” [388, p. 1] L . This led to the introduction of the *Consistency, Availability, Latency* (CAL) theorem [387, 388]. The authors’ main idea was to replace “*tolerance for network partitioning*” with “*tolerance for network latency*” as network partitioning can be understood as “*limiting case of network latency*” [388, p. 2]. In addition, they distinguish systems with *centralized coordination* from systems with *decentralized coordination*. In the former case, “*inconsistency remains bounded by a chosen numerical value at the cost that unavailability becomes unbounded under network partitioning*” [388, p. 47], while in the latter case, “*unavailability remains bounded [...] at the cost that inconsistency becomes unbounded under network partitioning*” [388, p. 47].

All three theorems have in common that they emphasize that not all desirable properties of distributed systems can be fulfilled at the same time. As a result, it is essential to make well-considered trade-offs and design choices that must be coordinated across manufacturers prior to the development of CSs. Uncertainties, such as the unavailability of subsystems or the existence of inconsistencies in their situational overviews, as well as appropriate methods for reducing or dealing with such kind of uncertainties, must be explicitly considered, i.e., specified and verified, in the design of cooperative SoCPSs.

From a practitioner’s perspective, the “*key challenge in distributed systems is the fact that processes fail*” [351, Section 9.7.1]. This includes, for example, failures due to communication outages and delays, inconsistent interaction protocols as well as incorrect implementation of individual CSs that other CSs may not be aware of. “*The presence of failing processes is a defining characteristic of distributed systems, and it has implications for protocols that strive to achieve consensus.*” [351, Section 9.7.1]. In addition to the uncertainties inherent in cooperative SoCPSs, we will therefore take a closer look at failures of (parts of) individual CSs and their effects on cooperative system behavior.

In his well-known essay entitled *The Byzantine Generals Problem* [379], Lamport proposed a theoretical problem for studying arbitrary failures in distributed systems. The essence of this problem is summarized very well in [351, Section 9.7.1]:

“The Byzantine generals problem presents an analogy to illustrate the severity of failing processes. In its simplest form, there are three generals that each lead a part of the army of the Byzantine empire; one general is a commander and the other two are lieutenants. The three generals have surrounded a city and they are deciding whether to attack or retreat. If only one or two decide to attack, the city’s defenses are strong enough that the attackers will definitely lose. As such, either all three need to decide to attack or all three retreat. [...] The commander sends its command to the two lieutenants, who exchange messages with each other to confirm that everyone received the same order. If all three of the generals are loyal, then the messages [...] would be the same. That is, if the commander decides to order an attack, the lieutenants confirm with each that the commander said to attack; similarly, if the order is to retreat, then all of the messages are the same. However, one of the generals might be treacherous and might be trying to make the Byzantine army fail. If the commander is treacherous, then [one of the] message[s] might say to attack and [another one] might say to retreat; the two lieutenants then exchange conflicting messages. But if the commander is loyal and one of the lieutenants is the disloyal general, then the messages from the commander would match, but the lieutenants’ messages would conflict. From the perspective of a loyal lieutenant, both situations would appear identical.”

The most important point highlighted by the Byzantine Generals problem is that the failure or misbehavior of one process can prevent other processes—or the distributed system as a whole—from functioning properly. In this context, it may be the case that failures in one process—or in the communication between the processes involved—also prevent the consensus required to agree on cooperative maneuvers. In addition, there may be situations in which *“one process appears as a failure to some correct processes while also appearing correct to other processes”* [351, Section 9.7.2]. In general, the problem illustrates that the effects of arbitrary failures in distributed systems can be diverse, which means that identifying and correcting these failures can be a very challenging task.

Regarding the CLTA example introduced in Section 1.2.1, it is unclear from the perspective of one of the cooperation partners whether the cooperation recommendation is incorrect, whether it was falsified during transmission, or whether the cooperation partner accidentally sent an evaluation for a different maneuver. The used cooperation approach from [363] is designed in such a way that a lack of consensus leads to the termination of the negotiation process and thus to the rejection of the cooperative maneuver. In this case, safe local driving automation would be used as a fallback level, reliably taking over the driving task in full compliance with the applicable traffic rules. However, there is a serious risk that also time-critical cooperative processes in other systems may fail.

The problems discussed are exacerbated by asynchronous communication, since it may not be clear which response is to be assigned to which original message, without appropriate communication protocols or explicit references to the replied messages. In summary, the processes interacting in real-world distributed systems—including the CSs of cooperative SoCPSs—may have *“access to different, conflicting observations and cannot*

1 Introduction

determine what information is reliable” [351, Section 9.7.2]. The underlying problems must be addressed by suitable interaction protocols as well as careful system design that includes the integration of effective safety mechanisms and fallback levels.

In addition to arbitrary failures, many other undesirable effects may arise during SoS operation that have to be anticipated during system design and must be adequately addressed by suitable prevention strategies. Keating and Katina provide a comprehensive overview of systemic pathologies in SoSs [344]. Each of these pathologies brings its own challenges to the design and operation of cooperative systems. Due to the enormous complexity, we will only consider phenomena that are related to timing and real-time behavior. In addition, we will also focus on cooperative SoCPSs with a manageable number of CSs. Although the concepts, methods, and tools developed are applicable to larger ensembles of CSs, complex phenomena observed in dynamical nonlinear systems such as *unpredictable chaotic behavior* [273] are beyond the scope of this work.

1.2.3 Challenges in Ensuring Real-Time Behavior

The realization of desired properties of cooperative SoCPSs is accomplished during the design and the subsequent implementation of the individual CSs. The discipline of *System of Systems Engineering* (SoSE) [140, 505, 517, 547] deals with this and other tasks centered around a systematic, continuous, and coherent engineering of SoSs. The SoSE community promotes a list of seven so-called “*pain points*” [319], which describe the main challenges in SoS development. They can also be seen as general challenges for the design of cooperative systems. For our purposes, however, these points are too general and provide only few clues for deriving specific challenges in ensuring real-time behavior. The few relevant points have already been mentioned in the previous sections, such as the fact that SoSs typically consist of CSs with different owners and stakeholders, independent development processes, and interdependencies that may change over time.

In order to gain a better understanding of the differences between state-of-the-art engineering approaches comprised by classical *Systems Engineering* (SE), which addresses the challenges from the first problem area, and SoSE, addressing the challenges from the second problem area that are the focus of this work, it is worth considering the differences between systems and SoSs. Table 1.6 compares the typical characteristics of the two system classes. While single systems typically have a well-defined set of stakeholders, clear objectives, a defined purpose, a clear management structure, clear accountability, clear operational priorities, a single lifecycle, and clear ownership, SoSs are more complex and diverse in each of these dimensions. Thus, in the context of SoSE, multiple levels of relevant stakeholders with mixed and possibly competing interests, several, possibly contradictory, objectives and purposes as well as disparate management structures without clear accountability have to be taken into account. Additionally, the individual CSs have different operational priorities, independent development processes, individual life cycles, and different owners who make self-sufficient decisions about use of resources.

As described in Section 1.1, there are no suitable methods and tools for ensuring real-time behavior of cooperative SoCPSs. However, studies from other research disciplines, such as biology [469], psychology [124, 217, 218], physics [605], mathematics [343],

Tab. 1.6: Differences between systems and SoSs (based on [322, p. 2])

	<i>Systems</i>	<i>Systems of Systems</i>
<i>Stakeholders</i>	Clear set of stakeholders	Multiple levels of stakeholders with mixed and possibly competing interests
<i>Objectives</i>	Clear objectives and purpose	Multiple, and possibly contradictory, objectives and purposes
<i>Management</i>	Clear management structure and clear accountability	Disparate management structure without clear accountability
<i>Priorities</i>	Clear operational priorities, with escalation to resolve priorities	Multiple, and sometimes different, operational priorities without clear escalation routes
<i>Lifecycle</i>	Single lifecycle	Multiple lifecycles with elements being implemented asynchronously
<i>Ownership</i>	Clear ownership with the ability to move resources between elements	Multiple owners making independent resourcing decisions

and control engineering [139, 537], indicate that the functional behavior of cooperative systems can be strongly influenced by their temporal behavior. In fact, there are powerful interactions between the real-time behavior of individual CSs and the cooperative behavior of higher-level SoCPSs. These interactions can be viewed from two different perspectives:

1. The first perspective concerns the influence of time and timing properties on cooperative behavior. Time delays, as an instance of relevant time phenomena, come in large varieties. They can be divided into constant time delays, which are well-understood, and varying delays, which pose significant challenges, for example, in the design of control systems [139, 605]. Depending on the current traffic situation, unlimited delays in the coordination and execution of cooperative driving maneuvers between CHAVs may lead to critical and potentially life-threatening situations.
2. The second perspective focuses on the influence of cooperation on the timing behavior of cooperative systems. Cooperation between actors is inherently based on communication. To put it more precisely, negotiating cooperative maneuvers usually requires multiple communication and coordination steps, which can only be parallelized to a limited extent. Therefore, temporal properties of communication and cooperation processes between individual CSs have a significant influence on the timing behavior of other CSs as well as on higher-level SoCPSs.

Consequently, the first challenge in ensuring real-time behavior of cooperative SoCPSs is to enable process support for developing such systems. Other challenges arise from typical characteristics of SoCPSs that cannot be taken into account by using existing specification and verification means. Here, we adopt an application-oriented view and consider the most important properties that need to be addressed when developing cooperative driving functions. In total, we identified four key challenges in enabling a continuous and consistent handling of real-time properties in the development of cooperative systems:

1 Introduction

Systems of Cyber-Physical Systems The first challenge is to provide *process support for the design of cooperative SoCPSs*. This includes the explicit consideration of safety-critical, cross-system coordination processes in early system design. Due to the lack of suitable approaches to ensure real-time behavior of cooperative systems, a method for cross-manufacturer coordination and specification of non-functional properties of cooperative systems is needed to enable a consistent formulation of top-level timing requirements and the derivation of subsystem requirements.

Dynamic System Architectures The second challenge is related to the explicit consideration of *dynamic system architectures*. It involves modeling, specification, and verification of the limited availability of individual CSs, behavioral changes at run-time, and dynamic reconfiguration. During the journey of a CHAV along its route, for example, other vehicles may enter and leave the individual sensing and communication areas. In addition, system functions may be degraded, e.g., as a result of the occurrence of faults. Systems may also be updated with new functionality and must operate safely in different configurations of components and interfaces.

Unreliable Communication The third challenge focuses on the handling of potentially *unreliable communication*. We recall that proper functioning of V2X communication is an essential prerequisite for exchanging information between vehicles, e.g., for negotiating cooperative driving maneuvers. However, due to technical limitations and transmission errors, temporary and permanent communication failures may occur. In order to ensure safe and reliable interaction in cooperative systems, communication failures and corresponding error correction techniques such as retransmissions and timeouts must be made explicit in early design phases.

Imprecise Local Clocks The fourth challenge concerns the mapping of specifications to *imprecise local clocks*. While the timing behavior of cooperative SoCPSs is typically specified based on an implicit, infinitely precise “global clock” at design time, no such clock exists in the real world. Instead, CSs have their own time domains with individual offsets, resolutions, and inaccuracies. The challenge becomes particularly relevant when time-stamped messages are exchanged between cooperating CHAVs or when they attempt, for example, to agree on a specific start time for a planned cooperative maneuver. In addition, cross-system real-time requirements, such as end-to-end latencies, which have been decomposed into system-level requirements, must be verified, e.g., by applying distributed run-time monitoring.

1.3 Research Questions

As already stated, the focus of thesis is the development of concepts, methods, and tools to ensure real-time behavior of (systems of) cooperating cyber-physical systems. Based on the identified challenges in establishing a SoCPS-wide timing coherence, the following research questions have been derived and will be addressed as part of this work:

- RQ-1 How can engineers be supported in formulating consistent real-time requirements for (systems of) cooperating cyber-physical systems?
 - RQ-1.1 Which formalisms are particularly suited to specify the behavior of cooperative systems?
 - RQ-1.2 How must a design process be structured to ensure consistent requirement formulation?
 - RQ-1.3 How can the results be translated into system-level specifications?
- RQ-2 How can existing specification means be extended to cover relevant aspects of cooperative systems?
 - RQ-2.1 How can dynamic system architectures be handled?
 - RQ-2.2 How can unreliable communication be considered?
 - RQ-2.3 How can specifications be mapped to imprecise local clocks?
- RQ-3 How must verification methods for single systems be adapted to support virtual integration testing of real-time requirements for cooperative systems?
- RQ-4 How can the resulting verification methods be implemented and connected to industry-relevant modeling tools in a modular and extensible way?

1.4 Outline

The thesis is divided into eight chapters, which are organized as follows:

Methods and Tools for Ensuring Real-Time Behavior The second chapter presents the state of the art, focusing on methods and tools for ensuring real-time behavior. Following a detailed analysis of existing frameworks for timing specification of single systems, their individual characteristics are compared using a set of previously defined evaluation criteria. The comparison results in the selection of a suitable candidate for the planned extensions to support the design of cooperative SoCPSs. Afterwards, a gap analysis between the features of the selected methodology and the requirements from research questions RQ-1 to RQ-4 reveals four methodological and two technical gaps, which are addressed by the contribution of this work.

Consistent Specification of Top-Level Timing Requirements Chapter 3 focuses on providing process support for the design of cooperative SoCPSs. It presents a correct-by-design approach for the consistent formulation and refinement of real-time requirements on SoCPS level, which aims at supporting engineers in developing such kind of systems. In a first step, we review established modeling and specification languages that allow the description of cooperative behavior. Based on the results, we select a suitable formalism and adapt it to fit the specific needs of specifying real-time behavior. The resulting formalism is then used as the basis for extending the existing methodology. Finally, all process steps of the proposed design approach are presented and demonstrated using the CLTA application example.

Handling of Dynamic Architectures The fourth chapter is dedicated to the handling of dynamic architectures. First, existing approaches for modeling and specifying relevant system properties are reviewed. Subsequently, consistent extensions to the specification language provided as part of the existing methodology are made, which support the handling operating modes, mode transitions, and mode-dependent timing behavior. The results lead to the definition of a new version of the specification language, which is referred to as MTSL 1.5+MC. In addition, the verification of correct refinement of the enhanced specifications is addressed. The practical applicability of the results is demonstrated using realistic application examples. The chapter is completed by discussing approaches to model dynamic reconfiguration.

Modeling of Unreliable Communication Chapter 5 addresses the modeling of unreliable communication. The chapter begins with an overview of existing approaches for modeling and analyzing unreliable communication. Based on a systematic literature review that focuses on performance evaluations of domain-specific communication standards, a set of relevant time phenomena is identified, which are then addressed through the development of a generic communication model. The resulting communication model incorporates MTSL 1.5+MC specifications and facilitates early consideration of the temporal properties of unreliable communication.

Mapping of Specifications to Imprecise Local Clocks The sixth chapter deals with the mapping of specifications to imprecise local clocks. First, existing approaches for handling local clocks in technical systems are reviewed. Afterwards, clock relations are defined in order to formally reason about the relation between an implicit, infinitely precise “global clock” used at design time and local clocks with typical properties. For this purpose, both faithful and approximate clock relations are investigated. The results are prepared for further use by adding corresponding patterns and pattern extensions to the MTSL 1.5+MC specification language.

Tool Support Chapter 7 presents the developed tool support, which consists of a modular and extensible toolchain for performing real-time analyses, such as virtual integration tests. The chapter begins with an overview of the functional and non-functional requirements that were collected prior to the development. Following a conceptual description of the toolchain, the individual components are considered in more detail. In a final step, the developed tool support is evaluated in terms of specification language support and performance of the toolchain components.

Conclusion The final chapter summarizes the results. To this end, the outcomes of all previous chapters are reviewed from the perspective of research questions RQ-1 to RQ-4. This includes an evaluation of whether the original goals were achieved. The thesis concludes with a discussion of limitations and possible extensions.

The chapters are supplemented by two appendices. Appendix A provides an overview of existing versions of the specification language and compares the language support of the individual versions. In addition, Appendix B presents consistent modeling guidelines that facilitate the application of the developed methods in industrial design processes.

2 Methods and Tools for Ensuring Real-Time Behavior

Over the past decades, numerous methods and tools have been developed to ensure the correct and timely functioning of safety-critical systems. They are typically based on theoretical formalisms or general-purpose modeling languages that have been augmented with appropriate description means, operations, and tool support to address the particular challenges of the application domain. This chapter provides an overview of the current state of the art in ensuring real-time behavior and gives some indications of the state of the practice in this area, especially from the perspective of the automotive domain.

Today, engineers have a complete toolbox of different methods and tools at their disposal that support real-time engineering tasks in almost all phases of the development process. The existing solutions address the first problem area introduced in Section 1.1 and support modeling, specification, analysis, implementation, verification, and validation of real-time requirements and properties at different levels of abstraction and from different perspectives, focusing on the systematic and standards-compliant design of single systems.

All of today's methods and tools for ensuring real-time behavior are related to or can be integrated with *Model-Based Systems Engineering* (MBSE). According to the generally accepted definition of the *International Council on Systems Engineering* (INCSE), MBSE is the “*formalized application of modeling to support system requirements, design, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases*” [320, p. 15]. It has emerged from the evolution of model-based development processes for software products to support SE of hardware/software systems. MBSE is typically based on three main principles:

Frontloading The first principle is to move key process steps into early, conceptual design phases. The goal of this shift is to analyze relevant system properties as early as possible in order to identify potential problems earlier and avoid the late detection of errors, which is generally costly and time-consuming.

Consistent Interpretability The second principle is to establish consistent interpretability of model elements and relationships, e.g., through the use of meta-models and the definition of a common semantics. This interpretability enables the creation of unambiguous and executable system descriptions, which serve as a basis for introducing tool automation to manage system complexity.

Continuous Traceability The third principle is to ensure a continuous traceability along the process. This involves defining explicit relationships between design and de-

velopment artifacts, thus enabling the use of process automation (e.g., automated transformations and impact analyses) to reduce design and debugging efforts.

The consistent use of MBSE is considered a prerequisite for the development of safety-critical systems in many application domains. In this context, *Model-Based Safety Assessment* (MBSA) [562] is the logical extension of the MBSE concept. It builds upon previously created system models and adds explicit modeling of safety-related design artifacts such as hazards, safety requirements, verification and validation activities, test results, and SACs [554, 555]. The use of traceability to integrate these artifacts into model-based design approaches [16, 553] allows, for example, requirements consistency verification, fault propagation analyses, and risk quantification [58, 138, 436]. MBSE and MBSA are complemented by *Model-Driven Development* (MDD) [79, 472] techniques for implementation and *Model-Based Testing* (MBT) [526] approaches for testing purposes, which close the last remaining gaps in complete process coverage.

Methods and tools for ensuring real-time behavior can, to some extent, be located at the interface between MBSE and MBSA. At system level, real-time requirements are derived from hazard analyses in terms of *fault-tolerant time intervals* (FTTIs) [233] and are gradually decomposed during the design process. By formalizing these requirements by means of specification languages, (semi-)formal specifications are created that can be automatically processed and analyzed [92, 93]. System models with annotated timing specifications are used as input for various types of analyses, e.g., to verify the specifications' consistency [164] or the compliance of functional models and implementations with these specifications [58, 206]. The analysis results can be used, for example, to identify inconsistencies in specifications, to correct or improve the temporal behavior of functional models, or to optimize the mapping of software components to available processing units [368]. At the same time, they serve as an input for providing evidence about the compliance with the higher-level safety goals and the safety requirements of the overall system [58], and thus contribute to the SAC, which is the heart of MBSA.

2.1 Existing Approaches and Technologies

As described above, the state of the art includes various methods and tools that support ensuring real-time behavior of individual systems. This section presents existing approaches and technologies that can be used to specify, model, and verify real-time properties. They form the basis for addressing the challenges in the design of cooperative SoCPSs presented in Section 1.2.3 and for answering the research questions from Section 1.3.

In the following, we will first look at existing frameworks for timing specification in Section 2.1.1. Then, in Section 2.1.2, we review established modeling and specification languages for describing systems and their timing behavior. Section 2.1.3 provides an overview of the contract-based design paradigm, which is used in this thesis as a framework for specifying real-time behavior of cooperative SoCPSs. Solutions to the closely related challenge of verifying the consistency of previously formulated specifications are investigated in Section 2.1.4. Section 2.1.5 deals with real-time and scheduling analyses

necessary for the continuation of activities from the early design phases on technical level. The last section gives a brief overview of relevant methods for run-time monitoring of temporal properties, whose integration is essential for a complete coverage of the lifecycle of cooperative SoCPSs, i.e., for monitoring the compliance with previously formulated real-time requirements during the operation of the overall system.

2.1.1 Frameworks for Timing Specification

There are a number of existing frameworks for timing specifications that are particularly suited to the design of automotive and aerospace systems, but can also be applied to systems from other application domains. Although some of these frameworks are already used in practice, they lack support for relevant time phenomena of cooperative SoCPSs.

This section provides an overview of the existing frameworks. All frameworks presented have in common that they support the integrated modeling of systems as well as the specification of timing properties. In addition to the modeling of system architectures and the specification of real-time requirements, they ensure the integration into higher-level design methodologies. All frameworks offer tool support for modeling, specification, and analysis of (parts of) systems and their behavior. Most of the frameworks can also be employed as an exchange format between different design and analysis steps. The candidates are typically based on well-established, general-purpose modeling languages such as the *Unified Modeling Language* (UML) [455] or the *Systems Modeling Language* (SysML) [454] and complement these with appropriate description means, operations, and tool support to cover relevant aspects of the systems under consideration.

The selected approaches are presented in order of their position in the design process, i.e., approaches that primarily address early design phases are described before those that relate more to later development phases. Subsequently, we will take a closer look at the suitability of the presented frameworks for ensuring real-time behavior of cooperative SoCPSs by comparing their ability to address the identified challenges in Section 2.2.

EAST-ADL with Timing Augmented Description Language

As one of the most expressive candidates, the *Timing Augmented Description Language* (TADL)¹ [89, 336, 474] enables a consistent formulation of real-time requirements. It was developed in the TIMMO and TIMMO-2-USE projects and focuses on early design phases. The specification language follows a pattern-based approach and supports considerations at different levels of abstractions and from different perspectives. It comprises a variety of specifications, including patterns for defining repetitive, sporadic, irregular, and burst event occurrences, weak and strong delay constraints, synchronization constraints, event ordering constraints, execution time constraints, reaction and age constraints as well as simple mode dependencies and basic support for handling multiple time bases. Additionally, the language incorporates probabilistic extensions for most of the specification patterns, thereby enabling the use of probability distributions (e.g., uniform, normal, Gumbel, Fréchet, and Weibull) to constrain the occurrence of events within time intervals.

¹In the following, the acronym TADL is used to reference the current version TADL2 as defined in [89].

The birth of the first framework for timing specification was the integration of TADL timing constraints in the *Electronics Architecture and Software Technology – Architecture Description Language* (EAST-ADL) [88, 161, 197]. EAST-ADL is a standard architecture description language for automotive applications, aligned with the most common AUTOSAR [5, 6] concepts. Like TADL, it has a strong focus on early design phases. For later phases and development tasks on lower abstraction levels, AUTOSAR can easily be employed due to the established alignment of the concepts of both languages. EAST-ADL enables applying formal analyses, e.g., by converting EAST-ADL models with TADL specifications into *Timed Automata* (TA) [23, 68] networks [247], which allows the subsequent analysis with UPPAAL [69]. In addition, tool support for developing EAST-ADL models exists, e.g., in Enterprise Architect [549] and Papyrus [202] as well as in some domain-specific tools such as MetaEdit+ [419] and SystemWeaver [566].

MULTIC and MULTIC-Tooling

The second framework of particular relevance to this work is the MULTIC methodology [92, 93, 164, 205]. It has been developed in two projects entitled *Multi-Layer Time Coherency in ADAS and Automated Driving* (MULTIC) and MULTIC-Tooling, both initiated and commissioned by the *German Association of the Automotive Industry* (VDA), more precisely by the working group on *Software and Electronics* of the *Research Association for Automotive Technology* (FAT) [240]. The MULTIC methodology is a contract-based design method that relies on *Assume/Guarantee* (A/G) contracts [70] integrated into a compositional semantic framework. The framework consists of a well-defined component model as well as the possibility to integrate behavioral models in terms of *models of computation* (MoCs) with different notions of time (e.g., untimed, synchronous, discrete, continuous) that interact via specialized *interaction components*. The methodology focuses on early design phases, but can also be employed to support technical system design. In this context, for example, it supports the transition to the technical perspective by explicitly mapping contract-based specifications to *AUTOSAR Classic Platform* (AUTOSAR-CP) elements such as software components, runnables, and specialized events [92].

The MULTIC methodology includes a pattern-based specification language called *MULTIC Timing Specification Language* (MTSL) that is particularly suited to describe the timing behavior of automotive systems. It is based on enhancements of a subset of the TADL specification patterns and was formerly known as *Requirement Specification Language* (RSL) [496]. The MTSL comprises specification patterns such as **Repetition**, **Reaction**, and **Age**, each with non-causal and causal versions, including additional parameters and options for refinement. In addition, it provides specification means for defining local clocks, including support for typical properties such as resolution, drift, and skew, as well as mechanisms for binding individual patterns to these local clocks. Over the years, various adaptations and revisions of the original MTSL 1.0 have been developed [164, 205], culminating in the latest intermediate version referred to as MTSL 1.5 [559].

According to the MULTIC methodology, the development of systems is carried out along well-specified requirements, formulated in A/G contracts that consist of MTSL specification patterns. By expressing specifications in terms of contracts with well-defined

operations and corresponding proof obligations, the correctness of each refinement step in the design process can be proven through *virtual integration tests* (VITs). As a result, the implementation of the individual subsystems can easily be divided between different teams. After checking the compliance of the realized timing behavior with the refined specifications, a statement on the original top-level specification can be made without further investigations. Tool support for modeling, model validation including syntax checking, simulation-based VITs, and result visualization exists in terms of an Eclipse-based tool prototype [93, 164]. In addition, a set of *design rules* has been defined to facilitate the use by industrial requirements engineers [93, Chapter 7].²

UML/MARTE with Clock Constraint Specification Language

The *Modeling and Analysis of Real-Time Embedded Systems* (MARTE) [457] framework is defined as a cross-domain UML profile that “*adds capabilities [...] for model-driven development of Real-/Time Embedded Systems [and] provides support for specification, design, and verification/validation*” [457, p. 1] of such systems. It allows the modeling of both hardware and software elements in a standardized way, thus enabling “*interoperability between development tools*” [457, p. 1]. The MARTE profile is composed of a number of submodels that focus on modeling the system architecture, system behavior, time (incl. multiple time bases), resources, and allocations of software to hardware elements.

From the perspective of system design, MARTE provides a generic component model, high-level application models as well as detailed resource models to describe the characteristics of embedded real-time systems. In accordance with the objectives of MBSE, the main focus of MARTE is on applying comprehensive analyses in early design phases to “*make quantitative predictions regarding real-time and embedded features*” [457, p. 1]. This goal is instantiated by defining dedicated analysis models that support performance and schedulability analysis. It also contains a generic analysis framework that can be used to define custom analysis types. All modeling concepts are compatible with other UML-compliant languages and can be combined to cover different aspects [502].

The comprehensive description means of MARTE have been further enhanced by the definition of *Clock Constraint Specification Language* (CCSL) [34, 35, 407, 408], which allows detailed specification of local clocks, their relations, and clock constraints. The language was introduced in an appendix of the original MARTE specification [457], so that the two languages are very well harmonized and often used in combination. CCSL allows the specification and verification of multi-clock systems [622] and can be used, for example, to automatically synthesize requirement models with safe timing behavior [299]. MARTE and CCSL are supported by general-purpose modeling tools such as Enterprise Architect [549], MagicDraw [168], and Papyrus [202] as well as by specialized engineering tools for embedded applications such as Rhapsody Architect for Software [307].

²An earlier version of this paragraph has been published in [206]. The corresponding passage in the original publication was written by me and was finalized with editorial changes by my co-authors.

MechatronicUML

MechatronicUML [232] is an academic tool suite that supports MBSE and MDD. It is based on two closely interlinked model-driven requirements engineering and design methods [196, 293] that enable formal analysis of requirements and design models:

1. The *MechatronicUML Requirements Language* allows the scenario-based specification of requirements for interacting systems by means of *Modal Sequence Diagrams* (MSDs) [271]. It also supports the automatic transformation of MSDs into component-based specifications. The description of structural elements of the systems involved is based on a subset of UML that includes *UML Collaborations*.
2. The *MechatronicUML Design Language for Platform-Independent Modeling* adds support for translating MSD specifications into component models. It helps engineers to integrate real-time coordination protocols into the structural system models that contribute to the realization of scenario-based requirements modeled as MSDs.

The tool suite covers the first stages of top-down design approaches. It can be used from scenario-based specification through functional system design to the refinement of the underlying models and ensures a seamless transition to technical system design.

MechatronicUML does not define a specification language for temporal properties. Instead, it uses basic description means such as clocks and clock resets provided by MSDs to track the passage of time, and integrates other more expressive formalisms such as a revised version of *Real-Time Statecharts* (RTSCs) [243] to specify the temporal behavior of coordination protocols. The timing specifications of RTSCs are based on a finite set of clocks, as in TA. Based on these clocks, constraints are defined that can be used as invariants of states or as guards on transitions. It is also possible to add relative and absolute deadlines to transitions, defining a time interval in which the firing must be completed. Moreover, explicit synchronization between parallel RTSCs is supported.

Over time, various extensions to the framework have been developed. Holtmann et al., for example, combined MechatronicUML with MARTE and CCSL to specify non-functional requirements such as timing and resource constraints, model generic resources, and explicate allocations of software elements to execution platform components. The authors applied early timing analysis based on scenario requirements and platform models by simulating CCSL models with TimeSquare [291, 292]. To this end, CCSL models were automatically derived from platform-specific MSD specifications and employed in timing simulations “to identify platform-induced real-time requirement violations [...] that could otherwise be revealed only in late engineering phases” [292, p. 37]. Other related work has pursued model checking of design models using UPPAAL, e.g., to verify real-time coordination protocols [196]. MechatronicUML also supports the generation of Matlab/Simulink [569, 570] and Modelica [427] models for simulation-based analysis.

Eclipse CHESS

Eclipse CHESS [145, 176, 201] provides a model-driven, component-based methodology and tool support for the development of embedded real-time systems. The academic

tool suite implements the CHESSE UML/SysML profile [318], which is a specialization of the MARTE profile [457]. CHESSE was originally developed to support the design of aerospace systems. It focuses on hardware/software design, but also provides basic support for earlier design phases such as functional system design. In addition to a modeling environment with validation support, it offers model-to-model and model-to-text transformations that enable “*model-based schedulability, dependability and contract-based design analysis, [including] code generation toward multiple language targets*” [201].

The CHESSE UML/SysML profile allows, for example, explicit modeling of failure modes, including *Safety Integrity Level* (SIL) classification and determination of local fault propagation. This enables the application of failure propagation and state-based quantitative analysis [201]. CHESSE also defines a profile for modeling contracts, supporting A/G contracts, strong and weak contract types, refinement relations, and the mapping of contract-based specifications to system components. Nevertheless, the framework does not define a particular specification language. In practice, such a language has to be chosen depending on the analysis purpose and the available tools, e.g., Othello contracts [152], which are supported by the contract-based verification tool OCRA [147]. Othello, as the specification language most commonly used with CHESSE, does not provide support for high-level specifications close to natural language, i.e., real-time and safety properties must be encoded by using the description means of the underlying temporal logic [148]. Known uses of CHESSE include contract refinement proofs with OCRA [147], functional verification using nuXmv [129], and timing analysis based on MAST [269].

Rubus Component Model

The *Rubus Component Model* (RCM) [115, 117, 267, 435] is a “*domain-specific language for developing predictable and analyzable control functions in resource-constrained, real-time embedded systems*” [117, p. 1086]. It has been developed by Arcticus Systems in close cooperation with Mälardalen University. The language is accompanied by an industrial tool suite called Rubus Tool Suite [39], also known as Rubus-ICE, which provides support for modeling, validation, analysis, transformation, and code generation of RCM models. According to its developers, RCM “*covers all abstraction levels of automotive system development*” [117, p. 1086], but focuses on hardware/software design and implementation.

The RCM language consists of four packages that can be used to describe *hardware, software, allocations, and timing* properties. The *hardware* model allows the modeling of hardware platforms in terms of “*processing units and network busses [sic!] abstracting from low-level details such as memory hierarchies or the operating system*” [117, p. 1086]. The *software* model “*is used for describing software systems in terms of software functions and connections among them*” [117, p. 1087]. During the creation of RCM software models, the intended functionality must be decomposed into so-called *Software Circuits* (SCs), which represent the smallest functional units of RCM models. The interactions between these SCs can be modeled as data flows or control flows. An important feature of SCs is the strict separation of the execution infrastructure from the functional code, as this facilitates reusability and debugging of system descriptions. While the *allocation* model supports the definition of mappings between the elements of software and hardware

models, the *timing* model is dedicated to the specification of real-time requirements and timing properties. It implements interpretations of most of the TADL constraints mentioned above [435]. Thus, RCM supports specifications for defining different types of event occurrences, weak and strong delay constraints, synchronization constraints, event ordering constraints, execution time constraints as well as reaction and age constraints. In addition, RCM enables the use of “*all [...] timing constraints from the AUTOSAR standard, and allows designers to specify real-time properties such as worst-case, best-case, and average-case execution times, as well as stack usage*” [117, p. 1087].

The *Rubus analysis framework* provides extensive tool support for various types of analyses, including response-time and end-to-end data-propagation delay analyses [434]. The combined use of EAST-ADL, TADL, and RCM helps to establish a system-level end-to-end timing model that supports design activities in most process phases [435] and comes very close to the “system-wide timing coherence” (for individual systems) described in Section 1.3. Moreover, existing research has defined a mapping between AMALTHEA and RCM models that allows the translation of AMALTHEA architectures modeled in Eclipse APP4MC into an RCM architecture to perform high-precision timing analyses, and to propagate the analysis results back into the original AMALTHEA model [116].

AUTOSAR with Timing Extensions

In the automotive domain, AUTOSAR [5, 6] is a standardized software framework that aims to be the “*global established standard for software and methodology enabling open E/E system architectures for future intelligent mobility supporting high levels of dependability*” [4, p. 8]. AUTOSAR focuses on later design phases, especially on the implementation and deployment of software applications. It only provides limited support for early system design and demands a fast transition to software-oriented design activities.

The first step in modeling the functionality of a system is to define a functional architecture, which is used differently depending on the AUTOSAR variant. AUTOSAR is divided into two coexisting methodologies called AUTOSAR-CP for classical *Electronic Control Unit* (ECU)-based systems and *AUTOSAR Adaptive Platform* (AUTOSAR-AP) for high-performance computing, e.g., for ADAS and AD applications:

Classic Platform The AUTOSAR-CP methodology starts with a mapping of the functions from the functional architecture to *Software Components* (SWCs). The communication between these SWCs is realized by means of so-called *Virtual Functional Buses* (VFBs), which abstract from real bus systems. SWCs are implemented as a set of runnables that are deployed by mapping them to tasks, which can then be allocated to processing units or cores of target platforms. The AUTOSAR *Run-Time Environment* (RTE) is responsible for realizing the communication according to the VFBs specifications. Task execution is typically scheduled by using OSEK schedulers. In addition, a set of *Basic Software* (BSW) provides hardware abstractions and other shared services to support the execution of SWCs.

Adaptive Platform The AUTOSAR-AP methodology maps functions from the functional architecture to so-called *Adaptive Applications* (AAs), which can later be imple-

2.1 Existing Approaches and Technologies

mented as POSIX applications. The deployment of these applications is done in terms of *software packages* and can be modified at run-time, leading to *adaptive* deployment schemes. Each software package must report its state to a higher-level execution manager that orchestrates the execution on target platforms. Software tasks are scheduled by a dedicated *Execution Manager*, which also provides an optional *Deterministic Client* for deterministic execution.

AUTOSAR provides comprehensive specification means for timing properties in terms of the *AUTOSAR Timing Extensions* (AUTOSAR-TE) [10, 11] as well as recommended methods and practices for timing analysis [12], which are widely used in industrial development processes. AUTOSAR-TE provides a “*timing model as specification basis for a contract[-]based development process, in which the development is carried out by different organizations in different locations and time frames*” [11, p. 14] and focuses on event-based specifications. It defines separate views for specifying the timing behavior of the overall system and of relevant AUTOSAR-CP elements such as VFBs, SWCs, BSW, and ECUs. It also supports the specification of timing properties of AUTOSAR-AP elements by providing timing views for VFBs, machines, executables, and services. In addition, event chains can be defined in both standards to specify end-to-end timing behavior across functional units. Timing constraints can thus be formulated on the basis of single events or event chains and for individual executable entities. Similar to TADL specifications, AUTOSAR-TE supports the use of *event triggering constraints* to define periodic, sporadic, pattern-based, burst, and arbitrary event occurrences, *latency timing constraints* and *age constraints* to specify relations between two or more events, *synchronization timing constraints*, and *offset timing constraints*. It also supports explicit modeling of traceability relations between constraints to facilitate automated verification.

As AUTOSAR is an industry standard, there is powerful tool support for modeling software architectures and behavior, code generation, configuration, and deployment. The modeling of AUTOSAR architectures, for example, is supported by SystemDesk [629] and DaVinci Developer [587, 588], among other industry tools. TargetLink [630], for example, can be used to generate production code from AUTOSAR models. In the academic world, Beringer and Wehrheim proposed an approach to verify AUTOSAR software models by transforming AUTOSAR architectures and timing constraints into TA [73]. Cimatti et al. applied software model checking to verify contract-annotated AUTOSAR models with Kratos [151] and *Othello Contracts Refinement Analysis* (OCRA) [147] in [150].

Eclipse APP4MC

The Eclipse *Application Platform Project for Multi-Core* (APP4MC) [200, 296, 607] tool suite is the result of years of collaboration in an open source initiative with strong industrial participation. It provides an interoperable and extensible open source development platform for embedded multi-core and many-core systems based on the AMALTHEA meta-model. Eclipse APP4MC has a strong focus on late development phases, particularly on the modeling and analysis of complex hardware/software systems.

The AMALTHEA meta-model consists of two hierarchical submodels that cover all relevant process steps of typical design flows for embedded real-time systems, including software modeling, partitioning, platform modeling, optimization, simulation, and execution. It is divided into the *AMALTHEA System Model* and the *AMALTHEA Trace Model*. The former includes hardware and software models, timing and mapping constraints as well as software mappings, while the latter can be used to capture the results of simulations and executions. The system model also contains a simple component model that can be used to represent functional architectures. In addition, several types of constraints are defined that allow the specification of *requirements*, *affinity constraints* that restrict the mapping of executables to processing units, *ordering constraints*, *data age constraints*, and *timing constraints* that restrict “the time span between events or the duration of event chains” [199, Chapter 3.6]. The requirements are further refined to support the definition of, for example, *time metrics* such as execution times, response times, waiting times, and memory access times, and *inter-process metrics* such as activate-to-active, end-to-end, and end-to-start latencies, which can be used as quality metrics for evaluating design alternatives. In summary, APP4MC provides a very comprehensive set of requirements and constraints that are well-suited for describing hardware/software systems at low levels of abstraction, but may be too fine-grained for use in higher-level specifications.

The framework is integrated into ISO 26262-compliant design processes [18, 579] and can be used as a reference format to facilitate data exchange in cross-organizational development projects [531]. From an analysis perspective, APP4MC supports, for example, execution time analyses and deployment optimization [53, 368]. Additional support is provided for performing VITs and satisfaction checks [58] as well as for timing analysis using industrial tools such as chronSUITE [310] or TA Tool Suite [589].

Architecture Analysis & Design Language

The *Architecture Analysis & Design Language* (AADL) [221, 223] has originally been developed for the aerospace industry and was standardized by the SAE [510]. It allows the description of both software and hardware elements as well as the annotation of timing requirements. In general, AADL has a similar expressiveness compared to EAST-ADL and MARTE [305]. Systems are modeled by composing a set of components, each with a clear separation between *block interfaces* and *implementations*. Based on these models, AADL supports the specification of low-level timing properties, i.e., execution times, memory consumption, and fault descriptions, that can be attached to model elements such as components, connectors, features, and instances. In addition, it enables the specification of other characteristics and constraints such as clock frequencies of processors or bandwidths of buses. However, AADL does not provide support for formulating higher-level timing specifications. User-defined properties could be used to extend the existing specification means by adding custom requirements to AADL models.

The *Open Source AADL Tool Environment* (OSATE) [222] provides tool support for the design and analysis of AADL models. It includes a large set of available analysis tools with a strong focus on response-time and scheduling analysis, such as verification techniques provided by Cheddar [540] and MAST [269]. Thus, existing research focuses

on early timing analysis and simulation model synthesis [400] as well as real-time model checking of AADL models [77]. In addition, importers for Matlab/Simulink [569, 570], SCADE [3], Enterprise Architect [549], and Papyrus [202] exists, which can be used to read in external models. All AADL model elements are mapped to SystemC [460] language constructs, allowing efficient simulation of the expected system behavior.

2.1.2 Modeling and Specification Languages

Due to the existence of various modeling and programming languages used in different application domains, there are numerous ways to handle timing properties in the development of cyber-physical systems. Over the years, two main approaches have emerged for specifying real-time properties of technical systems. On the one hand, real-time extensions such as MARTE [457] of general-purpose modeling languages such as UML [455] and SysML [454] allow pragmatic and efficient, but in some cases also ambiguous, modeling and specification of timing behavior. This class of description means also includes, for example, the frameworks introduced in the previous section. On the other hand, a large number of theoretical formalisms, which typically have well-defined semantics and provable properties, have been defined or extended to support real-time properties.

Alur and Henzinger provide a comprehensive overview on logic- and automata-based real-time formalisms [27]. They classify the existing approaches according to four characteristic properties of their *trace*, *interleaving*, and *fictitious-clock* semantics that results from abstractions of the concrete semantics defined in terms of timed state sequences:

1. Temporal dimensionality (*observation sequences* vs. *state sequences*)
2. Basic time entity (*time points* vs. *time intervals*)
3. Monotonicity (*weakly monotonic time* vs. *strictly monotonic time*)
4. Discretization (*integer time* vs. *real-numbered time*)

The properties can be understood as *dimensions* of the semantics of temporal logics and group the existing formalisms into a total of 16 categories, which result from the combination of one possible value for each characteristic. Each combination represents a possible semantics that can be used to describe the behavior of real-time systems. In the following, we will take a closer look at *temporal logics*, *graph- and automata-based formalisms*, *pattern-based specification languages*, *graphical formalisms* as well as *practical approaches* that are particularly suited for specifying real-time properties. In doing so, we will use the dimensions defined by Alur and Henzinger to relate them to each other.

Temporal Logics

Temporal logics were first proposed by Amir Pnueli in [483]. These formalisms typically rely on temporal interpretations of the operators of model logics. Consequently, the basic operator $\Box$ is interpreted as *always*, and its counterpart $\Diamond$ as *eventually*. Temporal logics allow to express *qualitative* temporal requirements by default. The existing logics can be divided into two main classes based on their temporal dimensionality:

1. *Linear-time logics* employ an *observation-oriented* semantics, i.e., “time is treated as if each moment in time has a unique possible future” [438, p. 20]. They are used to describe behavior based on a single timeline. Well-known examples of linear-time logics are *Linear Temporal Logic* (LTL) [483], *Metric Temporal Logic* (MTL) [367], *Metric Interval Temporal Logic* (MITL) [25], and *Signal Temporal Logic* (STL) [404].
2. *Branching-time logics* have a *state-oriented* semantics, i.e., “each moment in time may split into various possible futures” [438, p. 20]. Logics of this class are used to specify behavior based on multiple timelines. *Unified System of Branching Time* (UB) [67], *Computation Tree Logic* (CTL) [153], and the generalization of LTL and CTL called CTL* [209] are typical examples of such logics.

Classical temporal operators such as the one used in LTL and CTL do not support metric time and, hence, are insufficient for specifying *quantitative* temporal requirements, i.e., for formalizing real-time requirements of realistic systems. For this reason, real-time extensions for classical temporal logics have been developed. The extensions include, among others, the replacement of temporal operators by time-bounded versions [25, 367, 465], the inclusion of references to times of temporal contexts [26] as well as the use of explicit clock variables and constraints [275]. As a result, extended versions of both linear-time (e.g., MTL, MITL, *Timed Propositional Temporal Logic* (TPTL) [26], *Explicit Clock Temporal Logic* (XCTL) [275]) and branching-time (e.g., *Real-Time Computation Tree Logic* (RTCTL) [210], *Timed Computation Tree Logic* (TCTL) [22]) logics were defined. Moreover, probabilistic extensions such as the *Probabilistic Computation Tree Logic* (PCTL) [268] allow the explicit formulation of “soft deadlines” [268, p. 530] and the reasoning about reliability properties. The specification of hybrid real-time systems is enabled by means of specialized logics such as *Hybrid Linear Temporal Logic with Regular Expressions* (HRELTL) [148], for which automated analyses implemented in OCRA exist.

In addition, further formalisms such as the interval logic *Duration Calculus* (DC) [135] that reasons about duration integrals of states within arbitrary intervals can be employed to verify correctness of safety-critical systems. Meyer et al., for example, translated DC specifications into an automata-based semantics and applied model checking to verify emergency procedures of the ETCS [421]. In order to facilitate the selection of suitable temporal logics for a given use case, Bellini et al. introduced a list of nine “*criteria for assessing the capabilities of temporal logics for the specification, validation, and verification of real-time systems*” [65, p.12]. The authors presented a total of 16 temporal logics and compared their characteristic properties on the basis of the criteria which, in addition to Alur and Henzinger’s dimensions, also include logic decidability, soundness and completeness of the deductive system as well as the executability of specifications.

Graph- and Automata-Based Formalisms

In addition to temporal logics, graph and automata-based formalisms can be used for specifying real-time behavior. One of the most prominent representatives of this class of specification languages are TA. The formalism was introduced by Alur and Dill in [23] and has become a general-purpose tool for many real-time applications. TA can be employed

for generating or accepting infinite sequences of states and may be restricted by specifying timing constraints. The formalism represents an abstract model of real-time systems that incorporates a generalization of finite-state machines over infinite strings. Each TA consists of a set of *propositions*, a finite set of *locations*, a subset of *initial locations*, a finite set of real-valued *clocks*, a set of *edges*, different labeling functions, and a set of *acceptance sets* of locations. All clocks proceed at the same rate and measure the amount of time that has elapsed since they were started or reset. Edges may reset a number of clocks. Each location puts certain constraints on the values of the atomic propositions and the values of the clocks. Control can reside in a particular location only if the values of the propositions and clocks satisfy the corresponding constraints.

TA have originally been defined based on a point-based strictly-monotonic real-time semantics [23]. Later, an interval-based variant have been proposed [25], which is primarily used today. In both cases, the semantics is defined in terms of timed state sequences called *runs*. The automaton configuration of TA can be determined by identifying the active location and the values of all propositions and clocks. A comprehensive discussion of the semantics, algorithms (e.g., for reachability analysis and property checking), and tool support can be found in [68]. As the main tool for modeling and verification of (networks of) TA, UPPAAL [69] provides support for automated analysis based on simulation and model checking techniques. In this context, Vogel et al. introduced a catalog of specification patterns to formulate real-time properties for system verification [593]. Over the years, several variants of TA have been defined, such as *Timed Büchi Automata* (TBA) [24], whose semantics is defined in terms of sets of accepted timed words.

Timed Transition Systems (TTSs) are another relevant class of formalisms, which were introduced as a generalization of classical transition systems [346, 483]. In contrast to untimed transition systems, they impose timing constraints on individual transitions [288, 399]. Each TTS consists of a set of *propositions*, a set of *initial observations*, and a finite set of *transitions*. Time is incorporated by restricting the times at which transitions may occur (e.g., by using lower and upper bound requirements). All transitions are assumed to happen instantaneously. Since TTSs and TA usually have a similar expressiveness, semantic-preserving translations between the two worlds are possible in most cases.

Other application-specific automata-based formalisms exist, such as real-time extensions of *Abstract State Machines* (ASMs) [95, 98], which are used as part of the *ASM mETAm modeling* (ASMETA) framework [38]. Furthermore, *PLC-Automata* [183] deal with real-time properties of programmable logic controllers. Details about the relation between temporal logics and automata-based formalisms can be found in [405].

Pattern-Based Specification Languages

Many of the frameworks presented in Section 2.1.1 do not directly involve temporal logics or automata-based formalisms, but instead rely on higher-level specification means. Pattern-based languages such as TADL [89, 336, 474], MTSL [92, 93], CCSL [34, 35, 407, 408], and AUTOSAR-TE [10, 11] are close to natural language and comparatively easy to learn for industrial engineers due to their similarity to programming languages. At the same time, most of the languages are based on theoretical formalisms and have well-

defined semantics that allow formal reasoning about their properties. Depending on their purpose, pattern-based specification languages provide support for specifying different aspects of real-time systems and behavior, ranging from high-level timing requirements of event-based systems (e.g., TADL, MTSL) to clock characteristics and relations (e.g., CCSL) as well as timing aspects of hardware/software systems (e.g., AUTOSAR-TE).

Graphical Formalisms

Graphical specification languages for real-time behavior such as *Live Sequence Charts* (LSCs) [104, 162, 272] and *Timing Diagrams* [184, 529] use compositional techniques to define automata semantics, which can be exploited for observer-based verification. These formalisms typically come from the academic world, but have already found their way into industrial development tools [49, 81]. In the broadest sense, graphical specification languages can be considered as a special form of pattern-based specifications which use graphical elements as description means and underpin them with formal semantics.

Existing formalisms for graphical specification serve different purposes, ranging from the specification of the timing behavior of single systems to formalizations of the timing of interactions between multiple systems. Alfonso et al., for example, introduced *Visual Timed Event Scenarios* (VTS) [19] as an instance of visual event-based specifications. VTS relies on the formulation of existential predicates for scenarios and can thus be used, e.g., to “*express infringements of safety or progress requirements*” [19, p. 168]. *Simplified Universal Pattern* (SUP) [81], as another representative of graphical specifications for single systems, represent real-time requirements in terms of “*temporal trigger/action relationships*” [81, p. 3]. The *triggers* and *actions* are explicitly defined and visualized in the graphical notation in order to avoid ambiguity and to facilitate quick understanding.

Other formalisms focus on the interaction between individual systems. LSCs [104, 162, 272], for example, are a well-known example of this type of specifications. LSCs are defined as an extension of basic sequence diagrams and focus on the modeling of communicating processes. While the *lifelines* show the participation of individual processes in communication, time progresses from top to bottom. LSCs can be interpreted as *universal*, i.e., to express “*restrictions over all possible system runs*” [272, p. 3]), or *existential*, i.e., to specify behavior that “*must be satisfied by at least one system run*” [272, p. 3]). The behavior before particular LSCs can be characterized by means of *precharts*. In addition, hierarchical specifications are supported by using so-called *subcharts* [162].

Practical Approaches

Over the years, alternative approaches for specifying real-time behavior have emerged in industrial practice. A common solution is to enrich behavioral models with temporal constraints. Bouskela et al., for example, annotated Modelica [427] models with requirements formalized in the *FOrmal Requirements Modeling Language* (FORM-L) [101], which supports the specification of timing constraints by means of the *Extended Temporal Language* (ETL) [100]. The approach allows simulation-based verification of requirements and has been demonstrated on a high-lift system from the aerospace domain.

In addition to approaches for formalizing real-time properties, concepts for dealing with heterogeneous specifications and models are an important building block. A well-known example is the interaction of different MoCs [386] through so-called *conversion blocks* in Matlab/Simulink [569, 570]. By providing strategies for the discretization and interpolation of signals, heterogeneous ensembles of discrete, continuous, and synchronous subsystems can be considered in an integrated manner. While modeling in Matlab/Simulink is mainly limited to functional aspects, other tools, such as the Ptolemy II framework [208], support modeling and analysis of heterogeneous real-time systems. With the use of so-called *directors*, which can be seen as particular schedulers, and specialized interaction components called *adaptors*, the Metropolis framework [169] also enables the integration of heterogeneous components into a higher-level and consistent system model.

Giotto [286, 287], which is based on the *Logical Execution Time* (LET) paradigm [353], is a particular instant of the Metropolis framework. It is based on a strict time-triggered execution paradigm and supports modeling of multi-rate, multi-mode systems. The underlying scheduling policy can be exploited for realization on target platforms. Giotto serves as a basis for the design of other specification languages, such as *Timing Definition Language* (TDL) [489], which provides a high-level textual notation for defining the timing behavior of real-time applications. TDL is based on a simple component model consisting of a set of *modules* that can have multiple *modes* in which different *tasks* are executed. Moreover, tool support for modeling and simulation of TDL specifications in Matlab/Simulink as well as for code generation and platform deployment exist.

2.1.3 Contract-Based Design

An important step towards uniting the numerous specification languages into a common framework was the elaboration and increasing use of *contract-based design* (CBD) [70, 71]. CBD is a design paradigm that is intended to be used “*at all stages of system design, from early requirements capture, to embedded computing infrastructure and detailed design involving circuits and other hardware*” [521, p. 219]. It is based on the annotation of formal or semiformal specifications to structural system models to establish direct links between the specifications and the entities to be specified. The system models are typically “*divided hierarchically into components and developed in a top-down manner, using contracts as a means to divide responsibilities and manage the complexity of the system*” [394, p. 77]. In the context of this work, we rely on a special form of contracts known as *Assume/Guarantee* (A/G) contracts and use them as a tool for specifying real-time behavior. Figure 2.1 shows the annotation of an A/G contract to a component.



Fig. 2.1: Annotation of an A/G contract to a functional component

A/G contracts are specifications expressing (1) what the legal context of a component is (*assumption*) and (2) what the component is supposed to do (*guarantee*). Such contracts are not designed for arbitrary contexts and can be understood as “Lego Bricks” of specification. In the context of this work, we consider A/G contracts formulated over input/output event traces of components, i.e., assumptions and guarantees each reference one or more ports and specify valid behavior in terms of sets of valid event traces. The semantics of a given A/G contract $\mathcal{C} = (A, G)$ is defined as $A \implies G$. We introduce the logical values *true* and *false* as the simplest form of assumptions and guarantees. Based on this, we can define the trivial contract $(\text{true}, \text{true})$, which is always fulfilled.

Formal reasoning about the fulfillment of contracts and their relations is enabled by a set of contract operations, namely *satisfaction*, *conjunction*, *composition*, and *refinement*. To define these operations, we consider a given system S that consists of a set of hierarchically nested components $\mathbf{C}$. Each component $c \in \mathbf{C}$ has a set of ports P_c . In addition, we assume that the behavior of $c \in \mathbf{C}$ can be characterized as a timed language L_c . The corresponding system model is formally defined in Section 2.3.1 and refined in Section 4.2.2.

The *satisfaction* operator expresses that a component fulfills its contract:

Definition 1 (Satisfaction). Let $c \in \mathbf{C}$ be a component and $\mathcal{C}_c = (\Phi_c^A, \Phi_c^G)$ a contract that specifies valid behavior of c in terms of a set of valid timed traces L_φ over P_c . We say that c *satisfies* $\mathcal{C}_c$, denoted by $c \models \mathcal{C}_c$, if $L_c \cap L_{\Phi_c^A} \subseteq L_{\Phi_c^G}$. $\diamond$

In this context, the logical conjunction operator ($\wedge$) can be used to combine different assumptions or guarantees. The operator is typically used implicitly, i.e., a component must satisfy all guarantees if the environment complies with all specified assumptions. Consequently, the semantics of a given A/G contract $\mathcal{C}_c = (\Phi_c^A, \Phi_c^G)$ of $c \in \mathbf{C}$ with $\Phi_c^A = \{A_1, \dots, A_m\}$, $\Phi_c^G = \{G_1, \dots, G_n\}$, and $m, n \in \mathbb{N}$ can be refined to

$$(A_1 \wedge \dots \wedge A_m) \implies (G_1 \wedge \dots \wedge G_n) \quad (2.1)$$

based on [93, p. 10]. In addition, the operator can also be used to combine the assumptions and guarantees of several contracts that are assigned to a component $c \in \mathbf{C}$:

Definition 2 (Conjunction). Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two contracts. We say that $\mathcal{C}$ results from the *conjunction* of $\mathcal{C}_1$ and $\mathcal{C}_2$, i.e., $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$, if $\Phi^A = \Phi_1^A \cup \Phi_2^A$ and $\Phi^G = \Phi_1^G \cap \Phi_2^G$. $\diamond$

CBD relies heavily on compositional reasoning schemes based on two contract operations called *composition* ($\otimes$) and *refinement* ($\preceq$). These operations concern the relationship between multiple contracts and enable independent development of system components. From a general point of view, the *composition* operator defines “*how new components are obtained from the composition of other (sub-)components*” [370, p. 35]. In terms of CBD, the operator focuses on the composition of contracts and is defined as follows:

Definition 3 (Composition). Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two contracts. We say that $\mathcal{C}$ is *composed* of $\mathcal{C}_1$ and $\mathcal{C}_2$, denoted by $\mathcal{C} = \mathcal{C}_1 \otimes \mathcal{C}_2$, if $\mathcal{C}$ results from the combination of $\mathcal{C}_1$ and $\mathcal{C}_2$. We refer interested readers to Benveniste et al. [70, p. 45–50] for further details. $\diamond$

2.1 Existing Approaches and Technologies

The operator can also be used in the other direction, i.e., to decompose a given higher-level contract $\mathcal{C}$ of a component $c \in \mathbf{C}$ into two or more lower-level contracts $\mathcal{C}_1, \dots, \mathcal{C}_n$ of components $c_1, \dots, c_n \in \mathbf{C} \setminus \{c\}$ with $\mathcal{C} = \mathcal{C}_1 \otimes \dots \otimes \mathcal{C}_n$ and $n \in \mathbb{N}$. In practice, the iterative application of this principle allows the incremental decomposition of architectures and specifications during system design. However, the composition operator does not allow any conclusions to be drawn about the validity of the derived specifications.

To close this gap, the *refinement* operator defines “under which circumstances a component can be replaced by another without violating existing specifications” [370, p. 35]:

Definition 4 (Refinement). Let $\mathcal{C}$ and $\mathcal{C}'$ be two contracts. We say that $\mathcal{C}'$ *refines* $\mathcal{C}$, denoted by $\mathcal{C}' \preceq \mathcal{C}$, if $L_{\Phi A'} \supseteq L_{\Phi A}$ and $L_{\Phi G'} \subseteq L_{\Phi G}$. In the other writing direction, we say that $\mathcal{C}$ *is refined by* $\mathcal{C}'$, denoted by $\mathcal{C} \succcurlyeq \mathcal{C}'$, if and only if $\mathcal{C}' \preceq \mathcal{C}$. $\diamond$

According to the definition, the *refinement* relation requires that the assumptions $\Phi^{A'}$ of a refined contract $\mathcal{C}'$ must allow more context than the assumptions Φ^A of the original contract $\mathcal{C}$. On the other hand, the guarantees $\Phi^{G'}$ of the refined contract $\mathcal{C}'$ must confine the guarantees Φ^G of $\mathcal{C}$. Note that $\mathcal{C}'_1 \preceq \mathcal{C}_1$ and $\mathcal{C}'_2 \preceq \mathcal{C}_2$ implies $\mathcal{C}'_1 \otimes \mathcal{C}'_2 \preceq \mathcal{C}_1 \otimes \mathcal{C}_2$.

The interplay between composition and refinement of A/G contracts is illustrated in Figure 2.2. The top-level contract $\mathcal{C}_{tl}$ shown in the upper part is decomposed into two lower-level contracts $\mathcal{C}_1$ and $\mathcal{C}_2$, which together refine the top-level specification, i.e., $\mathcal{C}_{tl} \succcurlyeq \mathcal{C}_1 \otimes \mathcal{C}_2$. Each of the two lower-level specifications is again decomposed into a number of leaf specifications $\mathcal{C}_{1.1}, \dots, \mathcal{C}_{1.m}$ and $\mathcal{C}_{2.1}, \dots, \mathcal{C}_{2.n}$ with $m, n \in \mathbb{N}$, which together refine their individual higher-level specifications, i.e., $\mathcal{C}_1 \succcurlyeq \mathcal{C}_{1.1} \otimes \dots \otimes \mathcal{C}_{1.m}$ and $\mathcal{C}_2 \succcurlyeq \mathcal{C}_{2.1} \otimes \dots \otimes \mathcal{C}_{2.n}$. The iterative decomposition of higher-level specifications into lower-level specifications, along with the rigorous verification of refinement relations, enables *independent implementability* of the resulting specifications and the associated components. Thus, CBD is a valuable tool for use in incremental design approaches.

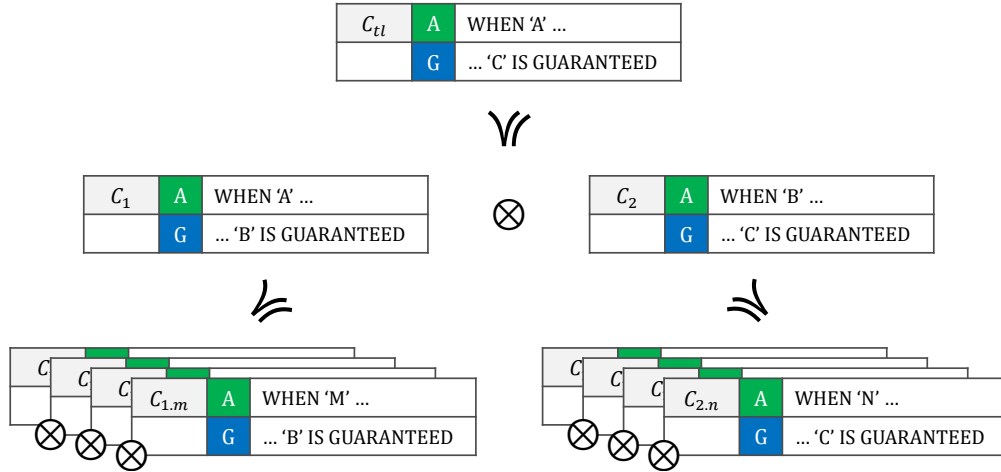


Fig. 2.2: Composition and refinement of A/G contracts

Over the past decades, CBD has been used for different purposes and in different application domains. A canonical use case is to employ contracts to provide process

support for the design of safety-critical systems. Förster et al., for example, developed a contract-based approach for dependable reuse and guarded integration of automotive software components [230]. Guissouma et al. follow a similar approach, but focus more on the continuous development of over-the-air updates [257]. In addition, efforts have been made to combine scenario-based and contract-based verification approaches [263].

The use of contract-based specifications to support common engineering tasks such as requirements formalization, timing verification of functional models, and correct implementation has been extensively covered in the literature. Dragomir et al., for example, applied contract-based modeling and verification of timed safety requirements within SysML models [189]. The integration of contract-based timing specifications into an IEC 61499-compliant modeling approach to enable automated timing verification of industrial distributed control systems [112] is another practical example of CBD. More formal approaches focus on the verification of refinement relations between functional models, e.g., by defining a formal semantics for Matlab/Simulink [563] and proving the correct refinement of Matlab/Simulink block diagrams [621]. In addition, other related work aims at ensuring the implementability of contract-based specifications. For example, Yu et al. introduced the concept of *contract replaceability* to overcome the *implementation problem* of classical A/G contracts, which can lead to situations where the implementation of a refined contract specification is not part of the original specification [615].

2.1.4 Consistency Analysis

Requirement consistency is an important prerequisite for ensuring consistent real-time behavior of cooperative systems. In recent years, several approaches have been developed to prove the consistency of real-time requirements and specifications. For example, Langenfeld et al. have introduced a scalable consistency analysis for real-time requirements based on *Phase Event Automata* (PEA) with automated tool support [382]. Following similar goals, Chen et al. aimed at achieving consistency verification of safety requirements [141]. Choi et al. proposed a systematic approach for verifying timing consistency of behavioral UML/MARTE models using rule-based and model checking-based techniques [144]. Other approaches focus on ensuring (partial) consistency of scenario-based specifications, e.g., of *Traffic Sequence Charts* (TSCs) in the field of highly automated driving [57, 59].

Another relevant application of consistency analysis is the use of consistency verification methods to perform VITs. In this context, the combined use of MBSE and formal (or semiformal) specifications “allows systems to be virtually integrated based on the models of their subsystem and the architecture specification of the system. Such virtual integration thus allows detecting potential integration problems up front, [i.e.,] in the early phases” [521, p. 223] of the development process. The systematic use of VITs is one of the main building blocks of CBD [521] and supports the verification of refinement relations. Several methods and tools have been developed for proving consistency and compatibility of contract-based specifications, e.g., the *Contracts-Refinement Proof System* [149] with tool support provided by OCRA [147] and approaches for multi-aspect virtual integration of real-time and safety properties, enabling incremental architecture design [166, 241].

2.1.5 Real-Time and Scheduling Analysis

Today, engineers can rely on wide variety of methods and tools for analyzing previously modeled real-time systems, which can be divided into *simulation-based* [310, 316, 460, 589], *analytical* [53, 132, 269, 285, 310, 368], and *state-based* [30, 58, 69, 150, 151, 284] techniques. The existing approaches serve different purposes such as worst-case execution time (WCET) analysis [15, 211, 352], response-time analysis [30, 53, 269, 285, 310, 589], timing and schedulability analysis of specific MoCs, e.g., of synchronous data flow graphs [539], as well as data-propagation delay analysis in cause-effect chains [60, 246, 285].

In many cases, existing verification means for real-time and scheduling analysis have already been transferred to industrial application and are part of commercial products such as TA Tool Suite [589], chronSUITE [310], EmbeddedSpecifier [49] and EmbeddedValidator [50] as well as Timingsuite T1 [525]. Other tools such as SystemDesk [629] and APP4MC [200] allow the modeling of AUTOSAR architectures and detailed hardware/software systems that can be transformed into input models for state-of-the-art analysis tools.

Existing analysis techniques differ in terms of (1) the accuracy of their analysis results and (2) their computational complexity [475]. Figure 2.3 focuses on the first aspect and compares the accuracy of simulation-based, analytical, and state-based approaches. As a reference, an example system behavior is assumed, which is shown on the left. The horizontal lines represent possible behaviors, e.g., response times of a real-time task that is executed on a processing unit. The shortest response time is referred to as the *best-case* response time, while the longest response time is denoted as *worst-case* response time.

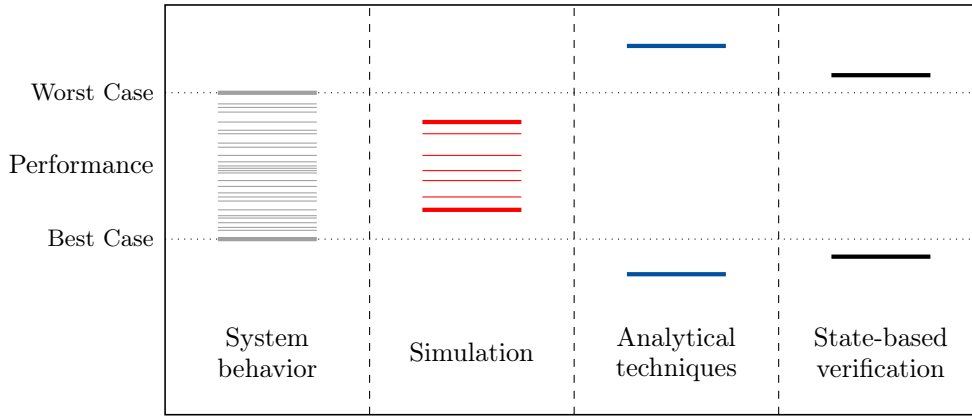


Fig. 2.3: Comparison of the accuracy of common analysis classes (based on [475, p. 9])

Simulation-based approaches investigate system behavior by examining individual runs of a system based on a predefined simulation strategy, so that typically only parts of the overall behavior are considered and analyzed. Depending on the strategy and the chosen parameters, there is no guarantee that the best and worst case times will be found. Simulation-based techniques can thus be considered as *underapproximating* analysis methods. Analytical real-time analyses attempt to characterize the temporal behavior of systems by using (combinations of) mathematical equations. Since it is generally

impossible to describe all edge cases with sufficient accuracy, it is usually necessary to compromise on accuracy. The results of such analysis methods are *overapproximating* in most cases, i.e., they include behavior that can never be seen in reality. State-based analysis methods completely unroll the state space and explore it to check whether the specified properties are satisfied. A complete unrolling and exploration of the state space is commonly referred to as *model checking*. Analysis tools of this class are generally highly accurate, but have limited scalability. The underlying problem of *state space explosion*, which is also known as the *state explosion* problem [154], can be addressed by several techniques [443], but still remains one of the main problems in applying state-based verification techniques such as model checking in industrial environments.

2.1.6 Run-Time Monitoring³

Over the past decades, observers and monitoring techniques have been widely used in system design, especially as part of simulations or to support model checking. Typical examples related to real-time behavior are the execution of requirement observers generated with design tool such as EmbeddedSpecifier [49] on hardware-in-the-loop (HIL) platforms [627] or the use of observer automata in UPPAAL for testing real-time properties [289]. In general, a formal representation of desirable or safety-relevant properties serves as input for the manual construction or the automatic generation of observers that can be used for *design-time* or *run-time monitoring* [226, 530]. In addition to basic formalisms such as temporal logics, higher-level specification languages can be used for this purpose [109, 206]. In the following, we focus on the monitoring of real-time properties at *run-time*, i.e., during or after the execution of (parts of) the system in the operating phase.

In the field of run-time monitoring, there is a large body of work and different approaches for dealing with common verification tasks. On the one hand, *online monitoring* techniques can be used to obtain a verdict on the validity of the system’s behavior with respect to a set of requirements during execution [180, 206]. In this case, a “*trade[-]off between detection latency and resource overhead*” [429, p. 252] must usually be made. The term of *online verification* refers to the systematic and focused use of online safety checks and run-time monitoring as part of the system to ensure operation in compliance with a given set of specifications [473, 551]. On the other hand, *offline monitoring* enables the retrospective analysis of recorded data from the operation of systems, e.g., based on traces over a selected set of observables [188]. *Distributed run-time monitoring* [226, 530] applies run-time monitoring techniques to distributed systems. It enables the continuation of timing-related *Verification & Validation* (V&V) activities that have been started at design time and keep them up over the entire system’s life cycle until decommissioning. Furthermore, it is foreseeable that not all behaviors resulting from the interaction of individual CSs in highly dynamic environments can be conclusively verified and validated at design time. Run-time monitoring can play an crucial role in closing this gap.

³An earlier version of this section has been published in [206]. The corresponding passage in the original publication was written by me and was finalized with editorial changes by my co-authors.

To ensure the correct and timely functioning of monitoring approaches for distributed systems, additional components are usually required that have to be integrated into the systems under observation. These include, for example, specialized components that make relevant events visible to the monitoring mechanisms, logging facilities, or clock synchronization techniques that help to establish a common time domain between the distributed components. At the same time, consistency between the V&V methods must be ensured at design time and at run-time. The monitoring infrastructure for safety-critical systems must also be regarded as safety-critical and must thus be developed with the same care as the actual system in order to meet safety and real-time requirements. Ehmen et al., for example, have implemented an approach for measurement-based online verification of timing properties in distributed systems that relies on an external measurement infrastructure [206]. It is based on observers generated from MTSL specifications iteratively refined during design time. The approach ensures consistency between methods in different lifecycle phases by reusing specifications from the design phase, while reducing the impact of monitoring on system behavior by using a measurement infrastructure.

Run-time monitoring can also be an important tool for ensuring consistent and coherent handling of real-time behavior, especially when applied in iterative development processes. It allows the collection of data in the operating phase that can be used to develop software updates, e.g., for debugging, optimization, or adaptation. This aspect is also relevant for cooperative SoCPSs, since software updates can contribute to ensuring interoperability and facilitate continuous optimization of cooperative processes. Due to the scope of this work, we will focus on developing methods and tools for ensuring real-time behavior from a system design perspective, but ensure connectivity to monitoring techniques.

2.2 Comparison of Existing Frameworks

Most of the existing frameworks presented in Section 2.1.1 provide comprehensive support for ensuring real-time behavior of individual systems, including modeling, specification, analysis, implementation, verification, and validation of real-time requirements and properties at different levels of abstraction and from different perspectives. They focus on the systematic and standards-compliant design of non-cooperative systems.

To support the design of cooperative SoCPSs, it is desirable to build upon and extend an existing framework to avoid duplication of work and to achieve higher levels of process coverage. In this context, we see the opportunity to establish a method for the design of both cooperative and non-cooperative systems at different levels of abstraction. Due to the large number of possible candidates, a careful selection of a suitable methodology to serve as the basis for the planned extensions is required. We will evaluate the suitability of the existing frameworks for ensuring real-time behavior of cooperative SoCPSs (1) according to four selection criteria, and (2) by comparing their ability to address the identified challenges based on the available specification and verification means. To make it more concrete, a methodology is sought that meets the following *selection criteria*:

Focus on Early Design Phases The framework *must* allow modeling and specification of (systems of) systems in early, conceptual design phases, in which details of the technical implementation are not yet available.

Support of Timing Specifications The framework *must* enable specification of timing properties of reactive, component-based (systems of) systems, subsystems, and components.

Cross-Layer Design Support The framework *must* support modeling and specification of (systems of) systems at different levels of abstraction and from different viewpoints/perspectives.

Transferability into Industrial Practice The framework *must* be comprehensible to (requirements) engineers and compatible with established modeling tools to facilitate a later transfer into industrial practice.

In addition, it would be desirable to already have basic support for at least some of the *relevant aspects* that cover the challenges identified in Section 1.2.3:

Systems of Cyber-Physical Systems The framework *should* support the handling of SoCPSs.

Dynamic System Architectures The framework *should* support the handling of dynamic system architectures.

Unreliable Communication The framework *should* support the handling of unreliable communication.

Imprecise Local Clocks The framework *should* support the handling of imprecise local clocks.

Following an extensive literature review, the existing frameworks were evaluated according to the selection criteria and relevant aspects. In all cases, the availability and expressiveness of the design support were rated in terms of three quality categories, i.e., *fully supported* (●), *partially supported* (○), and *not supported* (–). The results were then transferred to Table 2.1 and annotated with references to allow for comparative analysis.

The table summarizes the findings from Section 2.1.1. Early system design, which is particularly relevant for the consistent specification and modeling of cooperative behavior, is addressed by EAST-ADL with TADL, MULTIC/MULTIC-Tooling, UML/MARTE with CCSL, MechatronicUML and Eclipse CHESS. The Rubus Component Model, Eclipse APP4MC, and AADL are also capable of describing, for example, functional architectures, but have a stronger focus on later development phases such as hardware/software design and implementation. All frameworks provide support for specifying timing properties, but focus on different aspects. Eclipse APP4MC and AADL, for example, do not explicitly support formulating higher-level timing specifications and are more concerned with low-level timing properties of hardware/software systems. Except for AADL, all existing frameworks have cross-layer design support. However, AADL allows the import of models

2.2 Comparison of Existing Frameworks

Tab. 2.1: Comparison of existing frameworks

<i>Existing Framework</i>	<i>Selection Criteria</i>				<i>Relevant Aspects</i>				<i>References</i>
	<i>Focus on Early Design Phases</i>	<i>Support of Timing Specifications</i>	<i>Cross-Layer Design Support</i>	<i>Transferability into Industrial Practice</i>	<i>Systems of Cyber-Physical Systems</i>	<i>Dynamic System Architectures</i>	<i>Unreliable Communication</i>	<i>Imprecise Local Clocks</i>	
EAST-ADL with TADL	●	●	●	●	–	○	●	●	[161, 197, 474]
MULTIC, MULTIC-Tooling	●	●	●	●	–	–	○	○	[92, 93]
UML/MARTE with CCSL	●	●	●	○	○	○	–	●	[34, 35, 457]
MechatronicUML	●	●	●	○	●	○	–	●	[196, 292, 293]
Eclipse CHES (incl. OCRA)	●	●	●	○	○	○	–	–	[145, 149, 176]
Rubus Component Model	○	●	●	●	–	○	○	–	[39, 115, 117]
AUTOSAR, AUTOSAR-TE	○	●	●	●	○	○	–	○	[5, 6, 10, 11]
Eclipse APP4MC	○	○	●	●	–	○	–	○	[200, 296, 607]
AADL	○	○	○	●	–	○	–	○	[221, 223, 510]

● = *fully supported* ○ = *partially supported* – = *not supported*

from other design phases. All of the frameworks are already used in industry or are in principle suitable for transfer into industrial practice. Nevertheless, UML/MARTE with CCSL, MechatronicUML, and Eclipse CHES have limited applicability due to the complexity of the underlying modeling languages or methodologies and would most likely require more extensive training of industrial requirements engineers.

With respect to the support for relevant aspects of cooperative SoCPSs, an important commonality of most of the existing approaches is the limitation to the handling of an individual system without taking into account communication or cooperation processes with other participants. MechatronicUML is the only framework that provides basic support for designing SoSs by relying on scenario-based specifications. Most frameworks offer lightweight support for handling dynamic system architectures by defining description means for operating modes and simple mode dependencies. The modeling of unreliable communication is supported by EAST-ADL with TADL and the Rubus Component Model in terms of probabilistic timing specifications that constrain the occurrence of events within time intervals. In addition, MULTIC/MULTIC-Tooling introduces pattern extensions that can be used to formulate quantitative properties, i.e., specifications that must only apply in k out of n cases. EAST-ADL with TADL, UML/MARTE with CCSL, and MechatronicUML provide extensive support for handling imprecise local clocks and allow reasoning about their relations based on CCSL constraints. Moreover, first attempts to map global specifications to local clocks were made in MULTIC/MULTIC-Tooling.

The selection criteria defined above are fully met by both EAST-ADL with TADL and MULTIC/MULTIC-Tooling. Therefore, we need to consider other factors when making the selection. On the one hand, EAST-ADL with TADL innately provides slightly better support for relevant aspects of cooperative SoCPSs. On the other hand, the author of

this thesis was directly involved in developing the MULTIC methodology and supported the implementation of prototypical tool support in MULTIC-Tooling. For this reason, in-depth knowledge about most relevant aspects of the methodology and the internals of the Eclipse-based tooling are available. Moreover, the work can benefit from unrestricted access to all previous works due to personal involvement in both projects.

Based on careful considerations following the comparative analysis and the pragmatic decision to aim for easy extensibility, the MULTIC methodology is selected as the basis for the planned extensions. In addition, parts of the results are inspired by the scenario-based approach of MechatronicUML and selected aspects of other frameworks, which are described in more detail in Chapters 3 to 6. As a first step, we will take a closer look at the MULTIC/MULTIC-Tooling framework to identify extension points for integrating enhanced specification and verification means to add support for cooperative SoCPSs.

2.3 Multi-Layer Time Coherency Methodology

As we have seen in Sections 2.1.2 to 2.1.6, there is a wide variety of methods and tools for various engineering tasks that contribute to the design of safety-critical systems. However, the final results of timing verification and safety assurance are mainly based on expert knowledge, since the available methods and tools represent more or less “isolated” solutions for handling real-time properties that have to be integrated manually. Recall that ADAS/AD components are typically developed by different teams using different tools for different functional purposes. This causes the emergence of heterogeneous designs with breaches across control layers with respect to time. The resulting discontinuities between specification, implementation, and verification of ADAS and AD functions lead to high design and debugging efforts as well as high efforts for verifying real-time properties.

A well-known solution to these challenges is the MULTIC/MULTIC-Tooling framework [92, 93], which have already been introduced in Section 2.1.1 and was selected as a candidate for the planned extensions. The framework consists of two main components:

1. The *MULTIC methodology* provides rigorous methods to ensure correct and timely functioning of ADAS and AD functions with layered architectures [92, 205]. It is based upon four essential design paradigms that address the challenges of the first problem area. They can be instantiated in various design processes using different modeling and programming languages. The methodology is tool independent and can be applied at different levels of abstraction as well as in combination with different standards and process models from different application domains. The MULTIC methodology is introduced by presenting the four design paradigms in Section 2.3.1 and discussing an example design approach in Section 2.3.2.
2. The *MULTIC tool support* comprises design and analysis tools for modeling, validation, and analysis of functional system architectures annotated with MTSL timing contracts. In particular, it provides support for the consistent formulation and correct refinement of timing specifications by means of simulation-based VITs [93, 164]. The MULTIC tool support is described in detail in Section 2.3.3.

2.3.1 Design Paradigms

The results of the MULTIC project can be summarized by giving the four building blocks of the proposed methodology, which are referred to as *design paradigms* [92, 205]. All paradigms are generally well-known in academia or industrial practice. Even though they were extended as part of the MULTIC project in order to fit the particular needs of ADAS/AD design, it is foremost their interplay which is the novelty of the approach.

Compositional Semantic Framework

The first design paradigm is the *Compositional Semantic Framework*. It provides a generic component model consisting of hierarchically nested *components*. Each component has *interfaces* that are connected by *signals*⁴. The component interfaces are defined in terms of sets of *ports*, whose types determine which values can be observed on them. Note that the behavior of components is visible at their interfaces, which enables interaction. The compositional semantic framework serves as a carrier for the other three design paradigms and allows the annotation of *timing specifications*, the integration of MoCs, and the modeling of complex interactions by means of *interaction components*.

According to the formal definition of the component model [92, 93], each system S is modeled as a set of hierarchically nested components $\mathcal{C}$. Each component $c \in \mathcal{C}$ is equipped with two disjoint sets of *input ports* P_c^{in} and *output ports* P_c^{out} . In order to provide a meaningful functionality, each component must have at least one port belonging to one of these classes, i.e., $P_c := P_c^{in} \cup P_c^{out} \neq \emptyset$. The set of *system ports* $\mathbf{P} = \mathbf{P}_{in} \cup \mathbf{P}_{out}$ is defined as the union of all ports of all components that are part of the system, i.e.,

$$\mathbf{P}_{in} = \bigcup_{c \in \mathcal{C}} P_c^{in} \quad \text{and} \quad \mathbf{P}_{out} = \bigcup_{c \in \mathcal{C}} P_c^{out}. \quad (2.2)$$

The interaction of two or more components $c_1, \dots, c_n \in \mathcal{C}$ with $n \in \mathbb{N}$ and $n \geq 2$ is represented by *signals* $s : \mathbf{P} \rightarrow 2^{\mathbf{P}}$. A signal usually connects an output port $p_o \in \mathbf{P}_{out}$ with a set of input ports $P_{in} \subseteq \mathbf{P}_{in}$. For connections between two hierarchy levels, however, signals can also be defined between two or more input ports $s_{in} : p_{i_1} \mapsto \{p_{i_2}, \dots\}$, or two or more output ports $s_{out} : p_{o_1} \mapsto \{p_{o_2}, \dots\}$, with $p_{i_1}, p_{i_2} \in \mathbf{P}_{in}$ and $p_{o_1}, p_{o_2} \in \mathbf{P}_{out}$. The set of signals between the system's components $\mathcal{C}$ is denoted by $\mathbf{S}$.⁵

Timing Specifications

The *Timing Specifications* paradigm is instantiated by a pattern-based specification language called *MULTIC Timing Specification Language* (MTSL). The language extends its predecessor RSL [496], which is a derivative of the already proven TADL, with additional patterns and establishes a coherent time domain $\mathbb{T}$ by using notions of *dense* or *superdense* time [106], depending on the version of the specification language. In

⁴In contrast to the original term *channels*, we use the term *signals* to avoid ambiguity.

⁵An earlier version of this paragraph has been published in [371] and [370]. The corresponding passage in the original publication was written in close collaboration with Janis Kröger and Ingo Stierand.

order to support relevant time phenomena and complex interactions, the original version MTSL 1.0 allows to specify event occurrences (e.g., “*E* occurs every 50 ms with jitter 2 ms.”), event relations (e.g., “Whenever *A* occurs then *B* occurs within 100 ms.”), and causal event relations (e.g., “Reaction(*A*, *B*) within [10, 20] ms.”) in terms of contract-based specifications. In addition, pattern extensions for handling offsets, unique event occurrences, and quantitative properties can be used to formulate MTSL specifications.

In the following, a revised version of the specification language called MTSL 1.5 is used as the basis for the planned extensions. MTSL 1.5 adapts the syntax and semantics of the original MTSL 1.0, but follows the same basic principle. A detailed description of MTSL 1.5 as well as a comparison with previous and subsequent language versions can be found in Appendix A. Both language versions have in common that they rely on the same formal foundations. The behavior of a system S or a component $c \in \mathcal{C}$ is defined based on events that occur on their ports $p \in \mathcal{P}$, or $p \in \mathcal{P}_c$, respectively. In this context, the behavior observable at port p is restricted to its value domain Σ_p specified by the port type. We assume a special value $\perp$ to be a member of every value domain, which represents the absence of a value. Each port has non-absent values for all $t \in T \subset \mathbb{T}$. An event e occurring at port $p \in \mathcal{P}_{in/out}$ is defined as a tuple $e = (\sigma, t)$ that consists of an event value $\sigma \in \Sigma_p$ and a time value $t \in T$. Based on this, the semantics of ports is defined in terms of timed traces, for which we rely on definitions from Böde et al. [93].

A *timed trace* over port p is defined as an infinite sequence $\omega_p = (\sigma_i, t_i)_{i \in \mathbb{N}}$, in which $\sigma_i \in \Sigma_p$ are elements from the value domain and $(t_i)_{i \in \mathbb{N}}$ forms a monotonic sequence of time instances. We require all timed traces to be non-zeno, i.e., for each $t \in \mathbb{T}$ exists (σ_i, t_i) such that $t_i \geq t$. Moreover, we denote the set of timed traces observable at port p by $\Omega_p = \{\omega = (\sigma_i, t_i)_{i \in \mathbb{N}}\}$. For each port set P , we define a set of timed traces $(\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}$ over P , where $\vec{\sigma}_i = (\sigma_1, \dots, \sigma_n) \in \Sigma_{p_1} \times \dots \times \Sigma_{p_n}$ holds. Similar to the consideration of individual ports, we denote the set of traces over a given set of ports P by $\Omega_P = \{\omega_P = (\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}\}$. The set of *possible timed traces* observable at the component’s interfaces can be characterized by a timed language $L_P \subseteq \Omega_P$. According to Definition 1, valid behavior of a component $c \in \mathcal{C}$ is defined as a set of *valid timed traces* of c , which is denoted by L_φ . In the following, we consider *specifications* as A/G contracts $\mathcal{C}_c = (\Phi_c^A, \Phi_c^G)$, whose assumptions Φ_c^A and guarantees Φ_c^G consist of sets of MTSL 1.5 specification patterns that define valid behavior of a component $c \in \mathcal{C}$.

Models of Computation

The third design paradigm, referred to as *Models of Computation*, concerns the embedding of MoCs into the compositional semantic framework, lifting them into a common semantic domain. In this context, MoCs can be used to model component behavior. Based on an identification of industrially relevant modeling and programming languages (e.g., Matlab/Simulink [569, 570], Stateflow [571], C/C++), the modeling of behavior using different notions of time (i.e., untimed, synchronous, discrete, continuous) is addressed.

Interaction Components

The *Interaction Components* paradigm allows the integration of heterogeneous MoCs with different notions of time by using standard interaction patterns (e.g., *n'th-order-hold* or *periodic sampling*), which are encapsulated in independent functional components. These components are referred to as *interaction components* and can be used to model component interactions. In addition, the integration of specialized components for interaction via service-oriented interfaces and complex buffering schemes is supported.

2.3.2 Design Approach

As part of the MULTIC project, the design paradigms were instantiated for use in early design phases [92, 205]. The proposed design approach focuses on formulating MTSL timing specifications based on previously derived real-time requirements, as well as iteratively refining such specifications along with the development of the functional architecture. All specifications are formulated in terms of A/G contracts, which enables formal reasoning about composition and refinement based on well-established contract operations (see Section 2.1.3). The approach was demonstrated using a simplified *Emergency Stop System* (ESS) as a real-world application example, which was defined based on the expertise of automotive *Original Equipment Manufacturers* (OEMs) and suppliers [92, Part II].

The basic principle of the MULTIC design approach is illustrated in Figure 2.4. Following the formulation of a top-level specification $\mathcal{C}_{tl}$ on system level, the specification is decomposed into a set of lower-level specifications $\mathcal{C}_{sub_1}, \dots, \mathcal{C}_{sub_n}$ with $n \in \mathbb{N}$ for the components $c_{sub_1}, \dots, c_{sub_n} \in \mathcal{C} \setminus \{c_{tl}\}$ on the next lower hierarchy level. The resulting specifications are then annotated to the components of an *Initial Functional Architecture*. The design step is completed by verifying the *refinement* relation between the top-level specification and the composition of its lower-level specifications, i.e., by proving $\mathcal{C}_{tl} \succcurlyeq \mathcal{C}_{sub_1} \otimes \dots \otimes \mathcal{C}_{sub_n}$ (see Definitions 3 and 4). For this purpose, a VIT is performed that ensures the correctness of the decomposition (see *Functional Design – Level A*).

With each decomposition of the functional architecture, the annotated MTSL timing specifications are decomposed into more detailed (sub-)specifications, which are annotated to the newly introduced (sub-)components (see *Functional Design – Level B*). The correctness of each decomposition step is ensured by performing corresponding VITs. This principle is applied down to the technical system design and can then be continued on the basis of technical architectures (see *Technical Design*). The final step is to verify that the implementations of all leaf components comply with their refined MTSL specifications. To do so, *satisfaction checks* are performed to prove *satisfaction* of all timing specifications annotated to components on the lowest hierarchy level (see Definition 1).

Virtual integration testing (see Section 2.1.4) is a vital element of the MULTIC methodology. To explain the basic principle of VITs as applied as part of the design approach, we consider the ESS application example from the original publication. The component model of the *Emergency Stop System* is shown in Figure 2.5. The system is designed to take over control of a vehicle if the driver becomes incapacitated and executes a minimum risk maneuver to prevent accidents. It receives *Frame* events generated by

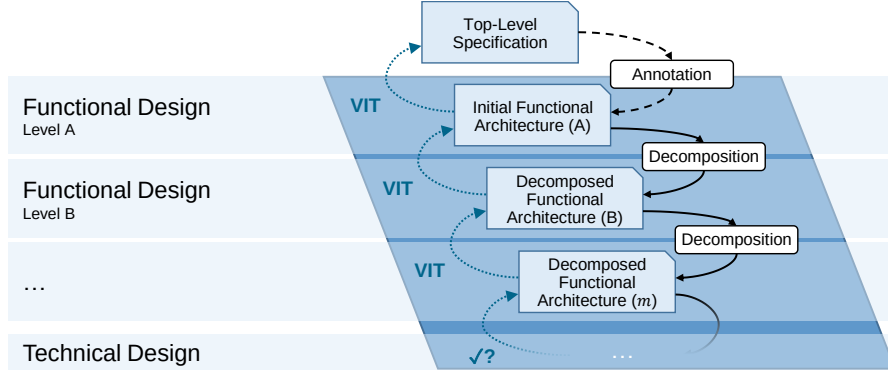


Fig. 2.4: Design approach according to the MULTIC methodology

an upstream *Imager* and *Egopose* events provided by an external *Electronic Stability Control* (ESC) component. Depending on the current traffic situation, the vehicle should either stop in its lane or safely steer to the shoulder. In order to provide the desired functionality, the system comprises four lower-level components. The *Lane Detection* component extracts curve segments representing the detected lane markings derived from the *Frame* events. These curve segments are then compiled by the *Environment Model* component, resulting in continuously evolving *RoadModel* updates. The *Trajectory Planning* component computes a safe path in terms of *TrajectoryPoints* with a given time horizon based on the *RoadModel* updates. Finally, the calculated *TrajectoryPoints* are processed by the *Dynamic Control* component, which computes corresponding *LR* and *QR* control values for longitudinal and lateral vehicle control to follow the safe path.⁶

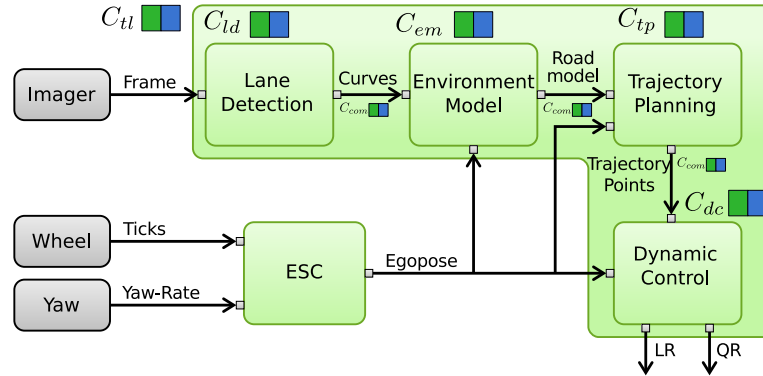


Fig. 2.5: Component model of the *Emergency Stop System* [92, p. 92]

As part of the MULTIC project, the timing behavior of the example system was specified in terms of five MTSL specifications $\mathcal{C}_{ul}$, $\mathcal{C}_{ld}$, $\mathcal{C}_{em}$, $\mathcal{C}_{tp}$, and $\mathcal{C}_{dc}$, which are listed in Tables 2.2 to 2.6. A detailed description of the intended system behavior and

⁶An earlier version of this paragraph has been published in [164]. The corresponding passage in the original publication was written in close collaboration with Ingo Stierand.

2.3 Multi-Layer Time Coherency Methodology

Tab. 2.2: Top-level specification of the *Emergency Stop System*

A	Frame occurs every 33 ms with offset [0, 33] ms and jitter 5 ms.	1
	Egopose occurs every 10 ms with offset [0, 10] ms and jitter 5 ms.	2
G	LR occurs every 5 ms with offset [0, 5] ms.	3
	QR occurs every 1 ms with offset [0, 1] ms.	4
	Age(Frame, LR) within (0, 800] ms.	5
	Age(Frame, QR) within (0, 800] ms.	6

Tab. 2.3: Specification $\mathcal{C}_{ld}$ of the *Lane Detection* component

A	Frame occurs every 33 ms with offset [0, 33] ms and jitter 5 ms.	1
G	Age(Frame, Curves) within (0, 60] ms.	2
	Reaction(Frame, Curves) within (0, 33] ms 3 out of 4.	3

Tab. 2.4: Specification $\mathcal{C}_{em}$ of the *Environment Model* component

A	Curves occurs every [33, 66] ms with offset [0, 66] ms and jitter 33 ms.	1
	Egopose occurs every 10 ms with offset [0, 10] ms and jitter 5 ms.	2
G	Age(Curves, RoadModel) within (0, 250) ms.	3
	RoadModel occurs every 100 ms with offset [0, 100] ms and jitter 50 ms.	4

Tab. 2.5: Specification $\mathcal{C}_{tp}$ of the *Trajectory Planning* component

A	RoadModel occurs every 100 ms with offset [0, 100] ms and jitter 100 ms.	1
	Egopose occurs every 10 ms with offset [0, 10] ms and jitter 5 ms.	2
G	Age(RoadModel, TrajectoryPoints) within (0, 250) ms.	3
	TrajectoryPoints occurs every [100, 150] ms with offset [0, 150] ms.	4

Tab. 2.6: Specification $\mathcal{C}_{dc}$ of the *Dynamic Control* component

A	TrajectoryPoints occurs every [100, 150] ms with offset [0, 150] ms.	1
	Egopose occurs every 10 ms with offset [0, 10] ms and jitter 5 ms.	2
G	Age(TrajectoryPoints, LR) within (0, 155] ms.	3
	Age(TrajectoryPoints, QR) within (0, 151] ms.	4
	LR occurs every 5 ms with offset [0, 5] ms.	5
	QR occurs every 1 ms with offset [0, 1] ms.	6

its implications for the timing specifications can be found in [93, Part II]. Note that simplified specification symbols $\mathcal{C}_{tl}$, $\mathcal{C}_{ld}$, $\mathcal{C}_{em}$, $\mathcal{C}_{tp}$, and $\mathcal{C}_{dc}$ had to be used in Figure 2.5 for technical reasons. All specifications were transferred from MTSL 1.0 to MTSL 1.5 in order to be able to reuse them for the evaluation of the tool support in Section 7.4.

As already mentioned earlier, the design approach relies heavily on the systematic use of VITs to prove correct refinement and consistency of MTSL timing specifications. The verification technique aims to answer the following two key verification questions:

1. Are the assumptions of all lower-level components consistent with the guarantees of neighbor components and the assumptions of the top-level component?
2. Do the specifications of the lower-level components imply the specification of the top-level component?

2 Methods and Tools for Ensuring Real-Time Behavior

Based on the VIT proof obligation $\mathcal{C}_{tl} \succ \mathcal{C}_{sub_1} \otimes \dots \otimes \mathcal{C}_{sub_n}$ (see Definitions 3 and 4) with $\mathcal{C}_{tl} = (A_{tl}, G_{tl})$ and $\mathcal{C}_{sub_1} = (A_{sub_1}, G_{sub_1}), \dots, \mathcal{C}_{sub_n} = (A_{sub_n}, G_{sub_n})$, the verification questions can be translated into two practical conditions based on [241]:

$$A_{tl} \wedge G_{sub_1} \wedge \dots \wedge G_{sub_n} \implies A_{sub_1} \wedge \dots \wedge A_{sub_n} \quad (2.3a)$$

$$A_{tl} \wedge G_{sub_1} \wedge \dots \wedge G_{sub_n} \implies G_{tl} \quad (2.3b)$$

Note that the derived conditions are only valid, if $A_{sub_1}, \dots, A_{sub_n} \subseteq A_{tl}$ holds. In the context of the ESS example, the VIT proof obligation can be formulated as $\mathcal{C}_{tl} \succ \mathcal{C}_{ld} \otimes \mathcal{C}_{em} \otimes \mathcal{C}_{tp} \otimes \mathcal{C}_{dc}$, which results in the following two VIT conditions:

$$A_{tl} \wedge G_{ld} \wedge G_{em} \wedge G_{tp} \wedge G_{dc} \implies A_{ld} \wedge A_{em} \wedge A_{tp} \wedge A_{dc} \quad (2.4a)$$

$$A_{tl} \wedge G_{ld} \wedge G_{em} \wedge G_{tp} \wedge G_{dc} \implies G_{tl} \quad (2.4b)$$

After proving compliance with the conditions from Equation 2.3, or Equation 2.4 in the context of our application example, the incremental design approach can continue with the decomposition of all (or a subset of the) lower-level components into (sub-)components on the next lower hierarchy level. The result of the second decomposition of the ESS is depicted in Figure 2.6, which belongs to the *Functional Design – Level B* design phase. Consequently, a total of four VITs must be performed in order to prove the correct refinement of all lower-level specifications by their individual (sub-)specifications.

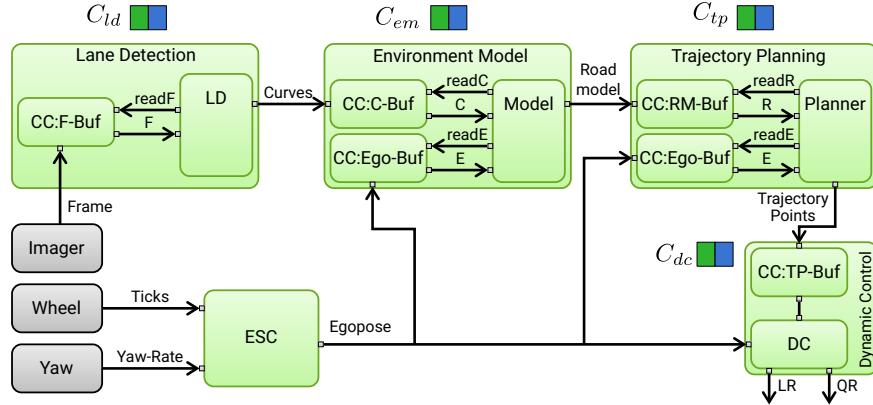


Fig. 2.6: Further decomposition of the *Emergency Stop System* [92, p. 104]

2.3.3 Tool Support

In the follow-up project MULTIC-Tooling, prototypical tool support for the automated execution of simulation-based VITs was implemented [93, 164]. The tool integrates a modeling environment based on Papyrus [202] that allows engineers to design systems using SysML and express real-time requirements in terms of MTSL timing contracts. It provides analysis of such models and requirements based on SystemC [460], which allows

2.3 Multi-Layer Time Coherency Methodology

for reasoning about the valid refinement of specifications along the system’s decomposition. To this end, it generates an executable SystemC simulation model consisting of event *generators* (G) and *monitors* (M), whose basic structure is exemplified in Figure 2.7.

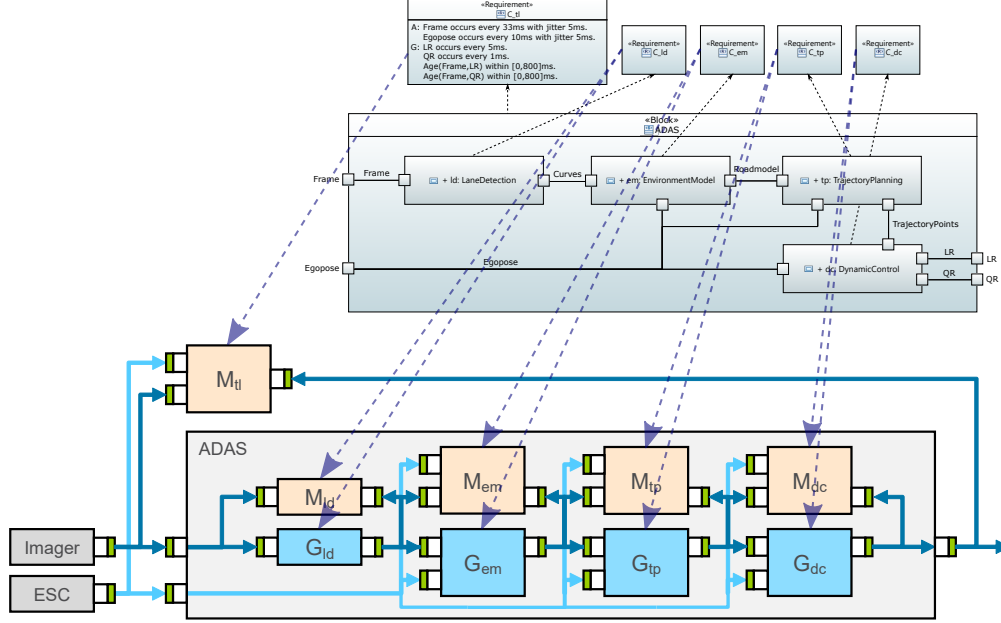


Fig. 2.7: Example SystemC simulation model [164, p. 27]

In a first step, event generators are instantiated for each top-level assumption and for each guarantee of lower-level components, generating event traces according to the previously defined timing specifications. Each block contains a monitor that checks whether the contract’s assumptions are valid and whether the top-level guarantees are implemented by the timing behavior of the lower-level components. To simplify the generation of analysis models, many SystemC-related aspects have been encapsulated into a library called *Timing Specification Language Simulator* (TSLsim), which provides all building blocks required to execute simulation-based VITs. Additional details about the generator and monitor implementation can be found in [93, Chapter 6].⁷

The MULTIC-Tooling tool prototype allows for the simulation-based evaluation of the conditions from Equation 2.3. The main function, which consists of the automated execution and diagnosis of VITs, is complemented by corresponding functions that check the syntax of previously created MTSL specifications and provide support for automated model validation. Engineers receive analysis results in the form of simulation traces in the *Value Change Dump* (VCD) format, annotations in the original design models, and textual messages indicating identified problems or required improvements. The tool prototype serves as a basis for the tool-supported specification and refinement of MTSL timing contracts. The basic principle is illustrated as an “engineering cycle” in Figure 2.8.

⁷An earlier version of this paragraph has been published in [206]. The corresponding passage in the original publication was written by me and was finalized with editorial changes by my co-authors.

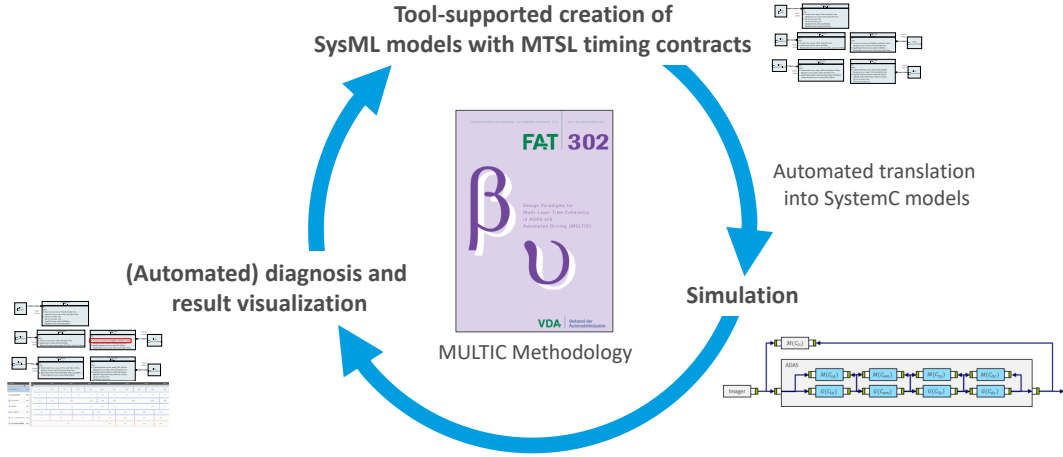


Fig. 2.8: Tool-supported specification and refinement of MTSL timing contracts

2.4 Contribution

The *design paradigms* and the *design approach* of the MULTIC methodology as well as the *MULTIC tool support* are the three main extension points that will be used to add support for ensuring real-time behavior of cooperative SoCPSs. To characterize the delta between the state of the art and the requirements of SoCPS design, a gap analysis between the results of the literature review on existing methods and tools from Section 2.1 and the research questions RQ-1 to RQ-4 was conducted. The analysis resulted in the identification of a total of six research gaps—divided into four *methodological* (MG-x) and two *technical* (TG-x) gaps—that need to be addressed to ensure real-time behavior of cooperative SoCPSs and to establish the desired SoCPS-wide timing coherence:

- MG-1** Missing process support for the design of cooperative SoCPSs
- MG-2** Missing support for handling dynamic system architectures
- MG-3** Insufficient support for modeling unreliable communication
- MG-4** Insufficient support for mapping specifications to imprecise local clocks
- TG-1** Missing integration of industry-relevant modeling tools
- TG-2** Limited tool support for automated analyses

The methodological gaps MG-1 to MG-4 each directly address one of the challenges derived in Section 1.2.3. The technical gaps TG-1 and TG-2, on the other hand, result from the (limited) technical capabilities of the MULTIC tool support presented in Section 2.3.3. The individual contributions of this work, which aim to close the identified research gaps, are presented in Table 2.7. They are each listed with the identifier and the title of the research gap, a brief description of the contribution that closes the identified gap as well as a list of research questions that are addressed by the contribution.

The first methodological gap “*Missing process support for the design of cooperative SoCPSs*” (MG-1) is addressed by the development of a correct-by-design approach for

Tab. 2.7: Research gaps and contributions

<i>ID</i>	<i>Research Gap</i>	<i>Contributions</i>	<i>RQs</i>
MG-1	Missing process support for the design of cooperative SoCPSs	Development of a correct-by-design approach for the consistent formulation and refinement of real-time requirements on SoCPS level • Scenario-based activity modeling of cooperative system behavior • Partitioning of activity models to derive system-level specifications	RQ-1
MG-2	Missing support for handling dynamic system architectures	• Extension of MTSL 1.5 to support operating modes, mode transitions, and mode-dependent timing behavior • Use of mode-dependent specifications for reasoning about dynamic reconfiguration • Enhancement of verification methods to analyze mode-dependent specifications	RQ-2.1, RQ-3
MG-3	Insufficient support for modeling unreliable communication	Derivation of a generic communication model with example parameter sets for IEEE 802.11p and C-V2X • Identification and quantification of relevant time phenomena based on a systematic literature review • Mapping of time phenomena to existing specification patterns • Definition of the communication model and parameter sets	RQ-2.2
MG-4	Insufficient support for mapping specifications to imprecise local clocks	• Reuse of existing concepts from MTSL 1.0 for defining local clocks and clock references • Systematic investigation of local clocks with typical characteristics • Formal definition of faithful and approximate clock relations	RQ-2.3, RQ-3
TG-1	Missing integration of industry-relevant modeling tools	• Implementation of a model transformation adapter for Enterprise Architect • Development of a lightweight intermediate format to facilitate future extensions	RQ-4
TG-2	Limited tool support for automated analyses	• Implementation of a modular and extensible toolchain for real-time analyses • Integration of analysis tools for performing VITs of enhanced specifications • Application of complexity reduction methods (e.g., analysis of partial models)	RQ-3, RQ-4

the consistent formulation and refinement of real-time requirements on SoCPS level. The proposed approach builds upon on the MULTIC methodology and adds support for early determination of cross-system coordination processes. It introduces a formalism called *Time-Constrained Activity Diagrams* (TCADs) that allows scenario-based modeling of cooperative behavior. The formalism enables the pre-competitive design of cooperative system functions in terms of implementation-independent interaction protocols. The resulting activity models are partitioned to assign activities and actions to individual subsystems, enabling the derivation of system-level specifications. This serves as a starting point for the parallel development of CSs based on the previously agreed top-level specifications. The enhancements aim to facilitate early collaboration between manufacturers and suppliers to develop consistent top-level specifications in early design phases that enable safe and reliable cooperation of highly automated systems.

To address the second methodological gap “*Missing support for handling dynamic system architectures*” (MG-2), the existing MTSL 1.5 is extended to support the specification of operating modes, mode transitions, and mode-dependent timing behavior. In addition, mode decomposition and resulting proof obligations for common contract operations such as composition and refinement are investigated. Subsequently, mode-dependent specifications are used to explicitly model dynamic reconfiguration. The reconfiguration processes considered concern both behavioral changes as well as adaptations of the available interfaces. Finally, existing verification means are enhanced to support analyzing mode-dependent specifications. The extensions add cross-layer process support for the specification and tool-supported refinement of mode-dependent timing specifications.

The third methodological gap “*Insufficient support for modeling unreliable communication*” (MG-3) is addressed by deriving a generic communication model with example parameter sets for IEEE 802.11p and *Cellular V2X* (C-V2X), as the two most important V2X communication standards from the automotive domain. For this purpose, a systematic literature review is conducted to identify and quantify relevant time phenomena. The identified phenomena are then mapped to existing MTSL specification patterns to enable their use in contract-based design methodologies such as the envisioned design approach. The resulting communication model and the example parameter sets allow early consideration of temporal properties of unreliable communication by making transmission failures and error correction mechanisms explicit in functional system design.

The fourth and final methodological gap “*Insufficient support for mapping specifications to imprecise local clocks*” (MG-4) is addressed by a systematic investigation of local clocks with typical characteristics, i.e., resolution, drift, skew, and difference constraints. Based on the results, the mapping of specifications to imprecise local clocks is formally defined, e.g., to support distributed run-time monitoring. The careful use of under- and overapproximations allows the construction of meaningful and traceable relations between the implicit “global clock” used at design time and local clocks. For this purpose, existing specification means from MTSL 1.0 for defining local clocks and clock references are reused. They are complemented by formal definitions of faithful and approximate clock relations. The results allow the explicit specification of clock properties, which enables the definition of local specifications tailored to the characteristics of local clocks.

Finally, the technical gaps “*Missing integration of industry-relevant modeling tools*” (TG-1) and “*Limited tool support for automated analysis*” (TG-2) are addressed by the enhancement of verification means as well as the implementation of tool support for modeling, transformation, and automated analyses based on an industrially relevant modeling tool. The contribution includes the implementation of a modular and extensible toolchain for real-time analyses. It comprises a model transformation adapter for Enterprise Architect that translates SysML design models into a lightweight intermediate format to facilitate future extensions. Existing analysis tools are integrated for performing VITs of enhanced specifications based on the intermediate format, thus enabling formal verification of contract refinement. Finally, complexity reduction methods such as the analysis of partial models are applied to cope with the resulting complexity.

The contribution addresses all research questions RQ-1 to RQ-4, and thus covers the key challenges in ensuring real-time behavior of cooperative SoCPSs. Most of the concepts, methods, and tools are demonstrated using the *Cooperative Left-Turn Assistant* (CLTA) introduced in Section 1.2.1, as an example system from the automotive domain. Nevertheless, the results can also be used to design systems from other application domains. The proposed design approach is closely linked to related work, but is unique in its entirety and provides an important contribution to the design of provably safe cooperative systems. The concepts, methods, and tools provide comprehensive support for early design phases and facilitate the transition to technical system design. The use of semiformal, contract-based specifications ensures compatibility with distributed run-time monitoring by enabling the reuse of timing specifications created at design time. Possible extensions and follow-up activities are discussed in more detail in Chapter 8.

3 Consistent Specification of Top-Level Timing Requirements

In order to address the first methodological gap “*Missing process support for the design of cooperative SoCPSs*” (MG-1), we introduce a correct-by-design approach for the consistent formulation and refinement of real-time requirements on SoCPS level. The envisioned design approach extends the MULTIC design approach and (parts of) the MULTIC design paradigms to support the engineering of real-time behavior of cooperative SoCPSs. It is based on the idea that ensuring correct and safe cooperative behavior at run-time requires close collaboration between the systems’ manufacturers at design time. Consequently, the enhancements aim to facilitate early collaboration between manufacturers and suppliers to develop consistent top-level timing specifications that enable safe cooperation of highly automated systems. The proposed design approach addresses research question RQ-1.

The chosen approach focuses on modeling cross-system coordination processes between the individual CSs of cooperative SoCPSs. In this context, the behavior of CSs is modeled in terms of *actions* that are part of higher-level *activities* and are coordinated according to previously agreed interaction protocols aimed at achieving common goals. Based on these models, real-time requirements on SoCPS level are defined that have to be fulfilled by the system in all relevant scenarios. The design approach contributes to the integration of the extended methodology into scenario-based design processes. It also ensures a seamless transition to the design of individual CSs based on the MULTIC methodology.

In the following, we will first review well-established modeling and specification languages for describing cooperative behavior and examine their suitability for use in this work. After selecting a suitable formalism, Section 3.2 gives an overview of the envisioned design approach. Sections 3.3 to 3.5 delve deeper into individual process phases and illustrate relevant design activities based on the CLTA application example.

3.1 Modeling and Specification of Cooperative Behavior

The consistent specification of cooperative behavior has strong ties to the field of *Requirements Engineering* (RE). Due to the inherent dependencies and the multidimensional system complexity of SoSs, RE is a challenging task for such systems. Lewis et al. have reviewed the underlying challenges and identified common goals in developing a generic SoS-focused RE approach [392]. Ncube and Lim discussed the weaknesses of existing RE approaches to deal with the complexity of SoSs, resulting in the formulation of a research agenda [442]. Over the years, several RE approaches for SoSs have been developed, such as SoS-ACRE [290] and REAP-SoS [194], which already address some

3 Consistent Specification of Top-Level Timing Requirements

of the identified challenges. However, most of the existing approaches focus on natural language requirements and do not explicitly support real-time properties.

A pragmatic solution to model SoSs is the use of general-purpose modeling languages such as UML [455] and SysML [454]. In addition to structural properties, these modeling languages allow the description of the dynamic behavior of CSs and cooperative SoCPSs using behavioral diagrams such as sequence and activity diagrams. For example, Lane and Bohn have applied SysML as part of a structured SoSE modeling approach [381]. Amyot et al. combined SysML with the *User Requirements Notation* (URN) [31] to explicitly model goals and contributions of different organizations in joint RE activities to support reasoning about trade-offs and adaptations in later design phases [32].

