PADERBORN UNIVERSITY
The University for the Information Society

Faculty for Computer Science, Electrical Engineering and Mathematics

IMPROVEMENT OF SOFTWARE REQUIREMENTS QUALITY BASED ON SYSTEMS ENGINEERING

PhD Thesis
to obtain the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

by
JÖRG HOLTMANN

Referees:
Prof. Dr.-Ing. Roman Dumitrescu
Prof. Dr. rer. nat. Joel Greenyer

Paderborn, June 2019

Abstract

Software-intensive systems increasingly pervade our society and economy, and their application in safety-critical contexts can even decide about life or death (e.g., driver assistance systems). Such systems are typically developed in a multidisciplinary manner, are often subject to real-time requirements, and are executed on distributed and concurrent platforms influencing their timing behavior.

A high quality of the requirements on these systems' software is crucial, because the software requirements are the basis for the software design and development. The application of models in Requirements Engineering (RE) is considered beneficial, because they foster automatic analysis techniques that aim at ensuring high-quality requirements. However, existing model-based RE approaches take neither the transition from multidisciplinary to discipline-specific RE phases nor platform-induced timing effects during real-time requirements validation sufficiently into account. This results in potential software requirements defects introduced during the transition as well as costly development iterations due to timing analyses conducted in late engineering phases.

This thesis proposes and evaluates a model-based RE approach that addresses these problems by means of two techniques. First, it presents a semi-automatic technique for the transition from multidisciplinary system models to software RE models. Second, it presents a technique for the semi-automatic verification of timing-relevant platform properties against real-time requirements as part of the software RE models. These contributions improve the quality of software requirements by reducing the likelihood to introduce requirements defects during the transition from multidisciplinary system to software RE models and by early revealing platform-induced real-time requirement violations.

The thesis evaluates the approach by conducting case studies based on an automotive Vehicle-to-X driver assistance system example. These case studies indicate the effectiveness and efficiency of both techniques.

Zusammenfassung

Software-intensive Systeme durchdringen zunehmend unsere Gesellschaft und Industrie, und ihre Anwendung in sicherheitskritischen Bereichen kann sogar über Leben und Tod entscheiden (z. B. im Fall von Fahrerassistenzsystemen). Solche Systeme werden typischerweise multidisziplinär entwickelt, unterliegen oft Echtzeitanforderungen und werden auf verteilten und nebenläufigen Plattformen ausgeführt, die ihr Zeitverhalten beeinflussen.

Eine hohe Qualität der Anforderungen an die Software dieser Systeme ist unabdingbar, da die Softwareanforderungen die Basis für den Entwurf und die Entwicklung der Software sind. Die Anwendung von Modellen im Requirements Engineering (RE) wird als vorteilhaft angesehen, da Modelle automatische Analysetechniken zur Sicherstellung von hochqualitativen Anforderungen fördern. Jedoch berücksichtigen existierende modellbasierte RE-Ansätze weder den Übergang von multidisziplinären zu disziplinspezifischen RE-Phasen noch plattforminduzierte Zeiteffekte während der Echtzeitanforderungsvalidierung in ausreichender Weise. Dies resultiert zum einen in potentiellen Softwareanforderungsdefekten, die sich während des Übergangs einschleichen. Zum anderen entstehen kostspielige Entwicklungsiterationen durch Zeitanalysen, die erst in späten Entwicklungsphasen durchgeführt werden.

Diese Dissertation präsentiert und evaluiert einen modellbasierten RE-Ansatz, der diese Probleme durch zwei Techniken adressiert. Zum einen führt sie eine semiautomatische Technik für den Übergang von multidisziplinären Systemmodellen zu Software-RE-Modellen ein, sodass sich die Wahrscheinlichkeit für die Entstehung von Anforderungsdefekten bei dem Übergang verringert. Zum anderen stellt sie eine Technik zur semiautomatischen Verifikation von Plattformeigenschaften gegenüber Echtzeitanforderungen in den Software-RE-Modellen vor, sodass eine plattforminduzierte Verletzung von Echtzeitanforderungen bereits in frühen Phasen aufgedeckt werden kann. Insgesamt verbessern diese Techniken somit die Qualität der Softwareanforderungen.

Diese Dissertation evaluiert den Ansatz mittels Fallstudien, die unter anderem ein Vehicle-to-X Fahrerassistenzsystem als Beispiel benutzen. Die Durchführung der Fallstudien zeigt die Effektivität und Effizienz beider Techniken.

Acknowledgements

"You're only as good as your team"—my working motto is also (or particularly) valid for the outcomes of this thesis.

First of all, I want to thank my original doctoral advisor Prof. Dr. Wilhelm Schäfer. Wilhelm incorporated me in the Software Engineering Group and the s-lab – Software Quality Lab but unfortunately could not supervise me until finishing my thesis due to health reasons. Likewise, I thank Prof. Dr.-Ing. Roman Dumitrescu for taking over the supervision on the finishing line. I further thank Roman and Prof. Dr. Joel Greenyer for writing their reports and additionally Joel for his valuable feedback regarding earlier versions. I thank Roman, Joel, Prof. Dr. Eric Bodden, Dr. Matthias Meyer, and Dr. Stefan Sauer for attending my PhD defense.

During the period of conceiving the approaches presented in this thesis, I had the pleasure to work together with a bunch of internal as well as external colleagues and students. Shouts out to (multiple mentions only in reasonable exceptions):

My Office Mates That is, I thank my longest and still current "spouse" David Schmelter, Dr. Dietrich Travkin, Dr. Markus Fockel, Dr. Jens