Other approaches focus on modeling SoS interfaces and aim to facilitate model-based integration of individual CSs. In this context, Bryans et al. defined a SysML modeling pattern called *Contract Pattern* to specify the interfaces between individual CSs [114]. Shames et al. have addressed the modeling of SoS interfaces across multiple levels of abstraction [534]. In addition, specifications for CSs that are to be integrated into existing SoSs have been derived based on SysML and were translated into the *COMPASS Modeling Language* (CML) [609] to enable verification of emergent properties [113]. Ingram et al. presented a case study that demonstrates the integration of an upgraded CS into an existing SoS [312]. Moreover, Mori et al. defined a viewpoint-based SysML profile for SoS modeling that focuses on describing relevant system properties from the perspective of seven viewpoints, including *structure*, *evolution*, *dynamicity*, and *emergence* [432].

Other groups of authors have pursued mission-oriented modeling. For example, Silva et al. considered *missions* as a special kind of goals and established a *mission level* modeling viewpoint [538]. The authors focused on defining the “*capabilities of constituent systems required to accomplish such missions*” [538, p. 31] and explicitly modeled “*interactions among these systems*” [538, p. 31]. The modeling of SoSs in *mKAOS Studio* is done by using the mKAOS mission-level modeling language, which is a specialized version of the *Keep All Objectives Satisfied* (KAOS) requirement engineering language [380].

There are also more generic approaches for model-based engineering of SoSs, such as the *Compositional Unified System-Based Engineering* (CUBE) methodology for feature-driven MBSE [250]. In line with the objectives of this thesis, INCOSE formulated the vision to shift the focus of MBSE from developing systems to SoSs by the end of 2035 [321]. However, current MBSE approaches typically either do not provide support for real-time properties, or their expressiveness is not sufficient to cover important properties of cooperative SoCPSs such as dynamic architectures and imprecise local clocks.

As mentioned above, we aim to select a modeling or specification language that enables process support for the design of cooperative SoCPSs. More precisely, we are either looking for a real-time formalism that is well suited for formalizing cooperative behavior, or a modeling language for cooperative systems that can be annotated with real-time requirements. In the following, we will examine the suitability of existing approaches with respect to these criteria. The available approaches can be divided into the five classes, namely *Classical Modeling Approaches*, *Business Process Modeling*, *Temporal Constraint Networks*, *Modeling Using Sequence Diagrams*, and *Activity Modeling*.

3.1.1 Classical Modeling Approaches

Most of the modeling and specification languages for single systems presented in Section 2.1.2 can also be used to model (parts of) the behavior of cooperative SoCPSs. In addition, there are some formalisms that are particularly suited for modeling such systems. Almost all of them have their origins in early attempts to model distributed real-time systems. For example, Thomesse et al. specified time constraints based on cooperation models by using temporal logic-based specifications [572]. Later, Lee addressed the modeling of concurrent real-time processes using discrete events [385].

Petri Nets (PNs) [478] are a basic formalism for modeling distributed systems. They consist of *places* and *transitions*, where the former can contain a number of tokens that are propagated through the system along the transitions according to a set of rules. Saglietti et al. modeled cooperative systems by means of *Colored Petri Nets* (CPNs) [518]. Specialized real-time variants of PNs such as *Time Petri Nets* (TPNs) [418] and *Timed Petri Nets* (TdPNs) [486] exist that allow reasoning about timing constraints.

Cooperative SoCPSs can also be modeled as networks of communicating TA [23, 68]. For example, Bischofink and Schwammberger designed fair controllers for coordinated driving maneuvers of CHAVs by modeling their behavior in terms of *Automotive-Controlling Timed Automata* (ACTA) and used UPPAAL Stratego [171] for formal verification and fairness analysis [87]. Alshalalfah and Mohamed focused on system-level modeling of cooperative driving functions by means of *Priced Timed Automata* (PTA) [63], which have prices on both transitions and locations [21]. The authors modeled the behavior of the involved systems, the used coordination protocol, and scenario uncertainties to perform safety analyses of coordinated emergency braking using UPPAAL SMC [170].

3.1.2 Business Process Modeling

As a key resource for companies and organizations, time represents an important aspect to be considered when modeling business processes. One of the first approaches for specifying time constraints in workflow systems enables the specification of *fixed-date constraints* as well as *lower* and *upper bound constraints* on time intervals between activities [203]. The work also includes a framework for “*computing internal deadlines so that the overall process deadline and external timing constraints are all satisfied*” [393, p. 191]. Over the years, a variety of different models and analysis methods have been developed, enabling support for handling optional, alternative, and concurrent workflow structures [204, 488].

Due to their formal semantics and their state-based nature, PNs are a widely used theoretical basis for the specification and analysis of workflow models. Li et al. give an overview over the variety of PN flavors that have evolved over time [393]. The authors also introduced a modeling formalism called *Timing Constraint Workflow Nets* (TCWF-nets) [393] that supports expressing “*explicit timing constraints, structural timing constraints, and internal activity deadlines*” [393, p. 191]. TCWF-nets combines the workflow modeling language *Workflow Nets* (WF-nets) and *Timing Constraint Petri Nets* (TCPNs) that extend simple PNs by “*adding minimum, maximum, and durational timing constraints*” [393, p. 182]. The annotation of execution durations and transition delays to workflow

models allows the algorithmic verification of schedulability and boundedness. Over the years, other PN-based approaches have been developed. For example, Du et al. introduced a formalism called *Logical Time Petri Nets* (LTPNs), which extends TPNs with temporal logic constraints [193]. The authors also defined the concept of *Interorganizational Logical Time Workflow Nets* (ILWNs) based on LTPNs and workflow techniques, which supports the modeling and analysis of non-physical, real-time cooperative systems.

Following a different approach, Gagne and Trudel proposed an extension of the widely used *Business Process Model and Notation* (BPMN) [453] to express *inflexible* (i.e., absolute time, single-process), *flexible* (i.e., relative time, single-process), and *temporal dependency* (i.e., inter-process) constraints in Time-BPMN [234]. Time-BPMN supports a wide variety of temporal constraints such as “*must start on*”, “*must finish on*”, “*as soon as possible*”, “*as late as possible*”, “*finish-to-start*”, and “*start-to-finish*”, among others.

The large number of available formalisms for modeling business processes makes the selection of an appropriate language a challenging task. To address this problem, Cheikrouhou et al. provided a comprehensive overview and comparison of existing approaches for specifying and verifying temporal constraints in business process models and workflow nets [137]. They consider many of the formalisms already mentioned, but also include other languages such as TPNs [300, 403], *Time Workflow Nets* (TWF-nets) [192], TA [255], and *Communicating Sequential Processes* (CSPs) [608].

3.1.3 Temporal Constraint Networks

As an alternative to time-constrained business process models, temporal constraint networks can be used “*for expressing and reasoning about temporal knowledge*” [177, p. 62]. Most existing formalisms of this type are “*supported by constraint-directed reasoning algorithm/s*” [177, p. 62] to automatically derive statements from a given set of assertions. As one of the first authors in this field of research, Allen introduced an interval algebra that represents the relations between (temporal) intervals by means of a hierarchy of reference intervals [20]. Malik and Binford specified temporal and spatial properties in terms of linear inequalities to automatically check satisfiability, determine lower and upper bounds of time variables, and evaluate different types of consistency [406].

Later, Valdés-Peréz have investigated approaches for analyzing spatio-temporal assertions in constraint networks and employed linear programming for consistency checking [583]. Dean and McDermott [175] proposed an approach to computationally solve uncertainty problems in constraint networks caused by incomplete information. Vilain et al. considered computational aspects of interval-based and point-based temporal representations. The authors defined a point algebra for temporal assertions as a computationally tractable alternative to Allen’s interval algebra and investigated different constraint propagation algorithms for temporal reasoning [592]. Dechter et al. included continuous variables into temporal constraint networks, enabling support for metric time information [177]. The authors also provided algorithms for “*finding all feasible times that a given event can occur, finding all possible relationships between two given events, and generating one or more scenarios consistent with the information provided*” [177, p. 61].

More recent work deals with the handling of dynamic constraint networks. For example, Cervoni et al. explicitly considered “*the possibility of both incremental constraint posting and constraint retraction*” [131, p. 1] in quantitative temporal constraint networks [131]. In this context, they analyzed “*conditions for dynamically maintaining consistent information*” [131, p. 6] and presented algorithms for modifying given constraint networks.

3.1.4 Modeling Using Sequence Diagrams

The description of communication processes and interaction scenarios using sequence diagrams is a well-established method in both research and industrial practice. *Message Sequence Charts* (MSCs) [274], for example, are a well-known and widely used formalism for specifying the interaction of loosely coupled systems. Two variants of MSCs have been standardized. On the one hand, *Basic Message Sequence Charts* (bMSCs) [327, pp. 17–48] allow the specification of communication scenarios. On the other hand, *High-Level Message Sequence Charts* (HMSCs) [327, pp. 119–124] can be used to define the execution order of multiple bMSCs by adding higher-level control flow structures.

The basic concept of MSCs has been refined over time to improve expressiveness and applicability of MSC specifications. For example, Alur et al. introduced *timers* and *delay intervals* to be able to define time-related properties [28]. Note that the authors focus on relative temporal constraints and do not take into account hard real-time requirements of real-world systems. Later on, Ben-Abdallah and Leue considered timing constraints in MSC specifications [66]. They reviewed four proposed MSC extensions to express timing constraints and applied automated timing consistency analyses. Finally, Krüger extended the original ITU-T recommendation [326] to introduce guards, alternatives, sequential composition, interleaving, loops, and preemption [372]. He provided a formal definition of the extended MSC semantics and introduced different meanings, i.e., *existential*, *universal*, *exact*, and *negative* interpretations, that support various application scenarios. The results also included constructive transformation schemes for translating the considered class of MSCs into assumption/commitment specifications on component level and state-based automata. MSCs have been applied for various use cases, e.g., for model-based development of a keyless entry system by using the AutoFOCUS 3 tool [224].

As an alternative approach and to overcome the deficiencies of MSCs, Damm and Harel defined LSCs [104, 162]. As already mentioned in Section 2.1.2, LSCs focus on the modeling of communicating processes. The lifelines typical for sequence diagrams show the participation of individual processes in communication. LSCs can be interpreted as *universal*, i.e., to express “*restrictions over all possible system runs*” [272, p. 3]), or *existential*, i.e., to specify behavior that “*must be satisfied by at least one system run*” [272, p. 3]). The behavior before LSCs can be characterized by means of *precharts*. In addition, hierarchical specifications are supported by using so-called *subcharts* [162]. In contrast to other formalisms such as MSCs, a significant advantage of LSCs was the ability to ensure progress by specifying the *temperature* of *locations* and *messages* [104, p. 377]:

“LSCs overcome this drawback [of not being able to enforce progress] by associating a temperature with both locations and messages. The temperature can be

3 Consistent Specification of Top-Level Timing Requirements

either hot or cold, the former indicating that progress is enforced. The analogy here is that one cannot remain at a hot location for an infinite amount of time, because then one would burn ones feet. This [...] requires that a hot location has to be left, i.e.,] the following location has to be reached. At a cold location one can stay forever without harming ones feet, i.e.,] the following location need not be reached. In terms of messages this means that a hot message has to be delivered, whereas a cold message may be lost along the way.”

The temperature of LSC elements is visualized by means of their line types. Along the vertical dimension, which focuses on the definition of locations, *dashed line segments [are used] to depict provisional progress of the instance—the run may continue downward along the line—while solid lines indicate mandatory progress—the run must continue*” [162, p. 53]. In the horizontal dimension, which focuses on message exchange, *“a dashed arrow depicts provisional behavior—the communication may indeed complete—and a solid one depicts mandatory behavior—the communication must complete”* [162, p. 53].

Over the years, various extensions and variants of LSCs have been defined. For example, Klose and Wittke proposed an extension to include *timing annotations* [356]. These annotations *“allow to specify a lower and an upper bound between two subsequent locations on one instance”* [356, p. 514]. The proposed approach enables automated verification by transforming time-annotated LSCs into TBA [24]. Harel and Marelly extended LSCs to include *clocks* and *timing constraints* [272]. The authors added support for specifying *vertical delays* and *message delays* by using a combination of variable assignments to store the current time instant and continuously evaluated conditions to define minimum and maximum delays. In addition, the concept of temperatures of locations and messages was transferred to introduce *hot* and *cold* timing constraints. The consideration of constraint temperatures allows to specify possible consequences of their violation, defining whether the LSC is violated or has to be left. Moreover, the introduction of explicit modeling support for *time events* enables to *“react to the passage of time”* [272, p. 6].

Property Sequence Charts (PSCs) [45] are another scenario-based visual specification language for temporal properties based on MSCs and UML sequence diagrams. In contrast to MSCs and LSCs, PSCs focus on describing the behavior of component-based systems, but not on specifying requirements. The specification language relies on simple precedence constraints (i.e., *past*, *present*, and *future* constraints) that are implicitly defined by the relative order of events, without providing support for quantitative real-time constraints. Later, PSCs were used as a basis for the definition of *Timed Property Sequence Charts* (TPSCs) [619, 620], which introduced support for specifying real-time requirements. The extensions rely on the introduction of *clock variables*, which can be reset after sending a message or within conditions, and are referenced in clock constraints to restrict the time intervals in which *regular* or *required* PSC messages can be sent. The semantics of TPSCs was defined by means of TBA and can be found in [619].

The concept of *Modal Sequence Diagrams* (MSDs) was defined as a UML 2.0 profile [271] to pave the way for the application of verification, synthesis, and scenario-based execution techniques from LSCs to UML models. The original version did not support the specification of timing constraints. Later, Greenyer and Brenner et al. integrated LSC

concepts to handle time and timing into MSDs based on a dense notion of time [102, 251]. As one of many application examples, Holtmann et al. used MSDs for RE and timing specification to support the development of an emergency braking system in [291, 293].

3.1.5 Activity Modeling

Activity diagrams are a graphical language for modeling processes [212]. The use of partitions for individual subsystems allows the modeling of cross-system interaction processes [91]. Dickerson et al., for example, have used SysML activity diagrams with partitions to model the behavior of SoSs [182]. Meyer et al. applied them to capture the operating principles of AD functions for consideration as part of the CUBE methodology [420]. Over time, numerous variants of activity diagrams have been defined that support modeling the interactions of complex systems and specifying real-time behavior.

As a prerequisite for formal reasoning about temporal behavior, several groups of authors have dealt with the formal definition of the semantics of activity diagrams. For example, Jarraya et al. proposed an interpretation of the meaning of SysML activity diagrams [331]. Lima et al. defined the semantics of SysML activity diagrams based on CML [395]. Such activities served as a basis for extending activity diagrams to annotate temporal properties and to map their basic elements to semantic models for automated verification. Guelfi and Mammar, for example, defined *Timed Activity Diagrams* (TADs) as an extension of classical activity diagrams [254]. Based on the introduction of a *master clock* and additional *timers*, timer references were annotated to activities and transitions to track the progress of time in individual branches of TADs. The authors considered several types of constraints, focusing on the duration of individual activities and the time between the end of one activity and the start of a subsequent activity. TADs have also been mapped to *Clocked Transition Systems* [347] to enable the translation into PROMELA models [295] for automated analysis using the SPIN model checker [294].

Jarraya et al. translated activity diagrams with annotated activity durations and transition probabilities into *Discrete-Time Markov Chains* (DTMCs) [332] to evaluate performance characteristics using the PRISM model checker [375, 581]. Andrade et al. mapped SysML activity diagrams with MARTE annotations to *Time Petri Nets with Energy Constraints* (ETPNs) [33]. The formalism provides support for specifying *execution times* and *energy consumption* of activities and transitions using the **ResourceUsage** stereotype and tagged values **execTime** and **energy** provided by MARTE. The subsequent analyses were aimed at determining end-to-end latencies and total energy consumption. Huang et al. translated SysML activity diagrams with MARTE annotations into TA to verify safety and liveness properties with UPPAAL [301]. Also, SysML activity diagrams have been transformed into *Markov Decision Processes* (MDPs) [329, 330] and *Probabilistic Automata* (PA) [51, 466, 467, 468] to verify PCTL constraints with PRISM.

Other related work defines extensions of activity diagrams that do not focus on temporal properties. For example, Huang et al. introduced a formal mathematical notation of SysML activity diagrams and defined transformations into PNs [302] to enable state space analysis using PIPE2 [186]. The work of Staskal et al. resulted in a systematic translation approach for SysML state machines and activity diagrams [552]. The approach allows the

verification of functional properties by using nuXmv [129]. In addition, SysML activity diagrams have been used for automated test case generation [613].

3.2 Process Overview

One of the main contributions of this work is the definition of a systematic, cross-manufacturer design process for ensuring real-time behavior of cooperative SoCPSs. The proposed design approach is the result of extensive literature research and years of project work in the area of cooperative systems development. Figure 3.1 provides an overview of the envisioned design approach and highlights the most important design activities.

The design approach can be divided into two phases, namely *Pre-Competitive Design of Cooperative System Behavior* and *Manufacturer-Specific Design and Implementation*. The first phase is a cross-manufacturer phase that is carried out collaboratively in a pre-competitive setting, for example, as part of standardization committees, working groups, or research projects. It is the core element of the design approach and aims to jointly define cross-system coordination processes for cooperative driving maneuvers at a high level of abstraction. The second phase, on the other hand, is carried out individually by each system manufacturer. It comprises the detailed design and implementation of CSs, for which individual design processes, internal development procedures and tools as well as confidential information, such as intellectual property, can be used.

3.2.1 Pre-Competitive Design of Cooperative System Behavior

The first phase *Pre-Competitive Design of Cooperative System Behavior* begins with the definition of a top-level functional architecture on SoCPS level. This architecture is assumed to include one component for each external system, one component representing the SoCPS as well as one component for each CS, including corresponding *aggregation* relationships between the SoCPS and CS components. It also requires the definition of a set of *roles* for each CS, e.g., LEFT-TURNING VEHICLE and STRAIGHT-AHEAD VEHICLE in case of two CHAVs that are intended to perform a cooperative left-turn maneuver. In the simplest case, systems only have one possible role, e.g., the TMS from our CLTA example can only be the TRAFFIC MANAGEMENT SYSTEM. Such a top-level architecture on SoCPS level can be defined, for example, based on reference architectures such as the ones defined in Digitaler Knoten 4.0 [361], ViVre [237], and KoKoVI [364]. Note that communication channels between the CSs involved should be considered as individual (sub-)systems and must therefore be included as independent top-level components.

The definition of a common top-level architecture serves as a basis for iterative, scenario-based activity modeling on SoCPS level. This step is aimed at specifying the desired (timing) behavior of cooperative systems based on cooperative scenarios described by means of *Time-Constrained Activity Diagrams* (TCADs). The formalism has been developed as part of this thesis and is based on preliminary work on time-annotated activity diagrams [332]. TCADs provide support for modeling relevant actions and control flows that contribute to the safe and reliable execution of cooperative maneuvers. The

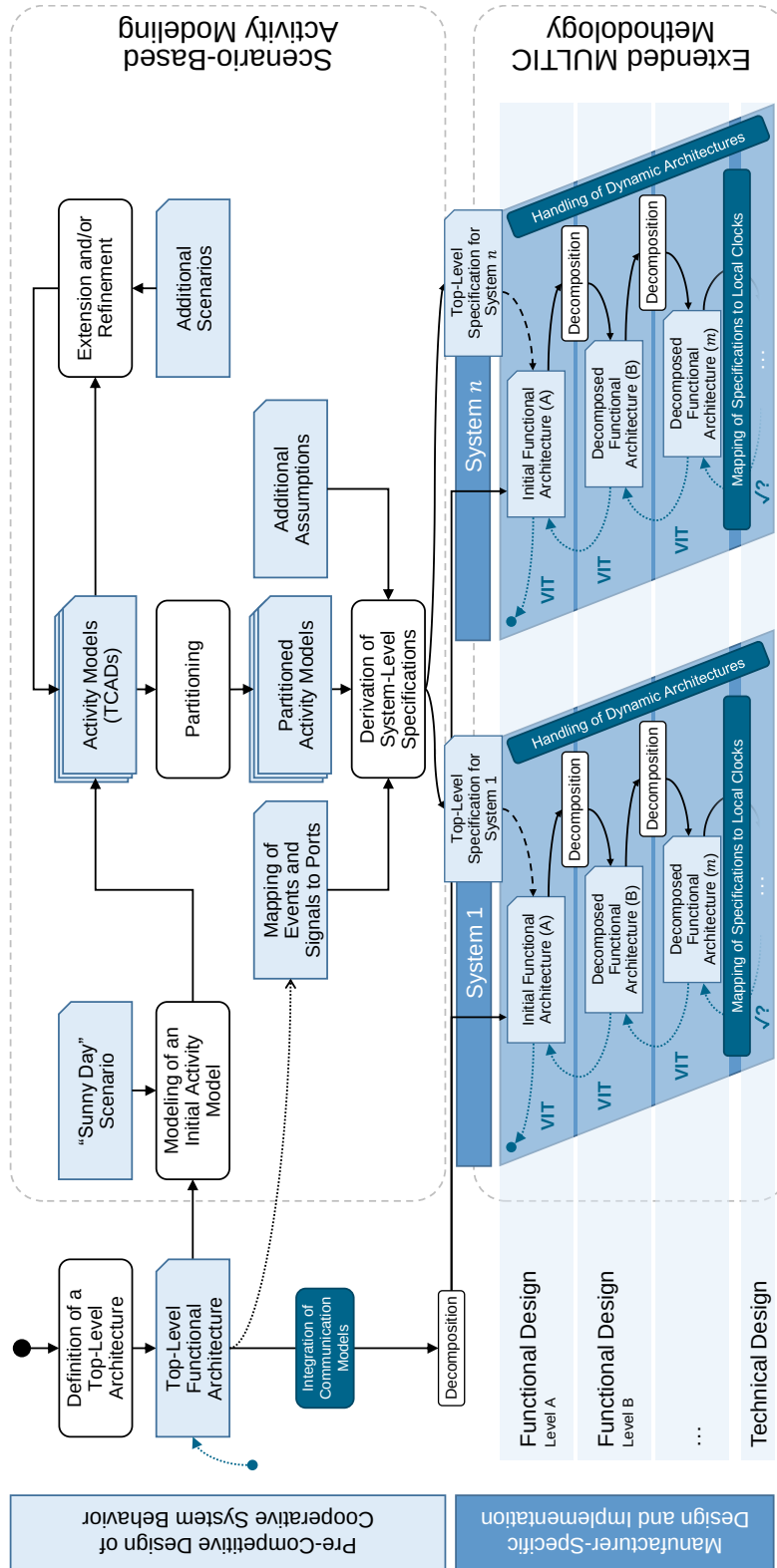


Fig. 3.1: Envisioned design approach

3 Consistent Specification of Top-Level Timing Requirements

annotation of timing constraints and resource limitations that can be derived from early hazard analyses [58], for example, allows simple yet effective specification of real-time behavior. Each action is mapped to a single partition that represents an involved actor. Based on this mapping, all actions are uniquely assigned to one of the components from the top-level architecture. More details about TCADs can be found in Section 3.3.1.

The iterative process of scenario-based activity modeling typically begins with the consideration and detailed analysis of a base scenario. This scenario should be kept as simple as possible and should include a complete run of a planned cooperative maneuver, possibly divided into a number of time phases. It can be based, for example, on “sunny day” assumptions, i.e., no errors should occur in this scenario, and only scenario evolutions that involve successful cooperation should be considered. The initial activity model is then iteratively refined and extended to cover additional scenarios, e.g., by including behavior in case of errors, rejection of cooperation requests, and abortion of maneuver execution. Finally, the resulting activity models, consisting of TCADs, are assumed to cover all scenarios relevant to the design of the cooperative driving function.

The next step is the partitioning of the activity models into a set of (sub-)models for each CSs. The goal of this step is to derive top-level specifications on system level. The design activities include the decomposition of all system-spanning action sequences into local ones by dividing them at system boundaries, i.e., at events, signals, and cross-system control flows. Concurrent action sequences (e.g., resulting from forks/joins) are preserved. The resulting (sub-)activity models focus on actions that the individual CSs must contribute to enable successful interaction on SoCPS level. At the same time, each CS must provide all actions required to fulfill its roles and must comply with all requirements that result from the activity models. The last two relevant steps are the mapping of events and signals to ports from the top-level architecture as well as the formulation of additional assumptions, e.g., about the occurrence of (external) events.

Finally, system-level specifications can be derived from the partitioned activity models. For this purpose, all actions are translated into modes (i.e., internal states) of the individual CSs, which take on the previously defined roles. The system-local action sequences resulting from the partitioning of TCADs, as well as the annotated timing constraints and resource limitations, can then be automatically translated into MTSL specifications, which form the basis for all design activities on (sub-)system level.

3.2.2 Manufacturer-Specific Design and Implementation

The second phase *Manufacturer-Specific Design and Implementation* builds upon the results of the first phase and realizes the manufacturer-specific design and subsequent implementation of the CSs involved in cooperative processes. The first step is the definition (or reuse) of a functional architecture for each CS. The previously derived real-time specifications can then be annotated to the initial component models on system level.

In the following steps, the extended MULTIC methodology is used to further decompose the functional architectures, analogous to the design approach presented in Section 2.3.2. The incremental design of the CSs involves a series of iterative decompositions of their functional architectures with parallel refinements of the annotated timing specifications.

Through the systematic use of analysis techniques such as VITs, relevant proof obligations such as the verification of refinement relations are fulfilled. This ensures consistency of the resulting lower-level specifications with the original system-level specifications and guarantees compatibility of the behavior of individual CSs on SoCPS level.

The final step is to verify the compliance of the implemented CSs with the refined timing specifications, e.g., using formal verification methods or test procedures. Based on the already verified refinement relations between the specifications at different levels of abstraction, proving compliance with the specifications at the lowest level also proves compliance with the specifications on system and SoCPS level. At this point, the effort for the structured approach and the incremental verification of refinement relations finally pays off, i.e., the correct-by-design nature of the design approach becomes effective.

The application of the extended MULTIC methodology follows a set of design rules that determine the order in which certain MTSL specification patterns should be used. For example, causal and non-causal specification patterns can be used throughout the entire design process. Once introduced, mode-dependent specifications for handling dynamic architectures (see Chapter 4) must be refined and may not be omitted in subsequent design steps. The mapping of specifications to local clocks (see Chapter 6) is done just before the transition to technical system design. The chosen coarse-to-fine approach avoids prematurely restricting the design space and ensures that the specified real-time behavior can be implemented in the final stages of system development.

3.3 Scenario-Based Activity Modeling

The process of scenario-based activity modeling is the heart of the envisioned design approach. The main goal of this design activity is to specify valid behavior of cooperative SoCPSs—with a specific focus on interactions and cooperative maneuvers between the CSs involved—based on a set of relevant scenarios. Such scenarios may, for example, be represented as *logical* scenarios, according to the classification of scenario descriptions introduced by Neurohr et al. [446] based on previous work by Menzel et al. [416]. In this context, logical scenarios are defined as “*model[s] of the time sequence of scenes whose parameters are defined as ranges, starting at a specific point in time*” [446, p. 18035]. Note that we do not take into account probability distributions to characterize parameter ranges and assume that the behavior of the main actors is specified as part of the scenarios.

The general procedure for creating activity models (e.g., as part of the *Modeling of an Initial Activity Model* or the *Extension and/or Refinement* design activities in Figure 3.1) consists of three main steps that must be performed either sequentially or iteratively:

1. Modeling of relevant actions in terms of activity diagrams
2. Specification and annotation of timing and resource constraints
3. Application of VITs on activity level to ensure consistency

Following the approach presented in Figure 3.1, it is particularly important to limit the modeling of activities and actions to those that are relevant for the interaction of the

CSs involved. This means that independent, subsystem-local activities and actions, such as parts of the individual driving tasks, should not be included in the activity models.

3.3.1 Time-Constrained Activity Diagrams

The modeling and specification of cross-system coordination processes in cooperative SoCPSs requires a modeling language that is suitable for capturing relevant system properties such as actions of single systems, joint activities of multiple systems, precedence constraints on the ordering of actions and events as well as duration constraints for activities, actions, and action sequences. In addition, trigger events, communication, and synchronization between individual CSs must be considered. Moreover, the specification of basic resource constraints would be desirable. Based on the literature review conducted in Section 3.1, activity diagrams are selected as the basis for activity modeling as part of this work. The basic concept of activity diagrams is complemented by specification means that are specifically tailored to specify the real-time behavior of cooperative systems.

Time-Constrained Activity Diagrams (TCADs) extend simple time-annotated SysML activity diagrams and are used to specify the timing behavior of a SoCPS in a specific scenario. The concept is inspired by previous work of Jarraia et al. [332] and represents a lightweight alternative to more complex specification means. TCADs provide an intuitive way of specifying timing requirements familiar or easily learned by most engineers. The domain-agnostic nature of activity diagrams and their similarities with simple flowcharts [323] facilitates communication and collaboration in interdisciplinary RE activities, which are very common in heterogeneous organizations, especially when developing SoCPSs. In addition, legacy models can be included by annotating existing activity diagrams with the extensions and constraints presented below. From a tooling perspective, the modeling of activity diagrams is supported by all UML/SysML modeling environments and should therefore be available in most tool landscapes used in research and industrial practice.

To enable proper integration into existing MBSE processes, we introduce the concept of *Time-Constrained Activity Models* (TCAMs) as a higher-level “container” that contains a number of TCADs. TCAMs correspond to (parts of) a top-level design model, e.g., a UML or SysML model describing the behavior of (parts of) a given SoCPS. TCADs, on the other hand, correspond to individual views on the system’s behavior that are part of the top-level design model, i.e., a SysML activity diagram that describes cooperative behavior in a specific scenario, including annotations of timing constraints and resource limitations. In the following, we will define the syntax and semantics of TCADs.

Syntax

The syntactic elements of TCADs can be divided into basic *modeling elements* and *constraints*. In addition, the optional specification of *temperatures* of a subset of modeling elements is supported by adapting certain properties of the graphical notation. Figure 3.2 provides an overview of the most important modeling elements and constraint types.

Modeling Elements		
	Activities	
	Partitions	
	Atomic Actions	
	Call Behavior Actions	
	Events	
	Signals	

Constraint Types	
	Implicit Precedence
	Explicit Precedence
	Duration
	Synchronization
	Action-to-Action Latency
	Uniqueness

Fig. 3.2: TCAD modeling elements and constraint types

Modeling Elements The modeling elements of TCADs are defined as a subset of the modeling elements of SysML activity diagrams [454, Chapter 11]. According to the general understanding, activity diagrams are “*used to model control flow and data flow dependencies among the functions/processes defined within a system*” [330, p. 401]. Such diagrams are “*composed of a set of actions related in a specific order of invocation (or execution) by control flow paths, optionally emphasizing [...] input and [...] output dependencies using data flow paths*” [330, p. 401]. In the context of this work, we will use the following elements as a basis for modeling the behavior of cooperative SoCPSs:

Activities *Activities* are the top-level modeling entities for capturing SoCPS behavior. They “*may be composed of a set of actions coordinated sequentially, concurrently, or a combination of these*” [330, p. 401] and thus represent “*flow[s] of functional behaviors*” [480]. Activities can be used to describe both system-local situation assessment and decision making as well as cross-system coordination processes.

Partitions *Partitions* (i.e., swimlanes) allow to explicitly assign actions and control nodes—as the key elements of behavioral models—to actors. They define responsibilities within the cooperative system by dividing cross-system coordination processes on SoCPS level into a set of interdependent (sub-)processes on CS level.

Actions *Actions*¹ are “*atomic activit[ies], which are primitive executable behavior[s]*” [480]. They are used to model individual process steps of a higher-level activity.

Atomic Actions *Atomic actions* are the most basic functional models and represent “*behavior specification[s that] cannot be further decomposed*” [330, p. 401].

Call Behavior Actions *Call behavior actions* build upon compositions of atomic actions and allow to specify a “*hierarchy among activities using call behavior action nodes, which may reference another activity definition*” [330, p. 401].

¹Note that *actions* and *activities* are different modeling elements in UML and SysML. According to the specifications [454, 455], an *activity* consists of several *actions*, which represent atomic functional units. *Activities* can be included in parent activity diagrams by using *call behavior actions*.

3 Consistent Specification of Top-Level Timing Requirements

Events *Events* represent either the acceptance of a previously sent signal or a time event. They can be used to model reactions to internal and external events.

Signals *Signals* represent the sending of a signal, which can be accepted by *events*. They can be used to model the triggering of events and are a basic tool for modeling interactions, e.g., for synchronizing concurrent processes.

Control Nodes *Control nodes* are “abstract activity node[s] that coordinate [...] the flow of control” [306]. They can be distinguished into the following subtypes:

Initial Nodes *Initial nodes* represent the start of an activity.

Final Nodes *Final nodes* represent the end (of parts) of an activity. They can be divided into *final activity nodes*, which denote “the end of all control flows” [550], and *final flow nodes*, which denote “the end of a single control flow” [550].

Decisions *Decisions* can be used to split one control flow into two alternative control flows. This allows modeling a “choice between two possible [control flow] paths based on the evaluation of a guard condition” [330, p. 401].

Merges *Merges* can be used to merge multiple alternative control flows into a single control flow. A merge node thus “specifies a point from where different incoming control [flow] paths have to follow the same path” [330, p. 401].

Forks *Forks* allow to split one control flow into multiple concurrent flows. A fork node represents the “beginning of multiple parallel control threads” [330, p. 401].

Joins *Joins* can be used to merge two or more concurrent flows into a single one, forcing “multiple parallel control threads to synchronize and rejoin” [330, p. 401].

Control Flows *Control flows* define the “flow of functional behaviors” [480]. These flows determine the order in which actions are executed and connect actions and control nodes to form a network that describes cross-system and system-local processes.

Note that less common modeling elements of activity diagrams such as *expansion regions*, *exception handlers*, and *interruptible activity regions* are currently not supported. Since it is assumed that they are only required to address special use cases, they are beyond the scope of this work. However, they can easily be included in future versions of the TCAD definition and represent a possible starting point for follow-up activities.

Constraints In addition to the modeling elements of activity diagrams, the SysML specification introduces, among other things, *constraint blocks* that can be used “to specify a network of constraints that represent mathematical expressions [...], which constrain the physical properties of a system” [454, p. 120]. In principle, these constraints could also be used to characterize the timing behavior of activity diagrams. However, the existing specification means are not sufficient to cover real-time behavior [332] and therefore need to be enhanced. To allow the specification of timing and resource constraints specifically tailored to the modeling of SoCPSs and interactions between CSs, we define *precedence*, *duration*, *synchronization*, *action-to-action latency*, and *uniqueness* constraints:

Precedence *Precedence constraints* restrict the order in which actions can be executed. In the following, we distinguish between implicit and explicit precedence constraints:

Implicit Precedence *Implicit precedence constraints* are defined by control flows. They restrict the possible order of execution so that the action from which the control flow originates must be completed before the following action starts.

Explicit Precedence *Explicit precedence constraints* can be used to handle concurrency across activity or subsystem boundaries. They are indicated by a dotted arrow between the actions involved, and can be simplified by displaying only the two ends of the arrow with a Roman number as a shared index.

Duration Each action may be annotated with a *duration constraint* that specifies a time interval in which the action has to be (fully) executed. For hierarchical actions (i.e., call behavior actions that call other activities), the duration constraint is derived from those of their children (i.e., by adding up the durations of sequential actions or by selecting the maximum duration in case of parallel actions). By default, *control flows* are timeless. They may also be annotated with duration constraints, implicitly defining a corresponding “virtual” action that represents the transmission process.

Synchronization Parallel actions that are required to start simultaneously can be specified using *synchronization constraints*. Such constraints are indicated with an asterisk (“*”) and an index for referencing. The specified time value is considered a synchronization bound, so the distance between the start times of the two actions must not be greater than δ time units.

Action-to-Action Latency *Action-to-action latency constraints* are indicated by a dashed line annotated with a duration constraint and limit the period of time between the start of the first action and the end of the last action. Note that events can only be used as first actions and signals can only be used as last actions.

Uniqueness *Uniqueness constraints* can be used to indicate that an action can only be executed once at a time, i.e., if one instance is already active, the execution of a second instance will be delayed until the first one has finished. Uniqueness constraints are modeled by annotating the constrained action with “ ≤ 1 ”. Note that alternative strategies for enforcing uniqueness constraints, such as discarding activating events, could be used, but are beyond the scope of this work.

All constraint types can be instantiated with concrete parameters and annotated to the basic modeling elements of a previously modeled activity diagram. In addition, *temperatures of actions* and *control flows* can be specified to enforce or dismiss progress.

Temperatures The *temperature of actions* and *control flows* can be expressed by adjusting the line style of the graphical modeling elements. By default, all basic modeling elements are *hot*, i.e., they are drawn with solid lines and must be executed as specified. In contrast, individual modeling elements can be marked as *cold* by using dashed lines. Such elements can be executed, but do not represent mandatory behavior.

Semantics

As a prerequisite for the formal definition of the partitioning approach as well as the derivation of system-level specifications, we formally define TCADs as follows:

Definition 5 (Time-Constrained Activity Diagram). A *Time-Constrained Activity Diagram* is a tuple $\text{TCAD} = (act, \mathcal{A}, \mathcal{N}, \mathcal{C}, \mathcal{P}, \Lambda, \Xi, \Phi)$ that describes (parts of) the behavior of a given system or SoCPS. It has the following properties:

- $act \in \mathcal{A}$ is the activity that is modeled by the TCAD.
- $\mathcal{A} = \mathcal{A}_{atom} \cup \mathcal{A}_{call} \cup \mathcal{A}_{event} \cup \mathcal{A}_{signal}$ is a non-empty set of *actions*:
 - $\mathcal{A}_{atom}$ is a set of (atomic) *actions*. Each action $a \in \mathcal{A}_{atom}$ represents a behavior specification that cannot be further decomposed.
 - $\mathcal{A}_{call}$ is a set of *call behavior actions*. Call behavior actions reference an activity $act(a) \in \mathcal{A}$, which can be used to specify complex action behaviors.
 - $\mathcal{A}_{event} = \mathcal{A}_{event}^{accept} \cup \mathcal{A}_{event}^{time}$ is a set of *events* that is composed of *accept events* $\mathcal{A}_{event}^{accept}$ and *time events* $\mathcal{A}_{event}^{time}$. Thus, events represent either the acceptance of a signal or a time event. The set of *activating events* is defined as

$$\mathcal{A}_{event}^{\uparrow} = \{e \in \mathcal{A}_{event} \mid \mathcal{N}_{init} = \emptyset \wedge \nexists s \in \mathcal{A}_{signal} : \Lambda(e) = \Lambda(s)\} . \quad (3.1)$$

- $\mathcal{A}_{signal} = \mathcal{A}_{signal}^{send}$ is a set of *signals*. Signals represent the sending of a signal, which can be accepted by events. The set of *terminating signals* is defined as

$$\mathcal{A}_{signal}^{\downarrow} = \{s \in \mathcal{A}_{signal} \mid \mathcal{N}_{final} = \emptyset \wedge \nexists e \in \mathcal{A}_{event} : \Lambda(s) = \Lambda(e)\} . \quad (3.2)$$

- $\mathcal{N} = \mathcal{N}_{init} \cup \mathcal{N}_{final} \cup \mathcal{D} \cup \mathcal{M} \cup \mathcal{F} \cup \mathcal{J}$ is a non-empty set of *control nodes*:
 - $\mathcal{N}_{init}$ is a set of *initial nodes*.
 - $\mathcal{N}_{final} = \mathcal{N}_{\odot} \cup \mathcal{N}_{\otimes}$ is a set of *final nodes* that consists of *final activity nodes* $\mathcal{N}_{\odot}$ and *final flow nodes* $\mathcal{N}_{\otimes}$.
 - $\mathcal{D}$ is a set of (guarded, non-probabilistic) *decision nodes*.
 - $\mathcal{M}$ is a set of *merge nodes*.
 - $\mathcal{F}$ is a set of *fork nodes*.
 - $\mathcal{J}$ is a set of *join nodes*.
- $\mathcal{C} \subseteq (\mathcal{A} \cup \mathcal{N}) \times (\mathcal{A} \cup \mathcal{N})$ is a non-empty set of *control flows*. Based on this set, we define two functions $in, out : \mathcal{A} \cup \mathcal{N} \rightarrow 2^{\mathcal{C}}$, which return all *incoming*, or *outgoing*, control flows of a given action or control node $an \in \mathcal{A} \cup \mathcal{N}$, respectively:
 - $in : an \mapsto C_{in}$ with $C_{in} = \{(an_1, an_2) \in \mathcal{C} \mid an = an_1\}$
 - $out : an \mapsto C_{out}$ with $C_{out} = \{(an_1, an_2) \in \mathcal{C} \mid an = an_2\}$

3.3 Scenario-Based Activity Modeling

Depending on its type, each action or control node $an \in \mathcal{A} \cup \mathcal{N}$ must have a certain number of incoming and outgoing control flows, i.e.,

$$|C_{in}| = \begin{cases} 1 & \text{if } an \in (\mathcal{A} \setminus \mathcal{A}_{event}) \cup \mathcal{N}_{final} \cup \mathcal{D} \cup \mathcal{F} \\ n \text{ for some } n \geq 2 & \text{if } an \in \mathcal{M} \cup \mathcal{J} \end{cases} \quad (3.3)$$

and

$$|C_{out}| = \begin{cases} 1 & \text{if } an \in (\mathcal{A} \setminus \mathcal{A}_{signal}) \cup \mathcal{N}_{init} \cup \mathcal{M} \cup \mathcal{J} \\ 2 & \text{if } an \in \mathcal{D} \\ m \text{ for some } m \geq 2 & \text{if } an \in \mathcal{F} \end{cases}, \quad (3.4)$$

to be considered *valid* in the sense of this definition. For practical reasons, we refer to $(an_1, an_2) \in \mathcal{C}$ as a *direct dependency* $an_1 \rightarrow an_2$. In addition, two actions or control nodes $an_1, an_n \in \mathcal{A} \cup \mathcal{N}$ have an *indirect dependency*, denoted as $an_1 \rightsquigarrow an_n$, if a subset $C_{\rightsquigarrow} \subseteq \mathcal{C}$ with $C_{\rightsquigarrow} = \{\rightarrow_1, \dots, \rightarrow_{n-1}\}$, $n \in \mathbb{N}$, and $an_1 \rightarrow_1 \dots \rightarrow_{n-1} an_n$ exists that connects an_1 and an_n via a series of direct dependencies.

An *action sequence* $\langle a_1 \rightarrow \dots \rightarrow a_n \rangle$ is a tuple $as = (A, N, C)$, where $A \subseteq \mathcal{A}$ is a subset of actions, $N \subseteq \mathcal{N}$ is a subset of control nodes, and $C \subseteq \mathcal{C}$ is a subset of control flows. The components of as must fulfill the following conditions:

1. $a_1 \in A \wedge \nexists (an, an') \in C : a_1 = an'$
2. $a_n \in A \wedge \nexists (an, an') \in C : a_n = an$
3. $\forall an \in (\mathcal{A} \cup \mathcal{N}) \setminus \{a_1\} : a_1 \rightsquigarrow an$
4. $\forall an \in (\mathcal{A} \cup \mathcal{N}) \setminus \{a_n\} \exists! (an, an') \in C : an' \in \mathcal{A} \cup \mathcal{N}$

Finally, we define $as^* : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{AS}$, $(a_1, a_n) \mapsto \{as_1, \dots, as_m\}$ with $\mathcal{AS} \subseteq 2^{\mathcal{A}} \times 2^{\mathcal{N}} \times 2^{\mathcal{C}}$ and $n, m \in \mathbb{N}$, which returns all *possible action sequences* from a_1 to a_n . Note that all elements of $\mathcal{AS}$ must be *valid*, i.e., satisfy conditions 1 to 4.

- $\mathcal{P}$ is a set of *partitions*. Each partition $p \in \mathcal{P}$ represents an actor's top-level component $c_{cs} \in \mathcal{C}$ and is associated with a set of *roles*, i.e., there is a mapping $roles : \mathcal{P} \rightarrow 2^{\mathcal{R}}$, $p \mapsto R$ that defines the correspondence between $\mathcal{P}$ and $\mathcal{R}$ so that

$$\forall p \in \mathcal{P} \exists! R \subseteq \mathcal{R} : roles(p) = R. \quad (3.5)$$

Note that the set of *relevant roles* $\mathcal{R}$ is defined as the union of all sets of *component roles* $R_c = roles_c(c)$ with $roles_c : \mathcal{C} \rightarrow 2^{\mathcal{R}}$, $c \mapsto \{r_1, \dots, r_m\}$ and $m \in \mathbb{N}$, i.e.,

$$\mathcal{R} = \bigcup_{i=1}^n R_{c_i} \quad (3.6)$$

based on $\mathcal{C} = \{c_1, \dots, c_n\}$. In addition, we define a function $actor : \mathcal{P} \rightarrow \mathcal{C}$, $p \mapsto c_{cs}$ that returns the actor's component c_{cs} that takes on the given roles R .

3 Consistent Specification of Top-Level Timing Requirements

- $\Lambda : \mathcal{P} \cup \mathcal{A} \rightarrow \mathcal{L}$ is a *labeling function* that assigns a label $l \in \mathcal{L}$ to each partition $p \in \mathcal{P}$ and each action $a \in \mathcal{A}$. Note that labels of partitions must be unique, i.e.,

$$\forall p \in \mathcal{P} \nexists p' \in \mathcal{P} : \Lambda(p) = \Lambda(p'). \quad (3.7)$$

- $\Xi : \mathcal{A} \cup \mathcal{N} \cup \mathcal{C} \rightarrow 2^{\mathcal{P}}$ is a *mapping function* that maps all actions, control nodes, and control flows to partitions. It assigns each $an \in \mathcal{A} \cup \mathcal{N}$ to a partition $p \in \mathcal{P}$, i.e.,

$$\forall an \in \mathcal{A} \cup \mathcal{N} \exists! p \in \mathcal{P} : \Xi(an) = p. \quad (3.8)$$

Each control flow $\rightarrow \in \mathcal{C}$ is assigned to either one or two partitions, i.e.,

$$\forall \rightarrow \in \mathcal{C} : (\exists p \in \mathcal{P} : \Xi(\rightarrow) = p) \vee (\exists p_1, p_2 \in \mathcal{P} : \Xi(\rightarrow) = \{p_1, p_2\}). \quad (3.9)$$

Based on this mapping, we define the set of *cross-system control flows* $C_{\leftrightarrow} \subseteq \mathcal{C}$ as the set of control flows that are assigned to two partitions, i.e.,

$$C_{\leftrightarrow} = \{\rightarrow \in \mathcal{C} \mid |\Xi(\rightarrow)| = 2\}. \quad (3.10)$$

- $\Phi = \Phi_{pred} \cup \Phi_{dur} \cup \Phi_{lat} \cup \Phi_{sync} \cup \Phi_{uni}$ is a set of *constraints*:
 - $\Phi_{pred} = \Phi_{impl} \cup \Phi_{expl}$ is a set of *precedence constraints*. Each precedence constraint is a tuple $\phi_{pred} = (an_1, an_2)$ with $an_1, an_2 \in \mathcal{A} \cup \mathcal{N}$ that restricts the execution order of an_1 and an_2 . The constraints are defined as follows:
 - * $\Phi_{impl} \subseteq (\mathcal{A} \cup \mathcal{N}) \times (\mathcal{A} \cup \mathcal{N})$ is a set of *implicit precedence constraints*. Implicit precedence constraints are derived from the set of control flows:

$$\Phi_{impl} = \{(an_1, an_2) \mid (an_1, an_2) \in \mathcal{C}\} \quad (3.11)$$

- * $\Phi_{expl} \subseteq (\mathcal{A} \cup \mathcal{N}) \times (\mathcal{A} \cup \mathcal{N})$ is a set of *explicit precedence constraints* that are explicitly defined and may not be directly related to control flows. Note that all $\phi_{expl} \in \Phi_{expl}$ must be consistent with any $\phi_{impl} \in \Phi_{impl}$, i.e.,

$$\forall (an_1, an_2) \in \Phi_{expl} : an_2 \not\prec an_1. \quad (3.12)$$

- $\Phi_{dur} \subseteq (\mathcal{A} \cup \mathcal{C}) \times \mathbb{T} \times \mathbb{T}$ is a set of *duration constraints*. Each duration constraint is a tuple $\phi_{dur} = (ac, lb, ub)$ with $ac \in \mathcal{A} \cup \mathcal{C}$ and $lb, ub \in \mathbb{T} = \mathbb{R}_0^+$ that limits the time interval in which the execution of an action or control flow must be completed. Note that there may only be one duration constraint per ac , i.e.,

$$\forall (ac, lb, ub) \in \Phi_{dur} \nexists (ac', lb', ub') \in \Phi_{dur} : ac = ac'. \quad (3.13)$$

For convenience, we define a *duration function* $dur : \mathcal{A} \cup \mathcal{C} \rightarrow \mathbb{T} \times \mathbb{T}$, $ac \mapsto (lb, ub)$ that returns the time values of the duration constraint assigned to ac :

$$dur(ac) = \begin{cases} (lb, ub) & \text{if } \exists (ac, lb, ub) \in \Phi_{dur} \\ (0, \infty) & \text{if } \nexists (ac, lb, ub) \in \Phi_{dur} \wedge ac \in \mathcal{A} \\ (0, 0) & \text{otherwise} \end{cases} \quad (3.14)$$

3.3 Scenario-Based Activity Modeling

- $\Phi_{lat} \subseteq \mathcal{AS} \times \mathbb{T} \times \mathbb{T}$ is a set of *action-to-action latency constraints*. Each of these constraints is a tuple $\phi_{lat} = (as^*(a_1, a_n), lb, ub)$ with $as^*(a_1, a_n) \subseteq \mathcal{AS}$, $n \in \mathbb{N}$, and $lb, ub \in \mathbb{T}$ that limits the execution times of all action sequences $as^*(a_1, a_n)$. As with duration constraints, we allow only one action-to-action latency constraint per set of possible action sequences $as^*(a_1, a_n)$, i.e.,

$$\forall (as^*, lb, ub) \in \Phi_{lat} \nexists (as'^*, lb', ub') \in \Phi_{lat} : as^* = as'^*. \quad (3.15)$$

In addition, we restrict Φ_{lat} to *locally observable* action sequences, i.e.,

$$\forall (as^*(a_1, a_n), lb, ub) \in \Phi_{lat} : \Xi(a_1) = \Xi(a_n). \quad (3.16)$$

Again, we define a convenience *latency function* $lat : \mathcal{A} \times \mathcal{A} \rightarrow \mathbb{T} \times \mathbb{T}$, $(a_1, a_n) \mapsto (lb, ub)$ that returns the time bounds assigned to actions a_1 and a_n :

$$lat(a_1, a_n) = \begin{cases} (lb, ub) & \text{if } \exists (as^*(a_1, a_n), lb, ub) \in \Phi_{lat} \\ (0, \infty) & \text{otherwise} \end{cases} \quad (3.17)$$

- $\Phi_{sync} \subseteq 2^{\mathcal{A}} \times \mathbb{T}$ is a set of *synchronization constraints*. Each synchronization constraint is a tuple $\phi_{sync} = (A, \delta)$ with $A \subseteq \mathcal{A}$ and $\delta \in \mathbb{T}$ that defines a time bound δ between the start time points of actions A . Based on Φ_{sync} , we define a *synchronization function* $sync : \mathcal{A} \rightarrow 2^{\mathcal{A}} \times \mathbb{T}$ that returns the set of synchronized actions A and the synchronization time bound δ :

$$sync(a) = \begin{cases} (A, \delta) & \exists (A, \delta) \in \Phi_{sync} : a \in A \\ (\emptyset, \infty) & \text{otherwise} \end{cases} \quad (3.18)$$

- $\Phi_{uni} \subseteq \mathcal{A} \times \mathbb{N}$ is a set of *uniqueness constraints*. Each uniqueness constraint is a tuple $\phi_{uni} = (a, n_a)$ with $a \in \mathcal{A}$ and $n_a \in \mathbb{N}$ that limits the number of instances that can be executed simultaneously. The *uniqueness function* $uni : \mathcal{A} \rightarrow \{0, 1\}$ returns whether more than one instance can be active:

$$uni(a) = \begin{cases} 1 & \text{if } \exists (a, n_a) \in \Phi_{uni} \\ 0 & \text{otherwise} \end{cases} \quad (3.19)$$

Finally, we define $A_{cold} \subseteq \mathcal{A}$ as the set of *cold* actions, and $C_{cold} \subseteq \mathcal{C}$ as the set of *cold* control flows, respectively. Note that the two sets A_{cold} and C_{cold} are empty by default, i.e., $A_{cold} = \emptyset$ and $C_{cold} = \emptyset$ apply. $\diamond$

Each TCAD can be annotated with an optional *activity duration constraint* $\phi_{dur}^{act} \in \Phi_{dur}^{act} \subseteq \mathcal{A} \times \mathbb{T} \times \mathbb{T}$ that is not part of the set of TCAD constraints, i.e., $\Phi \cap \Phi_{dur}^{act} = \emptyset$, but can be used to indicate temporal constraints on activity level. In addition, we denote the set of TCADs of a given TCAM as $\mathbb{D}_{TCAM}$. In order to reason about control flow dependencies

3 Consistent Specification of Top-Level Timing Requirements

between any two TCADs, we introduce a *call* relation between $\text{TCAD}, \text{TCAD}' \in \mathbb{D}_{\text{TCAD}}$, written $\text{TCAD} \rightsquigarrow \text{TCAD}'$, which indicates that TCAD directly calls TCAD', i.e.,

$$\exists s \in \mathcal{A}_{\text{signal}}^\downarrow \exists e \in \mathcal{A}_{\text{event}}^\uparrow : \Lambda(s) = \Lambda'(e) \quad (3.20)$$

with $\text{TCAD} = (\text{act}, \mathcal{A}, \mathcal{N}, \mathcal{C}, \mathcal{P}, \Lambda, \Xi, \Phi)$ and $\text{TCAD}' = (\text{act}', \mathcal{A}', \mathcal{N}', \mathcal{C}', \mathcal{P}', \Lambda', \Xi', \Phi')$.

The execution semantics of TCADs is defined based on the standard semantics of UML/SysML activity diagrams. Therefore, we rely on the concepts of *tokens* and *firing rules* that were informally introduced in the UML specification [455, Chapter 15]. Later, these concepts were formalized by Jarraya and Debbabi [330]. The authors considered a token as a “control locus that represents the current position of the control execution. For instance, at the starting of the diagram execution, the control locus (i.e., token) is placed on the initial node. Tokens may be duplicated (fork nodes) or merged (join nodes)” [330, p. 402]. In addition, firing rules “describe how tokens can flow within an activity diagram for each type of node. These rules are based on the conditions that tokens in the predecessor nodes are offered on their outgoing edges [...] and the ability of successor nodes to accept these tokens” [330, p. 402]. Jarraya et al. have also introduced a mathematical calculus called *Activity Calculus* (AC) [330, 331] to formally define the syntax and semantics of SysML activity diagrams. In the context of this thesis, we limit ourselves to a semiformal description of TCAD semantics and refer interested readers to the extensive preliminary work in this area [329, 330, 331, 455, 466, 467, 468]. To define the semantics of constraints, we complement the concepts of tokens and firing rules with a definition of *activation conditions* that characterize the circumstances under which the input tokens are consumed, initiating the execution of TCAD elements.

As starting points for the execution of activities, each *initial node* $n_{\text{init}} \in \mathcal{N}_{\text{init}}$ and each *activating event* $a \in \mathcal{A}_{\text{event}}^\uparrow$ produce a single token at the exact point in time when the activity starts. If more than one initial node exist, i.e., $|\mathcal{N}_{\text{init}}| > 1$, “invoking the [activity] starts multiple concurrent control flows, one for each [initial node]” [455, p. 387].

Tokens are propagated along the *control flows* from $\mathcal{C}$. This propagation follows the *implicit* and *explicit precedence constraints* $\phi_{\text{pred}} \in \Phi_{\text{pred}}$, i.e., it takes place along the direct and indirect dependencies between actions and control nodes. If no *duration constraint* $\phi_{\text{dur}} \in \Phi_{\text{dur}}$ is annotated to a given *control flow* $\rightarrow \in \mathcal{C}$, i.e., $\text{dur}(\rightarrow) = (0, 0)$, tokens are propagated in zero time, i.e., $t_{\text{end}}(\rightarrow) - t_{\text{start}}(\rightarrow) = 0$. Otherwise, if such a constraint exists, i.e., $\text{dur}(\rightarrow) = (lb, ub)$ with $lb, ub \in \mathbb{T}$ and $ub > 0$, the semantics of $\rightarrow = (an_1, an_2)$ is defined by a *virtual action* $a_{\text{virt}}(\rightarrow) \in A_{\text{virt}}$ that is executed within the specified time bounds, i.e., $t_{\text{end}}(\rightarrow) - t_{\text{start}}(\rightarrow) \in [lb, ub]$, as well as a set of *virtual control flows* $\rightarrow_{\text{virt}}(\rightarrow) = \{\rightarrow_1^{\text{virt}}, \rightarrow_2^{\text{virt}}\}$ with $\rightarrow_1^{\text{virt}} = (an_1, a_{\text{virt}}(\rightarrow))$ and $\rightarrow_2^{\text{virt}} = (a_{\text{virt}}(\rightarrow), an_2)$. We denote the set of virtual actions derived from the set of duration-constrained control flows of a given TCAD by A_{virt} with $A_{\text{virt}} \cap \mathcal{A} = \emptyset$, and the set of virtual control flows obtained in the same way by C_{virt} with $C_{\text{virt}} \cap \mathcal{C} = \emptyset$, respectively. Trivially, the semantics of *cold* control flows is less prescriptive. In this case, a control flow $\rightarrow \in C_{\text{cold}}$ may or may not be executed, i.e., tokens may sporadically be removed from cold control flows.

Actions $a \in \mathcal{A} \setminus (\mathcal{A}_{\text{event}} \cup \mathcal{A}_{\text{signal}})$ consume tokens from their incoming control flows *in*(a) according to *activation conditions* and produce tokens on their outgoing control

flows $out(a)$ according to *firing rules*. In this context, an action a consumes its input tokens and starts its execution if the following *activation conditions* are met:

1. All incoming control flows $in(a)$ must be assigned a token.
2. All *explicit precedence constraints* $\phi_{expl} \in \Phi_{expl}$ must be fulfilled.
3. If a *synchronization constraint* $\phi_{sync} \in \Phi_{sync}$ has been defined that requires the simultaneous start of the current action a and a set of actions A , i.e., $sync(a) = (A, \delta)$ holds, all synchronized actions must have tokens on their incoming control flows and must be able to start their execution within the specified time bound δ .
4. There are no *uniqueness constraint* $\phi_{uni} \in \Phi_{uni}$ for action a , i.e., $uni(a) = 0$ holds, or the action a is currently not being executed, if $uni(a) = 1$ applies.

In the event of a violation of one or more constraints, predefined rules are applied to resolve conflicts and inconsistent timing behavior. If conditions 1 and 2 are not (yet) satisfied, the execution is delayed until all incoming control flows have tokens and all explicit precedence constraints are met. If condition 3 is not (yet) fulfilled, the activation is deferred as long as possible to comply with the *synchronization constraint* and allow a synchronized start within the specified time bound. If condition 4 is violated, the execution of the current action instance is delayed as long as possible without violating any other constraints, in order to find a feasible schedule for serving the action's activation. If one or more of the annotated constraints are violated, the parent activity is aborted. Note that *implicit precedence constraints* $\phi_{impl} \in \Phi_{impl}$ cannot be violated by definition of TCADs, as the order of the actions' execution is derived from the set of control flows, i.e., all execution sequences implicitly satisfy all implicit precedence constraints.

After its activation, each action $a \in \mathcal{A} \setminus (\mathcal{A}_{event} \cup \mathcal{A}_{signal})$ is executed within the specified time bounds $[lb, ub]$ defined by the action's *duration constraint* $\phi_{dur} \in \Phi_{dur}$ with $dur(a) = (lb, ub)$ and $ub \neq \infty$, i.e., $t_{end}(a) - t_{start}(a) \in [lb, ub]$, or in zero time if no such constraint exists, i.e., $dur(a) = (0, \infty)$ applies. The possible points in time at which the execution may be terminated are specified by means of the following *firing rules*:

1. If a *duration constraint* $\phi_{dur} \in \Phi_{dur}$ is annotated to action a , i.e., $dur(a) = (lb, ub)$ with $lb, ub \in \mathbb{T}$ and $ub \neq \infty$, an output token is created for each outgoing control flow $out(a)$ so that $t_{end}(a) - t_{start}(a) \in [lb, ub]$ holds. Otherwise, if no such constraint exists, i.e., $dur(a) = (0, \infty)$, output tokens are generated in zero time.
2. If one or more *action-to-action latency constraints* $\phi_{lat} \in \Phi_{lat}$ exist that limit the execution time of action sequences $as = \langle a_1 \rightarrow \dots \rightarrow a \rangle$, i.e., $lat(a_1, a) = (lb', ub')$ with $lb', ub' \in \mathbb{T}$ and $ub' \neq \infty$, condition $t_{end}(a) - t_{start}(a_1) \in [lb', ub']$ must apply.

Again, if no valid termination time exists that satisfies all constraints, the parent activity is aborted. Note that *event actions* $e \in \mathcal{A}_{event}$ and *signal actions* $s \in \mathcal{A}_{signal}$ are timeless, i.e., $t_{end}(e) - t_{start}(e) = 0$, and $t_{end}(s) - t_{start}(s) = 0$ applies, respectively. Again, the semantics of *cold* actions is less prescriptive than the semantics of *hot* actions. Instances of

3 Consistent Specification of Top-Level Timing Requirements

this type of action may terminate without generating output tokens, resulting in violations of the annotated duration and action-to-action latency constraints.

The concepts of activation conditions and firing rules are also applied to define the semantics of control nodes. All control nodes $n \in \mathcal{N} \setminus (\mathcal{N}_{init} \cup \mathcal{N}_{final})$ are timeless by default, i.e., they produce output tokens on their outgoing control flows $out(n)$ as they consume input tokens on their incoming control flows $in(n)$. *Decisions* $d \in \mathcal{D}$ consume every token on their single incoming control flow and generate exactly one output token on one of the outgoing control flows. *Merges* $m \in \mathcal{M}$ take any input token and generate exactly one output token on their outgoing control flow. *Forks* $f \in \mathcal{F}$ and *joins* $j \in \mathcal{F}$ are activated if each incoming control flow is provided with a token. According to their firing rules, they immediately generate a token on each of their outgoing control flows. Note that fork nodes have exactly one incoming and more than two outgoing control flows, and merge nodes more than two incoming and exactly one outgoing control flow, respectively. *Final nodes* $n_{final} \in \mathcal{N}_{final}$ and *terminating signals* $a \in \mathcal{A}_{signal}^\downarrow$ are also timeless, but consume input tokens in any case and without generating new ones. While the consumption of a token by a *final flow node* $n_{final} \in \mathcal{N}_\otimes$ or a terminating signal $s \in \mathcal{A}_{signal}^\downarrow$ terminates the associated control flow, the consumption of a token by a *final activity node* $n_{final} \in \mathcal{N}_\odot$ leads to the immediate termination of TCAD execution.

In addition to the semantics of TCADs, we introduce a notion of *temporal phases* to reason about the behavior of (parts of) the system *before* and *after* the execution of an activity *act*, or TCAD, respectively. For this purpose, we define three types of phases:

1. An *activity phase* $\mathbf{p}_{act} \in \mathbf{P}$ begins with the start of *act* at time $t_{start}(act)$ and ends with the end of the activity at time $t_{end}(act)$, i.e., $\mathbf{p}_{act}$ is valid at all times $t \in [t_{start}(act), t_{end}(act)]$.
2. A *before phase* $\mathbf{p}_{act}^\triangleleft \in \mathbf{P}$ begins at time $t_{lb}^\triangleleft = t_{start}(act) - t_\delta$ for some $t_\delta \in \mathbb{T}$ with $t_\delta \leq t_{start}(act)$ and ends right before the start of *act* at time $t_{start}(act)$, i.e., $\mathbf{p}_{act}^\triangleleft$ is valid at all times $t \in [t_{lb}^\triangleleft, t_{start}(act))$. $\mathbf{p}_{act}^\triangleleft$ can also be referred to as “before(*act*)”.
3. An *after phase* $\mathbf{p}_{act}^\triangleright \in \mathbf{P}$ begins right after the end of *act* at time $t_{end}(act)$ and is valid until $t_{ub}^\triangleright = t_{end}(act) + t_\delta$ for some $t_\delta \in \mathbb{T}$, i.e., $\mathbf{p}_{act}^\triangleright$ is valid at all times $t \in (t_{end}(act), t_{ub}^\triangleright]$. $\mathbf{p}_{act}^\triangleright$ can also be referred to as “after(*act*)”.

Similar to activity phases, we define *instant time phases* $\mathbf{p}_{e/s}^\nabla \in \mathbf{P}$ for considering events $e \in \mathcal{A}_{event}$ and signals $s \in \mathcal{A}_{signal}$. Instant time phases have a length of $t_{end}(e) - t_{start}(e) = 0$, or $t_{end}(s) - t_{start}(s) = 0$, i.e., they are only valid in a single time point $t(e)$, or $t(s)$, respectively. Note that instant time phases $\mathbf{p}_{e/s}^\nabla$ can be also referred to as “on(*e/s*)”. The corresponding *before* and *after* phases $\mathbf{p}_{e/s}^\triangleleft, \mathbf{p}_{e/s}^\triangleright \in \mathbf{P}$ are defined analogously to those for activity phases. Note that the *before* phase $\mathbf{p}_{act}^\triangleleft$ of an activity *act* that is initiated by the occurrence of an activating event $e \in \mathcal{A}_{event}^\uparrow$ equals the *before* phase of *e*, i.e., $\mathbf{p}_{act}^\triangleleft = \mathbf{p}_e^\triangleleft$. In the other direction, the *after* phase $\mathbf{p}_{act}^\triangleright$ of an activity *act* that ends with the occurrence of a terminating signal $s \in \mathcal{A}_{signal}^\downarrow$ equals the *after* phase of *s*, i.e., $\mathbf{p}_{act}^\triangleright = \mathbf{p}_s^\triangleright$.

To provide a basis for using TCADs as a tool for specifying valid timing behavior of cooperative SoCPSs, we define *runs* of TCADs. In the context of this work, a *run* π_{TCAD}

is defined as a valid TCAD execution that starts with the generation of a set of initial tokens at *initial nodes* $n_{init} \in \mathcal{N}_{init}$ and *activating events* $a \in \mathcal{A}_{event}^\uparrow$ and ends with the consumption of either (1) one token at a *final activity node* $n_{final} \in \mathcal{N}_\odot$ or a *terminating signal* $a \in \mathcal{A}_{signal}^\downarrow$ or (2) all tokens at *final flow nodes* $n_{final} \in \mathcal{N}_\otimes$. We denote the set of possible runs of a given TCAD as Π_{TCAD} . The *language* of a given TCAD, denoted as L_{TCAD} , is defined as the set of runs that comply with the annotated constraints, i.e.,

$$L_{TCAD} = \{\pi_{TCAD} \in \Pi_{TCAD} \mid \pi_{TCAD} \models \Phi\} . \quad (3.21)$$

3.3.2 Application Example

In the following, we assume that we are part of a consortium of manufacturers and suppliers that aims to develop a cross-manufacturer interoperable CLTA. Recall that such a system supports the safe and efficient negotiation and execution of cooperative left-turn maneuvers at urban intersections, as illustrated in Section 1.2.1. We have already identified that two CHAVs, namely CHAV 1 in the role of the LEFT-TURNING VEHICLE and CHAV 2 in the role of the STRAIGHT-AHEAD VEHICLE, are the main actors. For practical reasons, however, both vehicles should be able to take on both roles. The two vehicles are supported by an intelligent TMS in the role of the TRAFFIC MANAGEMENT SYSTEM. All actors communicate via a shared V2X communication channel.

The top-level functional architecture of the CLTA that results from the *Definition of a Top-Level Architecture* design activity (see Figure 3.1) is shown in Figure 3.3. The CLTA consists of four subsystems, namely *CHAV 1*, *CHAV 2*, *V2X*, and *TMS*. The three subsystems that represent the main actors each have complex input ports *CommIn* and complex output ports *CommOut*, which are required for realizing the communication via the shared communication channel. The *V2X* component, on the other hand, has a set of corresponding input and output ports for each of the other components and thus interconnects the three main subsystems. Note that all complex ports will be replaced by concrete port instances that are defined during the design process. Each component has a variable port called *Mode*, which represents the internal state of the respective subsystem from a functional perspective. The characteristics of this special type of port are explained in more detail in Chapter 4, where they are also formally defined. For now, it is sufficient to know that each component can be in exactly one mode $m_c(t)$ at any point in time $t \in \mathbb{T}$, and that the set of all possible modes of a component $c \in \mathbb{C}$ is denoted by M_c . Note that all components may have additional input and output ports required to fulfill their desired functionality, which are omitted in Figure 3.3 for simplicity.

The intended system behavior is detailed by considering four key scenarios, which have already been introduced in Figure 1.4 and are presented again in Figure 3.4. In the first scenario, CHAV 1 and CHAV 2 approach an urban intersection and subscribe to updates from an intelligent TMS that is responsible for supporting CHAVs in a safe and efficient crossing (see Figure 3.4a). The TMS observes the situation and determines that traffic efficiency could be increased by CHAV 1 turning in front of CHAV 2. It therefore recommends to perform a cooperative left turn. As they approach the conflict area, both CHAVs evaluate the recommendation based on their own situational overviews (see

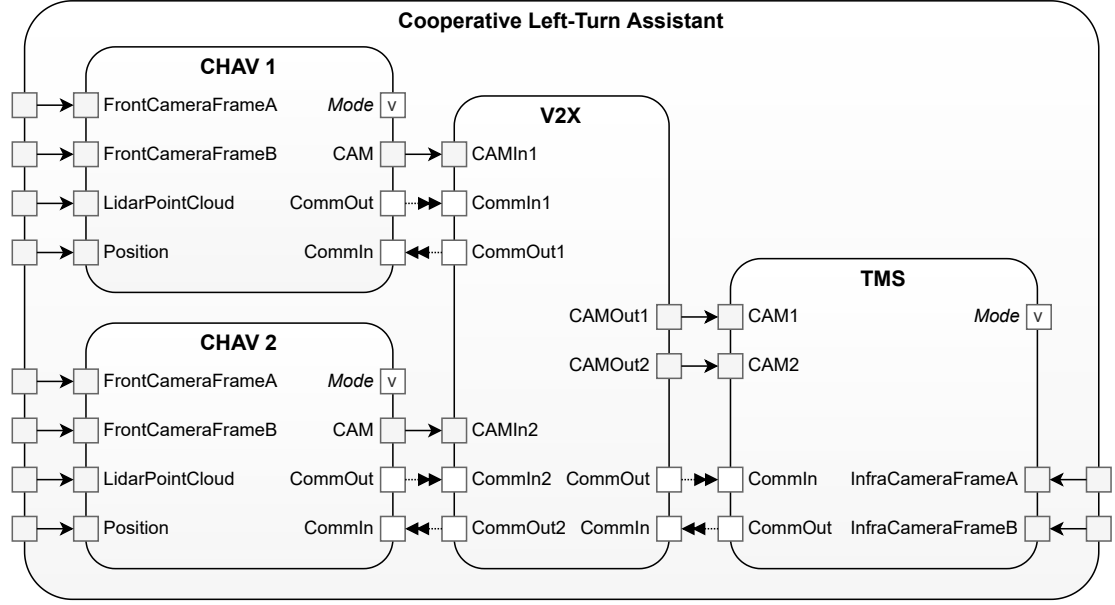


Fig. 3.3: Top-level architecture of the *Cooperative Left-Turn Assistant*

Figure 3.4b). They then jointly decide whether or not to execute the proposed maneuver. If the vehicles decide to perform the maneuver, the cooperative left turn is executed safely and efficiently by their individual ADSs (see Figure 3.4c), resulting in shorter waiting times, minimized braking stress, reduced emissions, and increased road safety. Finally, the two CHAVs unsubscribe from updates from the TMS while leaving the intersection area and continue their journey through urban traffic (see Figure 3.4d).

For further description, we assume that the consortium of manufacturers and suppliers collaborates closely in the early design phases. This includes both the joint selection and specification of relevant scenarios as well as the implementation of the initial phases of safety-oriented design. According to ISO 26262, the manufacturers and suppliers must first formulate an item definition and elicit initial functional requirements for the system. Here, both possible roles of the CHAV must be considered. As a next step, the consortium has to conduct an (extended) HARA based on the item definition to identify potential hazards and to evaluate the resulting risks according to their *severity* (S0 to S3), *exposure* (E0 to E4), and *controllability* (C0 to C3). Based on the results, each hazard is assigned an ASIL. In contrast with classical HARA, cooperative aspects must be taken into account, since the occurrence of hazards is closely related to the correct and timely functioning of cross-system coordination processes.² Two examples of possible hazards would be “overlooking an oncoming vehicle” and “misunderstanding during coordination”. Depending on the traffic situation, both hazards could result in a head-on collision

²Note that initial approaches for HARA of cooperative driving functions have already been proposed [72, 449, 484], but are not yet mature enough for industrial use. In the following, we therefore refer to the classical procedure according to ISO 26262 to demonstrate the envisioned design approach.

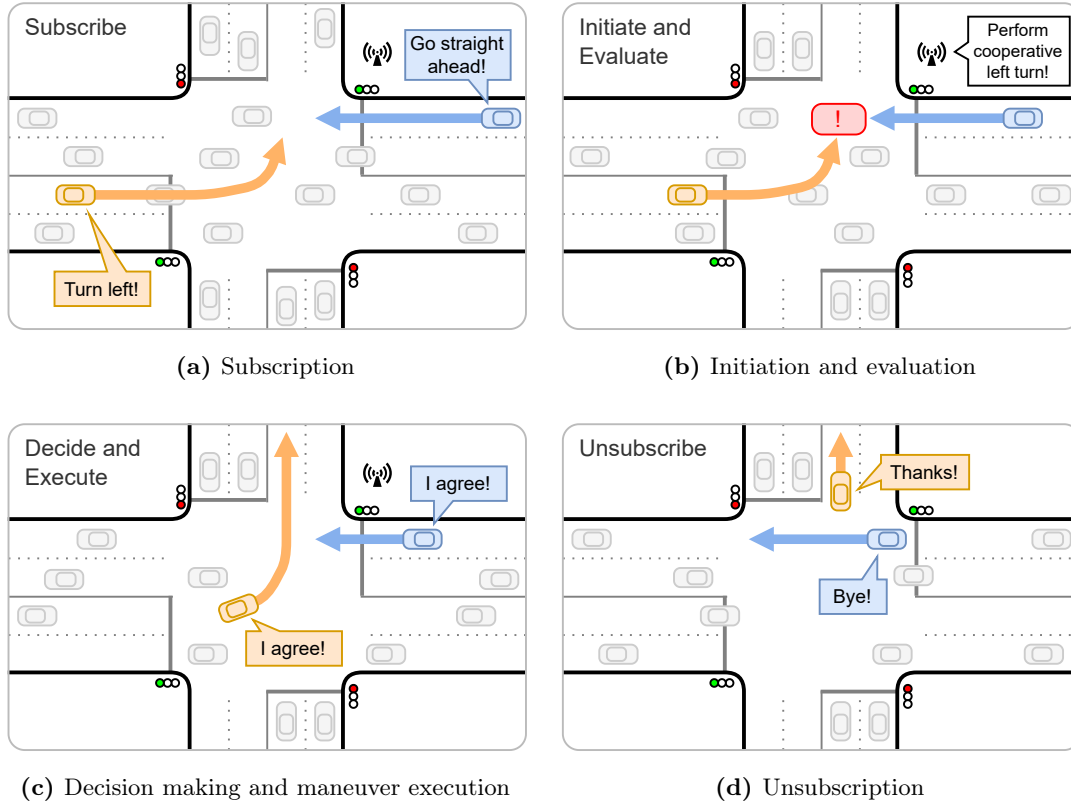


Fig. 3.4: Key scenarios relevant to the development of the CLTA

between two or more vehicles. Depending on the assumed role of the CHAV in focus, each hazard may imply different risks. For this reason, both alternatives must be considered as separate lines in the HARA table and must be evaluated individually:

1. Hazard “overlooking an oncoming vehicle”
 - CHAV in the role LEFT-TURNING VEHICLE: S3, E2, C3 → ASIL B
 - CHAV in the role STRAIGHT-AHEAD VEHICLE: S3, E2, C2 → ASIL B
2. Hazard “misunderstanding during coordination”
 - CHAV in the role LEFT-TURNING VEHICLE: S3, E3, C3 → ASIL C
 - CHAV in the role STRAIGHT-AHEAD VEHICLE: S3, E3, C3 → ASIL C

All example combinations can have very serious, potentially life-threatening consequences (S3). Although they occur in situations that are expected to happen with low or medium probability (E2/E3), they would be difficult to control (C3) by a human driver monitoring the system during operation. Consequently, the corresponding hazards are assigned ASIL B and ASIL C classifications, which impose high quality requirements on all subsequent process steps that focus on minimizing the resulting risks.

3 Consistent Specification of Top-Level Timing Requirements

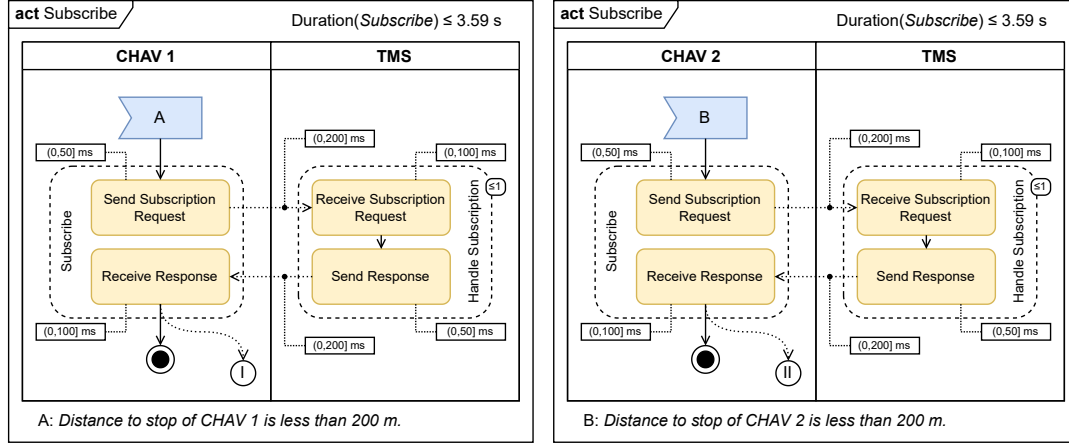
One of the required measures is the determination of FTTIs [233] for each hazard-role combination. Additional real-time requirements arise from the fact that the behavior of individual CSs must keep pace with physical processes, i.e., the driving behavior of the CHAVs as well as the evolution of the environment. In the example of the CLTA, timing and performance requirements can be calculated from speed limits, the distance to the stop line, and the maximum deceleration during comfort braking. Examples of such calculations can be found in [58, 360]. Corresponding assessments allow early determination of real-time requirements on CS level, which can then be decomposed into requirements for individual actions, such as sending a message or initiating a braking maneuver. This decomposition involves a number of inputs and techniques, such as expert knowledge about the timing behavior of external and legacy systems, the duration of physical processes, real-time behavior of communication mechanisms and protocols, and details of the technical implementation, such as sensor output frequencies, processing times, and end-to-end latencies. Moreover, it typically requires a careful time budgeting and involves trade-offs between communication and computation tasks.

In the following, the specification of the timing behavior of cooperative SoCPSs using TCADs is demonstrated based on the key scenarios from Figure 3.4. It is assumed that a HARA for the cooperative system has been performed beforehand and that FTTIs have been derived for all identified hazards. The specification of the application example addresses the iterative process of scenario-based activity modeling, in particular the *Modeling of an Initial Activity Model* and the *Extension and/or Refinement* activities from the envisioned design approach presented in Figure 3.1. To do this, we will first look at the *Subscription* scenario from Figure 3.4a and create an initial TCAM that includes a TCAD that specifies the timing behavior in this scenario. The model is then extended with corresponding TCADs covering the remaining three scenarios.

Subscription

The temporal behavior of the cooperative SoCPS in the *Subscription* scenario (see Figure 3.4a) is specified by the *Subscribe* activity, which is depicted in Figure 3.5. The activity starts if one of the CHAVs is less than 150 m from the stop line and ends when all actions have been completed. Initial hazard analyses and FTTI calculations have shown that the complete execution of the *Subscribe* activity must not take longer than 3.59 s.

After activation by the occurrence of an *A/B* event, CHAV 1/2 sends a subscription request to the TMS within 50 ms. The TMS receives the subscription request after a transmission time of 200 ms, processes it within a maximum of 100 ms and sends a response within 50 ms to acknowledge its receipt. After a further transmission time of 200 ms, the subscribing CHAV processes the response and completes the subscription process. The termination of the *Receive Response* action serves as prerequisite for the execution of subsequent actions, which are expressed by the explicit precedence constraints *I/II*. Note that both vehicles can perform the *Subscribe* activity at the same time, but do not have to. However, due to resource limitations, the TMS can only process one subscription request at a time. This is expressed by the uniqueness constraint that is annotated to the *Receive Subscription Request* and *Send Response* actions, indicated by the dashed-line

Fig. 3.5: TCAD for the *Subscribe* activity

Handle Subscription selection. Note that selections are not part of the formal TCAD syntax and semantics, but are used as a tool for easier visualization.

Initiation and Evaluation

The *Initiation and Evaluation* scenario (see Figure 3.4b) is covered by the *Initiate and Evaluate* activity presented in Figure 3.6. The activity begins when the distance to stop of one of the cooperation partners falls below 150 m, provided that the explicit precedence constraints *I* and *II* are satisfied. It ends when both CHAVs have completed their evaluation of the proposed maneuver. Again, hazard analyses and FTTI calculations revealed an upper time bound of 7.02 s for a complete execution of the activity.

Note that the current version of the scenario includes exactly two CHAVs that have a single conflict point. More complex scenarios, such as the cooperation of more than two CHAVs, the consideration of multiple conflict points, or the early termination of the *Initiate and Evaluate* activity due to the lack of potential conflicts or appropriate cooperation options, may be considered during iterative refinement of the scenario description as part of the *Extension and/or Refinement* design activity (see Figure 3.1).

After the activation the *Initiate and Evaluate* activity, the TMS checks the current traffic situation and detects a potential conflict situation between the two CHAVs in focus. To resolve the identified conflict, the TMS evaluates different cooperation options within 400 ms and sends a recommendation to perform a cooperative left-turn maneuver to CHAV 1 and CHAV 2 within 50 ms. The two vehicles receive the cooperation recommendation after a maximum transmission time of 300 ms, process the incoming messages within 150 ms, and immediately start the parallel evaluation of the proposed solutions based on their own situational overviews. Both CHAVs exchange their evaluation results by sending their version of the payoff matrix within 100 ms and receiving their cooperation partner's version within 50 ms. Due to the size of the messages to be transmitted, it is assumed that the transmission will take up to 300 ms, just as with the

3 Consistent Specification of Top-Level Timing Requirements

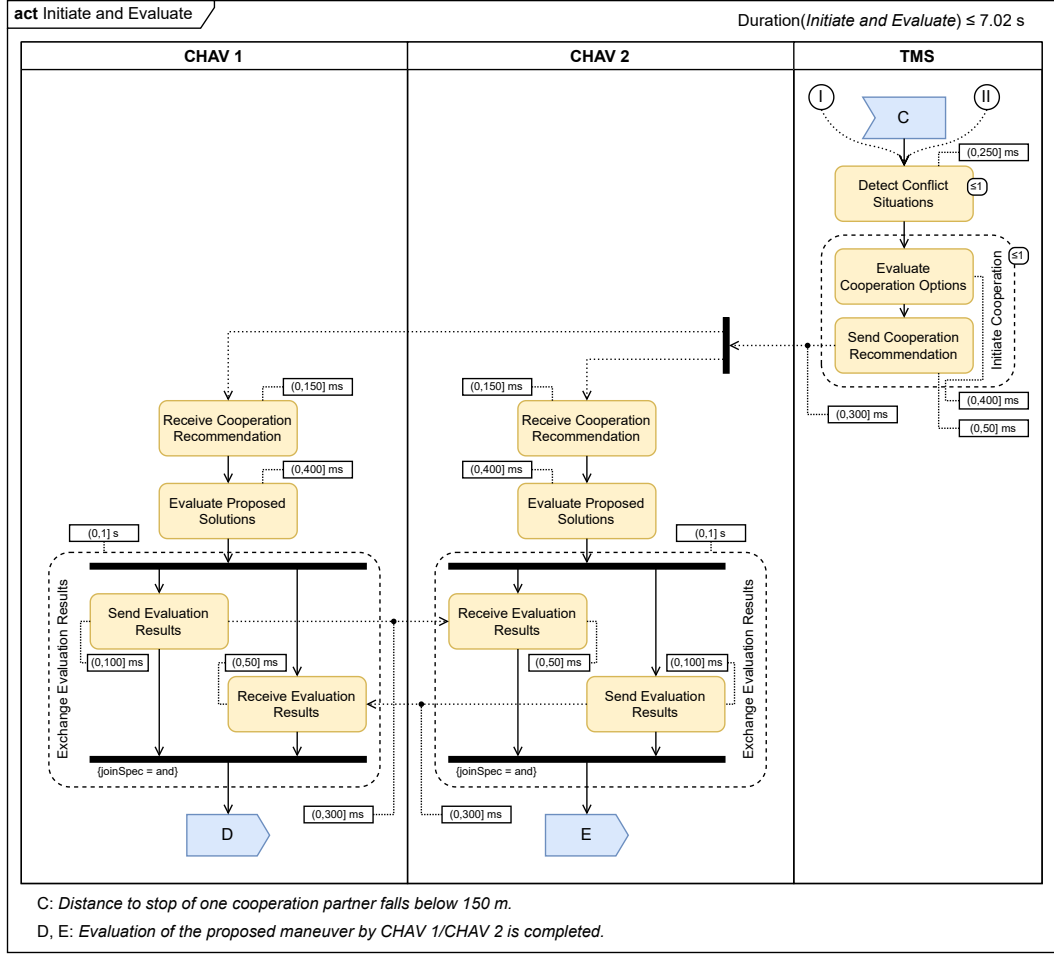


Fig. 3.6: TCAD for the *Initiate and Evaluate* activity

cooperation recommendation. The *Exchange Evaluation Results* selection, annotated with an action-to-action latency constraint that defines a maximum execution time of 1 s, is used to model a timeout mechanism. Thus, there are two possible ways to terminate the *Initiate and Evaluate* activity. In the first case, the activity ends if the terminating *D/E* signal is reached in time, i.e., in less than or equal to 1 s. In the second case, a timeout leads to a violation of the action-to-action latency constraint if the *D/E* signal is reached too late, i.e., after more than 1 s, and triggers an abortion of the activity execution.

Decision Making and Maneuver Execution

The *Decision Making and Maneuver Execution* scenario (see Figure 3.4c) is addressed by the *Decide and Execute* activity, which is shown in Figure 3.7. The activity begins when both CHAVs have completed the evaluation of the proposed maneuver. It ends either when a decision is made not to execute a proposed maneuver, or when the execution

3.3 Scenario-Based Activity Modeling

of a previously agreed upon cooperative maneuver is fully completed. Since the timing of the execution depends on the behavior of the environment, no duration constraint is defined on activity level. Note that there is no explicit communication during maneuver execution, which is consistent with the description of the CLTA in Section 1.2.1.

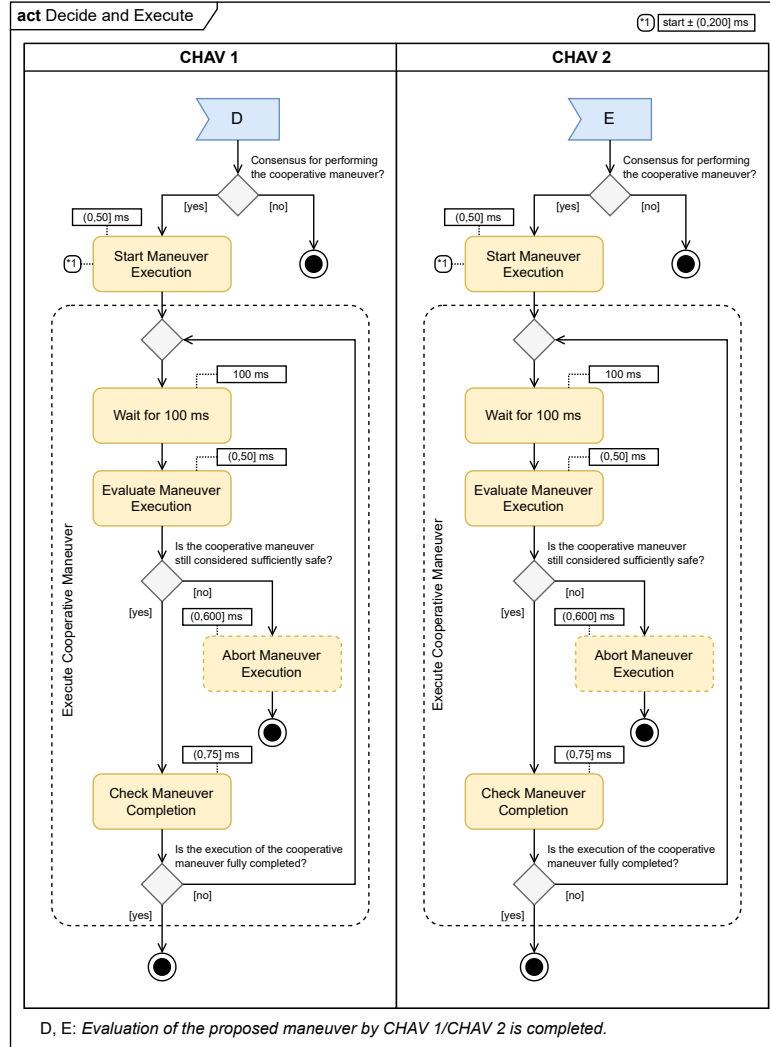


Fig. 3.7: TCAD for the *Decide and Execute* activity

The activity is initiated by the occurrence of an activating *D/E* event, which indicates that the evaluation of the proposed maneuver by CHAV 1/2 has been completed. The system first checks if there is a consensus between the cooperation partners to perform the cooperative maneuver. If not, the activity is aborted. If yes, the execution of the maneuver is started within a maximum of 50 ms. Note that the start of the maneuver execution of both CHAVs must take place synchronized, i.e., with a maximum time distance of 200 ms. After initiating the maneuver execution, the individual CLTA instances wait

3 Consistent Specification of Top-Level Timing Requirements

for 100 ms and then check whether the execution of the cooperative maneuver is still considered sufficiently safe. If the renewed risk assessment indicates that a safe execution is not possible, the maneuver is aborted within 600 ms. Since the *Abort Maneuver Execution* action is a complex process, it is modeled as a call behavior action, whose behavior can be detailed in subsequent versions of the scenario description. If the maneuver execution is considered sufficiently safe, it is checked within a maximum of 75 ms whether the maneuver has already been completed, and if not, a new iteration is started by returning control to the *Wait for 100 ms* action. After completing the maneuver execution, the *Decide and Execute* activity is terminated when the final activity node is reached.

Unsubscription

Finally, the temporal behavior in the *Unsubscription* scenario (see Figure 3.4d) is specified in terms of the *Unsubscribe* activity, which is depicted in Figure 3.8. The activity starts when one of the CHAVs leaves the intersection area and ends when all actions have been completed. Since the unsubscription process is not a safety-critical activity, the duration constraint on activity level, which requires a maximum execution time of 14.39 s, results from the assumptions regarding technical limitations, such as the assumed range of the V2X communication between the CHAVs and the TMS (see Section 1.2.1).

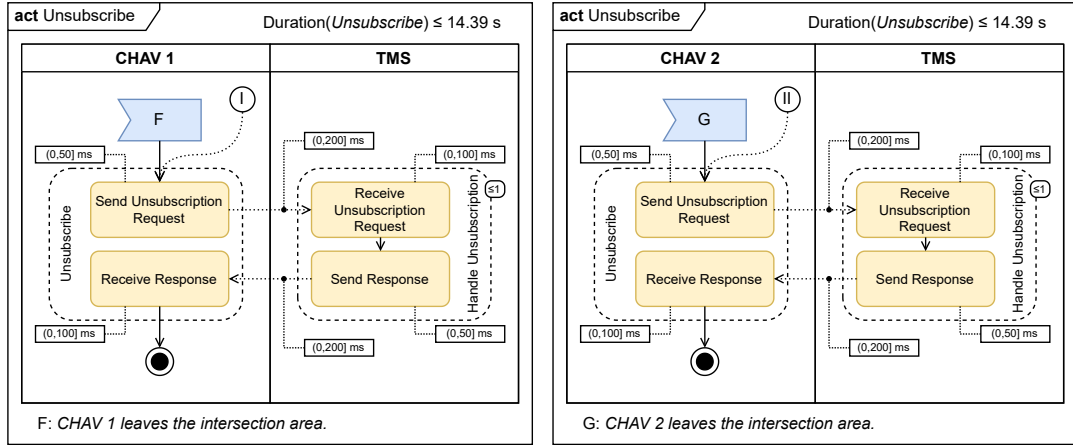


Fig. 3.8: TCAD for the *Unsubscribe* activity

The activity is modeled in the same way as the TCAD for the *Subscribe* activity and has identical timing and resource constraints. Since the use of the CLTA is limited to intersection situations, the scenario sequence ends with the unsubscription of both CHAVs. Accordingly, there are no precedence constraints to actions of subsequent activities.

3.4 Derivation of System-Level Timing Specifications

The modeling of the desired behavior of cooperative SoCPSs in terms of TCADs serves as the basis for the systematic derivation of component-based timing specifications on

system level. As a first step, the *Partitioning* design activity (see Figure 3.1) includes the decomposition of all system-spanning action sequences into local ones by dividing them at system boundaries, i.e., at events, signals, and cross-system control flows. The resulting (sub-)activity models focus on the actions that each CSs must contribute to enable successful interaction on SoCPS level. At the same time, each CS must provide all actions required to fulfill its roles and must satisfy all requirements resulting from the activity models. In a next step, all signals and events are mapped to interfaces from the functional architecture. At the same time, all actions of the involved systems are translated into operating modes of the corresponding top-level components. Finally, system-level timing specifications in the form of MTSL timing contracts are derived as part of the *Derivation of System-Level Specifications* design activity (see Figure 3.1).

Based on the formal definition of TCADs, the translation of a given TCAM, consisting of a set of TCADs, into a set of system-level timing specifications can be described by means of a four-step model transformation. As part of the process, the following translation steps must be completed for all relevant activities modeled as TCADs:

1. Partitioning of TCADs (see Section 3.4.1)
2. Mapping of events and signals to ports (see Section 3.4.2)
3. Formulation of system-level specifications (see Section 3.4.3)

3.4.1 Partitioning of Time-Constrained Activity Diagrams

The goal of the *Partitioning* design activity (see Figure 3.1) is to decompose each of the previously modeled TCADs into a set of partitioned TCADs for each *actor* that include all action sequences required to fulfill the defined *roles* in the scenarios considered as part of the design process. This ensures that the specified functionality of the cooperative system is systematically decomposed into functions of the individual CSs. As a basis for describing the partitioning approach, we define partitioned TCADs as follows:

Definition 6 (Partitioned Time-Constrained Activity Diagram). A *Partitioned Time-Constrained Activity Diagram* derived from a non-partitioned TCAD is a tuple $\text{pTCAD} = (act, \text{TCAD}, p, A, N, C, \Lambda', \Phi')$ with the following properties:

- $act \in \mathcal{A}$ is the activity that is (partly) modeled by pTCAD.
- TCAD is the non-partitioned TCAD that models $act \in \mathcal{A}$.
- $p \in \mathcal{P}$ is a selected *partition* that defines the scope of pTCAD. Recall that each partition p represents an actor's component $c_{cs} = \text{actor}(p) \in \mathcal{C}$ from the top-level architecture. Each partition is associated with a set of roles $\text{roles}(p)$.
- $A \subseteq \mathcal{A} \cup A_{\text{virt}} \cup A_{\text{event}}^{\leftrightarrow} \cup A_{\text{signal}}^{\leftrightarrow}$ is a set of *actions*, where A_{virt} is the set of *virtual actions* derived from the duration-constrained control flows of TCAD, $A_{\text{event}}^{\leftrightarrow}$ is a set

3 Consistent Specification of Top-Level Timing Requirements

of *interaction events*, and $A_{signal}^{\leftrightarrow}$ is a set of *interaction signals*. $A_{event}^{\leftrightarrow}$ and $A_{signal}^{\leftrightarrow}$ are both derived from the set of cross-system control flows $C_{\leftrightarrow}$ of TCAD:

$$\forall (an_1, an_2) \in C_{\leftrightarrow} \exists! e \in A_{event}^{\leftrightarrow} : e \rightarrow an_2 \quad (3.22)$$

$$\forall (an_1, an_2) \in C_{\leftrightarrow} \exists! s \in A_{signal}^{\leftrightarrow} : an_1 \rightarrow s \quad (3.23)$$

The set of *activating events* is defined as $A_{event}^{\uparrow} \subseteq A_{event}^{\uparrow} \cup A_{event}^{\leftrightarrow}$, and the set of *terminating signals* as $A_{signal}^{\downarrow} \subseteq A_{signal}^{\downarrow} \cup A_{signal}^{\leftrightarrow}$, respectively. All actions $a \in A$ are part of the selected partition p , i.e.,

$$\forall a \in A : \Xi(a) = p. \quad (3.24)$$

- $N \subseteq \mathcal{N}$ is a set of *control nodes*. The set of *initial nodes* is defined as $N_{init} \subseteq \mathcal{N}_{init}$, and the set of *final nodes* as $N_{final} \subseteq \mathcal{N}_{final}$. The set of final nodes $N_{final} = N_{\odot} \cup N_{\otimes}$ consists of a set of *final activity nodes* $N_{\odot} \subseteq \mathcal{N}_{\odot}$ and a set of *final flow nodes* $N_{\otimes} \subseteq \mathcal{N}_{\otimes}$. All control nodes $n \in N$ are part of the selected partition p , i.e.,

$$\forall n \in N : \Xi(n) = p. \quad (3.25)$$

In addition, A and N must fulfill the following conditions:

1. $|N_{init} \cup A_{event}^{\uparrow}| = 1$
 2. $|N_{\odot} \cup A_{signal}^{\downarrow}| = 1 \vee N_{\odot} \cup A_{signal}^{\downarrow} = \emptyset \wedge |N_{\otimes}| \geq 1$
- $C \subseteq (\mathcal{C} \setminus C_{\leftrightarrow}) \cup C_{virt} \cup C_{event}^{\leftrightarrow} \cup C_{signal}^{\leftrightarrow}$ is a set of *control flows*, where C_{virt} is the set of *virtual control flows* of TCAD, $C_{event}^{\leftrightarrow}$ is a set of *interaction event control flows*, and $C_{signal}^{\leftrightarrow}$ is a set of *interaction signal control flows*. $C_{event}^{\leftrightarrow}$ and $C_{signal}^{\leftrightarrow}$ are both derived from the set of cross-system control flows $C_{\leftrightarrow}$ of TCAD:

$$\forall (an_1, an_2) \in C_{\leftrightarrow} \exists! \rightarrow \in C_{event}^{\leftrightarrow} : \rightarrow = (e, an_2) \quad (3.26)$$

$$\forall (an_1, an_2) \in C_{\leftrightarrow} \exists! \rightarrow \in C_{signal}^{\leftrightarrow} : \rightarrow = (an_1, s) \quad (3.27)$$

All control flows $\rightarrow \in C$ are part of the selected partition p , i.e.,

$$\forall \rightarrow \in C : \Xi(\rightarrow) = p. \quad (3.28)$$

- $\Lambda' : A \rightarrow L$ is a *labeling function* that assigns a label $l \in L$ with $L \subseteq \mathcal{L}$ to each action $a \in A$. Note that labels of actions must be unique, i.e.,

$$\forall a \in A \nexists a' \in A : \Lambda'(a) = \Lambda'(a'). \quad (3.29)$$

- $\Phi' \subseteq \Phi$ is a set of *constraints*. Note that the set of *duration constraints* $\Phi'_{dur} \subseteq \Phi'$ does not contain any duration constraints for control flows, i.e.,

$$\forall \rightarrow \in C : dur(\rightarrow) = (0, 0), \quad (3.30)$$

since all duration-constrained control flows $\rightarrow = (an_1, an_2)$ are represented by virtual actions $a_{virt}(\rightarrow) \in A_{virt}$ and corresponding virtual control flows $\rightarrow_1^{virt}, \rightarrow_2^{virt} \in C_{virt}$ with $\rightarrow_1^{virt} = (an_1, a_{virt}(\rightarrow))$ and $\rightarrow_2^{virt} = (a_{virt}(\rightarrow), an_2)$. $\diamond$

3.4 Derivation of System-Level Timing Specifications

According to the definition, partitioned TCADs focus on (parts of) the behavior from a single TCAD partition. There are no top-level constraints and no duration constraints on control flows. In addition, there are also no cross-system control flows, since they are represented by corresponding pairs of terminating signals and activating events. All relevant constraints that are annotated to the elements of TCAD are also included in pTCAD. Since we assume that all action-to-action latency constraints are *locally observable* (see Equation 3.16), they can be included without further restrictions. Note that concurrent action sequences (e.g., resulting from forks/joins) are implicitly preserved. Although they may be split into multiple partitioned TCADs, all actions and control nodes, including their control flows, are included in one of the partitioned TCADs.

The execution semantics of partitioned TCADs is defined similarly to the semantics of non-partitioned TCADs. Consequently, a *run* π_{pTCAD} is defined as a valid pTCAD execution that starts with the generation of a set of initial tokens at *initial nodes* $n_{\text{init}} \in N_{\text{init}}$ and *activating events* $a \in A_{\text{event}}^{\uparrow}$ and ends with the consumption of either (1) one token at a *final activity node* $n_{\text{final}} \in N_{\odot}$ or a *terminating signal* $a \in A_{\text{signal}}^{\downarrow}$ or (2) all tokens at *final flow nodes* $n_{\text{final}} \in N_{\otimes}$. We denote the set of runs of a partitioned TCAD as Π_{pTCAD} , and the *language* of a partitioned TCAD as L_{pTCAD} , respectively.

Translation Procedure

As already mentioned, the goal of the *Partitioning* design activity (see Figure 3.1) is to partition a given TCAD into a set of partitioned TCADs. For this purpose, we define a *partitioning function* $\text{part} : \mathcal{T} \rightarrow 2^{\mathcal{P}}$, $\text{TCAD} \mapsto \{\text{pTCAD}_1, \dots, \text{pTCAD}_n\}$ with $n \in \mathbb{N}$, where $\mathcal{T}$ represents the set of all possible TCADs, and $\mathcal{P}$ the set of all possible partitioned TCADs, respectively. For the sake of simplicity, we denote the value of $\text{part}(\text{TCAD})$ as $\mathbb{P}_{\text{TCAD}}$. Note that we require $\text{part}(\text{TCAD}) = \mathbb{P}_{\text{TCAD}}$ to be *semantic-preserving*, i.e.,

$$L_{\text{TCAD}} = \bigcup_{i=1}^n L_{\text{pTCAD}_i}. \quad (3.31)$$

Based on the formal definitions, we define the translation procedure in terms of an algorithm that traverses all control flows of an input TCAD along the flow of tokens:

1. For each partition $p \in \mathcal{P}$:
 - a) For each initial node or activating event $an^{\uparrow} \in \mathcal{N}_{\text{init}} \cup \mathcal{A}_{\text{event}}^{\uparrow}$ with $\Xi(an^{\uparrow}) = p$:
 - i. Add a new $\text{pTCAD}_{\text{new}} = (\text{act}, \text{TCAD}, p, \emptyset, \emptyset, \emptyset, \Lambda', \emptyset)$ to $\mathbb{P}_{\text{TCAD}}$.
 - ii. Add the initial node or activating event $an^{\uparrow}$ to $\text{pTCAD}_{\text{new}}$.
 - iii. For each action or control node $an \in \mathcal{A} \cup \mathcal{N}$ with $\Xi(an^{\uparrow}) = p$ that indirectly depends on the initial node or activating event $an^{\uparrow}$, i.e., $an^{\uparrow} \rightsquigarrow an$, where $C_{\rightsquigarrow}$ does not include any cross-system control flows, i.e., $C_{\rightsquigarrow} \cap C_{\leftrightarrow} = \emptyset$:
 - A. Add the action or control node an to $\text{pTCAD}_{\text{new}}$. Note that this also includes the transfer of each terminating signal or final node $an^{\downarrow} \in \mathcal{A}_{\text{signal}}^{\downarrow} \cup \mathcal{N}_{\text{final}}$ with $\Xi(an^{\downarrow}) = p$ to $\text{pTCAD}_{\text{new}}$.

3 Consistent Specification of Top-Level Timing Requirements

- B. For each control flow $\rightarrow \in in(an) \cap C_{\rightsquigarrow}$:
 - If a duration constraint is annotated to control flow $\rightarrow = (an_1, an_2)$, i.e., $dur(\rightarrow) = (lb, ub)$ with $lb, ub \in \mathbb{T}$ and $ub > 0$:
 1. Add a virtual action $a_{virt} = a_{virt}(\rightarrow) \in A_{virt}$ with $dur(a_{virt}) = dur(\rightarrow)$ to $pTCAD_{new}$.
 2. Add two virtual control flows $\rightarrow_{virt}(\rightarrow) = \{\rightarrow_1^{virt}, \rightarrow_2^{virt}\} \subseteq C_{virt}$ with $\rightarrow_1^{virt} = (an_1, a_{virt})$ and $\rightarrow_2^{virt} = (a_{virt}, an_2)$ to $pTCAD_{new}$.
 - Otherwise: Add control flow $\rightarrow$ to $pTCAD_{new}$.
- iv. For each cross-system control flow $(an_1, an_2) \in C_{\leftrightarrow}$ with $\Xi(an_1) = p$:
 - A. Add an interaction signal $s \in A_{signal}^{\leftrightarrow}$ to $pTCAD_{new}$.
 - B. Add a control flow $(an_1, s) \in C_{signal}^{\leftrightarrow}$ to $pTCAD_{new}$.
 - C. If a duration constraint is annotated to control flow $(an_1, an_2) \in C_{\leftrightarrow}$, i.e., $dur((an_1, an_2)) = (lb, ub)$ with $lb, ub \in \mathbb{T}$ and $ub > 0$:
 1. Add a new $pTCAD'_{new} = (act, TCAD, p', \emptyset, \emptyset, \emptyset, \Lambda', \emptyset)$ to $\mathbb{P}_{TCAD}$, where $p' \in \mathcal{P}$ is a new partition that represents the communication system used to realize the interaction between the two actors.
 2. Add a virtual action $a_{\leftrightarrow} = a_{virt}((an_1, an_2)) \in A_{virt}$ with $dur(a_{\leftrightarrow}) = dur((an_1, an_2))$ to $pTCAD'_{new}$.
 3. Add an interaction event $e_{\leftrightarrow} \in A_{event}^{\leftrightarrow}$ to $pTCAD'_{new}$.
 4. Define a shared label for s and $e_{\leftrightarrow}$ with $\Lambda'(s) = \Lambda'(e_{\leftrightarrow})$ to establish a connection with s from step 1. a) iv. A.
 5. Add a control flow $(e_{\leftrightarrow}, a_{\leftrightarrow}) \in C_{event}^{\leftrightarrow}$ to $pTCAD'_{new}$.
 6. Add an interaction signal $s_{\leftrightarrow} \in A_{signal}^{\leftrightarrow}$ to $pTCAD'_{new}$.
 7. Add a control flow $(a_{\leftrightarrow}, s_{\leftrightarrow}) \in C_{signal}^{\leftrightarrow}$ to $pTCAD'_{new}$.
- b) For each cross-system control flow $(an_1, an_2) \in C_{\leftrightarrow}$ with $\Xi(an_2) = p$:
 - i. Add a new $pTCAD_{new} = (act, TCAD, p, \emptyset, \emptyset, \emptyset, \Lambda', \emptyset)$ to $\mathbb{P}_{TCAD}$.
 - ii. Add an interaction event $e \in A_{event}^{\leftrightarrow}$ to $pTCAD_{new}$.
 - iii. Define a label for e depending on the duration constraint of (an_1, an_2) :
 - If $dur((an_1, an_2)) = (0, 0)$: Define a shared label for s and e with $\Lambda'(s) = \Lambda'(e)$ to establish a connection with s from step 1. a) iv. A.
 - Otherwise: Define a shared label for $s_{\leftrightarrow}$ and e with $\Lambda'(s_{\leftrightarrow}) = \Lambda'(e)$ to establish a connection with $s_{\leftrightarrow}$ from step 1. a) iv. C. 6.
 - iv. Add a control flow $(e, an_2) \in C_{event}^{\leftrightarrow}$ to $pTCAD_{new}$.
 - v. Execute step 1. a) iii.
 - vi. Execute step 1. a) iv.

2. For each pTCAD $\in \mathbb{P}_{\text{TCAD}}$:

- a) For each explicit precedence constraint $\phi_{\text{expl}} = (an_1, an_2) \in \Phi_{\text{expl}}$:
 - If $\exists a \in A \setminus (A_{\text{event}} \cup A_{\text{signal}}) : a = an_1 \vee a = an_2$: Add ϕ_{expl} to Φ' .
 - Otherwise: Skip ϕ_{expl} .
- b) For each duration constraint $\phi_{\text{dur}} = (ac, lb, ub) \in \Phi_{\text{dur}}$:
 - If $\exists a \in A \setminus (A_{\text{event}} \cup A_{\text{signal}}) : a = ac$: Add ϕ_{dur} to Φ' .
 - Otherwise: Skip ϕ_{dur} .
- c) For each action-to-action latency constraint $\phi_{\text{lat}} = (as^*(a_1, a_n), lb, ub) \in \Phi_{\text{lat}}$:
 - If $\exists a'_1, a'_n \in A \setminus A_{\text{event}} \cup A_{\text{signal}} : a'_1 = a_1 \wedge a'_n = a_n$: Add ϕ_{lat} to Φ' .
 - Otherwise: Skip ϕ_{lat} .
- d) For each synchronization constraint $\phi_{\text{sync}} = (A_{\text{sync}}, \delta) \in \Phi_{\text{sync}}$:
 - If $\exists a \in A \setminus (A_{\text{event}} \cup A_{\text{signal}}) : a \in A_{\text{sync}}$: Add ϕ_{sync} to Φ' .
 - Otherwise: Skip ϕ_{sync} .
- e) For each uniqueness constraint $\phi_{\text{uni}} = (a, n_a) \in \Phi_{\text{uni}}$:
 - If $\exists a' \in A \setminus (A_{\text{event}} \cup A_{\text{signal}}) : a' = a$: Add ϕ_{uni} to Φ' .
 - Otherwise: Skip ϕ_{uni} .
- f) Add a label for each action $a \in A \setminus (A_{\text{event}}^{\leftrightarrow} \cup A_{\text{signal}}^{\leftrightarrow})$ so that $\Lambda'(a) = \Lambda(a)$.
- g) Add a label for partition p so that $\Lambda'(p) = \Lambda(p)$.

As a result of applying the translation procedure, the original TCAD is decomposed into a set of partitioned TCADs. Since each pTCAD only contains elements from a single partition $p \in \mathcal{P}$, which represent an actor, all elements can be assigned to the corresponding component $c_{cs} = \text{actor}(p) \in \mathcal{C}$ from the top-level architecture.

Besides the transfer of all relevant actions $A \subseteq \mathcal{A}$, control nodes $N \subseteq \mathcal{N}$, control flows $C \subseteq \mathcal{C}$, labels $L \subseteq \mathcal{L}$, and constraints $\Phi' \subseteq \Phi$, the translation procedure includes two particularly important translation steps. First, in step 1. a) iii. B., each duration-constrained control flow (see Figure 3.9a) is translated into a corresponding virtual action that is annotated with the same duration constraint (see Figure 3.9b).

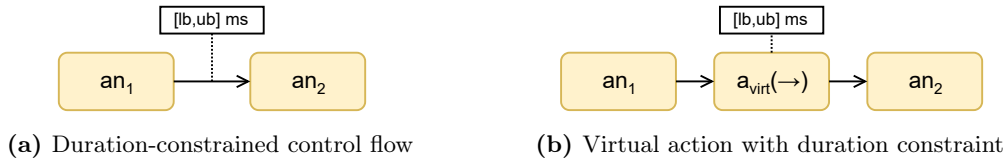


Fig. 3.9: Translation of duration-constrained control flows

Second, in steps 1. a) iv. and 1. b), each cross-system control flow (see Figure 3.10a) is translated into a pair of interaction signals and events (see Figure 3.10b). The proposed translation procedure also supports handling combinations of both types of control flows,

3 Consistent Specification of Top-Level Timing Requirements

i.e., the translation of duration-constrained, cross-system control flows (see step 1. a) iv. C.). Note that all corresponding translations are semantic-preserving, since signals $s \in \mathcal{A}_{signal}$, events $e \in \mathcal{A}_{event}$, and (unconstrained) control flows $\rightarrow \in \mathcal{C}$ are timeless.

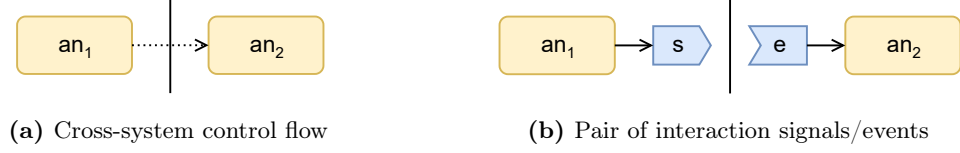


Fig. 3.10: Translation of cross-system control flows

Application Example

As part of the *Modeling of an Initial Activity Model* and the *Extension and/or Refinement* design activities (see Figure 3.1), the intended system behavior of the CLTA has been modeled by means of four TCADs covering the *Subscribe*, *Initiate and Evaluate*, *Decide and Execute*, and *Unsubscribe* activities presented in Figure 3.4 (see Section 3.3.2). The next step is to use the partitioning function $part(TCAD)$ to translate each $TCAD \in \{TCAD_{sub}, TCAD_{init/eval}, TCAD_{dec/exec}, TCAD_{unsub}\}$ into a set of partitioned TCADs. The translation results in the derivation of sets of partitioned TCADs, each of which can be assigned to exactly one actor. Note that all virtual actions, signals, and events resulting from the translation of duration-constrained, cross-system control flows between the two CHAVs and the TMS are mapped to the *V2X* component (see Figure 3.3).

Since both CHAVs share a common set of roles, their TCAD partitions include the same modeling elements. Additionally, the interfaces of the actors' components are identical (see Figure 3.3). In order to avoid redundancy, we therefore introduce a virtual *CHAV* component as a “base class” of both components that is used as a proxy. This allows us to combine all subsequent process steps related to CHAVs, meaning that all operations performed using the CHAV shortcut are applied to both CHAV 1 and CHAV 2.

Subscription The TCAD for the *Subscribe* activity (see Figure 3.5) is partitioned into a set of five partitioned TCADs, which are listed in Figure 3.11. Note that the partitioned TCADs for CHAV 1 and CHAV 2 are combined using the CHAV shortcut. In a first step, the action sequences from the *A/B* events to the first cross-system control flows between the CHAVs and the TMS, which are annotated with duration constraints, are decomposed into $pTCAD_{sub}^1$ (see steps 1. a) i.–iii.) and $pTCAD_{sub}^2$ (see step 1. a) iv.).

Next, the actions of the TMS for processing the *Subscription Requests* are translated into $pTCAD_{sub}^3$ (see step 1. b)), and the cross-system control flows between the TMS and the CHAVs are decomposed into $pTCAD_{sub}^4$ (see step 1. a) iv.). Finally, the processing of the *Subscription Responses* and the termination of the activities by final nodes are represented by $pTCAD_{sub}^5$ (see step 1. b)). Note that all TCAD constraints such as precedence, duration, and uniqueness constraints are either preserved or translated into semantically preserving substitutes, e.g., in case of the two cross-system control flows.

3.4 Derivation of System-Level Timing Specifications

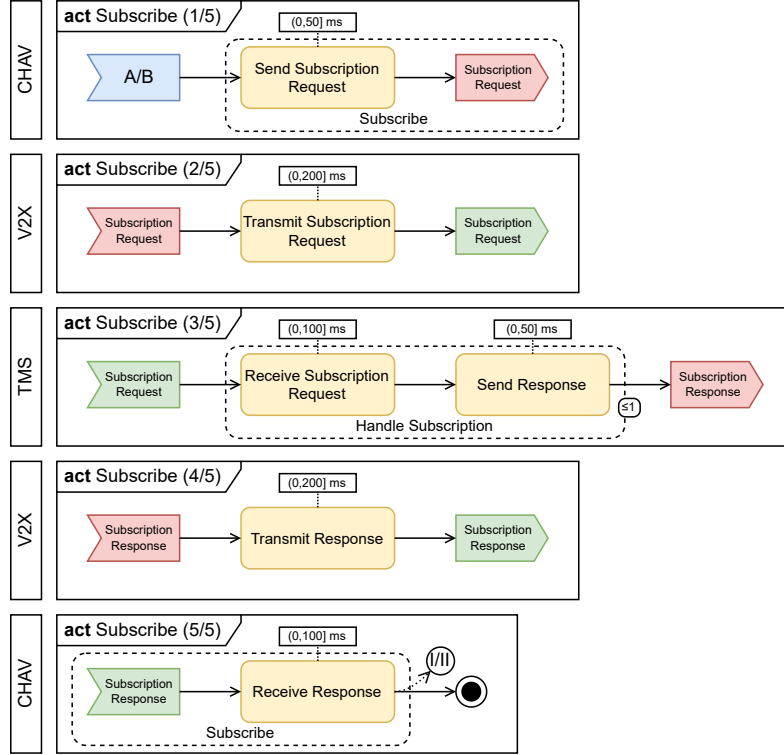


Fig. 3.11: Partitioned TCADs for the *Subscribe* activity

Initiation and Evaluation Analogously, the TCAD for the *Initiate and Evaluate* activity (see Figure 3.6) is translated into a set $\mathbb{P}_{\text{TCAD}_{init/eval}} = \text{part}(\text{TCAD}_{init/eval})$ of four partitioned TCADs, which are presented in Figure 3.12. At the beginning of the partitioning of $\text{TCAD}_{init/eval}$, the action sequence starting from the *C* event up to the first duration-constrained, cross-system control flow between the TMS and the two CHAVs is translated into pTCAD_{init}^1 (see steps 1. a) i.–iii.) and pTCAD_{init}^2 (see step 1. a) iv.).

The subsequent processing of the received *Cooperation Recommendations* by the two CHAVs as well as the evaluation of the proposed solutions and the exchange of the evaluation results up to the termination of the activity by sending the terminating *D/E* signals are combined in pTCAD_{eval}^1 (see step 1. b)). The exchange of the individual *Evaluation Results* is realized in terms of pTCAD_{eval}^2 (see step 1. a) iv.). Again, all constraints, including the action-to-action-latency constraint of the *Exchange Evaluation Results* selection are transferred to the corresponding partitioned TCADs.

Decision Making and Execution The translation of the TCAD for the *Decide and Execute* activity (see Figure 3.7) can be done very easily. Since only the two CHAVs are involved, both TCADs are modeled analogously, and no communication is required, the resulting set $\mathbb{P}_{\text{TCAD}_{dec/exec}} = \text{part}(\text{TCAD}_{dec/exec})$ contains only a single partitioned TCAD. The application of the translation procedure is therefore limited to the execution

3 Consistent Specification of Top-Level Timing Requirements

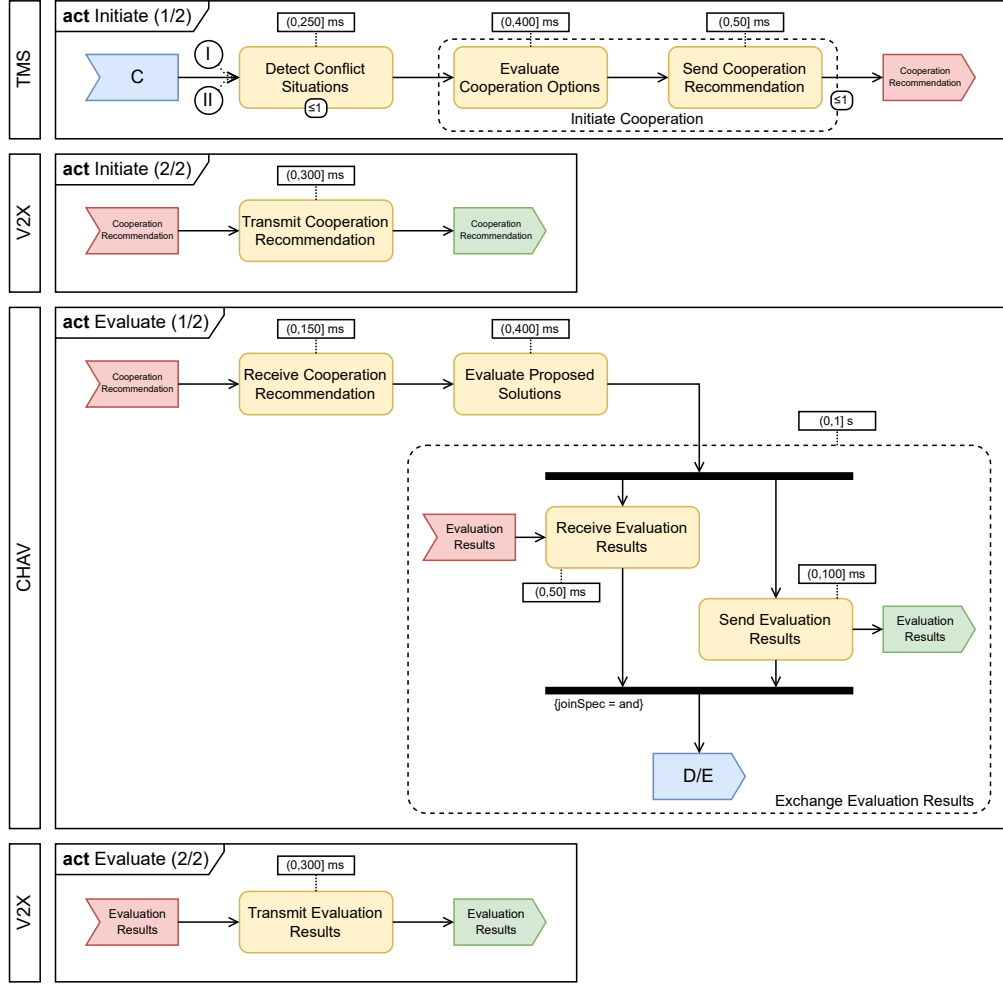


Fig. 3.12: Partitioned TCADs for the *Initiate* and *Evaluate* activity

of steps 1. a) i.–iii. and includes the transfer of all actions and control nodes from the *D/E* events to the final nodes. Note that the resulting $\text{pTCAD}_{dec/exec}^1$ is not shown explicitly, as it is a trivial combination of the two TCAD parts from Figure 3.7.

Unsubscription As a final step, the TCAD for the *Unsubscribe* activity (see Figure 3.8) is decomposed into another set $\mathbb{P}_{\text{TCAD}_{unsub}} = \text{part}(\text{TCAD}_{unsub})$ of five partitioned TCADs, which are depicted in Figure 3.13. Due to the analogous modeling of TCAD_{sub} and TCAD_{unsub} , the partitioning follows the same procedure. Thus, the action sequences from the *F/G* events to the cross-system control flows between the CHAVs and the TMS are translated into pTCAD_{unsub}^1 (see steps 1. a) i.–iii.) and pTCAD_{unsub}^2 (see step 1. a) iv.).

Afterwards, the actions of the TMS for processing the *Unsubscription Requests* are translated into pTCAD_{unsub}^3 (see step 1. b)), and the cross-system control flows between the TMS and the CHAVs are decomposed into pTCAD_{unsub}^4 (see step 1. a) iv.). Finally,

3.4 Derivation of System-Level Timing Specifications

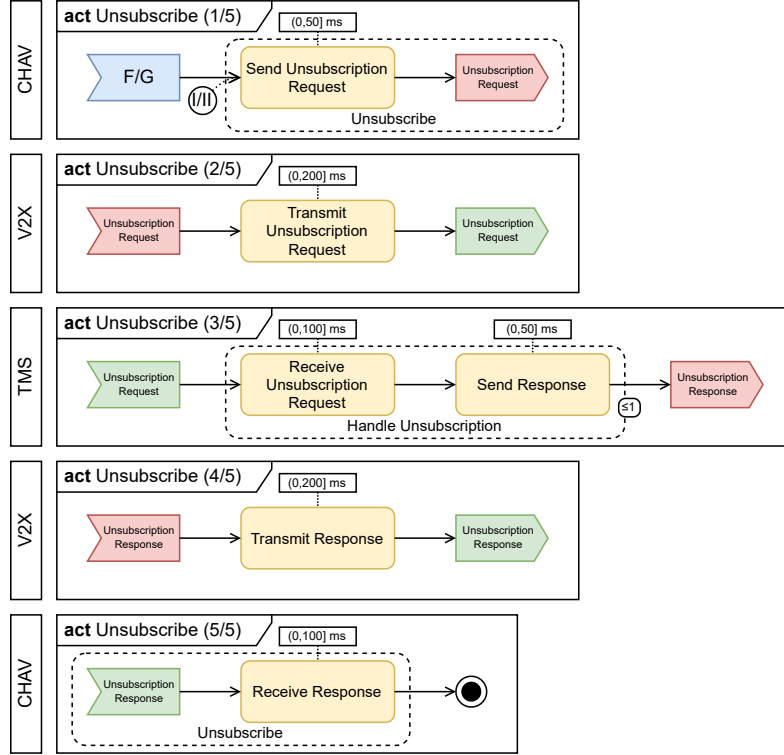


Fig. 3.13: Partitioned TCADs for the *Unsubscribe* activity

the processing of the *Unsubscription Responses* and the termination of the activities by final nodes are represented by pTCAD_{unsub}⁵ (see step 1. b)). The translation of TCAD_{unsub} into $\mathbb{P}_{\text{TCAD}_{unsub}}$ completes the partitioning of all TCADs relevant to the development of the CLTA. The result is a combined set of 15 partitioned TCADs that fully represent all relevant actions of the involved actors required to ensure safe and efficient cooperation.

3.4.2 Mapping of Events and Signals to Ports

The last missing puzzle pieces for translating the previously derived set of partitioned TCADs into a set of timing specifications are (1) the mapping of events and signals to ports from the top-level architecture, and (2) the formulation of additional assumptions, e.g., about the occurrence of external events. Both steps aim to establish a direct correspondence between the activity models and the functional architecture on SoCPS level defined at the beginning of the envisioned design approach (see Figure 3.1).

Recall that the top-level functional architecture consists of a set of hierarchically nested components $c \in \mathcal{C}$, each of which has a set of ports $P_c \subseteq \mathcal{P}$ connected by a set of signals $\mathcal{S}$ (see Section 2.3.1). As part of the description of the top-level architecture of the CLTA in Section 3.3.2, we have already briefly introduced that each component $c \in \mathcal{C}$ can have a dedicated *Mode* port that stores information about its internal state. We also defined that each of these components is in exactly one mode $m_c(t)$ at any point in time $t \in \mathbb{T}$,

3 Consistent Specification of Top-Level Timing Requirements

and that the set of all possible modes of a component is denoted by M_c . During the following translation steps, each action $a \in A \setminus (A_{event} \cup A_{signal})$ that is allocated to a partition $p \in \mathcal{P}$ is translated into a mode $m_a = m(a)$ with $m_a \in M_{c_{cs}}$ of the actor's top-level component $c_{cs} = actor(p) \in \mathcal{C}$. To establish a relationship between “classical” *event ports* (see Section 2.3.1) and *variable ports* (see Section 4.2.2), we use two types of events: *set*(p, v) events assign a value $v \in V_p$ to a variable port $p \in \mathcal{P}$. If the value of p changes as a result of this assignment, a *change*(p, v) event occurs, representing a value change to v . Note that all concepts related to mode-dependent specifications, i.e., variable ports, modes, *set*($\cdot$), and *change*($\cdot$) events, are formally defined in Chapter 4.

The previously created activity models include all relevant actions that are required to fulfill the defined roles and to enable successful cooperation between CSs. However, TCADs do not define under which circumstances an activity may be started or in which states the individual CSs remain after the completion of a TCAD execution. These issues are addressed by the explicit definition of so-called *pre*- and *postconditions* for each activity/actor combination. The goal of this preparatory activity is to define assumptions about the behavior of the system prior to the execution of an activity (*precondition*), as well as an assured end state that is valid after the—either successful or aborted—execution of an activity (*postcondition*). The following steps must be performed for each TCAD $\in \mathbb{D}_{TCAM}$, defining pre- and postconditions for all pTCAD $\in \mathbb{P}_{TCAD}$:

1. For each partition $p \in \mathcal{P}$:
 - a) Determine the actor's top-level component $c_{cs} = actor(p) \in \mathcal{C}$.
 - b) Check if the activity is directly called by another activity:
 - If the activity is not called by another activity, i.e., $\nexists TCAD_{pre} \in \mathbb{D}_{TCAM} : TCAD_{pre} \rightsquigarrow TCAD$: Define a *precondition mode* $m_{c_{cs}}^{\triangleleft}$ for c_{cs} that has to be active before the activity starts, i.e., during before phase $p_{act}^{\triangleleft}$.
 - Otherwise: Define a *call mode* $m_{c_{cs}}^{\nabla}$ for c_{cs} that has to be active upon reaching the terminating signal $s \in \mathcal{A}_{signal}^{\downarrow}$ of $TCAD_{pre}$ that triggers the activating event $e \in \mathcal{A}_{event}^{\uparrow}$ of $TCAD$, i.e., during instant time phase $p_{e/s}^{\nabla}$.
 - c) Define a *postcondition mode* $m_{c_{cs}}^{\triangleright}$ for c_{cs} that must be active after the activity has been completed or aborted, i.e., during after phase $p_{act}^{\triangleright}$.
2. Define a shortcut for each resulting mode.

Note that the postcondition of an activity that calls another activity, i.e., $TCAD_{pre} \rightsquigarrow TCAD$, is interpreted as a default value that only becomes effective if the activity is aborted before the terminating signal is sent, e.g., in case of constraint violations.

With regard to the CLTA application example, we define pre- and postconditions for all relevant activity/actor combinations, which are listed in Table 3.1. For this purpose, we assume that before and during the approach to the intersection, the CHAVs are in a *FreeFlow* mode, in which the driving task is performed either by a human driver or an ADS. The TMS is in *Idle* mode by default and is only set to functional modes by

3.4 Derivation of System-Level Timing Specifications

external requests. An important prerequisite for activating the *Initiate and Evaluate* activity is the fact that both CHAVs must have completed their individual subscription processes. Accordingly, the precondition for this activity/actor combination is that both CHAVs must be in *Subscribed* mode. The same applies to the *Unsubscribe* activity, since it is only necessary to unsubscribe after a successful subscription. The *Decide and Execute* activity is called directly via the signal/event pairs included in $\text{TCAD}_{init/eval}$ and $\text{TCAD}_{dec/exec}$, therefore no precondition is defined for this case. All postconditions have been set accordingly, using the *Subscribed* mode as the default mode of the CHAVs for the first three activities, and the initial *FreeFlow* mode as the final postcondition.

Tab. 3.1: Pre- and postconditions

<i>Activity</i>	<i>Preconditions</i>		<i>Postconditions</i>	
	CHAV	TMS	CHAV	TMS
<i>Subscribe</i>	<i>FreeFlow</i>	<i>Idle</i>	<i>Subscribed</i>	<i>Idle</i>
<i>Initiate and Evaluate</i>	<i>Subscribed</i>	<i>Idle</i>	<i>Subscribed</i>	<i>Idle</i>
<i>Decide and Execute</i>	$\neg^1$	$\neg^2$	<i>Subscribed</i>	$\neg^2$
<i>Unsubscribe</i>	<i>Subscribed</i>	<i>Idle</i>	<i>FreeFlow</i>	<i>Idle</i>

¹ directly called ² not involved

The definition of pre- and postconditions for all activity/actor combinations serves as the basis for mapping all events $e \in A_{event}$ and signals $s \in A_{signal}$ of the partitioned TCADs to ports from the functional architecture. In addition, all initial nodes $n_{init} \in \mathcal{N}_{init}$ and final nodes $n_{final} \in \mathcal{N}_{final}$ are included in the mapping. To ensure a consistent mapping between the two worlds, each input pTCAD must meet the following requirements:

1. Each initial node $n_{init} \in \mathcal{N}_{init}$ has exactly one subsequent action, i.e.,

$$\forall n_{init} \in \mathcal{N}_{init} \exists! a \in A \setminus (A_{event} \cup A_{signal}) : n_{init} \rightarrow a. \quad (3.32)$$

2. Each event $e \in A_{event}$ has exactly one subsequent action, i.e.,

$$\forall e \in A_{event} \exists! a \in A \setminus (A_{event} \cup A_{signal}) : e \rightarrow a. \quad (3.33)$$

Based on these assumptions, the mapping can be established by using a third translation that defines a *port/event mapping function* $pe : \mathcal{N}_{init} \cup A_{event} \cup A_{signal} \cup \mathcal{N}_{final} \rightarrow P_{c_{cs}} \cup \Sigma$. The following steps must be completed for all $\text{TCAD} \in \mathbb{D}_{\text{TCAM}}$ of a given TCAM:

1. For each pTCAD $\in \mathbb{P}_{\text{TCAD}}$:
 - a) Determine the actor's top-level component $c_{cs} = \text{actor}(p) \in \mathcal{C}$.
 - b) Map each initial node $n_{init} \in \mathcal{N}_{init}$ to a $\text{change}(\text{Mode}, m_a)$ event that occurs as a result of a mode change of c_{cs} to $m_a = m(a)$, where $a \in A \setminus (A_{event} \cup A_{signal})$ is the subsequent action of n_{init} , i.e., $n_{init} \rightarrow a$ (see Equation 3.32).

3 Consistent Specification of Top-Level Timing Requirements

- c) For each activating event $e \in A_{event}^{\uparrow} \setminus A_{event}^{\leftrightarrow}$:
 - If the activity is not directly called by another activity, i.e., $\nexists \text{TCAD}_{pre} \in \mathbb{D}_{\text{TCAM}} : \text{TCAD}_{pre} \rightsquigarrow \text{TCAD}$: Define a new *virtual event* $\sigma^{virt} \in \Sigma$ in terms of an event specification that specifies how e is derived from a set of input events that occur on ports $P_{in}^{virt} \subseteq P_{c_{cs}}^{in}$. Map e to σ^{virt} .
 - Otherwise: Map e to a $\text{change}(\text{Mode}, m_{c_{cs}}^{\nabla})$ event that occurs as a result of a mode change of c_{cs} to the call mode $m_{c_{cs}}^{\nabla}$.
- d) For each event $e \in A_{event} \setminus (A_{event}^{\uparrow} \setminus A_{event}^{\leftrightarrow})$:
 - If a corresponding input port $p_i \in P_{c_{cs}}^{in}$ of c_{cs} exists: Map e to p_i .
 - Otherwise: Add a new input port $p'_i \in P_{c_{cs}}^{in}$ to c_{cs} . Map e to p'_i .
- e) For each signal $s \in A_{signal} \setminus (A_{signal}^{\downarrow} \setminus A_{event}^{\leftrightarrow})$:
 - If a corresponding output port $p_o \in P_{c_{cs}}^{out}$ of c_{cs} exists: Map s to p_o .
 - Otherwise: Add a new output port $p'_o \in P_{c_{cs}}^{out}$ to c_{cs} . Map s to p'_o .
- f) For each terminating signal $s \in A_{signal}^{\downarrow} \setminus A_{signal}^{\leftrightarrow}$:
 - If the activity directly calls another activity, i.e., $\exists \text{TCAD}_{post} \in \mathbb{D}_{\text{TCAM}} : \text{TCAD} \rightsquigarrow \text{TCAD}_{post}$: Map s to a $\text{set}(\text{Mode}, m_{c_{cs}}^{\nabla})$ event that triggers a mode change of c_{cs} to the call mode $m_{c_{cs}}^{\nabla}$.
 - Otherwise: Map s to a $\text{set}(\text{Mode}, m_{c_{cs}}^{\triangleright})$ event that triggers a mode change of c_{cs} to the postcondition mode $m_{c_{cs}}^{\triangleright}$.
- g) Map each final node $n_{final} \in N_{final}$ to a $\text{set}(\text{Mode}, m_{c_{cs}}^{\triangleright})$ event that triggers a mode change of c_{cs} to the postcondition mode $m_{c_{cs}}^{\triangleright}$.

2. Define a shortcut for each relevant port.

The translation procedure is also applied to the CLTA application example. For this purpose, all initial nodes, events, signals, and final nodes included in the partitioned TCADs from $\mathbb{P}_{\text{TCAD}} = \mathbb{P}_{\text{TCAD}_{sub}} \cup \dots \cup \mathbb{P}_{\text{TCAD}_{unsub}}$ are mapped to ports from the functional top-level architecture (see Figure 3.3). The resulting mapping is shown in Table 3.2, which also introduces shortcuts for all relevant ports, facilitating later use in system-level timing specifications. According to the translation procedure, all activating events A/B , C , D/E , and F/G are mapped to either corresponding virtual events, i.e., *SubTrigger*, *RecTrigger*, and *UnsubTrigger*, or *change()* events that occur as a result of a mode change to $m_{c_{cs}}^{\nabla}$ (see step 1. c)). On the other hand, the terminating D/E signal and all final nodes are mapped to *set()* events, which trigger either a mode change to their call mode $m_{c_{cs}}^{\nabla}$ or their postcondition mode $m_{c_{cs}}^{\triangleright}$ (see step 1. f)-g)). Note that CHAV 1 and CHAV 2 are referred to as CHAV, indicating that their ports and events are identical.

All other events and signals are mapped to the input and output ports of the actors' top-level components according to steps 1. d) and 1. e). For example, the *Subscription Request* event included in pTCAD_{sub}^1 (see Figure 3.11) is mapped to the *SubRequest* output ports of the *CHAV 1* and *CHAV 2* components. The correspondence between

3.4 Derivation of System-Level Timing Specifications

Tab. 3.2: Mapping of events and signals to ports

<i>Actor</i>	<i>Event/Signal</i>	<i>Port/Event</i>	<i>Type</i>	<i>Shortcut</i> ¹
CHAV	SUBSCRIBE			
	<i>A/B</i>	<i>SubscriptionTrigger</i>	$\rightsquigarrow$	<i>SubTrigger</i>
	<i>Subscription Request</i>	<i>SubscriptionRequest</i>	$\leftarrow$	<i>SubRequest</i>
	<i>Subscription Response</i>	<i>SubscriptionResponse</i>	$\rightarrow$	<i>SubResponse</i>
	Final node	<i>set(Mode,Subscribed)</i>	$\circlearrowleft$	–
	INITIATE AND EVALUATE			
	<i>Cooperation Rec.</i>	<i>CooperationRecommendation</i>	$\rightarrow$	<i>CoopRec</i>
	<i>Evaluation Results</i>	<i>EvaluationResults</i>	$\rightarrow$	<i>EvalResultsIn</i>
	<i>Evaluation Results</i>	<i>EvaluationResults</i>	$\leftarrow$	<i>EvalResultsOut</i>
	<i>D/E</i>	<i>set(Mode,EvalCompleted)</i>	$\circlearrowleft$	–
	DECIDE AND EXECUTE			
	<i>D/E</i>	<i>change(Mode,EvalCompleted)</i>	$\circlearrowleft$	–
	Final nodes	<i>set(Mode,Subscribed)</i>	$\circlearrowleft$	–
	UNSUBSCRIBE			
	<i>F/G</i>	<i>UnsubscriptionTrigger</i>	$\rightsquigarrow$	<i>UnsubTrigger</i>
	<i>Unsubscription Request</i>	<i>UnsubscriptionRequest</i>	$\leftarrow$	<i>UnsubRequest</i>
	<i>Unsubscription Response</i>	<i>UnsubscriptionResponse</i>	$\rightarrow$	<i>UnsubResponse</i>
	Final node	<i>set(Mode,FreeFlow)</i>	$\circlearrowleft$	–
V2X	SUBSCRIBE			
	<i>Subscription Request</i>	<i>SubscriptionRequest</i>	$\rightarrow$	<i>SubRequestIn</i> ²
		<i>SubscriptionRequest</i>	$\leftarrow$	<i>SubRequestOut</i>
	<i>Subscription Response</i>	<i>SubscriptionResponse</i>	$\rightarrow$	<i>SubResponseIn</i>
		<i>SubscriptionResponse</i>	$\leftarrow$	<i>SubResponseOut</i> ²
	INITIATE AND EVALUATE			
	<i>Cooperation Rec.</i>	<i>CooperationRecommendation</i>	$\rightarrow$	<i>CoopRecIn</i>
		<i>CooperationRecommendation</i>	$\leftarrow$	<i>CoopRecOut</i> ²
	<i>Evaluation Results</i>	<i>EvaluationResults</i>	$\rightarrow$	<i>EvalResultsIn</i> ²
		<i>EvaluationResults</i>	$\leftarrow$	<i>EvalResultsOut</i> ²
	UNSUBSCRIBE			
	<i>Unsubscription Request</i>	<i>UnsubscriptionRequest</i>	$\rightarrow$	<i>UnsubRequestIn</i> ²
		<i>UnsubscriptionRequest</i>	$\leftarrow$	<i>UnsubRequestOut</i>
	<i>Unsubscription Response</i>	<i>UnsubscriptionResponse</i>	$\rightarrow$	<i>UnsubResponseIn</i>
		<i>UnsubscriptionResponse</i>	$\leftarrow$	<i>UnsubResponseOut</i> ²
TMS	SUBSCRIBE			
	<i>Subscription Request</i>	<i>SubscriptionRequest</i>	$\rightarrow$	<i>SubRequest</i>
	<i>Subscription Response</i>	<i>SubscriptionResponse</i>	$\leftarrow$	<i>SubResponse</i>
	INITIATE AND EVALUATE			
	<i>C</i>	<i>RecommendationTrigger</i>	$\rightsquigarrow$	<i>RecTrigger</i>
	<i>Cooperation Rec.</i>	<i>CooperationRecommendation</i>	$\leftarrow$	<i>CoopRec</i>
	UNSUBSCRIBE			
	<i>Unsubscription Request</i>	<i>UnsubscriptionRequest</i>	$\rightarrow$	<i>UnsubRequest</i>
	<i>Unsubscription Response</i>	<i>UnsubscriptionResponse</i>	$\leftarrow$	<i>UnsubResponse</i>

$\rightsquigarrow$ = Virtual $\rightarrow$ = Input $\leftarrow$ = Output $\circlearrowleft$ = Internal ¹ for use in specifications

² All ports related to individual CHAVs must be instantiated once for each CHAV.

3 Consistent Specification of Top-Level Timing Requirements

Tab. 3.3: Specification $\mathcal{C}_{chav}^{base}$ of the CHAV 1/2 components

A	True.	1
G	// Send “CAM” events every 500 ms CAM occurs every 500 ms with offset $[0, 500]$ ms.	2 3

Tab. 3.4: Specification $\mathcal{C}_{v2x}^{base}$ of the V2X component

A	CAMIn1 occurs every $(0, \infty)$ ms with offset. CAMIn2 occurs every $(0, \infty)$ ms with offset.	1 2
G	// Transmit “CAM” events within 200 ms Reaction(CAMIn1, CAMOut1) within $(0, 200]$ ms. Reaction(CAMIn2, CAMOut2) within $(0, 200]$ ms.	3 4 5

TCAD events and signals on the one hand and the ports and events from the functional architecture on the other hand also serves as a basis for deriving information about the required signals that connect the input and output ports of the top-level components.

Note that the definition of virtual events $\sigma^{virt} \in \Sigma$ based on event specifications (see Section A.4.2) in step 1. c) may require the formulation of additional assumptions about the occurrence of events. In the context of the CLTA application example, we assume that the A/B event is triggered either by (1) the detection of an intersection ahead by one of the vehicle’s front cameras, (2) the localization of the vehicle in a digital map that indicates the position of the intersection using a lidar-based *Simultaneous Localization and Mapping* (SLAM) [195] approach, or (3) the localization of the vehicle in a digital map based on GNSS positioning. The C event is triggered either by the detection of a vehicle with one of the infrastructure cameras or the reception of a *Cooperative Awareness Message* (CAM) [214] from one of the vehicles which indicates that the sending vehicle is approaching the intersection. Similar to the A/B event, the F/G event is triggered based on the same inputs, but indicates that the vehicle is leaving the intersection area.

Formulating additional assumptions to define virtual events may require introducing additional specifications if some of the events are provided by other CSs. In the CLTA example, the virtual C event depends on receiving a CAM event sent by one of the CHAVs. In order to make the occurrence and propagation of such events explicit, we define two specifications $\mathcal{C}_{chav}^{base}$ and $\mathcal{C}_{v2x}^{base}$, which are listed in Tables 3.3 and 3.4.

3.4.3 Formulation of System-Level Specifications

Finally, system-level specifications can be derived from the partitioned activity models. For this purpose, all actions are translated into modes (i.e., internal states) of the individual CSs, which take on the previously specified roles. The system-local action sequences resulting from the partitioning of TCADs, as well as the annotated timing constraints and resource limitations, can then be automatically translated into MTSL specifications, which form the basis for all subsequent design activities on (sub-)system level. As a prerequisite, each $pTCAD \in \mathbb{P}_{TCAD}$ must comply with the following assumptions:

3.4 Derivation of System-Level Timing Specifications

1. All actions $a \in A \setminus (A_{event} \cup A_{signal})$ are annotated with a duration constraint, i.e., $dur(a) = (lb, ub)$ with $lb, ub \in \mathbb{T}$ and $ub > 0$. Note that all duration-constrained control flows have been translated into virtual actions $A_{virt} \subseteq A$.
2. All time values are given in the same time unit.

Mapping of Actions to System-Level Modes

As indicated in the previous sections, the general idea of the translation approach is to map all actions of the partitioned TCADs to corresponding modes of the actors' top-level components. The individual sets of valid transitions between these modes are derived from the sets of control flows and control nodes, which will be done in a second step.

In order to establish a well-defined mapping between the actions of each pTCAD and the modes of the actor's top-level component $c_{cs} = actor(p)$, we define a *mode function* $m : A \setminus (A_{event} \cup A_{signal}) \rightarrow M_{c_{cs}}, a \mapsto m$ in terms of a fourth translation procedure. The following steps must be completed for all $TCAD \in \mathbb{D}_{TCAM}$ of a given TCAM:

1. For each pTCAD $\in \mathbb{P}_{TCAD}$:
 - a) Determine the actor's top-level component $c_{cs} = actor(p) \in \mathcal{C}$.
 - b) For each action $a \in A \setminus (A_{event} \cup A_{signal})$:
 - If a has an influence on the timing behavior of c_{cs} , i.e., $\exists a' \in \mathcal{A} \setminus (A_{event} \cup A_{signal}) : \Xi(a) = p \wedge dur(a) \neq dur(a')$:
 - If a is a concurrent action, i.e., $\exists f \in F, j \in J : f \rightsquigarrow a \rightsquigarrow j$: Map a to a shared mode $m(a_1 \parallel \dots \parallel a_n)$ of c_{cs} that represents $a_1, \dots, a_n \in A_{con}$ with $A_{con} = \{a \in A \setminus (A_{event} \cup A_{signal}) \mid f \rightsquigarrow a \rightsquigarrow j\}$ and $n \in \mathbb{N}$.
 - Otherwise: Map a to a new mode $m_a = m(a)$ of c_{cs} .
 - Otherwise: Skip mapping of a .
2. Define a shortcut for each resulting mode.

The resulting mapping of actions to system-level modes for the CLTA example is shown in Table 3.5. Note that some of the modes have been combined for easier visualization based on the selections defined in the TCADs. In addition, all pre- and postconditions defined in Section 3.4.2 are listed as part of the mapping. Again, CHAV 1 and CHAV 2 are collectively referred to as CHAV, indicating that they have the same set of modes.

Translation of Activity Models into Timing Specifications

The representation of actions by corresponding modes of the actors' top-level components serves as the basis for the final translation step. The goal of this step is to derive a set of timing specifications on system-level for each TCAD, i.e., an MTSL timing contract $\mathcal{C}_{c_{cs}}^{act}$ for each activity/actor combination that specifies real-time requirements on the behavior of c_{cs} . In order to translate the (partitioned) TCADs into corresponding timing

3 Consistent Specification of Top-Level Timing Requirements

Tab. 3.5: Mapping of actions to system-level modes

<i>Actor</i>	<i>Phase/Action</i>	<i>Mode</i>	<i>Shortcut</i> ¹
CHAV	SUBSCRIBE		
	before(<i>A/B</i>)	<u>FreeFlow</u> ²	FreeFlow
	<i>Subscribe</i>	Subscribe	Subscribe
	after(<i>Subscribe</i>)	Subscribed	Subscribed
	INITIATE AND EVALUATE		
	before(<i>Cooperation Rec.</i>)	<u>Subscribed</u>	Subscribed
	<i>Receive Cooperation Rec.</i>	ReceiveCooperationRec.	ReceiveCoopRec
	<i>Evaluate Proposed Solutions</i>	EvaluateProposedSolutions	EvaluateSolutions
	<i>Exchange Evaluation Results</i>	ExchangeEvaluationResults	ExchangeResults
	on(<i>D/E</i>)	EvaluationCompleted	EvalCompleted
	after(<i>Initiate and Evaluate</i>)	Subscribed	Subscribed
	DECIDE AND EXECUTE		
	on(<i>D/E</i>)	EvaluationCompleted	EvalCompleted
	<i>Start Maneuver Execution</i>	StartManeuverExecution	StartExec
	<i>Wait for 100 ms</i>	Wait	Wait
	<i>Evaluate Maneuver Execution</i>	EvaluateManeuverExecution	EvaluateExec
	<i>Abort Maneuver Execution</i>	AbortManeuverExecution	AbortExec
	<i>Check Maneuver Completion</i>	CheckManeuverCompletion	CheckComp
	after(<i>Decide and Execute</i>)	Subscribed	Subscribed
	UNSUBSCRIBE		
	before(<i>F/G</i>)	<u>Subscribed</u>	Subscribed
	<i>Unsubscribe</i>	Unsubscribe	Unsubscribe
	after(<i>Unsubscribe</i>)	FreeFlow	FreeFlow
TMS	SUBSCRIBE		
	before(<i>Subscription Request</i>)	<u>Idle</u> ²	Idle
	<i>Handle Subscription</i>	HandleSubscription	HandleSub
	after(<i>Subscription Response</i>)	Idle	Idle
	INITIATE AND EVALUATE		
	before(<i>C</i>)	<u>Idle</u>	Idle
	<i>Detect Conflict Situations</i>	DetectConflictSituations	DetectConflicts
	<i>Initiate Cooperation</i>	InitiateCooperation	InitiateCoop
	after(<i>Cooperation Rec.</i>)	Idle	Idle
	UNSUBSCRIBE		
	before(<i>Unsubscription Request</i>)	<u>Idle</u>	Idle
	<i>Handle Unsubscription</i>	HandleUnsubscription	HandleUnsub
	after(<i>Unsubscription Response</i>)	Idle	Idle

¹ for use in specifications ² initial mode

3.4 Derivation of System-Level Timing Specifications

specifications, we follow a pattern-based approach, i.e., we map each instance from a set of predefined pTCAD patterns to a set of corresponding MTSL specification patterns.

As one of the most important MTSL specification patterns, which is also frequently used in the following, the **Reaction** pattern specifies a latency between two event specifications (see Appendix A). Since reactions cannot be timeless, we have to introduce minimum delays to be able to translate control flows $e \rightarrow a$ between (activating) events $e \in A_{event}$ and actions $a \in A \setminus (A_{event} \cup A_{signal})$ into valid **Reaction** patterns. Note that the introduction of delays may also affect subsequent time intervals, in particular the duration of action a . To properly account for these “context-dependent” latencies, we first define a *minimum delay constant* $t_\delta \in \mathbb{T} = \mathbb{R}_0^+$ with $t_\delta > 0$, which represents the smallest possible time delay that can be specified by using MTSL specification patterns. For most practical applications, this constant has a value of 1. However, the interpretation of this value also depends on the chosen (metric) time unit, which will be abstracted in the following. Next, we define an *adjusted duration function* $dur^* : \mathcal{A} \cup \mathcal{C} \rightarrow \mathbb{T} \times \mathbb{T}$ based on t_δ :

$$dur^*(a) = \begin{cases} (0, ub - t_\delta) & \text{if } dur(a) = (0, ub) \text{ with } 0 < ub < \infty \\ & \wedge \exists e \in A_{event} : e \rightarrow a \\ (lb - t_\delta, ub - t_\delta) & \text{if } dur(a) = (lb, ub) \text{ with } 0 < lb \leq ub < \infty \\ & \wedge \exists e \in A_{event} : e \rightarrow a \\ dur(a) & \text{otherwise} \end{cases} \quad (3.34)$$

Finally, we introduce a *latency function* $l : \mathcal{A} \rightarrow \mathcal{I}$, where $\mathcal{I}$ is the set of time intervals, i.e., the set of intervals of the form (lb, ub) (“open”), $[lb, ub]$ (“closed”), $(lb, ub]$ (“left-open”), or $[lb, ub)$ (“right-open”) with $lb, ub \in \mathbb{T}$ and $lb \leq ub$. l is defined as

$$l(a) = \begin{cases} (0, ub] & \text{if } dur^*(a) = (lb, ub) \text{ with } lb = 0 \text{ and } 0 < ub < \infty \\ [lb, ub] & \text{if } dur^*(a) = (lb, ub) \text{ with } 0 < lb \leq ub < \infty \\ [lb, \infty) & \text{if } dur^*(a) = (lb, ub) \text{ with } 0 < lb < \infty \text{ and } ub = \infty \\ (0, \infty) & \text{otherwise} \end{cases} \quad (3.35)$$

and establishes a correspondence between the duration of actions and the (metric) time intervals used as part of the resulting MTSL timing specifications.

Based on these definitions, we introduce the fifth and final translation procedure in terms of a set of pattern-based translation rules. In most cases, the *basic translation rules* R-1 to R-9 take an (extended) action sequence as input, each of which is translated into a set of **Reaction** patterns. Note that the original definition of action sequences only allows actions as first and last elements. For the definition of some of the translation rules, however, action sequences that end with a final node $n_{final} \in \mathcal{N}_{final}$ must also be considered. We circumvent this problem by omitting conditions 1 and 2 and denote the resulting class of sequences as *extended action sequences*. In addition to the basic translation rules, we define *shortcut translation rules* S-1 to S-3 that combine two or more basic rules to enable more efficient translation and improve the readability of the MTSL specifications. Note that all shortcuts contain simplifications that may combine individual actions and thus hide them in the resulting system-level specifications.

3 Consistent Specification of Top-Level Timing Requirements

As a final prerequisite, we introduce two operators to describe the translation of complex control flows. On the one hand, the “|” operator refers to *alternative control flows*, i.e., alternative actions $a_1 | a_2$ with $a_1, a_2 \in \mathcal{A} \setminus (\mathcal{A}_{event} \cup \mathcal{A}_{signal})$, which result from a decision node $d \in \mathcal{D}$ with $d \rightarrow a_1$ and $d \rightarrow a_2$. On the other hand, the “||” operator represents *concurrent control flows* resulting from the use of fork nodes $f \in \mathcal{F}$, i.e., concurrent actions $a_1 || a_2$ with $f \rightarrow a_1$ and $f \rightarrow a_2$. Based on these notations, the translation procedure is defined in terms of an algorithm that applies a set of translation rules. The following steps must be completed for all $\text{TCAD} \in \mathbb{D}_{\text{TCAM}}$ of a given TCAM:

1. For each $\text{pTCAD} \in \mathbb{P}_{\text{TCAD}}$:

- a) Determine the actor’s top-level component $c_{cs} = \text{actor}(p) \in \mathcal{C}$.
- b) Create a new timing specification $\mathcal{C}_{c_{cs}}^{act}$ for c_{cs} or reuse an existing one.
- c) Define a variable $m_{c_{cs}}^{prev} := m_{c_{cs}}^{\triangleleft}$ that is used to store the previous mode of c_{cs} during the translation process, if no such variable already exists. Automatically update $m_{c_{cs}}^{prev}$ every time a $\text{set}(\text{Mode}, \dots)$ event is used.
- d) Traverse all control flows along the flow of tokens and translate each of the following pTCAD patterns into a set of corresponding MTSL specification patterns. Add each of the resulting patterns to the guarantee $\Phi_{c_{cs}}^G$ of the previously created (or selected) timing specification $\mathcal{C}_{c_{cs}}^{act}$ of c_{cs} :

i. **R-1: Activating Event $\rightarrow$ Action**

Translate each action sequence $as = \langle e \rightarrow a \rangle$ with $e \in A_{event}^{\uparrow}$ and $a \in A \setminus (\mathcal{A}_{event} \cup \mathcal{A}_{signal})$ as follows:

$$\begin{aligned} & \langle e \rightarrow a \rangle \\ & \Downarrow \\ & \text{Reaction}(pe(e), \text{set}(\text{Mode}, m(a))) \text{ within } (0, t_{\delta}] \text{ in mode } m_{c_{cs}}^{prev}. \end{aligned} \tag{3.36}$$

ii. **R-2: Action₁ $\rightarrow$ Action₂**

Translate each action sequence $as = \langle a_1 \rightarrow a_2 \rangle$ with $a_1, a_2 \in A \setminus (\mathcal{A}_{event} \cup \mathcal{A}_{signal})$ as follows:

$$\begin{aligned} & \langle a_1 \rightarrow a_2 \rangle \\ & \Downarrow \\ & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), \text{set}(\text{Mode}, m(a_2))) \text{ within} \\ & \quad l(a_1) \text{ in mode } m(a_1). \end{aligned} \tag{3.37}$$

iii. **R-3: Action $\rightarrow$ Terminating Signal**

Translate each action sequence $as = \langle a \rightarrow s \rangle$ with $a \in A \setminus (\mathcal{A}_{event} \cup \mathcal{A}_{signal})$ and $s \in A_{signal}^{\downarrow}$ as follows:

3.4 Derivation of System-Level Timing Specifications

- If $\exists \text{pTCAD}' \in \mathbb{P}_{\text{TCAD}} : \text{pTCAD} \neq \text{pTCAD}' \wedge p = p'$ that has not yet been translated into a set of MTSL specification patterns:

$$\begin{aligned} & \langle a \rightarrow s \rangle \\ & \Downarrow \\ & \text{Reaction}(\text{change}(\text{Mode}, m(a)), pe(s)) \text{ within } l(a) \text{ in mode } m(a). \end{aligned} \quad (3.38)$$

- If $\nexists \text{pTCAD}' \in \mathbb{P}_{\text{TCAD}} : \text{pTCAD} \neq \text{pTCAD}' \wedge p = p'$ that has not yet been translated into a set of MTSL specification patterns:

$$\begin{aligned} & \langle a \rightarrow s \rangle \\ & \Downarrow \\ & \text{Reaction}(\text{change}(\text{Mode}, m(a)), \{pe(s), \text{set}(\text{Mode}, m_{c_{cs}}^\triangleright)\}) \\ & \text{ within } l(a) \text{ in mode } m(a). \end{aligned} \quad (3.39)$$

iv. R-4: Action $\rightarrow$ Final Node

Translate each (extended) action sequence $as' = \langle a \rightarrow n_{final} \rangle$ with $a \in A \setminus (A_{event} \cup A_{signal})$ and $n_{final} \in N_{final}$ as follows:

$$\begin{aligned} & \langle a \rightarrow n_{final} \rangle \\ & \Downarrow \\ & \text{Reaction}(\text{change}(\text{Mode}, m(a)), pe(n_{final})) \text{ within } l(a) \text{ in mode } m(a). \end{aligned} \quad (3.40)$$

v. R-5: Event/Action $\rightarrow$ Decision $\rightarrow$ (Action₁ | Action₂)

Translate each set of action sequences $AS = \{\langle d \rightarrow a_1 \rangle, \langle d \rightarrow a_2 \rangle\}$ with $d \in D$ and $a_1, a_2 \in A \setminus (A_{event} \cup A_{signal})$ as follows:

- If $\exists e \in A_{event}^\uparrow : e \rightarrow d$ and the activity is not directly called by another activity, i.e., $\nexists \text{TCAD}_{pre} \in \mathbb{D}_{\text{TCAM}} : \text{TCAD}_{pre} \leadsto \text{TCAD}$:

$$\begin{aligned} & \{\langle e \rightarrow d \rightarrow a_1 \rangle, \langle e \rightarrow d \rightarrow a_2 \rangle\} \\ & \Downarrow \\ & \text{Reaction}(pe(e), \text{set}(\text{Mode}, \{m(a_1), m(a_2)\})) \text{ within } (0, t_\delta] \\ & \text{ in mode } m_{c_{cs}}^{prev}. \end{aligned} \quad (3.41)$$

- If $\exists e \in A_{event}^\uparrow : e \rightarrow d$ and the activity is directly called by another activity, i.e., $\exists \text{TCAD}_{pre} \in \mathbb{D}_{\text{TCAM}} : \text{TCAD}_{pre} \leadsto \text{TCAD}$:

$$\begin{aligned} & \{\langle e \rightarrow d \rightarrow a_1 \rangle, \langle e \rightarrow d \rightarrow a_2 \rangle\} \\ & \Downarrow \\ & \text{Reaction}(pe(e), \text{set}(\text{Mode}, \{m(a_1), m(a_2)\})) \text{ within } (0, t_\delta] \\ & \text{ in mode } m_{c_{cs}}^\triangleright. \end{aligned} \quad (3.42)$$

- If $\exists a \in A \setminus (A_{event} \cup A_{signal}) : a \rightarrow d$:

$$\begin{aligned}
 & \{ \langle a \rightarrow d \rightarrow a_1 \rangle, \langle a \rightarrow d \rightarrow a_2 \rangle \} \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a)), \text{set}(\text{Mode}, \{m(a_1), m(a_2)\})) \\
 & \quad \text{within } l(a) \text{ in mode } m(a).
 \end{aligned} \tag{3.43}$$

Note that the translations can also be applied to a set of (extended) action sequences $AS' = \{ \langle d \rightarrow a_1 \rangle, \langle d \rightarrow s/n_{final} \rangle \}$, one of which ends with a terminating signal $s \in A_{signal}^\downarrow \setminus A_{signal}^{\leftrightarrow}$ or a final node $n_{final} \in N_{final}$. In this case, $\text{set}(\text{Mode}, \{m(a_1), m(a_2)\})$ is replaced by $\text{set}(\text{Mode}, \{m(a_1), m_{cs}^\triangleright\})$.

vi. **R-6: (Action₁ | ... | Action_n) → Merge → Action/Signal**

Translate each set of action sequences $AS = \{ \langle a_1 \rightarrow m \rangle, \dots, \langle a_n \rightarrow m \rangle \}$ with $a_1, \dots, a_n \in A \setminus (A_{event} \cup A_{signal})$, $n \in \mathbb{N}$, and $m \in M$ as follows:

- If $\exists a \in A \setminus (A_{event} \cup A_{signal}) : m \rightarrow a$:

$$\begin{aligned}
 & \{ \langle a_1 \rightarrow m \rightarrow a \rangle, \dots, \langle a_n \rightarrow m \rightarrow a \rangle \} \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), \text{set}(\text{Mode}, m(a))) \text{ within} \\
 & \quad l(a_1) \text{ in mode } m(a_1).
 \end{aligned} \tag{3.44}$$

⋮

$$\begin{aligned}
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_n)), \text{set}(\text{Mode}, m(a))) \text{ within} \\
 & \quad l(a_n) \text{ in mode } m(a_n).
 \end{aligned}$$

- If $\exists s \in A_{signal}^\downarrow : m \rightarrow s$:

$$\begin{aligned}
 & \{ \langle a_1 \rightarrow m \rightarrow s \rangle, \dots, \langle a_n \rightarrow m \rightarrow s \rangle \} \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), pe(s)) \text{ within } l(a_1) \text{ in} \\
 & \quad \text{mode } m(a_1).
 \end{aligned} \tag{3.45}$$

⋮

$$\begin{aligned}
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_n)), pe(s)) \text{ within } l(a_n) \text{ in} \\
 & \quad \text{mode } m(a_n).
 \end{aligned}$$

vii. **R-7: Event/Action → Fork → (Action₁ || ... || Action₂)**

Translate each set of action sequences $AS = \{ \langle f \rightarrow a_1 \rangle, \dots, \langle f \rightarrow a_n \rangle \}$ with $f \in F$, $a_1, \dots, a_n \in A \setminus (A_{event} \cup A_{signal})$, and $n \in \mathbb{N}$ as follows:

- If $\exists e \in A_{event}^\uparrow : e \rightarrow f$:

$$\begin{aligned}
 & \{ \langle e \rightarrow f \rightarrow a_1 \rangle, \dots, \langle e \rightarrow f \rightarrow a_n \rangle \} \\
 & \quad \Downarrow \\
 & \text{Reaction}(pe(e), \text{set}(\text{Mode}, m(a_1) \parallel \dots \parallel a_n)) \text{ within } (0, t_\delta] \\
 & \quad \text{in mode } m_{cs}^{prev}.
 \end{aligned} \tag{3.46}$$

3.4 Derivation of System-Level Timing Specifications

- If $\exists a \in A \setminus (A_{event} \cup A_{signal}) : a \rightarrow f$:

$$\begin{aligned}
 & \{\langle a \rightarrow f \rightarrow a_1 \rangle, \dots, \langle a \rightarrow f \rightarrow a_n \rangle\} \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a)), \text{set}(\text{Mode}, m(a_1 \parallel \dots \parallel a_n))) \\
 & \quad \text{within } l(a) \text{ in mode } m(a).
 \end{aligned} \tag{3.47}$$

For each concurrent action $a_{con} \in A \setminus (A_{event} \cup A_{signal})$ that sends a signal s , i.e., $a_{con} \rightarrow s$, with $s \in A_{signal} \setminus A_{signal}^\downarrow$ (**R-7a**):

$$\begin{aligned}
 & \langle \dots \rightarrow f \rightarrow a_{con} \rightarrow s \rangle \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_1 \parallel \dots \parallel a_n)), pe(s)) \text{ within} \\
 & \quad l(a_{con}) \text{ in mode } m(a_1 \parallel \dots \parallel a_n).
 \end{aligned} \tag{3.48}$$

viii. **R-8: (Action₁ $\parallel \dots \parallel$ Action_n) $\rightarrow$ Join $\rightarrow$ Action/Signal**

Translate each set of action sequences $AS = \{\langle a_1 \rightarrow j \rangle, \dots, \langle a_n \rightarrow j \rangle\}$ with $a_1, \dots, a_n \in A \setminus (A_{event} \cup A_{signal})$, $n \in \mathbb{N}$, and $j \in J$ as follows:

- If $\exists a \in A \setminus (A_{event} \cup A_{signal}) : j \rightarrow a$:

$$\begin{aligned}
 & \{\langle a_1 \rightarrow j \rightarrow a \rangle, \dots, \langle a_n \rightarrow j \rightarrow a \rangle\} \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_1 \parallel \dots \parallel a_n)), \text{set}(\text{Mode}, m(a))) \\
 & \quad \text{within } [\min(\min(l(a_1)), \dots, \min(l(a_n))), \infty) \text{ in mode} \\
 & \quad m(a_1 \parallel \dots \parallel a_n).
 \end{aligned} \tag{3.49}$$

- If $\exists s \in A_{signal}^\downarrow : j \rightarrow s$:

$$\begin{aligned}
 & \{\langle a_1 \rightarrow j \rightarrow s \rangle, \dots, \langle a_n \rightarrow j \rightarrow s \rangle\} \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_1 \parallel \dots \parallel a_n)), pe(s)) \text{ within} \\
 & \quad [\min(\min(l(a_1)), \dots, \min(l(a_n))), \infty) \text{ in mode} \\
 & \quad m(a_1 \parallel \dots \parallel a_n).
 \end{aligned} \tag{3.50}$$

Note that the translations from Equations 3.49 and 3.50 can be omitted, if $\forall a \in \{a_1, \dots, a_n\} : dur(a) = (0, ub)$ with $ub \in \mathbb{T}$ applies.

For each concurrent action $a_{con} \in A \setminus (A_{event} \cup A_{signal})$ that receives an event e , i.e., $e \rightarrow a_{con}$, with $e \in A_{event} \setminus A_{event}^\uparrow$ (**R-8a**):

- If $\exists a \in A \setminus (A_{event} \cup A_{signal}) : j \rightarrow a$:

$$\begin{aligned}
 & \{\langle \dots \rightarrow a_{con} \rightarrow j \rightarrow a \rangle, \langle e \rightarrow a_{con} \rightarrow j \rightarrow a \rangle\} \\
 & \quad \Downarrow \\
 & \text{Reaction}(pe(e), \text{set}(\text{Mode}, m(a))) \text{ within } l(a_{con}) \text{ in mode} \\
 & \quad m(a_1 \parallel \dots \parallel a_n).
 \end{aligned} \tag{3.51}$$

- If $\exists s \in A_{signal}^\downarrow : j \rightarrow s$:

$$\begin{aligned}
 & \{ \langle \dots \rightarrow a_{con} \rightarrow j \rightarrow s \rangle, \langle e \rightarrow a_{con} \rightarrow j \rightarrow s \rangle \} \\
 & \quad \Downarrow \\
 & \text{Reaction}(pe(e), pe(s)) \text{ within } l(a_{con}) \text{ in mode} \\
 & \quad m(a_1 \parallel \dots \parallel a_n).
 \end{aligned} \tag{3.52}$$

ix. **R-9: Action-to-Action Latency Constraint**

Translate each action-to-action latency constraint $\phi_{lat} \in \Phi'_{lat}$ with $\phi_{lat} = (as^*(a_1, a_n), lb, ub)$, $n \in \mathbb{N}$, and $a_1, a_n \in A$ as follows:

- If $\exists a \in A \setminus (A_{event} \cup A_{signal}) : a_n \rightarrow a$:

$$\begin{aligned}
 & (as^*(a_1, a_n), lb, ub) \wedge a_n \rightarrow a \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), \text{set}(\text{Mode}, m(a))) \text{ within} \\
 & \quad I(lb, ub).
 \end{aligned} \tag{3.53}$$

- If $\exists s \in A_{signal}^\downarrow : a_n \rightarrow s$:

$$\begin{aligned}
 & (as^*(a_1, a_n), lb, ub) \wedge a_n \rightarrow s \\
 & \quad \Downarrow \\
 & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), pe(s)) \text{ within } I(lb, ub).
 \end{aligned} \tag{3.54}$$

Note that $I : \mathbb{T} \times \mathbb{T} \rightarrow \mathcal{I}$ is defined analogously to Equation 3.35:

$$I(lb, ub) = \begin{cases} (0, ub] & \text{if } lb = 0 \text{ and } 0 < ub < \infty \\ [lb, ub] & \text{if } 0 < lb \leq ub < \infty \\ [lb, \infty) & \text{if } 0 < lb < \infty \text{ and } ub = \infty \\ (0, \infty) & \text{otherwise} \end{cases} \tag{3.55}$$

In addition to the previously defined *basic translation rules*, we introduce the following *shortcut translation rules*, which combine two or more basic rules:

i. **S-1: Activating Event $\rightarrow$ Action $\rightarrow$ Terminating Signal**

Translate each action sequence $as = \langle e \rightarrow a \rightarrow s \rangle$ with $e \in A_{event}^\uparrow$, $a \in A \setminus (A_{event} \cup A_{signal})$, and $s \in A_{signal}^\downarrow$ as follows:

$$\begin{aligned}
 & \langle e \rightarrow a \rightarrow s \rangle \\
 & \quad \Downarrow \\
 & \text{Reaction}(pe(e), pe(s)) \text{ within } l(a).
 \end{aligned} \tag{3.56}$$

Note that this shortcut is a combination of the “*Activating Event $\rightarrow$ Action*” (R-1) and “*Action $\rightarrow$ Terminating Signal*” (R-3) translation rules. Since

3.4 Derivation of System-Level Timing Specifications

it introduces a mode-independent specification pattern, all actions of c_{cs} must have the same duration constraint, i.e.,

$$\nexists a' \in \mathcal{A} : \Xi(a') = p \wedge \text{dur}(a') \neq \text{dur}(a). \quad (3.57)$$

ii. S-2: Activating Event $\rightarrow$ Action $\rightarrow$ Final Node

Translate each (extended) action sequence $as' = \langle e \rightarrow a \rightarrow n_{final} \rangle$ with $e \in A_{event}^\uparrow$, $a \in A \setminus (A_{event} \cup A_{signal})$, and $n_{final} \in N_{final}$ as follows:

$$\begin{aligned} & \langle e \rightarrow a \rightarrow n_{final} \rangle \\ & \Downarrow \\ & \text{Reaction}(pe(e), pe(n_{final})) \text{ within } l(a) \text{ in mode } m_{c_{cs}}^{prev}. \end{aligned} \quad (3.58)$$

Note that this shortcut is a combination of the “*Activating Event $\rightarrow$ Action*” (R-1) and “*Action $\rightarrow$ Final Node*” (R-4) translation rules.

iii. S-3: Action₁ $\rightarrow \dots \rightarrow$ Action_n $\rightarrow$ Action/Signal

Translate each action sequence $as = \langle a_1 \rightarrow \dots \rightarrow a_n \rangle$ with $a_1, \dots, a_n \in A \setminus (A_{event} \cup A_{signal})$ and $n \in \mathbb{N}$ as follows:

- If $\exists a \in A \setminus (A_{event} \cup A_{signal}) : a_n \rightarrow a$:

$$\begin{aligned} & \langle a_1 \rightarrow \dots \rightarrow a_n \rightarrow a \rangle \\ & \Downarrow \\ & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), \text{set}(\text{Mode}, m(a))) \text{ within} \\ & \quad l(a_1) + \dots + l(a_n) \text{ in mode } \{m(a_1), \dots, m(a_n)\}. \end{aligned} \quad (3.59)$$

- If $\exists s \in A_{signal}^\downarrow : a_n \rightarrow s$ and $\exists p\text{TCAD}' \in \mathbb{P}_{\text{TCAD}} : p\text{TCAD} \neq p\text{TCAD}' \wedge p = p'$ that has not yet been translated:

$$\begin{aligned} & \langle a_1 \rightarrow \dots \rightarrow a_n \rightarrow s \rangle \\ & \Downarrow \\ & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), pe(s)) \text{ within} \\ & \quad l(a_1) + \dots + l(a_n) \text{ in mode } \{m(a_1), \dots, m(a_n)\}. \end{aligned} \quad (3.60)$$

- If $\exists s \in A_{signal}^\downarrow : a_n \rightarrow s$ and $\nexists p\text{TCAD}' \in \mathbb{P}_{\text{TCAD}} : p\text{TCAD} \neq p\text{TCAD}' \wedge p = p'$ that has not yet been translated:

$$\begin{aligned} & \langle a_1 \rightarrow \dots \rightarrow a_n \rightarrow s \rangle \\ & \Downarrow \\ & \text{Reaction}(\text{change}(\text{Mode}, m(a_1)), \{pe(s), \text{set}(\text{Mode}, m_{c_{cs}}^\triangleright)\}) \\ & \quad \text{within } l(a_1) + \dots + l(a_n) \text{ in mode } \{m(a_1), \dots, m(a_n)\}. \end{aligned} \quad (3.61)$$

Note that this shortcut is a combination of the “*Action₁ $\rightarrow$ Action₂*” (R-2) and the “*Action $\rightarrow$ Terminating Signal*” (R-3) translation rules.

3 Consistent Specification of Top-Level Timing Requirements

- e) Remove all duplicate specification patterns from $\Phi_{c_{cs}}^G$.
- 2. For each timing specification $\mathcal{C}_{c_{cs}}^{act}$ derived in the previous steps that specifies the timing behavior of the actor's top-level component c_{cs} in the *act* activity:
 - a) Define an MTSL specification pattern for each input port $p_i \in P_{in}^{virt}$ used to define virtual events $\sigma^{virt} \in \Sigma$ that specifies assumptions about the occurrence of events. Add each of the resulting patterns to the assumption $\Phi_{c_{cs}}^A$ of $\mathcal{C}_{c_{cs}}^{act}$.
 - b) Optional: Define an MTSL specification pattern for each other relevant input port $p_i \in P_{c_{cs}}^{in} \setminus P_{in}^{virt}$ that specifies assumptions about the occurrence of events. Add each of the resulting patterns to the assumption $\Phi_{c_{cs}}^A$ of $\mathcal{C}_{c_{cs}}^{act}$.

Almost all resulting MTSL specifications rely on mode-dependent versions of the original specification patterns, which are introduced in Section 4.2. Since all possible action sequences are represented by the set of modes of the actor's top-level component and the set of valid transitions, all *implicit precedence constraints* defined by control flows are preserved. In addition, all *explicit precedence constraints* are implemented by carefully selecting appropriate pre- and postconditions. All *duration* and *action-to-action latency constraints* are translated into corresponding **Reaction** patterns. The *uniqueness constraints* specified as part of the TCADs are implicitly satisfied, since all actions are mapped to individual modes, with the mode-dependent specifications for entering such modes being disabled by mode changes. With the exception of the shared modes, which are used to represent concurrent control flows, modes can thus be considered as *critical sections* [105]. This restriction could easily be lifted by explicitly defining self-loops to allow overlapping activations, or by removing some of the mode conditions that bind the specification patterns to a set of modes in which they must be fulfilled.

Unfortunately, it is not yet possible to translate *synchronization constraints* into system-level specifications, since there is no suitable MTSL specification pattern for such kind of requirements. The development of corresponding language extensions would be a desirable topic for future activities. Note that Section 6.5 presents an early version of a **Synchronization** pattern and provides first insights into relevant properties.

To our own surprise, it was not necessary to define a translation rule to handle the pTCAD pattern starting with an initial node $n_{init} \in N_{init}$. This is due to the fact that the initiation of an activity by n_{init} is represented by a *change*(Mode, $m(a)$) event, which is already covered by the mapping of events and signals presented in Section 3.4.2.

Complex combinations of two or more decision nodes are not supported by the current version of R-5, but could be included by defining additional translation rules. Combinations of two or more merge nodes are already covered by R-6, since subsequent merge nodes can be combined into a single merge node without affecting the semantics. The current version of the translation procedure also does not support nested fork/join pairs. This is due to the fact that concurrent actions $a_1, \dots, a_n \in A$ are mapped to a shared mode $m(a_1 \parallel \dots \parallel a_n)$, which does not allow any distinction between the individual actions or concurrent control flows. Note that these restrictions could be circumvented by encapsulating the set of concurrent control flows of each fork/join pair into a hierarchical call behavior action, which have not yet been integrated into the presented translation approach.

3.4 Derivation of System-Level Timing Specifications

Tab. 3.6: Specification $\mathcal{C}_{chav}^{sub}$ of the *CHAV 1/2* components

A	// Additional assumptions about the occurrence of input events	1
	FrontCameraFrameA occurs every 33 ms with offset [0, 33] ms and jitter 5 ms.	2
	FrontCameraFrameB occurs every 33 ms with offset [0, 33] ms and jitter 5 ms.	3
	LidarPointCloud occurs every 60 ms with offset [0, 60] ms and jitter 10 ms.	4
	Position occurs every 10 ms with offset [0, 10] ms and jitter 2 ms.	5
	SubResponse occurs every $(0, \infty)$ ms with offset.	6
G	// Switch to the “Subscribe” mode after a “SubTrigger” event has occurred	7
	Reaction(SubTrigger, set(Mode, Subscribe)) within $(0, 1]$ ms in mode FreeFlow.	8
	// Send the “SubRequest” event within 50 ms	9
	Reaction(change(Mode, Subscribe), SubRequest) within $(0, 49]$ ms in mode Subscribe.	10
	// Switch to the “Subscribed” mode within 100 ms after receiving a “SubResponse” event	11
	Reaction(SubResponse, set(Mode, Subscribed)) within $(0, 100]$ ms in mode Subscribe.	12

Tab. 3.7: Specification $\mathcal{C}_{v2x}^{sub}$ of the *V2X* component

A	SubRequestIn1 occurs every $(0, \infty)$ ms with offset.	1
	SubRequestIn2 occurs every $(0, \infty)$ ms with offset.	2
	SubResponseIn occurs every $(0, \infty)$ ms with offset.	3
G	// Transmit “SubRequest” events within 200 ms	4
	Reaction(SubRequestIn1, SubRequestOut) within $(0, 200]$ ms.	5
	Reaction(SubRequestIn2, SubRequestOut) within $(0, 200]$ ms.	6
	// Transmit “SubResponse” events within 200 ms	7
	Reaction(SubResponseIn, {SubResponseOut1, SubResponseOut2}) within $(0, 200]$ ms.	8

Application Example

The final translation procedure is used to translate each of the partitioned TCADs from $\mathbb{P}_{TCAD} = \mathbb{P}_{TCAD_{sub}} \cup \dots \cup \mathbb{P}_{TCAD_{unsub}}$ derived in Section 3.4.1 into an MTSL timing specification for each activity/actor combination. The translation incorporates the results of all previous translation steps and is based in particular on the definition of pre- and postconditions (see Table 3.1), the mapping of events and signals to ports (see Table 3.2) as well as the mapping of actions to system-level modes (see Table 3.5).

Subscription The specification $\mathcal{C}_{chav}^{sub}$ of the *CHAV 1/2* components resulting from the translation of $\mathbb{P}_{TCAD_{sub}}$ (see Figure 3.11) is presented in Table 3.6. At the beginning of the process, $\mathbb{P}_{TCAD_{sub}}^1$ is translated by using R-1 (line 8) and R-3 (line 10). Later, S-2 is applied to $\mathbb{P}_{TCAD_{sub}}^5$ to derive a corresponding MTSL pattern (line 12). All input ports used to define the virtual *SubTrigger* event are provided with appropriate assumptions (lines 2–5). In addition, *SubResponse* is specified as a sporadic event (line 6).

The timing requirements for the *V2X* component are listed in Table 3.7. The $\mathcal{C}_{v2x}^{sub}$ specification contains three **Reaction** patterns that result from the application of shortcut S-1 to $\mathbb{P}_{TCAD_{sub}}^2$ (lines 5–6) and $\mathbb{P}_{TCAD_{sub}}^4$ (line 8). Again, the input events, namely *SubRequestIn1/2*, and *SubResponseIn*, are specified as sporadic events (lines 1–3).

3 Consistent Specification of Top-Level Timing Requirements

Tab. 3.8: Specification $\mathcal{C}_{tms}^{sub}$ of the *TMS* component

A	<i>SubRequest</i> occurs every $(0, \infty)$ ms with offset.	1
G	// Switch to the “HandleSub” mode after receiving a “SubRequest” event	2
	Reaction (<i>SubRequest</i> , set(<i>Mode</i> , <i>HandleSub</i>)) within $(0, 1]$ ms in mode <i>Idle</i> .	3
	// Send the “SubResponse” event and switch to the “Idle” mode within 150 ms	4
	Reaction (change(<i>Mode</i> , <i>HandleSub</i>), { <i>SubResponse</i> , set(<i>Mode</i> , <i>Idle</i>)}) within $(0, 149]$ ms in mode <i>HandleSub</i> .	5

Tab. 3.9: Specification $\mathcal{C}_{tms}^{init}$ of the *TMS* component

A	// Additional assumptions about the occurrence of input events	1
	<i>InfraCameraFrameA</i> occurs every 33 ms with offset $[0, 33]$ ms and jitter 5 ms.	2
	<i>InfraCameraFrameB</i> occurs every 33 ms with offset $[0, 33]$ ms and jitter 5 ms.	3
	<i>CAM1</i> occurs every $(0, \infty)$ ms with offset.	4
	<i>CAM2</i> occurs every $(0, \infty)$ ms with offset.	5
G	// Switch to the “DetectConflicts” mode after a “RecTrigger” event has occurred	6
	Reaction (<i>RecTrigger</i> , set(<i>Mode</i> , <i>DetectConflicts</i>)) within $(0, 1]$ ms in mode <i>Idle</i> .	7
	// Switch to the “InitiateCoop” mode within 250 ms	8
	Reaction (change(<i>Mode</i> , <i>DetectConflicts</i>), set(<i>Mode</i> , <i>InitiateCoop</i>)) within $(0, 249]$ ms in mode <i>DetectConflicts</i> .	9
	// Send the “CoopRec” event and switch to the “Idle” mode within 450 ms	10
	Reaction (change(<i>Mode</i> , <i>InitiateCoop</i>), { <i>CoopRec</i> , set(<i>Mode</i> , <i>Idle</i>)}) within $(0, 450]$ ms in mode <i>InitiateCoop</i> .	11

The specification $\mathcal{C}_{tms}^{sub}$ of the *TMS* component that results from the translation of pTCAD_{sub}^3 is shown in Table 3.8. This time, translation rule R-1 (line 3) and shortcut S-3 (line 5) are applied to derive a set of corresponding MTSL specification patterns. The assumption specifies that *SubRequest* events occur sporadically (line 1).

Initiation and Evaluation The set of partitioned TCADs for the *Initiate* and *Evaluate* activity (see Figure 3.12) is translated into specifications for the *TMS*, *V2X*, and *CHAV 1/2* components. The corresponding specification $\mathcal{C}_{tms}^{init}$ of the *TMS* component is shown in Table 3.9. It results from the application of translation rules R-1 (line 7) and R-2 (line 9) as well as shortcut S-3 (line 11) to $\text{pTCAD}_{init}^1 \in \mathbb{P}_{\text{TCAD}_{init/eval}}$. Again, assumptions are specified for all input ports used to define the virtual *RecTrigger* event (lines 2–5).

The specification $\mathcal{C}_{v2x}^{init/eval}$ resulting from the repeated use of shortcut S-1 (lines 5, 7–8) can be found in Table 3.10. It results from the translation of pTCAD_{init}^2 and pTCAD_{eval}^2 . The guarantees are complemented by assumptions for relevant input ports (lines 1–3).

Finally, pTCAD_{eval}^1 is translated into the $\mathcal{C}_{chav}^{eval}$ specification, which is listed in Table 3.11. The partitioned TCAD is processed by applying translation rules R-1 (line 4), R-2 (line 6), R-7 (line 8), R-7a (line 10), R-8a (line 12), and R-9 (line 14). Note that the last three translation steps are required to handle the concurrent control flow in combination with the *Evaluation Results* events and signals. In addition, two assumptions about the occurrence of the *CoopRec* and *EvalResultsIn* events are defined (lines 1–2).

3.4 Derivation of System-Level Timing Specifications

Tab. 3.10: Specification $\mathcal{C}_{v2x}^{init/eval}$ of the *V2X* component

A	<i>CoopRecIn</i> occurs every $(0, \infty)$ ms with offset.	1
	<i>EvalResultsIn1</i> occurs every $(0, \infty)$ ms with offset.	2
	<i>EvalResultsIn2</i> occurs every $(0, \infty)$ ms with offset.	3
G	// Transmit “ <i>CoopRec</i> ” events within 300 ms	4
	<i>Reaction</i> (<i>CoopRecIn</i> , { <i>CoopRecOut1</i> , <i>CoopRecOut2</i> }) within $(0, 300]$ ms.	5
	// Transmit “ <i>EvalResults</i> ” events within 300 ms	6
	<i>Reaction</i> (<i>EvalResultsIn1</i> , <i>EvalResultsOut1</i>) within $(0, 300]$ ms.	7
	<i>Reaction</i> (<i>EvalResultsIn2</i> , <i>EvalResultsOut2</i>) within $(0, 300]$ ms.	8

Tab. 3.11: Specification $\mathcal{C}_{chav}^{eval}$ of the *CHAV 1/2* components

A	<i>CoopRec</i> occurs every $(0, \infty)$ ms with offset.	1
	<i>EvalResultsIn</i> occurs every $(0, \infty)$ ms with offset.	2
G	// Switch to the “ <i>ReceiveCoopRec</i> ” mode after receiving an “ <i>CoopRec</i> ” event	3
	<i>Reaction</i> (<i>CoopRec</i> , <i>set</i> (<i>Mode</i> , <i>ReceiveCoopRec</i>)) within $(0, 1]$ ms in mode <i>Subscribed</i> .	4
	// Switch to the “ <i>EvaluateSolutions</i> ” mode within 150 ms	5
	<i>Reaction</i> (<i>change</i> (<i>Mode</i> , <i>ReceiveCoopRec</i>), <i>set</i> (<i>Mode</i> , <i>EvaluateSolutions</i>)) within $(0, 149]$ ms in mode <i>ReceiveCoopRec</i> .	6
	// Switch to the “ <i>ExchangeResults</i> ” mode within 400 ms	7
	<i>Reaction</i> (<i>change</i> (<i>Mode</i> , <i>EvaluateSolutions</i>), <i>set</i> (<i>Mode</i> , <i>ExchangeResults</i>)) within $(0, 400]$ ms in mode <i>EvaluateSolutions</i> .	8
	// Send the “ <i>EvalResults</i> ” event to the cooperation partner within 100 ms	9
	<i>Reaction</i> (<i>change</i> (<i>Mode</i> , <i>ExchangeResults</i>), <i>EvalResultsOut</i>) within $(0, 100]$ ms in mode <i>ExchangeResults</i> .	10
	// Switch to the “ <i>EvalCompleted</i> ” mode within 50 ms after receiving an “ <i>EvalResults</i> ” event	11
	<i>Reaction</i> (<i>EvalResultsIn</i> , <i>set</i> (<i>Mode</i> , <i>EvalCompleted</i>)) within $(0, 50]$ ms in mode <i>ExchangeResults</i> .	12
	// Switch to the “ <i>Subscribed</i> ” mode if no “ <i>EvalResults</i> ” event is received within 1 s	13
	<i>Reaction</i> (<i>change</i> (<i>Mode</i> , <i>ExchangeResults</i>), <i>set</i> (<i>Mode</i> , <i>Subscribed</i>)) within 1 s.	14

Decision Making and Execution The specification $\mathcal{C}_{chav}^{dec/exec}$ of the *CHAV 1/2* components derived from $\mathbb{P}_{\text{TCAD}_{dec/exec}}$ (see Figure 3.11) is presented in Table 3.12. It results from the application of translation rules R-5 (line 3), R-2 (line 5), R-2 (line 7), R-5 (line 9), R-4 (line 11), and R-5 (line 13) to $\text{pTCAD}_{dec/exec}^1$. Since the activity does not involve communication, no assumptions about input events are required.

Unsubscription The set of partitioned TCADs for the *Unsubscribe* activity (see Figure 3.13) is translated into timing specifications for the *CHAV 1/2*, *V2X*, and *TMS* components. Note that the translations are analogous to those for the *Subscribe* activity, since both activities are modeled the same. The specification $\mathcal{C}_{chav}^{unsub}$ resulting from $\mathbb{P}_{\text{TCAD}_{unsub}}$ (see Figure 3.13) is shown in Table 3.13. In a first step, translation rules R-1 (line 8) and R-3 (line 10) are used to derive specification patterns for pTCAD_{unsub}^1 . Afterwards, shortcut S-2 is applied to pTCAD_{unsub}^5 (line 12). All input ports that are part of the virtual *UnsubTrigger* event are provided with appropriate assumptions (lines 2–5). In addition, the *UnsubResponse* input is specified as a sporadic event (line 6).

3 Consistent Specification of Top-Level Timing Requirements

Tab. 3.12: Specification $\mathcal{C}_{chav}^{dec/exec}$ of the *CHAV* 1/2 components

A	True.	1
G	// Switch to the “StartExec” mode (if a decision has been made to execute the maneuver) or to the “Subscribed” mode (otherwise)	2
	Reaction(change(Mode, EvalCompleted), set(Mode, {StartExec, Subscribed})) within (0, 1] ms in mode EvalCompleted.	3
	// Switch to the “Wait” mode within 50 ms	4
	Reaction(change(Mode, StartExec), set(Mode, Wait)) within (0, 49] ms in mode StartExec.	5
	// Switch to the “EvalExec” mode within 100 ms	6
	Reaction(change(Mode, Wait), set(Mode, EvalExec)) within (0, 100] ms in mode Wait.	7
	// Switch to the “CheckComp” mode (if the maneuver execution is considered to be sufficiently safe) or to the “AbortExec” mode (otherwise) within 50 ms	8
	Reaction(change(Mode, EvalExec), set(Mode, {CheckComp, AbortExec})) within (0, 50] ms in mode EvalExec.	9
	// Switch to the “Subscribed” mode within 600 ms	10
	Reaction(change(Mode, AbortExec), set(Mode, Subscribed)) within (0, 600] ms in mode AbortExec.	11
	// Switch to the “Subscribed” mode (if the maneuver execution is completed) or to the “Wait” mode of the next iteration (otherwise) within 75 ms	12
	Reaction(change(Mode, CheckComp), set(Mode, {Subscribed, Wait})) within (0, 75] ms in mode CheckComp.	13

The timing requirements for the *V2X* component are listed in Table 3.14. The $\mathcal{C}_{v2x}^{unsub}$ specification is derived by applying shortcut S-1 to pTCAD_{unsub}^2 (line 5–6) and pTCAD_{unsub}^4 (line 8). Again, the input events are specified as sporadic events (lines 1–3).

The specification $\mathcal{C}_{tms}^{unsub}$ of the *TMS* component that results from the translation of pTCAD_{unsub}^3 is shown in Table 3.15. In this case, translation rule R-1 (line 3) and shortcut S-3 (line 5) are applied to derive corresponding MTSL specification patterns. The assumption specifies that *UnsubRequest* events occur sporadically (line 1).

3.5 Development of Constituent Systems

The development of CSs is organized according to the *Manufacturer-Specific Design and Implementation* process phase defined as part of the envisioned design approach (see Figure 3.1). It builds upon the results of the scenario-based activity modeling and realizes the manufacturer-specific design and implementation of the CSs involved in cooperative processes. The corresponding design activities consists of the following steps:

1. Definition of an initial functional architecture for each CS
2. Annotation of the derived MTSL timing specifications
3. Verification of correct refinement by performing a VIT on SoCPS level
4. Iterative decomposition and refinement of all CS models:
 - a) Decomposition of the functional architectures

3.5 Development of Constituent Systems

Tab. 3.13: Specification $\mathcal{C}_{chav}^{unsub}$ of the *CHAV 1/2* components

A	<i>// Additional assumptions about the occurrence of input events</i>	1
	<i>FrontCameraFrameA occurs every 33 ms with offset $[0, 33]$ ms and jitter 5 ms.</i>	2
	<i>FrontCameraFrameB occurs every 33 ms with offset $[0, 33]$ ms and jitter 5 ms.</i>	3
	<i>LidarPointCloud occurs every 60 ms with offset $[0, 60]$ ms and jitter 10 ms.</i>	4
	<i>Position occurs every 10 ms with offset $[0, 10]$ ms and jitter 2 ms.</i>	5
	<i>UnsubResponse occurs every $(0, \infty)$ ms with offset.</i>	6
G	<i>// Switch to the “Unsubscribe” mode after an “UnsubTrigger” event has occurred</i>	7
	Reaction(UnsubTrigger, set(Mode, Unsubscribe)) within $(0, 1]$ ms in mode Subscribed.	8
	<i>// Send the “UnsubRequest” event within 50 ms</i>	9
	Reaction(change(Mode, Unsubscribe), UnsubRequest) within $(0, 49]$ ms in mode Unsubscribe.	10
	<i>// Switch to the “FreeFlow” mode within 100 ms after receiving an “UnsubResponse” event</i>	11
	Reaction(UnsubResponse, set(Mode, FreeFlow)) within $(0, 100]$ ms in mode Unsubscribe.	12

Tab. 3.14: Specification $\mathcal{C}_{v2x}^{unsub}$ of the *V2X* component

A	<i>UnsubRequestIn1 occurs every $(0, \infty)$ ms with offset.</i>	1
	<i>UnsubRequestIn2 occurs every $(0, \infty)$ ms with offset.</i>	2
	<i>UnsubResponseIn occurs every $(0, \infty)$ ms with offset.</i>	3
G	<i>// Transmit “UnsubRequest” events within 200 ms</i>	4
	Reaction(UnsubRequestIn1, UnsubRequestOut) within $(0, 200]$ ms.	5
	Reaction(UnsubRequestIn2, UnsubRequestOut) within $(0, 200]$ ms.	6
	<i>// Transmit “UnsubResponse” events within 200 ms</i>	7
	Reaction(UnsubResponseIn, {UnsubResponseOut1, UnsubResponseOut2}) within $(0, 200]$ ms.	8

- b) Decomposition and refinement of timing specifications
- c) Execution of VITs to verify correct refinement

5. Verification of compliance with the refined timing specifications

The first step is the definition (or reuse) of a functional architecture for each CS. The previously derived real-time specifications can then be annotated to the initial component models on system level. For this purpose, the MTSL specifications defined for each activity/actor combination from Section 3.4 are composed to obtain a single top-level timing specification for each CS. In the context of the CLTA application example, these

Tab. 3.15: Specification $\mathcal{C}_{tms}^{unsub}$ of the *TMS* component

A	<i>UnsubRequest occurs every $(0, \infty)$ ms with offset.</i>	1
G	<i>// Switch to the “HandleUnsub” mode after receiving an “UnsubRequest” event</i>	2
	Reaction(UnsubRequest, set(Mode, HandleUnsub)) within $(0, 1]$ ms in mode Idle.	3
	<i>// Send the “UnsubResponse” event and switch to the “Idle” mode within 150 ms</i>	4
	Reaction(change(Mode, HandleUnsub), {UnsubResponse, set(Mode, Idle)}) within $(0, 149]$ ms in mode HandleUnsub.	5

3 Consistent Specification of Top-Level Timing Requirements

top-level specifications are derived by combining the base specifications $\mathcal{C}_{chav}^{base}$ and $\mathcal{C}_{v2x}^{base}$ defined in Section 3.4.2 with all relevant activity/actor specifications:

$$\mathcal{C}_{chav} = \mathcal{C}_{chav}^{base} \wedge \mathcal{C}_{chav}^{sub} \wedge \mathcal{C}_{chav}^{eval} \wedge \mathcal{C}_{chav}^{dec/exec} \wedge \mathcal{C}_{chav}^{unsub} \quad (3.62)$$

$$\mathcal{C}_{v2x} = \mathcal{C}_{v2x}^{base} \wedge \mathcal{C}_{v2x}^{sub} \wedge \mathcal{C}_{v2x}^{init/eval} \wedge \mathcal{C}_{v2x}^{unsub} \quad (3.63)$$

$$\mathcal{C}_{tms} = \mathcal{C}_{tms}^{sub} \wedge \mathcal{C}_{tms}^{init} \wedge \mathcal{C}_{tms}^{unsub} \quad (3.64)$$

Subsequently, verification is required to confirm that the real-time requirements on SoCPS level are correctly refined by the derived MTSL specifications. To this end, a VIT on SoCPS level is performed based on the top-level architecture. For this purpose, the architecture is annotated with all activity duration constraints $\phi_{dur}^{act} \in \Phi_{dur}^{act}$ of the considered TCADs as top-level guarantees, and all derived specifications on CS level as lower-level specifications. Note that the top-level guarantees can be obtained by translating each $\phi_{dur}^{act} = (act, lb, ub)$ into a corresponding **Reaction** pattern. In addition, all assumptions of lower-level components that are not fulfilled by guarantees of other lower-level components must be reflected by corresponding top-level assumptions and are therefore added to the top-level specification. In case of the CLTA example, the proof obligation for performing the desired VIT on SoCPS level is formulated as

$$\mathcal{C}_{clta} \succcurlyeq \mathcal{C}_{chav}^1 \otimes \mathcal{C}_{chav}^2 \otimes \mathcal{C}_{v2x} \otimes \mathcal{C}_{tms} \quad (3.65)$$

based on the top-level architecture from Figure 3.3. Note that the lower-level specification $\mathcal{C}_{chav}$ is instantiated once for each *CHAV* component to account for the fact that both vehicles should be able to take on both roles. To support other use cases, it is also possible to model the behavior required to take on the roles as separate sets of TCADs and translate them into individual MTSL specifications using the proposed approach.

In the following steps, the extended MULTIC methodology is applied to decompose and refine the initial CS models, analogous to the design approach presented in Section 2.3.2. The incremental design of the CSs involves a series of iterative decompositions of their functional architectures with parallel refinements of the annotated timing specifications. Through the systematic use of analysis techniques such as VITs, relevant proof obligations such as the verification of refinement relations are fulfilled. This ensures consistency of the resulting lower-level specifications with the original system-level specifications and guarantees compatibility of the behavior of individual CSs on SoCPS level.

Recall that the application of the methodology follows a set of design rules that determine the order in which certain MTSL specification patterns should be used. For example, causal and non-causal specification patterns can be used throughout the entire design process. Once introduced, mode-dependent specifications for handling dynamic architectures (see Chapter 4) must be refined and may not be omitted in subsequent design steps. The mapping of specifications to local clocks (see Chapter 6) is done just before the transition to the technical system design. The chosen coarse-to-fine approach avoids prematurely restricting the design space and ensures that the specified real-time behavior can be implemented in the final stages of system development. Engineers are

supported in applying the extended methodology by providing suitable tool support (see Chapter 7) and modeling guidelines for both process phases (see Appendix B).

After implementing the CSs, the final step is to verify the compliance with the refined timing specifications, e.g., using formal verification methods [56, 58], simulation-based or physical test procedures [93, 164], or online verification techniques [206]. Based on the already verified refinement relations between the specifications at different levels of abstraction, proving compliance with the specifications at the lowest level also proves compliance with the specifications on system and SoCPS level. At this point, the effort for the systematic approach and the incremental verification of refinement relations finally pays off, i.e., the correct-by-design nature of the design approach becomes effective.

The envisioned design approach enables the consistent formulation and refinement of real-time requirements on SoCPS level. It builds upon on the MULTIC methodology and adds support for early determination of cross-system coordination processes. The modeling of SoCPS behavior is achieved by introducing a novel formalism called *Time-Constrained Activity Diagrams* (TCADs) that allows scenario-based modeling of cooperative behavior. The proposed formalism enables the pre-competitive design of cooperative system functions in terms of implementation-independent interaction protocols. The resulting activity models are partitioned to assign activities and actions to individual subsystems, enabling the derivation of system-level specifications. This serves as a starting point for parallel development of CSs based on the previously agreed top-level specifications. The approach facilitates early collaboration between manufacturers and suppliers to develop consistent top-level specifications in early design phases that enable safe and reliable cooperation.

4 Handling of Dynamic Architectures

The consistent specification of dynamic system behavior is an essential prerequisite for the design of safe and reliable cooperative SoCPSs. At the same time, considering dynamic behavior as part of the specifications requires explicit handling of dynamic architectures. This includes the modeling, specification, and verification of the limited availability of individual CSs, behavioral changes of (parts of) the system at run-time, and dynamic reconfiguration, i.e., changes in the system’s configuration during operation. For example, from the perspective of a CHAV, other vehicles may enter and leave its sensing and communication areas. In addition, system functions may be degraded, e.g., as a result of the occurrence of faults. Systems may also be updated with new functionality and must therefore operate safely in different configurations of components and interfaces.

Reasoning about these aspects calls for appropriate specification means. In the previous chapter, we have already used mode-dependent specifications to formulate consistent top-level requirements for CSs involved in cooperative processes. However, these concepts were only introduced informally and have not yet been integrated into the MULTIC methodology (see Section 2.3). This chapter aims at closing this gap and addresses the second methodological gap “*Missing support for handling dynamic system architectures*” (MG-2). It extends the MULTIC design paradigms by enhancing MTSL 1.5 specification patterns to support specifying operating modes, mode transitions, and mode-dependent timing behavior. In addition, mode decomposition and the resulting proof obligations for common contract operations are investigated. The extensions serve as the basis for defining the MTSL 1.5+MC specification language (see Section A.2), which is one of the main results of this thesis. Finally, the use of mode-dependent specifications for modeling dynamic reconfiguration is discussed. The results serve as a starting point for enhancing analysis methods to support the verification of mode-dependent specifications, whose development is described in Chapter 7. Moreover, the contribution includes cross-layer process support for the specification and tool-supported refinement of mode-dependent timing specifications. The results contribute to answering RQ-2.1 and RQ-3.

The chapter is structured as follows. First, we first review existing approaches for modeling and specifying dynamic architectures in Section 4.1. Then, in Section 4.2, we investigate related work on mode-based specifications. Subsequently, we revise the system model described in Section 2.3.1 by introducing a set of basic concepts that are required for reasoning about component modes. The integration of operating modes, mode transitions, and mode-dependent timing behavior is then completed by defining consistent extensions of commonly used MTSL 1.5 specification patterns. Finally, the modeling of the dynamic reconfiguration of component behavior and interfaces is discussed in Section 4.3.

4.1 Modeling and Specification of Dynamic Architectures

As a first step in developing the required specification means, we consider existing approaches that are generally suitable for describing dynamic system behavior and dynamic architectures. However, we will find that most of the existing approaches are not compatible with the contract-based approach of the MULTIC methodology selected for the planned extensions. In addition, a common theme in many *Architecture Description Languages* (ADLs) is the use of modes and mode transitions to reason about dynamic architectures. For this reason, we will investigate further approaches that allow mode-based specification of timing behavior in Section 4.2.1. The considered approaches and related work from both areas form the basis for the selection of suitable methods and tools to extend the existing MTSL specification patterns for handling dynamic architectures.

According to common understanding, a *dynamic system* is a system “*that changes its configuration as it runs*” [283, p. 161]. The architectures of dynamic systems, referred to as *dynamic architectures*, can be considered as “*those that describe how architectural elements (components and connectors) can be created, interconnected, and/or removed during system execution*” [130, p. 31]. In this context, Mohsin et al. have defined four operations in [428, pp. 50–51] based on [445] that can be used to adapt dynamic architectures:

Addition “*A new CS is added to the SoS[.]*”

Removal “*CS may be removed from the system[.]*”

Replacement “*CS may be replaced [...], i.e., removed and substituted[.]*”

Rearrangement “*[The SoS] is dissolved and rebuilt in a different arrangement.*”

The operations indicate relevant properties that should be supported by specification means in order to be considered adequate. In the context of this work, we will focus on the first two operations, i.e., the *addition* and *removal* of CSs from a given SoCPS, as the most important operations for handling dynamic architectures of cooperative systems. From our point of view, the latter two operations *replacement* and *rearrangement* could be defined in follow-up activities based on the other two basic operations.

In addition to the frameworks presented in Section 2.2, which in most cases already provide lightweight support for handling dynamic architectures, there are other methods and description languages that can be used to handle dynamic systems. Existing approaches for modeling and specifying dynamic architectures that are relevant to this work can be divided into three categories, namely *System of Systems Engineering*, *Formal Modeling of Dynamic Systems*, and *Modeling of Dynamic Architectures*.

4.1.1 System of Systems Engineering

The dynamic nature of SoSs and, more specifically, SoCPSs is often understood as an inherent system property and is therefore already implicitly taken into account in corresponding design approaches. The discipline of *System of Systems Engineering* (SoSE) [140, 505, 517, 547] focuses on the systematic, continuous, and coherent engineering of SoSs,

addressing this and other challenges. First details about SoSE as well as a differentiation between classical SE and SoSE have already been presented in Section 1.2.3.

Several SoSE approaches support the development of (dynamic) architectures. For example, Cherfa et al. defined an approach to derive SoS architectures from conceptual mission models [142]. Baumgart et al. have pursued the same goal, but used sequence diagrams as the source model [52]. In addition, architecture patterns [311] and reference architectures [118] have been defined to systematize and accelerate the design of cooperative SoCPSs, including the handling of dynamic aspects of SoS architectures.

Other approaches deal with the modeling of configurations and context-sensitive changes at run-time. For example, Petitdemange et al. have proposed to model SoS configurations by the combined use of SysML and the *Unified Profile for DoDAF/MODAF* (UPDM) [458] in [477]. Wehrstedt et al. focused on modeling and analyzing context-sensitive changes of cooperative systems at run-time based on ontologies and capability models [602]. The authors also introduced variability models to account for the fact that “*capabilities provided in a certain state can differ from the capabilities in a future state*” [602, p. 133], thus providing support for modeling context-triggered reconfigurations.

Gutiérrez-García et al. have defined descriptive interfaces of CSs and obligations that must be fulfilled either by their own behavior or by other CSs [258]. The idea of the proposed multi-agent approach is that all agents strive to fulfill their obligations and must therefore establish links with other systems. Assuming the “*agents’ proactivity to propose obligations to other agents*” [258, p. 6], structural reconfiguration is used as a tool to achieve common goals. In addition, the “*incremental definition of the system allows adding and removing actions and policies [...] in a dynamic manner*” [258, p. 6].

Zhou et al. reviewed typical characteristics of SoSs, SoSE challenges, and different approaches for modeling such systems in order to derive a generic SoS model [624]. The resulting model combines (sub-)models of CSs with (sub-)models of the communication between these systems. The authors distinguished between *static* and *dynamic* descriptions of SoS components, the former including “*actions and goals one component can achieve alone*” [624, p. 366], and the latter “*[a]ctions and goals that [...] can be executed or achieved only through [...] cooperation*” [624, p. 366]. Dridi et al. conducted another review with similar goals, leading to the definition of a generic SoS engineering approach [190]. As part of their work, the authors considered *evolution*, *interaction*, *behavior*, and *reconfiguration* as the four main dimensions of dynamic architecture modeling.

4.1.2 Formal Modeling of Dynamic Systems

Early approaches for formal modeling of dynamic systems have focused on the modeling of communicating processes. In 1992, Milner et al. introduced the well-known π -calculus as a process algebra dedicated to the modeling of communicating processes with changing structure, i.e., the linkage of processes evolves over time [424, 425]. Based on this formalism, other programming and description languages for mobile processes were defined. Cardelli et al. introduced the object-oriented, distributed programming language Obliq [126], in which “*objects can redirect their behavior over the network, and where computations can roam between network sites*” [126, p. 295]. At the same time, Magee et al. proposed the

declarative specification language Darwin [401], which aims at describing the “*high-level organisation of computational elements and the interactions between those elements*” [401, p. 137], thus defining the software architecture of distributed dynamic systems.

Grosu and Stolen proposed an abstract, denotational model for timed, nondeterministic data-flow networks with point-to-point communication that considers a simple notion of mobility [252]. Haridi et al. defined the distributed programming system *Distributed Oz* [276] as an extension of the “*concurrent state-aware language Oz*” [276, p. 176] that answers key questions of distributed applications related to mobility, e.g., “*what information to transfer between sites and when and how to transfer it*” [584, p. 804]. The approach is build upon a programming language that “*abstracts away the network*” [276, p. 176] by implicitly invoking corresponding network operations without the need for manual interaction. It thus gives “*objects an arbitrary mobility behavior that is independent of the object’s definition*” [584, p.804]. However, since the focus of these approaches is on network-based information systems, there is no direct relation to SoCPSs.

About twenty years later, Broy proposed another modular “*denotational model for dynamic systems*” [107, p. 52] that focuses on describing “*discrete dynamic distributed systems*” [107, p. 39] in terms of a process algebra. The approach generalizes the static system model of the Focus approach [108], which was defined based on the π -calculus, and integrates “*concepts of causality, composition, abstraction, and system specification for dynamic systems*” [107, p. 39]. Again, the proposed formalism does not support the explicit handling of SoCPSs and cannot be easily integrated into CBD approaches.

4.1.3 Modeling of Dynamic Architectures

A more practical approach is the use of general-purpose modeling languages—or more specialized ADLs—to graphically model dynamic architectures. To this end, Medvidovic adapted techniques used in dynamic programming languages to extend static architecture specifications to consider dynamic architecture changes [415]. Kaminski et al. addressed the same problem by defining a “*dynamic event-based reconfiguration modeling framework*” [341, p. 1] that relies on MontiArc [260] component models and allows the modeling of dynamic architectures of self-adaptive systems [341]. The proposed framework supports both controlled component instantiation as well as “*adaptations of component interfaces without losing type-safety*” [341, p. 1] to capture dynamic aspects. The authors aim to support *data-triggered reconfiguration*, *service-based reconfiguration*, and *modes* as the three main reconfiguration approaches relevant to the development of dynamic systems.

Other ADLs for single systems, such as AADL [221, 223], which has already been presented in Section 2.1.1, follow the basic idea that “*[r]econfiguration can be modeled using modes and mode transitions defined in [...] finite state machine[s]*” [341, p. 2:5]. Further approaches such as AutoFOCUS 3 [224] and MontiArc [260] follow a similar principle. Moreover, Cavalcante et al. pursued modeling of dynamic software architectures [130] by means of π -ADL [463], which is based on the well-established π -calculus.

Kugele et al. have proposed an approach for model-based resource analysis and synthesis of service-oriented automotive software architectures based on a platform-independent meta-model [373]. Other groups of authors have compared architecture paradigms for

dynamic reconfigurable networks [527] and developed structured approaches for modeling service-oriented SoSs with SysML and SysML v2 [178]. In addition, Gandor et al. investigated a multi-viewpoint modeling approach for dynamically-coupled systems that support the explicit consideration of CSGs [236]. Marmsoler and Degenhardt applied model checking-techniques to verify design patterns for dynamic architectures [412].

Over the years, several governmental architecture frameworks, such as the *Department of Defense Architecture Framework* (DoDAF) [580], the *Ministry of Defense Architecture Framework* (MODAF) [426], and the *NATO Architecture Framework* (NAF) [450], have emerged as globally accepted standards for the definition and exchange of SoS architectures in the defense sector. UPDM [458] enables the modeling of DoDAF/MODAF design artifacts as part of an integrated architecture description. Pan et al. have focused specifically on the modeling and simulation of SoSs in the context of DoDAF [471]. The presented approach includes the modeling of SoSs with respect to the *Activity-Based Methodology* (ABM) [506] and the simulation of BPMN [453] models derived from the ABM models. Moreover, House combined UPDM and *Reusable Asset Specification* (RAS) [456] descriptions to define a “model of models” approach for SoS design [281] that addresses some of the “*pain points*” in SoSE [319] already mentioned in Section 1.2.3.

In addition to ADLs for single systems, specialized ADLs for designing SoS architectures have been developed. A first step in this direction was the introduction of X-UNITY [235], which supports the modeling of SoSs in terms of hierarchical hybrid systems. As another representative, CML [609] has been defined as a low-level formal specification language focusing on contracts of interactions that can be used to annotate SysML models. From a broader perspective, Guessi et al. identified required features of ADLs for describing SoS software architectures and assessed UML [455], SysML [454], SysML+CML [609], and X-UNITY [235] as potential candidates for modeling such architectures [256].

Later, Oquendo developed SosADL [461, 462] as a more expressive alternative to existing ADLs for SoSs. The author argued that the “*expressive power to describe the software architecture of a software-intensive SoS*” of conventional ADLs is not sufficient [256, 355]. The proposed approach includes the modeling of SoSs in terms of *systems*, i.e., architectural elements that provide interaction points and functionality, *mediators* that enable interaction, *coalitions* that constitute a “*temporary alliance for combined action among systems connected via mediators*” [462, pp. 2–3] as well as *environments* for each of the previously defined systems. All models are based on the “ π -calculus for SoS” [464], which has been defined as an SoS-specific extension of the original π -calculus.

4.2 Specification of Mode-Dependent Timing Behavior¹

As already made clear in the previous sections, we have found that most of the existing approaches do not support explicit handling of cooperative SoCPSs or cannot be easily

¹Earlier versions of this section have been published in [371] and [370]. Most of the work was carried out in close collaboration with Janis Kröger and Ingo Stierand. My contribution included the research of existing approaches, the compilation of formal foundations, the definition of (parts of) the syntax and semantics of pattern extensions as well as the specification of the application example.

integrated into CBD approaches. A bright spot is the fact that many ADLs rely on the basic concept of operating modes to reason about dynamic behavior and dynamic architectures. In this context, we will focus on extending the MULTIC methodology to support operating modes, mode transitions, and mode-dependent timing behavior as building blocks for specifying dynamic behavior. The main idea is to enable and disable individual MTSL specification patterns depending on the active mode of the component in focus. With these extensions, component behavior no longer depends only on input events, but also on internal state. The proposed extensions allow to describe the dynamic behavior of individual CSs and cooperative SoCPSs, including dynamic reconfiguration. At the same time, the envisaged extensions close the gap between the MULTIC design paradigms (see Section 2.3.1) and the requirements of mode-dependent timing specifications, which have already been extensively used to formulate system-level specifications as part of the five-step translation approach presented in Chapter 3.

In addition to modeling the dynamic system behavior of cooperative SoCPSs, there are other application scenarios in which the use of mode-based specifications is advantageous. As already explained in the introduction (see Section 1.1), the design of safety-critical systems relies heavily on the comprehensive elicitation and verification of safety-relevant system properties. FTA, for example, is a well-established—and in some application domains mandatory—method to investigate the impact of potential failures on the correctness of system functions. The results serve as input for the development of countermeasures that are to be integrated into the overall system design. The same applies to FMEA, which is devoted to reveal potential propagation paths of failures in the system. The development and verification of systems that are hardened against failures calls for system models that incorporate operating modes. For example, the behavior of redundancy mechanisms can be detailed to verify that safety-critical functions of components in failure modes are taken over by their backups as expected.

Modeling operating modes is also helpful in managing design complexity. For example, modern *adaptive cruise control* (ACC) systems in road vehicles are able to function in two modes. In *Cruise* mode, the vehicle's speed is kept to a value set by the driver. In *Follow* mode, the ACC is controlling the speed to maintain a safe distance from a previously detected vehicle ahead. The active mode is selected according to the current traffic situation. In order to reduce the design complexity, Damm et al. [165] have developed an approach in which operating modes are realized by individual components that interact via a dedicated protocol to transfer activity tokens. The approach allows to split the verification of system properties into the verification of the individual components and the correctness of the mode transition protocol, thus reducing the overall verification complexity. The reason for the reduction in complexity is that modes provide an automaton-like view that significantly restricts the set of state pairs that can actually follow each other, thus focusing and thereby facilitating sequential consistency proofs.

Correctness of mode transitions generally concerns two entangled aspects. First, they must occur consistently as specified and do not result in unexpected behavior. In case of the ACC example, this means that at any point in time exactly one mode is active. The second aspect is timing, which is closely related to the first, since mode transitions

typically do not occur simultaneously among the involved components but must be propagated in the system. Other timing aspects may also play an important role. Safety mechanisms, for example, are specified with respect to FTTIs, which define the maximum period of time allowed from the occurrence of a failure until the measures take effect.

The remainder of this section is organized as follows. Section 4.2.1 provides an overview of existing approaches for handling operating modes and places our contribution in its scientific context. Section 4.2.2 introduces the basic concepts that serve as building blocks for describing component-based system models and formally reasoning about operating modes. In Section 4.2.3, we investigate general properties and identify relevant timing phenomena using a simple example system that incorporates mode-dependent behavior. Based on this, we extend two commonly used MTSL specification patterns in Section 4.2.4. In Section 4.2.5, the developed concepts are applied to a realistic example system to verify their practical applicability. Finally, we discuss mode decomposition and relevant proof obligations when applying mode-dependent timing specifications in Section 4.2.6.

4.2.1 Existing Approaches

Existing research that is particularly relevant in the context of our work can be divided into two areas. The first area comprises *classical approaches* to modeling internal states and modes for system design. Related work in this area is presented in Section 4.2.1a. The second area focuses on approaches in the field of *contract-based design* (CBD), which are discussed in Section 4.2.1b. Previous work includes concepts that facilitate the *reuse of existing components* in new environments and application contexts, and addresses the use of operating modes as a tool for achieving *coordinated behavioral changes*.

Classical Modeling Approaches

In his seminal work [270], Harel introduced *statecharts* that extend classical state machines by concurrency and hierarchy to address the problem of *state and arc explosion*. The visual formalism introduces a notion of event-based communication by adding constructs to generate and receive events in different subcharts. Over the years, numerous variants and dialects have been developed [61]. Statecharts rely on a powerful semantics to consider hierarchical states. In terms of our approach, this would be equivalent to establishing a *gray box* perspective on component specifications, which contradicts the *black box* principle of CBD. In the following, we will therefore rely on an explicit mapping between higher-level and lower-level modes to reason about their relations.

About a decade later, Maraninchi and Remond proposed *mode-automata* [411] as a formal and executable formalism to describe mode structures of reactive systems. It allows the modeling of operating modes and mode transitions by means of automata, whose states are assigned to dataflow programs. The authors have also introduced operations for parallel and hierarchic composition, the latter adding support for hierarchical modes. In contrast to our approach, mode-automata do not provide descriptive means to express temporal properties. Therefore, the formalism is only of limited use for our purposes.

Finally, modes are also included in languages specialized for the design of safety-critical systems like *Giotto* [287], which adopts a task semantics for modes. In this context, a mode is defined as a fixed set of tasks that are called according to a certain pattern. Giotto also supports mode switches that enable or disable the invocation of tasks. However, since it is dedicated to technical system design, its focus is clearly on implementation and it is not intended to formulate timing specifications in early design phases.

All three formalisms have in common that there is no direct relation to CBD and thus no easy way to integrate them into the MULTIC methodology, which was selected as a candidate for the planned extensions. However, they define a set of basic concepts that are essential for handling operating modes in the context of our work.

Contract-Based Design

As already explained in Section 2.1.3, CBD facilitates component reuse by allocating responsibilities in specifications. A *contract* specifies, on the one hand, all environments in which a component is supposed to operate and, on the other hand, the expected behavior of the component if it is used in such an environment. The present work relies on the particular contract theory of A/G contracts developed in [70], in which the *assumption* A specifies the behavior of the environment, and the *guarantee* G specifies the behavior of the component. The explication of responsibilities between the environment and a component is a major advantage with respect to “flat” specifications. In addition, the use of natural language or semiformal requirements such as, for example, the specification patterns proposed in [92], helps engineers to capture the meaning of specifications more intuitively than by using temporal logics such as LTL [483] or MTL [367] (see Section 2.1.2).

Reuse of Existing Components The idea of A/G contracts is further developed by the concept of *fine-grained contracts* [542, 543]. Here, weak assumptions are employed for specifying alternative contexts in which components can be used. Corresponding guarantees allow to specify the behavior of the components individually for each of the assumed environments. The work is closely related to our approach, as both refine classical A/G contracts. Nevertheless, whereas fine-grained contracts allow to specify intended behavior for different environments, the concepts presented in this work are designed to specify dynamic system behavior within an unaltered or gradually changing context.

Reussner et al. have addressed specifications for components that require interface changes in order to be deployed in certain contexts by *parametric contracts* [500]. The concept can be used to explore different dimensions of compositionality when adapting components to new environments. Since parametric contracts are capable of modeling the dependencies between provided and required functionalities, they support predictions about the quality of service, analysis of architectural design, and automated protocol adaptation. In [501], specialized graph grammars are used to reason about these aspects. Firus et al. have applied parametric contracts for measuring the performance of software components [228]. The approach determines discrete response time distributions while considering statistical distributions of response times of environmental services. With regard to our work, parametric contracts do not provide support for handling operating

modes. Nevertheless, they allow the adaptation of component behavior and interfaces to changing contexts, thus ensuring interoperability in composite systems.

In more recent work, Kugele et al. have introduced a component model enriched with interface assertions and *mode-based contracts* [374], which are formulated by means of predicate logic formulas. The approach provides support for determining composed interface assertions for given architecture configurations. It is complemented by algorithms for checking underspecification, overspecification, and specification compliance. Compared to our approach, mode-based contracts do not focus on timing aspects and lack support for hierarchical component models. However, the approach provides “*methodological guidance for developers*” [374, p. 133] and facilitates tool-supported contract specification.

Coordination of Behavioral Changes The principle of CBD has also been employed to model dynamic behavioral changes and coordination mechanisms. In [165], Damm et al. exploited contract specifications to reason about the stability and safety of systems that consist of components, which provide the desired properties only in certain contexts. Control between the individual components is actively passed in terms of a token protocol. The method enables compositional reasoning to reduce the verification complexity.

Champion et al. have designed a mode-aware specification language called *CoCoSpec* [133], which extends the A/G paradigm and is designed to specify synchronous reactive systems. It provides benefits such as rigorous feedback for fault localization, scalable and adequate compositional analysis, and semantic analysis to identify oversights. The approach is developed as an extension of the synchronous programming language Lustre and provides support for mode-aware verification by using the Kind 2 model checker [134]. In contrast to our approach, CoCoSpec is an implementation-oriented language that is particularly suited for use in later design phases. It is therefore not specifically tailored to early requirements engineering. In addition, the focus is on functional aspects, without providing specification means for timing behavior. Similar to the present work, the relations between hierarchical modes are established by an explicit mapping.

The present work follows the same line as those mentioned in Section 4.2.1b. It builds upon extensive previous work on contract specifications by enhancing the MULTIC methodology introduced in Section 2.3. The envisaged extensions will allow the specification of component modes and, in particular, (the timing of) mode transitions and mode-dependent behavior, thus enabling a consistent reasoning about mode composition and the effects of individual modes on the behavior of the overall system.

4.2.2 Basic Concepts

In the following, we rely on an extended version of the system model defined in [92], which was later refined in [93] (see Section 2.3.1). According to this model, a system S consists of a set of hierarchically nested *components* C . Each component $c \in C$ is equipped with three disjoint sets of *input ports* P_c^{in} , *output ports* P_c^{out} , and *variable ports* P_c^{var} , which together represent the observables of the encapsulated component behavior. In order to provide a meaningful functionality, each component must have at least one port

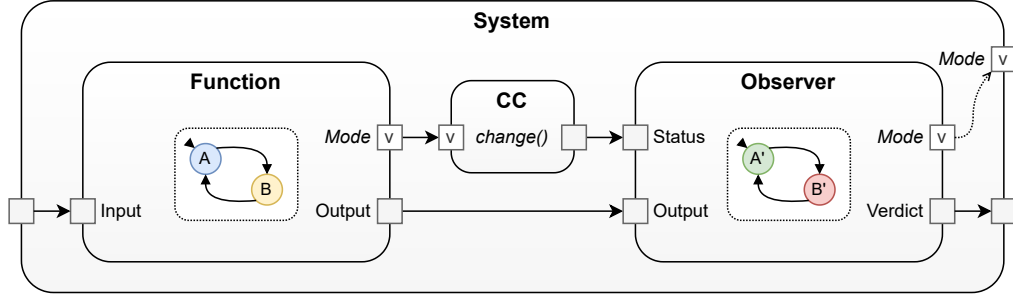


Fig. 4.1: Component model of the example system

belonging to one of these classes, i.e., $P_c := P_c^{in} \cup P_c^{out} \cup P_c^{var} \neq \emptyset$. The set of *system ports* $P = P_{in} \cup P_{out} \cup P_{var}$ is defined as the union of all ports of all components, i.e.,

$$P_{in} = \bigcup_{c \in C} P_c^{in}, \quad P_{out} = \bigcup_{c \in C} P_c^{out}, \quad \text{and} \quad P_{var} = \bigcup_{c \in C} P_c^{var}. \quad (4.1)$$

The interaction of two or more components $c_1, \dots, c_n \in C$ with $n \in \mathbb{N}$ and $n \geq 2$ is represented by *signals* $s : P \rightarrow 2^P$. A signal usually connects an output port $p_o \in P_{out}$ with a set of input ports $P_{in} \subseteq P_{in}$. For connections between two hierarchy levels, however, signals can also be defined between two or more input ports $s_{in} : p_{i_1} \mapsto \{p_{i_2}, \dots\}$, or two or more output ports $s_{out} : p_{o_1} \mapsto \{p_{o_2}, \dots\}$, with $p_{i_1}, p_{i_2} \in P_{in}$ and $p_{o_1}, p_{o_2} \in P_{out}$. The set of signals between the system's components C is denoted by S .

Figure 4.1 depicts a simple example system. The *System* is decomposed into two main components *Function* and *Observer*, each of which has multiple input and output ports as well as a variable port called *Mode*, whose characteristics are detailed in Sections 4.2.2a and 4.2.2b. An interaction component *CC* (see Section 2.3.1) serves as a converter channel, which forwards only selected events from the variable port *Function.Mode* to the *Observer.Status* input port. The ports of the lower-level components are interconnected with three signals that pass occurring events and value changes along the connections. In addition, the lower-level input and output ports *Input* and *Verdict* are linked to the corresponding ports of the top-level component. A dotted arrow connects the *Observer.Mode* and *System.Mode* ports, indicating a dependency between the two ports. The exact semantics of this relation is explained later in Section 4.2.3.

In order to characterize desired system behavior, each component $c \in C$ can be annotated with a set of *specifications* Φ_c . Naturally, each element $\varphi \in \Phi_c$ must refer only to the component ports P_c to make statements about behavior. Formally, the behavior observable at the components' interfaces is defined in terms of infinite timed traces. In order to specify Φ_c , we rely on declarative specifications in terms of A/G contracts within a compositional reasoning approach, which is detailed in Section 4.2.2c.

Event Ports

The sets of input and output ports are collectively referred to as *event ports* $P_{in/out} = P_{in} \cup P_{out}$. The behavior observable at port $p \in P_{in/out}$ is restricted to its value domain

Σ_p specified by the port type. We assume a special value $\perp$ to be a member of every value domain, which represents the absence of a value. A notion of *dense time* is used to characterize the occurrence of events. Therefore, we define $\mathbb{T} = \mathbb{R}_0^+$ to be the time domain. Each event port has non-absent values for all $t \in T \subset \mathbb{T}$. A single event e occurring at port $p \in P_{in/out}$ is defined as a tuple $e = (\sigma, t)$ that consists of an event value $\sigma \in \Sigma_p$ and a time of occurrence $t \in T$. This allows us to describe the semantics of event ports in terms of timed traces, for which we use definitions from Böde et al. [93].

A *timed trace* over port p is defined as an infinite sequence $\omega_p = (\sigma_i, t_i)_{i \in \mathbb{N}}$, in which $\sigma_i \in \Sigma_p$ are elements from the value domain and $(t_i)_{i \in \mathbb{N}}$ forms a monotonic sequence of time instances. We require all timed traces to be non-zeno, i.e., for each $t \in \mathbb{T}$ exists (σ_i, t_i) such that $t_i \geq t$. Moreover, we denote the set of timed traces observable at port p by $\Omega_p = \{\omega = (\sigma_i, t_i)_{i \in \mathbb{N}}\}$. For each port set P , we define a set of timed traces $(\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}$ over P , where $\vec{\sigma}_i = (\sigma_1, \dots, \sigma_n) \in \Sigma_{p_1} \times \dots \times \Sigma_{p_n}$ holds. Similar to the consideration of individual ports, we denote the set of traces over a given set of ports P by $\Omega_P = \{\omega_P = (\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}\}$. Based on this, the set of *possible timed traces* observable at the component's interfaces can be characterized by a timed language $L_P \subseteq \Omega_P$.

Note that most of the definitions related to events and timed traces have already been introduced in Section 2.3.1. However, they are repeated here for completeness. The definitions serve as a basis for the extension of the system model with variable ports, which in turn are the basis for the introduction of mode-dependent specifications.

Variable Ports

In order to specify operating modes, we define *variable ports* P_{var} , which represent simple variables with discrete evolution. Just like event ports, each variable port $p \in P_{var}$ has a clearly defined value domain V_p . In contrast to event ports, variable ports have a specified value at each point in time $t \in \mathbb{T}$, which evolves at discrete time points, i.e., no absent values exist. The behavior of a variable port $p \in P_{var}$ can thus be characterized as a *value history* $v_p : \mathbb{T} \rightarrow V_p$ that maps a well-defined value to each time point $t \in \mathbb{T}$.

To enable the interaction between event and variable ports, we introduce two new event types, namely *set()* and *change()*, that occur on an implicitly defined *virtual event port* $p_{virt}(p) \in P_{in/out}$ that is assigned to each variable port $p \in P_{var}$:

1. *set*(p, v) events represent the assignment of a value $v \in V_p$ to the variable port p and is called to initiate value changes. Assuming a *set()* event occurring at time $t_i \in \mathbb{T}$, the value of p is updated to $v_p(t_i) = v$.
2. *change*(p, v) events indicate a change in the value of p that results from a *set()* event. They can be used to react to value changes. Whenever a time point $t_i \in \mathbb{T}$ exists such that $v_p(t_i) \neq v_p(t_{i-1}) \wedge v_p(t_i) = v$, a *change()* event occurs at time t_i .

For both events, the explicit specification of $v \in V_p$ is optional. In case of a *set()* event, omitting v means that an arbitrary value from V_p is set to p . Omitting v in a *change()* event allows any change of the value of p . Later on, we will need to reason about traces of *change()* events and hence define a *change event trace* $(change()_i, t_i)_{i \in \mathbb{N}}$ for each $p_{virt}(p)$,

which contains all *change*(p) events. Concerning the observable behavior of a component $c \in \mathbf{C}$, note that each $p_{virt}(p)$ of a variable port $p \in P_c^{var}$ is a (virtual) member of $P_c^{in} \cap P_c^{out}$, and thus, *set*() and *change*() events are “visible” in the timed language L_P .

Specifications

Components can be equipped with *specifications*, which in our case consist of A/G contracts. To keep things simple, we assume exactly one contract $\mathcal{C}_c$ to be assigned to each component $c \in \mathbf{C}$, where $\mathcal{C}_c = (\Phi_c^A, \Phi_c^G)$ such that $\Phi_c = \Phi_c^A \cup \Phi_c^G$. In other words, the specifications in Φ_c are partitioned into those that specify the assumption of the contract and those that specify the guarantee. Each specification defines a set of *valid timed traces* of c , which is denoted by L_φ . As said above, all specifications are defined over the components’ interfaces, i.e., $L_\varphi \subseteq \Omega_{P_c}$. Furthermore, we define $L_\Phi = \bigcap_{\varphi \in \Phi} L_\varphi$ and designate the set of timed traces that comply with a given A/G contract as $L_{\mathcal{C}_c} = \Omega_{P_c} \setminus L_{\Phi_c^A} \cup L_{\Phi_c^G}$. In other words, the logical meaning of a contract is defined as $\Phi_c^A \implies \Phi_c^G$. A component is thus supposed to adhere to its guarantee only if its environment adheres to the component’s assumption. This correspondence gives reason to the definition of contract *satisfaction* (see Definition 1). Assuming that we denote the possible behavior of a component c by the language L_c , we say, c *satisfies* $\mathcal{C}_c$, written $c \models \mathcal{C}_c$, if $L_c \cap L_{\Phi_c^A} \subseteq L_{\Phi_c^G}$. In other words, c satisfies its contract $\mathcal{C}_c$ if it behaves as specified by the guarantee under all possible behaviors of its environment.

There are two other definitions in CBD that are relevant in the context of this work, which have already been explained in Section 2.1.3. Any contract theory relies on a *composition* operation that defines how new components are obtained from the composition of other (sub-)components. Along this operation on components, *composition* defines how to obtain a contract from the contracts of lower-level components, written $\mathcal{C} = \mathcal{C}_1 \otimes \mathcal{C}_2$ (see Definition 3). The *refinement* relation defines under which circumstances a component $c \in \mathbf{C}$ can be replaced by another component $c' \in \mathbf{C}$ without violating existing specifications. We say, $\mathcal{C}'$ *refines* $\mathcal{C}$, written $\mathcal{C}' \preceq \mathcal{C}$, if $L_{\Phi_{c'}^A} \supseteq L_{\Phi_c^A}$ and $L_{\Phi_{c'}^G} \subseteq L_{\Phi_c^G}$ (see Definition 4). It means that component c' allows for more behavior of its environment and has a more confined behavior than c . Note that $\mathcal{C}'_1 \preceq \mathcal{C}_1$ and $\mathcal{C}'_2 \preceq \mathcal{C}_2$ implies $\mathcal{C}'_1 \otimes \mathcal{C}'_2 \preceq \mathcal{C}_1 \otimes \mathcal{C}_2$.

Exploiting composition and refinement, we can define what it means for a system S to satisfy a specification. Given that S is specified by a contract $\mathcal{C}_S$, and S is composed by a set of components $\mathbf{C}$, S satisfies its specification, i.e., $S \models \mathcal{C}_S$, if $\bigotimes_{c \in \mathbf{C}} \mathcal{C}_c \preceq \mathcal{C}_S$.

4.2.3 Operating Modes

In general, an *operating mode* represents an internal state of a component. In the context of this work, we assume that the mode of a component does not change spontaneously, but is triggered by interactions via its event and variable ports. We define each $c \in \mathbf{C}$ to have a dedicated variable port $p_m \in P_c^{var}$ called *Mode*, which has a *mode history*

$$m_c(t) = v_{p_m}(t) \quad (4.2)$$

4.2 Specification of Mode-Dependent Timing Behavior

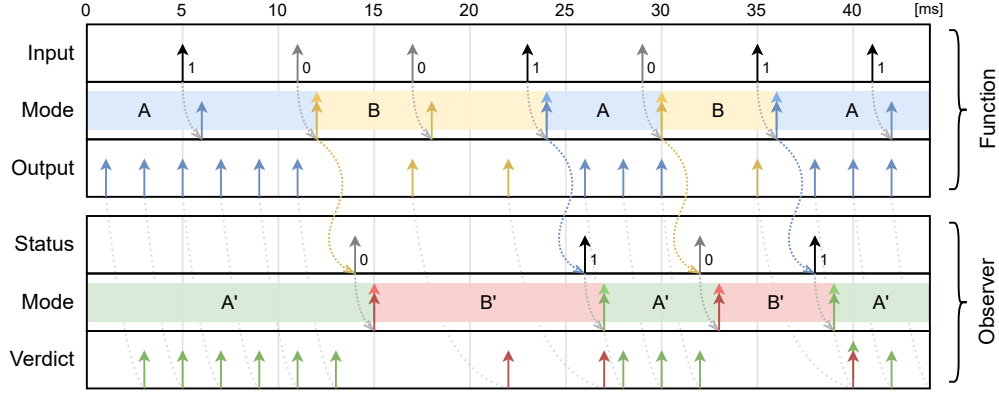


Fig. 4.2: Intended behavior of the example system

with $t \in \mathbb{T}$. The *active mode* $m_c(t)$ of c results from the initial mode $m_c(0) \in V_{p_m}$ as well as the sequence of *set()* events over $p_{virt}(p_m)$ that have occurred up to the current time t . The set of modes M_c of a component is given by $M_c = V_{p_m}$, where V_{p_m} has either been explicitly defined or is implicitly derived from the set of values specified in Φ_c . In the following, the change event trace $(change()_i, t_i)_{i \in \mathbb{N}}$ of port $p_{virt}(p_m)$ is denoted by $(m_i, t_i)_{i \in \mathbb{N}}$. We define a *time-bounded mode-set projection*

$$(m_i, t_i)_{M,I} = \{(m_i, t_i) \mid m_i \in M \wedge t_i \in I\}, \quad (4.3)$$

where M is a set of modes and $I \in \mathcal{I}$ is either a left-closed, right-open time interval $I = [a, b)$ or an open time interval $I = (a, b)$ with $a, b \in \mathbb{T}$.

To shed some light on the interaction of events occurring at the components' interfaces, i.e., on event ports $p \in P_c^{in} \cup P_c^{out}$ with $c \in \mathbb{C}$, and the evolution of modes, i.e., $m_c(t)$ with $t \in \mathbb{T}$, we again consider the example system presented in Figure 4.1. The figure indicates that the *Function* component has two modes A and B , which are visible at the variable port *Function.Mode*. The *Observer* component also has two modes, namely A' and B' .

Figure 4.2 depicts a set of example traces that illustrate the intended system behavior. The *Function* and *Observer* components sequentially process a sequence of events. *Function* receives periodic *Input* events that occur every 6 ms with an offset of 5 ms. Depending on the value of the *Input* events, which is either 0 or 1, the variable port *Function.Mode* is set with a delay of 1 ms. The effects of the *set(Function.Mode, A)* and *set(Function.Mode, B)* events on the value of the *Mode* port are indicated by dotted arrows. Some of the *set()* events lead to changes of the mode value of the *Function* component and thus to the occurrence of *change()* events at the virtual event port $p_{virt}(p_{m_{fun}})$. They are forwarded to the *Status* port of the *Observer* via the converter channel *CC*, which filters out events other than *change()* events. In addition, the *Function* component adapts the frequency of the *Output* events according to its active mode $m_{fun}(t)$, i.e., by using a period length of 2 ms in mode A and a period length of 5 ms in mode B .

The *Observer* component, on the other hand, receives the *Output* events and evaluates whether the observed behavior is consistent with the (time-delayed) knowledge about the

mode of the functional component. To this end, it stores the current mode of the *Function* component in $m_{obs}(t)$ by updating the value of the *Observer.Mode* port according to the received *Status* events. The component creates *Verdict* events to publish the results of the evaluation. A mode-dependent time span of 2 ms in mode A' and 5 ms in mode B' is required to create the verdict. Again, the dotted arrows in Figure 4.2 indicate how mode changes of the *Function* component are translated into *Status* events, which in turn lead to mode changes of the *Observer* component. The mode-dependent delay of the *Observer* is visualized by dotted connections between the *Output* and *Verdict* events.

In the further course of this chapter, we will define mode-dependent timing specifications that are able to cover component behavior as shown in the example. To determine whether a particular behavior is valid with respect to a mode-dependent specification, we must first evaluate which mode the component is in. To this end, we have to define time points at which the current mode of a component is evaluated. Here, we distinguish two cases: event occurrences and reactions to trigger events, which correspond to two specification parts that are used to formulate any contract $\mathcal{C}_c$ with $c \in \mathbf{C}$. For both cases, the *evaluation time points* are fully synchronized with the occurrence of events:

- *Instantaneous* parts specify the occurrence of events according to a specific model, such as the periodic *Output* events generated by the *Function* component. In this case, the mode of component c is evaluated at each (anticipated) time of occurrence $t_{eval} = t_e^\uparrow \in \mathbb{T}$ of event $e = (\sigma_e, t_e^\uparrow) \in \Sigma_p \times \mathbb{T}$ with $p \in P_c^{in} \cup P_c^{out}$. For a given set of modes M_{cond} , we define the corresponding condition as $\exists m \in M_{cond} : m_c(t_{eval}) = m$.
- *Reactive* parts specify the component's response to triggering events $e = (\sigma_e, t_e^\uparrow) \in \Sigma_p \times \mathbb{T}$ with $p \in P_c^{in}$, such as the *Verdict* events that occur as a result of the *Output* events. In this case, the condition $\exists m \in M_{cond} : m_c(t_{eval}) = m$ is evaluated at each time of occurrence $t_{eval} = t_e^\uparrow \in \mathbb{T}$ of the event e to be reacted to.

A missing puzzle piece is a mechanism for reasoning about the relations between modes at different levels of the component hierarchy. For the system shown in Figure 4.1, an important verification task is checking whether $\mathcal{C}_{fun} \otimes \mathcal{C}_{cc} \otimes \mathcal{C}_{obs} \preceq \mathcal{C}_{sys}$ holds. This check is also called VIT. In order to perform this test, we need to characterize the relation between the top-level modes M_{sys} and the lower-level modes of its subcomponents.

A simple solution is the definition of a *mode mapping function* $\mu_c : M_{sub_1} \times \dots \times M_{sub_n} \rightarrow M_c$ between the sets of modes $M_{sub_1} \times \dots \times M_{sub_n}$ of lower-level components $sub_1, \dots, sub_n \in \mathbf{C} \setminus \{c\}$ with $n \in \mathbb{N}$ and the set of modes M_c of a higher-level component $c \in \mathbf{C}$. In the example above, the mapping function simply maps the *Observer* mode to the *System* mode. More precisely, $\mu_{sys} : M_{fun} \times M_{cc} \times M_{obs} \rightarrow M_{sys}$ is defined such that $\mu_{sys}(m_{fun}, m_{cc}, m_{obs}) = m_{obs}$. It is important to note that μ_c must be an input of any verification engine that is used to perform VITs of mode-dependent specifications.

For this purpose, the mapping between the modes of lower-level and higher-level components is specified by using a dedicated **Mode Mapping** pattern. The pattern allows to express relevant combinations of lower-level modes and the resulting higher-level modes as defined by μ_c . The syntax of the **Mode Mapping** pattern is defined as

4.2 Specification of Mode-Dependent Timing Behavior

$\{m_{sub_1}, \dots, m_{sub_n}\}$ maps to m_c .

with $m_{sub_i} \in M_{sub_i}$, $i \in \mathbb{N}$, $i \leq n$, and $m_c \in M_c$. Note that all lower-level modes $\{m_{sub_1}, \dots, m_{sub_n}\}$ that do not affect m_c can be omitted. The pattern is intended for use in the specification $\mathcal{C}_c$ and is instantiated for each relevant combination of lower-level modes. Applying this approach establishes a tree-like hierarchy, in which well-defined and traceable relations between modes along the component hierarchy exist.

A disadvantage of this solution is the fact that the **Mode Mapping** pattern introduces an additional specification mechanism. Moreover, it does not explain how the specified mapping is realized during system design, e.g., as part of the system architecture. An alternative solution would be to integrate specialized *Mode Logic* components into the system that realize the mode mapping [369]. The main advantage of an approach as presented in [370] is the fact that *Mode Logic* components can also be used to control and orchestrate the modes of individual subcomponents, in addition to only aggregating them. Although the framework presented in this chapter supports this approach without any adaptations, the specification patterns developed in the following section are only sufficient for the specification of simple use cases. More sophisticated scenarios would require the introduction of specifications beyond purely event-based patterns. More specifically, it would be necessary to allow the use of variable ports as input ports of components, which is beyond the scope of this work. We will return to this idea in Section 4.2.5.

4.2.4 Mode-Dependent Timing Specifications

In order to facilitate the specification of modes and mode-dependent timing behavior, we aim to extend two existing specification patterns that allow to specify repetitive event occurrences and latencies. The basic idea is to extend each pattern with an optional set of modes M in which it has to be fulfilled. Since we assume a use in CBD, we will present consistent extensions for use in assumptions and guarantees. On the assumption side, we will allow mode-dependent counterparts of general assumptions. This enables the formulation of more precise and situation-tailored constraints on the component's context. An even more important reason for introducing mode-dependent assumptions is the advantage in terms of analyzability, since undefined or inconsistent mode combinations are made explicit and can thus be detected more easily. On the guarantee side, we need to define which mode is initially valid, which mode transitions exist, when and in what period they may be taken, and how the mode-dependent timing behavior of the component in focus looks like. In addition, immediate effects of mode changes in terms of adaptations of internal or output behavior will be described with mode-dependent guarantees.

In both cases, we may need to consider the behavior *before* and *after* a mode change in order to obtain a consistent and well-defined set of specifications. Inconsistencies could arise, for example, when leaving a particular mode and switching to another mode, in which a required input behavior is not (yet) provided by the interacting components. Two possible solutions would be to either ensure that the appropriate input events are provided before the mode change is implemented, or to define a limited period of time in which the absence of required input signals is tolerated after the transition is completed.

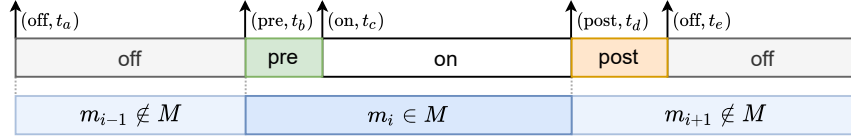


Fig. 4.3: Specification pattern states and component modes

The approach presented here aims at the second solution. We introduce *pre*- and *post*-phases, which allow to permit temporary deviations from the specified mode-dependent behavior within a predefined time interval. Pre-phases, on the one hand, specify a kind of “settling phase” from the time of a mode change $t_b \in \mathbb{T}$ to a fixed time bound $t_b + D_{pre}$ with $D_{pre} \in \mathbb{T}$, in which both the specified behavior of the active mode and (parts of) the behavior of the previous mode may apply. Post-phases, on the other hand, define a “tail phase” that starts with another mode change at time $t_d \in \mathbb{T}$ and ends at $t_d + D_{post}$ with $D_{post} \in \mathbb{T}$ at the latest. Within this period, the specifications that has to be fulfilled in the active mode as well as the specifications of the previous mode represent valid behavior. In the following, we assume default durations $D_{pre} = 0$ and $D_{post} = 0$ for instantaneous specifications as well as $D_{pre} = 0$ and $D_{post} = \infty$ for reactive specifications.

Based on the introduction of pre- and post-phases, we consider four *specification pattern states* $\mathcal{S} = \{\text{pre}, \text{on}, \text{post}, \text{off}\}$ that are illustrated in Figure 4.3. As long as a specification pattern is in the *on* state, the component behavior must comply with this pattern. In the *off* state, the pattern does not need to apply. In *pre* and *post*, the behavior applies as described in the paragraph before. We require that each pattern passes through all states in the given order. Note, however, that any state can be left after zero time.

For each change event trace $(m_i, t_i)_{i \in \mathbb{N}}$, we inductively define a *specification state trace* $st = (state_i, t_i)_{i \in \mathbb{N}}$ with $state_i \in \mathcal{S}$ based on a set of modes M . The trace st contains all changes between the specification pattern states $state_i \in \mathcal{S}$, including their time of occurrence $t_i \in \mathbb{T}$. The initial state of each trace is set to

$$(state_0, t_0) = \begin{cases} (\text{on}, 0) & \text{if } m_c(0) \in M \\ (\text{off}, 0) & \text{if } m_c(0) \notin M \end{cases} \quad (4.4)$$

based on the initial mode $m_c(0) \in M_c$ of a component $c \in \mathbb{C}$ in focus. For $(state_i, t_i)$ and $(m_j, t_j)(m_{j+1}, t_{j+1}) \cdots = (m_i, t_i)_{(t_i, \infty)}$, we extend the state trace based on the sequence of specification pattern states from Figure 4.3 as follows:

1. If $state_i = \text{off}$:

$$(state_{i+1}, t_{i+1}) = \begin{cases} (\text{off}, t_j) & \text{if } m_j \notin M \\ (\text{pre}, t_j) & \text{if } m_j \in M \end{cases} \quad (4.5)$$

$$(state_{i+2}, t_{i+2}) = (\text{on}, t_{i+2}) \wedge t_{i+2} \in T_{pre} \quad \text{if } m_j \in M \quad (4.6)$$

2. If $state_i = \text{on}$:

$$(state_{i+1}, t_{i+1}) = \begin{cases} (\text{on}, t_j) & \text{if } m_j \in M \\ (\text{post}, t_j) & \text{if } m_j \notin M \end{cases} \quad (4.7)$$

4.2 Specification of Mode-Dependent Timing Behavior

$$(state_{i+2}, t_{i+2}) = (\text{off}, t_{i+2}) \wedge t_{i+2} \in T_{\text{post}} \quad \text{if } m_j \notin M \quad (4.8)$$

with

$$T_{\text{pre}} = [t_j, \min(t_j + D_{\text{pre}}, t_k \mid (m_k, t_k)_{M_c \setminus M, [t_{j+1}, \infty)}) \quad (4.9)$$

and

$$T_{\text{post}} = [t_j, \min(t_j + D_{\text{post}}, t_k \mid (m_k, t_k)_{M, [t_{j+1}, \infty)}) \quad (4.10)$$

In order to evaluate st at time t , we define a *specification state function* $S_{st} : \mathbb{T} \rightarrow \mathcal{S}$ with

$$S_{st}(t) = \begin{cases} s_0 & \text{if } \exists (s_1, t_1) \in (state_i, t_i)_{i \in \mathbb{N}} : t < t_1 \\ s_i & \text{if } \exists (s_i, t_i)(s_{i+1}, t_{i+1}) \in (state_i, t_i)_{i \in \mathbb{N}} : t_i \leq t < t_{i+1} \end{cases} \quad (4.11)$$

that returns the active specification state. Given a state trace st , a subset of states $\mathcal{S} \subseteq \mathcal{S}$, and a timed trace $(\sigma_i, t_i)_{i \in \mathbb{N}}$, the *specification state projection*

$$\text{pr}_{st, \mathcal{S}}((\sigma_i, t_i)_{i \in \mathbb{N}}) = \{(\sigma_i, t_i) \mid S_{st}(t_i) \in \mathcal{S}\} \quad (4.12)$$

extracts all events from the timed trace that occur while one of the states in $\mathcal{S}$ is active. Note that infinitely many traces st may exist for every change event trace $(m_i, t_i)_{i \in \mathbb{N}}$, because state changes may occur at any point in time within the pre- and post-phases. Finally, we define the *state trace function*

$$St : (\Sigma \times \mathbb{T})^\omega \rightarrow 2^{(\mathcal{S} \times \mathbb{T})^\omega}, \quad (4.13)$$

which returns all state traces for a given change event trace $(m_i, t_i)_{i \in \mathbb{N}}$. Since each specification pattern is assigned to exactly one component that has only a single mode port by definition, explicitly specifying $(m_i, t_i)_{i \in \mathbb{N}}$ is optional.

Specification Patterns

In the following, we will complete the extension for three example specification patterns that allow to specify (repetitive) event occurrences, latencies, and data age. The basic idea is to extend each specification pattern with a set of modes M in which it has to be fulfilled either by the environment or the behavior of the component itself. Additionally, pre- and post-phase durations D_{pre} and D_{post} are defined for each pattern.

Repetition Pattern The **Repetition** pattern specifies an infinite sequence of recurring events. It expresses that a given event tuple $\vec{\sigma} \in \Sigma \times \dots \times \Sigma$, referred to as *EventSet*, occurs every $P \in \mathcal{I}$ time units, possibly further delayed by up to $J \in \mathbb{T}$ time units. In addition, an optional offset of $O \in \mathcal{I}$ time units can be specified, which defines the set of possible time points of the first event occurrence:

EventSet occurs every P [with [offset O and]? jitter J]?.

4 Handling of Dynamic Architectures

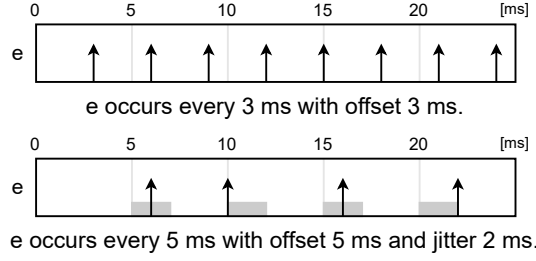


Fig. 4.4: Repetition pattern examples

We set $O = [0, 0]$ and $J = 0$ as default values in case some of the optional parts are omitted. Note that the pattern syntax has been slightly simplified for clarity. A complete definition of the **Repetition** pattern syntax can be found in Section A.4.1. In order to keep things simple, we require $J < \min(P)$. The **Repetition** pattern is a member of the class of instantaneous specifications and can thus be used either in assumptions or guarantees. According to MTSL 1.5, its mode-independent semantics is defined by a family of languages $L_{rep}(\vec{\sigma}, P, O, J)$, where $\vec{\sigma}$ represents the *EventSet*:

$$L_{rep} = \{(\vec{\sigma}, t_i)_{i \in \mathbb{N}} \mid t_i = u_i + j_i \wedge u_0 \in O \wedge u_{i+1} = u_i + p_i \wedge p_i \in P \wedge j_i \in [0, J]\} \quad (4.14)$$

Figure 4.4 depicts two examples of the **Repetition** pattern. In the upper part, a pattern instance with $\vec{\sigma} = e$ and a period/offset of 3 ms is shown. Therefore, the e events occur exactly with this periodicity. The lower part contains a **Repetition** pattern with $\vec{\sigma} = e$, a period/offset of 5 ms, and an additional jitter of 2 ms. In this case, L_{rep} includes all traces in which an e event occurs at a non-deterministic time in each jitter interval.

The syntax extensions for defining mode-dependent **Repetition** patterns are as follows:

$$\begin{aligned} & \text{EventSet occurs every } P \text{ [with [offset } O \text{ and]? jitter } J \text{]?} \\ & \text{in mode } M \text{ [pre } D_{pre} \text{]? [post } D_{post} \text{]?}. \end{aligned}$$

The additional components allow the definition of a set of modes M in which the pattern has to be fulfilled. Furthermore, the two optional parts **pre** D_{pre} and **post** D_{post} enable the explicit specification of corresponding pre- and post-phase durations.

Based on the results of the previous sections and the mode-independent **Repetition** semantics, the semantics of the mode-dependent **Repetition** pattern is defined as

$$\begin{aligned} L_{rep}^M = \{ \text{pr}_{st, \{on, post\}}((\vec{\sigma}, t_i)_{i \in \mathbb{N}}) \mid \exists st \in St \wedge t_i = u_i + j_i \\ \wedge u_0 \in t_{init} + O \wedge u_{i+1} = u_i + p_i \\ \wedge p_i \in P' \wedge j_i \in [0, J] \} \end{aligned} \quad (4.15)$$

such that $\exists (on, t_{init}) \in st : \nexists (on, t') \in st \wedge t' < t_{init}$ and

$$P' = \begin{cases} P & \text{if } \nexists (on, t_{on}) \in st : u_i < t_{on} \leq u_{i+1} \\ t_{on} - u_i + O & \text{if } \exists (on, t_{on}) \in st : u_i < t_{on} \leq u_{i+1} \end{cases}. \quad (4.16)$$

4.2 Specification of Mode-Dependent Timing Behavior

The timing behavior of an example mode-dependent **Repetition** pattern is shown in Figure 4.5. The *Output* events are specified to occur with a period/offset of 2 ms, but only in mode *A*. According to the semantics, the compliance of the component mode with the mode condition is evaluated at each (anticipated) time of occurrence $t_{eval} = t_{out}^\uparrow$ of the periodic event. If the component is in mode *A*, i.e., $m_c(t_{eval}) = A$, the *Output* event is written. Otherwise, the event is discarded. Since the default post-phase duration for instantaneous specifications is defined as $D_{post} = 0$, there are no “trailing” events.

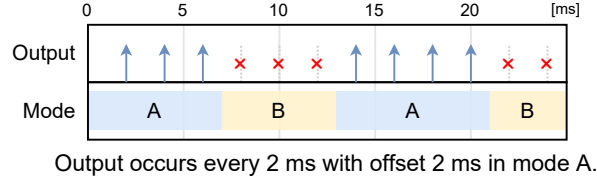


Fig. 4.5: Example trace of a mode-dependent **Repetition** pattern

Reaction Pattern The **Reaction** pattern expresses a “classical” latency between two events $\sigma_1, \sigma_2 \in \Sigma$, referred to as *EventSpec*₁ and *EventSpec*₂. It specifies that *EventSpec*₂ occurs $I \in \mathcal{I}$ time units after each occurrence of *EventSpec*₁:

$$\text{Reaction}(\text{EventSpec}_1, \text{EventSpec}_2) \text{ within } I.$$

Note that some of the available pattern parameters are omitted for clarity. A complete definition of the **Reaction** pattern syntax can be found in Section A.4.1. The pattern belongs to the class of reactive specifications and can thus only be used in guarantees. According to MTSL 1.5, its semantics is defined by $L_{rea}(\sigma_1, \sigma_2, I)$:

$$L_{rea} = \{(\sigma_i, t_i)_{i \in \mathbb{N}} \mid \forall(\sigma_1, t_i) \exists(\sigma_2, t_j) : t_j - t_i \in I\} \quad (4.17)$$

Note that both events $\sigma_1, \sigma_2 \in \Sigma$ may be replaced by corresponding virtual events $\sigma_1^{virt}, \sigma_2^{virt} \in \Sigma$, which are explicitly specified in terms of event specifications $es_1, es_2 \in \mathcal{ES}$ with $es_1 \rightsquigarrow \sigma_1^{virt}$ and $es_2 \rightsquigarrow \sigma_2^{virt}$ (see Section A.4.2). The combined use of **Reaction** patterns and event specifications is demonstrated in Section 4.2.5.

Figure 4.6 shows two examples of the **Reaction** pattern. The upper part depicts an example trace that results from a pattern instance with $\sigma_1 = e$, $\sigma_2 = f$, and a latency of 2 ms. In this case, all *f* events occur exactly 2 ms after the corresponding trigger events *e*. In the lower part, a **Reaction** pattern $\sigma_1 = e$, $\sigma_2 = f$, and a latency of 2 to 4 ms is shown. L_{rea} contains all traces in which an *f* event occurs 2 to 4 ms after an *e* event.

Analogous to the extensions of the **Repetition** pattern, the syntax of the **Reaction** pattern is enhanced by the definition of a set of modes *M* as well as the optional specification of corresponding pre- and post-phase durations D_{pre} and D_{post} :

$$\begin{aligned} &\text{Reaction}(\text{EventSpec}_1, \text{EventSpec}_2) \text{ within } I \\ &\text{in mode } M \text{ [pre } D_{pre} \text{]? [post } D_{post} \text{]?}. \end{aligned}$$

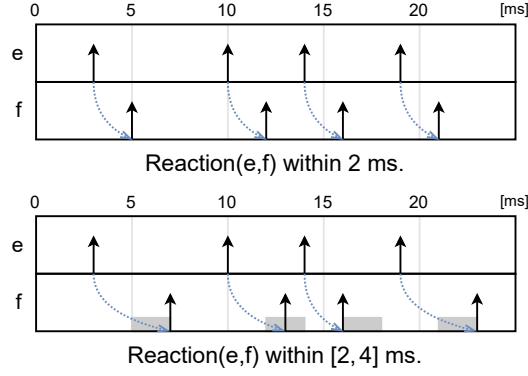


Fig. 4.6: Reaction pattern examples

The semantics of the pattern is adjusted accordingly. Each mode-dependent variant of the **Reaction** pattern specifies the occurrence of σ_2 events, which only become visible within the validity period and the subsequent post-phase, as a reaction to each trigger event σ_1 that occurs while one of the specified modes $m \in M$ is active:

$$\begin{aligned}
 L_{rea}^M = \{(\sigma_i, t_i)_{i \in \mathbb{N}} \mid \exists st \in St \wedge \forall (\sigma_1, t_i) \in pr_{st, \{on\}}((\sigma_1, t_i)_{i \in \mathbb{N}}) : \\
 \exists (\sigma_2, t_j) \in (\sigma_2, t_j)_{j \in \mathbb{N}} : t_j - t_i \in I \vee \\
 \exists (off, t_j) \in st : t_i < t_j \wedge t_j - t_i \in I\}
 \end{aligned} \quad (4.18)$$

The timing behavior of mode-dependent **Reaction** patterns is illustrated in Figure 4.7. The example pattern specifies that an *Output* event must occur 2 ms after each occurrence of an *Input* event in mode A. According to the semantics, the pattern's mode condition is evaluated at each time of occurrence $t_{eval} = t_{in}^\uparrow$ of the *Input* event. If the component is in mode A, i.e., $m_c(t_{eval}) = A$, the *Output* event is written with the specified delay. Otherwise, no *Output* event is generated. Since the default post-phase duration for reactive specifications is defined as $D_{post} = \infty$, no *Output* events are discarded.

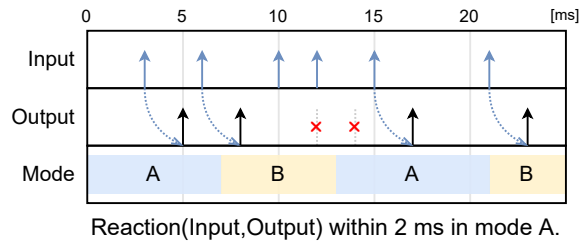


Fig. 4.7: Example trace of a mode-dependent Reaction pattern

Age Pattern The Age pattern specifies data age in terms of a “backward” latency between two events $\sigma_1, \sigma_2 \in \Sigma$, referred to as $EventSpec_1$ and $EventSpec_2$. It requires that $EventSpec_1$ has occurred $I \in \mathcal{I}$ time units before each occurrence of $EventSpec_2$:

4.2 Specification of Mode-Dependent Timing Behavior

Age(*EventSpec*₁, *EventSpec*₂) **within** *I*.

Similar to the definition of the **Reaction** pattern, both events $\sigma_1, \sigma_2 \in \Sigma$ may be replaced by corresponding virtual events $\sigma_1^{virt}, \sigma_2^{virt} \in \Sigma$ that are specified in terms of event specifications $es_1, es_2 \in \mathcal{ES}$ with $es_1 \rightsquigarrow \sigma_1^{virt}$ and $es_2 \rightsquigarrow \sigma_2^{virt}$ (see Section A.4.2).

Again, some of the available pattern parameters are omitted for clarity. A complete definition of the pattern syntax is included in Section A.4.1. The pattern belongs to the class of reactive specifications and can thus only be used in guarantees. According to MTSL 1.5, the semantics of the **Age** pattern is defined by $L_{age}(\sigma_1, \sigma_2, I)$:

$$L_{age} = \{(\sigma_i, t_i)_{i \in \mathbb{N}} \mid \forall(\sigma_2, t_i) \exists(\sigma_1, t_j) : t_j - t_i \in I\} \quad (4.19)$$

The pattern syntax is enhanced by the definition of a set of modes M as well as an optional specification of pre- and post-phase durations D_{pre} and D_{post} :

Age(*EventSpec*₁, *EventSpec*₂) **within** *I*
in mode M [**pre** D_{pre}]? [**post** D_{post}]?.

Each mode-dependent **Age** pattern requires that a σ_1 event has occurred before each σ_2 event that occurs while one of the specified modes $m \in M$ is active:

$$L_{age}^M = \{(\sigma_i, t_i)_{i \in \mathbb{N}} \mid \exists st \in St \wedge \forall(\sigma_2, t_i) \in pr_{st, \{on\}}((\sigma_2, t_i)_{i \in \mathbb{N}}) : \exists(\sigma_1, t_j) \in (\sigma_1, t_j)_{j \in \mathbb{N}} : t_j - t_i \in I\} \quad (4.20)$$

Fine-Grained Pre- and Post-Phase Durations

So far, we have only been able to specify common pre- and post-phase durations for all modes in which the pattern must be satisfied. The pre-phase of a pattern applies when the pattern becomes active, regardless of the previous mode. Likewise, the post-phase applies when the pattern becomes inactive, regardless of the subsequent mode.

In some application scenarios, however, it is desirable to define more fine-grained specifications. We hence further extend our patterns to be able to specify different pre- and post-phase durations depending on the previous (or subsequent) mode. To this end, we extend the pattern syntax by replacing **pre** D_{pre} and **post** D_{post} as follows:

EventSet **occurs every** P [**with** [**offset** O and]? **jitter** J]?
in mode M [**pre** *PreList*]? [**post** *PostList*]?.

Reaction(*EventSpec*₁, *EventSpec*₂) **within** I
in mode M [**pre** *PreList*]? [**post** *PostList*]?.

Age(*EventSpec*₁, *EventSpec*₂) **within** I
in mode M [**pre** *PreList*]? [**post** *PostList*]?.

The syntax of the *PreList* and *PostList* elements is defined by introducing two recursive lists of complementary, fine-grained pre- and post-phase durations:

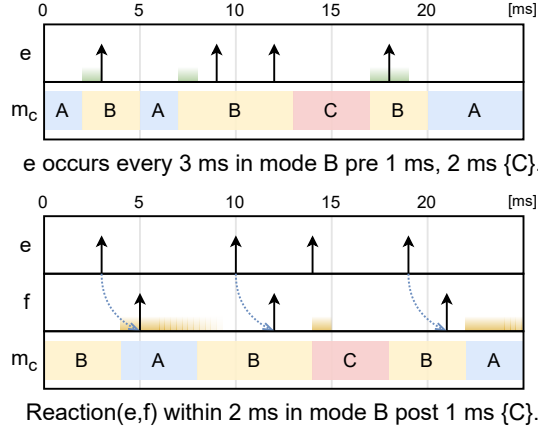


Fig. 4.8: Examples of fine-grained pre- and post-phases

$$\begin{aligned}
 PreList &::= D_{pre_i} [M_{pre_i}]? [, PreList]? \\
 PostList &::= D_{post_i} [M_{post_i}]? [, PostList]?
 \end{aligned}$$

With these extensions, we are able to specify optional mode sets $M_{pre_i} \subseteq M_c$ from which the component changes to a mode $m' \in M$ such that pre-phase duration D_{pre_i} applies. Similarly, the optional $M_{post_i} \subseteq M_c$ specify mode sets to which the component switches when leaving mode set M , taking into account the post-phase duration D_{post_i} . Note that any number of pairs $D_{pre_i} [M_{pre_i}]?$ and $D_{post_i} [M_{post_i}]?$ may be specified.

Figure 4.8 illustrates two examples of specification patterns with fine-grained pre- and post-phase durations. The upper part depicts an example trace of a **Repetition** pattern with a period of 3 ms and fine-grained pre-phases, which are highlighted in green. The specified duration of 1 ms applies to all pre-phases (see $t = 2$ ms and $t = 7$ ms), except for mode transitions from C to B . Here, the specified duration is 2 ms (see $t = 17$ ms). Consequently, the e events expected at $t = 3$ ms and $t = 18$ ms may or may not occur. In the lower part, an example trace of a **Reaction** pattern with a latency of 2 ms and fine-grained post-phases is shown. All post-phases are highlighted in orange. The pattern specifies that an f event must occur for each trigger event e . This is true even if the mode in which the pattern is active has already been left, provided that the post-phase duration has not yet expired (see $t = 5$ ms). Recall that the post-phase duration of **Reaction** patterns is ∞ by default. As a result of the post-phase duration of 1 ms for transitions from modes B to C , reactions to trigger events may be suppressed (see $t = 14$ ms).

The extensions still allow to specify common pre- and post-phase durations D_{pre} and D_{post} without using M_{pre} and M_{post} . If no other elements are specified as part of the *PreList* and *PostList* lists, the duration applies to all modes in $M_c \setminus M$, and thus $M_{pre} = M_c \setminus M$ and $M_{post} = M_c \setminus M$ are defined implicitly. Otherwise, it applies to all modes in M_c for which no other M_{pre_i} (or M_{post_i}) is specified, and M_{pre} (or M_{post}) is

4.2 Specification of Mode-Dependent Timing Behavior

assumed to be set accordingly. In addition, the following conditions must be fulfilled by all $M_{x_{i/j}}$, representing either pre- or post-phase mode sets $M_{pre_{i/j}}$ or $M_{post_{i/j}}$:

$$\begin{aligned} M_{x_i} &\subseteq M_c \\ M \cap M_{x_i} &= \emptyset \\ \forall i \neq j : M_{x_i} \cap M_{x_j} &= \emptyset \end{aligned} \quad (4.21)$$

Finally, the semantics is adapted by replacing D_{pre} in Equation 4.9 with D'_{pre} , i.e.,

$$T_{pre} = [t_j, \min(t_j + D'_{pre}, t_k \mid (m_k, t_k)_{M_c \setminus M, [t_{j+1}, \infty)})) \quad (4.22)$$

with

$$D'_{pre} = \begin{cases} D_{pre_1} & \text{if } m_{j-1} \in M_{pre_1} \\ \vdots & \\ D_{pre_i} & \text{if } m_{j-1} \in M_{pre_i} \\ D_{pre} & \text{otherwise} \end{cases} \quad (4.23)$$

and D_{post} in Equation 4.10 with D'_{post} , i.e.,

$$T_{post} = [t_j, \min(t_j + D'_{post}, t_k \mid (m_k, t_k)_{M, [t_{j+1}, \infty)})) \quad (4.24)$$

with

$$D'_{post} = \begin{cases} D_{post_1} & \text{if } m_j \in M_{post_1} \\ \vdots & \\ D_{post_i} & \text{if } m_j \in M_{post_i} \\ D_{post} & \text{otherwise} \end{cases} \quad (4.25)$$

4.2.5 Application Example

To illustrate the approach for handling operating modes, we consider a *Longitudinal Control System* (LCS), which was previously referred to as *Adaptive Cruise Control with Collision Avoidance* (ACCwCA) [370, 371]. The system was designed and implemented in the context of the Step-Up!CPS project [557], and is described in detail in [55]. The main focus was on the timing behavior of the system, so the overall architecture and time values were already fixed. An issue that became apparent in previous work was the lack of ability to specify operating modes in order to reflect the system behavior more accurately. The example presented here can therefore be seen as an attempt to address this issue.

The component model of the system is depicted in Figure 4.9. The example focuses on the early phases of system design, where the functional viewpoint is paramount. However, the concepts are not limited to these phases or this particular perspective, but can also be used in later development stages down to the detailed hardware/software design.

The LCS is an ADAS that has four modes, namely *Idle* (I), *Cruise* (C), *Follow* (F), and *Evade* (E). Initially, the system is in *Idle* mode. It switches into *Cruise* mode when being

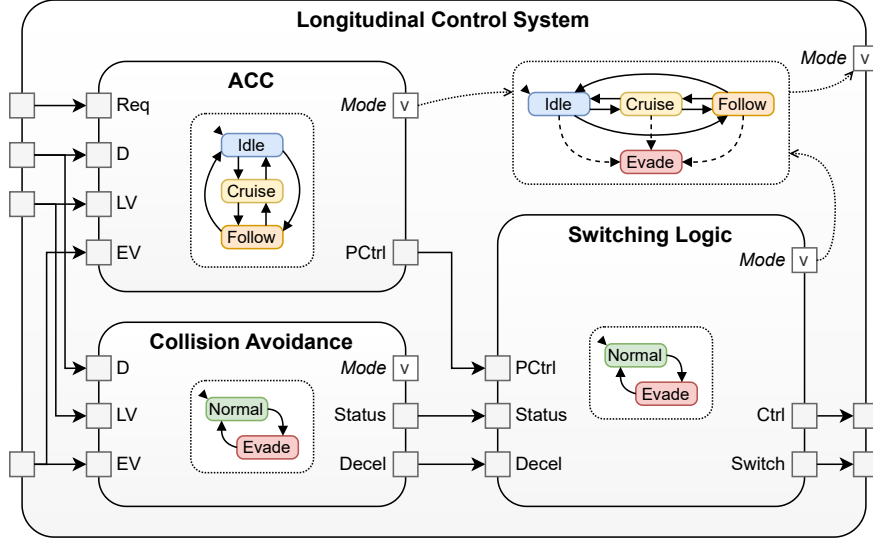


Fig. 4.9: Component model of the *Longitudinal Control System*

activated, where it maintains a constant speed set by the driver. If a slower vehicle is ahead of the ego vehicle, the *Follow* mode is activated, and the LCS adapts the speed to that of the leading vehicle in order to maintain a safe minimum distance. Additionally, the component provides a collision avoidance functionality. If the perceived distance between the leading and the ego vehicle falls below a safety-critical threshold, the high-priority *Evade* mode is activated to perform an emergency braking maneuver.

The top-level specification of the LCS is shown in Table 4.1. An upstream sensor processing unit periodically provides the distance D to the vehicle ahead and the velocity LV of the leading vehicle (line 1). The ego velocity EV is received from another external system (line 2). With the occurrence of a *Req* event in mode *Idle*, the system is activated (lines 3–4). After activation, the LCS computes regular updates of the control values $Ctrl$ as well as *Switch* values that indicate how the control values are to be interpreted. A transition from the *Idle* to the *Evade* mode is triggered automatically in safety-critical situations, i.e., without the need for manual activation (line 5). Depending on the active mode, the system shows different timing behaviors. This is partly to save resources and partly to avoid unnecessary restrictions in the specifications. In *Cruise* mode, the system reacts to its inputs within 245 to 250 ms (line 6). In modes *Follow* and *Evade*, the reaction requires a shorter period of time, i.e., between 195 and 200 ms (lines 7–9).

Taking into account the perceived traffic situation, the LCS switches between *Cruise*, *Follow*, and *Evade* until it is deactivated by a driver request (lines 10–11). To ensure continuous control, $Ctrl$ and *Switch* are provided every 3 to 7 ms in all operating modes other than *Idle* (line 12). When switching from *Idle* to *Cruise*, *Follow*, or *Evade*, the internal components need some time to adapt their behavior to the requested mode. Since a maximum delay of 30 ms in providing the first *Switch* and $Ctrl$ events is acceptable, a corresponding pre-phase duration of 30 ms has been defined (line 12).

4.2 Specification of Mode-Dependent Timing Behavior

Tab. 4.1: Top-level specification of the *Longitudinal Control System*

A	$\{D, LV\}$ occurs every 40 ms with jitter 5 ms.	1
	EV occurs every 10 ms with jitter 2 ms.	2
	Req occurs every $(0, \infty)$ ms with offset.	3
G	Reaction($Req, set(Mode)$) within $(0, 5]$ ms in mode <i>I</i> .	4
	Reaction($(EV, \{D, LV\}), set(Mode, E)$) within $(0, 5]$ ms in mode <i>I</i> .	5
	Reaction($(EV, \{D, LV\}), \{Ctrl, Switch\}$) within $[245, 250]$ ms in mode <i>C</i> post 0 ms <i>I</i> .	6
	Reaction($(EV, \{D, LV\}), Ctrl$) within $[195, 200]$ ms in mode $\{F, E\}$ post 0 ms <i>I</i> .	7
	Reaction($(EV, \{D, LV\}), Switch$) within $[195, 200]$ ms in mode <i>F</i> post 0 ms <i>I</i> .	8
	Reaction($(EV, \{D, LV\}), Switch.2$) within $[195, 200]$ ms in mode <i>E</i> post 0 ms <i>I</i> .	9
	Reaction($(EV, \{D, LV\}), set(Mode)$) within $(0, 5]$ ms in mode $\{C, F, E\}$.	10
	Reaction($Req, set(Mode, I)$) within $(0, 5]$ ms in mode $\{C, F, E\}$.	11
	$\{Ctrl, Switch\}$ occurs every $[3, 7]$ ms in mode $\{C, F, E\}$ pre 30 ms post 25 ms <i>I</i> .	12
	$\{AdaptiveCruiseControl.I, SwitchingLogic.N\}$ maps to <i>I</i> .	13
	$\{AdaptiveCruiseControl.C, SwitchingLogic.N\}$ maps to <i>C</i> .	14
	$\{AdaptiveCruiseControl.F, SwitchingLogic.N\}$ maps to <i>F</i> .	15
	$SwitchingLogic.E$ maps to <i>E</i> .	16

After the component switches to *Idle* mode, it must not provide any further *Ctrl* and *Switch* events. In order to take this into account, we specify a fine-grained post-phase duration of 0 ms for transitions to the *Idle* mode (lines 6–9). However, a post-phase duration of 25 ms remains for the periodic outputs events *Ctrl* and *Switch* (line 12), which does not comply with the previous requirements. Note that this situation arises for consistency reasons and is explained in more detail in Section 4.2.5b.

Component-Level Specifications

The LCS consists of three subcomponents, namely *Adaptive Cruise Control* (*ACC*), *Collision Avoidance* (*CA*), and *Switching Logic* (*SL*). All components have their own internal modes as depicted in Figure 4.9. Based on the definitions from Section 4.2.3, the top-level modes are mapped to combinations of the composed lower-level modes by using the mode mapping function $\mu_{lcs} : M_{acc} \times M_{ca} \times M_{sl} \rightarrow M_{lcs}$. The detailed mapping is shown in Table 4.2. Each line specifies a combination of modes of the *ACC*, *CA*, and *SL* components as well as the resulting mode of the top-level component.

Tab. 4.2: Mode combinations and top-level modes

M_{acc}	M_{ca}	M_{sl}	M_{lcs}
<i>Idle</i>	*	<i>Normal</i>	<i>Idle</i>
<i>Cruise</i>	*	<i>Normal</i>	<i>Cruise</i>
<i>Follow</i>	*	<i>Normal</i>	<i>Follow</i>
*	*	<i>Evade</i>	<i>Evade</i>

* = “any value”

Tab. 4.3: Excerpt from an example *Mode Logic* specification

G	Reaction(change(<i>ACC.Mode</i> , <i>I</i>),set(<i>Mode</i> , <i>I</i>)) within ...	1
	Reaction(change(<i>ACC.Mode</i> , <i>C</i>),set(<i>Mode</i> , <i>C</i>)) within ...	2
	Reaction(change(<i>ACC.Mode</i> , <i>F</i>),set(<i>Mode</i> , <i>F</i>)) within ...	3
	Reaction(change(<i>SL.Mode</i> , <i>E</i>),set(<i>Mode</i> , <i>E</i>)) within ...	4

The combinations show that the mode of the overall system depends primarily on the lower-level modes of the *ACC* and *SL* components. The *CA* affects the mode of the *LCS* only indirectly through output events that cause mode changes of the *SL*. The dependencies between the modes at different hierarchy levels presented in Table 4.2 are also encoded in the top-level specification (see Table 4.1, lines 13–16). Note that the interaction of the individual components is detailed below. As a result of the combinatorics, mode combinations that lead to inconsistencies between different specifications may arise, especially in transition phases. However, since these combinations are transient, a careful definition of pre- and post-phase durations D_{pre} and D_{post} effectively prevents inconsistent behavior. The effects are explained in more detail in the following subsections.

As indicated in Section 4.2.3, the mode mapping and corresponding mode transitions as shown in the upper right part of Figure 4.9 could also be realized in terms of a dedicated *Mode Logic* component. Such a component would be equipped with input ports that are connected with the mode ports of the *ACC* and *SL* components. An output port would present the aggregated mode of the *LCS* component. In turn, the mapping of the input modes and the aggregated top-level mode could be specified by means of a specification similar to those of the other components. An excerpt from an example *Mode Logic* specification is included in Table 4.3. In the example, the guarantee contains four **Reaction** patterns that pass mode changes of the *ACC* component to the output (lines 1–3), as well as a change of the *SL* to the *Evade* mode (line 4). However, if the *SL* component switches from *Evade* mode back to *Normal*, the mode previously set by the *ACC* component must be reestablished as the new top-level mode. This would require to store the *ACC* mode in terms of an internal state of the *Mode Logic* component. Conceptually, this could be achieved by introducing a variable in the component that stores the mode, which is supported by the framework through variable ports. Note that corresponding specification patterns would be required, which were defined in [369].

As a final remark, there is a fundamental difference between the two approaches. While mapping functions “realize” mode changes instantly, components with timing specifications do not allow instantaneous reactions. This has an impact on the top-level specification, as mode changes become visible to the environment only with a delay.

Adaptive Cruise Control The *Adaptive Cruise Control* (*ACC*) component computes control values in terms of *P Ctrl* events. Its specification is given in Table 4.4. The component has three modes, namely *Idle* (*I*), *Cruise* (*C*), and *Follow* (*F*), and is enabled and disabled by *Req* events (lines 4–5). Situation-dependent changes between the *Cruise* and *Follow* modes are possible at any time (line 6). In *Cruise* mode, the component

4.2 Specification of Mode-Dependent Timing Behavior

Tab. 4.4: Specification $\mathcal{C}_{acc}$ of the *Adaptive Cruise Control* component

A	$\{D, LV\}$ occurs every 40 ms with jitter 5 ms in mode $\{C, F\}$.	1
	EV occurs every 10 ms with jitter 2 ms in mode $\{C, F\}$.	2
	Req occurs every $(0, \infty)$ ms with offset.	3
G	Reaction(Req, set(Mode, $\{C, F\}$)) within $(0, 1]$ ms in mode I.	4
	Reaction(Req, set(Mode, I)) within $(0, 1]$ ms in mode $\{C, F\}$.	5
	Reaction((EV, $\{D, LV\}$), set(Mode)) within $(0, 1]$ ms in mode $\{C, F\}$.	6
	Reaction((EV, $\{D, LV\}$), PCtrl) within $[220, 225]$ ms in mode C post 200 ms, 0 ms I.	7
	Reaction((EV, $\{D, LV\}$), PCtrl) within $[170, 175]$ ms in mode F post 0 ms I.	8
	PCtrl occurs every 5 ms in mode $\{C, F\}$.	9

Tab. 4.5: Specification $\mathcal{C}_{ca}$ of the *Collision Avoidance* component

A	$\{D, LV\}$ occurs every 40 ms with jitter 5 ms.	1
	EV occurs every 10 ms with jitter 2 ms.	2
G	Reaction((EV, $\{D, LV\}$), set(Mode)) within $(0, 1]$ ms.	3
	Reaction(change(Mode, E), Status.1) within $(0, 1]$ ms in mode N.	4
	Reaction((EV, $\{D, LV\}$), Decel) within $[170, 175]$ ms in mode E post 0 ms.	5
	Reaction(change(Mode, N), Status.0) within $(0, 1]$ ms in mode E.	6
	Decel occurs every 5 ms in mode E post 5 ms.	7

needs 220 to 225 ms to respond to inputs by providing corresponding output events, while in *Follow* mode it needs to respond faster to react timely to the behavior of the vehicle ahead (lines 7–8). In case the ACC leaves the *Cruise* mode, we limit the reaction time to input events that have already been received to a maximum latency of 200 ms (line 7). When the ACC switches to *Idle*, no further PCtrl events are generated (lines 7–8). PCtrl events are provided every 5 ms in *Cruise* and *Follow* modes (line 9).

Collision Avoidance The *Collision Avoidance* (CA) component, whose specification is given in Table 4.5, checks whether the execution of an emergency braking maneuver is required. It has *Normal* (N) and *Evade* (E) modes. As long as there is no hazardous situation, the CA is in *Normal* mode. If a violation of the critical distance is detected, it switches to *Evade* (line 3) and transmits a *Status* event with a value of 1 to inform the SL component (line 4). The CA requires 170 to 175 ms to respond to inputs (line 5). After completing of the braking maneuver, it switches back to *Normal* mode and emits a *Status* event with a value of 0 (line 6). To prevent the system from running into inconsistent states, additional *Decel* events are provided for 5 ms after leaving the *Evade* mode (line 7).

Switching Logic The specification of the *Switching Logic* (SL) is presented in Table 4.6. The component has *Normal* (N) and *Evade* (E) modes. It receives PCtrl, Decel, and Status events as inputs (lines 1–3). Based on these values, it switches between *Normal* and *Evade* modes. In *Normal* mode, it forwards the received PCtrl and provides corresponding Switch events to enable a correct interpretation of the control values (line 4). A mode change to *Evade* is triggered by receiving a Status event with value 1 (line 5). In *Evade*

Tab. 4.6: Specification $\mathcal{C}_{sl}$ of the *Switching Logic* component

A	<i>P</i> Ctrl occurs every $(0, \infty)$ ms with offset.	1
	<i>Status</i> occurs every $(0, \infty)$ ms with offset.	2
	<i>Decel</i> occurs every 5 ms in mode <i>E</i> pre 5 ms.	3
G	Reaction(<i>P</i> Ctrl, {Ctrl, Switch}) within 25 ms in mode <i>N</i> .	4
	Reaction(<i>Status</i> .1, set(Mode, <i>E</i>)) within $(0, 1]$ ms in mode <i>N</i> .	5
	Reaction(<i>Decel</i> , {Ctrl, Switch.2}) within 25 ms in mode <i>E</i> .	6
	Reaction(<i>Status</i> .0, set(Mode, <i>N</i>)) within $(0, 1]$ ms in mode <i>E</i> .	7

mode, it passes the *Decel* values with a *Switch* value of 2 (line 6). The occurrence of a *Status* event with a value of 0 triggers a transition to *Normal* mode (line 7).

Resulting System Behavior

Table 4.2 shows that the mode of the top-level component is highly dependent on the mode of the *SL* component. Based on the descriptions in Section 4.2.5a, it is clear that the individual modes of the components have different criticality for the system behavior. If the safety-critical distance is violated, the highest priority is to perform an emergency braking maneuver to prevent the system from a collision. Therefore, by changing the mode of the *SL* component from *Normal* to *Evade*, the control values provided by the *CA* component are forwarded to the top-level output ports and the values of the *ACC* component are ignored. The mode change is propagated to the top-level component, as shown in Figure 4.10. The priority between the output values thus results indirectly from the combinatorics and the mapping between top-level and lower-level modes.

Figure 4.10 shows example traces that illustrate the system behavior resulting from the given specifications. Here, we focus on the evolution of component modes as well as the propagation of mode changes through the overall system. Again, active pre- and post-phases are highlighted in green and orange color. The system is activated at $t = 0$ ms and deactivated at $t = 402$ ms. The top-level mode, which is presented in the bottom line, directly results from the composed lower-level modes as described in Section 4.2.5a (see Table 4.2). Note that *Cruise* and *Follow* modes are ignored, if the high-critical *Evade* mode is active. In addition, the need to define appropriate pre- and post-phases is demonstrated. For example, in the top-level specification, a pre-phase is required to account for delays in providing initial results (see Table 4.1, line 12). In addition, post-phases in $\mathcal{C}_{acc}$ ensure that the control values calculated in *Follow* mode are not overwritten by outdated values from the previous *Cruise* mode (see Table 4.4, line 7).

In the following, we will examine two important non-trivial details of the system behavior more closely. Let us first take a look at the parts of Figure 4.10 that are highlighted by overlays ①, ②, and ③. Based on the specifications of the *ACC* and *CA* components, which generate periodic output events every 5 ms in most of their modes (see Table 4.4, line 9 and Table 4.5, line 7), and the non-varying delay of 25 ms introduced by the *SL* component (see Table 4.6, lines 4 and 6), we could conclude that the output events should also occur every 5 ms. Looking at the highlighted parts of the *Ctrl* and *Switch* lines,

4.2 Specification of Mode-Dependent Timing Behavior

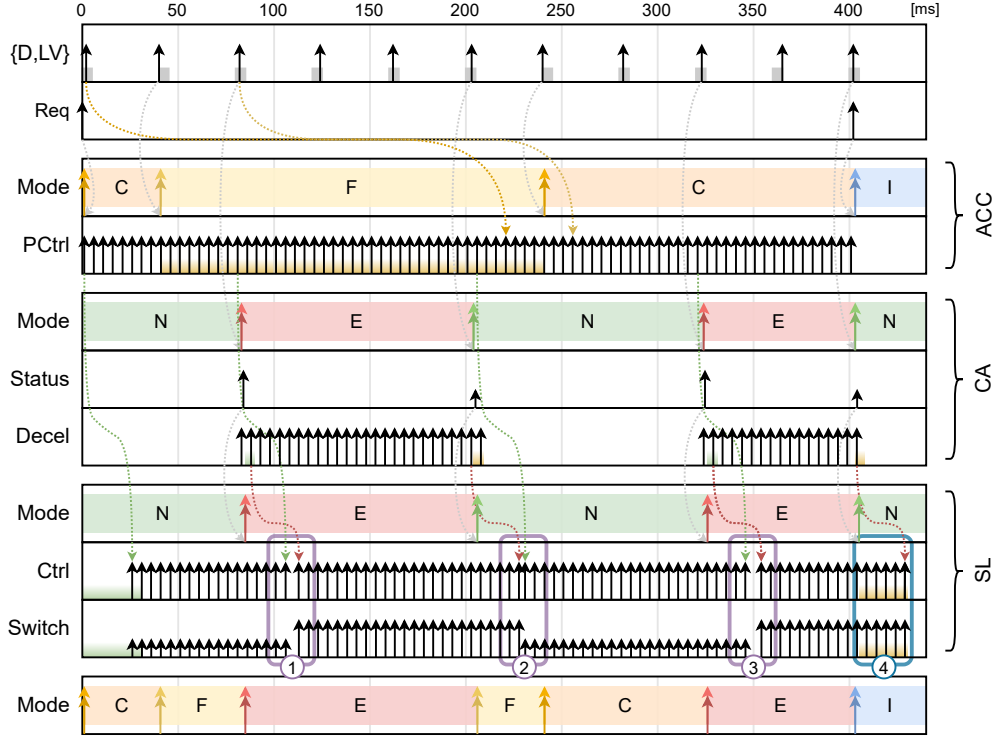


Fig. 4.10: Example traces of the *Longitudinal Control System*

we can see, however, that both outputs do not occur periodically every 5 ms as expected, but later or earlier. This behavior is a result of mode changes of the *SL* component that cause variations in the timing of the control values. Due to these variations, the top-level specification had to be adjusted to guarantee repetitive occurrences of *Ctrl* events with a period of $[3, 7]$ ms, instead of the expected 5 ms (see Table 4.1, line 12).

The second detail refers to the system when switching to the *Idle* mode. The corresponding part is highlighted by annotation ④ in Figure 4.10. As described in Section 4.2.5a, the system should not provide any further outputs after switching to the *Idle* mode. Contrary to the intended behavior, some trailing *Ctrl* and *Switch* events occur at the outputs after a transition to the *Idle* mode. According to the specification, this happens within a maximum time span of 25 ms. Due to its function as a simple arbiter, the output behavior of the *SL* component cannot be restricted by refining the post-phase durations in the specification. To this end, the integration of a mechanism to disable the *SL* component would be required. An alternative solution would be to integrate an additional component that filters the outputs of the *SL* depending on the modes of the internal components.

4.2.6 Discussion

The application example illustrates the expressiveness of the extensions and outlines possible applications for mode-dependent specifications. In this context, the specification

patterns considered in Section 4.2.4 already allow a detailed formulation of real-time requirements that address dynamic system behavior by means of operating modes, mode transitions, and mode-dependent behavior. Due to the variety of possible applications and the complexity of the combinatorics of inter-mode relations, the use of mode-dependent specifications requires a thorough consideration of the resulting proof obligations.

For components on the same hierarchy level that interact with each other, it is important to ensure that transitions between their individual operating modes take place in a coordinated manner. In particular, it is important to verify that the pre- and post-phase durations of all transitions defined in different specifications are consistent.

When considering mode relations across multiple hierarchy levels, the definition of a meaningful mapping is crucial to enable error-free composition. In order to achieve a consistent specification of the behavior of the overall system, all mode mapping functions μ_c with $c \in \mathbb{C}$ must be well-defined, i.e., they must satisfy the following conditions:

1. Each mapping μ_c must be *complete*, i.e., the modes of each higher-level component c must result from tuples of the individual modes of its lower-level components $sub_1, \dots, sub_n \in \mathbb{C} \setminus \{c\}$ with $n \in \mathbb{N}$, i.e., $\mu_c : M_{sub_1} \times \dots \times M_{sub_n} \rightarrow M_c$.
2. Each mapping μ_c must be *unambiguous*, i.e., it must indeed specify a function from the combination of the lower-level modes to a well-defined higher-level mode.

If both conditions are met, the mode-dependent specification of a higher-level component can be designed in the same way as the corresponding specifications of lower-level components. Checking the above conditions for a given μ_c is easily accomplished.

With regard to consistency of specifications at different hierarchy levels, the usual proof obligations for CBD [70] can be applied. In this context, the behavior specified in lower-level modes must refine the behavior of their associated higher-level modes. As a consequence, it is necessary to prove that the behavior in all operating modes is correct with respect to the assumed environments and that valid transitions are taken whenever the environment changes. The underlying contract theory (see Section 2.1.3) allows to check the validity of refinement relations by means of VITs [164, 166]. VITs can also be used to check proper interaction of components at the same hierarchy level with respect to their mode-dependent specifications. For *compatibility*, we can check whether the composition of multiple components refines the contract $(false, true)$. If so, then there is no environment in which the corresponding components interact with each other in a correct way, i.e., their specifications are not compatible. Similarly, *consistency* can be verified by checking whether the composed specifications refines the contract $(true, false)$. If so, this means that the composition does not yield a valid implementation.

For most specification formalisms, verification is expensive for larger system designs. CBD, on the other hand, allows managing complexity through a divide-and-conquer strategy along component hierarchies. Virtual integration testing ensures that higher-level specifications are satisfied if their composites do so, thus allowing to divide verification tasks into smaller subtasks. Contracts as a means of specifying requirements can be combined with modeling languages such as Matlab/Simulink and statecharts in order to verify contract satisfaction [383, 444]. In addition, contracts as formal specifications

are able to cover different functional, timing, and safety aspects in industrially relevant applications [340, 451, 452]. Besides tool support in the form of generalized verification engines such as UPPAAL [69] and SPIN [295], specialized tools like OCRA [149] exist.

Viewed from a different perspective, it becomes clear that the concepts are not only suitable for handling timing specifications, but can also be used to formalize safety properties. A first step in this direction was the use of mode-dependent specifications for modeling failure modes and corresponding safety mechanisms as part of the requirements of the *Model-based Safety Assessment and Traceability* (MobSTr) dataset [554], which is available on GitHub [555]. The results have recently been published in [58], along with the presentation of prototypical tool support for this type of specifications.

4.3 Modeling of Dynamic Reconfiguration

Dynamic reconfiguration is commonly understood as a “*mechanism for applying changes [in] the structure and/or behavior of a system during its execution (i.e., at run[-]time) with minimal or no interruption, while preserving consistency and integrity within the system*” [130, p. 31]. From a real-time perspective, the concept of operating modes and their instantiation in terms of mode-dependent timing specifications already cover most of the relevant effects. Thus, in the context of this work, dynamic reconfiguration is considered as a (series of) mode transition(s). The newly introduced MTSL 1.5+MC specification patterns allow the dynamic adaptation of component behavior, the exchange of information about the internal state of components via event ports, and the composition of lower-level modes to define higher-level modes. In addition, the verification of correct refinement of mode-dependent specifications between different hierarchy levels is supported.

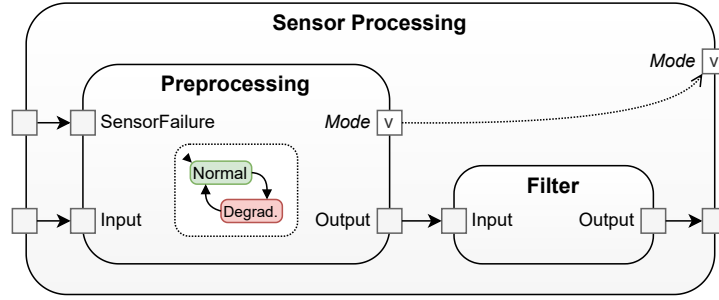
The developed MTSL 1.5+MC specifications can also be applied to address dynamic behavior on SoCPS level, i.e., by defining a set of top-level modes that are defined based on a combination of a number of lower-level modes of individual CSs. In the sense of TCADs, the resulting model of SoCPS behavior would correspond to the definition of a set of top-level activities, which are each composed of sets of lower-level actions. In order to coordinate behavioral adaptations—and thus dynamic reconfigurations—among ensembles of components, Kröger and Fränzle have developed consistent extensions [369], which have already been discussed above. From the perspective of SoCPS design, such an approach could be applied to structure cooperative processes that are coordinated by higher authorities, such as appropriately certified and authorized TMSs that coordinate mode changes of interacting CHAVs in critical areas, e.g., near intersections, tunnels, or construction sites. To support coordination-based cooperation approaches, we extend the system model revised in Section 4.2.2 by so-called *variable signals* $s_{var} : p_{v_1} \mapsto \{p_{v_2}, \dots\}$, which can be used to connect two or more variable ports $p_{v_1}, p_{v_2} \in P_{var}$. The extensions enable the explicit specification of components that can directly access the internal state of others, such as the mode management components considered in [369].

Consistent communication is an essential prerequisite for ensuring error-free interaction between cooperating CSs. As part of the process of dynamic reconfiguration, the used communication protocols must therefore be adapted accordingly. In this context, situations

Tab. 4.7: Top-level specification of the *Sensor Processing*

A	<i>Input occurs every 50 ms with offset [0, 50] ms and jitter 2 ms.</i>	1
	<i>SensorFailure occurs every [2000, ∞) ms with offset [2000, ∞) ms.</i>	2
G	<i>Reaction(Input, Output) within [5, 20] ms in mode Normal.</i>	3
	<i>Reaction(SensorFailure, set(Mode, Degraded)) within 1 ms in mode Normal.</i>	4
	<i>Reaction(Input, Output) within [20, 40] ms in mode Degraded.</i>	5
	<i>Preprocessing.Normal maps to Normal.</i>	6
	<i>Preprocessing.Degraded maps to Degraded.</i>	7

in which there is no information about the modes of interacting components are particularly challenging. Given such a situation, the question arises as to how consistent behavior can still be ensured. A simple solution to this problem is the use of sufficiently abstract assumptions about the behavior of interacting components. For example, consider a mode-dependent *Sensor Processing*, whose component model is presented in Figure 4.11. The system processes periodic *Input* events from an upstream sensor unit, which fails from time to time. An integrated safety mechanism detects any failures that occur and reports them to the *Sensor Processing* by triggering a *SensorFailure* event.

**Fig. 4.11:** Component model of the *Sensor Processing*

The top-level specification is shown in Table 4.7. The system expects *Input* events every 50 ms with a jitter of 2 ms (line 1), and processes them within 5 to 20 ms in *Normal* mode (line 3). After the occurrence of a *SensorFailure* event (line 2), the component switches to *Degraded* mode (line 4), in which a fallback processing task based on estimated values can only be performed slower, resulting in longer execution times of 20 to 40 ms (line 5). The lower-level *Preprocessing* component, whose specification is listed in Table 4.8, realizes the main functionality. It is responsible for taking over the first three top-level guarantees by computing *Output* events based on *Input* events in both modes (lines 1/3), including proper reactions to *SensorFailure* events (line 2). Afterwards, the processed values are smoothed by a *Filter* component, which takes another 2 to 10 ms (Table 4.9, line 2).

The key for capturing the varying output behavior of the *Preprocessing* component lies in the formulation of the assumption of the *Filter* component (Table 4.9, line 1). Although it is not mode-dependent, it covers all possible output behaviors through a careful selection of the jitter value. In principle, this design pattern works well, but overapproximates the actual behavior. An alternative solution is to explicitly notify other

4.3 Modeling of Dynamic Reconfiguration

Tab. 4.8: Specification $\mathcal{C}_{pre}$ of the *Preprocessing* component

A	<i>Input occurs every 50 ms with offset [0, 50] ms and jitter 2 ms.</i>	1
	<i>SensorFailure occurs every [2000, ∞) ms with offset [2000, ∞) ms.</i>	2
G	<i>Reaction(Input, Output) within [3, 10] ms in mode Normal.</i>	3
	<i>Reaction(SensorFailure, set(Mode, Degraded)) within 1 ms in mode Normal.</i>	4
	<i>Reaction(Input, Output) within [18, 30] ms in mode Degraded.</i>	5

Tab. 4.9: Specification $\mathcal{C}_{flt}$ of the *Filter* component

A	<i>Input occurs every 50 ms with offset [3, 53] ms and jitter 29 ms.</i>	1
G	<i>Reaction(Input, Output) within [2, 10] ms.</i>	2

components of mode changes. Note that this approach was used in previous examples, such as the example in Figure 4.1 and the LCS example in Section 4.2.5.

The supreme discipline in handling dynamic reconfiguration is the realization of higher-level operations for adapting dynamic architectures, such as the *Addition* and *Removal* operations introduced in Section 4.1. As a fundamental system property of dynamic systems, cooperative SoCPSs should allow certain types of systems to enter and leave the group of interacting CSs, thereby entering and leaving the system boundaries. For instance, consider the sequence of example driving scenarios presented in Figure 1.2. During its journey through inner-city traffic, the CHAV in focus continues to move in the physical world and comes within range of different TMSs. Assuming that the infrastructure systems support the initiation of cooperative left turns or other cooperative driving maneuvers, the same driving functions and cooperation approaches could be used to interact with different cooperation partners and TMSs. Depending on the perspective, either the CHAV enters and leaves the CSGs of the individual TMSs, or vice versa. In both cases, the CSs to interact with may or may not be available at a given time.

As we have found, dynamic reconfiguration involves both behavioral changes as well as adaptations of the available interfaces. However, the explicit handling of interface reconfigurations is a missing puzzle piece. In the following, we discuss possible extensions of the framework to distinguish between *behavioral reconfigurations*, i.e., dynamic adaptations of component behavior, and *interface reconfigurations*, i.e., adaptations of the component's interfaces. Note that the required extensions for handling dynamic reconfiguration are only outlined here, and are not considered in the following chapters. However, continuing the extensions would be an interesting topic for future research activities.

The general idea of the envisioned extensions is to use the basic concepts for specifying operating modes to handle both types of adaptations. In this context, we refer to “normal” modes defined in Section 4.2 as *component modes* $m^\circ \in M^\circ$, and denote “classical” mode ports as *component mode ports* $p_m^\circ \in P_m^\circ$ with $P_m^\circ \subseteq P_{var}$. Consequently, the second class of modes is referred to as *interface modes* $m^{\leftrightarrow} \in M^{\leftrightarrow}$, which are stored in *interface mode ports* $p_m^{\leftrightarrow} \in P_m^{\leftrightarrow}$ with $P_m^{\leftrightarrow} \subseteq P_{var}$. The latter describes which interfaces are “active”, i.e., which interacting systems are available. Note that the value domain $V_{p_m^{\leftrightarrow}}$ consists of the power set of all identifiers of systems that could potentially be interacted with. The

extensions could be completed by redefining the equations from Sections 4.2.3 and 4.2.4 to include tuples of component and interfaces modes instead of “simple” modes.

From an architectural point of view, we imagine that a top-level architecture similar to the one from Figure 3.3 is defined as a starting point. Afterwards, we could determine a set of *dynamic components* $\mathcal{C}_{dyn} \subseteq \mathcal{C}$, which are subject to dynamic reconfiguration. In other words, these components can adapt their behavior and interfaces depending on whether interacting components are available or not. Based on the extensions, the mode of each $c_{dyn} \in \mathcal{C}_{dyn}$ results from the combination of its *component mode* $m_{c_{dyn}}^{\odot}(t)$ and its *interface mode* $m_{c_{dyn}}^{\leftrightarrow}(t)$, which can be used to specify mode-dependent timing behavior. All other components $c \in \mathcal{C} \setminus \mathcal{C}_{dyn}$ could be designed based on the original definitions. For example, components that may enter or leave the cooperative SoCPS could be specified using corresponding *Available* and *Unavailable* modes. Finally, the mode mapping function μ of dynamic components would have to be adapted to reason about composition and refinement, i.e., by defining $\mu_{c_{dyn}}$ for each $c_{dyn} \in \mathcal{C}_{dyn}$.

In order to preserve scalability, the set of possible combinations of identifiers of other systems (or components) that could potentially be interacted with should not be explicitly built. Instead, it would be beneficial to maintain an *interaction history* for each c_{dyn} , similar to a mode history (see Equation 4.2), in which the identifiers of all available systems are stored for each time point $t \in \mathbb{T}$. The history is updated as follows:

1. Upon receiving the first event from an interacting CS that has not been available before, the system is added to the list of active systems, which is stored as a new entry in the history and triggers a corresponding *change()* event (*Addition*).
2. Existing elements are removed from the list if there is no response to a previously sent event within a specified time period, i.e., a timeout occurs. Again, the update is stored as a new history entry and triggers a *change()* event (*Removal*).

As an alternative, we could rely on the exchange of explicit *subscribe()* and *unsubscribe()* events, similar to the subscription and unsubscription requests sent by CHAVs that enter and leave the intersection area in the CLTA example (see Section 3.3.2). The main challenge here is to ensure that consistent interface modes are maintained, even if the events are not transmitted before one of the communication partners goes out of range. In both cases, the *change()* events triggered by interface reconfigurations can be used to initiate component-internal behavior, which may include behavioral reconfigurations.

The extensions discussed represent a first step toward comprehensive support for modeling dynamic reconfiguration. However, some of the concepts are based on restrictive assumptions that entail inherent limitations and restrict their practical applicability. For example, the introduction of interface modes allows for dynamic activation and deactivation of component interfaces. Nevertheless, the set of available interfaces remains static. This means that only reconfigurations that do not involve adding or removing interfaces are supported. Another limitation is the lack of support for handling multiple instances of (sub-)system types, each of which may have different and potentially inconsistent local states. The introduction of *specification templates* in Chapter 5, which can be instantiated using instance-specific parameter sets, could contribute to addressing this issue.

5 Modeling of Unreliable Communication

As we have seen in the CLTA application example, temporal properties of the communication between CSs can have a significant impact on the timing behavior of cooperative SoCPSs. For example, a delayed transmission of the evaluation results as part of the negotiation process during the *Initiate and Evaluate* activity (see Figure 3.6) can lead to an additional delay of up to 700 ms. In this case, the restriction of the maximum delay is due to a safety mechanism that has been integrated in the application example. In other situations, however, technical limitations and transmission errors may lead to significantly longer delays, which could potentially result in unbounded communication latencies.

Since the proper functioning of V2X communication is essential for exchanging safety-relevant information between CHAVs, e.g., for negotiating cooperative driving maneuvers, latency variations and temporary or permanent failures of communication systems must be carefully considered. This typically requires a thorough analysis of the failure modes of communication systems and their impact on the behavior of individual CSs. In the context of this work, we particularly focus on questions of consistency. To this end, we have developed a design approach for cooperative SoCPSs that includes the consistent formulation of top-level timing requirements on system level, thus enabling independent development of the individual subsystems. However, since the temporal behavior of cooperative systems is significantly affected by timing effects resulting from technical limitations and transmission errors, such an approach is only valid if we can prove that the behavior is consistent even under unreliable communication. Unfortunately, there are currently no contract-based specification means that allow to capture failure modes of communication systems and to characterize their effects on timing behavior.

This chapter addresses the third methodological gap “*Insufficient support for modeling unreliable communication*” (MG-3) by developing a generic communication model with example parameter sets for relevant communication standards, i.e., IEEE 802.11p and C-V2X. The idea is to model communication systems as components of the top-level architecture of SoCPSs and annotate them with timing specifications covering relevant time phenomena. The generic communication model is formulated as a set of parameterized specifications that can be used to instantiate concrete models for describing communication systems. To this end, parameter sets are defined for both communication standards.

The corresponding activities aim at reusing existing MTSL 1.5+MC specification patterns in order to keep the resulting number of patterns at a manageable level. At the same time, this means that communication systems are designed using the same methods and tools as systems that interact based on communication. The results allow to explicitly consider communication failures and corresponding error correction techniques such as retransmissions and timeouts in functional system design. The approach thus relies on the enhanced MULTIC design paradigms and contribute to answering RQ-2.2.

In the following, we first review existing approaches for modeling and analyzing unreliable communication in Section 5.1. Subsequently, typical properties of IEEE 802.11p and C-V2X are investigated as examples of communication systems used in the automotive domain. The time phenomena relevant for the characterization of unreliable communication are then identified based on a systematic literature review in Section 5.3. After deriving the generic communication model in the form of parameterized specifications, example parameter sets for IEEE 802.11p and C-V2X are defined in Section 5.4.

5.1 Modeling and Analysis of Unreliable Communication

In previous chapters, we have already discussed the effects of unreliable communication on the desired properties of cooperative SoCPSs. In this context, we have dealt with consistency in distributed systems and came across the Byzantine generals problem as a vivid illustration of the effects of faulty communication in Section 1.2.2. Given the importance of communication for the system class under consideration, it is not surprising that dealing with uncertainties has been an important research topic in the past decades.

As early as the 1980s, the first failure models for reasoning about unreliable communication in distributed systems were developed. For example, Fischer et al. and Dolev et al. elaborated on the impact of *crash* failures on achieving agreement in asynchronous systems [187, 229]. Other authors considered *omission*, *stopping*, and *arbitrary* failures [261, 262, 476]. Garay and Perry introduced a series of alternative failure models that aim to solve key problems of “*addressing consensus and performing reliable broadcast*” [239, p. 153] in systems with failing processes [239]. More recent works continue to contribute to this line of research by considering, for example, the effects of omissive faults on *approximate agreement* problems [47]. Other groups of authors focus on additional topics, such as the modeling of cascading failure dynamics in communication networks [498].

Several communication protocols have been developed over the years to overcome unreliability. *Time-Sensitive Networking* (TSN) [227], for example, focuses on establishing deterministic end-to-end transmission of Ethernet-based communication streams using well-established synchronization protocols to support hard real-time requirements. As an extension, *Deterministic Network Channels* (DNC) [44] was defined that is intended to achieve “*reliable and deterministic communication for distributed tasks*” [44, p. 1].

In industrial practice, pragmatic strategies for preventing or mitigating communication faults are typically applied. This includes the integration of redundant communication systems for safety-critical applications as well as the implementation of heartbeats and watchdogs to detect communication failures. Other approaches are specifically tailored to the field of connected and highly automated driving. For example, Thunberg et al. studied the safety perspective of cooperative driving and proposed an algorithm to efficiently compute safety times for emergency braking based on unreliable V2X communication [573]. Moreover, Hasan et al. developed an approach for detecting transient communication outages in vehicle platooning to ensure graceful degradation of safety-critical system functions, including the automated triggering of minimum risk maneuvers [280].

5.1.1 Communication Models

As one of the most prominent models of communication, the Shannon-Weaver model [535, 601] structures the process of communication into five basic components. According to this model, messages are first generated by an *information source* before being transmitted in the form of a signal by a *transmitter*. The signal is then transferred via a *channel*, which may be subject to noise, and is received by a *receiver* that passes the message on to the *destination*. Over the years, various communication theories and models have been developed [155], primarily in the field of information theory. The ongoing development resulted in the emergence of more complex communication models, such as Schramm’s model of communication [532], which incorporates a feedback loop and thus supports more interactive forms of communication. Although these theoretical models are only suitable to a limited extent for characterizing real-world communication systems, they can serve as a starting point for structuring models of technical communication.

Today, engineers can rely on a wide range of well-understood communication paradigms for realizing data exchange in technical systems. At a low level of abstraction, the *request/response* paradigm, for example, can be applied to realize data exchange in terms of method calls with return values. Moreover, the *publish/subscribe* interaction scheme and its variants [213, 568] support more flexible forms of communication that do not require knowledge about the recipients of a message. Early work in establishing reliable communication in real-time systems aimed at developing *clock-driven* [160, 365] and *timer-driven* [29, 85] protocols for inter-process communication. In this context, Verissimo introduced a generic model of causal delivery protocols [590]. Later, more abstract communication paradigms for multi-task real-time systems, such as implicit communication, which is also known as the *read-execute-write* paradigm [264], and *Logical Execution Time* (LET) [353] have been developed. Siron et al. introduced *Synchronous LET* [541] as an extension of LET, which assumes synchronous process execution and allows the interaction between processes only at the beginning and end of their execution windows. As an alternative, Caspi et al. introduced *Dynamic Buffering Protocol* (DBP) as a semantics-preserving protocol for inter-process communication [128]. Tang et al. investigated the effects of different communication paradigms, i.e., implicit communication, LET, and DBP, on end-to-end timing constraints in cause-effect chains [567].

At a higher level of abstraction, domain-specific communication protocols have been established that are tailored to the requirements of specific application domains. For example, Bruns et al. considered modeling of communication in industrial production systems based on IEC 61499 [111]. Protzmann and Anwar et al. have focused on developing communication models for V2X communications [36, 490, 491]. In addition, simulation models can serve as valuable source of information for the development of connected driving functions. The discrete-event network simulator ns-3 [504], for example, supports simulation of V2X communication via *Dedicated Short Range Communication* (DSRC) [514], *Long-Term Evolution* (LTE), and 5G, including relevant events such as interference, routing, and congestion. Communication models for co-simulation with traffic simulations, such as those provided by *Vehicles in Network Simulation* (VEINS) [198, 545], which is based on OMNeT++ [586], allow the virtual testing and validation of V2X applications.

VEINS also includes a comprehensive set of realistic *Medium Access Control* (MAC) and *Physical Layer* (PHY) behavior models. Additionally, Artery [503], which is developed as an extension of VEINS, provides simulation support for ETSI ITS-G5 [215].

5.1.2 Timing Analysis of Communication Systems

In the context of this work, we have already investigated well-established frameworks for timing specification in Section 2.1.1, which can also be used to specify and analyze the temporal behavior of communication systems. In addition, state-of-the-art real-time and scheduling analyses considered in Section 2.1.5 typically include the analysis of low-level communication between real-time tasks. Thus, corresponding analyses provide insights into internal communication processes that are part of the behavior of individual CSs. Finally, distributed run-time monitoring, which has been discussed in Section 2.1.6, can be applied to evaluate the timing behavior of communication on SoCPS level.

Early work in the field of distributed real-time system design focused on using fault injection techniques for analyzing real-time communication, among other strategies. In this context, Rosenberg and Shin developed a *Software Fault Injector* (SFI) that allows the injection of communication faults, such as message losses, message alterations, and message delays [507]. The authors demonstrated the practical applicability of the SFI by analyzing the “*effect/s of intermittent failures on the communication between two nodes*” [507, p. 213]. Han et al. continued the corresponding research activities by improving support for additional fault types and conducted an extended evaluation [266].

Timing analysis of fieldbus systems, such as *Controller Area Network* (CAN) [174, 574, 575, 576] and FlexRay [485] buses is another relevant research area. Additionally, the temporal behavior of Ethernet-based communication has been investigated. In this context, Cardoso et al. considered network latency and packet delay variation in local networks [127]. More recent publications deal with the analysis of state-of-the-art communication standards, such as the already mentioned TSN. Bruns et al. focused on timing effects in an IEC 61499 Ethernet/TSN communication scenario [110]. Furthermore, Houtan et al. analyzed data propagation delays in TSN-based in-vehicle communication [297].

From a broader perspective, probabilistic specifications and corresponding verification methods would be a natural fit to account for unreliable communication and handle the resulting temporal phenomena. More specifically, methods and tools for handling *weakly hard real-time systems* [74, 447], which aim at providing probabilistic guarantees on end-to-end timing behavior, are interesting candidates for addressing non-deterministic transmission delays and communication faults. Similar concepts have already been incorporated in the definition of the TADL specification language [89], which is used as part of the *EAST-ADL with TADL* framework (see Section 2.1.1). Existing analysis methods support probabilistic WCET analysis [75, 76], the computation of probabilistic scheduling guarantees [120], and the determination of worst-case deadline failure probabilities on hardware platforms with CAN buses [441]. Additionally, Köhler and Ernst extended a non-probabilistic compositional timing analysis tool to support weakly hard real-time systems [359]. As a proven tool, the probabilistic model checker PRISM [375, 581] allows the verification of real-time systems with discrete probability distributions [376]. For further

information, we refer interested readers to [172, 173], which provide a comprehensive overview of probabilistic timing analysis and schedulability analysis techniques.

Since many of the relevant properties are not stochastically independent, the definition of probabilistic communication models is a challenging task. Due to the limited scope of this work, we therefore refrain from integrating probabilistic specifications and analysis methods into the proposed methodology. In the following, we focus on covering as many relevant time phenomena as possible using existing MTSL 1.5+MC specification patterns and include `K out of N` pattern extensions (see Section A.4.4) whenever required.

5.2 Communication Standards

The remainder of this chapter focuses on the timing behavior of V2X communication. For this purpose, we consider IEEE 802.11p and C-V2X as the two most common communication standards for this type of communication. As part of our considerations, we limit ourselves to focus on the application and MAC layers, including selected physical effects with significant impact on temporal behavior at higher levels of abstractions. The following sections summarize the most important features of the two standards.

5.2.1 IEEE 802.11p

IEEE 802.11p [315] is defined as an amendment of the widely used IEEE 802.11 standard. It supports the realization of V2X applications based on mobile ad-hoc networks. The standard focuses on defining MAC and PHY layers for vehicular communication and relies on the 5.9 GHz frequency band, which is used slightly differently in the US and Europe. Its original scope was to provide support for exchanging data between high-speed vehicles and roadside infrastructure. However, as a result of its ongoing development, IEEE 802.11p has also become state-of-the-art in other application areas.

The standard is accompanied by other (sub-)standards that particularly focus on defining the application layer. In the United States, the *Wireless Access in Vehicular Environments* (WAVE) standard specified by IEEE 1609.x [314] and DSRC defined in SAE J2735 [514] are particularly relevant. In Europe, the ETSI ITS-G5 [215] standard, which supports the use of multiple communication channels and simultaneous communication on different channels, has become the established solution. IEEE 802.11p supports different kind of routing schemes, i.e., broadcast, unicast, and multicast. The underlying V2X messages can generally be divided into two groups based on the type of communication:

Event-Based Communication The first group includes messages that are only sent in case of certain events, e.g., when identifying hazards or performing emergency braking maneuvers. The messages are distributed using multi-hop geographical broadcasts, i.e., the information is distributed in a specific geographical area.

Awareness-Based Communication Messages of the second group are periodically sent and distributed based on single-hop broadcasts, meaning that the information is

distributed in a communication range around the sender. Consequently, the messages can be received by all participants that are close enough to the transmitter.

The basic principle of enabling vehicular communication via ad-hoc networks results in several challenges that have been examined in previous studies. For example, Bilgin and Gungor have reviewed known problems and challenges in IEEE 802.11p-based V2X communication. The authors identified routing challenges, the doppler effect, the *hidden terminal problem*, which “causes collisions when two nodes that are not in same communication range try to transmit data to the same node” [82, p. 3], questions about data security, latency variations, and the presence of unbounded end-to-end latencies as the main challenges in designing corresponding communication systems [82].

The performance of IEEE 802.11p has been thoroughly investigated. Analytical, simulation-based, and experimental techniques were applied to analyze end-to-end timing behavior, throughput, and the effects of interference [82, 207, 225, 493, 522, 544, 582, 600, 611]. Bilstrup et al. investigated the ability of different MAC techniques to support real-time applications in V2X communication [83]. Other groups of authors focused on the modeling of parts of the underlying communication protocols [125, 198, 519]. Moreover, Huang and Lai examined the scalability of vehicular ad-hoc networks [303]. Particular focus was also placed on analyzing the performance of the *Enhanced Distributed Channel Access* (EDCA) mechanism [564, 598, 623], which is an essential component of the MAC protocol [492, 595, 612]. Other works address clock synchronization in IEEE 802.11p multi-hop networks [143], or focus on the evaluation of potential extensions [86, 509].

5.2.2 Cellular V2X

Cellular V2X (C-V2X) is a generic term for standards and technologies that enable V2X communication via 4G/5G networks. It has been developed as a series of specifications for wireless vehicle connectivity via mobile communications and is actively developed as part of the global *3rd Generation Partnership Project* (3GPP) initiative. Similar to IEEE 802.11p, related standards primarily focus on defining MAC and PHY behavior.

Over the years, several releases of the standards have been developed, each using different basic technologies for implementing vehicular communication. The releases that are relevant to this work address the use of LTE for V2X communication (Release 14), the introduction of 5G for initial use cases and the preparation of the migration to 5G (Release 15), as well as full V2X support based on 5G standards (Release 16+). In the following, we focus on LTE-V2X and 5G-V2X as the most common technologies for C-V2X applications. Both standards support the same communication modes:

Device-to-Network The first mode addresses vehicle-to-network (V2N) communication via cellular links, e.g., to access cloud services or enable data exchange between vehicles that are too far apart for direct communication. In this mode, the communication is realized via the so-called Uu interface. V2N communication is also referred to as *Uplink* (UL), and *Downlink* (DL) in case of network-to-vehicle communication.

Device-to-Device Direct vehicle-to-vehicle (V2V), vehicle-to-infrastructure (V2I), and vehicle-to-pedestrian (V2P) communication with or without network scheduling are commonly referred to as device-to-device communication. This type of communication is realized via the PC5 interface and is denoted as *Sidelink* (SL) communication. The existing SL submodes can be divided into network-assisted (LTE-V2X, mode 3; 5G-V2X, mode 1) and sensing-based, “autonomous” (LTE-V2X, mode 4; 5G-V2X, mode 2) communication modes, depending on the release version of the C-V2X standard and the underlying resource allocation mechanism [249].

C-V2X has a number of known issues that may present challenges when developing V2X communication systems. These challenges include a limited number of available channels, limited data rates due to shadowing effects and other physical effects such as the Doppler effect, as well as reduced propagation speeds due to collisions that may result from retransmissions. Gyawali et al. provide an overview of the technical challenges in the context of LTE-V2X and 5G-V2X [259]. Due to the slot-based nature of mobile communications, C-V2X transmission latencies typically involve multiples of the *Transmission Time Interval* (TTI), which is defined as the “*minimum transmission time unit in cellular systems. Each data transmission, i.e., control signaling, scheduling configuration, retransmission process, and so on, consumes at least one TTI*” [389, p. 15801].

Similar to IEEE 802.11p, numerous publications exist that focus on evaluating the performance of C-V2X [253, 333, 389, 577, 610]. Additionally, Gonzalez-Martín et al. developed an analytical model of (parts of) the C-V2X communication protocol [248]. Other groups of authors, such as Morold et al., focused on specific applications and derived, for example, latency requirements for cooperative collision avoidance [433]. In recent years, controversial discussions have arisen about the technical superiority of one of IEEE 802.11p or C-V2X [2, 631]. Significant efforts have been made to compare the performance of both communication standards [41, 265, 409, 508, 520, 523, 536].

5.3 Relevant Time Phenomena

As we have found in the previous section, there is a large body of work that aims at characterizing performance and reliability of V2X communication. Related work includes analytical models of IEEE 802.11p and C-V2X, as well as detailed information about their timing behavior. The underlying data was obtained from various sources, including simulations, controlled experiments, and real-traffic measurement campaigns. For this reason, the decision was made not to collect any new data to identify the time phenomena relevant to modeling unreliable communication, but to systematically examine existing research results that focus on analyzing state-of-the-art V2X communication.

To this end, a systematic literature review (SLR) was conducted to identify and quantify relevant time phenomena. In this context, the focus was on characterizing (1) end-to-end timing behavior, (2) temporal behavior of (parts of) the underlying communication protocols, and (3) relevant failure modes that affect communication using IEEE 802.11p and C-V2X. The SLR results served as the basis for the definition of the

generic communication model in Section 5.4. Subsequently, a subset of the SLR results was used to quantify the identified time phenomena and extract typical parameter values, which are then used to define example parameter sets for IEEE 802.11p and C-V2X.

Figure 5.1 provides an overview of the SLR procedure. In a first step, eight Google Scholar queries were executed using the *Publish or Perish* tool [278] on March 10, 2023 (see *Execution of eight Google Scholar queries*). The search queries were performed using different keyword combinations, i.e., “V2X timing”, “V2X performance”, “V2X reliability”, “V2X delay”, “V2X latency”, “V2X packet loss”, “V2X experiment”, and “V2X evaluation”. Note that citation records and patents were excluded from the search results by selecting the appropriate options. After retrieving the results, the first 150 hits for each keyword combination were selected for further consideration. The obtained search results contained known good results [14, 409, 508, 523], which was interpreted as an indicator that a good data basis for identifying the time phenomena had been achieved. All results were exported as CSV files in order to facilitate subsequent importing and merging into a single Microsoft Excel file (see *Merging*). In total, 1 200 results were collected.

The Excel file served as the basis for all subsequent processing steps. Based on a quick manual review, duplicates and ineligible entries were excluded from the list of relevant results. All results that fulfill at least one of the exclusion criteria, i.e., non-English documents, results without available full-texts, non-peer-reviewed documents, documents retracted after publication, and duplicates due to merging, were removed (see *Exclusion of duplicates and ineligible results*). In total, 551 results were excluded, 649 remained.

Following the exclusion of ineligible results, all documents were automatically screened to exclude results that did not contribute to the topic. To this end, all abstracts were manually extracted from the publishers’ websites and transferred to the Excel file. The titles and abstracts were then analyzed in terms of three consecutive search queries:

No focus on IEEE 802.11p or C-V2X All results *without* hits for “802.11”, “DSRC”, “ITS-G5”, “C-V2X”, “cellular”, “LTE”, or “5G” were excluded. The relevant keywords for this search were selected based on the findings from Section 5.2.

Focus on security, privacy, or authentication methods All results *with* hits for “secur”, “privacy”, or “authentication” were excluded after a manual review. Note that this criterion was introduced retroactively, since most of the results from this field of research did not provide relevant information about temporal behavior.

No relation to real-time, timing, safety, or reliability All results *without* hits for “real-time”, “timing”, “delay”, “latenc”, “packet loss”, “packet delivery”, “packet reception”, “reliab”, or “safe” were excluded. The keywords for this search query were selected based on a review of commonly used keywords of known good results.

All entries that fulfilled at least one of the three exclusion criteria were excluded from further consideration (see *Automatic screening*). Note that the keywords have been deliberately chosen to cover both nouns and verbs. In total, the automatic screening process resulted in the exclusion of 345 results, leaving 304 relevant entries.

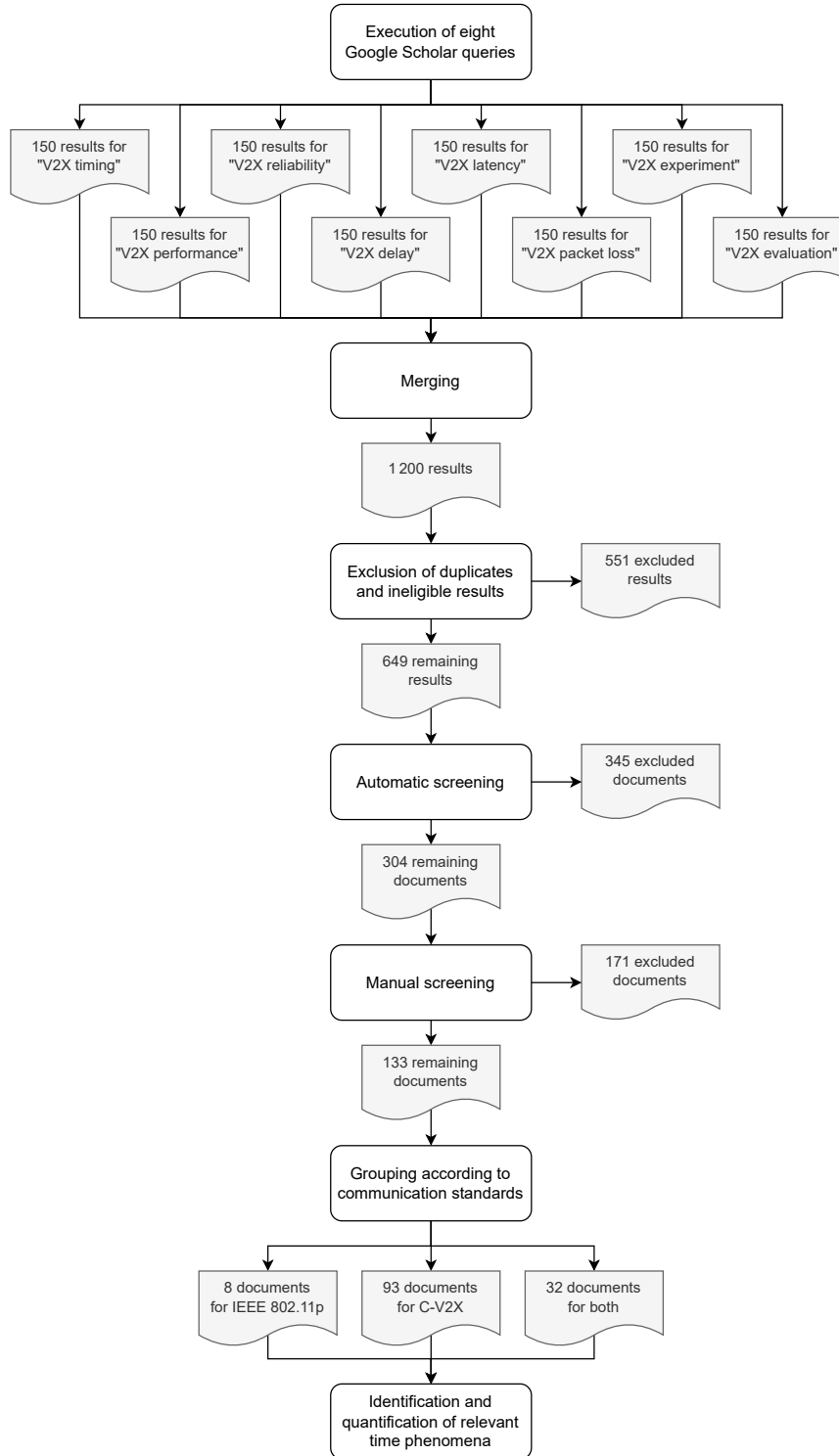


Fig. 5.1: Procedure for systematic literature review

5 Modeling of Unreliable Communication

Afterwards, a manual screening was conducted to reach a manageable number of results and exclude documents that were overlooked by the automatic screening (see *Manual screening*). The process involved manually cross-reading all remaining documents, which led to the exclusion of results that met one or more of the following conditions:

1. No system-level evaluation of IEEE 802.11p or C-V2X performance
2. No indication of parameter values (related to timing, safety, or reliability)
3. Exclusive consideration of physical data transmission

Although the manual screening was time-consuming, it was also very effective. A total of 171 unrelated documents could be identified. The remaining 133 entries were included in the final data set that served as the basis for the identification of relevant time phenomena. As a final preparation, all results were grouped according to the considered communication standards (see *Grouping according to communication standards*). The grouping divided the data set into three groups: 8 results for IEEE 802.11p (Group A), 93 results for C-V2X (Group B), and 32 results for IEEE 802.11p and C-V2X (Group C).

The dataset derived using the SLR is presented along Figures 5.2 to 5.5. Figure 5.2 shows the number of publications included per group per year. The data set contains results with publication dates between 2011 and 2023. The total of eight publications related to IEEE 802.11p (Group A) are distributed relatively evenly over the entire period, with a maximum in 2020. No publications related to C-V2X (Group B) or IEEE 802.11p and C-V2X (Group C) are included before 2016. From 2016, the number of publications increases significantly and reaches its peak in 2021. The low number of publications in 2023 is due to the timing of the search queries, which were conducted in March 2023.

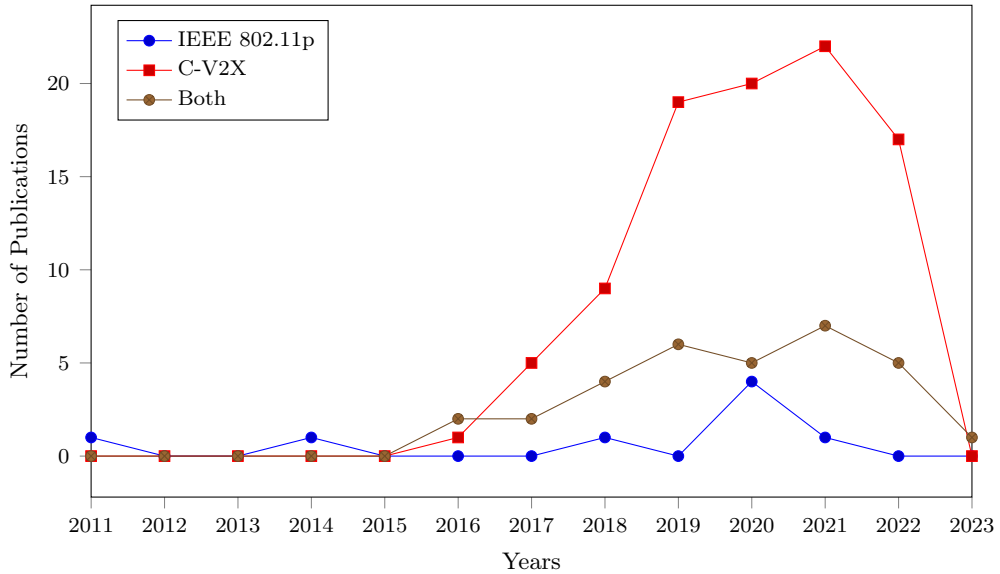


Fig. 5.2: Number of included publications per group per year

The number of included publications per communication standard is presented in Figure 5.3. This time, the results are not organized according of the three groups, but on the basis of a more detailed breakdown into their region- or release-specific (sub-)standards. Out of the 133 results, 31 relate to DSRC, and 11 to ETSI ITS-G5, respectively. The results relating to C-V2X are divided into 102 results for LTE-V2X and 37 for 5G-V2X. Note that many of the included publications consider more than one (sub-)standard (e.g., DSRC and ETSI ITS-G5), so that the numbers do not add up to the total number of 133. The works that focus on investigating C-V2X consider direct V2V communication via SL in 60 results, network-based V2N communication via UL/DL in 11 results, and both in 13 results. The other nine publications did not explicitly state the communication mode used, meaning that they could not be included in corresponding analyses.

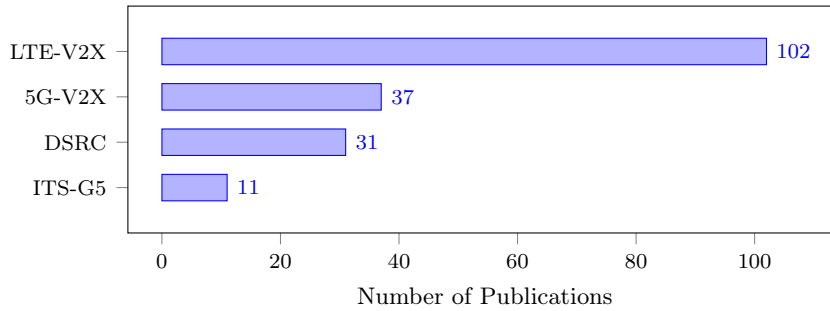


Fig. 5.3: Number of included publications per communication standard

Figure 5.4 shows the number of included publications per performance metric. The considered metrics range from ratios of successful to failed transmissions, end-to-end transmission latencies, throughput, probabilities of packet reception and collisions, error rates, and latencies of individual processing steps. With 72 out of 133 results, the *Packet Delivery Ratio* (PDR) [265, 495, 508, 599] is the most frequently used performance metric for V2X communication. However, with a total of 51 results, the *Transmission Latency* [265, 389, 495, 599] is also very common. Other metrics that were defined to allow for theoretical considerations or to assess extensions to the original communication standards were used less frequently, such as the *Latency Satisfaction Ratio* [181] or the *Delay Outage Probability* [253]. The metrics of the *Other Performance Metrics* group were in almost all cases related to physical effects and are therefore not considered relevant here. The other few metrics in this group were used to evaluate aspects that are not timing-related. For example, Moradi-Pari et al. investigated the effective communication range that can be achieved under line-of-sight or non-line-of-sight conditions [431]. Since most of the affected results also included other metrics, e.g., by considering the PDR in the case of [431], only two publications had to be excluded from further consideration.

All included publications have in common that they investigate the performance or reliability of V2X communication with respect to a set of performance metrics. However, the studies serve different purposes. While most of the results address evaluating the original communication standards (73 results), others focus on evaluating previously made extensions using standard behavior as a baseline (53 results). Further work concentrates on

5 Modeling of Unreliable Communication

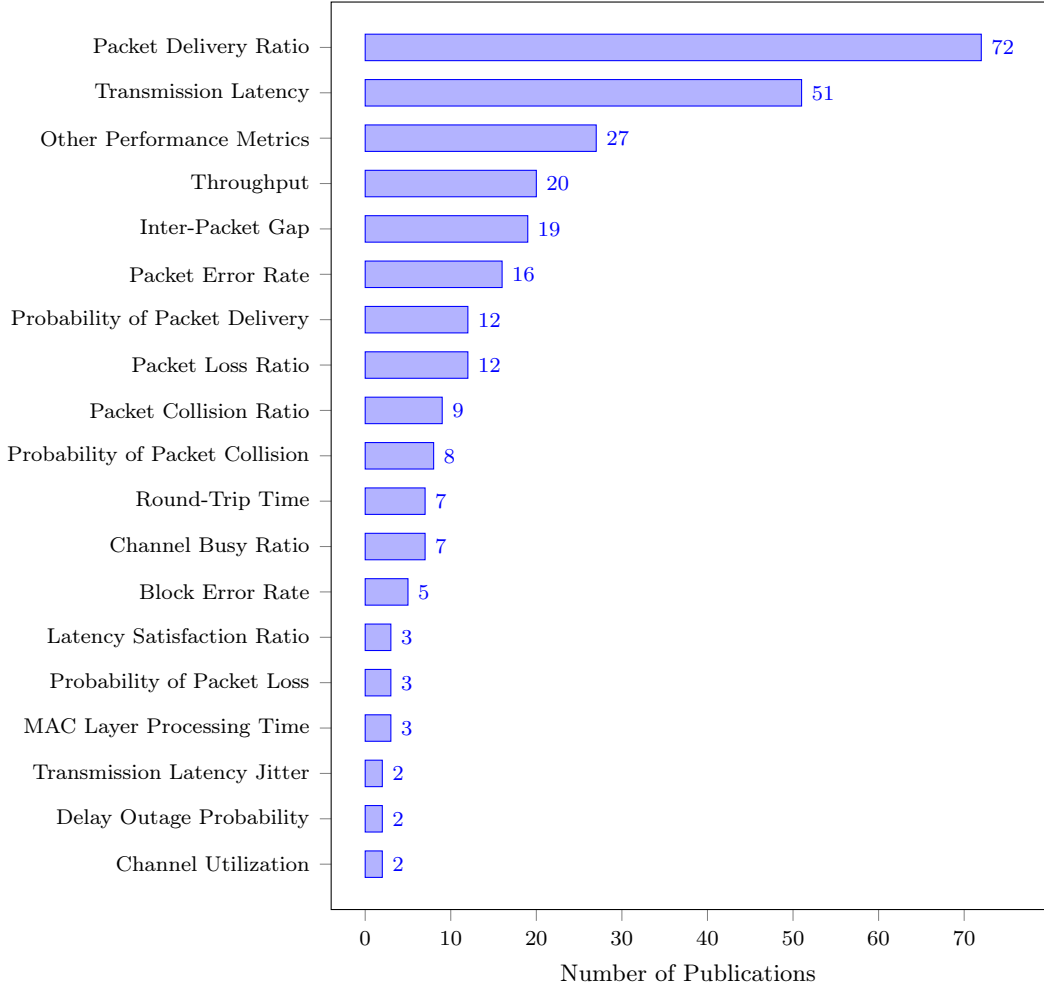


Fig. 5.4: Number of included publications per performance metric

evaluating test and simulation frameworks (5 results), specific V2X applications (1 result), or alternative communication techniques, such as visible light communication (1 result). In all cases, only those parts of the results that could be clearly assigned to the behavior of the standards considered in Section 5.2 were used. In this context, we were able to derive relevant information about the underlying time phenomena from results that were not directly related to the evaluation of the IEEE 802.11p or C-V2X standards.

Regarding the SLR results, the characterization of timing behavior and reliability relies on four types of data sources. Figure 5.5 shows the number of included publications per data source. With a total of 103 results, *simulations* were most frequently used for the evaluation. However, *analytical models* (23 results), *field observations* in real traffic (18 results), and *controlled experiments* (12 results), such as measurement campaigns in controlled environments and test drives on non-public test tracks, were also included. Except for the studies that focused exclusively on analytical models or abstract network

simulations (24 results), the SLR results take into account a wide range of scenarios. The data set includes evaluation results for *urban* (54 results), *highway* (52 results), *freeway* (11 results), and *rural* (4 results) scenarios. A small number of publications (11 results) considered alternative scenarios (e.g., parking lots) that could not be clearly assigned to one of the previous classes. Note that multiple data sources were used in some of the results, so that the numbers do not add up to the total number of 133 entries.

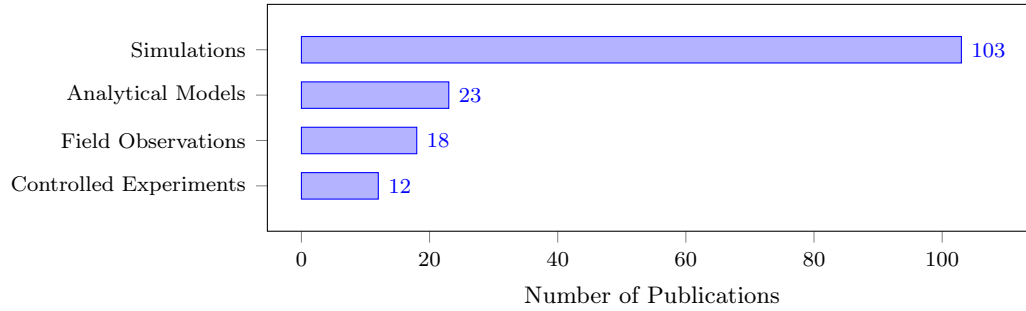


Fig. 5.5: Number of included publications per data source type

In order to approach the definition of the generic communication model, we divide the process of transmitting V2X messages into three (abstract) *communication phases*. The first phase, referred to as *Send* phase, includes all processing steps conducted by the *Sender*, i.e., the preparation of data, the request of channel access as well as the encoding and sending of the message. After the message has been sent, it is transmitted via a shared medium, where it may be affected by interference. The transmission is addressed by the *Transmit* phase. The reception of the message, the application of error correction techniques, and the extraction of relevant data by the *Receiver* are handled in the *Receive* phase. The three phases provide an intuitive pattern for modeling communication, which is in line with well-established and commonly used communication models [532, 535].

In the following, we use the communication phases to structure the identification of relevant time phenomena. As part of the manual analysis of the SLR results, all performance metrics were collected and assigned to one or more of the three phases. As a result, the underlying evaluation results were also mapped to the phases. The resulting mapping of performance metrics to communication phases is shown in Table 5.1. The table lists all metrics that have been considered in Figure 5.4, but excludes the metrics from the *Other Performance Metrics* group. This time, the performance metrics are not sorted according to their frequency, but based on their relevance to the considered phases. More general metrics that address all communication phases (*Transmission Latency* to *Inter-Packet Gap*) are listed first. Afterwards, all metrics that could be explicitly assigned to the *Send*, *Transmit*, and/or *Receive* phase are presented one after the other.

The mapping of the performance metrics to the communication phases is indicated in the *Phases* columns. All entries are marked either as having a *direct influence* (●), an *indirect influence* (○), or as *not be applicable* (–). All communication phases considered are covered by at least one metric. In addition, we see a strong focus on the *Transmit* phase, since many of the metrics refer to the characteristics of message transmission.

Tab. 5.1: Mapping of performance metrics to communication phases

Metric	Phases			Standards ¹			Sources ¹			References ²
	Send	Transmit	Receive	IEEE 802.11p	C-V2X (SL)	C-V2X (UL/DL)	Analytical Models	Simulations	Physical Tests	
Transmission Latency	●	●	●	⊗	⊗	⊗	12	28	20	[265, 389, 495, 599]
Transmission Latency Jitter	●	●	●	⊗	⊗	⊗	–	–	2	[354, 610]
Round-Trip Time	●	●	●	⊗	–	⊗	2	1	5	[354, 409]
Throughput	●	●	●	⊗	⊗	⊗	2	15	5	[265, 354, 409]
Inter-Packet Gap	●	●	●	⊗	⊗	⊗	1	16	2	[36, 333, 536]
MAC Layer Processing Time	●	–	–	⊗	⊗	⊗	–	3	–	[410, 577]
Channel Busy Ratio	○	●	–	⊗	⊗	⊗	1	6	1	[54, 606]
Channel Utilization	○	●	–	⊗	⊗	⊗	1	1	–	[606]
Packet Delivery Ratio	–	●	–	⊗	⊗	⊗	5	65	7	[265, 495, 508, 599]
Probability of Packet Delivery	–	●	–	⊗	⊗	⊗	3	11	1	[181, 495]
Packet Collision Ratio	–	●	–	–	⊗	–	3	9	–	[97, 333]
Probability of Packet Collision	–	●	–	⊗	⊗	⊗	4	7	–	[97, 253, 577]
Packet Loss Ratio	–	●	–	⊗	⊗	⊗	1	4	8	[354, 556]
Probability of Packet Loss	–	●	–	⊗	⊗	⊗	3	3	–	[248]
Latency Satisfaction Ratio ³	–	●	–	⊗	⊗	–	–	3	–	[181]
Delay Outage Probability	–	●	–	–	⊗	⊗	2	2	–	[253]
Packet Error Rate	–	●	○	⊗	⊗	⊗	–	10	8	[409, 536, 610]
Block Error Rate	–	●	○	⊗	⊗	⊗	–	4	1	[298, 422, 509, 523]

● = direct influence ○ = indirect influence ⊗ = covered by results – = not applicable

¹ Multiple entries per result were permitted. ² Example references were selected according to their relevance to this work. ³ Metrics were used exclusively to evaluate extensions.

The *Standards* columns indicate which communication standards from Section 5.2 are covered by the SLR results (⊗). Thus, the columns from this group express whether one or more results from the data set rely on the metric to evaluate temporal behavior and reliability of the given standard. Again, the results show good coverage, indicating that the SLR forms a valid basis for defining example parameter sets in Section 5.4.2.

In order to get an impression of the diversity and reliability of the underlying data, the *Sources* columns list how often certain data sources were used to evaluate the individual metrics. In this context, evaluation data based on simulations is available for almost all performance metrics. During the process, the data source types *Controlled Experiments* and *Field Observations* (see Figure 5.5) were combined into a new *Physical Tests* group. This was possible because the entries from both types showed similar results.

The last column provides a number of example references for each performance metric. Note that the references were selected according to their relevance to this work. When making the selection, we took particular care to ensure that all relevant standards and data sources were covered. Recall that the total number of included publications per

performance metric is provided in Figure 5.4. Finally, it should be noted that most of the performance metrics have multiple synonyms in the literature, e.g., *Communication Delay* and *End-to-End Latency* for *Transmission Latency*. For clarity, all synonyms found in the data set were collected and summarized using the most frequent term.

In the following, we use the performance metrics as “reference points” to identify and quantify relevant time phenomena in unreliable V2X communication. Since most of the metrics are uniformly defined, the results from different groups of authors, data sources, and scenarios can be related and compared with each other. Furthermore, this approach allows us to link results obtained for different communication standards with respect to the same metric. This enables us to draw more general conclusions about how unreliability affects V2X timing behavior. During the manual analysis of the SLR results, four basic time phenomena relevant to functional system design were identified:

1. Sending frequencies of V2X messages
2. Latencies in different scales
 - a) End-to-end transmission latency
 - b) Latencies of individual processing steps
 - c) Latencies of error correction mechanisms and timeouts
3. Data age of received information
4. Failure modes with effects on timing behavior

In addition to the phenomena, we identified relevant processing steps in transmitting V2X messages. Figure 5.6 provides an overview of processing steps with effects on temporal behavior. All steps were allocated to the *Send* and *Receive* phases. Moreover, examples of failure modes that should be considered in the *Transmit* phase, such as transient communication failures and high channel load, were included.

Send			Transmit	Receive		
Application Layer	Medium Access Control	Physical Layer	Wireless Transmission	Physical Layer	Medium Access Control	Application Layer
Data preparation	Contention	Message encoding	Transient communication failures	Message decoding	Error detection on object level	Data extraction
...	Random retries	Handling of bitsets/frames	High channel load	Error correction on bit level	Initiation of retransmissions	...
	Retransmissions	...	Shadowing effects	...	...	
	Relaying	...	Interferences			
	...		...			

Fig. 5.6: Processing steps with effects on temporal behavior

Due to the limited scope of this work, we were not able to consider all identified failure modes in the development of the intended communication model. Instead, we have deliberately chosen more abstract characterizations of unreliability that cover relevant sets of concrete failure modes. The following section defines a basic procedure for developing communication models, which can also serve as a basis for future extensions.

5.4 Generic Communication Model

The identification of relevant time phenomena and processing steps with effects on timing behavior provides the basis for developing the intended communication model. During development, we adhere to the basic principle of previous system descriptions and define the communication model based on two parts. As a foundation, an (abstract) component model defines the hierarchical structure and the interfaces of all participating components. Then, the actual communication model is formulated as a set of parameterized specifications that are annotated to the components. These specifications can be provided with predefined parameter sets to instantiate concrete communication models.

In a first step, the underlying component model is defined. The basic idea is to design the communication model to contain one *component type* for each communication phase, i.e., one component type for the *Send* phase, one for the *Transmit* phase, and one for the *Receive* phase. By instantiating the component types, concrete communication models can be created and adapted to application-specific requirements. At the same time, the explicit allocation of functions from different communication phases to individual components allows a transitive mapping of the performance metrics from Table 5.1 as well as the identified time phenomena and the relevant processing steps from Figure 5.6.

The resulting component model is depicted in Figure 5.7. At the highest hierarchy level, the model includes a component for each CS that participates in the communication. In addition, a central *Transmit* component acts as the communication channel and connects all involved CS components. Each CS is decomposed into a set of functional components and a communication interface that is represented by individual *Communication* components. Each communication interface consists of two lower-level components, namely *Send* and *Receive*, which perform the functionality defined by the corresponding communication phases. Each *Send* component has a *MessageIn* port that is used to receive messages from other components, and a *MessageOut* port for interaction with the *Transmit* component. The *Receive* components are structured in exactly the opposite way. They have a *MessageIn* port to connect to the *Transmit* component and a *MessageOut* port to forward received messages to other components that are part of the receiving CS. Both types of components have a *Mode* port that is used to store their internal state.

The *Transmit* component has a pair of input and output ports for each CS, i.e., *In1* and *Out1* for *Constituent System 1*, *In2* and *Out2* for *Constituent System 2*, and so on. Additionally, it comprises a *Mode* port that stores the internal state and two special input ports, denoted as *ChannelLoad* and *TransientFailure*, that can be used to trigger transitions into previously defined failure modes. Note that designating the communicating systems as CSs arises from the context of this work, but does not contradict the use of the generic communication model in other application contexts or domains.

5.4.1 Specification Templates

In order to model unreliable communication, we aim at formulating MTSL 1.5+MC specifications for each component of the communication model. However, since we are dealing with component types rather than concrete components, and would like to support

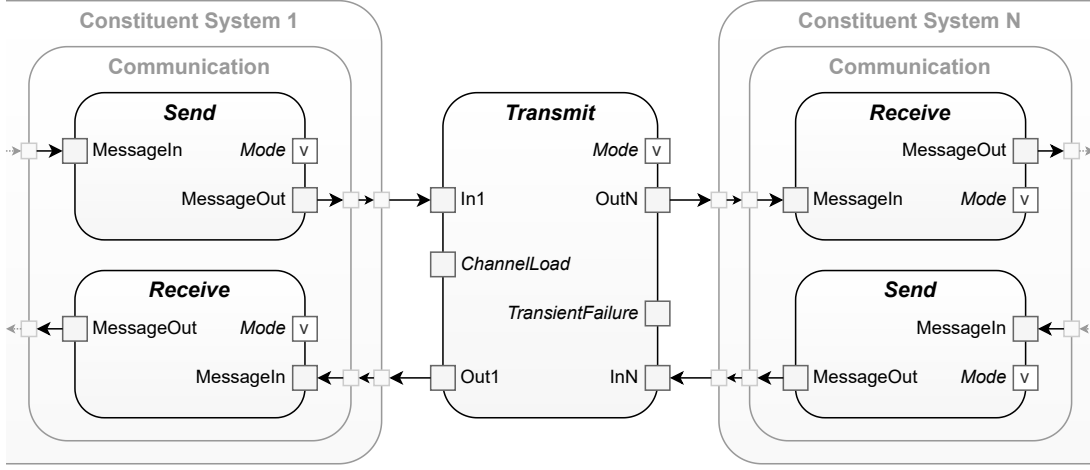


Fig. 5.7: Component model of the generic communication model

different communication standards based on the same model, the previously used form of specifications is not sufficient. Instead, we formulate the specifications in terms of *specification templates*. These templates represent abstract timing specifications that can be instantiated by assigning concrete parameter values to each of their variables.

As a prerequisite, we map all time phenomena and processing steps to MTSL 1.5+MC specification patterns, whose semantics suitably represent the underlying phenomena. In this context, **Repetition** patterns can be used to map different or varying sending frequencies of V2X messages. Latencies of different scales can be taken into account using **Reaction** patterns. The main purpose of **Age** patterns is to specify the (maximum) data age of received information. Mode-dependent specifications with optional **K out of N** pattern extensions allow the modeling of transient communication failures and support the specification of sporadic packet losses in certain failure modes. Moreover, corresponding specifications can also be used to model relevant details of communication protocols on high abstraction levels, such as latencies of individual processing steps or timeouts, by explicitly defining the validity duration of the corresponding modes. For example, this can be achieved by defining a **Reaction** pattern that constrains the time period between the occurrence of a *change*(Mode, A) event that occurs in the A mode and a subsequent *set*(Mode, B) event, which represents either the successful transition to the next processing step, or the triggering of a timeout. By applying the proposed mapping to the results from the previous section, suitable specification templates were defined for all three component types, i.e., *Send*, *Transmit*, and *Receive* components.

The specification template $\mathbb{C}_{send}$ of the *Send* components is presented in Table 5.2. Each of the individual components has four modes, namely *Idle* (I), *Process* (P), *Request* (R), and *Send* (S), which are typically executed one after the other. The component instances expect their neighboring components to provide *MessageIn* events every P_{msg} time units (line 1). After a *MessageIn* event has occurred in the *Idle* mode, the individual components switch to the *Process* mode with a minimum delay, i.e., within δ_{min} (line 3).

Tab. 5.2: Specification template $\mathbb{C}_{send}$ for sending messages

A	<i>MessageIn</i> occurs every P_{msg} with offset.	1
G	// Take incoming messages and switch to the “Process” mode within δ_{min}	2
	Reaction (<i>MessageIn</i> , set(<i>Mode</i> , <i>P</i>)) within δ_{min} in mode <i>I</i> .	3
	// Preprocess incoming messages within $\delta_{pre} - \delta_{min}$	4
	Reaction (change(<i>Mode</i> , <i>P</i>), set(<i>Mode</i> , <i>R</i>)) within $\delta_{pre} - \delta_{min}$ in mode <i>P</i> .	5
	// Request channel access within δ_{req}	6
	Reaction (change(<i>Mode</i> , <i>R</i>), set(<i>Mode</i> , <i>S</i>)) within δ_{req} in mode <i>R</i> .	7
	// Switch to the “Idle” mode if the channel is not accessible within $\delta_{req}^\times$	8
	Reaction (change(<i>Mode</i> , <i>R</i>), set(<i>Mode</i> , <i>I</i>)) within $\delta_{req}^\times$ in mode <i>R</i> .	9
	// Send the messages and switch to the “Idle” mode within δ_{send}	10
	Reaction (change(<i>Mode</i> , <i>S</i>), { <i>MessageOut</i> , set(<i>Mode</i> , <i>I</i>)}) within δ_{send} in mode <i>S</i> .	11

The latency of the preprocessing step is specified by means of the time difference between the *change*(*Mode*, *P*) and *set*(*Mode*, *R*) events, which is restricted to $\delta_{pre} - \delta_{min}$ to account for the additional delay of δ_{min} from the previous step (line 5). Subsequently, the channel access is requested within δ_{req} time units in mode *Request* (line 7). The request is terminated by a timeout if the channel cannot be accessed within $\delta_{req}^\times$ (line 9). If the channel access has been granted in time, i.e., in less than or equal to $\delta_{req}^\times$ time units, the sending of the messages is completed within δ_{send} (line 11). Once the *MessageOut* event occurs, the *Send* component switches back to the *Idle* mode, and waits for the next incoming message. Note that $\delta_{req} < \delta_{req}^\times$ must apply to ensure that the specification generates meaningful behavior. The *MessageIn* and *MessageOut* events could also be defined as virtual events triggered by event specifications (see Section A.4.2), which can be used to specify sets, selections, sequences, or coregions of relevant messages.

The intended timing behavior of the *Transmit* phase is specified by means of the specification template $\mathbb{C}_{transmit}$, which is listed in Table 5.2. The associated component has four modes, i.e., *Idle* (I), *Transmit* (T), *Load* (L), and *Failure* (F). The specification template includes an assumption for each input port (lines 1–5). In this context, the inter-arrival times of messages on input ports $In1, \dots, InN$ are specified by defining corresponding period intervals $P_{msg}^1, \dots, P_{msg}^n$ (lines 1–3). Additionally, *ChannelLoad* events are expected to occur every P_{load} (line 4), and *TransientFailure* events every P_{fail} (line 5), respectively. The guarantee specifies the transmission behavior in terms of different modes that represent either normal or faulty behavior. After an arbitrary set of input events has occurred in the *Idle* mode, the component switches to the *Transmit* mode with a minimum delay of δ_{min} (line 7). The incoming messages are then transmitted to all output ports within $\delta_{trans} - \delta_{min}$ time units (line 9). Thus, the specification template reflects broadcast communication by default. Other routing schemes, such as unicast or multicast, can be implemented by “ignoring” some of the input events, or by filtering events based on a set of previously defined event values. These values could include identifiers of other CSs that should or should not receive the annotated messages.

Once the transmission time window has closed, the component switches back to *Idle* mode (line 11). Note that an event coregion with a final *set*(*Mode*, *I*) would be required to

Tab. 5.3: Specification template $\mathbb{C}_{transmit}$ for transmitting messages

A	<i>In1</i> occurs every P_{msg}^1 with offset.	1
	$\vdots$	2
	<i>InN</i> occurs every P_{msg}^n with offset.	3
	<i>ChannelLoad</i> occurs every P_{load} with offset.	4
	<i>TransientFailure</i> occurs every P_{fail} with offset.	5
G	// Switch to the “Transmit” mode within δ_{min} after receiving one or more “In” events	6
	Reaction ($\{\{In1, \dots, InN\}\}, \text{set}(\text{Mode}, T)$) within δ_{min} in mode <i>I</i> .	7
	// Transmit the messages within $\delta_{trans} - \delta_{min}$	8
	Reaction ($\text{change}(\text{Mode}, T), [Out1, \dots, OutN]$) within $\delta_{trans} - \delta_{min}$ in mode <i>T</i> .	9
	// Switch back to the “Idle” mode within $\max(\delta_{trans} - \delta_{min})$	10
	Reaction ($\text{change}(\text{Mode}, T), \text{set}(\text{Mode}, I)$) within $\max(\delta_{trans} - \delta_{min})$ in mode <i>T</i> .	11
	// Switch to the “Load” mode within δ_{min} after receiving a “ChannelLoad” event	12
	Reaction (<i>ChannelLoad</i> , $\text{set}(\text{Mode}, L)$) within δ_{min} in mode <i>I</i> .	13
	// Transmit the messages in K_{load} out of N_{load} times within δ_{load}	14
	Reaction ($\{\{In1, \dots, InN\}\}, [Out1, \dots, OutN]$) within δ_{load} K_{load} out of N_{load} in mode <i>L</i> .	15
	// Switch back to the “Idle” mode within $\delta_{relief} - \delta_{min}$	16
	Reaction ($\text{change}(\text{Mode}, L), \text{set}(\text{Mode}, I)$) within $\delta_{relief} - \delta_{min}$ in mode <i>L</i> .	17
	// Switch to the “Failure” mode within δ_{min} after receiving a “TransientFailure” event	18
	Reaction (<i>TransientFailure</i> , $\text{set}(\text{Mode}, F)$) within δ_{min} in mode $\{I, L\}$.	19
	// Transmit no messages; switch back to the “Idle” or “Load” mode within $\delta_{recov} - \delta_{min}$	20
	Reaction ($\text{change}(\text{Mode}, F), \text{set}(\text{Mode}, \{I, L\})$) within $\delta_{recov} - \delta_{min}$ in mode <i>F</i> .	21

correctly specify the desired behavior. This would allow to specify that the time window closes at the exact time of the last *Out* event. However, such kind of event specifications are currently not supported by MTSL 1.5+MC. The underlying issue could be resolved by extending event specifications to support hierarchically nested event specifications. Since this extension would increase the complexity of the specifications excessively, we decided to leave the definitions as they are and indicate relevant limitations where necessary.

Besides the normal transmission behavior, two special modes were included to explicitly address unreliable communication. The occurrence of a *ChannelLoad* event triggers a transition to the *Load* mode (line 13). In this mode, all incoming messages are transmitted within δ_{load} and with respect to a certain PDR, i.e., only in K_{load} out of N_{load} cases (line 15). After the relief latency $\delta_{relief} - \delta_{min}$ has elapsed, the component returns to its *Idle* mode (line 17). The idea behind this style of specification is that high-load situations do not last permanently in real-world traffic scenarios, but resolve after some time due to continuous movement of the participating CHAVs. A similar design pattern was applied for modeling transient failures. After a *TransientFailure* event occurs, the component switches from *Idle* and *Load* modes to the *Failure* mode (line 19), in which no message transmission takes place. Once the recovery latency $\delta_{recov} - \delta_{min}$ has expired, the component switches back to either the *Idle* or the *Load* mode (line 21).

Finally, the specification template $\mathbb{C}_{receive}$ of the *Receive* components is given in Table 5.4. Each of the components has *Idle* (I), *Receive* (R), *Correct* (C), and *Process* (P) modes. The specification template is structured similarly to the template for the *Send* components. All instances of the *Receive* components expect *MessageIn* events to occur

Tab. 5.4: Specification template $\mathbb{C}_{receive}$ for receiving messages

A	<i>MessageIn</i> occurs every P_{msg} with offset.	1
G	// Switch to the “Receive” mode within δ_{min} after receiving a “MessageIn” event	2
	Reaction (<i>MessageIn</i> ,set(<i>Mode</i> , <i>R</i>)) within δ_{min} in mode <i>I</i> .	3
	// Receive incoming messages within δ_{rec}	4
	Reaction (change(<i>Mode</i> , <i>R</i>),set(<i>Mode</i> , <i>C</i>)) within δ_{rec} in mode <i>R</i> .	5
	// Switch to the “Idle” mode if the message could not be received within $\delta_{rec}^{\times}$	6
	Reaction (change(<i>Mode</i> , <i>R</i>),set(<i>Mode</i> , <i>I</i>)) within $\delta_{rec}^{\times}$ in mode <i>R</i> .	7
	// Correct errors in the received message within δ_{cor}	8
	Reaction (change(<i>Mode</i> , <i>C</i>),set(<i>Mode</i> , <i>P</i>)) within δ_{cor} in mode <i>C</i> .	9
	// Switch to the “Idle” mode if the errors could not be corrected within $\delta_{cor}^{\times}$	10
	Reaction (change(<i>Mode</i> , <i>C</i>),set(<i>Mode</i> , <i>I</i>)) within $\delta_{cor}^{\times}$ in mode <i>C</i> .	11
	// Pass the received messages and switch to the “Idle” mode within δ_{post}	12
	Reaction (change(<i>Mode</i> , <i>P</i>),{ <i>MessageOut</i> ,set(<i>Mode</i> , <i>I</i>)}) within δ_{post} in mode <i>P</i> .	13

every P_{msg} time units (line 1). As soon as a *MessageIn* event occurs, the components switch to their *Receive* mode with a minimum delay of δ_{min} (line 3). The incoming message is then received within δ_{rec} time units (line 5). Note that the reception process can be aborted if $\delta_{rec}^{\times}$ is exceeded (line 7). The same principle is applied for specifying error correction mechanisms. Although error correction should normally be completed within δ_{cor} (line 9), a second **Reaction** pattern triggers a timeout after $\delta_{cor}^{\times}$ that puts the component back into *Idle* mode (line 11). After the component switches to the *Process* mode, the processing of the received message is completed within δ_{post} , which also triggers a transition to the *Idle* mode (line 13). Note that concrete parameter values must comply with $\delta_{rec} < \delta_{rec}^{\times}$ and $\delta_{cor} < \delta_{cor}^{\times}$ to obtain valid specifications. Again, the *MessageIn* and *MessageOut* events can also be expressed in terms complex event specifications.

The current version of the specification templates covers all relevant time phenomena and processing steps identified in Section 5.3, but also has some limitations. For example, some of the parameters used in the specifications, such as δ_{pre} , P_{load} , δ_{relief} , P_{fail} , δ_{recov} , and δ_{post} depend on the application or scenario. Other parameters, such as the minimum time interval δ_{min} for specifying the most immediate reactions possible, can only be determined based on implementation details. Furthermore, the buffering of incoming messages and overlapping activations are currently not supported. Therefore, P_{msg} must be set appropriately in order to prevent overlapping activations. Finally, individual processing steps cannot be skipped or disabled due to the semantics of the **Reaction** pattern. Each “skipped” processing step adds an additional (minimum) delay of δ_{min} .

5.4.2 Example Parameter Sets

The generic communication model forms the basis for specifying temporal behavior of state-of-the-art V2X communication systems. It can be used to flexibly model different combinations of sending and receiving CSs of cooperative SoCPSs by instantiating *Send* and *Receive* components, and allows to trigger predefined failure modes of the shared communication channel represented by the *Transmit* component. However, to employ

the model for real-time analyses, such as performing VITs, concrete parameter sets are needed to transfer the symbolic specification templates into explicit specifications.

For this purpose, we aim at defining example parameter sets for the two considered communication standards. To this end, we focus on SLR results that evaluate both IEEE 802.11p and C-V2X communication, since they typically have recorded simulation or experimental data for both types of communication in the same setting, which ensures comparability. From the outset, it became apparent that some parameters of the specification templates are application- or scenario-specific, so that they cannot always be quantified on the basis of the SLR results. In the following, we will therefore focus on those parameters whose value ranges can be characterized using the results obtained.

The example parameter sets for IEEE 802.11p and C-V2X derived based on the SLR results are presented in Table 5.5. The table lists all parameters that are used in the specification templates, including their name, the assigned symbol, and concrete parameter values (or ranges) for three different settings. The first setting focuses on the use of IEEE 802.11p for broadcast communication (see *IEEE 802.11p*). The other two settings address unicast communication using C-V2X, covering SL (see *C-V2X (SL)*) and UL/DL (see *C-V2X (UL/DL)*) communication. All *application- or scenario-specific* parameters are marked with “*”, indicating that the values have not yet been considered. The corresponding entries can serve as a starting point for future activities, in which missing parameter values would then be derived from the SLR results or other sources. For example, the inter-arrival time of messages P_{msg} has a default value of $(0, \infty)$ ms, and can be quantified based on the message frequency of 1 to 10 Hz for *Cooperative Awareness Messages* (CAMs) [214]. In the meantime, VITs can be performed by replacing all marked parameters with δ_{min} , or by temporarily adjusting the specification templates.

Note that the derivation of the example parameter sets was intended primarily to demonstrate the practical suitability of the generic communication model. As we have only considered parts of the SLR results and did not conduct a comprehensive validation of the resulting model, we cannot make statements about the completeness or correctness of the results. Nevertheless, the example parameter sets provides an overview of the resulting timing behavior, and show how parameters for specification templates can be derived from literature reviews. The proposed procedure can be used to define application- and scenario-specific parameter sets, which allow the communication model to be used to verify consistency of specifications in the presence of unreliable communication.

It is obvious that the combination of the generic communication model and the example parameter sets characterizes the temporal effects in more detail than simple **Reaction** patterns, such as those used in the specifications of the V2X component defined as part of the CLTA application example (see Tables 3.4, 3.7, 3.10, and 3.14). At the same time, the development of the generic communication model emphasized the importance of choosing an appropriate level of abstraction when specifying temporal properties. Although the processing steps considered are simple, the resulting specification templates are not trivial. As we will see in Chapter 7, formally verifying MTSL 1.5+MC specifications of a comparable size can lead to very large state spaces that are difficult to analyze.

5 Modeling of Unreliable Communication

Tab. 5.5: Example parameter sets for IEEE 802.11p and C-V2X

Parameter	Symbol	Parameter Values		
		IEEE 802.11p ¹	C-V2X (SL) ²	C-V2X (UL/DL) ²
Inter-arrival time of messages	P_{msg}	*	*	*
Minimum latency	δ_{min}	$(0, 1] \mu s^3$	$(0, 1] ms^4$	$(0, 1] ms^4$
SEND				
Preprocessing latency	δ_{pre}	*	*	*
Channel access request latency	δ_{req}	$(0, 100] ms^5$	$[74.6, 79.1] ms^6$	$[264, 424] ms^6$
Channel access request timeout	δ_{req}^x	$> 100 ms$	$> 79.1 ms$	$> 424 ms$
Sending latency	δ_{send}	*	*	*
TRANSMIT				
Transmission latency	δ_{trans}	$[0, 200] ms^7$	$[20.5, 113.6] ms^6$	$[35.8, 110.8] ms^6$
Inter-arrival time of high load	P_{load}	*	*	*
High-load transmission latency	δ_{load}	$[0, 500] ms^7$	$[20.5, 113.6] ms^6$	$[35.8, 110.8] ms^6$
PDR (numerator) ⁸	K_{load}	7^9	4^{10}	9^{11}
PDR (denominator) ⁸	N_{load}	10^9	9^{10}	10^{11}
Relief latency	δ_{relief}	*	*	*
Inter-arrival time of failures	P_{fail}	*	*	*
Recovery latency	δ_{recov}	*	*	*
RECEIVE				
Receiving latency	δ_{rec}	$5 ms$	$3 ms^7$	$3 ms^7$
Receiving timeout	δ_{rec}^x	$> 5 ms$	$> 3 ms$	$> 3 ms$
Error correction latency	δ_{cor}	*	*	*
Error correction timeout	δ_{cor}^x	*	*	*
Postprocessing latency	δ_{post}	*	*	*

* = application- or scenario-specific ¹ Broadcast ² Unicast ³ Based on Section 6.1.1

⁴ Based on TTI definition in Section 5.2.2 ⁵ [577, p. 4] ⁶ [389, Table 2] ⁷ [265, Figure 3]

⁸ Assumed channel utilization of 80 %, vehicle speed of 50 km/h, and distance of 800 m

⁹ [265, Figure 13] ¹⁰ [618, Figure 3] ¹¹ [431, Figure 8]

Therefore, it is necessary to balance accurately representing relevant time phenomena with maintaining manageable levels of specification and analysis model complexity.

With respect to the envisioned design approach, the generic communication model and the example parameter sets contribute to the refinement of the initial top-level architecture in the first process phase (see Figure 3.1). The model can thus be understood as a tool to structure the decomposition of SoCPS architectures using predefined building blocks. It also provides initial timing specifications that address state-of-the-art V2X communication systems. The derived model and the example parameter sets enable early consideration of temporal properties of unreliable communication by considering relevant failure modes in functional system design. Finally, the results serve as input for VITs on SoCPS level, taking into account temporal effects of unreliable communication.

6 Mapping of Specifications to Imprecise Local Clocks¹

An essential basis for reasoning about timing properties is the existence of a common notion of time to which all parts of the system under consideration adhere. This is true, for example, for the processing units of a distributed system that are connected by buses and jointly realize overarching functionalities, such as the control of modern steer-by-wire airplanes or CHAVs. Interestingly, it also applies to systems in which relativistic effects play a role. One of the key technical challenges of GNSSs is to maintain clock synchronization among satellites with the highest possible accuracy.

While such a globally defined time domain is suitable for most design-time verification tasks, things get complicated when timing properties must be observed at run-time, for example, by safety-related monitoring mechanisms such as watchdogs. This is because technical implementations typically rely on hardware clocks that are inevitably subject to inaccuracies. The effects of imprecise clocks become apparent, for example, when a function of a CHAV has to calculate the age of a data packet sent by another vehicle, where the arrival time is obtained from its local clock, while the time stamp of the data generation was recorded based on the local clock of the sender. The difference between these two time stamps will not give the correct age if the two clocks are not the same.

The goal of this chapter is to map timing specifications to imprecise local clocks. To this end, we develop an as general as possible approach for comparing clocks based on the definition of *clock relations* between global and local clocks. More specifically, we formally reason about the conditions under which timing specifications are considered consistent, starting with very few assumptions on the clock relations involved. We then incrementally add further assumptions until we are able to define a set of MTSL 1.5+MC specification patterns, for which we can show that their global and local versions are consistent.

As we will find, it is generally impossible to establish *strict consistency* for most clock relations and timing specifications. For this reason, we will consider other, weaker notions of consistency. The careful use of *under-* and *overapproximations* allows the construction of meaningful and traceable relations between global and local clocks. For this purpose, we reuse existing specification patterns from MTSL 1.0 for defining clocks and clock references. They are complemented by formal definitions of faithful and approximate clock relations. The extensions concern the MULTIC design paradigms and address the

¹An earlier version of this chapter has been prepared in [558]. The results are the outcome of years of joint work with Ingo Stierand. The focus of my contribution was the detailed investigation of clock relations for clocks with typical properties, the definition of under- and overapproximations as well as the application of the resulting translation schemes on example specification patterns.

fourth methodological gap “*Insufficient support for mapping specifications to imprecise local clocks*” (MG-4). Moreover, the results contribute to answering RQ-2.3 and RQ-3.

The chapter is structured as follows. In a first step, we review existing approaches for handling imprecise local clocks in Section 6.1. Then, in Section 6.2, we formally define what it means to check compliance with specifications based on global and local clocks. Based on these definitions, we discuss potential solutions to the problem of establishing consistency between the two worlds in Section 6.3, depending on the specifications and the available knowledge about the relations between global and local clocks. For this purpose, we investigate local clocks with typical properties and check whether global specifications can be consistently mapped to them. In Section 6.4, we consider weaker notions of consistency to provide practical solutions to the problem. Finally, we instantiate run-time observers for a set of commonly used timing specification patterns in Section 6.5.

6.1 Handling of Imprecise Local Clocks

There is a large body of work on how to handle imprecise local clocks. In hardware design, for example, engineers have to deal with clock skew due to finite propagation delays of clock signals on the wires [136, 159, 487]. In addition, hardware-based techniques have been developed to maintain common clock domains for synchronization in bus systems, e.g., in FlexRay buses [9]. Considerable efforts have also been made to develop software solutions to mitigate clock drifts in distributed systems, such as the *Network Time Protocol* (NTP) [423] and the *Precision Time Protocol* (PTP) [317]. All approaches have in common that they typically reduce the influence of inaccuracies to a quantifiable level, but can never completely avoid them. In order to ensure safe and reliable interaction of individual CSs of cooperative SoCPSs based on implementations of their previously refined timing specifications, a careful consideration of the remaining uncertainties is required. As a first step, suitable concepts and formalisms for describing the characteristic properties of local clocks are being sought to define the desired clock relations.

In the following, we will review existing approaches for handling imprecise local clocks. The related work can be divided into five categories, namely *Clock Synchronization*, *Globally Asynchronous*, *Locally Synchronous Systems*, *Detection of Global State Predicates*, *Deterministic Coordination Approaches*, and *Modeling of Clock Characteristics*.

6.1.1 Clock Synchronization

One of the pioneering theoretical treatises on time, clocks, and the ordering of events in distributed systems was the seminal work of Leslie Lamport [377]. Lamport studied the maintenance of a global event ordering in the absence of a global clock. In this context, he proposed an algorithm that establishes synchronization between local clocks. The algorithm forces the involved clocks to keep up with the fastest clock in the system. However, the relation to a global clock was out of scope. Lamport’s results formed the basis for decades of research on clock synchronization algorithms, mainly in the field of chip design, very-large-scale integration, and parallel computing. Two particularly

relevant research areas are the synchronization of clocks in the presence of faults [159, 378, 414, 603] and clock synchronization in distributed real-time systems [366].

Even today, synchronization mechanisms continue to be used in industrial practice. Several concepts and strategies for clock synchronization such as the *remote clock reading* method [158], the *time transmission* method [40], or the *set-valued estimation* method [391] can be used to ensure a common notion of time between physically distributed systems [565]. One of the most popular representatives of this kind of algorithms is the *offset delay estimation* method employed by NTP [423]. In addition, there are other well-established software protocols for time synchronization among distributed systems, such as PTP [317], which provides a significantly higher accuracy than NTP due to the exclusive use of hardware time stamps written directly at the network interfaces.

In addition, there is a wide range of existing technical realizations, e.g., implementations of simple synchronization mechanisms based on the transmission of specialized bit patterns or more complex functions such as the *Timing Synchronization Function* (TSF) in IEEE 802.11p, which is “*required for frequency hopping and scheduling of sleep phases*” [279, p. 46] in V2X communication. The synchronization algorithm relies on a hardware counter with a resolution of 1 μ s used as a timer, and a local clock oscillator with a frequency accuracy of $\pm 0.01\%$ to ensure progress. The system can operate in different modes, supporting *centralized* communication with intelligent infrastructure systems and *decentralized* communication via ad-hoc networks between individual CHAVs. TSF was originally designed for use in single-hop networks. In addition, it has limited scalability and is not capable to efficiently handle congestion [279, p. 47]. As the number of communication nodes increases, the node with the fastest clock may have difficulty sending its beacon frames. As a result, its local clock drifts away from the other clocks, which is also known as the *fastest node asynchronism* problem [303]. In multi-hop networks, the emergence of so-called *clock islands* can occur, which may lead to *time partitioning* problems [335]. Rentel and Kunz, among others, proposed a solution to these problems by introducing a synchronization algorithm that improves accuracy, scalability, and robustness [499].

In order to standardize the use of synchronization mechanisms for automotive bus systems, the AUTOSAR consortium has developed detailed specifications for time synchronization over CAN [7] and FlexRay [9] buses as well as Ethernet-based communication networks [8]. Other standards, such as *Time-Sensitive Networking* (TSN) [227], focus on establishing deterministic end-to-end transmission of Ethernet-based communication streams using well-established synchronization protocols to support hard real-time requirements. Similarly, most industrial communication standards for safety-critical or time-sensitive applications rely on some form of high-precision clock synchronization.

Nevertheless, inaccuracies may arise from the mapping of specifications formulated at design time to imprecise local clocks used at run-time. In addition, safety-relevant residual uncertainties emerge in basically all known synchronization concepts. In this context, the use of higher synchronization frequencies can help to reduce the synchronization error, but does not effectively prevent it in general. Moreover, the trade-off between synchronization accuracy and communication overhead limits the applicability of high-precision protocols in real systems. Although there are approaches for distributed monitoring with *partial*

synchrony [96], clock synchronization does not solve all problems resulting from local time domains in cooperative SoCPSs. In summary, while clock synchronization is a valuable building block for establishing a common notion of time, it is not a panacea. Therefore, the resulting effects on timing behavior must always be carefully considered.

6.1.2 Globally Asynchronous, Locally Synchronous Systems

Another milestone in handling local time domains was the development of *Globally Asynchronous, Locally Synchronous* (GALS) systems [136]. Today, its basic principle serves as a basis for designing architectures of electronic circuits. It includes the subdivision of a given system into a number of *synchronous* components, each representing an independent clock module. A set of *asynchronous* components realize the communication between these modules according to their own timing behavior, e.g., by means of *First In, First Out* (FIFO) buffers. GALS introduces different measures to prevent the undesirable effect of so-called *metastability*, i.e., the emergence of undefined intermediate states when accessing shared data simultaneously. Two examples of such measures are the “stretching” of the sampling clock or the postponement of the data exchange by several cycles.

Over the years, numerous extensions have been developed. For example, the introduction of *pausable clocking control* helps to avoid synchronization failures between different modules by adjusting their local clocks [616]. In addition, the architectural design of GALS systems has been simplified by providing a library of predefined modules and wrapper components [437]. The development of a deterministic GALS variant called *synchro-tokens* [282], which employs a specialized wrapper logic to transform asynchronous signals into deterministic output sequences, was a first step to structure the inter-module communication. Finally, Potop-Butucaru et al. established a theory to connect locally-clocked components through correct-by-construction communication protocols [487].

6.1.3 Detection of Global State Predicates

The detection of global state predicates in systems of processes with local states and *asynchronous* or *partially-synchronous* real-time clocks is another relevant field of research. Based on Lamport’s *happened-before* reasoning, Cooper and Marzullo presented an initial solution [156] to detect global properties in asynchronous systems such as transmission commits that happen only when all data locks are valid. Later on, the concepts were extended to support partially-synchronized systems with a fixed error bound [413] and varying errors using approximately-synchronized clocks [560]. Schwarz and Mattern, among others, have applied similar approaches for detecting global state predicates to determine causal relations between events in distributed computations [533].

Although the investigations refer to the ordering of events and the evolution of system states, they do not reason about the relation between global and local clocks. Due to the consideration of approximately-synchronized clocks, all observations are impaired by the residual error of the underlying synchronization method. Thus, the accuracy of the detection of global state predicates depends on the accuracy of the synchronization protocol as well as the characteristics of the specifications to be observed.

6.1.4 Deterministic Coordination Approaches

In a similar way, Lohstroh et al. applied Lamport’s ideas for use in correct-by-construction design to establish a coordinated system behavior across multiple clocks [396, 398], but without considering physical effects such as clock drift or skew. The authors introduced the concept of *reactors* as a “*variation on actors that leverage logical time, following classical synchronous-reactive principles*” [398, p. 36:24]. Reactors are composed of a set of *reactions* that are “*triggered by discrete events and may also produce them*” [398, p. 36:8]. In addition, reactors rely on a similar notion of events as in our definitions, but assign logical time values to event values. The authors have also developed a coordination language called *Lingua Franca* (LF) that can be used to define and compose reactors.

Lohstroh et al. have also discussed the distinction between logical and physical time [397]. In this context, the authors introduced *logical time* as an “*engineering model to complement a notion of physical time, which models the physical passage of time*” [397, p. 313]. The idea is to use such an abstract model of time as a tool to achieve deterministic behavior of concurrent systems. While the focus is on establishing a common notion of logical time, the mapping of specifications to imprecise local clocks is out of scope.

In the same line of research, Jun et al. investigated LF to improve the efficiency of timing coordination in distributed discrete-event systems [339]. In this context, discrete-event systems are considered as a suitable MoC for time-sensitive distributed CPSs by instantiating concepts of the distributed simulation technique *High-Level Architecture* (HLA) [313] for the implementation of distributed CPSs. The authors aimed to optimize the performance of distributed discrete-event systems by using a dedicated run-time infrastructure that results in an efficient timing coordination for such kind of systems.

Other authors have focused on practical approaches to ensure error-free interaction of communicating processes. For example, Jubran and Westphal estimated maximum clock drifts in synchronized networks with tree topology to achieve correct timing behavior based on optimal *Time Division Multiple Access* (TDMA) guard times [337, 338].

6.1.5 Modeling of Clock Characteristics

From a modeling perspective, the UML profile for MARTE [455, 457] is one of the best known frameworks for handling imprecise local clocks. MARTE supports the model-based definition of clock properties [35] and clock constraints [34, 407] in UML diagrams by using CCSL (see Section 2.1.1). To this end, the framework provides detailed models for *logical* and *physical* times, dividing the latter into *dense* and *discrete* representatives. The semantics is based on a multifiform notion of time allowing different times to run non-uniformly and possibly independently of any direct relation to physical time.

In terms of the MARTE meta-model, clocks are understood as models that are defined based on a previously specified *time structures*, e.g., by instantiating *IdealClock* or *ChronometricClock* classes. Each clock comprises a *time unit*, a *resolution*, an *offset*, a specification of uncertainties as well as information about the dependencies on other clocks, such as *subclock* or *sampling* relationships. The available clock constraints can be divided into *coincidence-based*, *precedence-based*, and *mixed* constraints [407, pp. 310–311].

CCSL also allows the detailed characterization of imprecisions by explicitly specifying rate variations bounded in terms of so-called *stability* properties [35, p. 570].

Although the modeling artifacts of UML/MARTE allow to capture relevant clock properties, no statements can be made about the validity of combinations of certain parameters or the mapping of specifications to such clocks. None of the approaches from the previous sections is related to CBD. Thus, the explicit specification of clock properties in contract-based timing requirements and the reasoning about valid refinement relations based on imprecise local clocks is a novelty of the presented approach.

6.2 Problem Description

To our own surprise, we have not been able to find any related work that addresses the challenges discussed in this chapter. Almost all existing approaches deal exclusively with the synchronization or relation of local clocks at *run-time*, but not with their relation to a global reference clock or with the construction of local specifications based on global specifications at *design time*. From a conceptual perspective, the original work [377] by Lamport is the most closely related to this work. In addition, the modeling of clocks in UML/MARTE [35, 407] could serve as a basis for the implementation of our results.

When talking about *observations* in the following, we fix the entity of such observations to be the *system*, denoted by S , or one of its components $c \in \mathbf{C}$. We further denote the entities that we may observe as *events*. From our point of view, an event can be virtually anything, like the transmission of data or the change of a system state. According to the system model presented in Section 4.2.2, we assume the set of event values Σ with

$$\Sigma = \bigcup_{p \in \mathbf{P}} \Sigma_p, \quad (6.1)$$

to contain all event values that can potentially be observed. In order to give a meaning to the term *observation*, we consider $\mathbb{T}$ with $\mathbb{T} = \mathbb{R}_0^+$, i.e., the set of non-negative real numbers including zero, as the single global dense time domain. Whether such a time domain exists in the physical world is out of scope for the present work. In fact, we know that it does not exist—we just have to set parts of S in motion at different speeds. For our purposes, it is sufficient to postulate the existence of such *global clock* $\mathbb{T}$ that always indicates the correct time according to the previously defined time domain.

Given such notion of time and events, *observing* means to read the value $t \in \mathbb{T}$ of our global clock whenever an event $e = (\sigma, t)$ with $e \in \Sigma \times \mathbb{T}$ occurs. Based on the definition of events, we consider observations in terms of timed traces, i.e., infinite sequences $(\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}$, where $i \in \mathbb{N}$ denotes the i -th occurrence of one or more events $\vec{\sigma}_i \subseteq \Sigma$, and $t_i \in \mathbb{T}$ is the time of occurrence of the event set according to our global clock. We assume $t_i \leq t_j$ for $i < j$ with $j \in \mathbb{N}$. In order to avoid ambiguities, it should also be noted that events are instantaneous and non-zeno. The former means that an event occurs at a single determined point in time. The latter means that for any time point $t \in \mathbb{T}$ exists some $i \in \mathbb{N}$ such that $t \leq t_i$, i.e., we observe only finitely many events in any finite time interval. In addition, we define the set $\Omega(\Sigma)$, which contains all *possible observations* over Σ .

An infinite sequence $\pi = (\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}$ obtained by observing system S is referred to as a *run* of S . We assume that many different runs can be observed in a system, depending on the circumstances of their observation. This may happen because the system interacts with an environment that is subject to changes (e.g., due to weather conditions) or because of an inherent non-determinism of the system itself. We define the semantics of S by all runs we may observe, i.e., $L_S \subseteq \Omega(\Sigma)$. Finally, we give reason to observations by the definition of *specifications*. We argue that we observe the system in order to check whether the observations conform to some previously defined behavior. Formally, a specification is a set $\varphi \subseteq \Omega(\Sigma)$, and we ask whether S adheres to φ , denoted $S \models \varphi$ and defined as

$$S \models \varphi \iff L_S \subseteq \varphi. \quad (6.2)$$

The problem addressed in this chapter is the fact that there is no omniscient observer with an infinitely precise clock in reality. Instead, we typically have to deal with imperfect clocks that have limited resolutions and are subject to drift and other physical effects that lead to inaccuracies. Suppose we have n such clocks $\mathcal{T} = \{\mathbb{T}_1, \dots, \mathbb{T}_k, \dots, \mathbb{T}_n\}$ with $\mathbb{T}_k \subseteq \mathbb{T}$, $k, n \in \mathbb{N}$, and $1 \leq k \leq n$, each used to observe a set of events $\Sigma_k \subseteq \Sigma$ that may occur in the system. To take into account observations based on imprecise local clocks, we introduce *local observations*. In this context, we assume to have n observers observing only those events for which they are “responsible”, i.e., observer k observes only events from the set Σ_k . The time stamps for these events are taken from clock $\mathbb{T}_k$ and are annotated to the observations. As a result, we obtain n local observations $\pi_k = (\vec{\sigma}_i^k, t_i^k)_{i \in \mathbb{N}}$ with $\vec{\sigma}_i^k \subseteq \Sigma_k$ and $t_i^k \in \mathbb{T}_k$ for each run π of the system.

We assume that all local observations are collected in a single *local run* $\pi' = (\vec{\sigma}_i', t_i')_{i \in \mathbb{N}}$, such that for each event $\sigma \in \vec{\sigma}_i'$ holds that if $\sigma \in \Sigma_k$ then t_i' coincides with the time stamp of σ taken by the local clock $\mathbb{T}_k$. Furthermore, for every $(\vec{\sigma}_i^k, t_i^k) \in \pi_k$, there exists $(\vec{\sigma}_j', t_j') \in \pi'$ such that $\vec{\sigma}_i^k \subseteq \vec{\sigma}_j'$ and $t_i^k = t_j'$. Omitting the formal definition, we denote the corresponding operation as $\oplus$, i.e., $\pi' = \oplus_{\mathbb{T}_k \in \mathcal{T}} \pi_k$. Note that $\oplus$ may involve regrouping and reordering of events. We define $\Omega'(\Sigma)$ as the set of *possible runs* over a given set of local clocks, and denote $L'_S \subseteq \Omega'(\Sigma)$ the set of runs that can be obtained in this way.

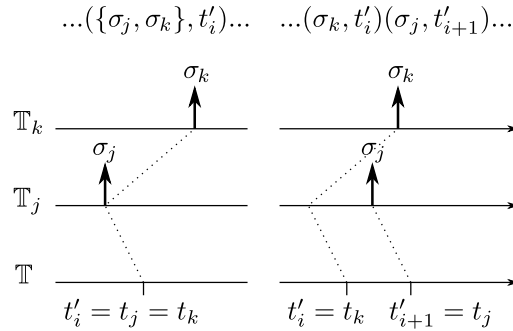


Fig. 6.1: Undesired observation effects due to local clocks

Using local clocks for observations can lead to undesired effects as shown in Figure 6.1. The upper part depicts the timelines of two local clocks $\mathbb{T}_j$ and $\mathbb{T}_k$ with $\mathbb{T}_j, \mathbb{T}_k \in \mathcal{T}$. In

addition, the lower part presents the timeline of the global clock $\mathbb{T}$. In both situations, shown on the left and right, two events σ_j and σ_k occur. Note that the positions of the events on the timelines indicate their time values with respect to the global time domain. The dotted lines between the individual timelines denote positions where all three clocks show exactly the same time value. Consequently, the events involved in the first situation, shown on the left, are assumed to occur simultaneously and are thus collected in a single observation $(\{\sigma_j, \sigma_k\}, t'_i)$. From the perspective of an omniscient observer, however, they occur at different times. Similarly, in the second situation shown on the right, σ_k is assumed to have occurred before σ_j , since the value of the clock $\mathbb{T}_j$ is ahead of $\mathbb{T}_k$. Nevertheless, the situation results in two local observations (σ_k, t'_i) and (σ_j, t'_{i+1}) . In fact, the opposite is true, since σ_j occurred before σ_k .

The examples clearly demonstrate that checking whether a system S satisfies a given specification φ can fail if the verification task relies on runs π' obtained by local observers. In the following, we investigate whether and how we can verify compliance with φ based on a set of observers that rely on imprecise local clocks. More precisely, we ask under which circumstances we can derive a property φ' from φ such that

$$L'_S \subseteq \varphi' \iff L_S \subseteq \varphi. \quad (6.3)$$

6.3 Clock Relations

As we will see, the answer to this question depends, among other things, on the information available about the local clocks and, in particular, on the relation between these clocks and the global clock. To characterize such relations, we define *clock relations* $tr \subseteq \mathbb{T}_1 \times \mathbb{T}_2$ based on the time values of two clocks $\mathbb{T}_1, \mathbb{T}_2 \in \mathcal{T} \cup \{\mathbb{T}\}$. An element $(t_1, t_2) \in tr$ represents the fact that the value t_1 of clock $\mathbb{T}_1$ coincides with the value t_2 of clock $\mathbb{T}_2$. We denote $tr(t_1)$ the set of time values $t_2 \subseteq \mathbb{T}_2$ such that $(t_1, t_2) \in tr$. Conversely, $tr(t_2)$ refers to the set of time values $t_1 \subseteq \mathbb{T}_1$. The only requirement we initially put on tr is that $tr(t_1) \neq \emptyset$ applies for any $t_1 \in \mathbb{T}_1$, and $tr(t_2) \neq \emptyset$ for any $t_2 \in \mathbb{T}_2$.

We use such relations to map local observations to global observations of an omniscient observer, and vice versa. Suppose we have n clock relations tr_k , one for each local clock $\mathbb{T}_k \in \mathcal{T}$ involved in the observation. For each $(\vec{\sigma}_i^k, t_i^k) \in \pi_k$, we obtain a set of pairs $(\vec{\sigma}_i^k, t_i)$, one for each $t_i \in tr_k(t_i^k)$. This mapping is lifted to the level of observations, resulting in a set of runs in which all local time values are mapped to global time values according to clock relations $tr_1, \dots, tr_n$. We denote this mapping by $\alpha(\pi')$ and define

$$\alpha(\pi') = \left\{ \oplus (\vec{\sigma}_i^k, t_i)_{i \in \mathbb{N}} \mid (\vec{\sigma}_i^k, t_i^k) \in \pi_k \wedge t_i \in tr_k(t_i^k) \right\}. \quad (6.4)$$

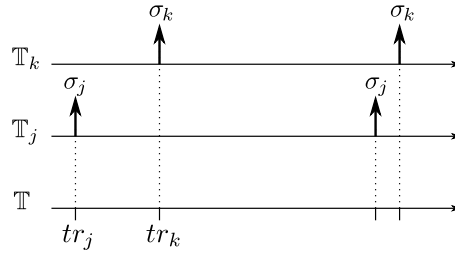
Figure 6.2 illustrates the mapping based on the two example situations from Figure 6.1. Again, the upper part shows the timelines of the local clocks $\mathbb{T}_k$ and $\mathbb{T}_j$, and the lower part the timeline of the global clock $\mathbb{T}$. On the left-hand side, the events σ_j and σ_k are both assigned to the same local time stamp t'_i as part of the local observation π'_1 . The translation into the global time domain results in the global observation $\alpha(\pi'_1)$. On the right-hand side, different local time stamps are assigned to the events. Here, the two

time values are included in $\alpha(\pi'_2)$ based on clock relations tr_j and tr_k . Similarly, we map global observations to local observations. Given a global observation π , we denote $\rho(\pi)$ the set of possible local observations with respect to clock relation tr_k . Finally, we extend the definitions to sets of runs. Given sets $\Pi' \subseteq \Omega'(\Sigma)$ and $\Pi \subseteq \Omega(\Sigma)$, we define

$$\alpha(\Pi') = \cup_{\pi' \in \Pi'} \alpha(\pi') \quad \text{and} \quad \rho(\Pi) = \cup_{\pi \in \Pi} \rho(\pi), \quad (6.5)$$

where $\alpha(\Pi')$ denotes the translation of a set of local runs into a set of global runs and $\rho(\Pi)$ the corresponding translation in the opposite direction.

$$\pi'_1 = \dots(\{\sigma, \sigma_k\}, t'_i) \dots \quad \pi'_2 = \dots(\sigma_k, t'_i)(\sigma_j, t'_{i+1}) \dots$$



$$\begin{aligned} \alpha(\pi'_1) &= \dots(\sigma_j, tr_j(t'_i)) \dots (\sigma_k, tr_k(t'_i)) \dots \\ \alpha(\pi'_2) &= \dots(\sigma_j, tr_j(t'_{i+1})) \dots (\sigma_k, tr_k(t'_i)) \dots \end{aligned}$$

Fig. 6.2: Translation of observations based on local clocks

As the most precise information about clock relations we can think of, suppose we know the exact value of each local clock $\mathbb{T}_k$ for each observation π' at each point in global time. We express this knowledge by relations $tr_{\pi_k, k} = \{(t_i^k, t_i) \mid (\vec{\sigma}_i^k, t_i^k) \in \pi_k\}$, which map each time stamp t_i^k of an observation π_k based on $\mathbb{T}_k$ to a single time point $t_i \in \mathbb{T}$. Thus, for each observation π' , all mappings $tr_{\pi_k, k}$ are uniquely defined and hence α and ρ are functions with $\alpha : \Omega'(\Sigma) \rightarrow \Omega(\Sigma)$ and $\rho : \Omega(\Sigma) \rightarrow \Omega'(\Sigma)$. In addition, $\rho(\alpha(\pi')) = \pi'$ and $\alpha(\rho(\pi)) = \pi$ applies. Therefore, equivalence from Equation 6.3 applies if and only if

$$\varphi = \alpha(\varphi') \quad (6.6)$$

holds. Because of the symmetry, the same holds if and only if

$$\varphi' = \rho(\varphi) \quad (6.7)$$

applies. However, the answer is not effective, since it only applies to observations that can potentially be made in the system. If we want to reason about some run $\pi \in \varphi$ that is no behavior of the system, i.e., $\pi \notin L_S$, we get into problems when talking about how local clock values are related to global time. We could mitigate this problem by extending the definition of $tr_{\pi_k, k}$ to all runs in $\Omega'(\Sigma)$, but the answer is not very helpful either, since it requires too much knowledge. If we know the exact relation between the global and all local clocks, then defining observations based on local clocks is kind of useless.

An alternative solution is to state a relation that holds for *all* observations that could potentially be made. This could be achieved by defining the union of the above relations, i.e., $tr_k = \bigcup tr_{\pi_k, k}$. The definition reflects the idea that we may know the properties of a local clock $\mathbb{T}_k$, but not how it behaves in the actual observation. Unfortunately, this may lead to severe interpretation problems, which we refer to as *snowball effect*.

For example, consider a simple clock whose tick rate gradually decreases due to aging effects, i.e., it runs slower and slower over time. The resulting mappings between the time values of such a local clock $\mathbb{T}_k$ and $\mathbb{T}$ at different observation times are illustrated in Figure 6.3. Figure 6.3a shows a situation right after system startup, when the local clock is still running at the correct speed. An event σ_i occurring at local time $t_k = 3$ is mapped to the identical global time value. If the local clock would keep its pace, then tr_k would remain an identity function and provide deterministic, unambiguous results, i.e., applying the mapping α to a particular local observation π' results in a single global observation π , and vice versa. However, as illustrated in Figure 6.3b, the aging of $\mathbb{T}_k$ leads to increasing differences between global and local time values. Since $\mathbb{T}_k$ shows $t_k = 351$ at the time of occurrence of σ_j , t_k would be mapped to the global time value $t = 351$ according to the previous mapping scheme. On the other hand, $\mathbb{T}$ actually shows $t = 352$ at the time of σ_j . Consequently, $\alpha(\pi')$ may result in multiple global observations, i.e., $\alpha(\pi') = \Pi$. Each of these runs $\pi \in \Pi$ may again require adding multiple observations Π' by applying the mapping ρ , i.e., $\rho(\pi) = \Pi'$, resulting in a series of iterative extensions of the set of possible observations. Formally, this means that $\alpha(\rho(\Pi)) \supseteq \Pi$ holds for this clock relation.

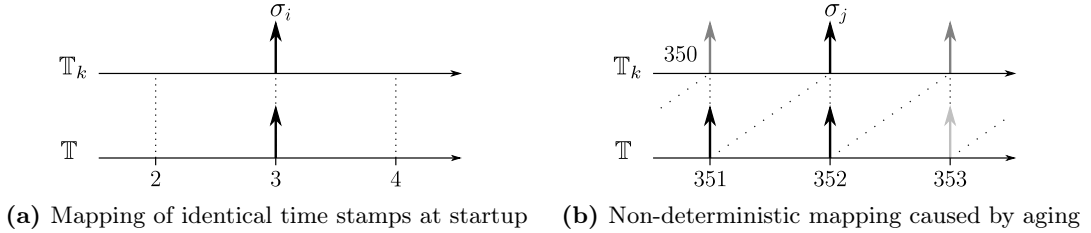


Fig. 6.3: Example of the snowball effect

The snowball effect makes it difficult, if not impossible, to construct properties that adhere to the desired correspondence defined in Equation 6.3. It prevents us from starting with a given global specification φ to derive a local specification $\varphi' = \rho(\varphi)$, because the reverse mapping will generally result in a specification $\tilde{\varphi} = \alpha(\varphi') \supseteq \varphi$, which illegally extends the set of valid runs². For example, consider a local observer that relies on an aging clock as the one considered in Figure 6.3. If such an observer would be responsible for checking whether a particular event occurs in time, situations may arise where parts of $\tilde{\varphi}$ satisfy the original specification φ and others violate it. We can avoid this problem for a certain class of clock relations: We say that tr_k is *faithful* if

$$\forall \Pi \subseteq \Omega(\Sigma) : \alpha(\rho(\Pi)) = \Pi \quad (6.8)$$

²Thus, as a consequence of non-determinism, there is no Galois connection between φ and φ' .

applies. Equivalently, we could say that $\mathbb{T}_k$ is faithful if $\forall \Pi' \subseteq \Omega'(\Sigma) : \rho(\alpha(\Pi')) = \Pi'$ holds. For faithful relations, Equations 6.6 and 6.7 can be generalized as $\alpha(\rho(\varphi)) = \varphi$ and $\rho(\alpha(\varphi')) = \varphi'$, which again are necessary and sufficient conditions for Equation 6.3. The result is a general one and solves our challenge for this kind of relations: If tr_k is faithful, we only need to map the desired property φ along the clock relation into φ' .

For the example above, we obviously did not succeed, since tr_k is not faithful. As it will turn out, faithful clock relations are not easy to find. To prove this, we will investigate local clocks with typical characteristics and check whether the resulting clock relations are faithful or not. As a prerequisite, we infer from Equation 6.8 that $\alpha(\rho(\varphi)) \subseteq \varphi$ must apply for tr_k to be faithful. Based on this, we derive the following two conditions:

$$\forall \pi_1 \in \varphi : \rho(\pi_1) \subseteq \varphi' \quad (6.9a)$$

$$\forall \pi_2 \in \rho(\pi_1) : \pi_1 \in \alpha(\pi_2) \quad (6.9b)$$

We will use the conditions from Equations 6.9a and 6.9b as a proof scheme to check whether faithful clock relations can be established for local clocks with typical properties. To do this, we will first translate a global specification φ into its local counterpart $\varphi' = \rho(\varphi)$, translate the resulting φ' with $\tilde{\varphi} = \alpha(\varphi')$ in the opposite direction, and check whether Equation 6.9 is satisfied by specifications φ , φ' , and $\tilde{\varphi}$. Since Equation 6.8 implies Equation 6.9, we can thus make statements about the *absence* of faithfulness of tr_k .

As a common theme in metric temporal logics, we will use the *time-bounded until* operator $e \mathcal{U}_I f$ [27, 210, 367] as an example property to conduct the proofs, or to give corresponding counter-examples. The operator restricts the time difference between consecutive occurrences of events e and f to an interval $I = [lb, ub]$. Extensions exist that allow the flexible use of open and half-open intervals [25]. Since the basic concept of *time-bounded until* forms a vital basis for a variety of key elements of higher-level specification languages used in practice, it can be understood as a touchstone for checking faithfulness of clock relations. If Equation 6.9 fails for such a fundamental concept, then obviously no complete and consistent mapping of temporal logics is possible in general.

As a first step, we will discuss *monotonicity* and *progress* as two basic properties of clocks. For this purpose, we will consider a modified version of clock relations. The corresponding revision of tr_k also helps us to express other clock properties more conveniently later on. We will show that the redefinition has no effect on the statements about tr_k , e.g., on the validity of Equations 6.8 and 6.9. Afterwards, we will investigate clocks with *finite resolution*, *drift*, and *skew* as well as a *maximum difference* to a reference clock.

6.3.1 Monotonicity and Progress

So far, we have defined clock relations as mappings between individual time values of two clocks. In this context, one might ask whether the snowball effect is—at least partly—caused by such a weak notion, and whether more structured clock relations might help to mitigate the problem. The chosen type of clock relations neglects the fact that the mapping of single time values mixes up different observations. A natural choice to avoid this problem is to consider mappings between sequences of time points. For

6 Mapping of Specifications to Imprecise Local Clocks

this purpose, we define $\tau = (t_i)_{i \in \mathbb{N}}$ as infinite *time sequences* of weakly monotonically increasing time values t_i , similar to runs. We denote the set of all time sequences over clock $\mathbb{T}$ as $\Omega(\mathbb{T})$, and the set of all time sequences over $\mathbb{T}_k$ as $\Omega(\mathbb{T}_k)$. Consequently, we define clock relations $tr_k^\tau \subseteq \Omega(\mathbb{T}) \times \Omega(\mathbb{T}_k)$, which map time sequences over clock $\mathbb{T}$ to (sets of) time sequences over clock $\mathbb{T}_k$, and vice versa. This definition corresponds to the previous one as follows: Each element $(\tau, \tau_k) \in tr_k^\tau$ induces a mapping on clock values $tr_{\tau, \tau_k} = \{(t_i, t_i^k) \mid t_i \in \tau, t_i^k \in \tau_k, i \in \mathbb{N}\}$, and $tr_k = \bigcup_{(\tau, \tau_k) \in tr_k^\tau} tr_{\tau, \tau_k}$, respectively.

The definition is also a natural choice because it implicitly adds the reasonable assumption that clocks are *monotonic*: For any $(\tau, \tau_k) \in tr_k^\tau$ and any two $(t_i, t_i^k), (t_j, t_j^k) \in tr_{\tau, \tau_k}$ holds that $t_i \leq t_j$ implies $t_i^k \leq t_j^k$. Note that we do not enforce strict monotonicity by replacing $\leq$ with $<$, since this would require that local clocks have infinite resolutions. Another reasonable assumption is to postulate *progress* of clocks. We define progress based on a minimal time span $\epsilon_k \geq 0$ for local clock $\mathbb{T}_k$, for which different time points can be safely distinguished, i.e., for all $t_i + \epsilon_k < t_j$ holds $t_i^k < t_j^k$. However, neither monotonicity nor progress changes much with respect to Equations 6.8 and 6.9.

6.3.2 Clocks with Finite Resolution

Almost all technical realizations of clocks are discrete, i.e., they advance in discrete steps defined by a *finite resolution*, say $res_k \in \mathbb{T}$, with $\epsilon_k = res_k$. The clock relation tr_k^τ contains all pairs of time sequences $((t_i)_{i \in \mathbb{N}}, (t_i^k)_{i \in \mathbb{N}})$, where global time values $t_i \in [n \cdot res_k, (n+1) \cdot res_k)$ with $n \in \mathbb{N}$ are mapped to local time stamps $t_i^k = n \cdot res_k$. We say that such a clock “partitions” the global time domain into segments of size res_k .

As mentioned earlier, we will use the *time-bounded until* operator $e \mathcal{U}_I f$ with $I = [lb, ub]$ as a touchstone for checking faithfulness of tr_k^τ . In this context, we assume a global specification φ defined by $e \mathcal{U}_I f$, which contains all runs $\pi \in \Pi$ so that

$$\forall (e, t_i) \in \pi \exists (f, t_j) \in \pi : t_j - t_i \in I \quad (6.10)$$

applies. At the same time, the operator induces a corresponding set of possible runs that comply with the timing constraint. It is thus sufficient to conduct the proofs by considering intervals I that represent the specification. To this end, we extend ρ to map global to local time intervals, and α to map local to global time intervals, respectively. The correspondence between intervals and specifications lies in the fact that for all pairs $((t_i)_{i \in \mathbb{N}}, (t_i^k)_{i \in \mathbb{N}})$ of tr_k^τ and $t_i, t_j \in (t_i)_{i \in \mathbb{N}}$ with $t_j - t_i \in I$ holds $t_j^k - t_i^k \in I'$.

The procedure is as follows: We first derive a realization function for intervals $\rho(I)$ from tr_k^τ , enabling us to translate a global specification interval I to I' . Afterwards, we derive an abstraction function for intervals $\alpha(I')$. While Equation 6.9a is fulfilled by the construction of ρ and α , we check whether $\tilde{I} = \alpha(I')$ is a valid refinement of I according to Equation 6.9b. If this check fails, i.e., $\tilde{I} \supset I$ holds, then tr_k^τ is not faithful given the correspondence between time intervals and timing specifications.

Translation of Global into Local Specifications

We derive from the definition of tr_k^τ for clocks with finite resolution res_k that the translation of a global specification interval $I = [lb, ub]$ to $I' = \rho(I)$ is defined by

$$I' = \left[\left\lfloor \frac{lb}{res_k} \right\rfloor \cdot res_k, \left\lfloor \frac{ub}{res_k} \right\rfloor \cdot res_k \right]. \quad (6.11)$$

As a result of the discretization, lb and ub are mapped to the next lower multiples of the clock resolution, i.e., $lb' = \lfloor \frac{lb}{res_k} \rfloor \cdot res_k$ and $ub' = \lfloor \frac{ub}{res_k} \rfloor \cdot res_k$. This reflects the translation of global time values $t_i \in [n \cdot res_k, (n+1) \cdot res_k)$ to local time stamps $t_i^k = n \cdot res_k$.

The resulting mapping is exemplified in Figure 6.4 using a local clock $\mathbb{T}_k$ with a resolution of $res_k = 2$. In order to show all relevant effects, we have chosen two global specifications φ_1 and φ_2 , which are defined by $e \mathcal{U}_{[1,4]} f$ and $e \mathcal{U}_{[9,11]} f$, respectively. Note that the e event is assumed to occur at time point $t = 0$ in both cases. In the upper part, the global time domain $\mathbb{T}$ including the resulting specification intervals $I_1 = [1, 4]$ and $I_2 = [9, 11]$ (dark blue) is shown. The global time values are mapped to the ticks of $\mathbb{T}_k$ according to tr_k^τ , which leads in the already mentioned partitioning of the time domain that is illustrated by the light-gray triangles. The lower part shows the local specification intervals $I'_1 = [0, 4]$ and $I'_2 = [8, 10]$ (medium blue) that results from Equation 6.11.

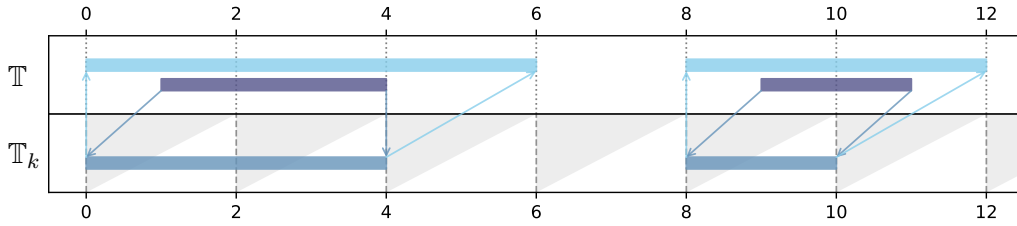


Fig. 6.4: Discretization effects and widenings for a finite-resolution clock ($res_k = 2$)

Here, two relevant effects arise that are explained based on the example mappings. In both situations, a widening on the left-hand side of the intervals occurs, since lb is not a multiple of the resolution res_k , i.e., $lb_1 = 1$ is translated to $lb'_1 = 0$, and $lb_2 = 9$ to $lb'_2 = 8$, respectively. On the other hand, shifts on the right-hand sides of the intervals occur, because the mapping comprises a rounding of ub to the next smaller multiple of res_k , i.e., $ub_1 = 4$ is mapped to $ub'_1 = 4$, and $ub_2 = 11$ to $ub'_2 = 10$, respectively.

Translation of Local into Global Specifications

In the opposite direction, a local specification interval $I' = [lb', ub']$ is translated to the corresponding global specification interval $\tilde{I} = \alpha(I')$ with respect to tr_k^τ according to

$$\tilde{I} = [lb', ub' + res_k]. \quad (6.12)$$

The lower and upper bounds lb' and ub' are mapped to the minimum value of the first and the maximum value of the last associated partition, respectively. Since both values

are multiples of res_k and each partition has size res_k , the translation can be simplified to Equation 6.12. Again, the relevant effects are explained using our example from Figure 6.4. The light blue bars in the upper part illustrate the abstracted specification intervals $\tilde{I}_1 = [0, 6)$ and $\tilde{I}_2 = [8, 12)$ that result from applying Equation 6.12 to $I'_1 = [0, 4]$, and $I'_2 = [8, 10]$, respectively. The shifted lower bounds $lb'_1 = 0$ and $lb'_2 = 8$ resulting from the translation from $\mathbb{T}$ to $\mathbb{T}_k$ are transferred back to the global time domain. In addition, all local upper bounds are shifted to the maximum value $n \cdot res_k$ of the current partition, i.e., $ub'_1 = 4$ is mapped to $ub_1 = 6$, and $ub'_2 = 10$ is mapped to $ub_2 = 12$. Note that these upper bounds are not part of the resulting right-open intervals. Since $\tilde{I} \subseteq I$ consequently does not hold, Equation 6.9b is violated. In other words, clock relations for global specifications with finite-resolution clocks are generally not faithful.

Use of Partitioned Global Specifications

A pragmatic solution to this situation is to discretize all global interval bounds with res_k . Assuming the use of a right-open specification interval defined by $e \mathcal{U}_{[lb, ub)} f$ with $lb = n \cdot res_k \in \mathbb{T}_k$, $ub = m \cdot res_k \in \mathbb{T}_k$, $n, m \in \mathbb{N}$, and $n < m$, Equation 6.11 is replaced by

$$I' = \left[\left\lfloor \frac{lb}{res_k} \right\rfloor \cdot res_k, \left\lfloor \frac{ub - \epsilon}{res_k} \right\rfloor \cdot res_k \right]. \quad (6.13)$$

We denote such kind of specification intervals as *partitioned global specifications*. After inserting the corresponding interval bounds lb and ub into Equation 6.13, it can be simplified to $I' = [lb, ub - res_k]$. The subsequent use of Equation 6.12 to translate the intervals in the other direction results in $\tilde{I} = I$, i.e., the mapping is faithful.

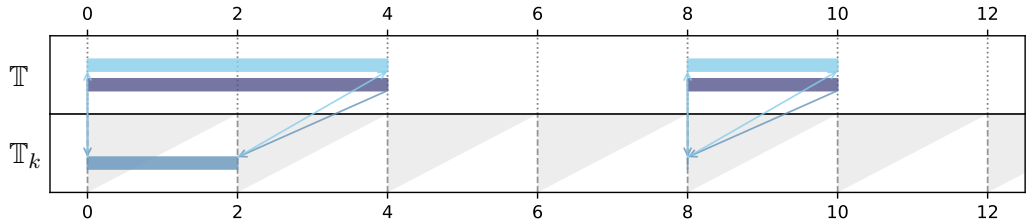


Fig. 6.5: Faithful mapping of partitioned global specifications ($res_k = 2$)

Figure 6.5 shows the translation of two example partitioned global specifications. Again, a local clock $\mathbb{T}_k$ with a resolution of $res_k = 2$ is considered. The use of specification intervals $I_1 = [0, 4)$ and $I_2 = [8, 10)$ allows both a faithful translation from global to local specifications, and vice versa. Since α is the inverse of ρ , the conditions of Equations 6.9a and 6.9b are fulfilled. The clock relation tr_k^τ for clocks with finite resolution res_k and partitioned global specifications is thus faithful according to Equation 6.8.

Note that this result can also be transferred to multiple local clocks. If clocks $\mathbb{T}_k, \dots, \mathbb{T}_l \in \mathcal{S}$ with resolutions $res_k, \dots, res_l \in \mathbb{T}$ are used, a consistent mapping between the time domains is possible if the bounds of the global specification interval $I = [lb, ub)$

are discretized by a common divisor of all local clock resolutions. We will discuss this and other approaches to address the handling of multiple clocks in Section 6.3.6.

As we have shown, the mapping of global specifications with arbitrary interval bounds to finite-resolution clocks causes the snowball effect. To consider more complex clock properties, it is thus not helpful to continue using arbitrary specification intervals, since the resulting clock relations would not be faithful anyway. In the following, we therefore rely on right-open global specification intervals $I = [lb, ub)$. As a special case, we will consider partitioned global specification intervals that satisfy the above conditions, i.e., intervals $I = [lb, ub)$ with $lb = n \cdot res_k \in \mathbb{T}_k$, $ub = m \cdot res_k \in \mathbb{T}_k$, $n, m \in \mathbb{N}$, and $n < m$.

6.3.3 Clocks with Drift

For many systems we can find other reasonable assumptions about clock relations. To this end, we follow ideas from [89, 457] for modeling typical effects on physical clocks. One of the most common effects is that clocks may run too fast or too slow, often referred to as *drift*. The problem has been anticipated since the early days of mechanical clocks, and is also known for clocks implemented by electronic circuitry. Since drift is typically not constant, but depends on environmental conditions such as temperature and humidity, we model this effect by an interval $d_k = [d_k^-, d_k^+] \in \mathbb{T} \times \mathbb{T}$ with $0 < d_k^- \leq d_k^+$. We characterize local clocks with $res_k \in \mathbb{T}$ and d_k by a set of time sequences $(tick_i)_{i \in \mathbb{N}}$ with

$$tick_0 = 0 \quad \text{and} \quad tick_{i+1} = tick_i + drift_i \cdot res_k, \quad (6.14)$$

where $drift_i$ is randomly chosen from d_k . All sequences $(tick_i)_{i \in \mathbb{N}}$ adhering to Equation 6.14 belong to the mapping between $\mathbb{T}$ and $\mathbb{T}_k$, i.e., tr_k^τ contains all elements $((t_i)_{i \in \mathbb{N}}, (t_i^k)_{i \in \mathbb{N}})$, where $(tick_i)_{i \in \mathbb{N}}$ exists such that all $t_i \in [tick_n, tick_{n+1})$ map to $t_i^k = n \cdot res_k$.

Translation of Global into Local Specifications

Based on the definition of tr_k^τ for clocks with drift, a right-open global specification interval $I = [lb, ub)$ is translated into its local counterpart $I' = \rho(I)$ by using

$$I' = \left[\left\lfloor \frac{lb}{res_k} \right\rfloor \cdot res_k \cdot d_k^-, \left\lfloor \frac{ub - \epsilon}{res_k} \right\rfloor \cdot res_k \cdot d_k^+ \right]. \quad (6.15)$$

The equation is almost identical to Equation 6.13, but includes the additional drift factors d_k^- and d_k^+ to account for clock drift. At first glance, the similarity of the two equations might suggest that we can use the same reasoning as for finite-resolution clocks to prove the faithfulness of tr_k^τ . However, as we have already shown, this is not the case. Consider, for example, the aging clock investigated in Figure 6.3. The increasing uncertainties have led to ambiguities in the translation between $\mathbb{T}$ and $\mathbb{T}_k$ that caused the snowball effect, thus preventing faithfulness of clock relation tr_k^τ . Note that Equation 6.15 can be simplified to $I' = [lb \cdot d_k^-, (ub - res_k) \cdot d_k^+]$, if lb and ub are multiples of res_k .

While $tick_0 = 0$ is not subject to any deviations, clock drift leads to accumulating uncertainties for all subsequent ticks. As long as the upper bound of the uncertainty interval around $tick_i$ does not exceed the lower bound of the interval around $tick_{i+1}$, i.e.,

$$i \cdot res_k \cdot d_k^+ < (i + 1) \cdot res_k \cdot d_k^- \quad (6.16)$$

with $i \in \mathbb{N}$, consecutive clock ticks can be safely distinguished. In other words, each partition of global time values $[i \cdot res_k, (i + 1) \cdot res_k) \subseteq \mathbb{T}$ can initially be assigned to a set of local clock values $[i \cdot res_k \cdot d_k^-, i \cdot res_k \cdot d_k^+] \subseteq \mathbb{T}_k$. Unfortunately, this only applies to a number of initial time values and will never hold again after its first violation.

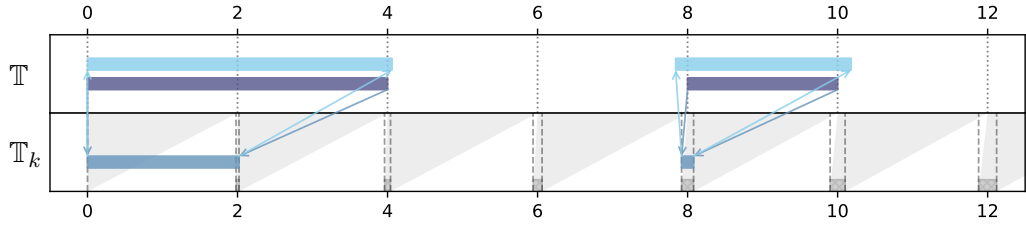


Fig. 6.6: Accumulating uncertainties due to clock drift ($res_k = 2$, $d_k = [0.99, 1.01]$)

In Figure 6.6, the uncertainties resulting from clock drift are illustrated using a local clock $\mathbb{T}_k$ with $res_k = 2$ and $d_k = [0.99, 1.01]$. Again, the light-gray triangles show how the partitions of global time values are mapped to sets of local time values that result from sequences of best-case or worst-case situations, respectively. Based on Equation 6.15, the specification intervals $I_1 = [0, 4)$ and $I_2 = [8, 10)$ shown in the upper part are translated to $I'_1 = [0, 2.02]$ and $I'_2 = [7.92, 8.08]$, which are depicted in the lower part.

Translation of Local into Global Specifications

In the other direction, a local specification interval $I' = [lb', ub']$ is translated into a global specification interval $\tilde{I} = \alpha(I)$ with respect to tr_k^τ according to

$$\tilde{I} = [lb' \cdot d_k^-, (ub' + res_k) \cdot d_k^+]. \quad (6.17)$$

Again, the factors d_k^- and d_k^+ must be taken into account and represent the lack of knowledge of the actual drift affecting a given observation. As a result, the forward and backward translations do not compensate each other and cause the snowball effect. In addition, if closed instead of right-open specification intervals would be used in Equation 6.15, the same problems would arise as for local clocks with finite resolution.

The resulting effects are illustrated in Figure 6.6. While the lower bound $lb'_1 = 0$ of I'_1 remains unaffected by the translation from $\mathbb{T}_k$ to $\mathbb{T}$, all other specification bounds show extensions. The local specification intervals $I'_1 = [0, 2.02]$ and $I'_2 = [7.92, 8.08]$ are related to $\tilde{I}_1 = [0, 4.0602)$ and $\tilde{I}_2 = [7.8408, 10.1808)$. Since $\tilde{I} \subseteq I$ does not hold, Equation 6.9b is violated. The considered clock relation tr_k^τ for clocks with drift is thus not faithful.

6.3.4 Clocks with Skew

Another common phenomenon we consider is referred to as clock *skew*. Due to interference effects in hardware and software, discrete clocks may be subject to jitter, i.e., slight deviations in the timing of incrementing the clock. We model this effect by sequences $(skew_i)_{i \in \mathbb{N}}$ of randomly chosen values from some interval $skew_i \in [0, s_k^+]$. We do not allow clock ticks to overtake others and thus require $s_k^+ < res_k$. We can characterize a clock $\mathbb{T}_k$, which is subject to all effects mentioned so far, i.e., finite resolution, drift, and skew, in terms of a clock relation tr_k^τ , which contains all elements $((t_i)_{i \in \mathbb{N}}, (t_i^k)_{i \in \mathbb{N}})$, where $(tick_i)_{i \in \mathbb{N}}$ and $(skew_i)_{i \in \mathbb{N}}$ exist such that all $t_i \in [tick_n + skew_n, tick_{n+1} + skew_{n+1})$ map to $t_i^k = n \cdot res_k$. Note that $(tick_i)_{i \in \mathbb{N}}$ is defined as for clock drift (see Equation 6.14).

Translation of Global into Local Specifications

For clocks with finite resolution, drift, and skew, the translation of a right-open global specification interval $I = [lb, ub)$ to $I' = \rho(I)$ in terms of tr_k^τ is defined by

$$I' = \left[\left\lfloor \frac{lb}{res_k} \right\rfloor \cdot res_k \cdot d_k^-, \left\lfloor \frac{ub - \epsilon}{res_k} \right\rfloor \cdot res_k \cdot d_k^+ + s_k^+ \right]. \quad (6.18)$$

Since we assume $s_k^+ > 0$, clock skew only affects the upper bounds of the local specification intervals. The effect can lead to a maximum delay of s_k^+ , which is added to the right part of Equation 6.18. According to the partitioning defined in Section 6.3.2, each subset of the global time domain $[i \cdot res_k, (i+1) \cdot res_k) \subseteq \mathbb{T}$ with $i \in \mathbb{N}$ is mapped to a set of local clock values $[i \cdot res_k \cdot d_k^-, i \cdot res_k \cdot d_k^+ + s_k^+) \subseteq \mathbb{T}_k$. Again, Equation 6.18 can be simplified to $I' = [lb \cdot d_k^-, (ub - res_k) \cdot d_k^+ + s_k^+]$, if lb and ub are multiples of res_k .

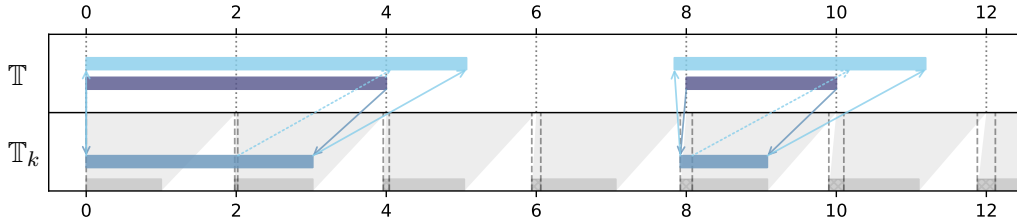


Fig. 6.7: Additional delays due to clock skew ($res_k = 2$, $d_k = [0.99, 1.01]$, $s_k^+ = 0.5$)

Figure 6.7 shows the translation of the example global specification intervals using a local clock $\mathbb{T}_k$ with $res_k = 2$, $d_k = [0.99, 1.01]$, and $s_k^+ = 0.5$. According to tr_k^τ , the lower and upper bounds of $I_1 = [0, 4)$ and $I_2 = [8, 10)$ are mapped to corresponding time values of $\mathbb{T}_k$, thus inducing two local specification intervals $I'_1 = [0, 2.52]$ and $I'_2 = [7.92, 8.58]$. Both intervals are only extended on their right-hand sides by adding an uncertainty interval of $[0, s_k^+]$. Up to this point, the translation of I to I' is easily possible. Since the non-deterministic delays due to clock skew only affect a single $tick_i$ at a time, there are no (additional) accumulating uncertainties that need to be taken into account.

Translation of Local into Global Specifications

In the other direction, $I' = [lb', ub']$ is translated to $\tilde{I} = \alpha(I')$ with

$$\tilde{I} = [lb' \cdot d_k^-, (ub' + res_k) \cdot d_k^+]. \quad (6.19)$$

Since a local observer has no precise information about the actual clock skew that affects the current $tick_i$, the uncertainties resulting from the translation of global to local specifications also affect the translation of local to global specifications. Suppose we want to observe sequences of events e and f with $e, f \in \Sigma$ using $\mathbb{T}_k$ and check whether their distance conforms to a global specification φ . We can formalize this property as

$$\varphi' := \left\{ \pi_k \mid \forall (e, t_i^k) \in \pi_k \exists (f, t_j^k) \in \pi_k : lb' \leq t_j^k - t_i^k \leq ub' \right\} \quad (6.20)$$

with $lb' = \lfloor \frac{lb}{res_k} \rfloor \cdot res_k$ and $ub' = (\lfloor \frac{ub}{res_k} \rfloor - 1) \cdot res_k + s_k^+$. Recall that we obtain global time values $t_i \in tr_k^\tau(t_i^k)$ and $t_j \in tr_k^\tau(t_j^k)$ for local time stamps $t_i^k \leq t_j^k \in \mathbb{T}_k$ according to tr_k^τ . When considering clock skew, the sets of global time values that correspond to $I' = [lb', ub']$ are bounded by $\tilde{lb} = lb'$ (earliest possible time point with $skew_i = 0$) and $\tilde{ub} = ub' + res_k$ (latest possible time point with $skew_i = s_k^+$). Since $\tilde{lb} = lb$ and $\tilde{ub} > ub$ holds, the use of clocks with skew leads to the occurrence of the snowball effect.

The translation from local to global specifications is illustrated in the upper part of Figure 6.7. In addition to the interval expansions due to drift, the uncertainties resulting from clock skew must be considered when determining the lower and upper bounds of $\tilde{I}_1$ and $\tilde{I}_2$. Consequently, the first specification interval $I'_1 = [0, 2.52]$ is translated to $\tilde{I}_1 = [0, 4.5652]$. On the other hand, the translation of $I'_2 = [7.92, 8.58]$ results in the abstracted specification interval $\tilde{I}_2 = [7.8408, 10.6858]$. Since $\tilde{I} \subseteq I$ does not hold, Equation 6.9b is violated. Again, it is not possible to establish faithful clock relations.

6.3.5 Clocks with Difference Constraints

A well-established practice that allows the specification of uncertainties that cannot (yet) be fully characterized is the use of (maximum) *difference constraints*. Such constraints add an uncertainty interval of size $2 \cdot \Delta_k$ with $\Delta_k \in \mathbb{T}$ to each local clock value and cannot be used in conjunction with drift or skew. We model this effect by another sequence $(diff_i)_{i \in \mathbb{N}}$ of randomly selected values $diff_i \in [-\Delta_k, \Delta_k]$. Again, we do not allow clock ticks to overtake others and require $\Delta_k < \frac{res_k}{2}$. The clock relation for local clocks $\mathbb{T}_k$ with a finite resolution $res_k \in \mathbb{T}$ and a maximum difference Δ_k is defined as follows:

$$tr_k^\tau = \left\{ ((t_i)_{i \in \mathbb{N}}, (t_i^k)_{i \in \mathbb{N}}) \mid \exists (tick_i)_{i \in \mathbb{N}}, (diff_i)_{i \in \mathbb{N}} \right. \\ \left. \wedge \forall i \in \mathbb{N} : t_i \in [tick_i - diff_i, tick_i + diff_i] \implies t_i^k = res_k \cdot i \right\} \quad (6.21)$$

Translation of Global into Local Specifications

In general, each partition of global time values $[i \cdot res_k, (i+1) \cdot res_k) \subseteq \mathbb{T}$ is assigned to a set of local clock values $[i \cdot res_k - \Delta_k, i \cdot res_k + \Delta_k] \subseteq \mathbb{T}_k$ that includes the characteristic

uncertainties. Consequently, when translating right-open specifications $I = [lb, ub)$ into their local counterparts, the lower and upper bounds can be derived by using

$$I' = \left[\left\lfloor \frac{lb}{res_k} \right\rfloor \cdot res_k - \Delta_k, \left\lfloor \frac{ub - \epsilon}{res_k} \right\rfloor \cdot res_k + \Delta_k \right]. \quad (6.22)$$

Again, the equation can be simplified to $I' = [lb - \Delta_k, ub - res_k + \Delta_k]$, if lb and ub are multiples of res_k . Since we have explicitly excluded overtaking clock ticks, there is no overlap between the resulting local time intervals of two consecutive partitions.

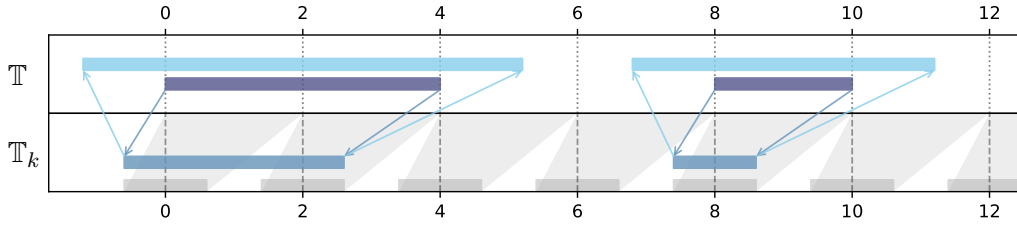


Fig. 6.8: Symmetric widening due to difference constraints ($res_k = 2$, $\Delta_k = 0.6$)

The symmetric widening of the specification intervals that occurs during the translation of global into local specifications is illustrated in Figure 6.8. In this example, the two global specifications I_1 and I_2 are translated to their local representations based on a local clock $\mathbb{T}_k$ with $res_k = 2$ and $\Delta_k = 0.6$. According to the clock relation tr_k^τ , the first specification interval I_1 yields $I'_1 = [-0.6, 2.6]$, and I_2 is mapped to $I'_2 = [7.4, 8.6]$.

Translation of Local into Global Specifications

Since there is no semantic difference between specifications based on $\mathbb{T}$ and those based on $\mathbb{T}_k$, the uncertainties resulting from difference constraints must also be taken into account when interpreting local observations. Thus, the uncertainty interval of size $2 \cdot \Delta_k$ must also be considered for the reverse mapping. This requires the set of possible local clock values $T_j^k = [j \cdot res_k - \Delta_k, j \cdot res_k + \Delta_k] \subseteq \mathbb{T}_k$ to be assigned to a set of possible global time values $\tilde{T}_j = [j \cdot res_k - 2 \cdot \Delta_k, j \cdot res_k + 2 \cdot \Delta_k] \subseteq \mathbb{T}$. Since $T_j \subseteq \tilde{T}_j$ holds, the snowball effect cannot be safely prevented even if the temporal distance between the two involved clocks is strictly bounded by a fixed value Δ_k . According to tr_k^τ , each local specification interval $I' = [lb', ub']$ is translated to the abstracted specification interval

$$\tilde{I} = [lb' - \Delta_k, ub' + res_k + \Delta_k]. \quad (6.23)$$

The corresponding effects are shown in Figure 6.8. Due to the inherent non-determinism, the local specification intervals I'_1 and I'_2 are extended during their translation to the global time domain, resulting in $\tilde{I}_1 = [-1.2, 5.2)$ and $\tilde{I}_2 = [6.8, 11.2)$. Since $\tilde{I} \subseteq I$ does not apply, tr_k^τ for clocks with difference constraints is also not faithful.

6.3.6 Multiple Clocks

So far, we have focused on the relation between the global time domain $\mathbb{T}$ and a single local clock $\mathbb{T}_k$. In this context, we have seen that strict consistency, i.e., faithfulness, is difficult to achieve in many practical scenarios. However, the situation becomes even more complex for specifications that include observations based on multiple local clocks.

In principle, we are able to use the clock relations defined in Section 6.3 to reason about multiple clocks, i.e., to define clock parameters such as resolution, drift, and skew for each local clock in order to relate global and local observations. For example, consider the *time-bounded until* property $e \mathcal{U}_I f$ and the fact that all local observations now involve two local clocks, namely $\mathbb{T}_k$ and $\mathbb{T}_l$, such as in the run $\pi = \dots (e, t_i^k) \dots (f, t_j^l) \dots$ with $t_i^k \in \mathbb{T}_k$ and $t_j^l \in \mathbb{T}_l$. In order to decide on the acceptance of π we would have to check whether $t_i^k - t_j^l \in I'$ is fulfilled. A global observer—for example, considering only clock drift—would observe e at some time point $t_i \in [t_i^k \cdot d_k^-, (t_i^k + res_k) \cdot d_k^+]$ that corresponds to the value of $\mathbb{T}_k$ according to a calculation of minimum and maximum clock drift values based on Equation 6.17. Similarly, the time stamp $t_j \in [t_j^l \cdot d_l^-, (t_j^l + res_l) \cdot d_l^+]$ of the f event can be determined. Based on the calculation of possible time values, we could then ask for some interval $I = [lb, ub]$ such that $t_j - t_i \in I$ if and only if $t_i^k - t_j^l \in I'$.

Unfortunately, there is no such I , since the difference between the clocks involved is generally unbounded. If any two of the bounds $d_k^-, d_k^+, d_l^-, d_l^+$ are unequal, then for any ub there may exist local time stamps $t_i^k - t_j^l \in I'$ that are large enough such that $t_j - t_i > ub$. In other words, for each interval I there exist time points $t^k \in \mathbb{T}_k, t^l \in \mathbb{T}_l$ such that $t_i^k - t_j^l \in I'$ and for at least some corresponding time stamps $t_i \in tr_k^\tau(t^k), t_j \in tr_l^\tau(t^l)$ holds $t_i - t_j \notin I$. Note that the problem cannot be mitigated by introducing (relative) clock drifts, say (d_{kl}^-, d_{kl}^+) , between the clocks $\mathbb{T}_k$ and $\mathbb{T}_l$. According to Equation 6.14, the distance between the two clocks remains unbounded if either $d_{kl}^- \neq 1$ or $d_{kl}^+ \neq 1$ holds.

In order to avoid unboundedness, we require that the clock relations tr_k^τ and tr_l^τ of both clocks are synchronized in some way. Since introducing relative clock drifts does not provide the desired synchronization bounds, we could ask for other reasonable constraints. For example, we could assume that all local clocks involved in the observation of a given global specification remain synchronized within a predefined bound. Therefore, we require that the distance between any two local clocks $\mathbb{T}_k$ and $\mathbb{T}_l$ is not greater than $b_{kl} \in \mathbb{T}$, i.e., for any global time value $t \in \mathbb{T}$ and any two local times values $t^k \in tr_k^\tau(t), t^l \in tr_l^\tau(t)$ holds that $|t^k - t^l| \leq b_{kl}$. Note that this bound is symmetric, i.e., for any two identical time stamps $t^k = t^l$ and any $t_i \in tr_k^\tau(t^k), t_j \in tr_l^\tau(t^l)$ holds $|t_i - t_j| \leq b_{kl}$. As a result, the findings on local clocks with difference constraints from Section 6.3.5 can be applied to multiple clocks. As we will see in Section 6.4, this allows the definition of approximate clock relations to derive over- and underapproximating specifications using $\Delta_k = b_{kl}$.

In addition to the requirements for local clocks and their clock relations, the global specifications that are to be mapped to multiple clocks must also fulfill certain conditions. For example, only specifications that can be mapped consistently to individual clocks can be used for multi-clock observations. In case of clocks with finite resolutions, this

holds if $\mathbb{T}_k, \dots, \mathbb{T}_l \in \mathcal{T}$ and global partitioned specification intervals $I = [lb, ub)$ with $lb = n \cdot \gcd(res_k, \dots, res_l)$, $ub = m \cdot \gcd(res_k, \dots, res_l)$, $n, m \in \mathbb{N}$, and $n < m$ are used.

6.4 Weaker Notions of Consistency

As part of the systematic investigation of typical clock properties in Section 6.3, we were only able to find faithful clock relations for a very limited number of clock characteristics. In summary, it is possible to define such mappings between $\mathbb{T}$ and $\mathbb{T}_k$ if $\mathbb{T}_k$ has a finite resolution $res_k \in \mathbb{T}$ without any uncertainties, i.e., $\forall i \in \mathbb{N} : skew_i \in [0, s_k^+]$ with $s_k^+ = 0$ and $drift_i \in [d_k^-, d_k^+]$ with $d_k^- = d_k^+ = 1$. In addition, all lower and upper bounds of global specification intervals I must consist of multiples of res_k and must have right-open interval bounds. These conditions are only met by very limited sets of specifications and local clocks. In the following, we will therefore investigate weaker notions of consistency in terms of *approximate clock relations* to overcome these restrictions in practice.

6.4.1 Overapproximations

In general, the translation of global into local specifications and vice versa leads to extensions of the original specification intervals. From an optimist's point of view, this means that we implicitly observe *overapproximations* of the specified behavior without additional effort. The investigations on the snowball effect show us that

$$\begin{array}{ccc}
 \varphi & \xrightarrow{\rho} & \varphi' \\
 \text{\scriptsize } \sqcap & \searrow \alpha & \\
 \tilde{\varphi} & &
 \end{array} \tag{6.24}$$

applies. Since $\varphi \subseteq \tilde{\varphi}$ holds, we know for sure that the original specification φ is violated if the abstracted specification $\tilde{\varphi}$ is not fulfilled by the system's behavior. For example, if the global specification φ requires that at least one event $e \in \Sigma$ occurs within a given time interval, φ is not fulfilled if no e event can be observed in the corresponding intervals of φ' and $\tilde{\varphi}$, respectively. This corresponds to the logical consequence

$$S' \not\models \varphi' \implies S \not\models \varphi. \tag{6.25}$$

In other words, a specification φ' obtained by applying ρ to φ , i.e., $\varphi' = \rho(\varphi)$, will never raise false alarms. On the downside, it generally does not detect all violations of φ .

6.4.2 Underapproximations

A second solution that is important for system design is to use *underapproximating* specifications. Here, the idea is to derive an alternative specification φ_δ from φ such that

$$\varphi \rightsquigarrow \varphi_\delta : \alpha(\rho(\varphi_\delta)) \subseteq \varphi \tag{6.26}$$

6 Mapping of Specifications to Imprecise Local Clocks

applies. Since $\varphi \supseteq \tilde{\varphi}_\delta$ holds, we can therefore make the statement that φ is fulfilled if φ_δ is fulfilled. Again, the opposite is not necessarily true. It may happen that the global specification φ is satisfied even though φ_δ is violated by the system's behavior:

$$\begin{array}{ccc}
 \varphi & \xrightarrow{\text{derive}} & \varphi_\delta & \xrightarrow{\rho} & \varphi'_\delta \\
 & \searrow \supseteq & \downarrow \sqcap & \swarrow \alpha & \\
 & & \tilde{\varphi}_\delta & &
 \end{array} \tag{6.27}$$

Consequently, we can characterize the relation between φ'_δ and φ with

$$S' \models \varphi'_\delta \implies S \models \varphi. \tag{6.28}$$

Note that this correspondence only holds if φ requires at least one event to occur within the specified time interval. If φ would require that no event shall occur in a given specification interval, the use of underapproximations does not allow any conclusions to be drawn about the compliance with φ . In this case, only statements about a *potential* violation of φ can be made. A solution to the challenge of observing alternating instances of both types of specification intervals, i.e., intervals in which events must occur and intervals in which no event must occur, it would be possible to combine multiple observers running in parallel, i.e., a first one that relies on an underapproximating specification to observe occurrences of required events and a second one that uses an overapproximating specification to raise an alarm whenever an impermissible event occurs.

In the following, we will focus on defining the *derive* relation $\rightsquigarrow$ used in Equations 6.26 and 6.27. In order to derive an underapproximating specification $I_\delta = [lb_\delta, ub_\delta]$ from $I = [lb, ub]$, a pair of lower and upper bounds lb_δ and ub_δ is calculated *offline*, i.e., at design time, based on the parameters of the global specification φ and the characteristics of $\mathbb{T}_k$. The procedure differs depending on the type of local clock and is presented for *clocks with drift and skew* ($\Delta_k = 0$) and *clocks with difference constraints* ($\Delta_k > 0$).

Clocks with Drift and Skew

According to the results of Sections 6.3.3 and 6.3.4, an underapproximating lower bound lb_δ for a local clock $\mathbb{T}_k$ with $d_k = [d_k^-, d_k^+]$ and $s_k = [0, s_k^+]$ has to fulfill the condition

$$\left\lfloor \frac{lb_\delta}{res_k} \right\rfloor \cdot res_k \cdot d_k^- \cdot d_k^- \geq lb \tag{6.29}$$

with $lb_\delta = n \cdot res_k$ and $n \in \mathbb{N}$ in order to allow a safe and consistent local observation. Consequently, we can determine lb_δ based on lb using Equation 6.30.

$$\begin{aligned}
 & \left\lfloor \frac{lb_\delta}{res_k} \right\rfloor \cdot res_k \cdot d_k^- \cdot d_k^- \geq lb \\
 \iff & \left\lfloor \frac{n \cdot res_k}{res_k} \right\rfloor \cdot res_k \cdot d_k^- \cdot d_k^- \geq lb \\
 \iff & lb_\delta = n \cdot res_k \geq \frac{lb}{d_k^- \cdot d_k^-}
 \end{aligned} \tag{6.30}$$

Similarly, the rules for determining an underapproximating upper bound ub_δ can be defined. Starting from a characterization of the uncertainties that arise during the translation of global into local specifications and vice versa, ub_δ must satisfy

$$\left(\left\lfloor \frac{ub_\delta}{res_k} \right\rfloor \cdot res_k \cdot d_k^+ + s_k^+ + res_k \right) \cdot d_k^+ \leq ub \quad (6.31)$$

with $ub_\delta = m \cdot res_k$ and $m \in \mathbb{N}$ in order to counteract the non-determinism introduced by the snowball effect. Again, we can determine ub_δ based on ub using Equation 6.32.

$$\begin{aligned} & \left(\left\lfloor \frac{ub_\delta}{res_k} \right\rfloor \cdot res_k \cdot d_k^+ + s_k^+ + res_k \right) \cdot d_k^+ \leq ub \\ \iff & \left(\left\lfloor \frac{m \cdot res_k}{res_k} \right\rfloor \cdot res_k \cdot d_k^+ + s_k^+ + res_k \right) \cdot d_k^+ \leq ub \\ \iff & ub_\delta = m \cdot res_k \leq \frac{ub - (res_k - s_k^+) \cdot d_k^+}{d_k^+ \cdot d_k^+} \end{aligned} \quad (6.32)$$

Clocks with Difference Constraints

In case of a local clock $\mathbb{T}_k$ with a bounded maximum difference $\Delta_k > 0$ to the global reference clock $\mathbb{T}$, an underapproximating lower bound lb_δ has to fulfill the condition

$$\left\lfloor \frac{lb_\delta}{res_k} \right\rfloor \cdot res_k - 2 \cdot \Delta_k \geq lb \quad (6.33)$$

with $lb_\delta = n \cdot res_k$ and $n \in \mathbb{N}$ in order to obtain an underapproximation $\tilde{\varphi}_\delta \subseteq \varphi$. The equation follows directly from the results of Section 6.3.5. The resulting inequality is presented in Equation 6.34 and can be used to determine lb_δ based on lb .

$$\begin{aligned} & \left\lfloor \frac{lb_\delta}{res_k} \right\rfloor \cdot res_k - 2 \cdot \Delta_k \geq lb \\ \iff & \left\lfloor \frac{n \cdot res_k}{res_k} \right\rfloor \cdot res_k - 2 \cdot \Delta_k \geq lb \\ \iff & lb_\delta = n \cdot res_k \geq lb + 2 \cdot \Delta_k \end{aligned} \quad (6.34)$$

In a similar way, an underapproximating upper bound ub_δ of the original upper bound ub reliably prevents invalid extensions of the specification intervals if and only if

$$\left\lfloor \frac{ub_\delta}{res_k} \right\rfloor \cdot res_k + 2 \cdot \Delta_k + res_k \leq ub \quad (6.35)$$

with $ub_\delta = m \cdot res_k$ and $m \in \mathbb{N}$ applies. Based on this, ub_δ can be derived from ub using Equation 6.36, which completes the derivation of I_δ for clocks with $\Delta_k > 0$.

$$\begin{aligned} & \left\lfloor \frac{ub_\delta}{res_k} \right\rfloor \cdot res_k + 2 \cdot \Delta_k + res_k \leq ub \\ \iff & \left\lfloor \frac{m \cdot res_k}{res_k} \right\rfloor \cdot res_k + 2 \cdot \Delta_k + res_k \leq ub \\ \iff & ub_\delta = m \cdot res_k \leq ub - res_k - 2 \cdot \Delta_k \end{aligned} \quad (6.36)$$

6.4.3 Particularities in Contract-Based Design

Finally, we discuss an approach to exploit approximate clock relations in the context of CBD (see Section 2.1.3). Recall that A/G contracts are expressed as tuples $\mathcal{C} = (A, G)$ with $A \implies G$, where A specifies assumptions about the behavior of the environment and G specifies guarantees about the behavior of a system or component.

In terms of A/G contracts, a refined contract $(A', G') \preceq (A, G)$ must allow for “more” context, $A' \supseteq A$, i.e., the component must be able to function properly in more diverse environments. Furthermore, it must confine the guarantee, $G' \subseteq G$, i.e., the component must provide “less” behavior compared to G . To put this into context, suppose the behavior of a realized component can be described as a contract $\mathcal{C}'$. The relation to ensure that $\mathcal{C}'$ satisfies the original specification $\mathcal{C}$ can be expressed as follows:

$$\mathcal{C}' \preceq_{\rho} \mathcal{C} : A' \supseteq \rho(A) \wedge G' \subseteq \rho(G) \quad (6.37)$$

Since an extension of the contract’s assumption is allowed, the first part of the refinement relation can be checked by observing overapproximations of the refined assumption (see Section 6.4.1). On the other hand, the refinement of the guarantee can be realized by observing underapproximations of the original guarantee (see Section 6.4.2). The systematic use of combined over- and underapproximations thus contributes to the safe and reliable observation of global A/G contracts based on imprecise local clocks.

6.5 Timing Specifications

We exploit the results of the previous sections by deriving underapproximating specifications for a set of commonly used timing specification patterns, which can be used to instantiate run-time observers. For this purpose, we reuse two MTSL 1.5 specification patterns that have already been introduced in Section 4.2.4 and consider an additional **Synchronization** pattern that can be used to specify synchronization constraints:

Repetition for the specification of recurring occurrences of event sets:

EventSet occurs every P [with [offset O and]? jitter J]?.

Reaction for the specification of latencies between two events specifications:

Reaction(*EventSpec*₁, *EventSpec*₂) within I .

Synchronization for the specification of synchronized occurrences of event sets:

Distance between *EventSet*₁ and *EventSet*₂ within D .

All three specification patterns have in common that they constrain the distance between two (or more) event sets or event specifications (see Section A.4.2) and are therefore related to the *time-bounded until* property considered earlier. Since event sets are defined as $\vec{\sigma} \subseteq \Sigma$ occurring at a single point in time $t \in \mathbb{T}$, we can abstract each of them as a single event without the loss of generality. Similar to Section 4.2.4, we consider

virtual events $\sigma^{virt} \in \Sigma$ with $es \rightsquigarrow \sigma^{virt}$ as appropriate abstractions of event specifications $es \in \mathcal{ES}$, and assume that all events are observed using the same local clock.

Each occurrence of the first event set or virtual event, which is referred to as *trigger event*, introduces a new *valid time window* in which the second event set or virtual event must occur. Since a local observer is only aware of the (discrete) value of its clock $\mathbb{T}_k$ as the sole true time base, the local time stamp t' of a trigger event e_t can be determined by

$$t'(e_t) = \left\lfloor \frac{t(e_t)}{res_k} \right\rfloor \cdot res_k. \quad (6.38)$$

Since all events are observed using the same local clock, we only have to consider the *difference drift* between the two time stamps, i.e., the drift that occurs between the occurrence of a trigger event e_t and a related occurrence of the second event.

In the following, we will first define the semantics of each specification pattern in terms of observations. Then, we will show how global observations of such patterns correspond to local observations based on $\mathbb{T}_k$. For this purpose, we exploit the assumptions about the clock relations between global and local clocks. To do so, we assume the use of a local clock $\mathbb{T}_k$ with a finite resolution res_k , a drift $d_k = [d_k^-, d_k^+]$, and a skew $s_k = [0, s_k^+]$. Afterwards, we derive approximate local specifications that can be used to observe the global specification patterns based on the local clock $\mathbb{T}_k$. Note that we focus on the derivation and practical use of underapproximating specifications based on our results from Section 6.4.2, since the observation of overapproximations is trivially possible.

As a prerequisite for the formulation of underapproximating local specifications, we reintroduce the **Clock Definition** pattern from MTSL 1.0 [93, pp. 21–24], which can be used to define local clocks with typical properties, and integrate it into MTSL 1.5+MC (see Section A.2.1). The syntax of the pattern is defined as follows:

Clock $\mathbb{T}_k$ **has** [**resolution** res_k]? [**drift** $[d_k^-, d_k^+]$]? [**skew** s_k^+]? [**maxdiff** Δ_k]?.

Note that the **Clock Definition** pattern is typically used in assumptions. In addition, we reuse the **Clock Reference** pattern extensions that have originally been defined as part of MTSL 1.0, but were not included in MTSL 1.5 and MTSL 1.5+Modes. The extensions can be used to bind individual specification patterns such as **Repetition**, **Reaction**, and **Age** to previously defined local clocks (see Section A.2.2). By combining the two specification parts, the results of the following sections can be used to formulate local A/G timing contracts, i.e., specifications tailored to a specific local clock $\mathbb{T}_k$.

6.5.1 Repetition Pattern

In accordance with Section 4.2.4, the semantics of the **Repetition** pattern is defined by a family of languages $L_{rep}(\vec{\sigma}, P, O, J)$, where $\vec{\sigma}$ represents the observed *EventSet*:

$$L_{rep} = \{(\vec{\sigma}, t_i)_{i \in \mathbb{N}} \mid t_i = u_i + j_i \wedge u_0 \in O \wedge u_{i+1} = u_i + p_i \wedge p_i \in P \wedge j_i \in [0, J]\} \quad (6.39)$$

Recall that the pattern specifies that the event set $\vec{\sigma}$ occurs every $P \in \mathcal{I}$ time units, possibly further delayed by up to $J \in \mathbb{T}$ time units. In addition, an optional offset of

$O \in \mathcal{I}$ time units can be specified, which defines the set of possible time points of the first event occurrence. Again, we require $J < \min(P)$. Finally, we assume a default offset of $O = [0, \max(P)]$ to ensure compatibility of the examples with the offset semantics of MTSL 1.0. Note that this does not limit the applicability of the approach, since both the general procedure and all equations are suitable for handling arbitrary offset values.

In order to check whether a timed trace complies with a **Repetition** pattern based on local clocks, we aim to derive an underapproximating specification interval for each valid time window. In a first step, we therefore have to ensure that a “shrinking” of the resulting time windows is possible to effectively counteract the snowball effect. For this reason, the pattern parameters must comply with $\min(O) < \max(O) + J$, $\min(P) < \max(P) + J$, or both. If none of these conditions is met, then no conversion into a corresponding right-open interval is possible. Consequently, it would also not be possible to map the **Repetition** pattern to a local clock $\mathbb{T}_k$ with drift $d_k^- \neq d_k^+$, skew $s_k^+ > 0$, or $\Delta_k > 0$.

Assuming the preconditions are met, a pair of underapproximating lower and upper bounds $lb_\delta, ub_\delta \in \mathbb{T}$ can be determined for lower bound $lb = \min(P)$ and upper bound $ub = \max(P) + J$ based on the properties of $\mathbb{T}_k$. For this purpose, Equations 6.30 and 6.32 are used for local clocks with drift and skew ($\Delta_k = 0$), and Equations 6.34 and 6.36 for clocks with difference constraints ($\Delta_k > 0$), respectively. Again, the results must comply with $lb_\delta < ub_\delta$ to obtain valid interval bounds. As a final step, we realize $I_\delta = [lb_\delta, ub_\delta)$ to derive a corresponding local specification interval $I'_\delta = \rho(\varphi_\delta) = [lb'_\delta, ub'_\delta]$ by using

$$\rho(I_\delta) = \begin{cases} \left[\left\lfloor \frac{lb_\delta}{res_k} \right\rfloor \cdot res_k \cdot d_k^-, \left\lfloor \frac{ub_\delta - \epsilon}{res_k} \right\rfloor \cdot res_k \cdot d_k^+ + s_k^+ \right] & \text{if } \Delta_k = 0 \\ \left[\left\lfloor \frac{lb_\delta}{res_k} \right\rfloor \cdot res_k - \Delta_k, \left\lfloor \frac{ub_\delta - \epsilon}{res_k} \right\rfloor \cdot res_k + \Delta_k \right] & \text{if } \Delta_k > 0 \end{cases} \quad (6.40)$$

based on the results from Section 6.3. The underapproximating specification interval I'_δ can then be used to compute a valid time window $I'_{\delta_i} = [lb'_{\delta_i}, ub'_{\delta_i}]$ for each $\vec{\sigma}_i$ by adding lb'_δ and ub'_δ to the time of occurrence $t'(\vec{\sigma}_{i-1}) \in \mathbb{T}_k$ of the previous (trigger) event:

$$lb'_{\delta_i} = \begin{cases} \Delta_k & \text{if } i = 0 \\ t'(\vec{\sigma}_{i-1}) + lb'_\delta & \text{if } i > 0 \end{cases} \quad ub'_{\delta_i} = \begin{cases} ub'_\delta & \text{if } i = 0 \\ t'(\vec{\sigma}_{i-1}) + ub'_\delta & \text{if } i > 0 \end{cases} \quad (6.41)$$

Note that $t'(\vec{\sigma}_{i-1})$ obviously does not exist for the first event set $\vec{\sigma}_0$. For this reason, the special case $i = 0$ was added according to the semantics of the **Repetition** pattern. In addition, it is important to note that the i -th valid time window is not affected by the $i - 1$ -th jitter. As a special feature of the **Repetition** pattern, u_0 and u_i from Equation 6.39 must therefore be used as the locally observed time of occurrence $t'(\vec{\sigma}_{i-1})$. A formal proof of the observability of such specifications is given in [93, pp. 45–47].

Validity Metric

To evaluate the validity of the underapproximating local specification φ'_δ compared to φ' , we define a *validity metric* $V_\delta : \Omega(\Sigma) \times \mathcal{T} \rightarrow \mathbb{R}$ that considers local specification intervals derived from $I_\delta = [lb_\delta, ub_\delta)$ based on a series of event occurrence times $(t)_{i \in \mathbb{N}}$ with

$$V_\delta(\varphi, \mathbb{T}_k) = \sum_{i=0}^n \frac{ub'_{\delta_i} - lb'_{\delta_i}}{ub'_i - lb'_i} \cdot 100, \quad (6.42)$$

where $lb'_i = \rho(lb_i)$ and $ub'_i = \rho(ub_i)$ refer to the corresponding local lower and upper bounds of the i -th valid time window induced by the original global specification φ .

The notion of validity applied here refers to the proportion of the underapproximating specification intervals $I'_{\delta_i} = [lb'_{\delta_i}, ub'_{\delta_i}]$ that are included in the original local specification intervals $I'_i = [lb'_i, ub'_i]$. Statements about the statistical coverage of event occurrences, i.e., about the *confidence* of the observations made, are only reasonable in combination with knowledge about the distributions of events. In the following, we assume for simplicity that all events occur according to a uniform distribution. For realistic systems, this assumption would have to be aligned with the actual distribution of events.

Examples

Given the procedure for deriving underapproximating specification intervals for a set of valid time windows induced by a global **Repetition** pattern, the question arises as to the practical implications of the associated loss of accuracy. Using the validity metric introduced in Equation 6.42, we now have a tool to shed some more light on the effects and interactions between the parameters of global timing specification patterns and the properties of imprecise local clocks. In order to investigate the strengths and weaknesses of the chosen approach, we examine three meaningful examples in the following.

First, we consider a local clock $\mathbb{T}_1$ with $res_1 = 2$ ms, $d_1 = [0.99, 1.01]$, and $s_1^+ = 0.5$ ms, which is used to check the compliance of a series of events with the **Repetition** pattern

$$\varphi_1 : e \text{ occurs every } [6, 8] \text{ ms with jitter } 1 \text{ ms.}$$

specified based on the global time domain $\mathbb{T}$. Figure 6.9 shows an example trace that already reveals some important aspects with direct implications for practical use. Again, we focus on the relation between global and local clocks $\mathbb{T}$ and $\mathbb{T}_1$, global and local specifications φ_1 , φ'_1 , and $\tilde{\varphi}_1$, as well as underapproximations $\varphi_{1\delta}$, $\varphi'_{1\delta}$, and $\tilde{\varphi}_{1\delta}$.

Similar to the examples in Section 6.3, the global time domain $\mathbb{T}$ is shown in the upper part and the local time domain $\mathbb{T}_1$ in the lower part. As part of the example, three e events occur at times 7 ms, 14(+1) ms, and 22 ms. Note that the second event is delayed by a jitter of 1 ms, which is indicated by the densely dotted line between the reference point at 14 ms and the actual time of occurrence at 15 ms. Based on these time values, three global (sub-)specification intervals $I_{1\#1} = [0, 9]$ ms, $I_{1\#2} = [13, 16]$ ms, and $I_{1\#3} = [20, 23]$ ms (dark blue) are successively induced by the **Repetition** pattern. All global specification intervals are translated into local (sub-)specification intervals $I'_{1\#1} = [0, 8.58]$ ms, $I'_{1\#2} =$

6 Mapping of Specifications to Imprecise Local Clocks

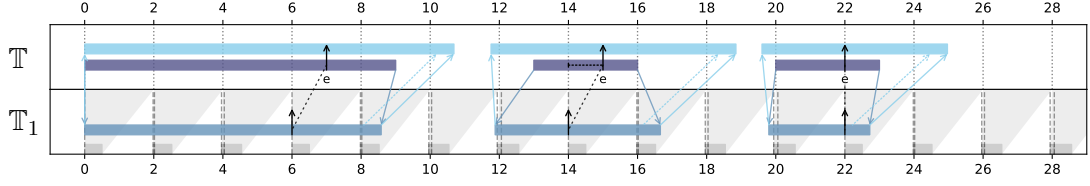


Fig. 6.9: Example specification intervals (blue) and underapproximations (absent) of φ_1 derived for clock $\mathbb{T}_1$ with $res_1 = 2$ ms, $d_1 = [0.99, 1.01]$, and $s_1^+ = 0.5$ ms

$[11.88, 16.66]$ ms, and $I'_{1\#3} = [19.8, 22.72]$ ms (medium blue). The local specification intervals are then mapped back into the global time domain, resulting in abstracted (sub-)specification intervals $\tilde{I}_{1\#1} = [0, 10.6858]$ ms, $\tilde{I}_{1\#2} = [11.7612, 18.8466]$ ms, and $\tilde{I}_{1\#3} = [19.602, 24.9672]$ ms (light blue). Once again, the characteristic extensions of the specification intervals during translation can be seen as a result of the snowball effect.

Following the procedure for deriving underapproximations of **Repetition** patterns, a pair of underapproximating lower and upper bounds lb_δ and ub_δ is determined prior to the observation. In this example, Equations 6.30 and 6.32 result in $lb_\delta = 8$ ms and $ub_\delta = 6$ ms, which violated the condition $lb_\delta < ub_\delta$ formulated above and thus represent invalid interval bounds. In this case, the too narrow specification intervals prevent a sufficient “shrinking” to effectively counteract the snowball effect. For this reason, no underapproximations for φ_1 and $\mathbb{T}_1$ can be generated with the presented approach:

$$V_\delta(\varphi_1, \mathbb{T}_1) = \frac{0 \text{ ms}}{16.28 \text{ ms}} = 0\% \quad (6.43)$$

Although these first results seem sobering, the approach fails for obvious reasons. The unrealistically low resolution of $\mathbb{T}_1$ is too coarse relative to the parameters of φ_1 to obtain effective underapproximations. In order to improve the validity of the approximations, we must either refine the clock resolution or widen the specification intervals by choosing more appropriate pattern parameters. In practice, the specifications are usually fixed, so choosing a higher resolution clock would be the way to go. Nevertheless, we use an adapted version of specification φ_1 to demonstrate the increase in validity by improving the ratio of the clock resolution to the (relative) width of the specification intervals.

For the second example, we reuse $\mathbb{T}_1$, but refer to it as $\mathbb{T}_2$ to make it easier to visually associate the clock index with the example. Moreover, we will focus on a modified version of the **Repetition** pattern with doubled period bounds and a jitter of 5 ms:

$$\varphi_2 : e \text{ occurs every } [12, 16] \text{ ms with jitter } 5 \text{ ms}.$$

The resulting example traces are shown in Figure 6.10. This time, the e events occur at global time points 16(+1) ms, 33(+2) ms, and 49 ms according to the pattern semantics. Again, two of the events are affected by a jitter of 1 ms in case of e_1 , and 2 ms in case of e_2 , respectively. The lower and upper bounds of the global specification intervals $I_{2\#1} = [0, 21]$ ms, $I_{2\#2} = [28, 37]$ ms, and $I_{2\#3} = [45, 54]$ ms are translated into local specification intervals $I'_{2\#1} = [0, 20.7]$ ms, $I'_{2\#2} = [27.72, 36.86]$ ms, and $I'_{2\#3} = [43.56, 55.04]$ ms.

$\tilde{\varphi}_2$ indicates the widening of the resulting specification intervals $\tilde{I}_{2\#1} = [0, 22.927]$ ms, $\tilde{I}_{2\#2} = [27.4428, 39.2486]$ ms, and $\tilde{I}_{2\#3} = [43.1244, 57.6104]$ ms due to the snowball effect.

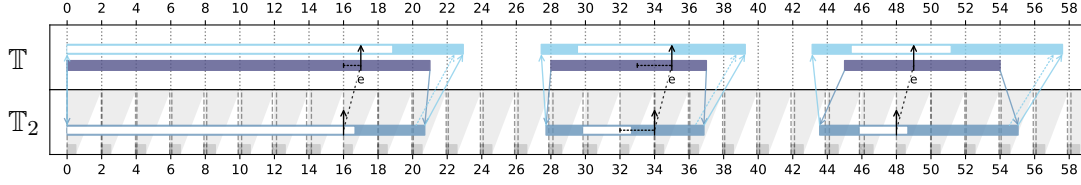


Fig. 6.10: Example specification intervals (blue) and underapproximations (white) of φ_2 derived for clock $\mathbb{T}_2$ with $res_2 = 2$ ms, $d_2 = [0.99, 1.01]$, and $s_2^+ = 0.5$ ms

In contrast to the previous example, calculating underapproximating lower and upper bounds $lb_\delta = 14$ ms and $ub_\delta = 18$ ms yields valid interval bounds that satisfy $lb_\delta < ub_\delta$. Following the procedure presented above, Equation 6.40 is used to determine local bounds $lb'_\delta = 13.86$ ms and $ub'_\delta = 16.66$ ms, which can be used by a local observer to compute valid time windows for each subsequent event $e_{i+1} \in \Sigma$ based on $t'(e_i) \in \mathbb{T}_2$ of event $e_i \in \Sigma$ according to Equation 6.41. Due to the more suitable ratio between the clock resolution and the width of the valid time windows, the resulting uncertainties that arise are relatively smaller and can be compensated more easily due to the relatively larger specification intervals. For local clock $\mathbb{T}_2$ and the **Repetition** pattern φ_2 , valid underapproximations (white) are found for all specification intervals, i.e., $I'_{\delta_{2\#1}} = [0, 16.66]$ ms, $I'_{\delta_{2\#2}} = [29.86, 32.66]$ ms, and $I'_{\delta_{2\#3}} = [45.86, 48.66]$ ms, which are translated into corresponding abstracted (sub-)specification intervals $\tilde{I}_{\delta_{2\#1}} = [0, 18.8466]$ ms, $\tilde{I}_{\delta_{2\#2}} = [29.5614, 35.0066]$ ms, and $\tilde{I}_{\delta_{2\#3}} = [45.4014, 51.1666]$ ms. This results in a validity estimate of

$$V_\delta(\varphi_2, \mathbb{T}_2) = \frac{22.26 \text{ ms}}{41.32 \text{ ms}} \approx 53.87222 \% . \quad (6.44)$$

The results are much more encouraging than the validity estimate obtained for the first example. However, realistic local clocks have much higher resolutions (typically by a factor of 10^6 to 10^9), which we investigate in a third and final example. Here, we consider a realistic local clock $\mathbb{T}_3$ as well as a **Repetition** pattern that has been given practical values. The three relevant parameters of $\mathbb{T}_3$ are listed in Table 6.1 and have been extracted from the technical data sheet of a state-of-the-art clock generator.

Tab. 6.1: Selected parameters of a state-of-the-art clock generator

Parameter	Value Range	Selected Value	Converted Value
Clock Frequency	[300, 1 500] MHz	500 MHz	$res_3 = 2$ ns
Clock Drift	± 5 ppm	± 5 ppm	$d_3 = [0.999995, 1.000005]$
Clock Skew	0.2 ps	0.2 ps	$s_3^+ = 0.0002$ ns

In the following, we use $\mathbb{T}_3$ with $res_3 = 2$ ns, $d_3 = [0.999995, 1.000005]$, and $s_3^+ = 0.0002$ ns to verify the compliance of a series of events with the **Repetition** pattern

$\varphi_3 : \text{ObjectList occurs every } [100, 200] \text{ ms with jitter } 15 \text{ ms.}$

The example *ObjectList* events occur at global time points 118.637792(+6.859233) ms, 315.858727(+12.764847) ms, ..., which are translated into local time points 118637792 ns, 315858726 ns, ... along the clock relation tr_k^τ . Subsequently, the underapproximating lower and upper bounds $lb_\delta = 100.001002$ ms and $ub_\delta = 214.997848$ ms are determined based on Equations 6.30 and 6.32 and satisfy $lb_\delta < ub_\delta$. Based on these values, we obtain a pair of underapproximating local specification bounds $lb'_\delta = 100000501.99499$ ns and $ub'_\delta = 214998920.98943$ ns using Equation 6.40, which can be used to generate valid time windows for each event $\text{ObjectList}_{i+1} \in \Sigma$ based on the time of occurrence $t'(\text{ObjectList}_i) \in \mathbb{T}_3$ of the previous event $\text{ObjectList}_i \in \Sigma$. Applying Equation 6.41 gives the corresponding local specification intervals $I'_{\delta_1} = [0, 214998920.98943]$ ns, $I'_{\delta_2} = [218638293.99499, 333636712.98943]$ ns, ..., which can be used to evaluate whether the timing of required events is correct. The validity metric returns a value of

$$V_\delta(\varphi_3, \mathbb{T}_3) = \frac{692390482848.3158 \text{ ns}}{692409481001.4031 \text{ ns}} \approx 99.99726 \% \quad (6.45)$$

for an assumed observation period of 15 minutes with 6 016 occurring *ObjectList* events. Note that similar results were obtained for both longer (30 min., 60 min., ...) and shorter (10 min., 5 min., ...) observation periods. The high validity estimate clearly demonstrates the approach's suitability and indicates the high accuracy of the underapproximations for realistic clocks. Thus, the approximations provide strong evidence for correct behavior and—assuming a uniform distribution of events—allow the reliable detection of $\sim 99.99\%$ of delayed or omitted events that were required by the global specification.

6.5.2 Reaction Pattern

The **Reaction** pattern expresses a “classical” latency between two events $\sigma_1, \sigma_2 \in \Sigma$, referred to as *EventSpec*₁ and *EventSpec*₂. It specifies that *EventSpec*₂ occurs $I \in \mathcal{I}$ time units after each occurrence of *EventSpec*₁. According to Section 4.2.4, the semantics of the **Reaction** pattern is defined by the family of languages $L_{rea}(\sigma_1, \sigma_2, I)$:

$$L_{rea} = \{(\sigma_i, t_i)_{i \in \mathbb{N}} \mid \forall(\sigma_1, t_1) \exists(\sigma_2, t_2) : t_2 - t_1 \in I\} \quad (6.46)$$

The procedure for deriving underapproximating specification intervals for this class of timing specifications is similar to the one used to observe **Repetition** patterns. In order to allow a “shrinking” of the generated (sub-)specification intervals, the pattern parameters must comply with $\min(I) < \max(I)$. Under this assumption, the underapproximating bounds $lb_\delta, ub_\delta \in \mathbb{T}$ with $lb = \min(I)$ and $ub = \max(I)$ can be calculated based on the pattern parameters and the properties of the local clock $\mathbb{T}_k$ using Equations 6.30 and 6.32 for clocks with drift and skew ($\Delta_k = 0$), and Equations 6.34 and 6.36 for clocks with difference constraints ($\Delta_k > 0$), respectively. All subsequent process steps and conditions remain unchanged. The only exception is the special case for determining the first valid time window in Equation 6.41, which can be omitted for **Reaction** patterns. Trivially, all occurrences of the recurring event set $\vec{\sigma}$ must be replaced by the trigger event σ_1 .

Note that the validity metric introduced in Section 6.5.1a can also be applied to **Reaction** patterns without modifications. The examples given in Section 6.5.1b illustrate all relevant phenomena that occur when observing corresponding specifications. For this reason, no further examples that focus on the **Reaction** pattern are included.

6.5.3 Synchronization Pattern

The **Synchronization** pattern expresses that two event sets $\vec{\sigma}_1, \vec{\sigma}_2 \subseteq \Sigma$, namely *EventSet₁* and *EventSet₂*, must occur together in the sense that they have a bounded distance $D \in \mathcal{I}$. The semantics of this pattern is defined by the family of languages $L_{syn}(\vec{\sigma}_1, \vec{\sigma}_2, D)$:

$$L_{syn} = \{(\sigma_i, t_i)_{i \in \mathbb{N}} \mid \forall(\vec{\sigma}_1, t_i) \exists(\vec{\sigma}_2, t_j) \wedge \forall(\vec{\sigma}_2, t_j) \exists(\vec{\sigma}_1, t_i) : |t_j - t_i| \leq D\} \quad (6.47)$$

Similar to the **Reaction** pattern, the procedure for deriving underapproximations of **Synchronization** patterns differs only slightly from the explanations in Section 6.5.1. In this case, a global specification must satisfy $\min(D) < \max(D)$ to allow for the adjustment of the interval bounds and to ensure a safe and reliable observation. Again, Equations 6.30 and 6.32 for clocks with drift and skew ($\Delta_k = 0$), and Equations 6.34 and 6.36 for clocks with difference constraints ($\Delta_k > 0$), are used to determine a pair of underapproximating lower and upper bounds $lb_\delta, ub_\delta \in \mathbb{T}$ with $lb = \min(D)$ and $ub = \max(D)$ based on the properties of $\mathbb{T}_3$. The translation of $I_\delta = [lb_\delta, ub_\delta)$ into an underapproximating local specification interval I'_δ as well as the calculation of corresponding valid time windows $I'_{\delta_i} = [lb'_{\delta_i}, ub'_{\delta_i}]$ can be done analogously to the approach presented in Section 6.5.1.

In contrast to the semantics of the previous patterns, no order of the referenced events is implied. For this reason, the first occurrence of $\text{de}\vec{\sigma}_1$ or $\vec{\sigma}_2$ is interpreted as *trigger event*, i.e., the next occurrence of the other event set is constrained by the specified time bound. Both the validity metric from Section 6.5.1a and the phenomena explained in Section 6.5.1b—except for the special case $i = 0$ —can be applied without further adjustments. Therefore, we do not provide additional examples in this case either.

7 Tool Support

As we have found, the formulation of consistent timing specifications for real-world systems—and, in particular, for cooperative SoCPSs—is a complex task. At the same time, the careful verification of refinement relations between top-level specifications derived as part of the envisioned design approach (see Figure 3.1) and iteratively refined specifications on lower hierarchy levels is essential to ensure the validity of the contract-based approach and to establish the desired SoCPS-wide timing coherence. One of the main goals of this thesis is to support engineers in ensuring real-time behavior of cooperative SoCPSs by providing suitable tool support. In this context, the availability of modeling and analysis tools that support the enhanced specification means is considered as a key enabler for facilitating the transfer of the results into industrial practice.

While developing the tool support, we faced four main challenges. First, several modeling tools have emerged as de facto standards in industrial practice, so there is no single true modeling environment. Second, diverse requirements exist for the analysis to be performed. Whereas in some cases the focus is on quickly exploring the solution space using simulation-based analysis, formal verification of safety-critical components is mandatory. During the development, we prioritized the efficient, modular, and extensible implementation of model transformations and intermediate formats to facilitate future extensions. A third challenge stems from the fact that multiple MTSL versions have developed over time (see Appendix A) and must be supported to allow flexible use and provide legacy support. The developed tool support aims at unifying all MTSL versions into a single toolchain. It provides standardized data formats and interfaces to ensure comparability and transferability of MTSL specifications across different versions.

The fourth and most significant challenge that arose during the development was that the only existing VIT implementation for MTSL specifications, based on the MULTIC-Tooling tool prototype, was not suitable for use as a basis for the planned extensions. For this reason, the original work plan had to be revised. Since VIT support is considered as a prerequisite for the approach’s success, the focus of the development was shifted to reimplementing corresponding analysis methods. These methods were then extended to include support for mode-dependent specifications. Consequently, the results of this chapter primarily focus on the second phase of the design approach, i.e., the application of the extended MULTIC methodology. At the same time, the results can be used as building blocks to develop tool support for scenario-based activity modeling. A first step in this direction was the definition of a set of modeling guidelines that allow consistent, SysML-compliant modeling of TCADs using Enterprise Architect (see Appendix B).

In the following, we present the UACMbox Toolchain, a modular and extensible toolchain for real-time analyses, as a key result of this thesis. It comprises a model transformation adapter for Enterprise Architect that translates SysML design models into

a lightweight intermediate format to facilitate future extensions. Existing analysis tools are integrated for performing VITs of enhanced specifications based on the intermediate format, thus enabling formal verification of contract refinement. Finally, complexity reduction methods, such as the analysis of partial models, are used to address complex design models. The developed toolchain builds upon the basic idea of the MULTIC-Tooling tool prototype (see Section 2.3.3). It addresses the technical gaps “*Missing integration of industry-relevant modeling tools*” (TG-1) and “*Limited tool support for automated analysis*” (TG-2), thus contributing to answering research questions RQ-3 and RQ-4. Moreover, the toolchain demonstrates the practical applicability of the approach and serves as a proof-of-concept implementation for VITs of mode-dependent specifications.

The chapter is organized as follows. In Section 7.1, we present the functional and non-functional requirements that were collected as a basis for the development. Section 7.2 then describes the UACMbox Toolchain from a conceptual perspective. The implementations of the Enterprise Architect adapter, the intermediate format toolbox, and the RTana_{2sim} analysis backend are presented in Section 7.3. Finally, in Section 7.4, we evaluate the applicability and practical use of the developed tool support in terms of specification language support and performance of the most important toolchain components.

7.1 Requirements

Prior to the implementation of the intended tool support, functional and non-functional requirements for the planned technical realization were identified. In particular, the results of Chapters 3 to 6 as well as the experiences from the development of the MULTIC-Tooling tool prototype (see Section 2.3.3) were taken into account. In total, six functional top-level requirements with 15 lower-level requirements were collected:

- R-F-1** The tool *must* allow the modeling of functional architectures.
 - R-F-1.1** The tool *must* allow the modeling of hierarchical components.
 - R-F-1.2** The tool *must* allow the modeling of event and variable ports.
 - R-F-1.3** The tool *must* allow the modeling of signals between ports.
- R-F-2** The tool *must* allow the annotation of MTSL specifications.
 - R-F-2.1** The tool *should* allow the use of MTSL 1.0 specifications.
 - R-F-2.2** The tool *must* allow the use of MTSL 1.5 specifications.
 - R-F-2.3** The tool *must* allow the use of MTSL 1.5+MC specifications.
- R-F-3** The tool *should* support the formulation of MTSL specifications.
 - R-F-3.1** The tool *should* include an editor to formulate specifications.
 - R-F-3.2** The tool *should* provide syntax highlighting.
- R-F-4** The tool *should* support automated model validation.
 - R-F-4.1** The tool *should* ensure modeling language compliance.

- R-F-4.2** The tool *should* allow automated MTSL syntax checks.
- R-F-4.3** The tool *should* enable basic consistency checks between the functional architecture and the annotated MTSL specifications.
- R-F-5** The tool *must* be able to perform VITs.
 - R-F-5.1** The tool *must* support VITs of mode-independent specifications.
 - R-F-5.2** The tool *must* support VITs of mode-dependent specifications.
 - R-F-5.3** The tool *should* support model-checking-based VITs.
 - R-F-5.4** The tool *should* support simulation-based VITs.
- R-F-6** The tool *should* visualize the results of VITs.

In addition, eight non-functional requirements were elicited, focusing primarily on minimum requirements on the number of available modeling and analysis tools, as well as on general features such as modularity and extensibility already mentioned in RQ-4:

- R-NF-1** The tool *must* interface at least one industrial modeling tool.
- R-NF-2** The tool *must* be easily extensible to integrate additional modeling tools.
- R-NF-3** The tool *must* support industry-relevant modeling languages.
- R-NF-4** The tool *must* interface at least one analysis tool.
- R-NF-5** The tool *must* be easily extensible to integrate additional analysis tools.
- R-NF-6** The tool *must* be modular, i.e., there must be as few dependencies as possible between the modeling and analysis tools.
- R-NF-7** The tool *must* be executable under Windows 10 (or higher).
- R-NF-8** The tool *should* provide optional support for command line execution (e.g., to transfer computationally expensive tasks on computing platforms).

Based on the identified requirements, a preliminary selection of the frameworks for timing specification investigated in Section 2.1.1 was made. The MULTIC-Tooling tool prototype appeared to be the most suitable candidate for the planned extension due to its focus on applying the MULTIC methodology in early design phases (R-F-1) and its existing MTSL 1.0 support (R-F-2). It also offers automated model validation (R-F-4), simulation-based VITs of mode-independent specifications (R-F-5.1, R-F-5.4), and result visualization (R-F-6). However, a more detailed analysis revealed that an extension of the existing prototype would have been difficult and would most likely not have achieved the desired degree of modularity and extensibility (cf. R-NF-2, R-NF-5, R-NF-6).

Although the prototype is based on the industry-relevant modeling language SysML (R-NF-3), it is deeply integrated into a local instance of the open-source modeling environment Papyrus [202], which is not suitable for transfer into industrial practice (cf. RQ-4). At the same time, the execution of the model validation and the MTSL 1.0 syntax check depend on features of the analysis backend TSLsim, which contradicts the required modularity (cf. R-NF-6). In addition, the exclusive use of open-source

components and the prototype nature resulted in performance limitations. Other reasons included a lack of personal expertise in C++ template programming, which was used extensively in the development of the TSLsim analysis backend, and the limited availability of the original core developers, some of whom had moved on to other employers, resulting in a lack of detailed knowledge about the internal implementation. During the evaluation, we also discovered that some of our test models that were not part of the original test suite [93, Chapter 11] produced inconsistent (i.e., potentially incorrect) analysis results, which could be due to errors in our own design models or specifications, errors in the translation into the analysis model, or errors in the analysis backend itself. Due to limited experience in C++ template programming and the lack of knowledge about the internal implementation, debugging such problems would have been very time-consuming.

For the above reasons, we decided to implement a new modular and extensible toolchain for real-time analysis and integrate parts of the original tool prototype developed in MULTIC-Tooling (in particular the analysis backend) into it. The resulting toolchain is referred to as the *UACMbox Toolchain* and is presented in the following section.

7.2 UACMbox Toolchain

From a tooling perspective, the *UACMbox Toolchain* is the main result of this thesis. It includes integrations into industry-relevant modeling tools, model transformations from design to analysis models as well as interfaces to different analysis backends. Most of the toolchain components—except for the modeling and analysis tools themselves—have been implemented as part of this work. The toolchain constitutes a modular and extensible reimplement of the MULTIC-Tooling tool prototype (see Section 2.3.3). The integration of industry-relevant modeling environments enables the modeling of systems at different levels of abstraction. Based on this, the toolchain enables real-time analysis of systems and components. As part of this thesis, the focus is on providing support for simulation- and model-checking-based VITs of MTSL timing specifications.

Figure 7.1 provides an overview of the UACMbox Toolchain, which is divided into three stages. A set of *Modeling Environments and Data Sources* provides system descriptions in the form of *design models*, which can be automatically translated into an internal intermediate representation using model transformation adapters. The resulting *Unified Abstract Component Model* (UACM) models with annotated MTSL specifications can then be read by the *UACM Toolbox* (UACMbox), which converts them into suitable *analysis models* using a set of automated model transformations called *transformers*. The third stage, *Analysis Tools and Monitoring Frameworks*, comprises analysis backends for timing analyses and monitoring frameworks, e.g., to enable distributed run-time monitoring. After performing the analysis or monitoring tasks, the results are fed back to the model transformation adapters from the first stage, based on references to the intermediate representation. The adapters are then responsible for displaying the analysis or monitoring results to the engineers, e.g., by using model annotations to the design models, dialog boxes, as well as error, warning, or info messages to indicate identified problems or required improvements. Returning control to the engineers closes the “engineering cycle”

(see Figure 2.8), i.e., after the manual adaptation of the previously analyzed design models, the next run of the toolchain can be initiated to obtain updated results.

According to their intended functionality, each of the stages is responsible for fulfilling a subset of the functional and non-functional requirements identified in Section 7.1. The *Modeling Environments and Data Sources* stage directly addresses functional requirements R-F-1 to R-F-4 and non-functional requirements R-NF-1 to R-NF-3. The second stage, *Intermediate Representation and Model Transformations*, contributes to R-NF-6 by introducing an intermediate format that decouples modeling and analysis tools. Finally, the *Analysis Tools and Monitoring Frameworks* stage addresses R-F-5 and R-NF-4 to R-NF-5 by providing VIT support based on different analysis backends. R-F-6 can be mapped both to the first and last stages. The requirements related to the execution of the toolchain (see R-NF-7 and R-NF-8) affect the implementation of all stages. In the following, the individual components of the three stages are described in more detail.

7.2.1 Modeling Environments and Data Sources

The current version of the *Modeling Environments and Data Sources* stage includes a model transformation adapter for Enterprise Architect called *MULTIC-Tooling Industrial* (MTI) that supports validating SysML models with annotated MTSL specifications and automatically translating such models into consistent UACM representations. The MTI add-in for Enterprise Architect has been developed as part of this thesis and was inspired by the implementation of the MULTIC-Tooling tool prototype. Additional details about the architecture and the functional scope of the add-in can be found in Section 7.3.1.

In future activities, the first stage could be extended by integrating additional modeling tools. Possible extensions include the interfacing with other SysML modeling environments such as Papyrus [202], Rhapsody Architect for Systems Engineers [308], or Cameo Systems Modeler [167]. Additionally, it would be desirable to add support for SysML v2. Other ideas include the implementation of an adapter for APP4MC [200] that would allow to import component models and hardware/software designs from AMALTHEA models.

7.2.2 Intermediate Representation and Model Transformations

The second stage, *Intermediate Representation and Model Transformations*, relies heavily on an intermediate format called *Unified Abstract Component Model* (UACM), which was developed as part of this work. The language is inspired by the more sophisticated *OCRA System Specification* (OSS) [146] used as the input format of the well-known contract verification tool OCRA [147]. The UACM is intended to provide a lightweight alternative to OSS specifications that can easily be adopted to facilitate the efficient development of model transformation adapters. The UACM meta-model is implemented in terms of a parser and corresponding data structures as part of the UACMbox, which is presented in Section 7.3.2. In the following, we consider UACM from a conceptual perspective.

The UACM is designed to specify abstract component models annotated with specifications such as MTSL timing specifications. It is intended to be used as an intermediate format for the exchange of system descriptions between different modeling and analysis

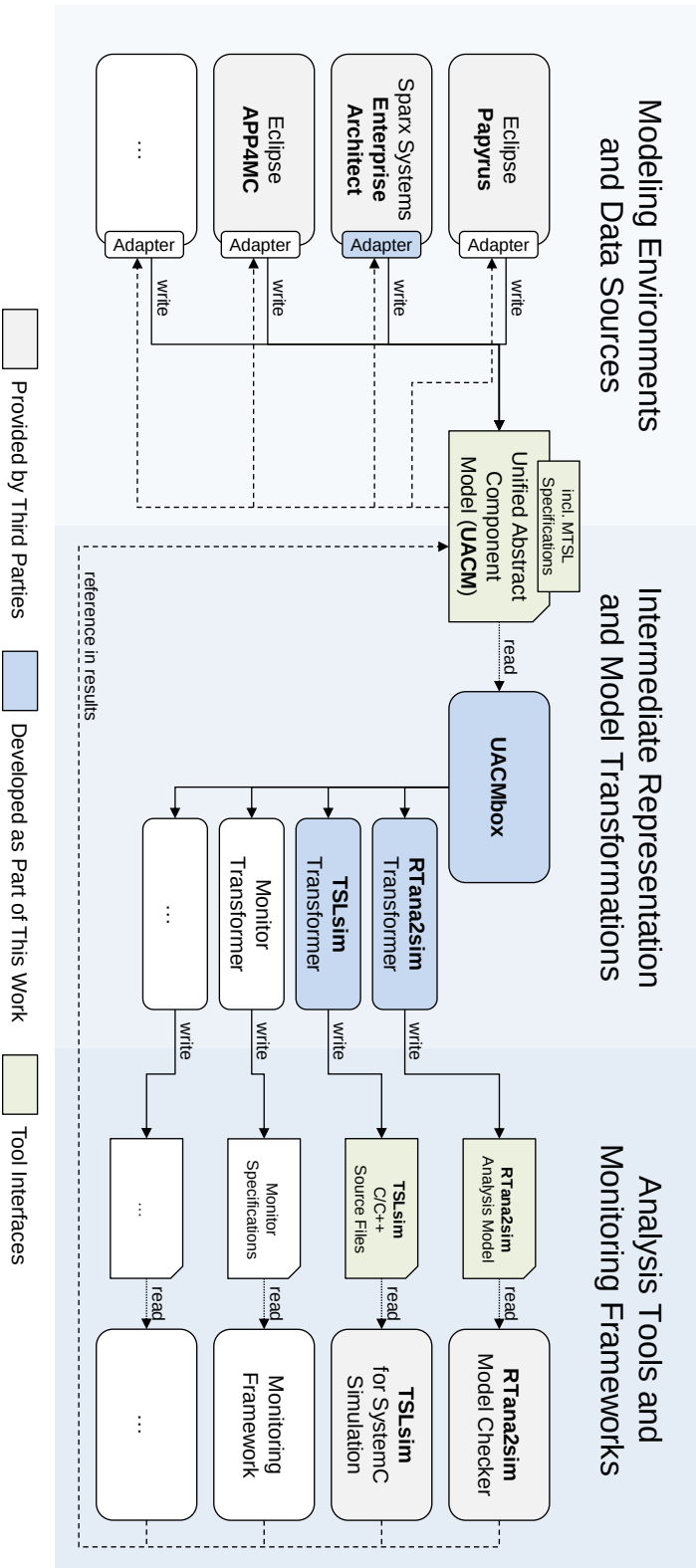


Fig. 7.1: UACMbox toolchain

tools. Moreover, it can also be used to ensure cross-organizational model exchange, e.g., as part of the *Definition of a Top-Level Architecture* design activity (see Figure 3.1).

The underlying meta-model is presented in Figure 7.2. Each model consists of a set of hierarchically nested *components* that can be equipped with any number of *ports*. A port can represent either an *input*, an *output*, or a *variable*. To provide a meaningful functionality, each component must have at least one port that belongs to one of these classes. The interaction of two or more components is represented by the definition of *signals*. *Event signals* usually connect an output port with a set of input ports. For connections between two hierarchy levels, however, event signals can also be defined between two or more input ports as well as between two or more output ports. In addition, *variable signals* can be used to connect variables. Each component can be annotated with a set of *specifications*. These can refer to the previously defined input, output, and variable ports to make statements about the desired behavior of the component.

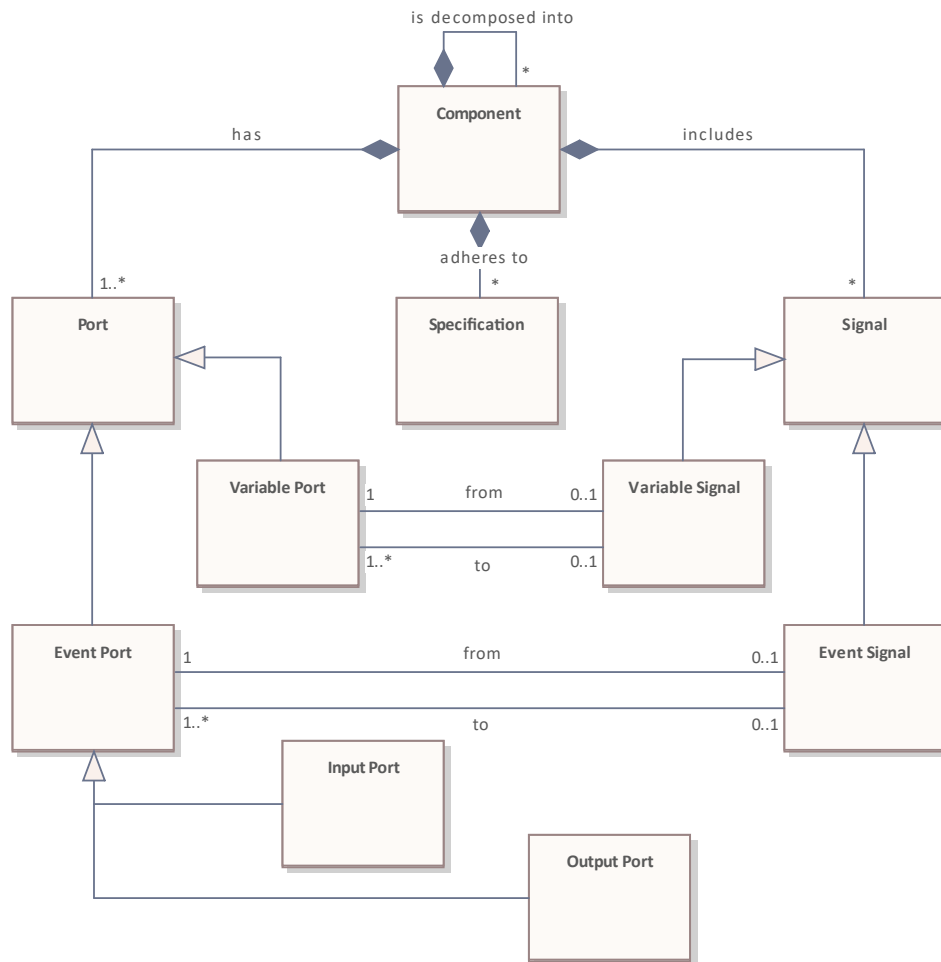


Fig. 7.2: UACM meta-model

Syntax

The UACM syntax is divided into two parts. On the one hand, the structure of UACM models is defined based on a set of syntactical elements that introduce *components*, *ports*, *specifications*, and *signals*, which are presented in Section 7.2.2a. On the other hand, the specifications annotated to UACM components can be formulated using different specification languages. As already mentioned, we focus on MTSL timing specifications in different language versions. Accordingly, we introduce the high-level MTSL syntax rules as an entry point in Section 7.2.2b. Note that, in principle, any textual specification language can be used to formulate specifications annotated to UACM components. In this case, the $\langle \textit{TimingContract} \rangle$ element used to integrate the MTSL syntax would need to be replaced by the top-level syntax rule of an alternative specification language.

Model Structure Each system is represented by a top-level component that is defined according to the $\langle \textit{Comp} \rangle$ syntax rule. Consequently, each component consists of a set of ports, a set of specifications, a set of lower-level components, and a set of signals.

$\langle \textit{Comp} \rangle$	$::= \text{'comp'} \langle \textit{CompName} \rangle \text{'.'} [\langle \textit{Port} \rangle]^* [\langle \textit{Spec} \rangle]^* [\langle \textit{Comp} \rangle]^* [\langle \textit{Signal} \rangle]^* \text{'endcomp'};$
$\langle \textit{Port} \rangle$	$::= \langle \textit{PortType} \rangle \langle \textit{PortName} \rangle [\text{':='} \langle \textit{PortValue} \rangle] \text{';'}$
$\langle \textit{PortType} \rangle$	$::= \text{'in\_port'} \mid \text{'out\_port'} \mid \text{'var\_port'}$
$\langle \textit{Spec} \rangle$	$::= \text{'spec'} [\langle \textit{SpecName} \rangle \text{'.'}] \langle \textit{TimingContract} \rangle \text{'endspec'};$
$\langle \textit{Signal} \rangle$	$::= \text{'signal'} \langle \textit{PortRef} \rangle \text{'->} \langle \textit{PortRef} \rangle [\text{' ,' } \langle \textit{PortRef} \rangle]^* \text{';'}$
$\langle \textit{PortRef} \rangle$	$::= \langle \textit{PortName} \rangle \mid \langle \textit{CompName} \rangle \text{'.'} \langle \textit{PortName} \rangle$

Note that $\langle \textit{CompName} \rangle$, $\langle \textit{PortName} \rangle$, and $\langle \textit{SpecName} \rangle$ are defined as strings without special characters. The $\langle \textit{PortValue} \rangle$ element represents either a string or an integer.

Timing Specifications The structure of the $\langle \textit{TimingContract} \rangle$ element is detailed in terms of the high-level syntax rules of the MTSL (see Section A.1). The syntax allows to define assumptions and guarantees. Both parts of the specification can be formulated using the logical values *true* or *false*, or a set of MTSL specification patterns.

$\langle \textit{TimingContract} \rangle$	$::= \langle \textit{Assumption} \rangle \langle \textit{Guarantee} \rangle$
$\langle \textit{Assumption} \rangle$	$::= \text{'A:'} \langle \textit{TimingSpec} \rangle$
$\langle \textit{Guarantee} \rangle$	$::= \text{'G:'} \langle \textit{TimingSpec} \rangle$
$\langle \textit{TimingSpec} \rangle$	$::= \text{'True.'} \mid \text{'False.'}$ $\mid \langle \textit{TimingPattern} \rangle [\langle \textit{TimingPattern} \rangle]^*$
$\langle \textit{TimingPattern} \rangle$	$::= \dots$

Note that this paragraph only contains part of the MTSL syntax (see Section A.1). As already mentioned, several MTSL versions have been developed over time:

1. A complete specification including a formal definition of the semantics of the original language version MTSL 1.0 can be found in [92, 93] (see Section A.5).
2. Later on, a revised version of the specification language was defined. This version focuses on non-causal specification patterns, simplifies the syntax, and introduces slight adaptations to the semantics (“MTSL 1.5”, see Section A.4).
3. More recent works in the context of this thesis [370, 371] have extended MTSL 1.5 to support the handling of operating modes, mode transitions, and mode-dependent timing behavior (“MTSL 1.5+Modes”, see Chapter 4 and Section A.3).
4. Subsequently, specification patterns and pattern extensions for defining and referencing local clocks from MTSL 1.0 were reintegrated (see Chapter 6), which led to the development of the current version MTSL 1.5+MC (see Section A.2).

A detailed description of the syntax and semantics of existing MTSL versions as well as a comparison of their individual features can be found in Appendix A.

File Format

In order to ensure interoperability between different modeling and analysis tools, a number of guidelines for the definition of UACM model files have been defined:

1. Each UACM model must be stored in a separate file with the `*.uacm` file extension.
2. The beginning and end of each UACM file are identified by corresponding start and end tags `[#uacm]` and `[#enduacm]`, between which the actual model description is to be placed according to the syntax defined in Section 7.2.2.
3. The MTSL version can be specified by means of *MTSL version tags* either *globally*, i.e., for any specification defined in the model, or *locally*, i.e., for an individual specification. For this purpose, one of the following tags must be included in the first line of the model and can be inserted in the first line of each specification:
 - MTSL 1.0 → `[#mtsl-1.0]`
 - MTSL 1.5 → `[#mtsl-1.5]`
 - MTSL 1.5+Modes → `[#mtsl-1.5+modes]`
 - MTSL 1.5+MC → `[#mtsl-1.5+modes+clocks]`
4. For documentation purposes, line comments can be inserted with `//...`
5. Indentations should be used to visually separate the different hierarchy levels more easily. This can be done either with tabs or spaces.

Example

Figure 7.3 shows the component model of an example system that is used as a basis for explaining the structure of UACM models. The top-level component *System* contains two lower-level components *SubSystemA* and *SubSystemB*, two input ports *FirstInput* and *SecondInput*, an output port *Output*, and a variable port *Mode*. The input and output ports are connected to the ports of the lower-level components by three signals.

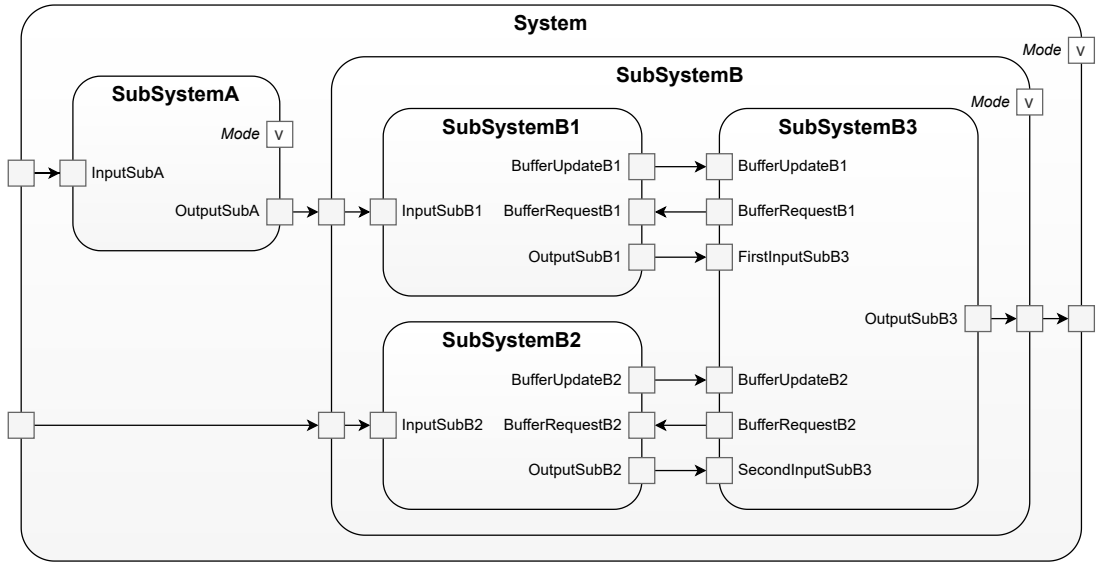


Fig. 7.3: Component model of the UACM example system

The first subcomponent *SubSystemA* has an input port *InputSubA*, an output port *OutputSubA*, and a variable port *Mode*. No other components are subordinate to it. In a real system, this component could be, for example, a preprocessing unit that performs a conversion or correction of incoming values. *SubSystemB* has two input ports *FirstInputSubB* and *SecondInputSubB*, an output port *OutputSubB*, and another variable port named *Mode*. It is decomposed into three subcomponents, namely *SubSystemB1*, *SubSystemB2*, and *SubSystemB3*. All three components have multiple input and output ports that are connected by a set of signals. For example, *SubSystemB1* and *SubSystemB2* could represent two buffers that temporarily store incoming values and pass them on request to the *SubSystemB3* component, which provides the actual functionality.

Listing 7.1 shows the UACM representation of the example system. The model hierarchy is represented by a set of components, whose representation is supported by the indentation of the corresponding lines. Each component contains only signals that connect its own input and output ports to those of the components on the next lower hierarchy level. MTSL 1.5 was selected as the global specification language (see line 2). However, the specification of the top-level component was formulated in MTSL 1.0 (see line 11). Note

that initial values for all variable ports, an example timing specification for the top-level component, and some comments have been added to the original system description.

```

1  [#uacm]
2  [#mtsl-1.5]
3  comp System:
4      in_port FirstInput;
5      in_port SecondInput;
6      out_port Output;
7
8      var_port Mode := Idle;
9
10     spec C_tl:
11         [#mtsl-1.0]
12         // Specification of 'System'
13         A: FirstInput occurs every 33 ms with jitter 5 ms.
14           SecondInput occurs every 50 ms.
15         G: Reaction(FirstInput,Output) within (0,300] ms.
16     endspec;
17
18     comp SubSystemA:
19         in_port InputSubA;
20         out_port OutputSubA;
21
22         var_port Mode := Idle;
23
24         spec C_SubA:
25             // Specification of 'SubSystemA'
26             // ...
27         endspec;
28     endcomp;
29
30     comp SubSystemB:
31         in_port FirstInputSubB;
32         in_port SecondInputSubB;
33         out_port OutputSubB;
34
35         var_port Mode := Idle;
36
37         spec C_SubB:
38             // Specification of 'SubSystemB'
39             // ...
40         endspec;
41
42     comp SubSubSystemB1:
43         in_port InputSubB1;
44         in_port BufferRequestB1;
45         out_port BufferUpdateB1;
46         out_port OutputSubB1;
47
48         spec C_SubSubB1:
49             // Specification of 'SubSubSystemB1'
50             // ...

```

```

51         endspec;
52     endcomp;
53
54     comp SubSubSystemB2:
55         in_port InputSubB2;
56         in_port BufferRequestB2;
57         out_port BufferUpdateB2;
58         out_port OutputSubB2;
59
60         spec C_SubSubB2:
61             // Specification of 'SubSubSystemB2'
62             // ...
63         endspec;
64     endcomp;
65
66     comp SubSubSystemB3:
67         in_port FirstInputSubB3;
68         in_port SecondInputSubB3;
69         in_port BufferUpdateB1;
70         in_port BufferUpdateB2;
71         out_port BufferRequestB1;
72         out_port BufferRequestB2;
73         out_port OutputSubB3;
74
75         spec C_SubSubB3:
76             // Specification of 'SubSubSystemB2'
77             // ...
78         endspec;
79     endcomp;
80
81     // Input ports of 'SubSystemB'
82     signal FirstInputSubB -> SubSubSystemB1.InputSubB1;
83     signal SecondInputSubB -> SubSubSystemB2.InputSubB2;
84
85     // Output ports of 'SubSubSystemB1'
86     signal SubSubSystemB1.BufferUpdateB1 ->
87     ↪ SubSubSystemB3.BufferUpdateB1;
88     signal SubSubSystemB1.OutputSubB1 ->
89     ↪ SubSubSystemB3.FirstInputSubB3;
90
91     // Output ports of 'SubSubSystemB2'
92     signal SubSubSystemB2.BufferUpdateB2 ->
93     ↪ SubSubSystemB3.BufferUpdateB2;
94     signal SubSubSystemB2.OutputSubB2 ->
95     ↪ SubSubSystemB3.SecondInputSubB3;
96
97     // Output ports of 'SubSubSystemB3'
98     signal SubSubSystemB3.BufferRequestB1 ->
99     ↪ SubSubSystemB1.BufferRequestB1;
100    signal SubSubSystemB3.BufferRequestB2 ->
101    ↪ SubSubSystemB2.BufferRequestB2;
102    signal SubSubSystemB3.OutputSubB3 -> OutputSubB;
103 endcomp;

```

```

98
99 // Input ports of 'System'
100 signal FirstInput -> SubSystemA.InputSubA;
101 signal SecondInput -> SubSystemB.SecondInputSubB;
102
103 // Output ports of 'SubSystemA'
104 signal SubSystemA.OutputSubA -> SubSystemB.FirstInputSubB;
105
106 // Output ports of 'SubSystemB'
107 signal SubSystemB.OutputSubB -> Output;
108 endcomp;
109 [#enduacm]

```

Lst. 7.1: UACM representation of the example system

7.2.3 Analysis Tools and Monitoring Frameworks

The current version of the *Analysis Tools and Monitoring Frameworks* stage comprises two analysis tools that have been integrated into the UACMbox Toolchain. In a first step, the existing TSLsim developed in MULTIC-Tooling [93, 164] was selected for reuse. It supports simulation-based VITs of MTSL 1.0 specifications. In a second step, the discrete-time model checker RTana_{2sim} was included to allow formal verification of refinement relations. In the context of this work, RTana_{2sim} is used to perform model-checking-based VITs of MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC specifications. Note that the current version focuses on the automated execution of VITs of mode-independent and mode-dependent specifications. Due to RTana_{2sim}'s discrete time model, verification of specifications relating to local clocks (see Chapter 6) is currently not supported. Details on the implementation and practical use of RTana_{2sim} can be found in Section 7.3.3.

Possible extensions in future activities include the integration of specialized analysis tools such as OCRA [147] and general-purpose verification engines such as nuXmv [129] or SPIN [295]. In addition, monitoring frameworks such as tool suites for generating bare-metal C++ observers, observers for HIL platforms such as Scalexio [628], and *Robot Operating System* (ROS)-based monitoring applications [78] could be integrated to enable the use of MTSL timing specifications for distributed run-time monitoring.

7.3 Tool Architectures and Components

The UACMbox Toolchain is composed of a set of individual tools that have been combined to realize an end-to-end solution for performing real-time analysis. Many of the tools, transformations, and models used were developed from scratch or have been adapted and integrated based on the results of previous research projects.

In the current version, the individual tool components are not yet fully integrated from a control flow perspective, i.e., the tools must be executed manually one after the other. However, the interfaces are already prepared for a deeper integration, so that only the appropriate calls would need to be added to complete this step. The data exchange

between the different stages is based on reading/writing model files, which is mainly for development and debugging purposes. In future versions, the interaction could, for example, also be realized based on data stream between the individual tools.

The decoupling of modeling and analysis tools through the introduction of the UACM intermediate format allows the independent development of toolchain components, e.g., using different programming languages, technologies, and engineering processes. At the same time, the use of the three-stage toolchain structure implies a clear assignment of responsibilities. This means that model transformation adapters for modeling tools only need to be able to generate consistent UACM models, but do not need to consider the specifics of certain analysis tools. On the other hand, the analysis tools only need to consume their analysis models, so that the second stage—and, in particular, the UACMbox as the heart of the toolchain—becomes the “single point of implementation” for most MTSL-related tasks. This includes the realization of a UACM parser, a hierarchical multi-version MTSL parser, corresponding data structures, and modules for the construction of semantic-preserving analysis models based on the previously read UACM models.

7.3.1 MULTIC-Tooling Industrial

The *MULTIC-Tooling Industrial* (MTI) add-in for Enterprise Architect [549] is the first building block for the reimplementation of the MULTIC-Tooling tool prototype based on an industry-relevant modeling environment. It supports the annotation of MTSL timing specifications in arbitrary language versions to SysML models and allows the validation of such kind of models. In addition, it is capable to automatically transform them into corresponding UACM representations that can be read by subsequent components. The add-in can also be configured to trigger the execution of external analysis tools.

In the context of this work, the MTI add-in was developed from scratch based on the architecture and design rules of the MULTIC-Tooling tool prototype [93, 164]. The work steps included the design, implementation, deployment, test, and integration of all tool components. The implementation was done in C# using Microsoft Visual Studio 2019. MTI can be used with Enterprise Architect 15, 16, and 17 and has dependencies to the .NET framework run-time environment as well as the latest version of the Enterprise Architect Software Developers Kit (SDK) [548]. Regular intermediate releases and updates were evaluated by colleagues and external project partners. The model transformation adapter also comes with a test suite consisting of example Enterprise Architect models that include functional architectures modeled in SysML with timing specifications in different MTSL versions, i.e., MTSL 1.0, MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC.

As a first enhancement, a research assistant has extended the add-in to allow the transformation of component models designed in safeTbox [231], a toolbox for the development and certification of safety-critical systems developed by Fraunhofer IESE, into corresponding UACM models. The extensions allow this type of model to be edited and transformed with the same user interface and output behavior as SysML models.

The tool architecture and the interfaces of the MTI add-in are presented in Figure 7.4. As soon as Enterprise Architect (EA) is started, the modeling tool initiates all previously registered add-ins via the *EA Add-In Registry*. For the MTI add-in, the

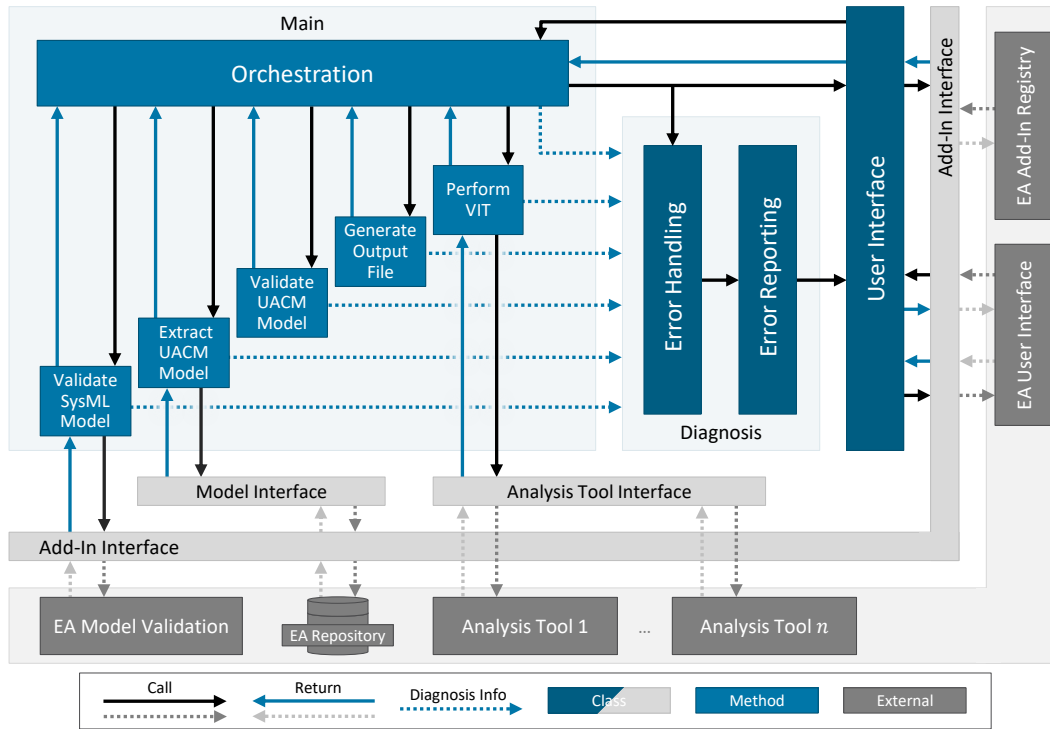


Fig. 7.4: Architecture and interfaces of the MTI add-in (based on [93, p. 69])

MTI_AddInInterface class, which encapsulates all internal implementations, is set as the entry point. In the startup phase, the **MTI_AddInInterface** organizes the construction of all add-in components and establishes the link between the MTI control panels, defined as part of the **MTI_UserInterface**, and the *EA User Interface*. After the successful initialization, menu items for using the add-in are displayed as part of the *EA User Interface*, as shown in Figure 7.5. The extensions include contributions to the ribbon menu (see Figure 7.5a) and the context menu (see Figure 7.5b), whereby the latter enables the execution of add-in functions for previously selected modeling elements. Each of the menu items represents one of the main functions of the model transformation adapter:

Validate Model The *Model Validation* function allows to validate SysML models. The validation procedure includes methods for checking the compliance with SysML syntax rules and ensuring the consistency of resulting UACM models. Note that checking the syntax of MTSL specifications is not part of the validation process.

Perform Virtual Integration Test The *Virtual Integration Test* function executes VITs for selected top-level elements, taking an optional *descent depth* into account that specifies how many hierarchical levels should be considered. The analysis task can be performed using different tools, which can be selected prior to execution.

Edit Specifications The *Specification Editor* function enables efficient formulation of MTSL timing specifications in different language versions. For this purpose, the add-in provides a simple editor that supports MTSL syntax highlighting and allows the specification of timing properties of previously modeled systems.

Configure The *Configuration* function allows to configure the MTI add-in. For example, the *global specification language* relevant for the generation of UACM output files can be defined. In addition, it is possible to specify the paths to the executables of the UACMbox and the analysis tools as well as different formatting options.

Note that each function provides additional extensions to the graphical user interface in the form of individual *views* that are displayed after clicking the corresponding menu items. These views allow engineers to set additional options and/or execute validation or analysis tasks. The two most important views, the *Specification Editor* view (see Figure 7.6a) and the *Virtual Integration Test* view (see Figure 7.6b), are shown in Figure 7.6. More details about the structure and the functionality of the two views are provided below.

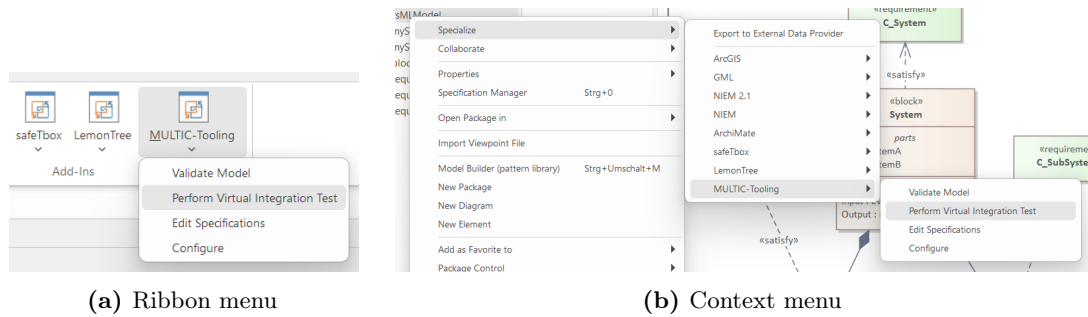


Fig. 7.5: Extensions to the graphical user interface of Enterprise Architect

The practical use of the MTI add-in is demonstrated using a simple example system referred to as *Tiny SysML Model*, which has been defined in [93]. The component model of the example system is shown in Figure 7.7. It consists of a top-level *System* component, which is decomposed into two lower-level components *SubSystemA* and *SubSystemB*. The lower-level components are connected in series, i.e., the top-level *Input* port is connected to the *InputSubA* port, the *OutputSubA* port to the *InputSubB* port, and the *OutputSubB* port to the top-level *Output* port. The corresponding MTSL 1.5 timing specifications are listed in Tables 7.1 to 7.3 and include **Repetition** and **Reaction** patterns.

In a first step, the *Tiny SysML Model* is modeled in terms of a SysML model. For this purpose, we rely on three types of SysML diagrams based on the approach presented in [93, 164]. The diagram types include *Block Definition Diagrams* (BDDs) for modeling the component hierarchy, *Internal Block Diagrams* (IBDs) for modeling the component interfaces, i.e., event and variable ports, as well as the connections between these interfaces, and one or more *Requirement Diagrams* (REQs) for specifying timing requirements.

Excerpts from the SysML model are shown in Figure 7.8. All components from Figure 7.7 are represented by corresponding SysML *blocks*. Based on this, Figure 7.8a

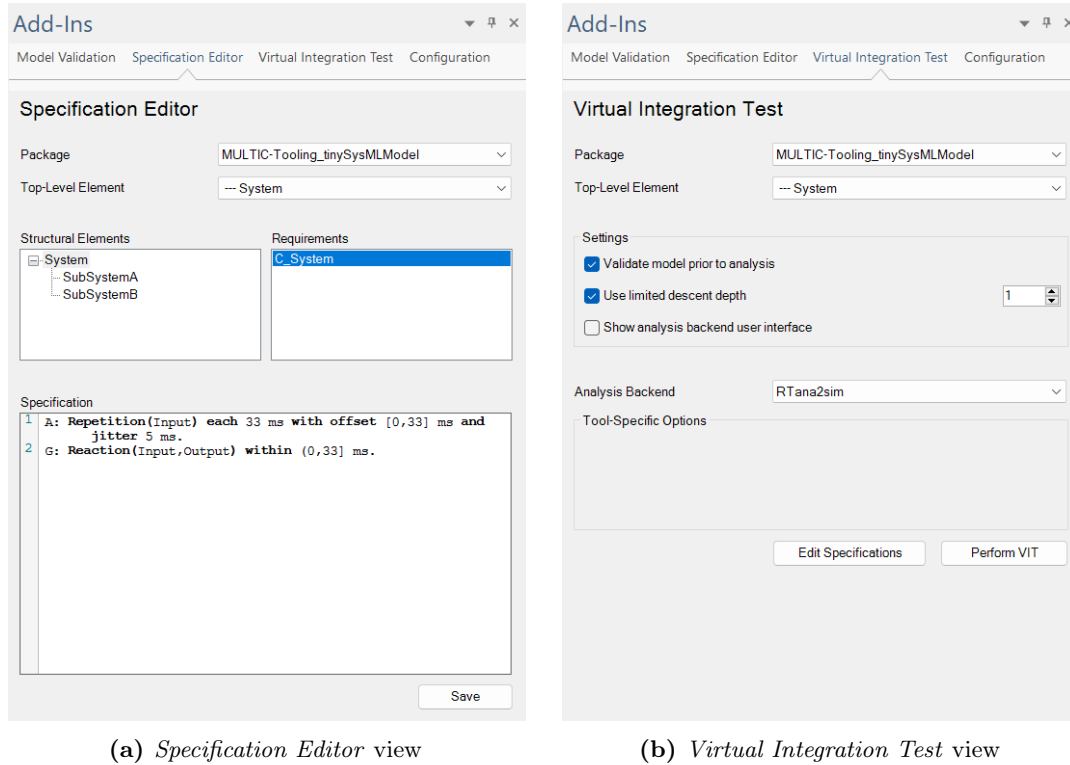


Fig. 7.6: Graphical user interface of the MTI add-in

shows the representation of event and variable ports by *proxy ports* and the modeling of signals in terms of *binding connectors*. Additionally, the *requirement blocks* that contain the MTSL timing specifications as well as their «*satisfy*» links are depicted in Figure 7.8b. Note that the BDD, which defines blocks and their hierarchy in terms of *part association* relationships, since all relevant blocks and relations are also visible in Figure 7.8b.

After creating the SysML model, the *Model Validation* function can be used to validate the resulting system model. For this purpose, engineers first open the *Model Validation* view and click on *Validate Model*. The MTI add-in then automatically executes all relevant process steps. Initially, the `ValidateSysMLModel()` method is called, which accesses the *EA Model Validation* via the `MTI_AddInInterface` (see Figure 7.4) and applies a set of predefined SysML validation rules provided by EA. After transforming the SysML model into an in-memory UACM representation using the `ExtractUACMModel()` method, the `ValidateUACMModel()` method is called to validate the resulting UACM model.

Before or after validation, the *Specification Editor* function can be used to add MTSL timing specifications to the requirement blocks defined as part of the SysML model, or adapt previously added specifications. The simple editor provided as part of the *Specification Editor* view (see Figure 7.6a) helps engineers in formulating such specifications. Note that the specifications must comply with the syntax rules defined in Appendix A.

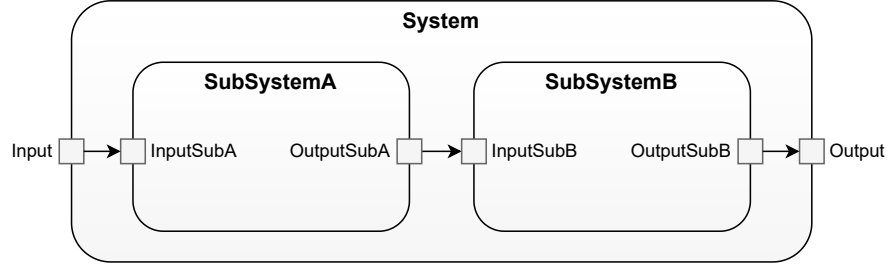


Fig. 7.7: Component model of the *Tiny SysML Model*

Tab. 7.1: Top-level specification of the *Tiny SysML Model*

A	<i>Input</i> occurs every 33 ms with offset [0, 33] ms and jitter 5 ms. 1
G	$\text{Reaction}(\text{Input}, \text{Output})$ within (0, 33] ms. 2

Tab. 7.2: Specification $\mathcal{C}_{suba}$ of the *SubSystemA* component

A	<i>InputSubA</i> occurs every 33 ms with offset [0, 33] ms and jitter 5 ms. 1
G	$\text{Reaction}(\text{InputSubA}, \text{OutputSubA})$ within [10, 20] ms. 2

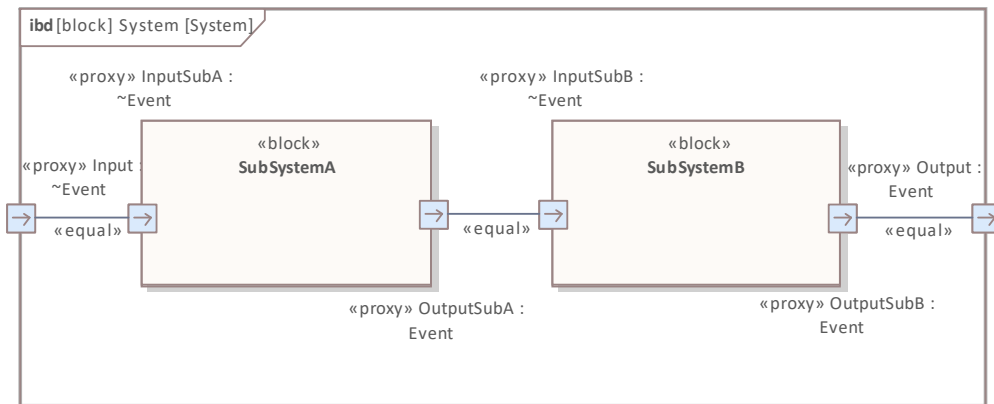
Tab. 7.3: Specification $\mathcal{C}_{subb}$ of the *SubSystemB* component

A	<i>InputSubB</i> occurs every 33 ms with offset [10, 43] ms and jitter 15 ms. 1
G	$\text{Reaction}(\text{InputSubB}, \text{OutputSubB})$ within [5, 10] ms. 2

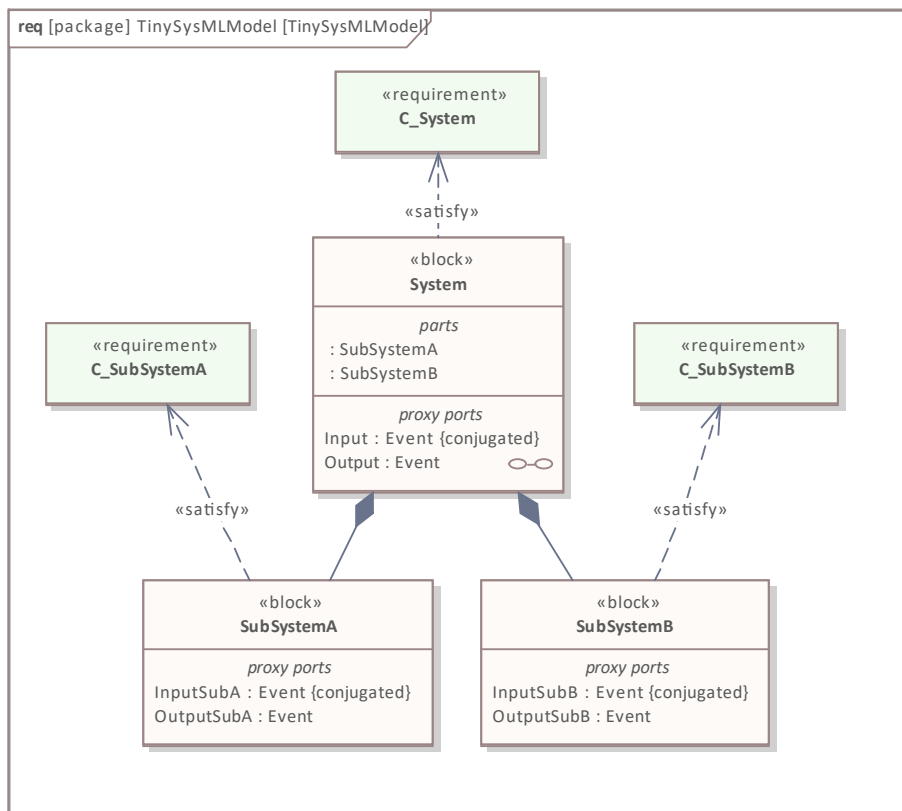
In the context of our example, all specifications $\mathcal{C}_{sys}$, $\mathcal{C}_{suba}$, and $\mathcal{C}_{subb}$ are transferred to the corresponding requirement blocks **C_System**, **C_SubSystemA**, and **C_SubSystemB**.

The *Virtual Integration Test* function allows to verify refinement relations between the top-level and lower-level specifications. In a first step, engineers open the corresponding *Virtual Integration Test* view (see Figure 7.6b) by selecting the *Perform Virtual Integration Test* item from the ribbon or context menu. After specifying the top-level element, the analysis procedure is triggered by clicking on the *Perform VIT* button. The corresponding analysis task executes a VIT for a selected top-level element, taking into account the optional *descent depth* that specifies how many hierarchical levels should be considered. In Figure 7.6b, for example, a limited descent depth of 1 is set, i.e., the top-level component and all lower-level components on the next lower hierarchy level are included in the UACM model. Note that the option to explicitly set the descent depth is a simple but effective measure to reduce the complexity of the resulting analysis models.

Prior to the actual VIT execution, the process steps for model validation are executed again. Thus, the `ValidateSysMLModel()`, `ExtractUACMModel()`, and `ValidateUACMModel()` methods are called one after the other (see Figure 7.4). The resulting in-memory UACM representation is provided in the form of an instance of the `MTI_SystemModel` class, which serves as the basis for generating UACM output files. The `GenerateOutputFile()` method serializes the system model into a string representation by recursively calling



(a) Internal block diagram



(b) Requirement diagram

Fig. 7.8: SysML model of the *Tiny SysML Model*

`ToString()` methods of all UACM elements. The model string is then written to an output file with the `*.uacm` file extension. With respect to our example, Listing 7.2 presents the resulting UACM model for the *Tiny SysML Model* example. After the generation of the UACM output file, the `MTI_AnalysisToolInterface` is responsible for executing the UACMbox (see Section 7.3.2) to transform the UACM model into an analysis model compatible with the previously selected analysis tool (see Figure 7.6b) such as `RTana2sim` (see Section 7.3.3), and triggers the analysis execution. Finally, the results are returned to the MTI add-in and are displayed to the engineers via the `MTI_UserInterface`. The engineers can then use the results to adapt the system model or the specifications.

```

1  [#uacm]
2  [#mtsl-1.5]
3  comp System:
4      in_port Input;
5      out_port Output;
6
7      spec C_System:
8          // Specification of 'System'
9          A: Repetition(Input) each 33 ms with offset [0,33] ms and
↪          jitter 5 ms.
10         G: Reaction(Input,Output) within (0,33] ms.
11     endspec;
12
13     comp SubSystemA:
14         in_port InputSubA;
15         out_port OutputSubA;
16
17         spec C_SubSystemA:
18             // Specification of 'SubSystemA'
19             A: Repetition(InputSubA) each 33 ms with offset [0,33] ms
↪             and jitter 5 ms.
20             G: Reaction(InputSubA,OutputSubA) within [10,20] ms.
21         endspec;
22     endcomp;
23
24     comp SubSystemB:
25         in_port InputSubB;
26         out_port OutputSubB;
27
28         spec C_SubSystemB:
29             // Specification of 'SubSystemB'
30             A: Repetition(InputSubB) each 33 ms with offset [10,43]
↪             ms and jitter 15 ms.
31             G: Reaction(InputSubB,OutputSubB) within [5,10] ms.
32         endspec;
33     endcomp;
34
35     // Input ports of 'System'
36     signal System.Input -> SubSystemA.InputSubA;
37
38     // Output ports of 'SubSystemA'

```

```

39     signal SubSystemA.OutputSubA -> SubSystemB.InputSubB;
40
41     // Output ports of 'SubSystemB'
42     signal SubSystemB.OutputSubB -> System.Output;
43 endcomp;
44 [#enduacm]

```

Lst. 7.2: UACM model of the *Tiny SysML Model*

To support the main functions, the MTI add-in has its own module for diagnosing errors that may occur in the individual process steps. The structure of this module is also based on the architecture of the MULTIC-Tooling tool prototype and provides the same functionality. All errors that occur during the execution of validation or analysis tasks are collected in an *error list* that is updated by the `MTI_ErrorHandling` class (see Figure 7.4). The `MTI_ErrorReporting` then outputs information about identified problems or required improvements via the `MTI_UserInterface`, which accesses the *EA User Interface*.

The MTI add-in has already reached a stable state and is currently being used in several research projects. The model transformation adapter can be employed for model validation, the formulation and editing of MTSL timing specifications as well as the reliable transformation of SysML models with annotated specifications into consistent UACM representations. In the future, the add-in could be improved to better support engineers in formulating real-time requirements. For example, an assistance function could be added that automatically calculates offset intervals of **Repetition** patterns based on the timing specifications of upstream components. At the same time, ongoing improvements in model validation and error handling are expected to yield more precise recommendations for enhancing model quality and consistency. Another key point is the planned integration with other components of the UACMbox Toolchain, such as with the UACMbox and *RTana_{2sim}*. Due to the clear separation between UACM- and EA-specific implementations by means of the `MTI_AddInInterface`, the add-in implementation could also serve as a basis for developing adapters for other modeling environments.

With regard to the design approach presented in Chapter 3, it would also be desirable to add a new main function that enables the automated translation of SysML activity diagrams with timing constraints into SysML models with MTSL specifications for each involved CS. The necessary extensions to include this functionality have already been prepared and can be integrated without affecting other toolchain components.

7.3.2 UACM Toolbox

The *UACM Toolbox* (UACMbox) allows to transform UACM models into different output formats. The tool is maintained as a core component of the UACMbox Toolchain and serves as an intermediate representation and exchange format for decoupling modeling environments and analysis tools. It can be used as a standalone transformation toolbox via command line interfaces and is also prepared for use as a library, e.g., to integrated it into other tools to directly access UACM models. The main function of the UACMbox is to transform UACM input models into analysis models for different analysis tools.

In the context of this work, the UACMbox was developed from scratch, taking into account an early version of the RTana2sim parser as a source of inspiration. The work steps included the iterative extension of the parser to fully support MTSL 1.0, MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC. In a second step, the *MTSL Toolbox* (MTSLbox) was build around the hierarchical, multi-version MTSL parser. It provides a data model for the efficient handling of all MTSL specification patterns. Afterwards, the same parser structure and similar data types were used to develop the UACM parser as well as a corresponding data model to handle UACM descriptions. The implementation of the UACMbox was done in C++. Due to the planned release as open source software, we made sure from the outset to have as few dependencies on external tools and libraries as possible. As a result, the current version of the UACMbox has no external dependencies besides the obligatory C++ standard library. The toolbox is complemented by a test suite consisting of example UACM models annotated with timing specifications in different MTSL versions, i.e., MTSL 1.0, MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC.

The tool architecture of the UACMbox is shown in Figure 7.9. When invoking the UACMbox, the path to the UACM input model, the desired output format, and the target directory must be specified as parameters. After the start of the UACMbox, the UACM input file is first read into a `uacmText` variable, which is then preprocessed. In this step, all comments as well as the UACM start and end tags are removed from the string representation. At the same time, the global specification language version is read and stored temporarily for later use. The *UACM Parser* is then called with

```
UACMModel * uacmModel = new UACMModel(uacmText, mtslVersion);
```

to extract all relevant information from the UACM input model. Following the generation of an abstract syntax tree, the tree is systematically evaluated to transfer the hierarchical model structure and the annotated MTSL specifications into the instance of the `UACMModel` class. According to the UACM meta-model introduced in Section 7.2.2, each UACM model consists of a top-level component whose internal structure defines the structure of the model. Recall that each component has a set of ports, a set of specifications, a set of lower-level components, and a set of signals. Based on this principle, all modeling elements are converted into suitable data objects that fully reflect the UACM input model.

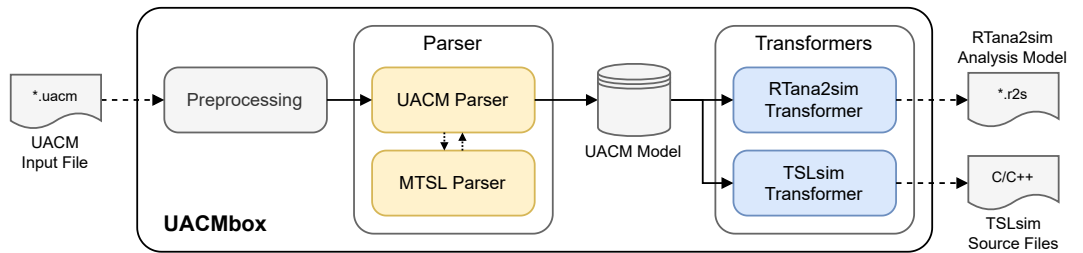


Fig. 7.9: Architecture and interfaces of the UACMbox

Parsing the specifications is a special feature here. The specification texts contained in the specification blocks are extracted as a whole and transferred to the *MTSL Parser*, which is realized as an instance `mtslParser` of the `MTSLParser` class. The method call

```
mtslParser.parseContract(contractText, specName, specContract);
```

establishes the connection between the two parsers. The *MTSL Parser* is responsible for parsing the `contractText` of the specification `specName` and returns the resulting `MTContract` object using the call-by-reference parameter `specContract`. In this context, the `parseContract(...)` method serves as the entry point of the hierarchical, multi-version MTSL parser. Initially, the parser checks whether the `contractText` contains a specification language tag that specifies a different local specification language version than the global specification language tag. If so, the local specification language version is temporarily stored, and the tag is removed from the input text. The `contractText` is then split into a list of specification lines, each of which is passed to the MTSL (sub-)parser that supports the previously selected MTSL version. Note that the local specification version only overrides the global specification version for the current UACM specification. Additionally, only one specification language tag per specification is supported.

Each MTSL (sub-)parser analyzes the specification lines according to the same scheme. In all cases, the specification line is converted into an abstract syntax tree based on the syntax rules from Appendix A, which is then traversed recursively. The extracted pattern parameters are stored in union-like `MTPParameter` objects that support all relevant data types. Finally, the pattern parameters extracted are used to construct concrete instances of the `MTAbstractPattern` class, which are added to either the assumption or the guarantee of the `specContract` object, e.g., via the following method call:

```
specContract.addGuaranteePattern(currentPattern);
```

Note that several pattern classes exist for each MTSL specification pattern. These classes represent the individual pattern variants that results from the use of different MTSL versions. For example, the `Repetition` pattern is implemented by the `MTRepetitionPatternV1`, `MTRepetitionPatternV1_5`, `MTRepetitionPatternV1_5_Modes`, and `MTRepetitionPatternV1_5_Modes_Clocks` classes. Note that all pattern variant classes share the same parameter implementations and have identifiable pattern types, which can be used to cast specification patterns between different language versions. After successfully parsing the MTSL specification, the recursive evaluation of the UACM syntax tree is continued until all nodes have been converted into corresponding data objects.

The resulting in-memory `UACMModel` can be used to generate analysis models for different analysis tools. To this end, it provides methods for retrieving information about the model structure and the annotated timing specifications. As previously mentioned, the current version of the UACMbox supports the generation of analysis models for `RTana2sim` and C/C++ source files for `TSLsim`. For this purpose, a `GenericTransformer` has been implemented as a base class for transformer realizations. It provides basic methods for handling target directory paths and global specification language versions and serves as a starting point for defining model transformers. Further details on the implementation of

the existing transformers can be found in the following subsections. After writing the analysis models to corresponding output files, the UACMbox terminates.

RTana_{2sim} Transformer

In the current version of the UACMbox Toolchain, RTana_{2sim} is the main analysis tool used for performing VITs. As part of this work, the analysis tool has been integrated by implementing an *RTana_{2sim} Transformer* as a contribution to the UACMbox. The transformer is based on the **GenericTransformer** class and provides methods for translating UACM models with annotated MTSL timing specifications into RTana_{2sim} analysis models that enable the execution of model-checking-based VITs. The latest version of the *RTana_{2sim} Transformer* supports MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC.

Each RTana_{2sim} analysis model consists of a set of *ports*, a set of *port mappings*, a set of *signals*, a set of *event sources*, a set of *tasks* that are allocated to *schedulers* or a set of *components*, a set of *observers*, and a set of *properties* to be checked. Listing 7.3 shows an example of such a model, which will be explained in detail later. The basic idea is that event sources generate events on ports according to predefined patterns, which are then propagated along port mappings and signals to tasks and/or components. All tasks allocated to a shared scheduler are executed according to a previously defined scheduling policy. Components represent a special type of such tasks that are allocated to a dedicated scheduler, meaning they can start execution immediately without delays or preemption.

Both tasks and components are defined in terms of *execution automata*, each of which has an internal state composed of an *execution state*, e.g., *Idle* (I), *Running* (R), or *Preempted* (R), and a *mode* (e.g., a value 0, . . . , 7). Depending on its internal state, each automaton is sensitive to events that occur on a set of ports defined by triggers

trigger in ($\langle state \rangle, \langle mode \rangle$) on $\langle ports \rangle$:

(see, for example, Listing 7.3, line 20). The reactions of a given execution automaton to the trigger events are specified in terms of lists of actions that are selected based on a case distinction over the read event values. For this purpose, a list of possible cases

case [$\langle port \rangle ? \langle value \rangle$]*:

is specified for each **trigger** (see Listing 7.3, lines 21/26). Each **case** contains a list of actions that are executed sequentially in case the condition is fulfilled. The most common action is writing an output event with a specified delay, which can be defined using the

after $\langle interval \rangle$ **write** [$\langle port \rangle ! \langle value \rangle$]*;

statement (see Listing 7.3, lines 22/27). The final part of each **trigger** is a **goto** statement

goto ($\langle state \rangle, \langle mode \rangle$);

that defines the subsequent state to which the automaton switches after all actions have been successfully executed (see, for example, Listing 7.3, lines 23/28).

In addition to tasks and components, the RTana_{2sim} system model includes *observers* as another type of execution automata. They can be used to observe the behavior of tasks and components and check whether the resulting event traces comply with the expected behavior encoded in the observer definition. Each observer consists of a set of *modes*, each of which has an *invariant* and a set of *transitions*, which allow switching to other modes (see Listing 7.3, lines 60–67). In the context of this work, we rely on a subset of the observer syntax and use the following statements to define modes and transitions:

```
mode <ID> [ [<invariant>] ]?
-- <port> [<interval>] -> <ID>;
```

Additionally, each observer has an implicitly defined **BAD** mode, to which it automatically switches if (1) no suitable transition is available or (2) the temporal invariant of the current mode is violated. The **BAD** mode of observers can be used, for example, to specify required behavior in terms of *properties*. To define a *property*, the statement

```
property <observer>.mode != BAD;
```

can be used (see, for example, Listing 7.3, lines 90–92). Note that more elaborated rules for defining RTana_{2sim} observers exist, which are beyond the scope of this work.

In order to employ RTana_{2sim} for performing VITs, the underlying analysis problem must be encoded using the elements of the RTana_{2sim} system model. Recall that the goal of a VIT is to verify contract refinement, i.e., to check whether the condition

$$\mathcal{C}_{tl} \succ \mathcal{C}_{sub_1} \otimes \dots \otimes \mathcal{C}_{sub_n} \quad (7.1)$$

with $n \in \mathbb{N}$ is fulfilled. In Equation 2.3, we have already presented two practical conditions that can be used to implement corresponding analysis methods. In case of the RTana_{2sim} transformer, the general idea is to transform UACM input models annotated with MTSL timing specifications into a set of pattern generators in terms of *components* and a set of pattern observers in terms of *observers*. The individual automata are then combined by following a similar approach than the one used in the MULTIC-Tooling prototype (see Section 2.3.3), resulting in an RTana_{2sim} analysis model that realizes the VIT.

The procedure for generating the analysis model is as follows. After the transformer is called by the UACMbox, the previously read UACM model is validated based on a set of analysis tool-specific validation rules. For the RTana_{2sim} transformer, this step includes basic checks to ensure that a consistent analysis model can be created. The corresponding method validates, for example, that all components are assigned a specification and that all input and output ports referenced in the specifications actually exist. It also ensures that components with mode-dependent specifications have a dedicated *Mode* port, and that variable ports are only referenced as part of *set()* and *change()* events.

The next step is to create an empty RTana_{2sim} model, which serves as the basis for constructing the analysis model. Throughout the process, the UACM model is used to extract all relevant parameters. The following transformation steps must be completed:

1. Add a *top-level assumption generator* for each specification pattern of the top-level assumption. The generators are responsible for generating valid input event traces that together fulfill the top-level assumption, i.e., A_{tl} from Equation 2.3a.

7 Tool Support

2. Add a *top-level guarantee observer* for each specification pattern of the top-level guarantee. The observers are responsible for checking whether the output event traces comply with the top-level guarantee, i.e., G_{tl} from Equation 2.3b.
3. Optional: Add a *lower-level assumption observer* for each specification pattern of lower-level assumptions (except for assumptions of leaf components¹).
4. Optional: Add a *lower-level guarantee observer* for each specification pattern of lower-level guarantees (except for guarantees of leaf components).
5. Add a *leaf assumption observer* for each specification pattern of assumptions of leaf components. The observers are responsible for checking whether the input event traces comply with the leaf assumptions, i.e., $A_{sub_1}, \dots, A_{sub_n}$ from Equation 2.3.
6. Add a *leaf guarantee generator* for each specification pattern of guarantees of leaf components. The generators are responsible for generating valid output event traces that fulfill the leaf guarantees, i.e., $G_{sub_1}, \dots, G_{sub_n}$ from Equation 2.3.
7. Add a *mode automaton* for each component with mode-dependent specifications that is responsible for implementing mode transitions, generating $change(Mode, \langle value \rangle)$ events based on $set(Mode, \langle value \rangle)$ events, and passing mode changes of lower-level to higher-level components based on the specified mode mappings.
8. Add all relevant port mappings and signals to the analysis model.

Pattern generators for input/output ports whose behavior is constrained by multiple assumption/guarantee patterns must be composed in order to obtain valid analysis models. All generator components have access to a shared **Analysis.Trigger** variable port that is initially assigned with an event and serves as a “global clock” to ensure progress. Since the event is not removed from the port, all components can reference it in **trigger** and **case** statements to make sure that their actions are executed in the next time step.

In order to handle complex event specifications, such as event sequences, special components denoted as *event specification acceptors* are created for each input event specification, and *event specification generators* for each output event specification. While the former are responsible for accepting the event specification and creating a virtual event on a dedicated **EventSpecTarget** port that can be referenced by the pattern generators and observers, the latter are triggered by an event on the **EventSpecTrigger** port and generate event sequences that comply with the given specification. Note that both single events and event sets are handled individually by the pattern generators and observers.

For UACM models with mode-dependent specifications, each *Mode* port is mapped to a $\langle component \rangle.Mode$ variable port. In addition, $\langle component \rangle.SetMode$ and $\langle component \rangle.ChangeMode$ ports at which $set()$ and $change()$ events occur are instantiated for each mode-dependent component. An additional $\langle component \rangle\_ChangeEventSink$ component ensures that top-level $change()$ events are consumed to prevent buffer overflows. All pattern generators and observers are provided with internal modes that allow the components

¹The term *leaf components* refers to components at the lowest hierarchy level.

to be enabled and disabled depending on the mode condition of the specification pattern. In order to be able to react to mode changes as part of **trigger/case** blocks during the execution of previously initiated actions, all processes that take time are transferred to separate components. These components are referred to as **Delay** components in case of simple activations and **Instance** components if multiple, overlapping activations may occur. In the second case, an additional **InstanceSelector** component selects and activates the individual instances by assigning input events to their **Trigger** ports.

Back to our *Tiny SysML Model* example, the UACM model generated by the MTI add-in (see Listing 7.2) is passed to the UACMbox. The *UACM Parser* parses the input model and extracts all relevant information, which is stored in a **UACMModel** object. As previously mentioned, this model provides methods for retrieving information about the model structure and the annotated timing specifications. It thus serves as the basis for generating the *RTana_{2sim}* model. Once the *RTana_{2sim} Transformer* is invoked, an analysis model for performing model-checking-based VITs is generated. The model is then written to an output file and can be analyzed using *RTana_{2sim}* (see Section 7.3.3). Listing 7.3 shows the resulting analysis model for the *Tiny SysML Model* example.

```

1  // Ports
2  port Analysis.Trigger[0]:=1;
3  port System.Input;
4  port C_System_A1.Input_OffsetPeriod;
5  port System.Output[0];
6  port SubSystemA.InputSubA;
7  port SubSystemB.InputSubB;
8  port SubSystemA.OutputSubA;
9  port SubSystemB.OutputSubB;
10
11 // Port mappings
12 portmap System.Input -> SubSystemA.InputSubA;
13 portmap SubSystemA.OutputSubA -> SubSystemB.InputSubB;
14 portmap SubSystemB.OutputSubB -> System.Output;
15
16 // Components
17 // Offset/period generator for top-level assumption
18 ↪ "Repetition(Input) each 33 ms with offset [0,33] ms and jitter 5
19 ↪ ms."
20 comp C_System_A1_OffsetPeriod:
21   // Handle first event occurrence
22   trigger in (I,0) on Analysis.Trigger:
23     case Analysis.Trigger?1:
24       after [0,33] write C_System_A1.Input_OffsetPeriod!1;
25       goto (I,1);
26   // Handle subsequent event occurrences
27   trigger in (I,1) on Analysis.Trigger:
28     case Analysis.Trigger?1:
29       after [33,33] write C_System_A1.Input_OffsetPeriod!1;
30       goto (I,1);
31 endcomp;

```

7 Tool Support

```

31 // Jitter generator for top-level assumption "Repetition(Input) each
↪ 33 ms with offset [0,33] ms and jitter 5 ms."
32 comp C_System_A1_Jitter:
33 // Delay the event occurrence by the duration of the jitter
34 trigger in (I,0) on C_System_A1.Input_OffsetPeriod:
35 case C_System_A1.Input_OffsetPeriod?1:
36 after [0,5] write System.Input!1;
37 goto (I,0);
38 endcomp;
39
40 // Generator for leaf guarantee "Reaction(InputSubA,OutputSubA)
↪ within [10,20] ms."
41 comp C_SubSystemA_G1:
42 // React to input events by writing output events
43 trigger in (I,0) on SubSystemA.InputSubA:
44 case SubSystemA.InputSubA?1:
45 after [10,20] write SubSystemA.OutputSubA!1;
46 goto (I,0);
47 endcomp;
48
49 // Generator for leaf guarantee "Reaction(InputSubB,OutputSubB)
↪ within [5,10] ms."
50 comp C_SubSystemB_G1:
51 // React to input events by writing output events
52 trigger in (I,0) on SubSystemB.InputSubB:
53 case SubSystemB.InputSubB?1:
54 after [5,10] write SubSystemB.OutputSubB!1;
55 goto (I,0);
56 endcomp;
57
58 // Observers
59 // Observer for top-level guarantee "Reaction(Input,Output) within
↪ (0,33] ms."
60 obs C_System_G1:
61 mode 0:
62 // Read the input events
63 -- System.Input -> 1;
64 mode 1 [33]:
65 // Check whether the reaction takes place in time
66 -- System.Output [1,33] -> 0;
67 endobs;
68
69 // Observer for leaf assumption "Repetition(InputSubA) each 33 ms
↪ with offset [0,33] ms and jitter 5 ms."
70 obs C_SubSystemA_A1:
71 mode 0 [38]:
72 // Handle first event occurrence
73 -- SubSystemA.InputSubA [0,38] -> 1;
74 mode 1 [38]:
75 // Handle subsequent event occurrences
76 -- SubSystemA.InputSubA [28,38] -> 1;
77 endobs;
78

```

```

79 // Observer for leaf assumption "Repetition(InputSubB) each 33 ms
↪   with offset [10,43] ms and jitter 15 ms."
80 obs C_SubSystemB_A1:
81   mode 0 [58]:
82     // Handle first event occurrence
83     -- SubSystemB.InputSubB [10,58] -> 1;
84   mode 1 [48]:
85     // Handle subsequent event occurrences
86     -- SubSystemB.InputSubB [18,48] -> 1;
87 endobs;
88
89 // Properties
90 property C_System_G1.mode != BAD;
91 property C_SubSystemA_A1.mode != BAD;
92 property C_SubSystemB_A1.mode != BAD;

```

Lst. 7.3: RTana_{2sim} analysis model for the *Tiny SysML Model*

Since the RTana_{2sim} transformer was developed as a proof-of-concept implementation, several limitations exist that might be addressed in future activities. For example, the current version of the transformer only supports rather simple event specifications, i.e., event specifications that consist of *single events*, *event sets*, and *event sequences*. For reasons of complexity, the second interpretation variant for event specifications was used instead of the standard interpretation, defined as part of the MTSL 1.5 language definition (see Section A.4.2). The implementation of mechanisms for handling *event selections* and *coregions* is work in progress. In addition, the corresponding execution automata created for generating or accepting events specifications currently lack support for event specifications that occur in parallel or overlap, so that the matching of one specification must always be completed before the matching of the next specification can start.

Additionally, the use of mixed time units in UACM input models is currently not supported. Therefore, all time values used in the MTSL specifications have to be defined based on the same time unit. Although an initial version of a **TimeTransformer** class, which should be responsible for converting time values between different time units, has been prepared, it has not yet been integrated into the UACMbox implementation.

The current version of the transformer does also not support combinations of pattern generators. In other words, it only supports input models whose leaf specifications contain a maximum of one guarantee per output port. In addition, the top-level specification may only contain a single assumption per input port. Future activities should focus on implementing a mechanism that combines multiple pattern generators based on their parallel composition, which would be the most effective way to handle such situations.

Last but not least, we would like to point out that **Reaction** and **Age** patterns are not yet fully supported. Currently, only single reactions similar to those intended by the **once** option are considered. Moreover, there is only limited support for handling overlapping activations. This results from restrictions in the internal implementations of RTana_{2sim} (see Section 7.3.3). Since the maximum number of modes per component is limited to eight due to restrictions that aims at reducing the memory consumption when unrolling the state space, only up to eight overlapping activations can be handled by assigning

them to one of the automatically generated instance automata. Finally, the specification of quantitative properties using K out of N constraints is not yet supported, but can be addressed by defining translation rules for generating corresponding automata.

TSLsim Transformer

In order to add legacy support for performing simulation-based VITs of MTSL 1.0 specifications, the TSLsim analysis backend (see Section 2.3.3) has been integrated by implementing a *TSLsim Transformer*. The transformer automates the translation of a given UACM model with MTSL 1.0 specifications into a set of C/C++ source files and a CMake build script. Following the translation, the generated files can be used to compile a SystemC-based simulation model for virtual integration testing (see Figure 2.7).

The transformer implementation was validated by comparing the generated source files with those produced by the MULTIC-Tooling tool prototype. All source files obtained by transforming the considered example models were identical. Note that compiling the source files for the models with MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC specifications would result in errors, as recent language versions are not supported. For this reason, the TSLsim transformer was provided with a pretest that prevents the transformation of UACM models with specification language versions other than MTSL 1.0.

7.3.3 RTana_{2sim}

RTana_{2sim} is a discrete-time model checker for real-time systems, dedicated to the verification of task networks. Its input model is based on a combination of TA-like task networks with real-world scheduling strategies such as fixed-priority and OSEK scheduling. The model-checking engine is based on an explicit state space exploration that handles all states as they occur in the system and does not rely on symbolic representations. The result is that RTana_{2sim} often generates very large state spaces even for rather small systems. On the other hand, the explicit representation of the state space simplifies comprehensibility and facilitates debugging of the resulting state sequences. The tool is not intended to be used in industrial scales, although it may also prove useful there.

The key features of RTana_{2sim} are the highly expressive system model, which supports all kind of architectures at various levels of granularity, a versatile property-checking machinery based on observer automata, and heuristics that allows to find relevant system states such as violations of specified properties. Moreover, RTana_{2sim} comes with a graphical user interface and provides automated and user-guided simulation support to derive initial analysis results despite the *state explosion* problem (see Section 2.1.5).

The graphical user interface of RTana_{2sim} is shown in Figure 7.10. After loading an RTana_{2sim} model, the input model is displayed in the upper left control panel (“*Input*”), which can also be used to manually edit models. The upper right control panel displays a status console (“*Log*”), which provides progress information. It is also used to present the analysis results, i.e., to state whether the specified properties are satisfied under all possible evolutions of the system’s behavior. The most important control elements for starting and stopping the analysis, i.e., the *Go* button that toggles to *Stop* while an

analysis is running, are located in the middle of the *RTana2sim* user interface. In addition, control elements for selecting options such as the analysis mode (*Analysis* or *Simulation*), the analysis strategy (*Full*, *Bound*, or *Random*), and optional termination criteria (i.e., a maximum number of states and/or a maximum analysis run time) are provided.

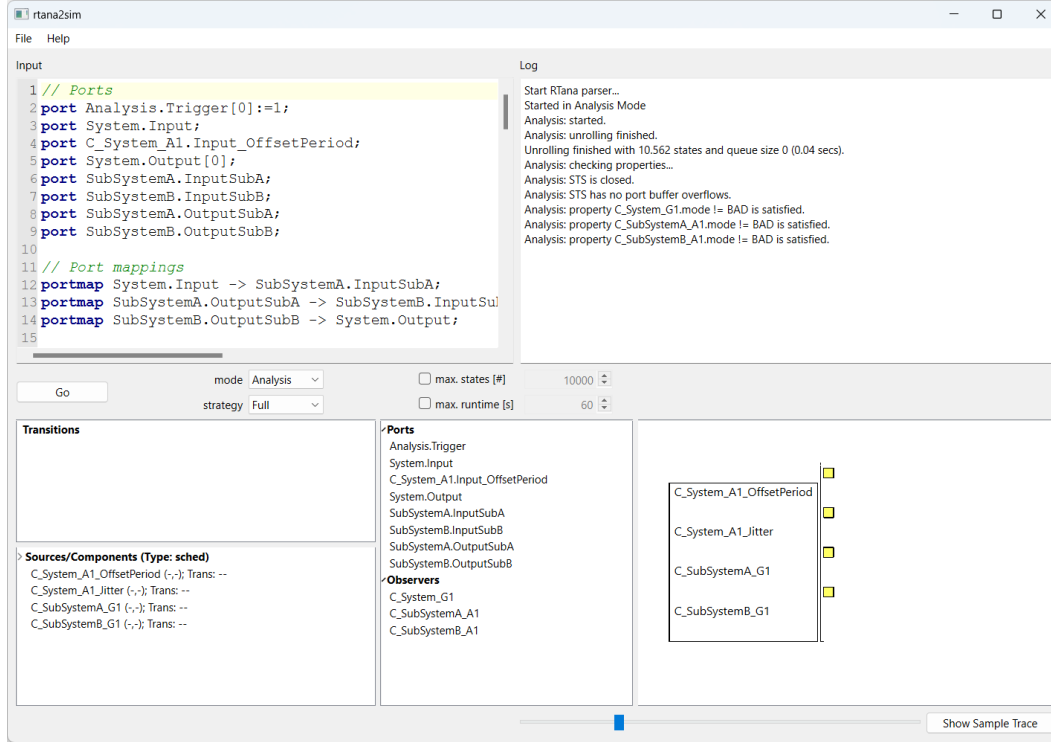


Fig. 7.10: Graphical user interface of *RTana2sim*

In the two lower left control panels, detailed information about possible transitions (“*Transitions*”) as well as the current states and modes of event sources, tasks, and components (“*Sources/Components*”) can be viewed after selecting a specific state (represented by small yellow rectangles) in the *Sample/Error Trace* viewer, which is located in the lower right control panel. The lower middle panel shows the current values of all ports (“*Ports*”) and the observer modes (“*Observers*”) for the selected state. The zoom level of the trace viewer can be adjusted using the slider at the bottom of the window. By clicking on the *Show Sample Trace* button (only visible if all properties are satisfied) or the *Show Error Trace* button (only visible if at least one property is violated), a corresponding example trace can be generated and displayed in the trace viewer.

The analysis model for the *Tiny SysML Model* application example, which was previously generated using the *RTana2sim Transformer* (see Section 7.3.2a), must currently be loaded manually by selecting the *Open* item from the *RTana2sim* menu. The analysis model is then displayed in the *Input* editor as described above. After selecting the *Analysis* mode and the *Full* strategy, the optional termination criteria are disabled. Then, the verification of the analysis model is initiated by clicking on the *Go* button, which triggers

7 Tool Support

the analysis that begins with the unrolling of the state space. The analysis model listed in Listing 7.2 yields a total of 10.562 states. In the next step, the analysis engine processes the resulting state space and verifies that the specified properties are satisfied by checking that none of the observers can reach its BAD mode. The analysis results are then displayed in the status console and indicate that all three guarantees are fulfilled. As a result, the VIT is successfully passed according to the conditions from Equation 2.3.

```
1 Start RTana parser...
2 Started in Analysis Mode
3 Analysis: started.
4 Analysis: unrolling finished.
5 Unrolling finished with 10.562 states and queue size 0 (0.04 secs).
6 Analysis: checking properties...
7 Analysis: STS is closed.
8 Analysis: STS has no port buffer overflows.
9 Analysis: property C_System_G1.mode != BAD is satisfied.
10 Analysis: property C_SubSystemA_A1.mode != BAD is satisfied.
11 Analysis: property C_SubSystemB_A1.mode != BAD is satisfied.
```

Lst. 7.4: RTana_{2sim} analysis output for the *Tiny SysML Model*

As an alternative to the graphical user interface, the execution of RTana_{2sim} analyses can also be initiated via command line interfaces (“batch mode”). For example, the VIT analysis model from Listing 7.3 can be analyzed by executing the following command:

```
./rtana2sim -b TinySysMLModel_System_vit.r2s
```

In this case, the user-guided, simulation-based exploration of the resulting state space is not supported. All other analysis options, such as the analysis strategy and the termination criteria, can be configured using corresponding parameters. Note that the analysis results are output in text format only, i.e., in accordance with Listing 7.4. Nevertheless, error traces can be generated in the form of *Best Trace Format* (BTF) and VCD files.

In the context of this work, the tool was initially used as is. Later, necessary adaptations were implemented as contributions to the RTana_{2sim} main repository. The extensions included adapting the RTana_{2sim} parser and the internal system model, as well as parts of the analysis engine, to support lists of port writes, port reads, and port references as activation conditions for observer transitions. These changes were necessary to handle the simultaneous occurrence of multiple events, e.g., of *change()* events that disable a specific pattern observer and input events that should be detected by the observer in parallel. All changes have been thoroughly tested and reviewed by the RTana_{2sim} core developers.

As mentioned earlier, RTana_{2sim} is not intended for industrial use, i.e., the tool is not optimized with regard to analysis run times or memory consumption. In particular, complex input models with a large number of tasks or components, long chains of interdependent interactions, large interval lengths, and large differences between the smallest and largest time values (or units) of an input model may result in long analysis run times. Therefore, follow-up activities should focus on optimizing internal implementations to enable efficiency gains and increase the practical benefits of using RTana_{2sim}.

Further limitations arise from restrictions in the underlying system model. Since port mappings are realized in terms of port substitutions in the in-memory input model, transient port mappings are currently not supported, which would be necessary to perform VITs for UACM models with more than two hierarchy levels. Corresponding extensions could aim to automatically replace transient port mappings with signals that already exist in the *RTana_{2sim}* input language, but cannot be used as flexibly as required.

In addition, limitations in verifying VIT analysis models result from restrictions in the internal implementations of tasks, components, and observers. For instance, the current version of *RTana_{2sim}* does not support the non-deterministic selection of port values, which prevents the instantiation of non-deterministic pattern generators. Since *RTana_{2sim}* already supports corresponding value selections for event sources, these implementations could serve as a basis for the desired extensions. Moreover, enhancements to support the automatic generation of task and component instances would be helpful, as they would simplify the handling of overlapping activations and eliminate the need to statically generate instance automata as part of the analysis models. Finally, observers should be extended to support multiple port reads and writes per transition, thus facilitating the observation of execution automata that read or write multiple events simultaneously.

The last group of possible extensions focuses on the graphical user interface. The current version of *RTana_{2sim}* provides all important control elements for performing model checking-based real-time analyses, but is not specifically tailored to the execution of VITs. For example, it would be desirable to establish an alternative trace view in which the events occurring in the system are made explicitly visible. Regarding the analysis of mode-based specifications, graphical visualization of component modes would facilitate user-guided simulation and allow for easier interpretation of analysis results.

7.4 Evaluation

In order to evaluate the applicability and practical use of the tool support, we assess the toolchain components based on their specification language support and performance. However, we should keep in mind that the current versions of the analysis backends were not designed for industrial use, but rather as proof-of-concept implementations supporting up-to-date MTSL versions. Consequently, they aim to demonstrate the practical applicability of the approach. For this reason, we also chose a lightweight evaluation method that does not include an in-depth analysis of the approach's scalability, but provides some first insights on how the complexity of the SysML+MTSL design models affects the execution times of model transformations and automated VITs.

In follow-up activities, a more detailed evaluation would be desirable. An extended evaluation could include, for example, the transformation and analysis of systematically generated or varied test models with increasing numbers of components, specifications, and operating modes as well as different numbers of hierarchy levels. Since the two analysis tools used as part of this thesis have no overlap in the supported MTSL versions, a comparative analysis is not helpful due to the different pattern semantics. However,

following the integration of further analysis tools, corresponding investigations would be highly relevant to identify preferred analysis methods for given system properties.

7.4.1 Specification Language Support

In a first step, we evaluate the individual components of the UACMbox Toolchain with respect to their specification language support. In this context, we consider the existing MTSL versions, different types of event specifications, and other aspects relevant to the specification of real-time behavior of cooperative SoCPSs as evaluation criteria.

Table 7.4 shows the specification language support of all four toolchain components that have been developed or integrated as part of this thesis. The MTI add-in and the UACMbox provide support for all available language versions, all kind of event specifications, i.e., single events, event sets, event selections, event sequences, and event coregions, as well as all aspects relevant to the formulation of timing specifications.

Tab. 7.4: Specification language support of different toolchain components

Toolchain Component	Language Versions				Event Specifications					Relevant Aspects					
	MTSL 1.0	MTSL 1.5	MTSL 1.5+Modes	MTSL 1.5+MC	Events	Event Sets	Event Selections	Event Sequences	Event Coregions	Event Occurrences	Event Relations	Quant. Properties	Operating Modes	Local Clocks	Causality
MULTIC-Tooling Industrial	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
UACM Toolbox	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
RTana _{2sim} Transformer	–	○	○	○	●	●	–	●	–	○	○	–	●	–	–
TSLsim Transformer	○	–	–	–	●	●	–	●	–	○	○	○	–	–	○

● = fully supported ○ = partially supported – = not supported

As mentioned above, the two transformers divide responsibility for supporting the available MTSL versions, i.e., TSLsim supports MTSL 1.0, and RTana_{2sim} supports MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC. Although both transformers and analysis tools can deal with events, event sets, and event sequences, they currently cannot handle event selections and event coregions. The RTana_{2sim} and TSLsim transformers both support the most common specification patterns for describing event occurrences and relations, i.e., **Repetition**, **Reaction**, and **Age**, but cannot handle all (combinations of) pattern extensions. For example, the specification of quantitative properties using **K out of N** constraints as well as causal relations are currently only supported by the TSLsim backend. As one of the results of this thesis, the RTana_{2sim} transformer has been developed to support performing VITs of mode-dependent timing specifications.

7.4.2 Performance

In a second step, the performance of the toolchain is evaluated. To this end, we rely on a total of six example models with different complexity, which are listed in Table 7.5. The first group of example models includes the *Tiny SysML Model* (see Figure 7.7 and Tables 7.1 to 7.3), the *Emergency Stop System (Level A)* (see Figure 2.5 and Tables 2.2 to 2.6), and its decomposed version as part of the *Emergency Stop System (Level A/B)* model (see Figure 2.6; specifications based on [92, Section 11.2]). The second group consists of example models with mode-dependent MTSL 1.5+MC specifications and comprises the *Sensor Processing* example (see Figure 4.11, Tables 4.7 to 4.9), the *Longitudinal Control System* (see Figure 4.9, Table 4.1, and Tables 4.4 to 4.6), as well as the *Cooperative Left-Turn Assistant* (see Figure 3.3, Tables 3.3 to 3.4, and Tables 3.6 to 3.15).

Tab. 7.5: Example models used for performance evaluation

Example Model	Spec. Language	# Comp.	# Patterns	Source(s)
<i>Tiny SysML Model</i>	MTSL 1.5	3	6 (A: 3/G: 3)	[93]
<i>Emergency Stop System (Level A)</i>	MTSL 1.5	5	23 (A: 9/G: 14)	[92, 93]
<i>Emergency Stop System (Level A/B)</i>	MTSL 1.5	15	43 (A: 15/G: 28)	[92, 93]
<i>Sensor Processing</i>	MTSL 1.5+MC	3	14 (A: 5/G: 9)	–
<i>Longitudinal Control System</i>	MTSL 1.5+MC	4	39 (A: 11/G: 28)	[370, 371]
<i>Cooperative Left-Turn Assistant</i> ¹	MTSL 1.5+MC	5	99 (A: 43/G: 56)	–

¹ Note that a top-level specification with ten specification patterns reflecting the assumptions of lower-level components has been added to obtain a valid analysis model.

All example models were prepared in terms of SysML models according to the modeling guidelines presented in Appendix B using Enterprise Architect 17. Then, the MTSL timing specifications were inserted into *Requirement* blocks using the *Specification Editor* of the MTI add-in. In this step, all right-open time intervals $[lb, \infty)$ ms, and (lb, ∞) ms, with $lb \in \mathbb{T}$ were replaced by corresponding right-closed time intervals $[lb, t_{max}]$ ms, and $(lb, t_{max}]$ ms, with $t_{max} \in \mathbb{T}$. Note that the time value $t_{max} = 2500$ ms was selected for all subsequent experiments, since infinite interval bounds are not supported by RTana_{2sim}. However, any other (finite) time value could be used to adjust the specifications.

During the evaluation, the following steps were performed for each example model:

SysML-to-UACM Transformation First, the SysML models with annotated MTSL timing specifications were opened in Enterprise Architect 17. Then, corresponding UACM models were generated using the MTI add-in. During the transformation process, the default settings and an unlimited descent depth were used. The only exception was the *Emergency Stop System (Level A)* that was processed using a limited descent depth of 1 to extract the relevant parts of the ESS model.

UACM-to-RTana₂ Transformation The resulting UACM output files were then passed to the UACMbox, which was previously configured to use the RTana_{2sim} transformer for generating RTana_{2sim} analysis models. Note that the generation of C/C++ source files for the legacy TSLsim backend has not been investigated.

Analysis Execution After loading the analysis models into RTana_{2sim} , the execution of model-checking-based VITs was conducted. The experiments included VITs on different execution platforms and operating systems. All test runs were performed using the graphical user interface of RTana_{2sim} on a standard Windows notebook and in “batch mode” on a high-performance Linux computing server.

Before conducting the experiments, all involved toolchain components, i.e., the MTI add-in, the UACMbox, and RTana_{2sim} were instrumented with method calls to measure their execution times. For this purpose, the `System.Diagnostics.Stopwatch` class was used for the MTI add-in written in C#, and `std::chrono::high_resolution_clock` in case of the C++ implementations of the UACMbox and RTana_{2sim} . All time measurements were taken in milliseconds and were then converted to seconds with an accuracy of three decimal places. Note that each test run was executed 10 times in order to handle variations in the measured execution times. Finally, the subsequent processing of the time values to determine minimum, average, and maximum values was done using Microsoft Excel.

The resulting execution times of the two model transformations are listed in Table 7.6. In all cases, the transformations were completed without any errors. The execution times for both the SysML-to-UACM and the UACM-to- RTana_{2sim} transformations were very short, meaning that the results are usually generated in less than a second. For larger input models (see *Emergency Stop System (Level A/B)* and *Cooperative Left-Turn Assistant*), the execution times increase slightly, but are still below two seconds.

Tab. 7.6: Execution times of model transformations

<i>Example Model</i>	<i>SysML-to-UACM Transformation¹</i>			<i>UACM-to-RTana_{2sim} Transformation¹</i>		
	<i>Min.²</i>	<i>Avg.²</i>	<i>Max.²</i>	<i>Min.²</i>	<i>Avg.²</i>	<i>Max.²</i>
<i>Tiny SysML Model</i>	0.24	0.26	0.28	0.04	0.05	0.05
<i>Emergency Stop System (Level A)</i>	0.63	0.65	0.67	0.13	0.13	0.14
<i>Emergency Stop System (Level A/B)</i>	1.71	1.78	1.95	0.29	0.30	0.30
<i>Sensor Processing</i>	0.29	0.31	0.33	0.08	0.08	0.09
<i>Longitudinal Control System</i>	0.59	0.62	0.64	0.23	0.23	0.24
<i>Cooperative Left-Turn Assistant</i>	1.33	1.36	1.44	0.59	0.60	0.62

¹ All test runs were executed on a standard notebook (Intel Core i7, 4x 3.00 GHz, 16 GB RAM, SSD, Windows 11). ² All execution times are given in seconds.

Further experiments have also demonstrated the scalability of the MTI add-in. For example, a SysML model that includes the top-level functional architecture and corresponding MTSL 1.5+MC specifications of an entire CHAV with a total of 113 components, 416 ports, 113 specifications, and 272 signals was translated into a consistent UACM output file in less than 15 seconds (min.: 14.09s, avg.: 14.37s, max.: 14.84s.).

Tab. 7.7: Analysis execution times on different platforms

<i>Example Model</i>	<i>Standard Notebook</i> ¹			<i>Computing Server</i> ²		
	<i>Min.</i> ³	<i>Avg.</i> ³	<i>Max.</i> ³	<i>Min.</i> ³	<i>Avg.</i> ³	<i>Max.</i> ³
<i>Tiny SysML Model</i>	0.05	0.05	0.05	0.05	0.05	0.05
<i>Emergency Stop System (Level A)</i>	—	> 3600.00	—	—	> 1694.81	—
<i>Emergency Stop System (Level A/B)</i>	—	> 3600.00	—	—	> 1656.58	—
<i>Sensor Processing</i>	—	> 3600.00	—	349.88	353.64	355.72
<i>Longitudinal Control System</i>	—	> 3600.00	—	—	> 2245.42	—
<i>Cooperative Left-Turn Assistant</i>	Not possible			Not possible		

¹ Execution on a standard notebook (Intel Core i7, 4x 3.00 GHz, 16 GB RAM, SSD, Windows 11) ² Execution on a computing server (AMD Epyc 7542, 20x 2.90 GHz, 128 GB RAM, SSD, Ubuntu 24.04) ³ All execution times are given in seconds.

Tab. 7.8: Key figures related to analysis execution

<i>Example Model</i>	<i># States</i> ¹	<i>Status</i> ¹	<i>Result</i> ¹	<i>Problem(s)</i> ¹
<i>Tiny SysML Model</i>	10 562	Complete	✓	—
<i>Emergency Stop System (Level A)</i>	> 123 281 823	Partial	(✓) ²	Memory overflow
<i>Emergency Stop System (Level A/B)</i>	> 131 477 370	Partial	(×) ²	Memory overflow
<i>Sensor Processing</i>	30 042 174	Complete	✓	—
<i>Longitudinal Control System</i>	> 72 288 027	Partial	(×) ²	Memory overflow
<i>Cooperative Left-Turn Assistant</i>	Unknown	Not possible	?	No response

¹ All test runs were executed on a computing server (AMD Epyc 7542, 20x 2.90 GHz, 128 GB RAM, SSD, Ubuntu 24.04). ² Analysis results were obtained using partial state spaces.

As described above, the execution of VITs is evaluated by analyzing the previously generated analysis models using RTana_{2sim}.² Prior to the experiments, a maximum execution time of one hour, i.e., 3 600 s, was defined as a common termination criterion for all test runs. In addition, model-specific termination criteria in terms of maximum numbers of states that can be safely unrolled were identified in pretests to prevent memory overflows and allow to obtain analysis results based on partial state spaces.

The evaluation results related to the analysis execution are presented in Tables 7.7 and 7.8. The first of the two tables, Table 7.7, indicates minimum, average, and maximum analysis run times for performing VITs on both execution platforms, i.e., the standard notebook and the more powerful computing server. In addition, Table 7.8 lists the resulting number of states, information about the completeness of the analyses, the final verification results, and potential problems. The analysis results are as follows:

Tiny SysML Model The VIT for the *Tiny SysML Model* using RTana_{2sim} was performed without any errors and with reasonable run times. Unrolling and analyzing the resulting 10 562 states took 0.05 s on both execution platforms (see Table 7.7). The successful analysis proves the correct refinement of the top-level specification.

²Note that minor adjustments were necessary to complete the analyses. All manual changes were carefully documented and have mostly been implemented. Since the impact on implementation is minimal and will result in smaller state spaces, we expect improvements in analysis performance.

Emergency Stop System (Level A) Due to the number of resulting states, the state space of the *Emergency Stop System (Level A)* model could not be fully unrolled and analyzed on both execution platforms. While the maximum execution time of one hour was exceeded on the standard notebook (see Table 7.7), the analysis execution caused memory overflows on the computing server (see Table 7.8), which led to the automatic termination of the RTana_{2sim} processes in all test runs.

As a pragmatic solution, we continued performing VITs based on partial state spaces. As described above, RTana_{2sim} is able to check previously specified properties in this case as well. In a first step, we unrolled 100 000 states on the standard notebook, which took less than a second. Subsequently, we gradually increased the number of states to be investigated by an order of magnitude, resulting in the exploration of 1 000 000 (< 10 s) and 10 000 000 (< 2 min.) states. These experiments shed some light on the interaction of the specifications and already revealed first inconsistencies that could be manually resolved using the MTI add-in. After generating an updated analysis model, the next iteration of the “engineering cycle” (see Figure 2.8) was initiated, starting again with unrolling and analyzing the first 100 000 states. Note that 120 000 000 states were used to generate the results presented in Table 7.8.

Emergency Stop System (Level A/B) As expected, it was not possible to complete the verification of the *Emergency Stop System (Level A/B)* model for the same reasons as the previous model (see Table 7.7). Following the iterative assessment on the standard notebook, we investigated 130 000 000 states using the computing server to obtain the analysis results presented in Table 7.8. Note that this was the maximum number of states we were able to analyze with the available memory.

Further analysis revealed that the property violations may have been incorrect and resulted from RTana_{2sim}’s lack of support for transient port mappings. Note that this issue has already been identified as a required extension in Section 7.3.3. Therefore, we plan to continue the evaluation after the implementation of the analysis tool has been improved. Once again, the proposed procedure proved to be a helpful tool for identifying issues with the generated analysis models.

Sensor Processing The state space of the *Sensor Processing* model comprised 30 042 174 states, which led to the maximum execution time of one hour being exceeded in all test runs on the standard notebook (see Table 7.7). However, on the computing server, the model-checking-based VIT provided error-free results after about six minutes, stating that all properties were satisfied (see Table 7.8). Since we thoroughly investigated the entire state space, the analysis results verify that the included mode-dependent MTSL 1.5+MC specifications were refined correctly.

Longitudinal Control System The use of mode-dependent specifications leads to a higher complexity of the RTana_{2sim} analysis models. For this reason, the VIT for the *Longitudinal Control System* model could not be completed using the standard notebook (see Table 7.7). The resulting state space could also only be partially analyzed on the computing server (see Table 7.8). During the first analysis run using

a partial states space with 100 000 states, an error in the handling of multi-read input ports, i.e., the *D*, *LV*, and *EV* ports of the top-level component (see Figure 4.9), was identified and fixed in the implementation of the *RTana_{2sim}* transformer.

Based on the analysis of partial state spaces, two inconsistencies were identified in the MTSL 1.5+MC specifications.³ First, a buffer overflow on the *Request* port of the lower-level *ACC* component was detected. This indicated that the inter-arrival time of the occurring events was too short to allow proper processing. In order to fix this issue, we adjusted the lower offset and period bounds of the corresponding *Repetition* patterns (see Table 4.1, line 3, and Table 4.4, line 3) to 5 ms.

After correcting the two specifications, the experiments were continued by unrolling and analyzing 100 000 states. This time, *RTana_{2sim}* reported `C_lcs_g8.mode != BAD` as the analysis result, indicating that the specification pattern

`Reaction(Req,set(Mode,I)) within (0,5] ms in mode {C,F,E}.`

included in the top-level guarantee was violated (see Table 4.1, line 11). An in-depth analysis revealed that the desired behavior was not fully reflected in the specifications of the lower-level components. The corresponding *Reaction* patterns

`Reaction(Req,set(Mode,{C,F})) within (0,1] ms in mode I.`
`Reaction(Req,set(Mode,I)) within (0,1] ms in mode {C,F}.`

that are part of the lower-level guarantee of the *ACC* component (see Table 4.4, lines 4–5) do not specify reactions to incoming *Request* events in *Evade* mode. The absence of such a specification is intentional, since the top-level mode is derived from the lower-level *Evade* mode of the *SL* component. However, a corresponding statement in the *SL* component’s specification (see Table 4.6) is missing. Consequently, the specifications are inconsistent and violate the VIT conditions. Note that the analysis results from Table 7.8 were obtained using 70 000 000 states.

Cooperative Left-Turn Assistant During the evaluation, the *Cooperative Left-Turn Assistant* model could not be analyzed. While loading the analysis model into *RTana_{2sim}*, the tool became unresponsive and had to be restarted (see Table 7.8). This behavior is most likely due to limitations in the implementation of the analysis tool, which only allows a maximum number of eight modes per execution automaton. Consequently, the 14 modes of the *CHAV* components could not be mapped to internal modes of *RTana_{2sim}* components and observers, resulting in undesired behavior. Note that the maximum number of allowed modes accepted as part of the input model validation of the *RTana_{2sim}* transformer had to be increased manually from 8 to 16 to successfully transfer all 14 modes to the analysis model. During normal use, such situations would be detected early on, and appropriate warnings would be issued. Nevertheless, it would be desirable to be able to analyze

³Note that the MTSL 1.5+Modes specifications in the original publications [370, 371] were formulated without appropriate tool support. Thus, all specifications were written “by hand” and could only be validated by discussing manually generated example traces, such as those shown in Figure 4.10.

components with more than eight modes. RTana_{2sim} should therefore be enhanced in future activities to address this issue. The evaluation can then be continued.

We are well aware that the investigation of partial state spaces does not allow to prove the absence of property violations, and thus does not suffice to formally verify contract refinement. However, in industrial design processes, requirement engineers could rely on such iterative approaches to first identify and analyze easy-to-find inconsistencies based on partial state spaces on their workstations, before formally verifying correct refinement of the revised specifications on high-performance computing servers. Based on the UACMbox Toolchain, early assessment of potential consistency problems could also be supported by performing simulation-based VITs of the same design models.

8 Conclusion

The focus of this thesis was the development of concepts, methods, and tools to ensure real-time behavior of cooperative SoCPSs. The results were intended as building blocks that aim at establishing a SoCPS-wide timing coherence during system design. In all phases, priority was given to the question of how engineers that are faced with challenges in designing, verifying, validating, and approving increasingly complex, connected, highly automated, and cooperative systems can be supported. Additionally, parts of the results may serve as input for certification bodies and standardization committees.

The procedure for addressing the goals was as follows. Based on a detailed analysis of the characteristics of cooperative systems in Sections 1.2.1 and 1.2.2, challenges in ensuring real-time behavior were identified in Section 1.2.3. The challenges formed the basis for the formulation of four research questions RQ-1 to RQ-4 in Section 1.3. In Chapter 2, a careful analysis of the state of the art was first conducted in Section 2.1, culminating in the selection of a suitable methodology for specifying temporal properties of single systems in Section 2.2. Following a detailed review of the existing methodology in Section 2.3, a gap analysis with respect to the previously identified challenges revealed four methodological and two technical gaps, which led to the definition of the contribution in Section 2.4. At the same time, the gaps were used to structure the thesis. Chapters 3 to 7 present the results that address the identified gaps and answer RQ-1 to RQ-4.

This chapter summarizes the results and identifies further research needs. To this end, we review the outcomes of all previous chapters from the perspective of the research questions in Section 8.1. In doing so, we evaluate whether the original goals were achieved. Finally, we discuss current limitations and possible extensions in Section 8.2.

8.1 Review of Goals

As previously mentioned, the goals of this thesis were defined in terms of four research questions (see Section 1.3). The following subsections summarize the results and indicate which parts of the thesis contribute to answering the individual questions. For reasons of clarity, the top-level research question in focus is repeated for each topic area.

8.1.1 Consistent Formulation of Real-Time Requirements

RQ-1 *How can engineers be supported in formulating consistent real-time requirements for (systems of) cooperating cyber-physical systems?*

The first research question was addressed through the development of a correct-by-design approach for the consistent formulation and refinement of real-time requirements on SoCPS

level, which is presented in Chapter 3. The envisioned design approach builds upon on the MULTIC methodology (see Section 2.3) and adds support for early determination of cross-system coordination processes (see Section 3.2). It introduces a formalism called *Time-Constrained Activity Diagrams* (TCADs) that allows scenario-based modeling of cooperative behavior (see Section 3.3). The formalism enables the pre-competitive design of cooperative system functions in terms of implementation-independent interaction protocols. The resulting activity models are partitioned to assign activities and actions to individual subsystems, enabling the derivation of system-level specifications (see Section 3.4). This serves as a starting point for the parallel development of CSs based on the previously agreed top-level specifications (see Section 3.5). The approach facilitates early collaboration between manufacturers and suppliers to develop consistent top-level specifications in early design phases that enable safe and reliable cooperation.

From a technical perspective, the developed tool support for performing VITs (see Chapter 7) can be understood as another building block to support engineers in formulating consistent real-time requirements. Moreover, the MTSL has been improved to cover the most important time phenomena relevant to the development of cooperative SoCPSs (see Appendix A). Finally, a comprehensive set of modeling guidelines that address relevant design activities, i.e., scenario-based activity modeling (see Section B.1) and the application of the extended MULTIC methodology (see Section B.2), have been defined. The modeling guidelines can be used to create relevant design artifacts in terms of SysML models, thus facilitating the transfer of the results into industrial practice.

8.1.2 Specification Support for Cooperative Systems

RQ-2 *How can existing specification means be extended to cover relevant aspects of cooperative systems?*

The second research question was addressed by developing suitable specification means to provide engineering support for both process phases of the envisioned design approach, i.e., the *Pre-Competitive Design of Cooperative System Behavior* on SoCPS level and the *Manufacturer-Specific Design and Implementation* on CS level (see Section 3.2).

The first process phase, *Pre-Competitive Design of Cooperative System Behavior*, is dedicated to the specification of real-time behavior on SoCPS level (see Section 3.3). It relies on the TCAD formalism that aims at specifying the desired (timing) behavior of cooperative systems based on cooperative scenarios. TCADs support the modeling of relevant actions and control flows of individual CSs that contribute to safe and reliable interaction in cooperative SoCPSs. The annotation of timing and resource constraints allows simple yet effective specification of real-time behavior. The formalism supports specifying precedence, duration, synchronization, action-to-action latency, and uniqueness constraints. In addition, temperatures of actions and control flows can be defined to ensure or dismiss progress (see Section 3.3.1). Each action is mapped to a single partition that represents an involved actor. Based on this mapping, all actions are uniquely assigned to individual CSs that are part of the cooperative system, which explicitly

defines responsibilities. The practical suitability of the developed specification means was demonstrated using the CLTA application example (see Section 3.3.2). A systematic translation approach allows the derivation of consistent timing specifications on system level, thus ensuring a seamless transition to the second process phase (see Section 3.4).

The second process phase, *Manufacturer-Specific Design and Implementation*, focuses on specifying real-time behavior of individual CSs. In this context, the MTSL 1.5 specification language used in the MULTIC methodology was enhanced through consistent extensions to address MG-2 (see Chapter 4), MG-3 (see Chapter 5), and MG-4 (see Chapter 6). The extensions, which led to the definition of MTSL 1.5+MC as a new version of the MTSL specification language (see Appendix A), were completed in three stages:

Handling of Dynamic Architectures In order to address the second methodological gap “*Missing support for handling dynamic system architectures*” (MG-2), the existing MTSL 1.5 was extended to support the specification of operating modes, mode transitions, and mode-dependent timing behavior. In a first step, the original system model (see Section 2.3.1) was revised by including relevant concepts required for reasoning about component modes (see Section 4.2.2). Subsequently, the specification of dependencies between modes on different hierarchy levels was addressed by introducing suitable description means (see Section 4.2.3). The integration of operating modes, mode transitions, and mode-dependent timing behavior was completed by defining consistent extensions of commonly used specification patterns (see Section 4.2.4). Finally, mode-dependent specifications were used to model dynamic reconfiguration (see Section 4.3). The considered reconfiguration processes concern both behavioral changes as well as adaptations of interfaces.

Modeling of Unreliable Communication The third methodological gap “*Insufficient support for modeling unreliable communication*” (MG-3) was addressed by defining a generic communication model with example parameter sets for IEEE 802.11p and C-V2X, as the two most important V2X communication standards from the automotive domain (see Section 5.2). For this purpose, a systematic literature review was conducted to identify relevant time phenomena (see Section 5.3). The identified phenomena were then mapped to MTSL 1.5+MC specification patterns to enable their use in contract-based design methodologies such as the envisioned design approach. The communication model and the example parameter sets (see Section 5.4) allow early consideration of temporal properties of unreliable communication by considering relevant failure modes in functional system design.

Mapping of Specifications to Imprecise Local Clocks The fourth methodological gap “*Insufficient support for mapping specifications to imprecise local clocks*” (MG-4) was addressed by systematically investigating local clocks with typical properties, i.e., resolution, drift, skew, and difference constraints. To this end, clock relations for each type of clock were formally defined that allow the mapping of specifications to imprecise local clocks (see Section 6.3). The careful use of under- and overapproximations enabled the construction of meaningful and traceable relations between the

two worlds (see Section 6.4). In order to support contract-based specification, specification means from MTSL 1.0 for defining local clocks and clock references were integrated into MTSL 1.5+MC (see Section 6.5). The results facilitate the explicit specification and tracing of clock properties as well as the automated derivation of underapproximating specifications for use in technical system design.

8.1.3 Automated Verification of Timing Specifications

RQ-3 *How must verification methods for single systems be adapted to support virtual integration testing of real-time requirements for cooperative systems?*

The third research question was addressed by adapting existing verification means to support VITs on SoCPS level and to ensure correct refinement of mode-dependent specifications during functional system design. Additionally, the developed approach for mapping design-time specifications to imprecise local clocks enables distributed run-time monitoring of real-time behavior. The results related to RQ-2 thus contributed to addressing MG-2 (see Chapter 4), MG-4 (see Chapter 6), and TG-2 (see Chapter 7).

The verification methods developed as part of this thesis primarily focus on the *Manufacturer-Specific Design and Implementation* phase (see Section 3.2.2). However, the design approach for the consistent formulation and refinement of real-time requirements on SoCPS level, which is part of the first process phase, sets the stage for effectively applying the developed methods. Due to their formal semantics (see Section 3.3.1), all models and formalisms used as part of the approach are accessible to rigorous verification techniques. Following the scenario-based activity modeling (see Section 3.3), the resulting TCADs are translated into consistent MTSL 1.5+MC timing specifications for each CS using a systematic, five-step translation procedure (see Section 3.4). As a first step to verify correct refinement of the initial top-level requirements, which are formulated in terms of activity duration constraints of TCADs, the envisioned design approach requires the execution of a VIT based on the initial top-level architecture (see Section 3.5).

The second process phase focuses on the development of the individual CSs based on the previously derived MTSL 1.5+MC specifications. The approach builds upon the contract-based MULTIC methodology (see Section 2.3) and is backed up by a rich set of timing specifications that enable specifying the timing behavior of cooperative SoCPSs. The formulation of specifications in terms of A/G contracts with well-defined operations and corresponding proof obligations (see Section 2.1.3) enables independent development and incremental verification. In order to support VITs for mode-dependent specifications, the procedure for generating analysis models (see Section 2.3.3) was adapted to include mode-dependent pattern generators and observers. Moreover, specialized mode automata that realize the mapping between lower-level and higher-level modes were included (see Section 7.3.2). The extensions provide cross-layer process support for the specification and tool-supported refinement of mode-dependent timing specifications.

Furthermore, the generic communication model was designed to serve as a tool for verifying cooperative SoCPSs under unreliable communication (see Section 5.4). The

model captures relevant failure modes of V2X communication systems and characterizes their effects on timing behavior. It can thus be used to analyze the consistency of error correction mechanisms and fallback strategies to ensure proper interaction. The developed approach for mapping specifications to imprecise local clocks supports distributed run-time monitoring. In this context, effective mappings were established by defining faithful and approximate clock relations (see Sections 6.3 and 6.4). The use of underapproximating MTSL 1.5+MC specifications enabled the reliable detection of $> 99.99\%$ of delayed or omitted events that were required by the global specification (see Section 6.5).

The systematic use of VITs fulfills relevant proof obligations and generates required evidence as part of the process. This ensures the consistency of the resulting lower-level specifications with the original system-level specifications, and guarantees the compatibility of the behavior of CSs on SoCPS level. Following the implementation of the individual CSs, the final step is to verify the compliance with their refined specifications. To this end, formal verification methods [58], simulation-based approaches [93, 164], and online verification techniques [206] have been applied in previous publications.

8.1.4 Modular and Extensible Tool Support

RQ-4 *How can the resulting verification methods be implemented and connected to industry-relevant modeling tools in a modular and extensible way?*

The fourth research question was addressed by developing a modular and extensible toolchain for real-time analyses, referred to as UACMbox Toolchain (see Section 7.2). The toolchain integrates an industry-relevant modeling tool and provides support for automated VITs of MTSL specifications in different versions. For this purpose, it comprises a model transformation adapter for Enterprise Architect (see Section 7.3.1) that translates SysML+MTSL design models into a lightweight intermediate format (see Section 7.3.2). Existing analysis tools are integrated for performing VITs based on the intermediate format. As the main analysis backend, the discrete-time model checker RTana_{2sim} enables formal verification of contract refinement for MTSL 1.5, MTSL 1.5+Modes, and MTSL 1.5+MC specifications (see Section 7.3.3). In addition, the TSLsim backend can be used to perform simulation-based VITs of MTSL 1.0 specifications. The optional use of complexity reduction methods, such as the analysis of partial models, was enabled to address complex design models. Finally, the tool support was evaluated with respect to specification language coverage and performance (see Section 7.4). Although the toolchain is not yet fully integrated, it has already proven useful. A set of modeling guidelines provides engineers with guidance for creating valid design models (see Appendix B).

8.2 Limitations and Future Work

In Chapters 3 to 7, we have already identified a number of limitations and ideas for possible extensions. This section reviews the most significant limitations and provides an

overview of further research and development needs. Note that the findings are presented along the structure of this work, i.e., by one topic area for each chapter.

Consistent Specification of Top-Level Timing Requirements The first phase of the envisioned design approach is based on the iterative scenario-based modeling of relevant activities. In this context, a conceivable extension would be to strengthen the ties to scenario-based V&V processes. Through appropriate extensions, existing approaches for identifying hazardous scenarios [430, 446] could be leveraged in order to select relevant scenarios that could complement the initial “sunny day” scenarios. This would allow the consideration of critical scenarios at an early stage.

With regard to the translation process, it would be desirable to ensure complete support for all TCAD modeling elements and constraints. For example, the translation of synchronization constraints is currently not supported due to the lack of corresponding MTSL specification patterns. Section 6.5 already discussed an early version of such a pattern that could be used as the basis for the extensions.

In order to provide early feedback on the consistency of specifications, another idea is to develop and implement “VITs on activity level”. Corresponding analysis methods could build upon the same basic principle of classical VITs, elevating the underlying concepts to a higher level of abstraction to ensure requirement consistency and compatibility based on TCAD timing constraints.

The proposed design approach is defined as a purely top-down procedure. In industrial practice, however, existing systems are often integrated for use in other contexts or to perform new tasks. In this context, it would be desirable to provide support for bottom-up or meet-in-the-middle approaches that allow the use of available knowledge about existing systems and legacy code as well as data on execution times or communication density. At the same time, the planned extensions would provide guidance for the effective decomposition of top-level requirements.

Handling of Dynamic Architectures From a modeling perspective, the existing design approach could be enhanced by including graphical models of available modes and valid transitions in the form of SysML state machines. This would allow the implementation of more effective validation methods for mode-dependent specifications, such as automatic checks to ensure that timing specifications have been defined for all operating modes and that all transitions have been considered.

In order to address performance limitations of the developed analysis support, alternative approaches for constructing the analysis models to perform VITs of mode-dependent specifications should be evaluated. To this end, the suitability of existing verification techniques [163] for integration into model transformations or analysis tools as part of the UACMbox Toolchain could be examined.

Furthermore, a more detailed investigation of dynamic reconfiguration processes would be desirable. In this context, the extensions proposed in Section 4.3 could initially be assessed using suitable application examples, e.g., by considering vehicle

platooning [17, 350]. The most urgent question is probably how considering dynamic systems or components affects design and verification complexity.

Modeling of Unreliable Communication The generic communication model developed as part of this thesis can be understood as a first step toward the early consideration of unreliable communication in the design of cooperative SoCPSs. In order to refine the developed communication model, the example parameter sets could be enhanced by conducting an in-depth analysis of the SLR results. Finally, the consistency of the resulting specification templates should be verified by performing VITs.

In more extensive follow-up activities, the generic communication model could be validated, e.g., by comparing the end-to-end timing behavior resulting from the model with simulation results obtained using VEINS [545] or Artery [503]. The validation could be completed by comparing the results with timing information from real-world datasets, such as DAIR-V2X [614] or TUMTraf-V2X [626].

Mapping of Specifications to Imprecise Local Clocks Regarding the handling of imprecise local clocks, a desirable extension would be to lift the implicit assumption about the order of events. Recall that we have used *time-bounded until* properties $e \mathcal{U}_I f$ to systematically investigate faithfulness of clock relations. The operator implies that the e event occurs before the f event. However, for some specification patterns, such as the **Age** pattern, consideration in the opposite direction is required. Thus, the extensions would improve support for mapping specifications.

In addition, the investigation of clocks with typical properties could be enhanced by considering the ground truth of global and local observations. The current approach does not distinguish between phenomena inherent to imprecise local clocks and phenomena that arise from using them to map certain types of specifications. Establishing such a distinction would strengthen the consistency arguments and could be used as a basis for formally proving the correctness of clock relations.

From a verification perspective, the next logical step would be to add support for performing VITs to ensure consistency between global and (approximate) local specifications. The extensions should also address the mapping of mode-dependent MTSL 1.5+MC specifications to imprecise local clocks, which could be achieved, for example, by enabling and disabling local observers based on the active mode.

Tool Support The next step in developing the tool support is to close the remaining gaps in the implementation. Among other things, this includes to lift the restriction of RTana_{2sim} to a maximum of eight modes per execution automata and implement appropriate extensions to address the manual adjustments necessary for successful verification. In addition, the use of multiple pattern generators per port should be enabled by implementing methods to derive composed pattern generators.

As mentioned earlier, RTana_{2sim} is not intended for industrial use, i.e., the tool is not optimized with regard to analysis run times or memory consumption. Therefore, follow-up activities should focus on optimizing internal implementations to enable

8 Conclusion

efficiency gains and increase the practical benefits of model-checking-based VITs. In addition, it would be helpful to examine alternative analysis tools with regard to their suitability for performing VITs of MTSL 1.5+MC specifications.

Following the previously mentioned enhancements, the control flow integration of the UACMbox Toolchain could be completed. For this purpose, the MTI add-in could be used to orchestrate the execution of other toolchain components.

The last group of extensions aims at extending the toolchain's functionality. In this context, the modeling of TCADs in the form of SysML activity diagrams based on the guidelines presented in Appendix B could serve as a basis for implementing the translation procedure developed in Section 3.4. This would enable the automated generation of system-level MTSL specifications based on TCADs. Finally, the UACMbox Toolchain could be enhanced with extensions to perform satisfaction checks [58] and generate observers for distributed run-time monitoring [206].

A Timing Specification Language

The *MULTIC Timing Specification Language* (MTSL) is an integral part of the MULTIC methodology, which is extended as part of this thesis to support the characteristics of cooperative systems. The original version of the specification language, which is referred to as MTSL 1.0, was the result of two one-year projects called MULTIC [92] and MULTIC-Tooling [93] that were supported by the *Research Association for Automotive Technology* (FAT) of the *German Association of the Automotive Industry* (VDA).

The current version of the timing specification language is an outcome of years of iterative development, driven by a number of national and European research projects. Over the years, subsequent language versions, such as MTSL 1.5, which revises the syntax and semantics of the original MTSL 1.0, and MTSL 1.5+Modes, which adds support for handling operating modes and mode-dependent timing behavior [370, 371], have been defined. This thesis represents a further milestone in the evolution of the MTSL specification language. All relevant results from the previous chapters have been incorporated into a new intermediate version, referred to as MTSL 1.5+MC. It forms the basis for future extensions, such as the reintegration of causal timing specifications.

This appendix provides an overview of all language versions and compares their individual features. Table A.1 lists the MTSL versions and indicates the included *specification patterns* as well as the supported *aspects*. The assessment of the individual entries is visualized by means of classifications *fully supported* (●), *partially supported* (○), and *not supported* (–). Depending on the relevance of the MTSL version to this thesis, the definitions of its syntax and semantics are either given directly or referenced.

Tab. A.1: Comparison of existing MTSL versions

Version	Specification Patterns								Aspects				References	
	Single Event	Repetition	Reaction	Age	Causal Reaction	Causal Age	Causality Function	Clock Definition	Mode Mapping	Quant. Properties	Operating Modes	Local Clocks		Causality
MTSL 1.5+MC	–	●	●	●	–	–	–	●	●	○	●	●	–	(this thesis)
MTSL 1.5+Modes	–	●	●	●	–	–	–	–	●	○	●	–	–	[370, 371]
MTSL 1.5	–	●	●	●	–	–	–	–	–	○	–	–	–	[559]
MTSL 1.0	●	●	●	●	○	○	○	●	–	○	–	○	○	[92, 93]

● = fully supported ○ = partially supported – = not supported

The existing MTSL versions can be differentiated based on (1) the supported specification patterns and (2) whether and how relevant aspects are taken into account. The following sections aim to harmonize the language definitions and provide a basis for developing tool support in Chapter 7. First, a set of common definitions is presented in Section A.1. The description of the individual language versions begins with MTSL 1.5+MC in Section A.2. Then, all previous versions are presented in descending order of their version numbers in Sections A.3 to A.5. While reading this appendix, please note that syntax definitions from earlier sections supersede those from later sections.

A.1 Common Definitions

The available language versions are built upon a set of common definitions, which are presented below. In order to ensure a seamless integration into the extended MULTIC methodology, a number of general modifications compared to the previously published versions of MTSL 1.0, MTSL 1.5, and MTSL 1.5+Modes have been applied:

1. Inclusion of a top-level syntactical element called $\langle \textit{TimingContract} \rangle$ that represents an *Assume/Guarantee* (A/G) contract consisting of a number of assumptions and guarantees formulated using MTSL specification patterns (see Section A.1.1)
2. Definition of a $\langle \textit{TimingSpec} \rangle$ element to express the logical values *true* and *false* as well as lists of $\langle \textit{TimingPattern} \rangle$ elements to add explicit language support for composing multi-part assumptions and guarantees (see Section A.1.1)
3. Integration of a simple type system for event values, supporting string and integer values, for example, to specify data and mode dependencies and reason about their evolution over time (see Sections A.1.2 to A.1.5)

A.1.1 Timing Contracts

The structure of the $\langle \textit{TimingContract} \rangle$ element is detailed in terms of the high-level syntax rules of the MTSL. The syntax allows to define A/G timing contracts, consisting of assumptions and guarantees. Both parts of the specification can be formulated using the logical values *true* or *false*, or a set of MTSL specification patterns.

$$\begin{aligned}
 \langle \textit{TimingContract} \rangle &::= \langle \textit{Assumption} \rangle \langle \textit{Guarantee} \rangle \\
 \langle \textit{Assumption} \rangle &::= \text{'A:'} \langle \textit{TimingSpec} \rangle \\
 \langle \textit{Guarantee} \rangle &::= \text{'G:'} \langle \textit{TimingSpec} \rangle \\
 \langle \textit{TimingSpec} \rangle &::= \text{'True.'} \mid \text{'False.'} \\
 &\quad \mid \langle \textit{TimingPattern} \rangle [\langle \textit{TimingPattern} \rangle]^*
 \end{aligned}$$

Note that $\langle \textit{TimingPattern} \rangle$ elements that are used as part of assumptions may only contain references to input ports, while guarantees may reference both input and output ports. For this reason, event relations (i.e., $\langle \textit{Reaction} \rangle$, $\langle \textit{Age} \rangle$, $\langle \textit{CausalReaction} \rangle$, and $\langle \textit{CausalAge} \rangle$) can only be specified in guarantees. $\langle \textit{ModeMapping} \rangle$ patterns (MTSL 1.5+Modes and higher) are also intended for use in guarantees. $\langle \textit{ClockDefinition} \rangle$ patterns (MTSL 1.0 and MTSL 1.5+MC), on the other hand, are used in assumptions, since “clocks are typically not controlled by the respective component” [93, p. 23].

A.1.2 Intervals and Time Expressions

All MTSL versions use intervals and time expressions as parameters of specification patterns. Note that the original $\langle \textit{Value} \rangle$ element (MTSL 1.0 and MTSL 1.5) has been renamed to $\langle \textit{TimeValue} \rangle$ (MTSL 1.5+Modes and higher) to clearly distinguish between time and event values. Additionally, round parentheses “(” and “)” have been introduced as alternative symbols for specifying open interval bounds in MTSL 1.5.

$\langle \textit{Interval} \rangle$	$::= \langle \textit{TimeExpr} \rangle \mid \langle \textit{Boundary} \rangle \langle \textit{TimeValue} \rangle \text{' , ' } \langle \textit{TimeValue} \rangle \langle \textit{Boundary} \rangle \langle \textit{TimeUnit} \rangle$
$\langle \textit{TimeExpr} \rangle$	$::= \langle \textit{TimeValue} \rangle \langle \textit{TimeUnit} \rangle$
$\langle \textit{Boundary} \rangle$	$::= \text{' [' } \mid \text{'] ' } \mid \text{' (' } \mid \text{') ' }$
$\langle \textit{TimeValue} \rangle$	$::= \langle \textit{Number} \rangle \mid \langle \textit{Number} \rangle \text{' . ' } \langle \textit{Number} \rangle$
$\langle \textit{TimeUnit} \rangle$	$::= \text{' s ' } \mid \text{' ms ' } \mid \text{' us ' } \mid \text{' ns ' }$

A.1.3 Factor Expressions

Another common element are factor expressions, which can be used to specify drift values as part of $\langle \textit{ClockDefinition} \rangle$ patterns (MTSL 1.0 and MTSL 1.5+MC). These values are specified using $\langle \textit{FactorInterval} \rangle$ elements, which also support point intervals.

$\langle \textit{FactorInterval} \rangle$	$::= \langle \textit{Factor} \rangle \mid \langle \textit{Boundary} \rangle \langle \textit{Factor} \rangle \text{' , ' } \langle \textit{Factor} \rangle \langle \textit{Boundary} \rangle$
$\langle \textit{Factor} \rangle$	$::= \langle \textit{Number} \rangle \text{' . ' } \langle \textit{Number} \rangle$

A.1.4 Basic Expressions

A set of basic expressions allows the definition of component, port, mode, and clock names. Additionally, event values, which represent either a $\langle \textit{Name} \rangle$ or a $\langle \textit{Number} \rangle$, are defined.

A Timing Specification Language

$\langle ComponentName \rangle ::= \langle Name \rangle$
 $\langle PortName \rangle ::= \langle Name \rangle$
 $\langle EventValue \rangle ::= \langle Name \rangle \mid \langle Number \rangle$
 $\langle ModeName \rangle ::= \langle Name \rangle$
 $\langle ClockName \rangle ::= \langle Name \rangle$

A.1.5 Data Types

The basic data types for names and numbers are defined in terms of corresponding $\langle Name \rangle$ and $\langle Number \rangle$ elements. They are used in most higher-level syntax definitions.

$\langle Name \rangle ::= 'a' .. 'z' ['a' .. 'z']^*$
 $\langle Number \rangle ::= '0' .. '9' ['0' .. '9']^*$

A.2 MTSL 1.5+MC

The *MTSL 1.5+Modes+Clocks* (MTSL 1.5+MC) specification language is one of the main results of this thesis. The specification language includes consistent extensions of MTSL 1.5 that adds support for specifying operating modes, mode transitions, and mode-dependent timing behavior (see Chapter 4) as well as local clocks with typical properties (see Chapter 6). Due to the iterative development of the specification language, the extensions for handling operating modes were published before the submission of this thesis [370, 371]. The resulting intermediate version of the specification language was retrospectively named MTSL 1.5+Modes in order to be able to distinguish between the different work statuses. Therefore, both versions are listed separately in this appendix. However, as MTSL 1.5+Modes is a subset of MTSL 1.5+MC, the current version of the specification language is referred to as MTSL 1.5+MC in the context of this thesis.

A.2.1 Specification Patterns

MTSL 1.5+MC supports the specification of $\langle Repetition \rangle$, $\langle Reaction \rangle$, $\langle Age \rangle$, $\langle ModeMapping \rangle$, and $\langle ClockDefinition \rangle$ patterns. In addition, single event occurrences can be represented by using $\langle Repetition \rangle$ patterns with an $\langle Offset \rangle$ interval (see Section A.4.3) that specifies the valid time of occurrence of the first (and only) event and a period of ∞ to prevent subsequent event occurrences. Furthermore, sporadic events can be specified by defining $\langle Repetition \rangle$ patterns with a period of $(0, \infty)$ and a default offset.

$\langle \text{TimingPattern} \rangle$	$::= \langle \text{Repetition} \rangle$ $ \langle \text{Reaction} \rangle \langle \text{Age} \rangle$ $ \langle \text{ModeMapping} \rangle$ $ \langle \text{ClockDefinition} \rangle$
$\langle \text{Repetition} \rangle$	$::= \text{'Repetition'} (\langle \text{Event} \rangle \langle \text{EventSet} \rangle) \text{' each' } \langle \text{Interval} \rangle [$ $\text{'with' } \langle \text{EOModalities} \rangle]? [\langle \text{ModeCondition} \rangle]? [$ $\langle \text{ClockReference} \rangle]? \text{'.'}$ $ (\langle \text{Event} \rangle \langle \text{EventSet} \rangle) \text{' occurs every' } \langle \text{Interval} \rangle [\text{'with'}$ $\langle \text{EOModalities} \rangle]? [\langle \text{ModeCondition} \rangle]? [\langle \text{ClockReference} \rangle]?$ '.'
$\langle \text{Reaction} \rangle$	$::= \text{'Reaction'} (\langle \text{EventSpec} \rangle \text{' , ' } \langle \text{EventSpec} \rangle \text{' }) [\langle \text{TRModalities} \rangle]?$ $\text{' within' } \langle \text{Interval} \rangle [\langle \text{TRModalities} \rangle]? [\langle \text{ModeCondition} \rangle]? [$ $\langle \text{ClockReference} \rangle]? \text{'.'}$ $ \text{' Whenever' } \langle \text{EventSpec} \rangle \text{' occurs then' } \langle \text{EventSpec} \rangle$ $\text{' occurs within' } \langle \text{Interval} \rangle [\langle \text{TRModalities} \rangle]? [$ $\langle \text{ModeCondition} \rangle]? [\langle \text{ClockReference} \rangle]? \text{'.'}$
$\langle \text{Age} \rangle$	$::= \text{'Age'} (\langle \text{EventSpec} \rangle \text{' , ' } \langle \text{EventSpec} \rangle \text{' }) [\langle \text{TRModalities} \rangle]?$ $\text{' within' } \langle \text{Interval} \rangle [\langle \text{TRModalities} \rangle]? [\langle \text{ModeCondition} \rangle]? [$ $\langle \text{ClockReference} \rangle]? \text{'.'}$ $ \text{' Whenever' } \langle \text{EventSpec} \rangle \text{' occurs then' } \langle \text{EventSpec} \rangle$ $\text{' has occurred within' } \langle \text{Interval} \rangle [\langle \text{TRModalities} \rangle]? [$ $\langle \text{ModeCondition} \rangle]? [\langle \text{ClockReference} \rangle]? \text{'.'}$
$\langle \text{ModeMapping} \rangle$	$::= \langle \text{CompModeConfig} \rangle \text{' maps to' } \langle \text{Mode} \rangle \text{'.'}$
$\langle \text{ClockDefinition} \rangle$	$::= \text{'Clock' } \langle \text{ClockName} \rangle \text{' has' } [\text{' resolution' } \langle \text{TimeExpr} \rangle]? [$ $\text{' drift' } \langle \text{FactorInterval} \rangle]? [\text{' skew' } \langle \text{TimeExpr} \rangle]? [\text{' maxdiff'}$ $\langle \text{TimeExpr} \rangle]? \text{'.'}$

A.2.2 Clock References

The optional use of $\langle \text{ClockReference} \rangle$ pattern extensions allows to bind the semantics of specification patterns to the time domain of the referenced local clock. Note that a $\langle \text{ClockDefinition} \rangle$ pattern must be used to define a clock with the same $\langle \text{ClockName} \rangle$.

$\langle \text{ClockReference} \rangle$	$::= \text{'using clock' } \langle \text{ClockName} \rangle$

A.3 MTSL 1.5+Modes

The MTSL 1.5+Modes specification language is defined as a subset of MTSL 1.5+MC and comprises a substantial part of the language extensions developed in the context of this work. Most of the concepts have been published in previous publications [370, 371]. The current version revises the offset semantics of the $\langle \text{Repetition} \rangle$ pattern, introduces a

mode-dependent version of the $\langle Age \rangle$ pattern, and defines a $\langle ModeMapping \rangle$ pattern as a tool for specifying mappings between lower-level and higher-level modes.

A.3.1 Specification Patterns

MTSL 1.5+Modes supports the specification of $\langle Repetition \rangle$, $\langle Reaction \rangle$, $\langle Age \rangle$, and $\langle ModeMapping \rangle$ patterns. The definitions of the first three patterns, i.e., $\langle Repetition \rangle$, $\langle Reaction \rangle$, and $\langle Age \rangle$, are almost identical to those in MTSL 1.5. However, the patterns were enhanced to include an optional $\langle ModeCondition \rangle$ that restricts the validity of the timing specification to a given set of modes. Note that the semantics of the first three patterns is defined in Section 4.2.4. Further information about the semantics and practical use of $\langle ModeMapping \rangle$ patterns can be found in Sections 4.2.3 and 4.2.5.

$\langle TimingPattern \rangle$	$::= \langle Repetition \rangle$ $\langle Reaction \rangle$ $\langle Age \rangle$ $\langle ModeMapping \rangle$
$\langle Repetition \rangle$	$::= \text{'Repetition' } (\langle Event \rangle \langle EventSet \rangle) \text{' each' } \langle Interval \rangle [$ $\text{'with' } \langle EOModalities \rangle]? [\langle ModeCondition \rangle]? \text{'.'}$ $ (\langle Event \rangle \langle EventSet \rangle) \text{' occurs every' } \langle Interval \rangle [\text{'with'}$ $\langle EOModalities \rangle]? [\langle ModeCondition \rangle]? \text{'.'}$
$\langle Reaction \rangle$	$::= \text{'Reaction' } \langle EventSpec \rangle \text{' , ' } \langle EventSpec \rangle \text{')' } [\langle TRModalities \rangle]?$ $\text{'within' } \langle Interval \rangle [\langle TRModalities \rangle]? [\langle ModeCondition \rangle]? \text{'.'}$ $ \text{'Whenever' } \langle EventSpec \rangle \text{' occurs then' } \langle EventSpec \rangle$ $\text{'occurs within' } \langle Interval \rangle [\langle TRModalities \rangle]? [$ $\langle ModeCondition \rangle]? \text{'.'}$
$\langle Age \rangle$	$::= \text{'Age' } \langle EventSpec \rangle \text{' , ' } \langle EventSpec \rangle \text{')' } [\langle TRModalities \rangle]?$ $\text{'within' } \langle Interval \rangle [\langle TRModalities \rangle]? [\langle ModeCondition \rangle]? \text{'.'}$ $ \text{'Whenever' } \langle EventSpec \rangle \text{' occurs then' } \langle EventSpec \rangle$ $\text{'has occurred within' } \langle Interval \rangle [\langle TRModalities \rangle]? [$ $\langle ModeCondition \rangle]? \text{'.'}$
$\langle ModeMapping \rangle$	$::= \langle CompModeConfig \rangle \text{' maps to' } \langle Mode \rangle \text{'.'}$

A.3.2 Event and Variable Expressions

In order to provide support for *set()* and *change()* events, the syntax definitions for MTSL 1.5 event specifications (see Section A.4.2) were extended. To this end, syntax rules that allow referencing the two virtual events were included. The $\langle ValueSet \rangle$ element enables the specification of non-deterministic selections from a set of possible values.

$\langle Event \rangle ::= \langle Port \rangle [\text{'->'} \langle EventValue \rangle]?$
 $\quad | \text{'set('} \langle Port \rangle [\text{' ,' } \langle ValueSet \rangle]? \text{')'}$
 $\quad | \text{'change('} \langle Port \rangle [\text{' ,' } \langle ValueSet \rangle]? \text{')'}$
 $\langle ValueSet \rangle ::= \text{'{' } \langle ValueList \rangle \text{'}' } | \langle Value \rangle$
 $\langle ValueList \rangle ::= \langle Value \rangle [\text{' ,' } \langle ValueList \rangle]?$

A.3.3 Mode Conditions

The $\langle ModeCondition \rangle$ element can be used as an optional extension of all timing-related MTSL 1.5+Modes and MTSL 1.5+MC specification patterns. It allows the specification of a set of valid modes as well as corresponding pre- and post-phases. Consequently, the $\langle PrePhases \rangle$ and $\langle PostPhases \rangle$ elements enable the specification of (fine-grained) pre- and post-phase durations for subsets of the available component modes (see Section 4.2.4).

$\langle ModeCondition \rangle ::= \text{'in mode'} \langle ModeSet \rangle [\langle PrePhases \rangle]? [\langle PostPhases \rangle]?$
 $\langle ModeSet \rangle ::= \text{'{' } \langle ModeList \rangle \text{'}' } | \langle Mode \rangle$
 $\langle ModeList \rangle ::= \langle Mode \rangle [\text{' ,' } \langle ModeList \rangle]?$
 $\langle Mode \rangle ::= \langle ModeName \rangle$
 $\langle PrePhases \rangle ::= \text{'pre'} \langle PhaseList \rangle$
 $\langle PostPhases \rangle ::= \text{'post'} \langle PhaseList \rangle$
 $\langle PhaseList \rangle ::= \langle PhaseDuration \rangle [\langle ModeSet \rangle]? [\text{' ,' } \langle PhaseList \rangle]?$
 $\langle PhaseDuration \rangle ::= \langle TimeExpr \rangle$

A.3.4 Mode Configurations

As part of the $\langle ModeMapping \rangle$ pattern, mode configurations are used to specify (sets of) lower-level modes that are mapped to a given higher-level mode (see Section 4.2.3). Each $\langle CompMode \rangle$ element consists of a $\langle ComponentName \rangle$ and a $\langle ModeName \rangle$, which are defined according to the basic expressions introduced in Section A.1.4.

$\langle CompModeConfig \rangle ::= \text{'{' } \langle CompModeList \rangle \text{'}' } | \langle CompMode \rangle$
 $\langle CompModeList \rangle ::= \langle CompMode \rangle [\text{' ,' } \langle CompModeList \rangle]?$
 $\langle CompMode \rangle ::= \langle ComponentName \rangle \text{'.' } \langle ModeName \rangle$

A.4 MTSL 1.5

The MTSL 1.5 specification language [559] was defined as the successor of MTSL 1.0. It aims to improve the applicability and implementability of core specification patterns by refining their syntax and semantics. Since the semantics of MTSL 1.0 patterns for specifying causal event relations proved impractical during implementation, the corresponding $\langle CausalReaction \rangle$ and $\langle CausalAge \rangle$ patterns were removed. Additionally, event expressions (see Section A.5.2) were replaced by more expressive event specifications (see Section A.4.2) that support a more diverse set of applications.

A.4.1 Specification Patterns

MTSL 1.5 supports the specification of $\langle Repetition \rangle$, $\langle Reaction \rangle$, and $\langle Age \rangle$ patterns. With this selection, the specification language returns to its core competencies. At the same time, the expressive power of the remaining specification patterns has increased due to the introduction of new parameters and modalities (see Sections A.4.3 and A.4.4).

$\langle TimingPattern \rangle$	$::= \langle Repetition \rangle$ $\langle Reaction \rangle$ $\langle Age \rangle$
$\langle Repetition \rangle$	$::= \text{'Repetition'} (\langle Event \rangle \langle EventSet \rangle) \text{' each' } \langle Interval \rangle [$ $\text{'with' } \langle EOModalities \rangle]? \text{'.'}$ $(\langle Event \rangle \langle EventSet \rangle) \text{' occurs every' } \langle Interval \rangle [\text{'with'}$ $\langle EOModalities \rangle]? \text{'.'}$
$\langle Reaction \rangle$	$::= \text{'Reaction'} (\langle EventSpec \rangle \text{' , ' } \langle EventSpec \rangle \text{')' } [\langle TRModalities \rangle]?$ $\text{'within' } \langle Interval \rangle [\langle TRModalities \rangle]? \text{'.'}$ $\text{'Whenever' } \langle EventSpec \rangle \text{' occurs then' } \langle EventSpec \rangle$ $\text{' occurs within' } \langle Interval \rangle [\langle TRModalities \rangle]? \text{'.'}$
$\langle Age \rangle$	$::= \text{'Age'} (\langle EventSpec \rangle \text{' , ' } \langle EventSpec \rangle \text{')' } [\langle TRModalities \rangle]?$ $\text{'within' } \langle Interval \rangle [\langle TRModalities \rangle]? \text{'.'}$ $\text{'Whenever' } \langle EventSpec \rangle \text{' occurs then' } \langle EventSpec \rangle$ $\text{' has occurred within' } \langle Interval \rangle [\langle TRModalities \rangle]? \text{'.'}$

A.4.2 Event Specifications¹

MTSL 1.5 timing specifications can be formulated using various types of event specifications, whose syntax is defined in terms of an $\langle EventSpec \rangle$ element. Each $\langle Event \rangle$ represents a single event. $\langle EventSet \rangle$ elements can be used to specify event sets, i.e., a set of events that occur at the same time. The other event specifications, i.e., $\langle EventSelection \rangle$, $\langle EventSequence \rangle$, and $\langle EventCoregion \rangle$, specify arbitrary subsets of a given list of event sets, sequences of events with a fixed order, or event sequences with arbitrary orders.

¹An earlier version of this section has been prepared as part of [559]. The results are the outcome of collaboration with Ingo Stierand, Janis Kröger, Frank Poppen, Boris Wirtz, and Rolf Schmedes.

$\langle EventSpec \rangle$	$::= \langle Event \rangle \mid \langle EventSet \rangle \mid \langle EventSelection \rangle \mid \langle EventSequence \rangle \mid \langle EventCoregion \rangle$
$\langle Event \rangle$	$::= \langle Port \rangle [\text{'->'} \langle EventValue \rangle]?$
$\langle Port \rangle$	$::= \langle PortName \rangle \mid \langle ComponentName \rangle \text{'.'} \langle PortName \rangle$
$\langle EventSet \rangle$	$::= \text{'{' } \langle Events \rangle \text{'}'}$
$\langle Events \rangle$	$::= \langle Event \rangle [\text{' ,' } \langle Events \rangle]?$
$\langle EventSelection \rangle$	$::= \text{'{' } \langle EventSelectionList \rangle \text{'}'}$
$\langle EventSelectionList \rangle$	$::= \langle EventSet \rangle \mid \langle EventSet \rangle \text{' ,' } \langle EventList \rangle \mid \langle Event \rangle \text{' ,' } \langle EventSelectionList \rangle$
$\langle EventList \rangle$	$::= (\langle Event \rangle \mid \langle EventSet \rangle) [\text{' ,' } \langle EventList \rangle]?$
$\langle EventSequence \rangle$	$::= \text{'(' } \langle EventList \rangle \text{'}'}$
$\langle EventCoregion \rangle$	$::= \text{'[' } \langle EventList \rangle \text{'}'}$

In the following, we define the semantics of event specifications based on the definitions from Sections 2.3.1 and 4.2.2. Recall that we have defined timed traces as infinite sequences $\omega = (\sigma_i, t_i)_{i \in \mathbb{N}}$. In a first step, we introduce a finite ordered list of indices $K = \{k_1, \dots, k_n\}$ with $k_i \in \mathbb{N}$ and $i < j \implies k_i < k_j$. Based on this set, we define *indexed timed traces* $\omega|_K = (\sigma_{k_1}, t_{k_1}) \dots (\sigma_{k_n}, t_{k_n})$, which include a finite number of events.

Let es be an *event specification* defined by a syntactical $\langle EventSpec \rangle$ element. In this context, we say that es *matches* $\omega|_K$, denoted by $es \models \omega|_K$, if and only if the corresponding condition for the given type of event specification is fulfilled:

1. If es is an *event*, i.e., $es = e$:

$$e = \sigma_{k_1} \tag{A.1}$$

2. If es is an *event set*, i.e., $es = \{e_1, \dots, e_n\}$:

$$[e_1, \dots, e_n] \models \omega|_K \wedge \forall i, j \in \mathbb{N} : t_{k_i} = t_{k_j} \tag{A.2}$$

3. If es is an *event selection*, i.e., $es = \{\{e_{11}, \dots, e_{1n_1}\}, \dots, \{e_{m1}, \dots, e_{mn_m}\}\}$:

$$\text{largest set } \{e_{k_1}, \dots, e_{k_n}\} \in es : [e_{k_1}, \dots, e_{k_n}] \models \omega|_K \wedge \forall i, j \in \mathbb{N} : t_{k_i} = t_{k_j} \tag{A.3}$$

4. If es is an *event sequence*, i.e., $es = (e_1, \dots, e_n)$:

$$\forall i \in \mathbb{N} : e_i = \sigma_{k_i} \tag{A.4}$$

5. If es is an *event coregion*, i.e., $es = [e_1, \dots, e_n]$:

$$\text{exists any ordering } (e_{o1}, \dots, e_{on}) \models \omega|_K \tag{A.5}$$

A Timing Specification Language

Note that $n, m \in \mathbb{N}$, $1 \leq i \leq n$, and $1 \leq j \leq n$ apply in Equations A.1 to A.5. The set of *valid event specifications* that comply with one of the conditions is denoted by $\mathcal{ES}$.

Based on the previous definitions, we are able to derive an *event specification trace* $(\sigma_i^{es}, t_i)_{i \in \mathbb{N}}$ for each timed trace $\omega = (\sigma_i, t_i)_{i \in \mathbb{N}}$ and event specification es so that

1. σ_i^{es} is a *virtual event* that occurs at a virtual event port $p_{virt}(\sigma_i^{es}) \in P_{in/out}$.
2. For all (es_i, t_i) exists K_i such that $es \models \omega|_{K_i}$ and

$$\omega|_{K_i} = (\sigma_{k_1^i}, t_{k_1^i}) \dots (\sigma_{k_n^i}, t_{k_n^i}) \implies t_i = t_{k_n^i}. \quad (\text{A.6})$$

Additionally, we define an operator that indicates the relationship between event specifications and virtual events. We say, *es triggers σ^{virt}* , written $es \rightsquigarrow \sigma^{virt}$, with $es \in \mathcal{ES}$ and $\sigma^{virt} \in \Sigma$, if and only if $\sigma^{virt} = \sigma^{es}$. Note that σ^{virt} occurs whenever es matches, i.e., at all $t_i \in \mathbb{T}$ from the event specification trace $(\sigma_i^{es}, t_i)_{i \in \mathbb{N}}$. More details about the interpretation of event specifications can be found in [559].

A.4.3 Event Occurrence Modalities

The $\langle \text{RepetitionOptions} \rangle$ element (see Section A.5.3) was replaced by corresponding $\langle \text{EOModalities} \rangle$. The element continues to support specifying $\langle \text{Offset} \rangle$ and $\langle \text{Jitter} \rangle$, but relies on a more flexible syntax. Additionally, the offset semantics has been revised, i.e., the default offset has been set from $[0, \max(P)]$ to $[0, 0]$. Using the **offset** keyword without an $\langle \text{Interval} \rangle$ specifies an arbitrary (finite) offset of the first event occurrence.

$$\begin{aligned} \langle \text{EOModalities} \rangle &::= \langle \text{EOModality} \rangle \mid \text{'and'} \langle \text{EOModalities} \rangle \mid ? \\ \langle \text{EOModality} \rangle &::= \langle \text{Offset} \rangle \mid \langle \text{Jitter} \rangle \\ \langle \text{Offset} \rangle &::= \text{'offset'} \mid \langle \text{Interval} \rangle \mid ? \\ \langle \text{Jitter} \rangle &::= \text{'jitter'} \mid \langle \text{TimeExpr} \rangle \end{aligned}$$

A.4.4 Timed Relation Modalities

Analogously, modalities for $\langle \text{Reaction} \rangle$ and $\langle \text{Age} \rangle$ patterns can be used. The available options allow the specification of simple quantitative properties (“K out of N”), the enforcing of single reactions (“**once**”), and the definition of initiation phases (“**init**”).

$$\begin{aligned} \langle \text{TRModalities} \rangle &::= \langle \text{TRModality} \rangle \mid \text{'and'} \langle \text{TRModalities} \rangle \mid ? \\ \langle \text{TRModality} \rangle &::= \langle \text{Restricted} \rangle \mid \langle \text{Once} \rangle \mid \langle \text{Init} \rangle \\ \langle \text{Restricted} \rangle &::= \langle \text{Number} \rangle \text{'out of'} \langle \text{Number} \rangle \mid \text{'times'} \mid ? \\ \langle \text{Once} \rangle &::= \text{'once'} \\ \langle \text{Init} \rangle &::= \text{'init'} \mid \langle \text{TimeExpr} \rangle \end{aligned}$$

The semantics of the $\langle \textit{Restricted} \rangle$ modality, i.e., K out of N pattern extensions, is defined based on the semantics of the $\langle \textit{Reaction} \rangle$ and $\langle \textit{Age} \rangle$ patterns given in Equations 4.17 and 4.19. The default interpretation, also referred to as *probabilistic* interpretation, expresses that the event relation must apply with at least a specified probability [559].

Let $k, n \in \mathbb{N}$ with $k \leq n$ be the $\langle \textit{Number} \rangle$ parameters of the $\langle \textit{Restricted} \rangle$ element. Furthermore, we define a *probability function* $P : \Psi \rightarrow [0, 1]$ that assigns a probability to each condition $\psi \in \Psi$, i.e., $P(\psi) := \Pr(\{\omega \in \Omega \mid \omega \models \psi\})$, where Ψ is a set of *conditions* and $\Pr$ is the *probability measure* of $(\Omega, \mathcal{F}, \Pr)$ with $\mathcal{F} \subseteq 2^\Omega$. Based on this function, the semantics of the restricted $\langle \textit{Reaction} \rangle$ pattern is defined by $L_{rea}^{k/n}(\sigma_1, \sigma_2, I, k, n)$:

$$L_{rea}^{k/n} = \left\{ (\sigma_i, t_i)_{i \in \mathbb{N}} \mid P(\forall(\sigma_1, t_i) \exists(\sigma_2, t_j) : t_j - t_i \in I) \geq \frac{k}{n} \right\} \quad (\text{A.7})$$

Analogously, the semantics of the restricted $\langle \textit{Age} \rangle$ pattern is defined by $L_{age}^{k/n}(\sigma_1, \sigma_2, I, k, n)$:

$$L_{age}^{k/n} = \left\{ (\sigma_i, t_i)_{i \in \mathbb{N}} \mid P(\forall(\sigma_2, t_i) \exists(\sigma_1, t_j) : t_j - t_i \in I) \geq \frac{k}{n} \right\} \quad (\text{A.8})$$

As an alternative to Equations A.7 and A.8, we retain the MTSL 1.0 semantics [93, pp. 16–17] by introducing a second interpretation, referred to as *sliding window* interpretation. In this case, the “restriction [...] is interpreted as a sliding window, which must hold for any sequence of n matching occurrences of $[\sigma_2]$ at least k times” [93, pp. 16–17]. The introduction of the alternative interpretation ensures backward compatibility while enabling engineers to select the semantics best suited to their specific use cases.

The semantics of the $\langle \textit{Once} \rangle$ modality is derived by replacing the existential quantifiers in Equations 4.17 and 4.19, or Equations A.7 and A.8, respectively, by unique existential quantifiers. Finally, the semantics of the $\langle \textit{Init} \rangle$ modality is defined by restricting the value of t_i to $t_i \geq t_{init}$, where $t_{init} \in \mathbb{T}$ is the parameter of the $\langle \textit{Init} \rangle$ element.

A.5 MTSL 1.0

The original version of the specification language, which is referred to as MTSL 1.0, was developed in the MULTIC [92] and MULTIC-Tooling [93] projects. The syntax and semantics have already been discussed in previous publications [92, 93, 164, 205]. For this reason, we focus on the necessary adaptations to align the MTSL versions.

A.5.1 Specification Patterns

MTSL 1.0 supports the specification of $\langle \textit{SingleEvent} \rangle$, $\langle \textit{Repetition} \rangle$, $\langle \textit{Reaction} \rangle$, $\langle \textit{Age} \rangle$, $\langle \textit{CausalReaction} \rangle$, $\langle \textit{CausalAge} \rangle$, $\langle \textit{CausalFuncDecl} \rangle$, and $\langle \textit{ClockDefinition} \rangle$ patterns. Comprehensive definitions of their semantics can be found in [92, 93]. Note that the syntax of $\langle \textit{Reaction} \rangle$ and $\langle \textit{Age} \rangle$ patterns has been slightly adapted compared to the original version [93, pp. 24–25], i.e., the keyword “**Whenever**” is capitalized for consistency reasons. In addition, the “**drift** $\langle \textit{Interval} \rangle$ ” subrule was replaced by “**drift** $\langle \textit{FactorInterval} \rangle$ ” to clearly distinguish between time and factor intervals (see Sections A.1.2 and A.1.3).

A Timing Specification Language

$\langle \text{TimingPattern} \rangle$	::= $\langle \text{SingleEvent} \rangle \mid \langle \text{Repetition} \rangle$ $\mid \langle \text{Reaction} \rangle \mid \langle \text{Age} \rangle$ $\mid \langle \text{CausalReaction} \rangle \mid \langle \text{CausalAge} \rangle$ $\mid \langle \text{CausalFuncDecl} \rangle$ $\mid \langle \text{ClockDefinition} \rangle$
$\langle \text{SingleEvent} \rangle$	::= $\langle \text{EventList} \rangle$ ‘occurs within’ $\langle \text{Interval} \rangle$ [‘using clock’ $\langle \text{ClockName} \rangle$]? ‘.’
$\langle \text{Repetition} \rangle$	::= $\langle \text{EventList} \rangle$ ‘occurs every’ $\langle \text{Interval} \rangle$ [‘with’ $\langle \text{RepetitionOptions} \rangle$]? [‘using clock’ $\langle \text{ClockName} \rangle$]? ‘.’
$\langle \text{Reaction} \rangle$	::= ‘Whenever’ $\langle \text{EventExpr} \rangle$ ‘occurs then’ $\langle \text{EventExpr} \rangle$ ‘occurs within’ $\langle \text{Interval} \rangle$ [‘once’]? [$\langle \text{Number} \rangle$ ‘out of’ $\langle \text{Number} \rangle$ ‘times’]? [‘using clock’ $\langle \text{ClockName} \rangle$]? ‘.’
$\langle \text{Age} \rangle$	::= ‘Whenever’ $\langle \text{EventExpr} \rangle$ ‘occurs then’ $\langle \text{EventExpr} \rangle$ ‘has occurred within’ $\langle \text{Interval} \rangle$ [‘once’]? [$\langle \text{Number} \rangle$ ‘out of’ $\langle \text{Number} \rangle$ ‘times’]? [‘using clock’ $\langle \text{ClockName} \rangle$]? ‘.’
$\langle \text{CausalReaction} \rangle$	::= ‘Reaction(’ $\langle \text{EventSpec} \rangle$ ‘,’ $\langle \text{EventSpec} \rangle$ ‘) within’ $\langle \text{Interval} \rangle$ [‘using clock’ $\langle \text{ClockName} \rangle$]? ‘.’
$\langle \text{CausalAge} \rangle$	::= ‘Age(’ $\langle \text{EventSpec} \rangle$ ‘,’ $\langle \text{EventSpec} \rangle$ ‘) within’ $\langle \text{Interval} \rangle$ [‘using clock’ $\langle \text{ClockName} \rangle$]? ‘.’
$\langle \text{CausalFuncDecl} \rangle$	::= $\langle \text{CausalFuncName} \rangle$ ‘(’ $\langle \text{Port} \rangle$ ‘,’ $\langle \text{Port} \rangle$ ‘) :=’ $\langle \text{CausalRelation} \rangle$
$\langle \text{ClockDefinition} \rangle$	::= ‘Clock’ $\langle \text{ClockName} \rangle$ ‘has’ [‘resolution’ $\langle \text{TimeExpr} \rangle$]? [‘drift’ $\langle \text{FactorInterval} \rangle$]? [‘skew’ $\langle \text{TimeExpr} \rangle$]? [‘maxdiff’ $\langle \text{TimeExpr} \rangle$]? ‘.’

A.5.2 Event Expressions, Events, and Ports

As a precursor to event specifications (see Section A.4.2), event expressions support the specification of events, event sequences, and event sets. Note that the syntactical element for separating port names and event values was replaced by “->” to clearly distinguish between “ $\langle \text{ComponentName} \rangle$ ‘.’ $\langle \text{PortName} \rangle$ ” and “ $\langle \text{Port} \rangle$ ‘->’ $\langle \text{EventValue} \rangle$ ”.

$\langle \text{EventExpr} \rangle$	::= $\langle \text{EventSpec} \rangle \mid \langle \text{EventList} \rangle \mid \{ \langle \text{EventList} \rangle \}$
$\langle \text{EventList} \rangle$	::= $\langle \text{EventSpec} \rangle$ [‘,’ $\langle \text{EventSpec} \rangle$]*
$\langle \text{EventSpec} \rangle$	::= $\langle \text{Port} \rangle$ [‘->’ $\langle \text{EventValue} \rangle$]?
$\langle \text{Port} \rangle$	::= $\langle \text{PortName} \rangle \mid \langle \text{ComponentName} \rangle$ ‘.’ $\langle \text{PortName} \rangle$

A.5.3 Repetition Options

The $\langle \text{RepetitionOptions} \rangle$ can be used to define additional parameters for $\langle \text{Repetition} \rangle$ patterns. The corresponding syntax rules served as the basis for defining event occurrence modalities (MTSL 1.5 and higher) in terms of the $\langle \text{EOModalities} \rangle$ element (see Section A.4.3). Note that the order in which the rules for $\langle \text{Offset} \rangle$ and $\langle \text{Jitter} \rangle$ are defined has been adapted to be consistent with the definitions of subsequent language versions.

$$\begin{aligned} \langle \text{RepetitionOptions} \rangle &::= \langle \text{Offset} \rangle [\text{'and'} \langle \text{Jitter} \rangle]? \mid \langle \text{Jitter} \rangle [\text{'and'} \langle \text{Offset} \rangle]? \\ \langle \text{Offset} \rangle &::= \text{'offset'} \langle \text{Interval} \rangle \\ \langle \text{Jitter} \rangle &::= \text{'jitter'} \langle \text{TimeExpr} \rangle \end{aligned}$$

A.5.4 Causal Relations

The use of $\langle \text{CausalReaction} \rangle$ and $\langle \text{CausalAge} \rangle$ patterns requires the definition of causal relations by means of $\langle \text{CausalFuncDecl} \rangle$ patterns. In this context, the $\langle \text{CausalFuncName} \rangle$ and $\langle \text{CausalRelation} \rangle$ parameters can be used to characterize such relations. Further information about the underlying semantics can be found in [93, pp. 17–20].

$$\begin{aligned} \langle \text{CausalFuncName} \rangle &::= \text{'|>'} \mid \text{'<|'} \\ \langle \text{CausalRelation} \rangle &::= \text{'FIFO'} \mid \text{'LIFO'} \mid \text{'ID'} \end{aligned}$$

B Modeling Guidelines

As part of this thesis, concepts, methods, and tools for ensuring real-time behavior of cooperative SoCPSs have been developed. The results were intended as building blocks that aim at establishing a SoCPS-wide timing coherence during system design. In all phases, priority was given to the question of how engineers can be supported in developing safe and reliable cooperative systems. In order to facilitate the transfer of the results into industrial practice, a set of consistent modeling guidelines has been developed.

In the following, the final version of the modeling guidelines is presented. The guidelines address relevant design activities from both phases of the envisioned design approach, i.e., the scenario-based activity modeling (see Section B.1) and the application of the extended MULTIC methodology (see Section B.2). In both cases, the formal concepts defined and used as part of this thesis were mapped to corresponding SysML elements. The guidelines can thus be used to create relevant design artifacts in the form of SysML models, facilitating the transfer of the proposed methodology to industrial design processes. All modeling guidelines were developed and tested using Enterprise Architect 17.

B.1 Time-Constrained Activity Diagrams

The heart of the envisioned design approach is the scenario-based activity modeling (see Section 3.3), which aims at capturing cooperative behavior in terms of *Time-Constrained Activity Diagrams* (TCADs). The formalism is based on a carefully selected set of basic elements of SysML activity diagrams [454] and earlier work by Jarraya et al. [332]. In previous chapters, we have already stated that automated tool support for modeling TCADs and translating them into system-level specifications would be desirable.

A first step in this direction was the definition of a set of consistent modeling guidelines that allow the SysML-compliant modeling of TCADs. The guidelines consist of three best practices and 17 mandatory guidelines. As a prerequisite, the higher-level *Time-Constrained Activity Model* (TCAM) can be represented by a top-level Enterprise Architect package that includes all relevant TCADs. In order to ensure clarity and comprehensibility of the resulting models, we recommend adhering to the following three best practices:

- BP-1** Prior to the modeling of TCADs, the initial top-level architecture *should* have been modeled according to the guidelines presented in Section B.2.
- BP-2** Each model *should* contain one *Activity Diagram* for each TCAD that defines all relevant modeling elements, except for constraints.
- BP-3** Each model *should* contain one *Requirement Diagram* for each TCAD that defines constraints and their allocation to relevant modeling elements.

In addition to the best practices, a total of 17 modeling guidelines were defined that map all TCAD modeling elements and constraints to corresponding SysML elements. Since there is currently no tool support for the automated translation of TCADs into system-level MTSL specifications, the results can be seen as an initial proposal that may be updated after implementation. For now, the presented guidelines enable the consistent modeling of all TCAD elements and constraints, as defined in Section 3.3.1.

- MG-1** Each TCAD *must* be represented by a SysML activity.
- MG-2** Each partition *must* be represented by a SysML (allocation activity) partition that allocates its elements to a block from the top-level architecture.
- MG-3** Each atomic action *must* be represented by a SysML action.
- MG-4** Each call behavior action *must* be represented by a SysML call behavior action that references a previously defined SysML activity.
- MG-5** Each accept event *must* be represented by a SysML accept event action.
- MG-6** Each time event *must* be represented by a SysML time event action.
- MG-7** Each signal *must* be represented by a SysML send signal action.
- MG-8** Each terminating signal *must* be represented by a SysML send signal action and a SysML final node, which are connected via a SysML control flow.
- MG-9** Each initial node *must* be represented by a SysML initial node.
- MG-10** Each final activity node *must* be represented by a SysML final node.
- MG-11** Each final flow node *must* be represented by a SysML final flow node.
- MG-12** Each decision node *must* be represented by a SysML decision node.
- MG-13** Each merge node *must* be represented by a SysML merge node.
- MG-14** Each fork node *must* be represented by a SysML fork node.
- MG-15** Each join node *must* be represented by a SysML join node.
- MG-16** Each control flow *must* be represented by a SysML control flow.
- MG-17** Each constraint *must* be represented by a SysML requirement.
 - MG-17.1** Each requirement *must* have a set of «*satisfy*» dependencies that assign the requirement to one or more actions, control nodes, or control flows, depending on the type of constraint.
 - MG-17.2** Each requirement *should* have exactly one timing or resource constraint specified as a string in the tagged value **text**.

B.2 Functional Architectures with Timing Specifications¹

The tool support developed in the context of this thesis enables the modeling of systems in terms of SysML+MTSL design models. In order to successfully perform VITs, the included modeling elements must adhere to a number of design rules. Based on the results of the MULTIC and MULTIC-Tooling projects [92, 93], we present consistent guidelines for modeling components, input, output, and variable ports, specifications, and signals. The guidelines form the basis for automated model validation procedures and model transformations, which have been implemented as part of the UACMbox Toolchain.

When modeling systems according to the extended MULTIC methodology, we recommend to consider three best practices. Although not mandatory, adhering to these practices improves the clarity and comprehensibility of the resulting models.

- BP-1** Each model *should* contain one *Block Definition Diagram* that defines components and their hierarchical structure.
- BP-2** Each model *should* contain one *Internal Block Diagram* for each decomposed component that defines component ports and signals between these ports.
- BP-3** Each model *should* contain one *Requirement Diagram* for each decomposed component that defines specifications and their allocation to components.

In order to obtain valid design models, a total of six modeling guidelines must be taken into account. These guidelines ensure that all elements of the formal system model, as introduced in Section 2.3.1 and extended in Section 4.2.2, are consistently mapped to SysML elements. Note that the tool support developed in Chapter 7 handles deviations and violations by displaying appropriate warnings or error messages.

- MG-1** Each component *must* be represented by a SysML block.
 - MG-1.1** Each block (except for the top-level block) *must* have a *part association* that defines the relation to the higher-level block.
 - MG-1.2** Each block *must* have at least one port.
 - MG-1.3** Each block *should* have at least one «*satisfy*» dependency that assigns a requirement to the block.
 - MG-1.4** Each block that has at least one mode-dependent specification *must* have a variable port called “Mode”.
- MG-2** Each port *must* be represented by a SysML proxy port.
 - MG-2.1** Each event port *must* have its **Type** set to “Event”.
 - MG-2.2** Each input port *must* have the option **isConjugated** enabled.
 - MG-2.3** Each output port *must* have its **isConjugated** disabled.

¹An earlier version of this section has been published in [93, Chapter 7]. The corresponding passage in the original publication was written by me and was finalized with editorial changes by my co-authors.

- MG-2.4** Each variable port *must* have its **Type** set to “Variable”.
- MG-2.5** Each variable port *must* have an initial value specified as a simple alphabetic string in the **Notes** field.
- MG-3** Each specification *must* be represented by a SysML requirement.
 - MG-3.1** Each requirement *must* have a set of «*satisfy*» dependencies that assign the requirement to one or more blocks.
 - MG-3.2** Each requirement *should* have exactly one $\langle \textit{TimingContract} \rangle$ element specified as a string in the tagged value **text**.
- MG-4** Each signal *must* be represented by a set of SysML binding connectors.
 - MG-4.1** Each event signal *must* be represented by a set of binding connectors, each of which connects two input or output ports.
 - MG-4.2** Each variable signal *must* be represented by a set of binding connectors, each of which connects two variable ports.
 - MG-4.3** Each input port of a higher-level block *must* be connected to one or more input ports of lower-level blocks.
 - MG-4.4** Each output port of a lower-level block *must* be connected to (1) input ports of blocks on the same hierarchy level, (2) output ports of the higher-level block, or (3) a combination of both.
- MG-5** Each model *must* have a single SysML interface block denoted as “Event”.
 - MG-5.1** The interface block “Event” *must* have exactly one SysML flow property assigned that is designated as “EventFlow”.
 - MG-5.2** The flow property “EventFlow” *must* have the attribute **Direction** set to **out**.
- MG-6** Each model that contains components with mode-dependent specifications *must* have a single SysML interface block denoted as “Variable”.
 - MG-6.1** The interface block “Variable” *must* have exactly one SysML flow property assigned that is designated as “VariableFlow”.
 - MG-6.2** The flow property “VariableFlow” *must* have the attribute **Direction** set to **inout**.

List of Figures

1.1	Levels of driving automation	2
1.2	Sequence of example driving scenarios	9
1.3	Example conflict areas	11
1.4	Cooperative left-turn maneuver	12
1.5	Time sequence of communication	15
2.1	Annotation of an A/G contract to a functional component	41
2.2	Composition and refinement of A/G contracts	43
2.3	Comparison of the accuracy of common analysis classes	45
2.4	Design approach according to the MULTIC methodology	54
2.5	Component model of the <i>Emergency Stop System</i>	54
2.6	Further decomposition of the <i>Emergency Stop System</i>	56
2.7	Example SystemC simulation model	57
2.8	Tool-supported specification and refinement of MTSL timing contracts . .	58
3.1	Envisioned design approach	71
3.2	TCAD modeling elements and constraint types	75
3.3	Top-level architecture of the <i>Cooperative Left-Turn Assistant</i>	86
3.4	Key scenarios relevant to the development of the CLTA	87
3.5	TCAD for the <i>Subscribe</i> activity	89
3.6	TCAD for the <i>Initiate and Evaluate</i> activity	90
3.7	TCAD for the <i>Decide and Execute</i> activity	91
3.8	TCAD for the <i>Unsubscribe</i> activity	92
3.9	Translation of duration-constrained control flows	97
3.10	Translation of cross-system control flows	98
3.11	Partitioned TCADs for the <i>Subscribe</i> activity	99
3.12	Partitioned TCADs for the <i>Initiate and Evaluate</i> activity	100
3.13	Partitioned TCADs for the <i>Unsubscribe</i> activity	101
4.1	Component model of the example system	134
4.2	Intended behavior of the example system	137
4.3	Specification pattern states and component modes	140
4.4	Repetition pattern examples	142
4.5	Example trace of a mode-dependent Repetition pattern	143
4.6	Reaction pattern examples	144
4.7	Example trace of a mode-dependent Reaction pattern	144
4.8	Examples of fine-grained pre- and post-phases	146

List of Figures

4.9	Component model of the <i>Longitudinal Control System</i>	148
4.10	Example traces of the <i>Longitudinal Control System</i>	153
4.11	Component model of the <i>Sensor Processing</i>	156
5.1	Procedure for systematic literature review	167
5.2	Number of included publications per group per year	168
5.3	Number of included publications per communication standard	169
5.4	Number of included publications per performance metric	170
5.5	Number of included publications per data source type	171
5.6	Processing steps with effects on temporal behavior	173
5.7	Component model of the generic communication model	175
6.1	Undesired observation effects due to local clocks	187
6.2	Translation of observations based on local clocks	189
6.3	Example of the snowball effect	190
6.4	Discretization effects and widenings for a finite-resolution clock	193
6.5	Faithful mapping of partitioned global specifications	194
6.6	Accumulating uncertainties due to clock drift	196
6.7	Additional delays due to clock skew	197
6.8	Symmetric widening due to difference constraints	199
6.9	Example specification intervals and underapproximations of φ_1	208
6.10	Example specification intervals and underapproximations of φ_2	209
7.1	UACMbox toolchain	218
7.2	UACM meta-model	219
7.3	Component model of the UACM example system	222
7.4	Architecture and interfaces of the MTI add-in	227
7.5	Extensions to the graphical user interface of Enterprise Architect	228
7.6	Graphical user interface of the MTI add-in	229
7.7	Component model of the <i>Tiny SysML Model</i>	230
7.8	SysML model of the <i>Tiny SysML Model</i>	231
7.9	Architecture and interfaces of the UACMbox	234
7.10	Graphical user interface of RTana _{2sim}	243

List of Tables

1.1	Proposed strategies for CHAV 1	16
1.2	Proposed strategies for CHAV 2	16
1.3	Initial payoff matrix	16
1.4	Half-filled payoff matrices resulting from maneuver evaluation	17
1.5	Merged payoff matrix for decision making	17
1.6	Differences between systems and SoSs	23
2.1	Comparison of existing frameworks	49
2.2	Top-level specification of the <i>Emergency Stop System</i>	55
2.3	Specification $\mathcal{C}_{ld}$ of the <i>Lane Detection</i> component	55
2.4	Specification $\mathcal{C}_{em}$ of the <i>Environment Model</i> component	55
2.5	Specification $\mathcal{C}_{tp}$ of the <i>Trajectory Planning</i> component	55
2.6	Specification $\mathcal{C}_{dc}$ of the <i>Dynamic Control</i> component	55
2.7	Research gaps and contributions	59
3.1	Pre- and postconditions	103
3.2	Mapping of events and signals to ports	105
3.3	Specification $\mathcal{C}_{chav}^{base}$ of the <i>CHAV 1/2</i> components	106
3.4	Specification $\mathcal{C}_{v2x}^{base}$ of the <i>V2X</i> component	106
3.5	Mapping of actions to system-level modes	108
3.6	Specification $\mathcal{C}_{chav}^{sub}$ of the <i>CHAV 1/2</i> components	117
3.7	Specification $\mathcal{C}_{v2x}^{sub}$ of the <i>V2X</i> component	117
3.8	Specification $\mathcal{C}_{tms}^{sub}$ of the <i>TMS</i> component	118
3.9	Specification $\mathcal{C}_{tms}^{init}$ of the <i>TMS</i> component	118
3.10	Specification $\mathcal{C}_{v2x}^{init/eval}$ of the <i>V2X</i> component	119
3.11	Specification $\mathcal{C}_{chav}^{eval}$ of the <i>CHAV 1/2</i> components	119
3.12	Specification $\mathcal{C}_{chav}^{dec/exec}$ of the <i>CHAV 1/2</i> components	120
3.13	Specification $\mathcal{C}_{chav}^{unsub}$ of the <i>CHAV 1/2</i> components	121
3.14	Specification $\mathcal{C}_{v2x}^{unsub}$ of the <i>V2X</i> component	121
3.15	Specification $\mathcal{C}_{tms}^{unsub}$ of the <i>TMS</i> component	121
4.1	Top-level specification of the <i>Longitudinal Control System</i>	149
4.2	Mode combinations and top-level modes	149
4.3	Excerpt from an example <i>Mode Logic</i> specification	150
4.4	Specification $\mathcal{C}_{acc}$ of the <i>Adaptive Cruise Control</i> component	151
4.5	Specification $\mathcal{C}_{ca}$ of the <i>Collision Avoidance</i> component	151
4.6	Specification $\mathcal{C}_{sl}$ of the <i>Switching Logic</i> component	152

List of Tables

4.7	Top-level specification of the <i>Sensor Processing</i>	156
4.8	Specification $\mathcal{C}_{pre}$ of the <i>Preprocessing</i> component	157
4.9	Specification $\mathcal{C}_{flt}$ of the <i>Filter</i> component	157
5.1	Mapping of performance metrics to communication phases	172
5.2	Specification template $\mathbb{C}_{send}$ for sending messages	176
5.3	Specification template $\mathbb{C}_{transmit}$ for transmitting messages	177
5.4	Specification template $\mathbb{C}_{receive}$ for receiving messages	178
5.5	Example parameter sets for IEEE 802.11p and C-V2X	180
6.1	Selected parameters of a state-of-the-art clock generator	209
7.1	Top-level specification of the <i>Tiny SysML Model</i>	230
7.2	Specification $\mathcal{C}_{suba}$ of the <i>SubSystemA</i> component	230
7.3	Specification $\mathcal{C}_{subb}$ of the <i>SubSystemB</i> component	230
7.4	Specification language support of different toolchain components	246
7.5	Example models used for performance evaluation	247
7.6	Execution times of model transformations	248
7.7	Analysis execution times on different platforms	249
7.8	Key figures related to analysis execution	249
A.1	Comparison of existing MTSL versions	261

List of Listings

7.1	UACM representation of the example system	223
7.2	UACM model of the <i>Tiny SysML Model</i>	232
7.3	RTana _{2sim} analysis model for the <i>Tiny SysML Model</i>	239
7.4	RTana _{2sim} analysis output for the <i>Tiny SysML Model</i>	244

Bibliography

- [1] 5G Automotive Association. *A Visionary Roadmap for Advanced Driving Use Cases, Connectivity Technologies, and Radio Spectrum Needs*. White Paper. Version 2.0. 2022. URL: https://5gaa.org/content/uploads/2022/11/5GAA_White-Paper_Roadmap_2022.pdf.
- [2] 5G Automotive Association. *V2X Functional and Performance Test Report; Test Procedures and Results*. Technical Report 5GAA P-190033. 2018. URL: https://5gaa.org/content/uploads/2018/11/5GAA_P-190033_V2X-Functional-and-Performance-Test-Report_final-1.pdf.
- [3] ANSYS, Inc. *ANSYS SCADE Suite*. <https://www.ansys.com/products/embedded-software/ansys-scade-suite>. Last accessed on June 19, 2026.
- [4] AUTOSAR GbR. *AUTOSAR Introduction*. Presentation. URL: https://www.autosar.org/fileadmin/user_upload/AUTOSAR_Introduction_PDF/AUTOSAR_EXP_Introduction_Part1.pdf.
- [5] AUTOSAR GbR. *Methodology for Adaptive Platform*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/AP/AUTOSAR_AP_TR_Methodology.pdf.
- [6] AUTOSAR GbR. *Methodology for Classic Platform*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_TR_Methodology.pdf.
- [7] AUTOSAR GbR. *Specification of Time Synchronization over CAN*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_TimeSyncOverCAN.pdf.
- [8] AUTOSAR GbR. *Specification of Time Synchronization over Ethernet*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_TimeSyncOverEthernet.pdf.
- [9] AUTOSAR GbR. *Specification of Time Synchronization over FlexRay*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_TimeSyncOverFlexRay.pdf.
- [10] AUTOSAR GbR. *Specification of Timing Extensions for Adaptive Platform*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/AP/AUTOSAR_AP_TPS_TimingExtensions.pdf.
- [11] AUTOSAR GbR. *Specification of Timing Extensions for Classic Platform*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_TPS_TimingExtensions.pdf.

Bibliography

- [12] AUTOSAR GbR. *Timing Analysis and Design*. Release R24-11. 2024. URL: https://www.autosar.org/fileadmin/standards/R24-11/F0/AUTOSAR_F0_TR_TimingAnalysis.pdf.
- [13] Daniel Abadi. “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story”. In: *Computer* 45.2 (2012), pp. 37–42. DOI: 10.1109/MC.2012.33.
- [14] Shimaa A. Abdel Hakeem, Anar A. Hady, and HyungWon Kim. “5G-V2X: Standardization, Architecture, Use Cases, Network-Slicing, and Edge-Computing”. In: *Wireless Networks* 26.8 (2020), pp. 6015–6041. DOI: 10.1007/s11276-020-02419-8.
- [15] Jaume Abella et al. “WCET Analysis Methods: Pitfalls and Challenges on Their Trustworthiness”. In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2015, pp. 1–10. DOI: 10.1109/SIES.2015.7185039.
- [16] Ankit Agrawal and Jane Cleland-Huang. “Leveraging Traceability to Integrate Safety Analysis Artifacts into the Software Development Process”. In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. IEEE, 2023, pp. 475–478. DOI: 10.1109/REW57809.2023.00089.
- [17] Assad Alam et al. “Heavy-Duty Vehicle Platooning for Sustainable Freight Transportation: A Cooperative Method to Enhance Safety and Efficiency”. In: *IEEE Control Systems Magazine* 35.6 (2015), pp. 34–56. DOI: 10.1109/MCS.2015.2471046.
- [18] Karsten Albers et al. *Design Handbook*. Deliverable D6.5. PANORAMA project, 2022. URL: <https://itea4.org/project/workpackage/document/download/8143/D6.5DesignHandbook.pdf>.
- [19] Alejandra Alfonso et al. “Visual Timed Event Scenarios”. In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE, 2004, pp. 168–177. DOI: 10.1109/ICSE.2004.1317439.
- [20] James F. Allen. “Maintaining Knowledge About Temporal Intervals”. In: *Communications of the ACM* 26.11 (1983), pp. 832–843. DOI: 10.1145/182.358434.
- [21] Abdel-Latif Alshalalfah and Otmane Ait Mohamed. “System-Level Modeling and Safety Analysis of Vehicular Coordinated Emergency Braking Under Degraded Wireless Connectivity Using Priced Timed Automata”. In: *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2020, pp. 1–4. DOI: 10.1109/ICECS49266.2020.9294965.
- [22] Rajeev Alur, Costas Courcoubetis, and David Dill. “Model-Checking in Dense Real-Time”. In: *Information and Computation* 104.1 (1993), pp. 2–34. DOI: 10.1006/inco.1993.1024.
- [23] Rajeev Alur and David L. Dill. “A Theory of Timed Automata”. In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

- [24] Rajeev Alur and David Dill. “Automata for Modeling Real-Time Systems”. In: *Automata, Languages and Programming*. Ed. by Michael S. Paterson. Lecture Notes in Computer Science 443. Springer, Berlin, Heidelberg, 1990, pp. 322–335. DOI: 10.1007/BFb0032042.
- [25] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. “The Benefits of Relaxing Punctuality”. In: *Journal of the ACM* 43.1 (1996), pp. 116–146. DOI: 10.1145/227595.227602.
- [26] Rajeev Alur and Thomas A. Henzinger. “A Really Temporal Logic”. In: *Journal of the ACM* 41.1 (1994), pp. 181–203. DOI: 10.1145/174644.174651.
- [27] Rajeev Alur and Thomas A. Henzinger. “Logics and Models of Real Time: A Survey”. In: *Real-Time: Theory in Practice*. Ed. by J. W. de Bakker et al. Lecture Notes in Computer Science 600. Springer, Berlin, Heidelberg, 1992, pp. 74–106. DOI: 10.1007/BFb0031988.
- [28] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. “An Analyzer for Message Sequence Charts”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science 1055. Springer, Berlin, Heidelberg, 1996, pp. 35–48. DOI: 10.1007/3-540-61042-1_37.
- [29] Y Amir et al. “Transis: A Communication Subsystem for High Availability”. In: *1992 FTCS – The Twenty-Second International Symposium on Fault-Tolerant Computing*. IEEE, 1992, pp. 76–77. DOI: 10.1109/FTCS.1992.243613.
- [30] Tobias Amnell et al. “TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems”. In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Kim Guldstrand Larsen and Peter Niebert. Lecture Notes in Computer Science 2791. Springer, Berlin, Heidelberg, 2004, pp. 60–72. DOI: 10.1007/978-3-540-40903-8_6.
- [31] Daniel Amyot and Gunter Mussbacher. “User Requirements Notation: The First Ten Years, the Next Ten Years”. In: *Journal of Software* 6.5 (2011), pp. 747–768. DOI: 10.4304/jsw.6.5.747-768.
- [32] Daniel Amyot et al. “Towards Improved Requirements Engineering With SysML and the User Requirements Notation”. In: *2016 IEEE 24th International Requirements Engineering Conference (RE)*. IEEE, 2016, pp. 329–334. DOI: 10.1109/RE.2016.58.
- [33] Ermeson Andrade et al. “A Methodology for Mapping SysML Activity Diagram to Time Petri Net for Requirement Validation of Embedded Real-Time Systems with Energy Constraints”. In: *2009 Third International Conference on Digital Society*. IEEE, 2009, pp. 266–271. DOI: 10.1109/ICDS.2009.19.
- [34] Charles André and Frédéric Mallet. *Clock Constraints in UML/MARTE CCSL*. Research Report RR-6540. Inria, 2008. URL: <https://inria.hal.science/inria-00280941v2/document>.

Bibliography

- [35] Charles André, Frédéric Mallet, and Robert de Simone. “Modeling Time(s)”. In: *Model Driven Engineering Languages and Systems*. Ed. by Gregor Engels et al. Lecture Notes in Computer Science 4735. Springer, Berlin, Heidelberg, 2007, pp. 559–573. DOI: 10.1007/978-3-540-75209-7_38.
- [36] Waqar Anwar et al. “PHY Abstraction Techniques for V2X Enabling Technologies: Modeling and Analysis”. In: *IEEE Transactions on Vehicular Technology* 70.2 (2021), pp. 1501–1517. DOI: 10.1109/TVT.2021.3053425.
- [37] Krzysztof Apt, Frank S. De Boer, and Ernst-Rüdiger Olderog. “Distributed Programs”. In: *Verification of Sequential and Concurrent Programs*. Springer, London, 2009, pp. 373–406. ISBN: 978-1-84882-745-5. DOI: 10.1007/978-1-84882-745-5.
- [38] Paolo Arcaini et al. “A Journey with ASMETA from Requirements to Code: Application to an Automotive System with Adaptive Features”. In: *International Journal on Software Tools for Technology Transfer* (2024), pp. 379–401. DOI: 10.1007/s10009-024-00751-4.
- [39] Arcticus Systems AB. *Rubus Tool Suite*. <https://www.arcticus-systems.com/products/rubus-tool-suite/>. Last accessed on June 19, 2026.
- [40] Koshal Arvind. “Probabilistic Clock Synchronization in Distributed Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 5.5 (1994), pp. 474–487. DOI: 10.1109/71.282558.
- [41] Rachad Atat et al. “Delay Efficient Cooperation in Public Safety Vehicular Networks Using LTE and IEEE 802.11p”. In: *2012 IEEE Consumer Communications and Networking Conference (CCNC)*. IEEE, 2012, pp. 316–320. DOI: 10.1109/CCNC.2012.6181109.
- [42] Audi AG. *Audi Expands Traffic Light Information – Now Includes Speed Recommendations to Minimize Stops*. <https://media.audiusa.com/releases/301>. Last accessed on June 19, 2026.
- [43] Audi AG. *Tech Talk: Audi, Traffic Light Information and the Future of What—and How—to Drive*. <https://media.audiusa.com/releases/412>. Last accessed on June 19, 2026.
- [44] Henrik Austad et al. “Composable Distributed Real-Time Systems with Deterministic Network Channels”. In: *Journal of Systems Architecture* 137 (2023), p. 102853. DOI: 10.1016/j.sysarc.2023.102853.
- [45] Marco Autili, Paola Inverardi, and Patrizio Pelliccione. “Graphical Scenarios for Specifying Temporal Properties: An Automated Approach”. In: *Automated Software Engineering* 14 (2007), pp. 293–340. DOI: 10.1007/s10515-007-0012-6.
- [46] Autotalks. *Functional Safety for Enabling Present and Future V2X Use-Cases*. White Paper. 2021. URL: <https://auto-talks.com/wp-content/uploads/2021/07/Functional-Safety-for-V2X-use-cases.pdf>.

- [47] Mohammad H. Azadmanesh and Roger M. Kieckhafer. “Exploiting Omissive Faults in Synchronous Approximate Agreement”. In: *IEEE Transactions on Computers* 49.10 (2002), pp. 1031–1042. DOI: 10.1109/12.888039.
- [48] BMW AG. *Level 3 Highly Automated Driving Available in the New BMW 7 Series from Next Spring*. <https://www.press.bmwgroup.com/global/article/detail/T0438214EN/level-3-highly-automated-driving-available-in-the-new-bmw-7-series-from-next-spring>. Last accessed on June 19, 2026.
- [49] BTC Embedded Systems AG. *BTC EmbeddedSpecifier. Intuitive Formal Specification of Safety Requirements*. <https://www.btc-embedded.com/products/btc-embeddedspecifier/>. Last accessed on June 19, 2026.
- [50] BTC Embedded Systems AG. *BTC EmbeddedValidator. The Power of Proof for Safety-Critical Systems*. <https://www.btc-embedded.com/products/btc-embeddedvalidator/>. Last accessed on June 19, 2026.
- [51] Abdelhakim Baouya et al. “A Quantitative Verification Framework of SysML Activity Diagrams Under Time Constraints”. In: *Expert Systems with Applications* 42.21 (2015), pp. 7493–7510. DOI: 10.1016/j.eswa.2015.05.049.
- [52] Stephan Baumgart, Joakim Fröberg, and Sasikumar Punnekkat. “How to Analyze the Safety of Concepts for a System-of-Systems?” In: *2021 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE, 2021, pp. 1–8. DOI: 10.1109/ISSE51541.2021.9582527.
- [53] Mahmoud Bazzal, Lukas Krawczyk, and Carsten Wolff. “RTCAnalysis: Practical Modular Performance Analysis of Automotive Systems with RTC”. In: *Information and Software Technologies*. Ed. by Audrius Lopata, Daina Gudonienė, and Rita Butkienė. Communications in Computer and Information Science 1486. Springer, Cham, 2021, pp. 209–223. DOI: 10.1007/978-3-030-88304-1_17.
- [54] Alessandro Bazzi. “Congestion Control Mechanisms in IEEE 802.11p and Sidelink C-V2X”. In: *2019 53rd Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2019, pp. 1125–1130. DOI: 10.1109/IEEECONF44664.2019.9048738.
- [55] Yosab Bebawy et al. “Incremental Contract-Based Verification of Software Updates for Safety-Critical Cyber-Physical Systems”. In: *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2020. DOI: 10.1109/CSCI51800.2020.00318.
- [56] Jan Steffen Becker. “Model Checking AMALTHEA with SPIN”. In: *Software Engineering (Satellite Events)*. Ed. by Sebastian Götz et al. Lecture Notes in Informatics 2814. Gesellschaft für Informatik, 2021. URL: <https://ceur-ws.org/Vol-2814/paper-A3-1.pdf>.
- [57] Jan Steffen Becker. “Partial Consistency for Requirement Engineering with Traffic Sequence Charts”. In: *Software Engineering (Workshops)*. Ed. by Regina Hebig and Robert Heinrich. 2581. CEUR Workshop Proceedings, 2020. URL: <https://ceur-ws.org/Vol-2581/ase2020paper1.pdf>.

Bibliography

- [58] Jan Steffen Becker et al. “Providing Evidence for Correct and Timely Functioning of Software Safety Mechanisms”. In: *Software Engineering 2023 Workshops*. Ed. by Iris Groher and Thomas Vogel. Gesellschaft für Informatik, 2023, pp. 66–77. DOI: 10.18420/se2023-ws-09.
- [59] Jan Becker. “A Consistency Analysis Method for Traffic Sequence Charts”. arXiv:2409.03774 [eess.SY]. Aug. 2024.
- [60] Matthias Becker and Saad Mubeen. “Timing Analysis Driven Design-Space Exploration of Cause-Effect Chains in Automotive Systems”. In: *2018 44th Annual Conference of the IEEE Industrial Electronics Society (IECON)*. IEEE, 2018, pp. 4090–4095. DOI: 10.1109/IECON.2018.8592842.
- [61] Michael von der Beeck. “A Comparison of Statecharts Variants”. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by Hans Langmaack, Willem-Paul de Roever, and Jan Vytupil. Lecture Notes in Computer Science 863. Springer, Berlin, Heidelberg, 1994, pp. 128–148. DOI: 10.1007/3-540-58468-4_163.
- [62] Sagar Behere, Martin Törngren, and De-Jiu Chen. “A Reference Architecture for Cooperative Driving”. In: *Journal of Systems Architecture* 59.10 (2013), pp. 1095–1112. DOI: 10.1016/j.sysarc.2013.05.014.
- [63] Gerd Behrmann et al. “Minimum-Cost Reachability for Priced Time Automata”. In: *Hybrid Systems: Computation and Control*. Ed. by Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli. Lecture Notes in Computer Science 2034. Springer, Berlin, Heidelberg, 2001, pp. 147–161. DOI: 10.1007/3-540-45351-2_15.
- [64] Elham Yazdani Bejarbaneh, Haiping Du, and Fazel Naghdy. “Exploring Shared Perception and Control in Cooperative Vehicle-Intersection Systems: A Review”. In: *IEEE Transactions on Intelligent Transportation Systems* 25.11 (2024), pp. 15247–15272. DOI: 10.1109/TITS.2024.3432634.
- [65] Pierfrancesco Bellini, Riccardo Mattolini, and Paolo Nesi. “Temporal Logics for Real-Time System Specification”. In: *ACM Computing Surveys* 32.1 (2000), pp. 12–42. DOI: 10.1145/349194.349197.
- [66] Hanène Ben-Abdallah and Stefan Leue. “Timing Constraints in Message Sequence Chart Specifications”. In: *Formal Description Techniques and Protocol Specification, Testing and Verification*. Ed. by Tadanori Mizuno et al. Springer, Boston, MA, 1997, pp. 91–106. DOI: 10.1007/978-0-387-35271-8_6.
- [67] Ben-Ari, Mordechai and Pnueli, Amir and Manna, Zohar. “The Temporal Logic of Branching Time”. In: *Acta Informatica* 20 (1983), pp. 207–226. DOI: 10.1007/BF01257083.
- [68] Johan Bengtsson and Wang Yi. “Timed Automata: Semantics, Algorithms and Tools”. In: *Advanced Course on Petri Nets*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Lecture Notes in Computer Science 3098. Springer, Berlin, Heidelberg, 2003, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3.

- [69] Johan Bengtsson et al. “UPPAAL – A Tool Suite for Automatic Verification of Real-Time Systems”. In: *Hybrid Systems III*. Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Lecture Notes in Computer Science 1066. Springer, Berlin, Heidelberg, 1996, pp. 232–243. DOI: 10.1007/BFb0020949.
- [70] Albert Benveniste et al. “Contracts for System Design”. In: *Foundations and Trends® in Electronic Design Automation* 12.2-3 (2018), pp. 124–400. DOI: 10.1561/10000000053.
- [71] Albert Benveniste et al. “Multiple Viewpoint Contract-Based Specification and Design”. In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Lecture Notes in Computer Science 5382. Springer, Berlin, Heidelberg, 2008, pp. 200–225. DOI: 10.1007/978-3-540-92188-2_9.
- [72] Carl Bergenhem, Mario Majdandzic, and Stig Ursing. “Concepts and Risk Analysis for a Cooperative and Automated Highway Platooning System”. In: *Dependable Computing – EDCC 2020 Workshops*. Ed. by Simona Bernardi et al. Communications in Computer and Information Science 1279. Springer, Cham, 2020, pp. 200–213. DOI: 10.1007/978-3-030-58462-7_16.
- [73] Steffen Beringer and Heike Wehrheim. “Verification of AUTOSAR Software Architectures with Timed Automata”. In: *Critical Systems: Formal Methods and Automated Verification*. Ed. by Maurice H. ter Beek, Stefania Gnesi, and Alexander Knapp. Lecture Notes in Computer Science 9933. Springer, Cham, 2016, pp. 189–204. DOI: 10.1007/978-3-319-45943-1_13.
- [74] Guillem Bernat, Alan Burns, and Albert Liamsi. “Weakly Hard Real-Time Systems”. In: *IEEE Transactions on Computers* 50.4 (2001), pp. 308–321. DOI: 10.1109/12.919277.
- [75] Guillem Bernat, Antoine Colin, and Stefan M. Petters. “WCET Analysis of Probabilistic Hard Real-Time Systems”. In: *2002 23rd IEEE International Real-Time Systems Symposium (RTSS)*. IEEE, 2002, pp. 279–288. DOI: 10.1109/REAL.2002.1181582.
- [76] Guillem Bernat, Antoine Colin, and Stefan Petters. *pWCET: A Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems*. Technical Report YCS-2003-353. University of York, 2003.
- [77] Bernard Berthomieu et al. *Real-Time Model Checking Support for AADL*. Research Report 15036. LAAS-CNRS, 2015. URL: <https://hal.science/hal-01121605/file/rtaadlmodelcheck.pdf>.
- [78] Tobias Betz et al. “How Fast Is My Software? Latency Evaluation for a ROS 2 Autonomous Driving Software”. In: *2023 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2023. DOI: 10.1109/IV55152.2023.10186585.
- [79] Sami Beydeda, Matthias Book, and Volker Gruhn, eds. *Model-Driven Software Development*. Springer, Berlin, Heidelberg, 2015. ISBN: 978-3-540-25613-7.

Bibliography

- [80] Yougang Bian et al. “Cooperation of Multiple Connected Vehicles at Unsignalized Intersections: Distributed Observation, Optimization, and Control”. In: *IEEE Transactions on Industrial Electronics* 67.12 (2020), pp. 10744–10754. DOI: 10.1109/TIE.2019.2960757.
- [81] Tom Bienmüller et al. “Modeling Requirements for Quantitative Consistency Analysis and Automatic Test Case Generation”. In: *Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems*. 2016. URL: <https://www.btc-embedded.com/downloads/whitepapers/modeling-requirements-for-quantitative-consistency-analysis-and-automatic-test-case-generation.pdf>.
- [82] Bilal Erman Bilgin and Vehbi Cagri Gungor. “Performance Comparison of IEEE 802.11p and IEEE 802.11b for Vehicle-to-Vehicle Communications in Highway, Rural, and Urban Areas”. In: *International Journal of Vehicular Technology* 2013 (2013). DOI: 10.1155/2013/971684.
- [83] Katrin Bilstrup et al. “On the Ability of the 802.11p MAC Method and STDMA to Support Real-Time Vehicle-to-Vehicle Communication”. In: *EURASIP Journal on Wireless Communications and Networking* 2009 (2009), pp. 1–13. DOI: 10.1155/2009/902414.
- [84] John Birch et al. “Safety Cases and Their Role in ISO 26262 Functional Safety Assessment”. In: *Computer Safety, Reliability, and Security*. Ed. by Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche. Lecture Notes in Computer Science 8153. Springer, Berlin, Heidelberg, 2013, pp. 154–165. DOI: 10.1007/978-3-642-40793-2_15.
- [85] Kenneth Birman, André Schiper, and Pat Stephenson. “Lightweight Causal and Atomic Group Multicast”. In: *ACM Transactions on Computer Systems* 9.3 (1991), pp. 272–314. DOI: 10.1145/128738.128742.
- [86] Daniel Bischoff et al. “Prioritizing Relevant Information: Decentralized V2X Resource Allocation for Cooperative Driving”. In: *IEEE Access* 9 (2021), pp. 135630–135656. DOI: 10.1109/ACCESS.2021.3116317.
- [87] Christopher Bischopink and Maike Schwammberger. “Verification of Fair Controllers for Urban Traffic Manoeuvres at Intersections”. In: *Formal Methods. FM 2019 International Workshops*. Ed. by Emil Sekerinski et al. Lecture Notes in Computer Science 12232. Springer, Cham, 2020, pp. 249–264. DOI: 10.1007/978-3-030-54994-7_18.
- [88] Hans Blom et al. *EAST-ADL – An Architecture Description Language for Automotive Software-Intensive Systems*. White Paper Version 2.1.12. EAST-ADL Association, 2013. URL: http://maenad.eu/public/conceptpresentations/EAST-ADL_WhitePaper_M2.1.12.pdf.

- [89] Hans Blom et al. *Language Syntax, Semantics, Metamodel V2*. Deliverable D11. TIMMO-2-USE project, 2012. URL: <https://itea4.org/project/workpackage/document/download/850/D11.TIMMO-2-USE-Languagesyntax,semantics,metamodelV2.pdf>.
- [90] John Boardman and Brian Sauser. “System of Systems – The Meaning of of”. In: *2006 IEEE/SMC International Conference on System of Systems Engineering*. IEEE, 2006, pp. 118–123. DOI: 10.1109/SYSOSE.2006.1652284.
- [91] Conrad Bock. “SysML and UML 2 Support for Activity Modeling”. In: *Systems Engineering* 9.2 (2006), pp. 160–186. DOI: 10.1002/sys.20046.
- [92] Eckard Böde et al. “Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC)”. In: FAT Series 302. Research Association for Automotive Technology, 2017. URL: <https://www.vda.de/vda/de/aktuelles/publikationen/publication/fat-schriftenreihe-302>.
- [93] Eckard Böde et al. “MULTIC-Tooling”. In: FAT Series 316. Research Association for Automotive Technology, 2019. URL: <https://www.vda.de/vda/de/aktuelles/publikationen/publication/fat-schriftenreihe-316>.
- [94] Birthe Böhm et al. “Engineering of Collaborative Embedded Systems”. In: *Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology*. Ed. by Wolfgang Böhm et al. Springer, Cham, 2021, pp. 15–48. DOI: 10.1007/978-3-030-62136-0_2.
- [95] Andrea Bombarda et al. “Extending ASMETA with Time Features”. In: *Rigorous State-Based Methods*. Ed. by Alexander Raschke and Dominique Méry. Lecture Notes in Computer Science 12709. Springer, Cham, 2021, pp. 105–111. DOI: 10.1007/978-3-030-77543-8_8.
- [96] Borzoo Bonakdarpour et al. “Approximate Distributed Monitoring under Partial Synchrony: Balancing Speed and Accuracy”. arXiv:2408.05033 [cs.LO]. Aug. 2024.
- [97] Nestor Bonjorn, Fotis Foukalas, and Paul Pop. “Enhanced 5G V2X Services Using Sidelink Device-to-Device Communications”. In: *2018 17th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*. IEEE, 2018. DOI: 10.23919/MedHocNet.2018.8407085.
- [98] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, Berlin, Heidelberg, 2003. DOI: 10.1007/978-3-642-18216-7.
- [99] Marilisa Botte et al. “An Overview of Cooperative Driving in the European Union: Policies and Practices”. In: *Electronics* 8.6 (2019), p. 616. DOI: 10.3390/electronics8060616.
- [100] Daniel Bouskela and Audrey Jardin. “ETL: A New Temporal Language for the Verification of Cyber-Physical Systems”. In: *2018 Annual IEEE International Systems Conference (SysCon)*. IEEE, 2018, pp. 1–8. DOI: 10.1109/SYSCON.2018.8369502.

Bibliography

- [101] Daniel Bouskela et al. “Formal Requirements Modeling for Cyber-Physical Systems Engineering: An Integrated Solution Based on FORM-L and Modelica”. In: *Requirements Engineering* 27.1 (2022), pp. 1–30. DOI: 10.1007/s00766-021-00359-z.
- [102] Christian Brenner et al. “ScenarioTools Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study”. In: *Electronic Communications of the EASST* 67 (2014). DOI: 10.14279/tuj.eceasst.67.948.925.
- [103] Eric Brewer. “CAP Twelve Years Later: How the ‘Rules’ Have Changed”. In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.
- [104] Matthias Brill et al. “Live Sequence Charts”. In: *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*. Ed. by Hartmut Ehrig et al. Lecture Notes in Computer Science 3147. Springer, Berlin, Heidelberg, 2004, pp. 374–399. DOI: 10.1007/978-3-540-27863-4_21.
- [105] Michael C. Brogioli. “Embedded and Multicore System Architecture – Design and Optimization”. In: *Software Engineering for Embedded Systems*. Ed. by Robert Oshana and Mark Kraeling. Newnes, 2019, pp. 89–100. DOI: 10.1016/B978-0-12-809448-8.00003-5.
- [106] David Broman et al. “Requirements for Hybrid Cosimulation Standards”. In: *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. ACM, 2015, pp. 179–188. DOI: 10.1145/2728606.2728629.
- [107] Manfred Broy. “A Model of Dynamic Systems”. In: *From Programs to Systems. The Systems Perspective in Computing*. Ed. by Saddek Bensalem, Yassine Lakhneck, and Axel Legay. Springer, Berlin, Heidelberg, 2014, pp. 39–53. DOI: 10.1007/978-3-642-54848-2_3.
- [108] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, New York, 2001. ISBN: 978-1-4612-6518-4.
- [109] Jean-Yves Brunel et al. “Softcontract: An Assertion-Based Software Development Process That Enables Design-by-Contract”. In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2004, pp. 358–363. DOI: 10.1109/DATE.2004.1268873.
- [110] Friederike Bruns, Wolfgang Nebel, and Jörg Walter. “A Detailed Analysis of Timing Effects in an IEC 61499 Ethernet/TSN Communication Scenario”. In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2022, pp. 1–8. DOI: 10.1109/ETFA52439.2022.9921436.
- [111] Friederike Bruns, Bianca Wiesmayr, and Alois Zoitl. “Supporting Model-Based Network Specification for Time-Critical Distributed Control Systems in IEC 61499”. In: *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2023, pp. 1–7. DOI: 10.1109/CASE56687.2023.10260604.

- [112] Friederike Bruns et al. “Enabling Automated Timing Verification: A Unified Approach for Industrial Distributed Control Systems”. In: *2024 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 2024, pp. 1–6. DOI: 10.1109/ICIT58233.2024.10540790.
- [113] Jeremy Bryans et al. “Maintaining Emergence in Systems of Systems Integration: A Contractual Approach Using SysML”. In: *INCOSE International Symposium* 24.1 (2014), pp. 166–181. DOI: 10.1002/j.2334-5837.2014.tb03142.x.
- [114] Jeremy Bryans et al. “SysML Contracts for Systems of Systems”. In: *2014 9th International Conference on System of Systems Engineering (SoSE)*. IEEE, 2014, pp. 73–78. DOI: 10.1109/SYSOSE.2014.6892466.
- [115] Alessio Bucaioni and Matthias Becker. “Enabling Automated Integration of Architectural Languages: An Experience Report from the Automotive Domain”. In: *Journal of Systems and Software* 184 (2022), p. 111106. DOI: 10.1016/j.jss.2021.111106.
- [116] Alessio Bucaioni et al. “From AMALTHEA to RCM and Back: A Practical Architectural Mapping Scheme”. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 537–544. DOI: 10.1109/SEAA51224.2020.00089.
- [117] Alessio Bucaioni et al. “From Low-Level Programming to Full-Fledged Industrial Model-Based Development: The Story of the Rubus Component Model”. In: *Software and Systems Modeling* 22.4 (2023), pp. 1085–1097. DOI: 10.1007/s10270-023-01107-3.
- [118] Alessio Bucaioni et al. “Reference Architectures Modelling and Compliance Checking”. In: *Software and Systems Modeling* (2022). DOI: 10.1007/s10270-022-01022-z.
- [119] Daniel-Maurice Burgstrahler. “Collaborative Sensing in Automotive Scenarios”. PhD thesis. Technical University of Darmstadt, Feb. 2017.
- [120] Alan Burns et al. “Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems”. In: *Dependable Computing for Critical Applications*. IEEE, 1999, pp. 361–378. DOI: 10.1109/DCFTS.1999.814306.
- [121] CAR 2 CAR Communication Consortium. *Connected and Cooperative Automated Driving (CCAD)*. White Paper 2300. Version 1.0. 2023. URL: https://www.car-2-car.org/fileadmin/documents/General_Documents/C2CCC_WP_2300_ConnectedAndCooperativeAutomatedDriving_V1.0.pdf.
- [122] CAR 2 CAR Communication Consortium. *Guidance for Day2 and Beyond Roadmap*. White Paper 2072. Version 1.2. 2021. URL: https://www.car-2-car.org/fileadmin/documents/General_Documents/C2CCC_WP_2072_RoadmapDay2AndBeyond_V1.2.pdf.

Bibliography

- [123] CAR 2 CAR Communication Consortium. *Use Cases*. White Paper 2097. Version 1.0. 2023. URL: https://www.car-2-car.org/fileadmin/documents/General_Documents/C2CCC_UC_2097_UseCases_V1.0.pdf.
- [124] Philippe P. F. M. Van de Calseyde, Gideon Keren, and Marcel Zeelenberg. “Decision Time as Information in Judgment and Choice”. In: *Organizational Behavior and Human Decision Processes* 125.2 (2014), pp. 113–122. DOI: 10.1016/j.obhdp.2014.07.001.
- [125] Claudia Campolo et al. “Modeling Broadcasting in IEEE 802.11p/WAVE Vehicular Networks”. In: *IEEE Communications Letters* 15.2 (2011), pp. 199–201. DOI: 10.1109/LCOMM.2011.122810.102007.
- [126] Luca Cardelli. “A Language with Distributed Scope”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1995, pp. 286–297. DOI: 10.1145/199448.199516.
- [127] Janette Cardoso et al. “Network Latency and Packet Delay Variation in Cyber-Physical Systems”. In: *Proceedings of the 1st IEEE International Network Science Workshop*. IEEE, 2011, pp. 51–58. DOI: 10.1109/NSW.2011.6004658.
- [128] Paul Caspi et al. “Semantics-Preserving Multitask Implementation of Synchronous Programs”. In: *ACM Transactions on Embedded Computing Systems* 7.2 (2008). DOI: 10.1145/1331331.1331339.
- [129] Roberto Cavada et al. “The nuXmv Symbolic Model Checker”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Lecture Notes in Computer Science 8559. Springer, Cham, 2014, pp. 334–342. DOI: 10.1007/978-3-319-08867-9_22.
- [130] Everton Cavalcante, Thais Batista, and Flavio Oquendo. “Supporting Dynamic Software Architectures: From Architectural Description to Implementation”. In: *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2015, pp. 31–40. DOI: 10.1109/WICSA.2015.21.
- [131] Roberto Cervoni, Amedeo Cesta, and Angelo Oddi. “Managing Dynamic Temporal Constraint Networks”. In: *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*. AAAI Press, 1994, pp. 13–18.
- [132] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. “A General Framework for Analysing System Properties in Platform-Based Embedded Systems Designs”. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 190–195. DOI: 10.1109/DATE.2003.1253607.
- [133] Adrien Champion et al. “CoCoSpec: A Mode-Aware Contract Language for Reactive Systems”. In: *Software Engineering and Formal Methods*. Ed. by Rocco De Nicola and Eva Kühn. Lecture Notes in Computer Science 9763. Springer, Cham, 2016, pp. 347–366. DOI: 10.1007/978-3-319-41591-8_24.

- [134] Adrien Champion et al. “The Kind 2 Model Checker”. In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Lecture Notes in Computer Science 9780. Springer, Cham, 2016, pp. 510–517. DOI: 10.1007/978-3-319-41540-6_29.
- [135] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. “A Calculus of Durations”. In: *Information Processing Letters* 40.5 (1991), pp. 269–276. DOI: 10.1016/0020-0190(91)90122-X.
- [136] Daniel Marcos Chapiro. “Globally-Asynchronous Locally-Synchronous Systems”. PhD thesis. Stanford University, Oct. 1984.
- [137] Saoussen Cheikhrouhou et al. “The Temporal Perspective in Business Process Modeling: A Survey and Research Challenges”. In: *Service Oriented Computing and Applications* 9 (2015), pp. 75–85. DOI: 10.1007/s11761-014-0170-x.
- [138] DeJiu Chen et al. “Systems Modeling with EAST-ADL for Fault Tree Analysis Through HiP-HOPS”. In: *IFAC Proceedings Volumes* 46.22 (2013), pp. 91–96. DOI: 10.3182/20130904-3-UK-4041.00043.
- [139] Mingshuai Chen et al. “Indecision and Delays Are the Parents of Failure – Taming Them Algorithmically by Synthesizing Delay-Resilient Control”. In: *Acta Informatica* 58.5 (2021), pp. 497–528. DOI: 10.1007/s00236-020-00374-7.
- [140] Pin Chen and Jennie Clothier. “Advancing Systems Engineering for Systems-of-Systems Challenges”. In: *Systems Engineering* 6.3 (2003), pp. 170–183. DOI: 10.1002/sys.10042.
- [141] Xiaohong Chen et al. “Empowering Domain Experts with Formal Methods for Consistency Verification of Safety Requirements”. In: *IEEE Transactions on Intelligent Transportation Systems* 24.12 (2023), pp. 15146–15157. DOI: 10.1109/TITS.2023.3324022.
- [142] Imane Cherfa et al. “Systems of Systems: From Mission Definition to Architecture Description”. In: *Systems Engineering* 22.6 (2019), pp. 437–454. DOI: 10.1002/sys.21523.
- [143] Jui-Hao Chiang and Tzi-cker Chiueh. “Accurate Clock Synchronization for IEEE 802.11-Based Multi-Hop Wireless Networks”. In: *2009 17th IEEE International Conference on Network Protocols*. IEEE, 2009, pp. 11–20. DOI: 10.1109/ICNP.2009.5339703.
- [144] Jinho Choi, Eunkyong Jee, and Doo-Hwan Bae. “Timing Consistency Checking for UML/MARTE Behavioral Models”. In: *Software Quality Journal* 24.3 (2016), pp. 835–876. DOI: 10.1007/s11219-015-9290-6.
- [145] Antonio Cicchetti et al. “CHESS: A Model-Driven Engineering Tool Environment for Aiding the Development of Complex Industrial Systems”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 362–365. DOI: 10.1145/2351676.2351748.

Bibliography

- [146] Cimatti, Alessandro and Dorigatti, Michele and Tonetta, Stefano. *OCRA: Othello Contracts Refinement Analysis*. Version 2.1.0. 2021. URL: https://ocra.fbk.eu/download/OCRA_Language_User_Guide.pdf.
- [147] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. “OCRA: A Tool for Checking the Refinement of Temporal Contracts”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 702–705. DOI: 10.1109/ASE.2013.6693137.
- [148] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. “Requirements Validation for Hybrid Systems”. In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Lecture Notes in Computer Science 5643. Springer, Berlin, Heidelberg, 2009, pp. 188–203. DOI: 10.1007/978-3-642-02658-4_17.
- [149] Alessandro Cimatti and Stefano Tonetta. “Contracts-Refinement Proof System for Component-Based Embedded Systems”. In: *Science of Computer Programming* 97.3 (2015), pp. 333–348. DOI: 10.1016/j.scico.2014.06.011.
- [150] Alessandro Cimatti et al. “A Comprehensive Framework for the Analysis of Automotive Systems”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2022, pp. 379–389. DOI: 10.1145/3550355.3552408.
- [151] Alessandro Cimatti et al. “Kratos – A Software Model Checker for SystemC”. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science 6806. Springer, Berlin, Heidelberg, 2011, pp. 310–316. DOI: 10.1007/978-3-642-22110-1_24.
- [152] Alessandro Cimatti et al. “Validation of Requirements for Hybrid Systems: A Formal Approach”. In: *ACM Transactions on Software Engineering and Methodology* 21.4 (2013). DOI: 10.1145/2377656.2377659.
- [153] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Transactions on Programming Languages and Systems* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399.
- [154] Edmund M. Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Lecture Notes in Computer Science 7682. Springer, Berlin, Heidelberg, 2012, pp. 1–30. DOI: 10.1007/978-3-642-35746-6_1.
- [155] Paul Cobby and Peter J. Schulz. *Theories and Models of Communication*. Handbooks of Communication Science. De Gruyter Mouton, 2013. ISBN: 978-3-11-024044-3.
- [156] Robert Cooper and Keith Marzullo. “Consistent Detection of Global Predicates”. In: *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM, 1991, pp. 167–174. DOI: 10.1145/122759.122774.

- [157] Jorge Cortes et al. “Coverage Control for Mobile Sensing Networks”. In: *IEEE Transactions on Robotics and Automation* 20.2 (2004), pp. 243–255. DOI: 10.1109/TRA.2004.824698.
- [158] Flaviu Cristian. “Probabilistic Clock Synchronization”. In: *Distributed Computing* 3.3 (1989), pp. 146–158. DOI: 10.1007/BF01784024.
- [159] Flaviu Cristian, Houtan Aghili, and Ray Strong. “Clock Synchronization in the Presence of Omission and Performance Faults and Processor Joins”. In: *Proceedings of the 16th IEEE International Symposium on Fault Tolerant Computing Systems*. IEEE, 1986, pp. 218–223.
- [160] Flaviu Cristian et al. “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement”. In: *Information and Computation* 118.1 (1995), pp. 158–179. DOI: 10.1006/inco.1995.1060.
- [161] Philippe Cuenot et al. “The EAST-ADL Architecture Description Language for Automotive Embedded Software”. In: *Model-Based Engineering of Embedded Real-Time Systems*. Ed. by Holger Giese et al. Lecture Notes in Computer Science 6100. Springer, Berlin, Heidelberg, 2010, pp. 297–307. DOI: 10.1007/978-3-642-16277-0_11.
- [162] Werner Damm and David Harel. “LSCs: Breathing Life into Message Sequence Charts”. In: *Formal Methods in System Design* 19.1 (2001), pp. 45–80. DOI: 10.1023/A:1011227529550.
- [163] Werner Damm, Eike Möhlmann, and Astrid Rakow. “Component Based Design of Hybrid Systems: A Case Study on Concurrency and Coupling”. In: *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*. ACM, 2014, pp. 145–150. DOI: 10.1145/2562059.2562120.
- [164] Werner Damm et al. “Multi-Layer Time Coherency in the Development of ADAS/AD Systems: Design Approach and Tooling”. In: *Proceedings of the Workshop on Design Automation for CPS and IoT*. ACM, 2019, pp. 20–30. DOI: 10.1145/3313151.3313167.
- [165] Werner Damm et al. “Towards Component Based Design of Hybrid Systems: Safety and Stability”. In: *Time for Verification: Essays in Memory of Amir Pnueli*. Ed. by Zohar Manna and Doron A. Peled. Lecture Notes in Computer Science 6200. Springer, Berlin, Heidelberg, 2010, pp. 96–143. DOI: 10.1007/978-3-642-13754-9_6.
- [166] Werner Damm et al. “Using Contract-Based Component Specifications for Virtual Integration Testing and Architecture Design”. In: *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1023–1028. DOI: 10.1109/DATE.2011.5763167.
- [167] Dassault Systèmes SE. *Cameo Systems Modeler*. <https://www.3ds.com/products/catia/no-magic/cameo-systems-modeler>. Last accessed on June 19, 2026.
- [168] Dassault Systèmes SE. *MagicDraw*. <https://www.3ds.com/products/catia/no-magic/magicdraw>. Last accessed on June 19, 2026.

Bibliography

- [169] Abhijit Davare et al. “A Next-Generation Design Framework for Platform-Based Design”. In: *Conference on Using Hardware Design and Verification Languages*. Vol. 152. 2007.
- [170] Alexandre David et al. “UPPAAL SMC Tutorial”. In: *International Journal on Software Tools for Technology Transfer* 17 (2015), pp. 397–415. DOI: 10.1007/s10009-014-0361-y.
- [171] Alexandre David et al. “Uppaal Stratego”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Lecture Notes in Computer Science 9035. Springer, Berlin, Heidelberg, 2015, pp. 206–211. DOI: 10.1007/978-3-662-46681-0_16.
- [172] Robert I. Davis and Liliana Cucu-Grosjean. “A Survey of Probabilistic Schedulability Analysis Techniques for Real-Time Systems”. In: *Leibniz Transactions on Embedded Systems* 6.1 (2019), 04:1–04:53. DOI: 10.4230/LITES-v006-i001-a004.
- [173] Robert I. Davis and Liliana Cucu-Grosjean. “A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems”. In: *Leibniz Transactions on Embedded Systems* 6.1 (2019), 03:1–03:60. DOI: 10.4230/LITES-v006-i001-a003.
- [174] Robert I. Davis et al. “Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised”. In: *Real-Time Systems* 35.3 (2007), pp. 239–272. DOI: 10.1007/s11241-007-9012-7.
- [175] Thomas L. Dean and Drew V. McDermott. “Temporal Data Base Management”. In: *Artificial Intelligence* 32.1 (1987), pp. 1–55. DOI: 10.1016/0004-3702(87)90061-0.
- [176] Alberto Debiasi et al. “Model-Based Analysis Support for Dependable Complex Systems in CHeSS”. In: *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. Ed. by Slimane Hammoudi et al. SciTePress, 2021, pp. 262–269. DOI: 10.5220/0010269702620269.
- [177] Rina Dechter, Itay Meiri, and Judea Pearl. “Temporal Constraint Networks”. In: *Artificial Intelligence* 49.1-3 (1991), pp. 61–95. DOI: 10.1016/0004-3702(91)90006-6.
- [178] Jerker Delsing, Géza Kulcsár, and Øystein Haugen. “SysML Modeling of Service-Oriented System-of-Systems”. In: *Innovations in Systems and Software Engineering* (2022). DOI: 10.1007/s11334-022-00455-5.
- [179] Zhiyun Deng et al. “Cooperative Platoon Formation of Connected and Autonomous Vehicles: Toward Efficient Merging Coordination at Unsignalized Intersections”. In: *IEEE Transactions on Intelligent Transportation Systems* (2023), pp. 1–15. DOI: 10.1109/TITS.2023.3235774.
- [180] Jyotirmoy V Deshmukh et al. “Robust Online Monitoring of Signal Temporal Logic”. In: *Formal Methods in System Design* 51 (2017), pp. 5–30. DOI: 10.1007/s10703-017-0286-7.

- [181] Boya Di et al. “V2X Meets NOMA: Non-Orthogonal Multiple Access for 5G-Enabled Vehicular Networks”. In: *IEEE Wireless Communications* 24.6 (2017), pp. 14–21. DOI: 10.1109/MWC.2017.1600414.
- [182] Charles E. Dickerson, Siyuan Ji, and Rosmira Roslan. “Formal Methods for a System of Systems Analysis Framework Applied to Traffic Management”. In: *2016 11th System of Systems Engineering Conference (SoSE)*. IEEE, 2016, pp. 1–6. DOI: 10.1109/SYSOSE.2016.7542918.
- [183] Henning Dierks. “PLC-Automata: A New Class of Implementable Real-Time Automata”. In: *Theoretical Computer Science* 253.1 (2001), pp. 61–93. DOI: 10.1016/S0304-3975(00)00089-X.
- [184] Henning Dierks and Marc Lettrari. “Constructing Test Automata from Graphical Real-Time Requirements”. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by Werner Damm and Ernst-Rüdiger Olderog. Lecture Notes in Computer Science 2469. Springer, Berlin, Heidelberg, 2002, pp. 433–454. DOI: 10.1007/3-540-45739-9_25.
- [185] Digitaler Knoten 4.0 partners. *Digitaler Knoten 4.0 – Verkehr intelligent vernetzen*. <https://www.dlr.de/de/ts/forschung-und-transfer/projekte/digitaler-knoten-4.0>. Last accessed on June 19, 2026.
- [186] Nicholas J. Dingle, William J. Knottenbelt, and Tamas Suto. “PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets”. In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (2009), pp. 34–39. DOI: 10.1145/1530873.1530881.
- [187] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. “On the Minimal Synchronism Needed for Distributed Consensus”. In: *Journal of the ACM* 34.1 (1987), pp. 77–97. DOI: 10.1145/7531.7533.
- [188] Wei Dou, Domenico Bianculli, and Lionel Briand. “A Model-Driven Approach to Trace Checking of Pattern-Based Temporal Properties”. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2017, pp. 323–333. DOI: 10.1109/MODELS.2017.9.
- [189] Iulia Dragomir, Iulian Ober, and Christian Percebois. “Contract-Based Modeling and Verification of Timed Safety Requirements Within SysML”. In: *Software & Systems Modeling* 16.2 (2017), pp. 587–624. DOI: 10.1007/s10270-015-0481-1.
- [190] Charaf Eddine Dridi, Zakaria Benzadri, and Faïza Belala. “System of Systems Modelling: Recent Work Review and a Path Forward”. In: *2020 International Conference on Advanced Aspects of Software Engineering (ICAASE)*. IEEE, 2020, pp. 1–8. DOI: 10.1109/ICAASE51408.2020.9380125.
- [191] Haibo Du et al. “Distributed Finite-Time Cooperative Control of Multiple High-Order Nonholonomic Mobile Robots”. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.12 (2017), pp. 2998–3006. DOI: 10.1109/TNNLS.2016.2610140.

Bibliography

- [192] YanHua Du et al. “Dynamic Checking and Solution to Temporal Violations in Concurrent Workflow Processes”. In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 41.6 (2011), pp. 1166–1181. DOI: 10.1109/TSMCA.2011.2116003.
- [193] Yu Yue Du, Chang Jun Jiang, and Meng Chu Zhou. “Modeling and Analysis of Real-Time Cooperative Systems Using Petri Nets”. In: *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans* 37.5 (2007), pp. 643–654. DOI: 10.1109/TSMCA.2007.902622.
- [194] Felipe Lima Duarte, Angélica Félix de Castro, and Paulo Gabriel Gadelha Queiroz. “REAP-SoS: A Requirement Engineering Approach for System of Systems”. In: *Computer Science & Information Technology*. Vol. 8. CS & IT Conference Proceedings, 2018, pp. 131–148. DOI: 10.5121/csit.2018.80510.
- [195] Hugh Durrant-Whyte and Tim Bailey. “Simultaneous Localization and Mapping: Part I”. In: *IEEE Robotics & Automation Magazine* 13.2 (2006), pp. 99–110. DOI: 10.1109/MRA.2006.1638022.
- [196] Stefan Dziwok et al. *The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling*. Technical Report tr-ri-16-352. Fraunhofer Institute for Mechatronic Systems Design and Heinz Nixdorf Institute, 2016.
- [197] EAST-ADL Association. *EAST-ADL – Domain Model Specification*. Version 2.1.12. 2013. URL: https://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf.
- [198] David Eckhoff, Christoph Sommer, and Falko Dressler. “On the Necessity of Accurate IEEE 802.11p Models for IVC Protocol Simulation”. In: *2012 IEEE 75th Vehicular Technology Conference (VTC Spring)*. IEEE, 2012, pp. 1–5. DOI: 10.1109/VETECS.2012.6240064.
- [199] Eclipse Foundation, Inc. *Eclipse APP4MC Documentation*. <https://eclipse.dev/app4mc/help/latest/index.html>. Last accessed on June 19, 2026.
- [200] Eclipse Foundation, Inc. *Eclipse APP4MC*. <https://eclipse.dev/app4mc/>. Last accessed on June 19, 2026.
- [201] Eclipse Foundation, Inc. *Eclipse CHESS*. <https://www.eclipse.dev/chess/>. Last accessed on June 19, 2026.
- [202] Eclipse Foundation, Inc. *Eclipse Papyrus. Modeling Environment*. <https://eclipse.dev/papyrus/>. Last accessed on June 19, 2026.
- [203] Johann Eder, Euthimios Panagos, and Michael Rabinovich. “Time Constraints in Workflow Systems”. In: *Seminal Contributions to Information Systems Engineering: 25 Years of CAiSE*. Ed. by Janis Bubenko et al. Springer, Berlin, Heidelberg, 2013, pp. 191–205. DOI: 10.1007/978-3-642-36926-1_15.

- [204] Johann Eder et al. “Time Management in Workflow Systems”. In: *3rd International Conference on Business Information Systems*. Ed. by Witold Abramowicz and Maria E. Orlowska. Springer, London, 1999, pp. 265–280. DOI: 10.1007/978-1-4471-0875-7_22.
- [205] Günter Ehmen et al. “Coherent Treatment of Time in the Development of ADAS/AD Systems: Design Approach and Demonstration”. In: *SAE World Congress Experience 2018*. SAE International, 2018. DOI: 10.4271/2018-01-0592.
- [206] Günter Ehmen et al. “Measurement-Based Online Verification of Timing Properties in Distributed Systems”. In: *2020 International Conference on Omni-Layer Intelligent Systems (COINS)*. IEEE, 2020, pp. 93–98. DOI: 10.1109/COINS49042.2020.9191647.
- [207] Stephan Eichler. “Performance Evaluation of the IEEE 802.11p WAVE Communication Standard”. In: *2007 66th IEEE Vehicular Technology Conference*. IEEE, 2007, pp. 2199–2203. DOI: 10.1109/VETECF.2007.461.
- [208] Johan Eker et al. “Taming Heterogeneity – the Ptolemy Approach”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 127–144. DOI: 10.1109/JPROC.2002.805829.
- [209] E. Allen Emerson and Joseph Y. Halpern. “‘Sometimes’ and ‘Not Never’ Revisited: On Branching Versus Linear Time Temporal Logic”. In: *Journal of the ACM* 33.1 (1986), pp. 151–178. DOI: 10.1145/4904.4999.
- [210] E. Allen Emerson et al. “Quantitative Temporal Reasoning”. In: *Real-Time Systems* 4.4 (1992), pp. 331–352. DOI: 10.1007/BF00355298.
- [211] Jakob Engblom et al. “Worst-Case Execution-Time Analysis for Embedded Real-Time Systems”. In: *International Journal on Software Tools for Technology Transfer* 4 (2003), pp. 437–455. DOI: 10.1007/s100090100054.
- [212] Gregor Engels et al. “Process Modeling Using UML”. In: *Process-Aware Information Systems: Bridging People and Software through Process Technology*. John Wiley & Sons, 2005, pp. 83–117. DOI: 10.1002/0471741442.ch5.
- [213] Patrick Th. Eugster et al. “The Many Faces of Publish/Subscribe”. In: *ACM Computing Surveys* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078.
- [214] European Telecommunications Standards Institute. *Intelligent Transport Systems (ITS); Facilities Layer; Cooperative Awareness Service*. Technical Specification ETSI TS 103 900. Version 2.2.1. 2025. URL: https://www.etsi.org/deliver/etsi_ts/103900_103999/103900/02.02.01_60/ts_103900v020201p.pdf.
- [215] European Telecommunications Standards Institute. *Intelligent Transport Systems (ITS); ITS-G5 Access Layer Specification for Intelligent Transport Systems Operating in the 5 GHz Frequency Band*. European Standard ETSI EN 302 663. Version 1.3.1. 2020. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/302663/01.03.01_60/en_302663v010301p.pdf.

Bibliography

- [216] European Telecommunications Standards Institute. *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications*. Technical Report ETSI TR 102 638. Version 2.1.1. 2024. URL: https://www.etsi.org/deliver/etsi_tr/102600_102699/102638/02.01.01_60/tr_102638v020101p.pdf.
- [217] Anthony M. Evans and Philippe P. F. M. van de Calseyde. “The Effects of Observed Decision Time on Expectations of Extremity and Cooperation”. In: *Journal of Experimental Social Psychology* 68 (2017), pp. 50–59. DOI: 10.1016/j.jesp.2016.05.009.
- [218] Anthony M. Evans and David G. Rand. “Cooperation and Decision Time”. In: *Current Opinion in Psychology* 26 (2019), pp. 67–71. DOI: 10.1016/j.copsyc.2018.05.007.
- [219] Haneen Farah et al. “Infrastructure for Automated and Connected Driving: State of the Art and Future Research Directions”. In: *Road Vehicle Automation 4*. Ed. by Gereon Meyer and Sven Beiker. Lecture Notes in Mobility. Springer, Cham, 2018, pp. 187–197.
- [220] Federal Motor Transport Agency. *KBA erteilt erste Genehmigung zum automatisierten Fahren*. https://www.kba.de/DE/Presse/Pressemitteilungen/Allgemein/2021/pm49_2021_erste_Genehmigung_automatisiertes_Fahren.html. Last accessed on June 19, 2026.
- [221] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Technical Note CMU/SEI-2006-TN-011. Carnegie Mellon University, 2006. URL: https://insights.sei.cmu.edu/documents/2103/2006_004_001_14678.pdf.
- [222] Peter Feiler. *Open Source AADL Tool Environment (OSATE)*. Presentation. Carnegie Mellon University, 2005.
- [223] Peter Feiler and Ed Colbert. *The SAE AADL Standard – An Architecture Analysis & Design Language for Embedded Real-Time Systems*. Presentation. Carnegie Mellon University, 2004.
- [224] Martin Feilkas et al. *A Refined Top-Down Methodology for the Development of Automotive Software Systems: The KeylessEntry-System Case Study*. Technical Report TUM-I1103. Technical University of Munich, 2011. URL: <https://mediatum.ub.tum.de/doc/1094230/1094230.pdf>.
- [225] Joseph A. Fernandez et al. “Performance of the 802.11p Physical Layer in Vehicle-to-Vehicle Environments”. In: *IEEE Transactions on Vehicular Technology* 61.1 (2012), pp. 3–14. DOI: 10.1109/TVT.2011.2164428.
- [226] 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*. IEEE, 2014, pp. 1–4. DOI: 10.1109/SIES.2014.7087332.

- [227] Norman Finn. “Introduction to Time-Sensitive Networking”. In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 22–28. DOI: 10.1109/MCOMSTD.2018.1700076.
- [228] Viktoria Firus, Steffen Becker, and Jens Happe. “Parametric Performance Contracts for QML-Specified Software Components”. In: *Electronic Notes in Theoretical Computer Science* 141.3 (2005), pp. 73–90. DOI: 10.1016/j.entcs.2005.04.036.
- [229] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *Journal of the ACM* 32.2 (1985), pp. 374–382. DOI: 10.1145/3149.214121.
- [230] Marc Förster. “Dependable Reuse & Guarded Integration of Automotive Software Components”. In: *FAT Series*. 247. Research Association for Automotive Technology, 2013. URL: <https://www.vda.de/vda/de/aktuelles/publikationen/publication/fat-schriftenreihe-247>.
- [231] Fraunhofer Institute for Experimental Software Engineering. *safeTbox*. <https://safetbox.de/>. Last accessed on June 19, 2026.
- [232] Fraunhofer Institute for Mechatronic Systems Design. *MechatronicUML*. <http://www.mechatronicuml.org/>. Last accessed on June 19, 2026.
- [233] Thomas Frese et al. “Fault Tolerance Time Interval”. In: *Neue Dimensionen der Mobilität: Technische und betriebswirtschaftliche Aspekte*. Ed. by Heike Proff. Springer Gabler, Wiesbaden, 2020, pp. 559–567. DOI: 10.1007/978-3-658-29746-6_45.
- [234] Denis Gagné and André Trudel. “Time-BPMN”. In: *2009 IEEE Conference on Commerce and Enterprise Computing*. IEEE, 2009, pp. 361–367. DOI: 10.1109/CEC.2009.71.
- [235] M. T. Gamble and R. F. Gamble. “Reasoning About Hybrid System of Systems Designs”. In: *International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*. IEEE, 2008, pp. 154–163. DOI: 10.1109/ICCBSS.2008.39.
- [236] Malin Gandor et al. “Architectures for Dynamically Coupled Systems”. In: *Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology*. Ed. by Wolfgang Böhm et al. Springer, Cham, 2021, pp. 95–124. DOI: 10.1007/978-3-030-62136-0_5.
- [237] Malin Gandor et al. *Architekturkonzepte für (teil-)virtualisierte Verkehrsinfrastruktur*. Deliverable D4.2. ViVre project, 2022.
- [238] Kai Gao et al. “Connected Vehicle as a Mobile Sensor for Real Time Queue Length at Signalized Intersections”. In: *Sensors* 19.9 (2019), p. 2059. DOI: 10.3390/s19092059.
- [239] Juan A. Garay and Kenneth J. Perry. “A Continuum of Failure Models for Distributed Computing”. In: *Distributed Algorithms*. Ed. by Adrian Segall and Shmuel Zaks. Lecture Notes in Computer Science 647. Springer, Berlin, Heidelberg, 1992, pp. 153–165. DOI: 10.1007/3-540-56188-9_11.

Bibliography

- [240] German Association of the Automotive Industry. *The Research Association of Automotive Technology (FAT)*. <https://www.vda.de/en/association/organization/departments/research-and-innovation>. Last accessed on June 19, 2026.
- [241] 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*. 2014.
- [242] Tarek Ghouil and Tarek Sayed. “Real-Time Signal-Vehicle Coupled Control: An Application of Connected Vehicle Data to Improve Intersection Safety”. In: *Accident Analysis & Prevention* 162 (2021). DOI: 10.1016/j.aap.2021.106389.
- [243] Holger Giese and Sven Burmester. *Real-Time Statechart Semantics*. Technical Report tr-ri-03-239. University of Paderborn, 2003.
- [244] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *ACM SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601.
- [245] Timo Gilgen. “‘Der Drive Pilot war unsere Mondlandung’”. In: *automotiveIT. Autonomes Fahren* (2024). URL: <https://www.automotiveit.eu/technology/autonomes-fahren/der-drive-pilot-war-unsere-mondlandung-51-955.html>.
- [246] Pourya Gohari, Mitra Nasri, and Jeroen Voeten. “Data-Age Analysis for Multi-Rate Task Chains Under Timing Uncertainty”. In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. ACM, 2022, pp. 24–35. DOI: 10.1145/3534879.3534893.
- [247] Arda Goknil et al. “Analysis Support for TADL2 Timing Constraints on EAST-ADL Models”. In: *Software Architecture*. Ed. by Khalil Drira. Lecture Notes in Computer Science 7957. Springer, Berlin, Heidelberg, 2013, pp. 89–105. DOI: 10.1007/978-3-642-39031-9_8.
- [248] Manuel Gonzalez-Martín et al. “Analytical Models of the Performance of C-V2X Mode 4 Vehicular Communications”. In: *IEEE Transactions on Vehicular Technology* 68.2 (2018), pp. 1155–1166. DOI: 10.1109/TVT.2018.2888704.
- [249] Edgar E. González, David Garcia-Roger, and Jose F. Monserrat. “LTE/NR V2X Communication Modes and Future Requirements of Intelligent Transportation Systems Based on MR-DC Architectures”. In: *Sustainability* 14.7 (2022). DOI: 10.3390/su14073879.
- [250] Christian Granrath et al. “Feature-Driven Systems Engineering Procedure for Standardized Product-Line Development”. In: *Systems Engineering* 24.6 (2021), pp. 456–479. DOI: 10.1002/sys.21596.
- [251] Joel Greenyer. “Scenario-Based Design of Mechatronic Systems”. PhD thesis. Paderborn University, Oct. 2011. URL: <https://digital.ub.uni-paderborn.de/hxsx/content/titleinfo/171048>.

- [252] Radu Grosu and Ketil Stølen. “A Model for Mobile Point-to-Point Data-Flow Networks Without Channel Sharing”. In: *Algebraic Methodology and Software Technology*. Ed. by Martin Wirsing and Maurice Nivat. Springer, Berlin, Heidelberg, 1996, pp. 504–519. DOI: 10.1007/BFb0014336.
- [253] Xin Gu et al. “Performance Analysis and Optimization for Semi-Persistent Scheduling in C-V2X”. In: *IEEE Transactions on Vehicular Technology* 72.4 (2022), pp. 4628–4642. DOI: 10.1109/TVT.2022.3223414.
- [254] Nicolas Guelfi and Amel Mammar. “A Formal Semantics of Timed Activity Diagrams and Its PROMELA Translation”. In: *12th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2005. DOI: 10.1109/APSEC.2005.7.
- [255] Nawal Guermouche. “Etude Des Interactions Temporisées Dans La Composition De Services Web”. PhD thesis. Henri Poincaré University, June 2010.
- [256] Milena Guessi, Everton Cavalcante, and Lucas B. R. Oliveira. “Characterizing Architecture Description Languages for Software-Intensive Systems-of-Systems”. In: *2015 IEEE/ACM 3rd International Workshop on Software Engineering for Systems-of-Systems*. IEEE, 2015, pp. 12–18. DOI: 10.1109/SESoS.2015.10.
- [257] Housseem Guissouma et al. “Lifecycle Management of Automotive Safety-Critical Over the Air Updates: A Systems Approach”. In: *IEEE Access* 10 (2022), pp. 57696–57717. DOI: 10.1109/ACCESS.2022.3176879.
- [258] J. Octavio Gutiérrez-García, Félix F. Ramos-Corchado, and Jean-Luc Koning. “Obligations As Constrainers, Descriptors, and Linkers of Open System of Systems”. In: *2009 IEEE International Conference on System of Systems Engineering (SoSE)*. IEEE, 2009, pp. 1–6.
- [259] Sohan Gyawali et al. “Challenges and Solutions for Cellular Based V2X Communications”. In: *IEEE Communications Surveys & Tutorials* 23.1 (2021), pp. 222–255. DOI: 10.1109/COMST.2020.3029723.
- [260] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. *MontiArc – Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Technical Report AIB-2012-03. RWTH Aachen, 2012. URL: <https://www.se-rwth.de/publications/MontiArc-Architectural-Modeling-of-Interactive-Distributed-and-Cyber-Physical-Systems.pdf>.
- [261] Vassos Hadzilacos. “Connectivity Requirements for Byzantine Agreement Under Restricted Types of Failures”. In: *Distributed Computing* 2.2 (1987), pp. 95–103. DOI: 10.1007/BF01667081.
- [262] Vassos Hadzilacos and Joseph Y. Halpern. “Message-Optimal Protocols for Byzantine Agreement”. In: *Mathematical Systems Theory* 26.1 (1993), pp. 41–102. DOI: 10.1007/BF01187074.
- [263] Georg Hake et al. “Integrating Scenario- and Contract-Based Verification for Automated Vessels”. In: *Journal of Marine Science and Technology* (2024), pp. 1–21. DOI: 10.1007/s00773-024-01008-0.

Bibliography

- [264] Arne Hamann et al. “Communication Centric Design in Complex Automotive Embedded Systems”. In: ed. by Marko Bertogna. *Leibniz International Proceedings in Informatics 76*. Dagstuhl Publishing, 2017, 10:1–10:20. DOI: 10.4230/LIPIcs.ECRTS.2017.10.
- [265] Zeeshan Hameed Mir and Fethi Filali. “LTE and IEEE 802.11p for Vehicular Networking: A Performance Evaluation”. In: *EURASIP Journal on Wireless Communications and Networking* 2014 (2014), pp. 1–15. DOI: 10.1186/1687-1499-2014-89.
- [266] Seungjae Han, Kang G. Shin, and Harold A. Rosenberg. “DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems”. In: *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. IEEE, 1995, pp. 204–213. DOI: 10.1109/IPDS.1995.395831.
- [267] Kaj Hänninen et al. “The Rubus Component Model for Resource Constrained Real-Time Systems”. In: *2008 International Symposium on Industrial Embedded Systems*. IEEE, 2008, pp. 177–183. DOI: 10.1109/SIES.2008.4577697.
- [268] Hans Hansson and Bengt Jonsson. “A Logic for Reasoning About Time and Reliability”. In: *Formal Aspects of Computing* 6 (1994), pp. 512–535. DOI: 10.1007/BF01211866.
- [269] M. González Harbour et al. “MAST: Modeling and Analysis Suite for Real-Time Applications”. In: *Proceedings of the 13th Euromicro Conference on Real Time Systems*. IEEE, 2001, pp. 125–134. DOI: 10.1109/EMRTS.2001.934015.
- [270] David Harel. “Statecharts: A Visual Formalism for Complex Systems”. In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9.
- [271] David Harel and Shahar Maoz. “Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams”. In: *Software and Systems Modeling* 7.2 (2008), pp. 237–252. DOI: 10.1007/s10270-007-0054-z.
- [272] David Harel and Rami Marelly. “Playing with Time: On the Specification and Execution of Time-Enriched LSCs”. In: *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*. IEEE, 2002, pp. 193–202. DOI: 10.1109/MASCOT.2002.1167077.
- [273] David Harel and Assaf Marron. “Introducing Dynamical Systems and Chaos Early in Computer Science and Software Engineering Education Can Help Advance Theory and Practice of Software Development and Computing”. In: *Leveraging Applications of Formal Methods, Verification and Validation: 10th International Symposium on Leveraging Applications of Formal Methods*. Lecture Notes in Computer Science 13036. Springer, Cham, 2021, pp. 322–334. DOI: 10.1007/978-3-030-89159-6_20.

- [274] David Harel and P. S. Thiagarajan. “Message Sequence Charts”. In: *UML for Real: Design of Embedded Real-Time Systems*. Ed. by Luciano Lavagno, Grant Martin, and Bran Selic. Springer, Boston, MA, 2003, pp. 77–105. DOI: 10.1007/0-306-48738-1_4.
- [275] Eyal Harel, Orna Lichtenstein, and Amir Pnueli. “Explicit Clock Temporal Logic”. In: *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1990, pp. 402–413. DOI: 10.1109/LICS.1990.113765.
- [276] Seif Haridi, Peter Van Roy, and Gert Smolka. “An Overview of the Design of Distributed Oz”. In: *Proceedings of the Second International Symposium on Parallel Symbolic Computation*. ACM, 1997, pp. 176–187. DOI: 10.1145/266670.266726.
- [277] Dennis Harmann, Sefa Yilmaz-Niewerth, and Christina Jacob. “Methodological Distribution of Virtual Stops for Ridepooling”. In: *Transportation Research Procedia* 62 (2022), pp. 442–449. DOI: 10.1016/j.trpro.2022.02.055.
- [278] Anne-Wil Harzing. *Publish or Perish*. <https://harzing.com/resources/publish-or-perish>. Last accessed on June 19, 2026.
- [279] Khondokar Fida Hasan et al. “Time Synchronization in Vehicular Ad-Hoc Networks: A Survey on Theory and Practice”. In: *Vehicular Communications* 14 (2018), pp. 39–51. DOI: 10.1016/j.vehcom.2018.09.001.
- [280] Shahriar Hasan, Svetlana Girs, and Elisabeth Uhlemann. “Characterization of Transient Communication Outages into States to Enable Autonomous Fault Tolerance in Vehicle Platooning”. In: *IEEE Open Journal of Intelligent Transportation Systems* 4 (2023), pp. 101–129. DOI: 10.1109/OJITS.2023.3237958.
- [281] Matthew C. Hause. “SOS for SoS: A New Paradigm for System of Systems Modeling”. In: *2014 IEEE Aerospace Conference*. IEEE, 2014, pp. 1–12. DOI: 10.1109/AERO.2014.6836335.
- [282] Matthew Heath and Ian Harris. “A Deterministic Globally Asynchronous Locally Synchronous Microprocessor Architecture”. In: *Proceedings of the 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*. IEEE, 2003, pp. 119–124. DOI: 10.1109/MTV.2003.1250272.
- [283] Peter Henderson. “Modelling Architectures for Dynamic Systems”. In: *Programming Methodology*. Ed. by Annabelle McIver and Carroll Morgan. Monographs in Computer Science. Springer, New York, 2003, pp. 161–174. DOI: 10.1007/978-0-387-21798-7_8.
- [284] Martijn Hendriks and Marcel Verhoef. “Timed Automata Based Analysis of Embedded System Architectures”. In: *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 179–186. DOI: 10.1109/IPDPS.2006.1639422.
- [285] 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. DOI: 10.1049/ip-cdt:20045088.

- [286] Thomas A. Henzinger, Ben Horowitz, and Christoph M. Kirsch. “Giotto: A Time-Triggered Language for Embedded Programming”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 84–99. DOI: 10.1109/JPROC.2002.805825.
- [287] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. “Giotto: A Time-Triggered Language for Embedded Programming”. In: *Embedded Software*. Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Lecture Notes in Computer Science 2211. Springer, Berlin, Heidelberg, 2001, pp. 166–184. DOI: 10.1007/3-540-45449-7_12.
- [288] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. “Temporal Proof Methodologies for Timed Transition-Systems”. In: *Information and Computation* 112.2 (1994), pp. 273–337. DOI: 10.1006/inco.1994.1060.
- [289] Anders Hessel et al. “Testing Real-Time Systems Using UPPAAL”. In: *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Springer, Berlin, Heidelberg, 2008, pp. 77–117. DOI: 10.1007/978-3-540-78917-8_3.
- [290] Jon Holt et al. “A Model-Based Approach for Requirements Engineering for Systems of Systems”. In: *IEEE Systems Journal* 9.1 (2015), pp. 252–262. DOI: 10.1109/JSYST.2014.2312051.
- [291] Jörg Holtmann. “Improvement of Software Requirements Quality Based on Systems Engineering”. PhD thesis. Paderborn University, June 2019.
- [292] Jörg Holtmann, Julien Deantoni, and Markus Fockel. “Early Timing Analysis Based on Scenario Requirements and Platform Models”. In: *Software and Systems Modeling* (2022), pp. 2171–2211. DOI: s10270-022-01002-3.
- [293] Jörg Holtmann et al. *The MechatronicUML Requirements Engineering Method: Process and Language*. Technical Report tr-ri-16-351. Fraunhofer Institute for Mechatronic Systems Design and Heinz Nixdorf Institute, 2016.
- [294] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521.
- [295] Gerard J. Holzmann. *The SPIN Model Cecker: Primer and Reference Manual*. Addison-Wesley, 2004. ISBN: 978-0-321-22862-8.
- [296] Robert Höttger et al. “APP4MC: Application Platform Project for Multi- and Many-Core Systems”. In: *it – Information Technology* 59.5 (2017), pp. 243–251. DOI: 10.1515/itit-2017-0019.
- [297] Bahar Houtan et al. “Supporting End-to-End Data Propagation Delay Analysis for TSN-Based Distributed Vehicular Embedded Systems”. In: *Journal of Systems Architecture* 141 (2023), p. 102911. DOI: 10.1016/j.sysarc.2023.102911.
- [298] Jinling Hu et al. “Link Level Performance Comparison Between LTE V2X and DSRC”. In: *Journal of Communications and Information Networks* 2.2 (2017), pp. 101–112. DOI: 10.1007/s41650-017-0022-x.

- [299] Ming Hu et al. “Automated Synthesis of Safe Timing Behaviors for Requirements Models Using CCSL”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.12 (2023), pp. 5127–5140. DOI: 10.1109/TCAD.2023.3285412.
- [300] Wenjia Huai, Xudong Liu, and Hailong Sun. “Towards Trustworthy Composite Service Through Business Process Model Verification”. In: *2010 7th International Conference on Ubiquitous Intelligence & Computing and 7th International Conference on Autonomic & Trusted Computing*. IEEE, 2010, pp. 422–427. DOI: 10.1109/UIC-ATC.2010.114.
- [301] Chuanlin Huang et al. “A MDE-Based Approach to the Safety Verification of Extended SysML Activity Diagram”. In: *Journal of Software* 10.1 (2015), pp. 56–70. DOI: 10.17706/jsw.10.1.56-70.
- [302] Edward Huang, Leon F. McGinnis, and Steven W. Mitchell. “Verifying SysML Activity Diagrams Using Formal Transformation to Petri Nets”. In: *Systems Engineering* 23.1 (2020), pp. 118–135. DOI: 10.1002/sys.21524.
- [303] Lifei Huang and Ten-Hwang Lai. “On the Scalability of IEEE 802.11 Ad Hoc Networks”. In: *Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking & Computing*. ACM, 2002, pp. 173–182. DOI: 10.1145/513800.513822.
- [304] Fabian Hub and Michael Oehl. “Design and Field Test of a Mobile Augmented Reality Human–Machine Interface for Virtual Stops in Shared Automated Mobility On-Demand”. In: *Electronics* 11.17 (2022). DOI: 10.3390/electronics11172687.
- [305] Jérôme Hugues and Guillaume Brau. “Analysis as a First-Class Citizen: An Application to Architecture Description Languages”. In: *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2014, pp. 214–221. DOI: 10.1109/ISORC.2014.60.
- [306] IBM Corporation. *Control Nodes*. <https://www.ibm.com/docs/en/dma?topic=diagrams-control-nodes>. Last accessed on June 19, 2026.
- [307] IBM Corporation. *IBM Engineering Rhapsody Architect for Software*. <https://www.ibm.com/products/engineering-rhapsody-architect-for-software>. Last accessed on June 19, 2026.
- [308] IBM Corporation. *IBM Engineering Rhapsody Architect for Systems Engineers*. <https://www.ibm.com/products/engineering-rhapsody-architect-for-systems-engineers>. Last accessed on June 19, 2026.
- [309] IMAGinE partners. *IMAGinE – Lösungen für kooperatives Fahren*. <https://www.imagine-online.de/>. Last accessed on June 19, 2026.
- [310] INCHRON GmbH. *chronSUITE. The Leading Toolkit for Timing Optimization*. <https://www.inchron.com/chronsuite/>. Last accessed on June 19, 2026.

Bibliography

- [311] Claire Ingram, Richard Payne, and John Fitzgerald. “Architectural Modelling Patterns for Systems of Systems”. In: *Insight* 19.3 (2016), pp. 63–66. DOI: 10.1002/inst.12112.
- [312] Claire Ingram et al. “Integrating an Upgraded Constituent System in a System of Systems: A SysML Case Study”. In: *INCOSE International Symposium* 25.1 (2015), pp. 1193–1208. DOI: 10.1002/j.2334-5837.2015.00124.x.
- [313] Institute of Electrical and Electronics Engineers. “IEEE Approved Draft Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules”. In: *IEEE Standard 1516-2025* (2025). URL: <https://standards.ieee.org/ieee/1516/6687/>.
- [314] Institute of Electrical and Electronics Engineers. “IEEE Guide for Wireless Access in Vehicular Environments (WAVE) Architecture”. In: *IEEE Standard 1609.0-2019* (2019). DOI: 10.1109/IEEESTD.2019.8686445.
- [315] Institute of Electrical and Electronics Engineers. “IEEE Standard for Information Technology – Local and Metropolitan Area Networks – Specific Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments”. In: *IEEE Standard 802.11p-2010* (2010). DOI: 10.1109/IEEESTD.2010.5514475.
- [316] Institute of Electrical and Electronics Engineers. “IEEE Standard for VHDL Language Reference Manual”. In: *IEEE Standard 1076-2008* (2019). DOI: 10.1109/IEEESTD.2019.8938196.
- [317] Institute of Electrical and Electronics Engineers. “IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems”. In: *IEEE Standard 1588-2019* (2019). DOI: 10.1109/IEEESTD.2020.9120376.
- [318] Intecs. *CHESS Modelling Language. A UML/MARTE/SysML Profile*. Version 1.0. 2020. URL: <https://eclipse.dev/chess/publis/CHESSMLprofile.pdf>.
- [319] International Council on Systems Engineering. *INCOSE Systems Engineering Handbook*. Wiley, 2023. ISBN: 978-1-119-81429-0.
- [320] International Council on Systems Engineering. *INCOSE Systems Engineering Vision 2020*. White Paper (INCOSE-TP-2004-004-02. Version 2.03. 2007.
- [321] International Council on Systems Engineering. *INCOSE Systems Engineering Vision 2035*. White Paper. 2021. URL: <https://www.incose.org/docs/default-source/se-vision/incose-se-vision-2035.pdf>.
- [322] International Council on Systems Engineering. *INCOSE Systems of Systems Primer*. Tech. rep. INCOSE-TP-2018-003-01.0. 2018.
- [323] International Organization for Standardization. *Information Processing — Documentation Symbols and Conventions for Data, Program and System Flowcharts, Program Network Charts and System Resources Charts*. ISO Standard 5807. 1985. URL: <https://www.iso.org/standard/11955.html>.

- [324] International Organization for Standardization. *Road Vehicles — Functional Safety*. ISO Standard 26262. 2018. URL: <https://www.iso.org/standard/68383.html>.
- [325] International Organization for Standardization. *Systems and Software Engineering — System of Systems (SoS) Considerations in Life Cycle Stages of a System*. ISO/IEC/IEEE Standard 21839. 2019. URL: <https://www.iso.org/standard/71955.html>.
- [326] International Telecommunication Union. “Message Sequence Chart (MSC)”. In: *ITU-T Recommendation Z.120* (1996). URL: <https://handle.itu.int/11.1002/1000/3920>.
- [327] International Telecommunication Union. “Message Sequence Chart (MSC)”. In: *ITU-T Recommendation Z.120* (2011). URL: <https://handle.itu.int/11.1002/1000/11063>.
- [328] Mo Jamshidi. *Systems of Systems Engineering: Principles and Applications*. CRC Press, 2008. ISBN: 978-1-4200-6589-3.
- [329] Yosr Jarraya and Mourad Debbabi. “Formal Specification and Probabilistic Verification of SysML Activity Diagrams”. In: *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 2012, pp. 17–24. DOI: 10.1109/TASE.2012.34.
- [330] Yosr Jarraya and Mourad Debbabi. “Quantitative and Qualitative Analysis of SysML Activity Diagrams”. In: *International Journal on Software Tools for Technology Transfer* 16.4 (2014), pp. 399–419. DOI: 10.1007/s10009-014-0305-6.
- [331] Yosr Jarraya, Mourad Debbabi, and Jamal Bentahar. “On the Meaning of SysML Activity Diagrams”. In: *2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. IEEE, 2009, pp. 95–105. DOI: 10.1109/ECBS.2009.25.
- [332] Yosr Jarraya et al. “Automatic Verification and Performance Analysis of Time-Constrained SysML Activity Diagrams”. In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*. IEEE, 2007, pp. 515–522. DOI: 10.1109/ECBS.2007.22.
- [333] Yongseok Jeon and Hyogon Kim. “An Explicit Reservation-Augmented Resource Allocation Scheme for C-V2X Sidelink Mode 4”. In: *IEEE Access* 8 (2020), pp. 147241–147255. DOI: 10.1109/ACCESS.2020.3015549.
- [334] Moez Jerbi et al. “An Infrastructure-Free Traffic Information System for Vehicular Networks”. In: *2007 IEEE 66th Vehicular Technology Conference*. IEEE, 2007, pp. 2086–2090. DOI: 10.1109/VETECF.2007.438.
- [335] M. Milton Joe and B. Ramakrishnan. “Review of Vehicular Ad Hoc Network Communication Models Including WVANET (Web VANET) Model and WVANET Future Research Directions”. In: *Wireless Networks* 22.7 (2016), pp. 2369–2386. DOI: 10.1007/s11276-015-1104-z.

Bibliography

- [336] Rolf Johansson et al. *TADL: Timing Augmented Description Language Version 2*. Deliverable D6. TIMMO project, 2009.
- [337] Oday Jubran and Bernd Westphal. “Formal Approach to Guard Time Optimization for TDMA”. In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. ACM, 2013, pp. 223–233. DOI: 10.1145/2516821.2516849.
- [338] Oday Jubran and Bernd Westphal. “Optimizing Guard Time for TDMA in a Wireless Sensor Network-Case Study”. In: *39th Annual IEEE Conference on Local Computer Networks Workshops*. IEEE, 2014, pp. 597–601. DOI: 10.1109/LCNW.2014.6927708.
- [339] Byeonggil Jun et al. “Efficient Coordination for Distributed Discrete-Event Systems”. In: *2024 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2024, pp. 114–118. DOI: 10.1109/MEMOCODE63347.2024.00019.
- [340] Bernhard Kaiser et al. “Contract-Based Design of Embedded Systems Integrating Nominal Behavior and Safety”. In: *Complex Systems Informatics and Modeling Quarterly* 4 (2015), pp. 66–91. DOI: 10.7250/csimq.2015-4.05.
- [341] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. “Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems”. In: *Journal of Object Technology* 18.2 (2019), 2:1–2:20. DOI: 10.5381/jot.2019.18.2.a2.
- [342] Burak Karakaya and Klaus Bengler. “Investigation of Driver Behavior During Minimal Risk Maneuvers of Automated Vehicles”. In: *Proceedings of the 21st Congress of the International Ergonomics Association (IEA 2021)*. Ed. by Nancy L. Black, W. Patrick Neumann, and Ian Noy. Lecture Notes in Networks and Systems 221. Springer, Cham, 2021, pp. 691–700. DOI: 10.1007/978-3-030-74608-7_84.
- [343] Andrew Keane, Bernd Krauskopf, and Claire M. Postlethwaite. “Climate Models with Delay Differential Equations”. In: *Chaos: An Interdisciplinary Journal of Non-linear Science* 27.11 (2017), pp. 114309–1–114309–15. DOI: 10.1063/1.5006923.
- [344] Charles B. Keating and Polinpapilinho F. Katina. “Prevalence of Pathologies in Systems of Systems”. In: *International Journal of System of Systems Engineering* 3.3-4 (2012), pp. 243–267. DOI: 10.1504/IJSSE.2012.052688.
- [345] Charles B. Keating and Polinpapilinho F. Katina. “Systems of Systems Engineering: Prospects and Challenges for the Emerging Field”. In: *International Journal of System of Systems Engineering* 2.2-3 (2011), pp. 234–256. DOI: 10.1504/IJSSE.2011.040556.
- [346] Robert M. Keller. “Formal Verification of Parallel Programs”. In: *Communications of the ACM* 19.7 (1976), pp. 371–384. DOI: 10.1145/360248.360251.
- [347] Yonit Kestent, Zohar Manna, and Amir Pnueli. “Verification of Clocked and Hybrid Systems”. In: *Lectures on Embedded Systems*. Ed. by Grzegorz Rozenberg and Frits W. Vaandrager. Lecture Notes in Computer Science 1494. Springer, Berlin, Heidelberg, 1998, pp. 4–73. DOI: 10.1007/3-540-65193-4_18.

- [348] Carmen Kettwich, Andreas Schrank, and Michael Oehl. “Teleoperation of Highly Automated Vehicles in Public Transport: User-Centered Design of a Human-Machine Interface for Remote-Operation and Its Expert Usability Evaluation”. In: *Multimodal Technologies and Interaction* 5.5 (2021). DOI: 10.3390/mti5050026.
- [349] Carmen Kettwich et al. “A Helping Human Hand: Relevant Scenarios for the Remote Operation of Highly Automated Vehicles in Public Transport”. In: *Applied Sciences* 12.9 (2022). DOI: 10.3390/app12094350.
- [350] Roozbeh Kianfar et al. “Design and Experimental Validation of a Cooperative Driving System in the Grand Cooperative Driving Challenge”. In: *IEEE Transactions on Intelligent Transportation Systems* 13.3 (2012), pp. 994–1007. DOI: 10.1109/TITS.2012.2186513.
- [351] Kirkpatrick, Michael S. *Open Computer Systems Fundamentals: Consensus in Distributed Systems*. <https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/DistConsensus.html>. Last accessed on June 19, 2026.
- [352] Raimund Kirner and Peter Puschner. “Classification of WCET Analysis Techniques”. In: *Eight IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’05)*. 2005, pp. 190–199. DOI: 10.1109/ISORC.2005.19.
- [353] Christoph M. Kirsch and Ana Sokolova. “The Logical Execution Time Paradigm”. In: *Advances in Real-Time Systems*. Ed. by Samarjit Chakraborty and Jörg Eberspächer. Springer, Berlin, Heidelberg, 2012, pp. 103–120. DOI: 10.1007/978-3-642-24349-3_5.
- [354] Martin Klapez, Carlo Augusto Grazia, and Maurizio Casoni. “Application-Level Performance of IEEE 802.11p in Safety-Related V2X Field Trials”. In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 3850–3860. DOI: 10.1109/JIOT.2020.2967649.
- [355] John Klein and Hans van Vliet. “A Systematic Review of System-of-Systems Architecture Research”. In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. ACM, 2013, pp. 13–22. DOI: 10.1145/2465478.2465490.
- [356] Jochen Klose and Hartmut Wittke. “An Automata Based Interpretation of Live Sequence Charts”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Wang Yi. Lecture Notes in Computer Science 2031. Springer, Berlin, Heidelberg, 2001, pp. 512–527. DOI: 10.1007/3-540-45319-9_35.
- [357] Ko-HAF partners. *Ko-HAF – Kooperatives hochautomatisiertes Fahren*. <https://www.ko-haf.de/>. Last accessed on June 19, 2026.
- [358] KoKoVI partners. *KoKoVI – Verkehr als kooperatives und vernetztes System*. <https://www.dlr.de/de/ts/forschung-und-transfer/projekte/kokovi>. Last accessed on June 19, 2026.

- [359] Leonie Köhler and Rolf Ernst. “Improving a Compositional Timing Analysis Framework for Weakly-Hard Real-Time Systems”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 228–240. DOI: 10.1109/RTAS.2019.00027.
- [360] Björn Koopmann, Günter Ehmen, and Philipp Ittershagen. *Entwurfs- und Analysemethoden für eine durchgängige Behandlung von Echtzeit*. Deliverable D5.1. Digitaler Knoten 4.0 project, 2019.
- [361] Björn Koopmann et al. *Beschreibung der Referenzarchitektur und der Migrationsstrategie*. Deliverable D3.1. Digitaler Knoten 4.0 project, 2019.
- [362] Björn Koopmann et al. “Challenges in Achieving Explainability for Cooperative Transportation Systems”. In: *2022 IEEE 30th International Requirements Engineering Conference Workshops (REW)*. IEEE, 2022, pp. 114–119. DOI: 10.1109/REW56159.2022.00028.
- [363] Björn Koopmann et al. “Cooperative Maneuvers of Highly Automated Vehicles at Urban Intersections: A Game-Theoretic Approach”. In: *Proceedings of the 6th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*. Ed. by Karsten Berns, Markus Helfert, and Oleg Gusikhin. SciTePress, 2020, pp. 15–26. DOI: 10.5220/0009351500150026.
- [364] Björn Koopmann et al. *Finale Version der KoKoVI-Referenzarchitektur. Systemarchitektur für mehrstufiges Verkehrsmanagement*. Milestone report MST 4104. KoKoVI project, 2024.
- [365] Hermann Kopetz and Günter Grünsteidl. “TTP – A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems”. In: *FTCS-23 – The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 1993, pp. 524–533. DOI: 10.1109/FTCS.1993.627355.
- [366] Hermann Kopetz and Wilhelm Ochsenreiter. “Clock Synchronization in Distributed Real-Time Systems”. In: *IEEE Transactions on Computers* 36.8 (1987), pp. 933–940. DOI: 10.1109/TC.1987.5009516.
- [367] Ron Koymans. “Specifying Real-Time Properties with Metric Temporal Logic”. In: *Real-Time Systems* 2.4 (1990), pp. 255–299. DOI: 10.1007/BF01995674.
- [368] Lukas Krawczyk et al. “Model-Based Timing Analysis and Deployment Optimization for Heterogeneous Multi-Core Systems Using Eclipse APP4MC”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2019, pp. 44–53. DOI: 10.1109/MODELS-C.2019.00013.
- [369] Janis Kröger and Martin Fränzle. “Mode Management in Contract-Based Design”. In: *SE 2024 – Companion*. Gesellschaft für Informatik, 2024, pp. 31–42. DOI: 10.18420/sw2024-ws_03.

- [370] Janis Kröger et al. “Contract-Based Specification of Mode-Dependent Timing Behavior”. In: *Innovations in Systems and Software Engineering* 20.1 (2024), pp. 31–47. DOI: 10.1007/s11334-023-00531-4.
- [371] Janis Kröger et al. “Handling of Operating Modes in Contract-Based Timing Specifications”. In: *Verification and Evaluation of Computer and Communication Systems*. Ed. by Ayoub Nouri et al. Lecture Notes in Computer Science 13187. Springer, Cham, 2022, pp. 59–74. DOI: 10.1007/978-3-030-98850-0_5.
- [372] Ingolf Heiko Krüger. “Distributed System Design with Message Sequence Charts”. PhD thesis. Technical University of Munich, July 2000.
- [373] Stefan Kugele, Philipp Obergfell, and Eric Sax. “Model-Based Resource Analysis and Synthesis of Service-Oriented Automotive Software Architectures”. In: *Software and Systems Modeling* 20.6 (2021), pp. 1945–1975. DOI: 10.1007/s10270-021-00896-9.
- [374] Stefan Kugele et al. “Verification of Component Architectures Using Mode-Based Contracts”. In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2016, pp. 133–142. DOI: 10.1109/MEMCOD.2016.7797758.
- [375] Marta Kwiatkowska, Gethin Norman, and David Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science 6806. Springer, Berlin, Heidelberg, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.
- [376] Marta Kwiatkowska et al. “Automatic Verification of Real-Time Systems with Discrete Probability Distributions”. In: *Theoretical Computer Science* 282.1 (2002), pp. 101–150. DOI: 10.1016/S0304-3975(01)00046-9.
- [377] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.
- [378] Leslie Lamport and P. M. Melliar-Smith. “Synchronizing Clocks in the Presence of Faults”. In: *Journal of the ACM* 32.1 (1985), pp. 52–78. DOI: 10.1145/2455.2457.
- [379] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176.
- [380] Axel van Lamsweerde and Emmanuel Letier. “From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering”. In: *Radical Innovations of Software and Systems Engineering in the Future*. Ed. by Martin Wirsing, Alexander Knapp, and Simonetta Balsamo. Lecture Notes in Computer Science 2941. Springer, Berlin, Heidelberg, 2004, pp. 325–340. DOI: 10.1007/978-3-540-24626-8_23.

Bibliography

- [381] Jo Ann Lane and Tim Bohn. “Using SysML Modeling to Understand and Evolve Systems of Systems”. In: *Systems Engineering* 16.1 (2013), pp. 87–98. DOI: 10.1002/sys.21221.
- [382] Vincent Langenfeld et al. “Scalable Analysis of Real-Time Requirements”. In: *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 2019, pp. 234–244. DOI: 10.1109/RE.2019.00033.
- [383] Diego Latella, Istvan Majzik, and Mieke Massink. “Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker”. In: *Formal Aspects of Computing* 11 (1999), pp. 637–664. DOI: 10.1007/s001659970003.
- [384] Edward A. Lee. “Cyber Physical Systems: Design Challenges”. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25.
- [385] Edward A. Lee. “Modeling Concurrent Real-Time Processes Using Discrete Events”. In: *Annals of Software Engineering* 7.1 (1999), pp. 25–45. DOI: 10.1023/A:1018998524196.
- [386] 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), pp. 1217–1229. DOI: 10.1109/43.736561.
- [387] Edward A. Lee et al. “Consistency vs. Availability in Distributed Cyber-Physical Systems”. In: *ACM Transactions on Embedded Computing Systems* 22.5s (2023), pp. 1–24. DOI: 10.1145/3609119.
- [388] Edward A. Lee et al. “Quantifying and Generalizing the CAP Theorem”. arXiv:2109.07771 [cs.DC]. Sept. 2021.
- [389] Kwonjong Lee et al. “Latency of Cellular-Based V2X: Perspectives on TTI-Proportional Latency and TTI-Independent Latency”. In: *IEEE Access* 5 (2017), pp. 15800–15809. DOI: 10.1109/ACCESS.2017.2731777.
- [390] Karsten Lemmer, ed. *Neue autoMobilität II. Kooperativer Straßenverkehr und intelligente Verkehrssteuerung für die Mobilität der Zukunft*. utzverlag GmbH, 2019. ISBN: 978-3-8316-4724-8.
- [391] Michael D. Lemmon, Joydeep Ganguly, and Lucia Xia. “Model-Based Clock Synchronization in Networks with Drifting Clocks”. In: *Proceedings of the Pacific Rim International Symposium on Dependable Computing*. IEEE, 2000, pp. 177–184. DOI: 10.1109/PRDC.2000.897300.
- [392] Grace A. Lewis et al. “Requirements Engineering for Systems of Systems”. In: *2009 3rd Annual IEEE Systems Conference*. IEEE, 2009, pp. 247–252. DOI: 10.1109/SYSTEMS.2009.4815806.

- [393] JianQiang Li, YuShun Fan, and MengChu Zhou. “Timing Constraint Workflow Nets for Workflow Analysis”. In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 33.2 (2003), pp. 179–193. DOI: 10.1109/tsmca.2003.811771.
- [394] Christian Lidström and Dilian Gurov. “Contract Based Embedded Software Design”. In: *Theoretical Aspects of Software Engineering*. Ed. by Cristina David and Meng Sun. Lecture Notes in Computer Science 13931. Springer, Cham, 2023, pp. 77–94. DOI: 10.1007/978-3-031-35257-7_5.
- [395] Lucas Lima, André Didier, and Márcio Cornélio. “A Formal Semantics for SysML Activity Diagrams”. In: *Formal Methods: Foundations and Applications*. Ed. by Juliano Iyoda and Leonardo de Moura. Lecture Notes in Computer Science 8195. Springer, Berlin, Heidelberg, 2013, pp. 179–194. DOI: 10.1007/978-3-642-41071-0_13.
- [396] Marten Lohstroh et al. “A Language for Deterministic Coordination Across Multiple Timelines”. In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 2020, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232939.
- [397] Marten Lohstroh et al. “Logical Time for Reactive Software”. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. ACM, 2023, pp. 313–318. DOI: 10.1145/3576914.3587494.
- [398] Marten Lohstroh et al. “Toward a Lingua Franca for Deterministic Concurrent Systems”. In: *ACM Transactions on Embedded Computing Systems* 20.4 (2021), 36:1–36:27. DOI: 10.1145/3448128.
- [399] Nancy A. Lynch and Hagit Attiya. “Using Mappings to Prove Timing Properties”. In: *Distributed Computing* 6.2 (1992), pp. 121–139. DOI: 10.1007/BF02252683.
- [400] Yue Ma et al. “System Synthesis from AADL Using Polychrony”. In: *2011 Electronic System Level Synthesis Conference (ESLsyn)*. IEEE, 2011. DOI: 10.1109/ESLsyn.2011.5952285.
- [401] Jeff Magee et al. “Specifying Distributed Software Architectures”. In: *Software Engineering – ESEC ’95*. Ed. by Wilhelm Schäfer and Pere Botella. Lecture Notes in Computer Science 989. Springer, Berlin, Heidelberg, 1995, pp. 137–153.
- [402] Mark W. Maier. “Architecting Principles for Systems-of-Systems”. In: *INCOSE International Symposium* 6.1 (1996), pp. 565–573. DOI: 10.1002/j.2334-5837.1996.tb02054.x.
- [403] Mouna Makni et al. “Satisfaction and Coherence of Deadline Constraints in Inter-Organizational Workflows”. In: *On the Move to Meaningful Internet Systems: OTM 2010*. Ed. by Robert Meersman, Tharam Dillon, and Pilar Herrero. Lecture Notes in Computer Science 6426. Springer, Berlin, Heidelberg, 2010, pp. 523–539. DOI: 10.1007/978-3-642-16934-2_39.

Bibliography

- [404] Oded Maler and Dejan Ničković. “Monitoring Properties of Analog and Mixed-Signal Circuits”. In: *International Journal on Software Tools for Technology Transfer* 15.3 (2013), pp. 247–268. DOI: 10.1007/s10009-012-0247-9.
- [405] Oded Maler, Dejan Nickovic, and Amir Pnueli. “Real Time Temporal Logic: Past, Present, Future”. In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Paul Pettersson and Wang Yi. Lecture Notes in Computer Science 3829. Springer, Berlin, Heidelberg, 2005, pp. 2–16. DOI: 10.1007/11603009_2.
- [406] Jitendra Malik and Thomas O. Binford. “Reasoning in Time and Space”. In: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., 1983, pp. 343–345.
- [407] Frédéric Mallet. “Clock Constraint Specification Language: Specifying Clock Constraints with UML/MARTE”. In: *Innovations in Systems and Software Engineering* 4.3 (2008), pp. 309–314. DOI: 10.1007/s11334-008-0055-2.
- [408] Frédéric Mallet. “Time: It Is Only Logical!” In: *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday*. Ed. by Jonathan P. Bowen, Qin Li, and Qiwen Xu. Lecture Notes in Computer Science 14080. Springer, Cham, 2023, pp. 323–347. DOI: 10.1007/978-3-031-40436-8_12.
- [409] Valerian Mannoni et al. “A Comparison of the V2X Communication Systems: ITS-G5 and C-V2X”. In: *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*. IEEE, 2019, pp. 1–5. DOI: 10.1109/VTCSpring.2019.8746562.
- [410] S. Manu and P. Sivraj. “Performance Comparison of Communication Technologies for V2X Applications”. In: *2020 5th International Conference on Communication and Electronics Systems (ICCES)*. IEEE, 2020, pp. 356–362. DOI: 10.1109/ICCES48766.2020.9137879.
- [411] Florence Maraninchi and Yann Rémond. “Mode-Automata: About Modes and States for Reactive Systems”. In: *Programming Languages and Systems*. Ed. by Chris Hankin. Lecture Notes in Computer Science 1381. Springer, Berlin, Heidelberg, 1998, pp. 185–199. DOI: 10.1007/BFb0053571.
- [412] Diego Marmosler and Silvio Degenhardt. “Verifying Patterns of Dynamic Architectures Using Model Checking”. In: *Formal Engineering Approaches to Software Components and Architectures*. Ed. by Jan Kofrön and Jana Tumova. Electronic Proceedings in Theoretical Computer Science 245. 2017, pp. 16–30. DOI: 10.4204/EPTCS.245.2.
- [413] Keith Marzullo and Gil Neiger. “Detection of Global State Predicates”. In: *Distributed Algorithms*. Ed. by Sam Toueg, Paul G. Spirakis, and Lefteris Kirousis. Lecture Notes in Computer Science 579. Springer, Berlin, Heidelberg, 1992, pp. 254–272. DOI: 10.1007/BFb0022452.
- [414] Keith Marzullo and Susan Owicki. “Maintaining the Time in a Distributed System”. In: *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1983, pp. 295–305. DOI: 10.1145/800221.806730.

- [415] Nenad Medvidovic. “ADLs and Dynamic Architecture Changes”. In: *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*. ACM, 1996, pp. 24–27. DOI: 10.1145/243327.243340.
- [416] Till Menzel, Gerrit Bagschik, and Markus Maurer. “Scenarios for Development, Test and Validation of Automated Vehicles”. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 1821–1827. DOI: 10.1109/IVS.2018.8500406.
- [417] Mercedes-Benz AG. *DRIVE PILOT. Support Speed of Up to 95 km/h on German Motorways*. <https://group.mercedes-benz.com/technology/autonomous-driving/driving/drive-pilot-95-kmh.html>. Last accessed on June 19, 2026.
- [418] Philip Merlin and David Farber. “Recoverability of Communication Protocols – Implications of a Theoretical Study”. In: *IEEE Transactions on Communications* 24.9 (1976), pp. 1036–1043. DOI: 10.1109/TCOM.1976.1093424.
- [419] MetaCase. *MetaEdit+ Domain-Specific Modeling (DSM) Environment*. <https://www.metacase.com/products.html>. Last accessed on June 19, 2026.
- [420] Max-Arno Meyer et al. “Scenario- and Model-Based Systems Engineering Procedure for the SOTIF-Compliant Design of Automated Driving Functions”. In: *2022 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2022, pp. 1599–1604. DOI: 10.1109/IV51971.2022.9827151.
- [421] Roland Meyer et al. “Model Checking Duration Calculus: A Practical Approach”. In: *Formal Aspects of Computing* 20.4 (2008), pp. 481–505. DOI: 10.1007/s00165-008-0082-7.
- [422] Manabu Mikami et al. “Field Experimental Evaluation on Latency and Reliability Performance of 5G NR V2V Direct Communication in Real Express Highway Environment”. In: *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*. IEEE, 2020, pp. 1–5. DOI: 10.1109/VTC2020-Spring48590.2020.9129379.
- [423] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Taylor & Francis, 2006. ISBN: 0-8493-5805-1.
- [424] Robin Milner, Joachim Parrow, and David Walker. “A Calculus of Mobile Processes, I”. In: *Information and Computation* 100.1 (1992), pp. 1–40. DOI: 10.1016/0890-5401(92)90008-4.
- [425] Robin Milner, Joachim Parrow, and David Walker. “A Calculus of Mobile Processes, II”. In: *Information and Computation* 100.1 (1992), pp. 41–77. DOI: 10.1016/0890-5401(92)90009-5.
- [426] Ministry of Defense. *MOD Architecture Framework*. <https://www.gov.uk/guidance/mod-architecture-framework>. Last accessed on June 19, 2026.
- [427] Modelica Association. *Modelica Language*. <https://modelica.org/language/>. Last accessed on June 19, 2026.

Bibliography

- [428] Ahmad Mohsin et al. “Modeling Approaches for System-of-Systems Dynamic Architecture: Overview, Taxonomy and Future Prospects”. In: *2019 14th Annual Conference System of Systems Engineering (SoSE)*. IEEE, 2019, pp. 49–56. DOI: 10.1109/SYSOSE.2019.8753877.
- [429] Aloysius K. Mok and Guangtian Liu. “Efficient Run-Time Monitoring of Timing Constraints”. In: *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. IEEE, 1997, pp. 252–262. DOI: 10.1109/RTTAS.1997.601363.
- [430] Adam Molin et al. “Specification-Guided Critical Scenario Identification for Automated Driving”. In: *Formal Methods*. Ed. by Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker. Lecture Notes in Computer Science 14000. Springer, Cham, 2023, pp. 610–621. DOI: 10.1007/978-3-031-27481-7_35.
- [431] Ehsan Moradi-Pari et al. “DSRC Versus LTE-V2X: Empirical Performance Analysis of Direct Vehicular Communication Technologies”. In: *IEEE Transactions on Intelligent Transportation Systems* 24.5 (2023), pp. 4889–4903. DOI: 10.1109/TITS.2023.3247339.
- [432] Marco Mori et al. “Systems-of-Systems Modeling Using a Comprehensive Viewpoint-Based SysML Profile”. In: *Journal of Software: Evolution and Process* 30.3 (2018), e1878. DOI: 10.1002/smr.1878.
- [433] Michel Morold et al. “Requirements on Delay of VRU Context Detection for Cooperative Collision Avoidance”. In: *2020 IEEE 92nd Vehicular Technology Conference (VTC2020-Fall)*. IEEE, 2020, pp. 1–5. DOI: 10.1109/VTC2020-Fall149728.2020.9348627.
- [434] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödín. “Support for End-to-End Response-Time and Delay Analysis in the Industrial Tool Suite: Issues, Experiences and a Case Study”. In: *Computer Science and Information Systems* 10.1 (2013), pp. 453–482. DOI: 10.2298/CSIS120614011M.
- [435] Saad Mubeen et al. “Supporting Timing Analysis of Vehicular Embedded Systems Through the Refinement of Timing Constraints”. In: *Software & Systems Modeling* 18 (2019), pp. 39–69. DOI: 10.1007/s10270-017-0579-8.
- [436] Peter Munk and Arne Nordmann. “Model-based Safety Assessment with SysML and Component Fault Trees: Application and Lessons Learned”. In: *Software and Systems Modeling* (2020), pp. 889–910. DOI: 10.1007/s10270-020-00782-w.
- [437] Jens Muttersbach et al. “Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-Chip Systems”. In: *Twelfth Annual IEEE International ASIC/SOC Conference*. IEEE, 1999, pp. 317–321. DOI: 10.1109/ASIC.1999.806526.
- [438] Sumit Nain and Moshe Y. Vardi. “Branching vs. Linear Time: Semantical Perspective”. In: *Automated Technology for Verification and Analysis*. Ed. by Kedar S. Namjoshi et al. Lecture Notes in Computer Science 4762. Springer, Berlin, Heidelberg, 2007, pp. 19–34. DOI: 10.1007/978-3-540-75596-8_4.

- [439] John Nash. “Non-Cooperative Games”. In: *Annals of Mathematics* 54.2 (1951), pp. 286–295. DOI: 10.2307/1969529.
- [440] National Science Foundation. *Cyber-Physical Systems (CPS)*. Program Solicitation NSF 11-516. URL: <https://www.nsf.gov/pubs/2011/nsf11516/nsf11516.htm>.
- [441] Nicolas Navet, Ye-Quiong Song, and Françoise Simonot. “Worst-Case Deadline Failure Probability in Real-Time Applications Distributed over Controller Area Network”. In: *Journal of Systems Architecture* 46.7 (2000), pp. 607–617. DOI: 10.1016/S1383-7621(99)00016-8.
- [442] Cornelius Ncube and Soo Ling Lim. “On Systems of Systems Engineering: A Requirements Engineering Perspective and Research Agenda”. In: *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 2018, pp. 112–123. DOI: 10.1109/RE.2018.00021.
- [443] Faranak Nejati et al. “Handling State Space Explosion in Component-Based Software Verification: A Review”. In: *IEEE Access* 9 (2021), pp. 77526–77544. DOI: 10.1109/ACCESS.2021.3081742.
- [444] Shiva Nejati et al. “Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 1015–1025. DOI: 10.1145/3338906.3340444.
- [445] Valdemar Vicente Graciano Neto et al. “Model-Based Engineering & Simulation of Software-Intensive Systems-of-Systems: Experience Report and Lessons Learned”. In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ACM, 2018. DOI: 10.1145/3241403.3241432.
- [446] Christian Neurohr et al. “Criticality Analysis for the Verification and Validation of Automated Vehicles”. In: *IEEE Access* 9 (2021), pp. 18016–18041. DOI: 10.1109/ACCESS.2021.3053159.
- [447] Guillem Bernat Nicolau. “Specification and Analysis of Weakly-Hard Real-Time Systems”. PhD thesis. University of the Balearic Islands, Jan. 1998.
- [448] Claus Ballegaard Nielsen et al. “Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions”. In: *ACM Computing Surveys* 48.2 (2015). DOI: 10.1145/2794381.
- [449] Josef Nilsson et al. “Functional Safety for Cooperative Systems”. In: *SAE World Congress & Exhibition 2013*. SAE International, 2013. DOI: 10.4271/2013-01-0197.
- [450] North Atlantic Treaty Organization. *NATO Architecture Framework*. Version 4.1. 2026. URL: <https://www.nato.int/content/dam/nato/webready/documents/publications-and-reports/NATO-Architecture-Framework-v4-1-en.pdf>.

Bibliography

- [451] Pierluigi Nuzzo and Alberto L. Sangiovanni-Vincentelli. “Hierarchical System Design with Vertical Contracts”. In: *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Ed. by Marten Lohstroh, Patricia Derler, and Marjan Sirjani. Lecture Notes in Computer Science 10760. Springer, Cham, 2018, pp. 360–382. DOI: 10.1007/978-3-319-95246-8_22.
- [452] Pierluigi Nuzzo et al. “A Contract-Based Methodology for Aircraft Electric Power System Design”. In: *IEEE Access* 2 (2013), pp. 1–25. DOI: 10.1109/ACCESS.2013.2295764.
- [453] Object Management Group, Inc. *Business Process Model and Notation (BPMN)*. Version 2.0.2. 2013. URL: <https://www.omg.org/spec/BPMN/2.0.2/PDF>.
- [454] Object Management Group, Inc. *OMG Systems Modeling Language (OMG SysML™)*. Version 1.6. 2018. URL: <https://www.omg.org/spec/SysML/1.6/PDF>.
- [455] Object Management Group, Inc. *OMG® Unified Modeling Language (OMG UML®)*. Version 2.5.1. 2017. URL: <https://www.omg.org/spec/UML/2.5.1/PDF>.
- [456] Object Management Group, Inc. *Reusable Asset Specification*. Version 2.2. 2005. URL: <https://www.omg.org/spec/RAS/2.2/PDF>.
- [457] Object Management Group, Inc. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Version 1.2. 2018. URL: <https://www.omg.org/spec/MARTE/1.2/PDF>.
- [458] Object Management Group, Inc. *Unified Profile for DoDAF and MODAF (UPDM)*. Version 2.1. 2013. URL: <https://www.omg.org/spec/UPDM/2.1/PDF>.
- [459] Reza Olfati-Saber, J. Alex Fax, and Richard M. Murray. “Consensus and Cooperation in Networked Multi-Agent Systems”. In: *Proceedings of the IEEE* 95.1 (2007), pp. 215–233. DOI: 10.1109/JPROC.2006.887293.
- [460] Open SystemC Initiative (OSCI). *SystemC*. <https://sourceforge.net/projects/systemc/>. Last accessed on June 19, 2026.
- [461] Flavio Oquendo. “Formally Describing the Architectural Behavior of Software-Intensive Systems-of-Systems with SosADL”. In: *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2016, pp. 13–22. DOI: 10.1109/ICECCS.2016.012.
- [462] Flavio Oquendo. “Formally Describing the Software Architecture of Systems-of-Systems with SosADL”. In: *2016 11th System of Systems Engineering Conference (SoSE)*. IEEE, 2016, pp. 1–6. DOI: 10.1109/SYSOSE.2016.7542926.
- [463] Flavio Oquendo. “ π -ADL: An Architecture Description Language Based on the Higher-Order Typed π -Calculus for Specifying Dynamic and Mobile Software Architectures”. In: *ACM SIGSOFT Software Engineering Notes* 29.3 (2004), pp. 1–14. DOI: 10.1145/986710.986728.
- [464] Flavio Oquendo. “ π -Calculus for SoS: A Foundation for Formally Describing Software-Intensive Systems-of-Systems”. In: *2016 11th System of Systems Engineering Conference (SoSE)*. IEEE, 2016, pp. 1–6. DOI: 10.1109/SYSOSE.2016.7542925.

- [465] Joël Ouaknine and James Worrell. “Some Recent Results in Metric Temporal Logic”. In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Franck Cassez and Claude Jard. Lecture Notes in Computer Science 5215. Springer, Berlin, Heidelberg, 2008, pp. 1–13. DOI: 10.1007/978-3-540-85778-5_1.
- [466] Samir Ouchani, Otmane Ait Mohamed, and Mourad Debbabi. “Efficient Probabilistic Abstraction for SysML Activity Diagrams”. In: *Software Engineering and Formal Methods*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Springer, Berlin, Heidelberg, 2012, pp. 263–277. DOI: 10.1007/978-3-642-33826-7_18.
- [467] Samir Ouchani, Otmane Aït Mohamed, and Mourad Debbabi. “A Formal Verification Framework for SysML Activity Diagrams”. In: *Expert Systems with Applications* 41.6 (2014), pp. 2713–2728. DOI: 10.1016/j.eswa.2013.10.064.
- [468] Samir Ouchani, Otmane Aït Mohamed, and Mourad Debbabi. “A Property-Based Abstraction Framework for SysML Activity Diagrams”. In: *Knowledge-Based Systems* 56 (2014), pp. 328–343. DOI: 10.1016/j.knosys.2013.11.016.
- [469] Saheb Pal et al. “Cooperation Delay Induced Chaos in an Ecological System”. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 30.8 (2020), pp. 083124–1–083124–21. DOI: 10.1063/5.0012880.
- [470] Rob Palin et al. “ISO 26262 Safety Cases: Compliance and Assurance”. In: *6th IET International Conference on System Safety 2011*. IEEE, 2011, pp. 1–6. DOI: 10.1049/cp.2011.0251.
- [471] Xing Pan, Baoshi Yin, and Jianmi Hu. “Modeling and Simulation for SoS Based on the DoDAF Framework”. In: *Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*. IEEE, 2011, pp. 1283–1287. DOI: 10.1109/ICRMS.2011.5979468.
- [472] Oscar Pastor et al. “Model-Driven Development”. In: *Informatik-Spektrum* 31 (2008), pp. 394–407.
- [473] Christian Pek et al. “Using Online Verification to Prevent Autonomous Vehicles from Causing Accidents”. In: *Nature Machine Intelligence* 2.9 (2020), pp. 518–528. DOI: 10.1038/s42256-020-0225-y.
- [474] Marie-Agnès Peraldi-Frati et al. “A Timing Model for Specifying Multi Clock Automotive Systems: The Timing Augmented Description Language v2”. In: *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. IEEE, 2012, pp. 230–239. DOI: 10.1109/ICECCS20050.2012.6299218.
- [475] Simon Perathoner. “Modular Performance Analysis of Embedded Real-Time Systems. Improving Modeling Scope and Accuracy”. PhD thesis. Swiss Federal Institute of Technology Zurich, July 2011.
- [476] Kenneth J. Perry and Sam Toueg. “Distributed Agreement in the Presence of Processor and Communication Faults”. In: *IEEE Transactions on Software Engineering* SE-12.3 (1986), pp. 477–482. DOI: 10.1109/TSE.1986.6312888.

Bibliography

- [477] Franck Petitedemange, Isabelle Borne, and Jérémy Buisson. “Modeling System of Systems Configurations”. In: *2018 13th Annual Conference on System of Systems Engineering (SoSE)*. IEEE, 2018, pp. 392–399. DOI: 10.1109/SYSOSE.2018.8428737.
- [478] Carl Adam Petri. “Kommunikation mit Automaten”. PhD thesis. Technical University of Darmstadt, June 1962.
- [479] Ana Petrovska, Thomas Hutzelmann, and Stefan Kugele. “A Theoretical Framework for Self-Adaptive Systems: Specifications, Formalisation, and Architectural Implications”. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2023, pp. 1440–1449. DOI: 10.1145/3555776.3577665.
- [480] PivotPoint Technology Corporation. *SysML FAQ: What Is an Activity Diagram (ACT) and How Is It Used?* <https://sysml.org/sysml-faq/what-is-activity-diagram.html>. Last accessed on June 19, 2026.
- [481] Jeroen Ploeg and Redmer de Haan. “Cooperative Automated Driving: From Platooning to Maneuvering”. In: *Proceedings of the 5th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*. Ed. by Oleg Gusikhin and Markus Helfert. SciTePress, 2019, pp. 5–10. DOI: 10.5220/0008346300050010.
- [482] Jeroen Ploeg et al. “Cooperative Automated Maneuvering at the 2016 Grand Cooperative Driving Challenge”. In: *IEEE Transactions on Intelligent Transportation Systems* 19.4 (2018), pp. 1213–1226. DOI: 10.1109/TITS.2017.2765669.
- [483] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.
- [484] Tony Polinori, Tony Rice, and Marc Monfalcone. *Hazard Analysis of Concept Heavy-Truck Platooning Systems*. Final Report DOT HS 813 065. National Traffic Highway Safety Administration, 2021.
- [485] Traian Pop et al. “Timing Analysis of the FlexRay Communication Protocol”. In: *Real-Time Systems* 39.1–3 (2008), pp. 205–235. DOI: 10.1007/s11241-007-9040-3.
- [486] Louchka Popova-Zeugmann. “Timed Petri Nets”. In: *Time and Petri Nets*. 2013, pp. 139–172. DOI: 10.1007/978-3-642-41115-1_4.
- [487] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. “Concurrency in Synchronous Systems”. In: *Formal Methods in System Design* 28.2 (2006), pp. 111–130. DOI: 10.1007/s10703-006-7844-8.
- [488] Heinz Pozewaunig, Johann Eder, and Walter Liebhart. “ePERT: Extending PERT for Workflow Management Systems”. In: *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems*. Ed. by Cornelis Joost van Rijsbergen. 1997, pp. 1–13. DOI: 10.14236/ewic/ADBIS1997.34.

- [489] Wolfgang Pree and Josef Templ. “Modeling with the Timing Definition Language (TDL)”. In: *Model-Driven Development of Reliable Automotive Services*. Ed. by Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. Lecture Notes in Computer Science 4922. Springer, Berlin, Heidelberg, 2008, pp. 133–144. DOI: 10.1007/978-3-540-70930-5_9.
- [490] Robert Protzmann. “V2X Communication in Heterogeneous Networks”. PhD thesis. Technical University of Berlin, Mar. 2011.
- [491] Robert Protzmann, Björn Schunemann, and Ilja Radusch. “The Influences of Communication Models on the Simulated Effectiveness of V2X Applications”. In: *IEEE Communications Magazine* 49.11 (2011), pp. 149–155. DOI: 10.1109/MCOM.2011.6069722.
- [492] Reddy G. Rajeswar and Ramanathan R. “An Empirical Study on MAC Layer in IEEE 802.11p/WAVE Based Vehicular Ad Hoc Networks”. In: *Procedia Computer Science* 143 (2018), pp. 720–727. DOI: 10.1016/j.procs.2018.10.443.
- [493] Ibrahim Rashdan et al. “Performance Evaluation of Vehicle-to-Vehicle Communication for Cooperative Collision Avoidance at Urban Intersections”. In: *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)*. IEEE, 2017, pp. 1–5. DOI: 10.1109/VTCFall.2017.8288274.
- [494] Swarn Singh Rathour et al. “Runtime Active Safety Risk-Assessment of Highly Autonomous Vehicles for Safe Nominal Behavior”. In: *SAE World Congress Experience 2020*. SAE International, 2020. DOI: 10.4271/2020-01-0107.
- [495] Andreas Rauch, Felix Klanner, and Klaus Dietmayer. “Analysis of V2X Communication Parameters for the Development of a Fusion Architecture for Cooperative Perception Systems”. In: *2011 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2011, pp. 685–690. DOI: 10.1109/IVS.2011.5940479.
- [496] Philipp Reinkemeier et al. “A Pattern-Based Requirement Specification Language: Mapping Automotive Specific Timing Requirements”. In: *Lecture Notes in Informatics*. Ed. by Gesellschaft für Informatik. P-184. Köllen Druck + Verlag GmbH, 2011, pp. 99–108.
- [497] Wei Ren and Randal W. Beard. “Consensus Seeking in Multiagent Systems Under Dynamically Changing Interaction Topologies”. In: *IEEE Transactions on Automatic Control* 50.5 (2005), pp. 655–661. DOI: 10.1109/TAC.2005.846556.
- [498] Wendi Ren et al. “A Stochastic Model of Cascading Failure Dynamics in Communication Networks”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.5 (2018), pp. 632–636. DOI: 10.1109/TCSII.2018.2822049.
- [499] Carlos H. Rentel and Thomas Kunz. “A Clock-Sampling Mutual Network Time-Synchronization Algorithm for Wireless Ad Hoc Networks”. In: *2005 IEEE Wireless Communications and Networking Conference*. Vol. 1. IEEE, 2005, pp. 638–644. DOI: 10.1109/WCNC.2005.1424575.

Bibliography

- [500] Ralf H. Reussner, Steffen Becker, and Viktoria Firus. “Component Composition with Parametric Contracts”. In: *Tagungsband der Net.ObjectDays*. 2004, pp. 155–169. URL: <https://sdqweb.ipd.kit.edu/publications/pdfs/reussner2004f.pdf>.
- [501] Ralf H. Reussner, Jens Happe, and Annegret Habel. “Modelling Parametric Contracts and the State Space of Composite Components by Graph Grammars”. In: *Fundamental Approaches to Software Engineering*. Ed. by Maura Cerioli. Lecture Notes in Computer Science 3442. Springer, Berlin, Heidelberg, 2005, pp. 80–95. DOI: 10.1007/978-3-540-31984-9_7.
- [502] Fabíola Gonçalves C. Ribeiro et al. “Model-Based Requirements Specification of Real-Time Systems with UML, SysML and MARTE”. In: *Software and Systems Modeling* 17.1 (2018), pp. 343–361. DOI: s10270-016-0525-1.
- [503] Raphael Riebl et al. “Artery: Extending Veins for VANET Applications”. In: *2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*. IEEE, 2015, pp. 450–456. DOI: 10.1109/MTITS.2015.7223293.
- [504] George F. Riley and Thomas R. Henderson. “The ns-3 Network Simulator”. In: *Modeling and Tools for Network Simulation*. Ed. by Klaus Wehrle, Mesut Güneş, and James Gross. Springer, Berlin, Heidelberg, 2010, pp. 15–34. DOI: 10.1007/978-3-642-12331-3_2.
- [505] Jack Ring and Azad M. Madni. “Key Challenges and Opportunities in ‘System of Systems’ Engineering”. In: *2005 IEEE International Conference on Systems, Man and Cybernetics*. IEEE, 2005, pp. 973–978. DOI: 10.1109/ICSMC.2005.1571272.
- [506] Steven J. Ring. “Activity-Based Methodology for Development and Analysis of Integrated DoD Architectures”. In: *Handbook of Enterprise Systems Architecture in Practice*. Ed. by Pallab Saha. IGI Global, 2007, pp. 85–113. DOI: 10.4018/978-1-59904-189-6.ch005.
- [507] Harold A. Rosenberg and Kang G. Shin. “Software Fault Injection and Its Application in Distributed Systems”. In: *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 1993, pp. 208–217. DOI: 10.1109/FTCS.1993.627324.
- [508] Pierre Roux et al. “System Level Analysis for ITS-G5 and LTE-V2X Performance Comparison”. In: *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2019, pp. 1–9. DOI: 10.1109/MASS.2019.00010.
- [509] Michael A. Ruder et al. “On the Coexistence Between LTE-V2X Sidelink and ITS-G5”. In: *2021 IEEE Vehicular Networking Conference (VNC)*. IEEE, 2021, pp. 162–169. DOI: 10.1109/VNC52810.2021.9644653.
- [510] SAE International. “Architecture Analysis & Design Language (AADL)”. In: *SAE Standard AS5506D* (2022). DOI: 10.4271/AS5506D.

- [511] SAE International. *SAE Levels of Driving AutomationTM Refined for Clarity and International Audience*. <https://www.sae.org/news/blog/sae-levels-driving-automation-clarity-refinements>. Last accessed on June 19, 2026.
- [512] SAE International. “Taxonomy and Definitions for Terms Related to Cooperative Driving Automation for On-Road Motor Vehicles”. In: *SAE Standard J3216_202107* (2021). DOI: 10.4271/J3216_202107.
- [513] SAE International. “Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles”. In: *SAE Standard J3016_202104* (2021). DOI: 10.4271/J3016_202104.
- [514] SAE International. “V2X Communications Message Set Dictionary”. In: *SAE Standard J2735_202409* (2024). DOI: 10.4271/J2735_202409.
- [515] Yassmina Saadna and Ali Behloul. “An Overview of Traffic Sign Detection and Classification Methods”. In: *International Journal of Multimedia Information Retrieval* 6 (2017), pp. 193–210. DOI: 10.1007/s13735-017-0129-8.
- [516] Mersedeh Sadeghi et al. “Interoperability of Heterogeneous Systems of Systems: Review of Challenges, Emerging Requirements and Options”. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2023, pp. 741–750. DOI: 10.1145/3555776.3577692.
- [517] Andrew P. Sage and Christopher D. Cuppan. “On the Systems Engineering and Management of Systems of Systems and Federations of Systems”. In: *Information Knowledge Systems Management* 2.4 (2001), pp. 325–345. DOI: 10.5555/1234195.1234200.
- [518] Francesca Saglietti et al. “Model-Based Design and Testing of Decisional Autonomy and Cooperation in Cyber-Physical Systems”. In: *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2015, pp. 479–483. DOI: 10.1109/SEAA.2015.68.
- [519] Fatma Salem, Yassin Elhillali, and Smail Niar. “Efficient Modelling of IEEE 802.11p MAC Output Process for V2X Interworking Enhancement”. In: *IET Networks* 7.4 (2018), pp. 210–219. DOI: 10.1049/iet-net.2017.0228.
- [520] Stephan Sand et al. “Radio Interference Measurements for Urban Cooperative Intelligent Transportation Systems”. In: *2021 IEEE 94th Vehicular Technology Conference (VTC2021-Fall)*. IEEE, 2021. DOI: 10.1109/VTC2021-Fall152928.2021.9625167.
- [521] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. “Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems”. In: *European Journal of Control* 18.3 (2012), pp. 217–238. DOI: 10.3166/ejc.18.217-238.

Bibliography

- [522] Sergio Saponara and Alessio Gagliardi. “Modeling and Analysis of 802.11p Physical Layer for V2X Connected Transport Systems Considering Harsh Operating Conditions and HW Device Performance”. In: *2018 International Conference of Electrical and Electronic Technologies for Automotive*. IEEE, 2018, pp. 1–6. DOI: 10.23919/EETA.2018.8493238.
- [523] Raja Sattiraju et al. “Link Level Performance Comparison of C-V2X and ITS-G5 for Vehicular Channel Models”. In: *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*. IEEE, 2020, pp. 1–7. DOI: 10.1109/VTC2020-Spring48590.2020.9129366.
- [524] Oliver Sawade et al. “A Cooperative Active Blind Spot Assistant as Example for Next-Gen Cooperative Driver Assistance Systems (CoDAS)”. In: *2014 IEEE Intelligent Vehicles Symposium Proceedings (IV)*. IEEE, 2014, pp. 76–81. DOI: 10.1109/IVS.2014.6856570.
- [525] Oliver Scheickl, Christoph Ainhauser, and Peter Gliwa. “Tool Support for Seamless System Development Based on AUTOSAR Timing Extensions”. In: *Embedded Real-Time Software and Systems*. 2012.
- [526] Ina Schieferdecker. “Model-Based Testing”. In: *IEEE Software* 29.1 (2012), pp. 14–18. DOI: 10.1109/MS.2012.13.
- [527] Marc Schindewolf et al. “A Comparison of Architecture Paradigms for Dynamic Reconfigurable Automotive Networks”. In: *2022 International Conference on Connected Vehicle and Expo (ICCVE)*. IEEE, 2022, pp. 1–7. DOI: 10.1109/ICCVE52871.2022.9742775.
- [528] Julian Schindler et al. “Infrastructure-Supported Cooperative Automated Driving in Transition Areas”. In: *2020 IEEE Vehicular Networking Conference (VNC)*. IEEE, 2020, pp. 1–8. DOI: 10.1109/VNC51378.2020.9318392.
- [529] Rainer Schlör and Werner Damm. “Specification and Verification of System-Level Hardware Designs Using Timing Diagrams”. In: *1993 European Conference on Design Automation with the European Event in ASIC Design*. IEEE, 1993, pp. 518–524. DOI: 10.1109/EDAC.1993.386409.
- [530] Rolf Schmedes, Philipp Ittershagen, and Kim Grüttner. “Towards Distributed Runtime Monitoring with C++ Contracts”. In: *Proceedings of the International Conference on Omni-Layer Intelligent Systems*. ACM, 2019, pp. 141–145. DOI: 10.1145/3312614.3312645.
- [531] David Schmelter et al. “Trustful Model-Based Information Exchange in Collaborative Engineering”. In: *Systems, Software and Services Process Improvement*. Ed. by Murat Yilmaz et al. Springer, Cham, 2023, pp. 156–170. DOI: 10.1007/978-3-031-42307-9_12.
- [532] Wilbur Schramm. “How Communication Works”. In: *The Process and Effects of Mass Communication*. Ed. by Wilbur Schramm. University of Illinois Press, 1960, pp. 3–26.

- [533] Reinhard Schwarz and Friedemann Mattern. “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail”. In: *Distributed Computing* 7.3 (1994), pp. 149–174. DOI: 10.1007/BF02277859.
- [534] Peter M. Shames, Marc A. Sarrel, and Sanford Friedenthal. “Modeling Systems-of-Systems Interfaces with SysML”. In: *14th International Conference on Space Operations (SpaceOps)*. American Institute of Aeronautics and Astronautics, Inc., 2016. DOI: 10.2514/6.2016-2500.
- [535] Claude Elwood Shannon. “A Mathematical Theory of Communication”. In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.
- [536] Takayuki Shimizu et al. “Comparative Analysis of DSRC and LTE-V2X PC5 Mode 4 with SAE Congestion Control”. In: *2020 IEEE Vehicular Networking Conference (VNC)*. IEEE, 2020. DOI: 10.1109/VNC51378.2020.9318353.
- [537] Jan Sieber and Bernd Krauskopf. “Complex Balancing Motions of an Inverted Pendulum Subject to Delayed Feedback Control”. In: *Physica D: Nonlinear Phenomena* 197.3 (2004), pp. 332–345. DOI: 10.1016/j.physd.2004.07.007.
- [538] Eduardo Silva, Thais Batista, and Everton Cavalcante. “A Mission-Oriented Tool for System-of-Systems Modeling”. In: *2015 IEEE/ACM 3rd International Workshop on Software Engineering for Systems-of-Systems*. IEEE, 2015, pp. 31–36. DOI: 10.1109/SESoS.2015.13.
- [539] Abhishek Singh, Pontus Ekberg, and Sanjoy Baruah. “Applying Real-Time Scheduling Theory to the Synchronous Data Flow Model of Computation”. In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Ed. by Marko Bertogna. Leibniz International Proceedings in Informatics 76. Dagstuhl Publishing, 2017, 8:1–8:22. DOI: 10.4230/LIPIcs.ECRTS.2017.8.
- [540] Frank Singhoff et al. “Cheddar: A Flexible Real Time Scheduling Framework”. In: *Proceedings of the 2004 Annual ACM International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies*. ACM, 2004, pp. 1–8. DOI: 10.1145/1032297.1032298.
- [541] Fabien Siron et al. “The Synchronous Logical Execution Time Paradigm”. In: *European Congress on Real-Time Systems*. 2022. URL: https://inria.hal.science/hal-03694950/file/ERTS_2022_paper_34.pdf.
- [542] Irfan Sljivo et al. “Fostering Reuse within Safety-Critical Component-Based Systems through Fine-Grained Contracts”. In: *Proceedings of the International Workshop on Critical Software Component Reusability and Certification across Domains*. 2013.
- [543] Irfan Sljivo et al. “Strong and Weak Contract Formalism for Third-Party Component Reuse”. In: *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2013, pp. 359–364. DOI: 10.1109/ISSREW.2013.6688921.

Bibliography

- [544] Christoph Sommer et al. “A Computationally Inexpensive Empirical Model of IEEE 802.11p Radio Shadowing in Urban Environments”. In: *2011 8th International Conference on Wireless On-Demand Network Systems and Services*. IEEE, 2011, pp. 84–90. DOI: 10.1109/WONS.2011.5720204.
- [545] Christoph Sommer et al. “Veins: The Open Source Vehicular Network Simulation Framework”. In: *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem*. Ed. by Antonio Virdis and Michael Kirsche. EAI/Springer Innovations in Communication and Computing. Springer, Cham, 2019, pp. 215–252. DOI: 10.1007/978-3-030-12842-5_6.
- [546] Li Song, Wei Fan, and Pengfei Liu. “Exploring the Effects of Connected and Automated Vehicles at Fixed and Actuated Signalized Intersections with Different Market Penetration Rates”. In: *Transportation Planning and Technology* 44.6 (2021), pp. 577–593. DOI: 10.1080/03081060.2021.1943129.
- [547] Andres Sousa-Poza, Samuel Kovacic, and Charles Keating. “System of Systems Engineering: An Emerging Multidiscipline”. In: *International Journal of System of Systems Engineering* 1.1-2 (2008), pp. 1–17. DOI: 10.1504/IJSSE.2008.018129.
- [548] Sparx Systems Pty Ltd. *Enterprise Architect Software Developers’ Kit*. 2010. URL: https://sparxsystems.com/downloads/resources/booklets/enterprise_architect_sdk.pdf.
- [549] Sparx Systems Pty Ltd. *Enterprise Architect*. <https://sparxsystems.com/products/ea/>. Last accessed on June 19, 2026.
- [550] Sparx Systems Pty Ltd. *UML 2 Tutorial – Activity Diagram*. <https://sparxsystems.com/resources/tutorials/uml2/activity-diagram.html>. Last accessed on June 19, 2026.
- [551] Tim Stahl et al. “Online Verification Concept for Autonomous Vehicles — Illustrative Study for a Trajectory Planning Module”. In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2020. DOI: 10.1109/ITSC45102.2020.9294703.
- [552] Orion Staskal et al. “Translating SysML Activity Diagrams for nuXmv Verification of an Autonomous Pancreas”. In: *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2022, pp. 1637–1642. DOI: 10.1109/COMPSAC54236.2022.00260.
- [553] Jan-Philipp Steghöfer et al. “Design Decisions in the Construction of Traceability Information Models for Safe Automotive Systems”. In: *2021 IEEE 29th International Requirements Engineering Conference (RE)*. IEEE, 2021, pp. 185–196. DOI: 10.1109/RE51729.2021.00024.
- [554] Jan-Philipp Steghöfer et al. “The MobSTr Dataset – An Exemplar for Traceability and Model-Based Safety Assessment”. In: *2021 IEEE 29th International Requirements Engineering Conference (RE)*. IEEE, 2021, pp. 444–445. DOI: 10.1109/RE51729.2021.00062.

- [555] Jan-Philipp Steghöfer et al. *The MobSTr Dataset: Model-Based Safety Assurance and Traceability*. <https://github.com/panorama-research/mobstr-dataset>. Last accessed on June 19, 2026. DOI: 10.5281/zenodo.4981480.
- [556] Jochen Stellwagen, Matthias Deegener, and Michael Kuhn. “Measurement of Dissemination Characteristics of LTE-V2X and ITS-G5 for Safety-Critical Vehicle-to-Vehicle Communication”. In: *2022 5th International Conference on Advanced Communication Technologies and Networking (CommNet)*. IEEE, 2022. DOI: 10.1109/CommNet56067.2022.9993903.
- [557] Step-Up!CPS partners. *Step-Up!CPS – Sicherheitskritische Systeme sind keine Smartphones!* <https://www.stepup-cps.de/>. Last accessed on June 19, 2026.
- [558] Ingo Stierand, Björn Koopmann, and Bert Bödecker. “Relating Global and Local Clocks in Timing Specifications”. Unpublished article, planned submission to the *Leibniz Transactions on Embedded Systems* journal (<https://ojs.dagstuhl.de/index.php/lites>).
- [559] Ingo Stierand et al. “MTSL 1.5 – Syntax and Semantics”. Language Specification. German Aerospace Center, 2023.
- [560] Scott D. Stoller. “Detecting Global Predicates in Distributed Systems with Clocks”. In: *Distributed Computing* 13.2 (2000), pp. 85–98. DOI: 10.1007/s004460050069.
- [561] Torben Stolte et al. “Taxonomy to Unify Fault Tolerance Regimes for Automotive Systems: Defining Fail-Operational, Fail-Degraded, and Fail-Safe”. In: *IEEE Transactions on Intelligent Vehicles* 7.2 (2022), pp. 251–262. DOI: 10.1109/TIV.2021.3129933.
- [562] Minghui Sun et al. “Defining and Characterizing Model-Based Safety Assessment: A Review”. In: *Safety Science* 172 (2024), p. 106425. DOI: 10.1016/j.ssci.2024.106425.
- [563] Quan Sun et al. “A Contract-Based Semantics and Refinement for Simulink”. In: *Dependable Software Engineering. Theories, Tools, and Applications*. Ed. by Wei Dong and Jean-Pierre Talpin. Lecture Notes in Computer Science 13649. Springer, Cham, 2022, pp. 134–148. DOI: 10.1007/978-3-031-21213-0_9.
- [564] Wei Sun et al. “Analytical Study of the IEEE 802.11p EDCA Mechanism”. In: *2013 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2013, pp. 1428–1433. DOI: 10.1109/IVS.2013.6629667.
- [565] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. “Clock Synchronization for Wireless Sensor Networks: A Survey”. In: *Ad Hoc Networks* 3.3 (2005), pp. 281–323. DOI: 10.1016/j.adhoc.2005.01.002.
- [566] Systemite AB. *SystemWeaver*. <https://systemweaver.com/>. Last accessed on June 19, 2026.
- [567] Yue Tang et al. “Comparing Communication Paradigms in Cause-Effect Chains”. In: *IEEE Transactions on Computers* 72.1 (2023), pp. 82–96. DOI: 10.1109/TC.2022.3197082.

Bibliography

- [568] Sasu Tarkoma. *Publish/Subscribe Systems: Design and Principles*. Ed. by David Hutchison, Serge Fdida, and Joe Sventek. Wiley Series in Communications Networking & Distributed Systems. John Wiley & Sons, 2012. ISBN: 978-1-119-95154-4.
- [569] The MathWorks, Inc. *MATLAB*. <https://www.mathworks.com/products/matlab.html>. Last accessed on June 19, 2026.
- [570] The MathWorks, Inc. *Simulink*. <https://www.mathworks.com/products/simulink.html>. Last accessed on June 19, 2026.
- [571] The MathWorks, Inc. *Stateflow*. <https://www.mathworks.com/products/stateflow.html>. Last accessed on June 19, 2026.
- [572] Jean-Pierre Thomesse, Zoubir Mammeri, and L. Vega. “Time in Distributed Systems Cooperation and Communication Models”. In: *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. 1995, pp. 41–49. DOI: 10.1109/FTDCS.1995.524967.
- [573] Johan Thunberg et al. “Unreliable V2X Communication in Cooperative Driving: Safety Times for Emergency Braking”. In: *IEEE Access* 9 (2021), pp. 148024–148036. DOI: 10.1109/ACCESS.2021.3124450.
- [574] Ken Tindell and Alan Burns. “Guaranteeing Message Latencies on Control Area Network (CAN)”. In: *Proceedings of the 1st International CAN Conference*. 1994, pp. 1–11.
- [575] Ken Tindell, Alan Burns, and Andy J. Wellings. “Calculating Controller Area Network (CAN) Message Response Times”. In: *Control Engineering Practice* 3.8 (1995), pp. 1163–1169. DOI: 10.1016/0967-0661(95)00112-8.
- [576] Ken Tindell, H. Hanssmon, and Andy J. Wellings. “Analysing Real-Time Communications: Controller Area Network (CAN)”. In: *Proceedings of the 15th IEEE International Real-Time Systems Symposium*. IEEE, 1994, pp. 259–263. DOI: 10.1109/REAL.1994.342710.
- [577] Behrad Toghi et al. “Spatio-Temporal Dynamics of Cellular V2X Communication in Dense Vehicular Networks”. In: *2019 IEEE 2nd Connected and Automated Vehicles Symposium (CAVS)*. IEEE, 2019, pp. 1–5. DOI: 10.1109/CAVS.2019.8887759.
- [578] Louis Calvin Touko Tcheumadjeu et al. “New Concepts to Improve Mobility by Digitization and Virtualization: An Analysis and Evaluation of the Technical Feasibility”. In: *Intelligent Transport Systems*. Ed. by Ana Lúcia Martins, Joao C. Ferreira, and Alexander Kocian. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 426. Springer, Cham, 2022, pp. 26–43. DOI: 10.1007/978-3-030-97603-3_3.
- [579] Maria Trei et al. “An ISO 26262 Compliant Design Flow and Tool for Automotive Multicore Systems”. In: *Product-Focused Software Process Improvement*. Ed. by Pekka Abrahamsson et al. Lecture Notes in Computer Science 10027. Springer, Cham, 2016, pp. 163–180. DOI: 10.1007/978-3-319-49094-6_11.

- [580] U.S. Department of War. *The DoDAF Architecture Framework Version 2.02*. <https://dowcio.war.gov/DODAF/>. Last accessed on June 19, 2026.
- [581] University of Oxford. *PRISM – Probabilistic Symbolic Model Checker*. <http://www.prismmodelchecker.org/>. Last accessed on June 19, 2026.
- [582] Paul Unterhuber, Andreas Lehner, and Fabian de Ponte Müller. “Measurement and Analysis of ITS-G5 in Railway Environments”. In: *Communication Technologies for Vehicles*. Ed. by Jaizki Mendizabal et al. Lecture Notes in Computer Science 9669. Springer, Cham, 2016, pp. 62–73. DOI: 10.1007/978-3-319-38921-9_7.
- [583] Raúl E. Valdés-Pérez. *Spatio-Temporal Reasoning and Linear Inequalities*. A.I. Memo 875. Massachusetts Institute of Technology, 1986. URL: <https://dspace.mit.edu/handle/1721.1/6440>.
- [584] Peter Van Roy et al. “Mobile Objects in Distributed Oz”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.5 (1997), pp. 804–851. DOI: 10.1145/265943.265972.
- [585] Maarten Van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 2017. ISBN: 978-90-815406-2-9.
- [586] Andras Varga. “OMNeT++”. In: *Modeling and Tools for Network Simulation*. Ed. by Klaus Wehrle, Mesut Güneş, and James Gross. Springer, Berlin, Heidelberg, 2010, pp. 35–59. DOI: 10.1007/978-3-642-12331-3_3.
- [587] Vector Informatik GmbH. *DaVinci Developer Adaptive. Optimal Assistance for MICROSAR Adaptive Projects*. <https://www.vector.com/int/en/products/products-a-z/software/davinci-developer-adaptive/>. Last accessed on June 19, 2026.
- [588] Vector Informatik GmbH. *DaVinci Developer Classic. Efficient Design of ECU Software Components*. <https://www.vector.com/int/en/products/products-a-z/software/davinci-developer-classic/>. Last accessed on June 19, 2026.
- [589] Vector Informatik GmbH. *TA Tool Suite*. <https://www.vector.com/int/en/products/products-a-z/software/ta-tool-suite/>. Last accessed on June 19, 2026.
- [590] Paulo Veríssimo. “Causal Delivery Protocols in Real-Time Systems: A Generic Model”. In: *Real-Time Systems* 10.1 (1996), pp. 45–73. DOI: 10.1007/BF00357883.
- [591] ViVre partners. *ViVre – Virtuelle Haltestellen für den automatisierten Verkehr der Zukunft*. <https://www.dlr.de/de/ts/forschung-und-transfer/projekte/vivre>. Last accessed on June 19, 2026.
- [592] Marc Vilain, Henry Kautz, and Peter van Beek. “Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report”. In: *Readings in Qualitative Reasoning About Physical Systems*. Ed. by Daniel S. Weld and Johan de Kleer. Elsevier, 1990, pp. 373–381. DOI: 10.1016/B978-1-4832-1447-4.50034-1.

Bibliography

- [593] Thomas Vogel et al. “A Property Specification Pattern Catalog for Real-Time System Verification with UPPAAL”. In: *Information and Software Technology* 154 (2022), p. 107100. DOI: 10.1016/j.infsof.2022.107100.
- [594] Volkswagen AG. *A Milestone for Greater Road Safety: More than Two Million Vehicles Produced with Car2X*. <https://www.volkswagen-newsroom.com/en/press-releases/a-milestone-for-greater-road-safety-more-than-two-million-vehicles-produced-with-car2x-19909>. Last accessed on June 19, 2026.
- [595] Chaeriah Bin Ali Wael et al. “Analysis of IEEE 802.11p MAC Protocol for Safety Message Broadcast in V2V Communication”. In: *2020 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*. IEEE, 2020, pp. 320–324. DOI: 10.1109/ICRAMET51080.2020.9298654.
- [596] Tamás Wágner et al. “SPaT/MAP V2X Communication Between Traffic Light and Vehicles and a Realization with Digital Twin”. In: *Computers and Electrical Engineering* 106 (2023), p. 108560. DOI: 10.1016/j.compeleceng.2022.108560.
- [597] Safat B. Wali et al. “Vision-Based Traffic Sign Detection and Recognition Systems: Current Trends and Challenges”. In: *Sensors* 19.9 (2019), p. 2093. DOI: 10.3390/s19092093.
- [598] Le Wang, Renato F. Iida, and Alexander M. Wyglinski. “Performance Analysis of EDCA for IEEE 802.11p/DSRC Based V2V Communication in Discrete Event System”. In: *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)*. IEEE, 2017, pp. 1–5. DOI: 10.1109/VTCFall1.2017.8288272.
- [599] Runmin Wang et al. “Research on Performance and Function Testing of V2X in a Closed Test Field”. In: *Journal of Advanced Transportation* 2021.1 (2021). DOI: 10.1155/2021/9970978.
- [600] Yunpeng Wang et al. “Throughput and Delay Limits of 802.11p and its Influence on Highway Capacity”. In: *Procedia – Social and Behavioral Sciences* 96 (2013), pp. 2096–2104. DOI: 10.1016/j.sbspro.2013.08.236.
- [601] Warren Weaver. “Recent Contributions to the Mathematical Theory of Communication”. In: *ETC: A Review of General Semantics* 10.4 (1953), pp. 261–281.
- [602] Jan Christoph Wehrstedt et al. “Modeling and Analyzing Context-Sensitive Changes During Runtime”. In: *Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology*. Ed. by Wolfgang Böhm et al. Springer, Cham, 2021, pp. 125–146. DOI: 10.1007/978-3-030-62136-0_6.
- [603] Jennifer Lundelius Welch and Nancy Lynch. “A New Fault-Tolerant Algorithm for Clock Synchronization”. In: *Information and Computation* 77.1 (1988), pp. 1–36. DOI: 10.1016/0890-5401(88)90043-0.

- [604] Gary D. Wells and Andrew P. Sage. “Engineering of a System of Systems”. In: *System of Systems Engineering: Innovations for the 21st Century*. Ed. by Mo Jamshidi. Wiley Series in Systems Engineering and Management. John Wiley & Sons, 2008, pp. 44–76. ISBN: 978-0-470-19590-1.
- [605] Hendrik Wernecke, Bulcsú Sándor, and Claudius Gros. “Chaos in Time Delay Systems, an Educational Review”. In: *Physics Reports* 824 (2019), pp. 1–40. DOI: 10.1016/j.physrep.2019.08.001.
- [606] Geeth P. Wijesiri N.B.A., Jussi Haapola, and Tharaka Samarasinghe. “A Discrete-Time Markov Chain Based Comparison of the MAC Layer Performance of C-V2X Mode 4 and IEEE 802.11p”. In: *IEEE Transactions on Communications* 69.4 (2021), pp. 2505–2517. DOI: 10.1109/TCOMM.2020.3044340.
- [607] Carsten Wolff et al. “AMALTHEA – Tailoring Tools to Projects in Automotive Software Development”. In: *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Vol. 2. IEEE, 2015, pp. 515–520. DOI: 10.1109/IDAACS.2015.7341359.
- [608] Peter Y. H. Wong and Jeremy Gibbons. “A Relative Timed Semantics for BPMN”. In: *Electronic Notes in Theoretical Computer Science* 229.2 (2009), pp. 59–75. DOI: 10.1016/j.entcs.2009.06.029.
- [609] Jim Woodcock et al. “Features of CML: A Formal Modelling Language for Systems of Systems”. In: *2012 7th International Conference on System of Systems Engineering (SoSE)*. IEEE, 2012, pp. 1–6. DOI: 10.1109/SYSSE.2012.6384144.
- [610] Ke Xu et al. “C-V2X Large-Scale Test Network Transmission Performance Data Analysis Method”. In: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2021, pp. 1334–1339. DOI: 10.1109/TrustCom53373.2021.00185.
- [611] Yuan Yao et al. “Delay Analysis and Study of IEEE 802.11p Based DSRC Safety Communication in a Highway Environment”. In: *Proceedings of the 32nd IEEE International Conference on Computer Communications*. IEEE, 2013, pp. 1591–1599. DOI: 10.1109/INFCOM.2013.6566955.
- [612] Yuan Yao et al. “On MAC Access Delay Distribution for IEEE 802.11p Broadcast in Vehicular Networks”. In: *IEEE Access* 7 (2019), pp. 149052–149067. DOI: 10.1109/ACCESS.2019.2946989.
- [613] Yufei Yin et al. “An Automated Test Case Generation Approach Based on Activity Diagrams of SysML”. In: *International Journal of Performability Engineering* 13.6 (2017), p. 922. DOI: 10.23940/ijpe.17.06.p13.922936.
- [614] Haibao Yu et al. “DAIR-V2X: A Large-Scale Dataset for Vehicle-Infrastructure Cooperative 3D Object Detection”. In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2022, pp. 21329–21338. DOI: 10.1109/CVPR52688.2022.02067.

Bibliography

- [615] Sheng-Jung Yu, Inigo Incer, and Alberto Sangiovanni-Vincentelli. “Contract Replaceability for Ensuring Independent Design Using Assume-Guarantee Contracts”. In: *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*. ACM, 2023, pp. 111–121. DOI: 10.1145/3610579.3611088.
- [616] Kenneth Y. Yun and Ryan P. Donohue. “Pausible Clocking: A First Step Toward Heterogeneous Systems”. In: *Proceedings of the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 1996, pp. 118–123. DOI: 10.1109/ICCD.1996.563543.
- [617] Hazim Zainudin et al. “Impact Analysis of Cooperative Perception on the Performance of Automated Driving in Unsignalized Roundabouts”. In: *Frontiers in Robotics and AI* 10 (2023). DOI: 10.3389/frobt.2023.1164950.
- [618] Fenghui Zhang, Michael Mao Wang, and Ruilong Deng. “On Reliability Bound and Improvement of Sensing-Based Semipersistent Scheduling in LTE-V2X”. In: *IEEE Internet of Things Journal* 8.7 (2021), pp. 6101–6113. DOI: 10.1109/JIOT.2020.3034132.
- [619] Pengcheng Zhang, Bixin Li, and Lars Grunske. “Timed Property Sequence Chart”. In: *Journal of Systems and Software* 83.3 (2010), pp. 371–390. DOI: 10.1016/j.jss.2009.09.013.
- [620] Pengcheng Zhang, Bixin Li, and Mingjie Sun. “A Timed Extension of Property Sequence Chart”. In: *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, 2008, pp. 197–206. DOI: 10.1109/HASE.2008.20.
- [621] Wei Zhang et al. “Proving Simulink Block Diagrams Correct via Refinement”. In: *Wireless Communications and Mobile Computing 2022* (2022). DOI: 10.1155/2022/8015896.
- [622] Yuanrui Zhang et al. “Specification and Verification of Multi-Clock Systems Using a Temporal Logic with Clock Constraints”. In: *Formal Aspects of Computing* 36.2 (2024). DOI: 10.1145/3670794.
- [623] Jun Zheng and Qiong Wu. “Performance Modeling and Analysis of the IEEE 802.11p EDCA Mechanism for VANET”. In: *IEEE Transactions on Vehicular Technology* 65.4 (2016), pp. 2673–2687. DOI: 10.1109/TVT.2015.2425960.
- [624] Bo Zhou et al. “Modeling System of Systems: A Generic Method Based on System Characteristics and Interface”. In: *2011 9th IEEE International Conference on Industrial Informatics*. IEEE, 2011, pp. 361–368. DOI: 10.1109/INDIN.2011.6034903.
- [625] Donghao Zhou et al. “Autonomous Vehicles’ Intended Cooperative Motion Planning for Unprotected Turning at Intersections”. In: *IET Intelligent Transport Systems* 16.8 (2022), pp. 1058–1073. DOI: 10.1049/itr2.12195.

- [626] Walter Zimmer et al. “TUMTraf V2X Cooperative Perception Dataset”. In: *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2024, pp. 22668–22677. DOI: 10.1109/CVPR52733.2024.02139.
- [627] dSPACE GmbH. *Real-Time Testing. Python-Based Real-Time Test Automation*. https://www.dspace.com/en/pub/home/products/sw/test_automation_software/automated_test_execution_in_re.cfm. Last accessed on June 19, 2026.
- [628] dSPACE GmbH. *SCALEXIO System. Modular and Scalable Real-Time Platform for Hardware-in-the-Loop (HIL) and Rapid Control Prototyping (RCP) Applications*. https://www.dspace.com/en/pub/home/products/hw/simulator_hardware/scalexio.cfm. Last accessed on June 19, 2026.
- [629] dSPACE GmbH. *SystemDesk. Your Starting Point for Simulation & Validation*. https://www.dspace.com/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm. Last accessed on June 19, 2026.
- [630] dSPACE GmbH. *TargetLink. Production Code Generation for the Highest Demands*. <https://www.dspace.com/en/pub/home/products/sw/pcgs/targetlink.cfm>. Last accessed on June 19, 2026.
- [631] u-blox AG. *A Review of 5GAA’s “V2X Functional and Performance Test Report” P-180106*. 2018. URL: <https://content.u-blox.com/sites/default/files/Comments-on-5GAA-PetitionWaver-18-357.pdf>.

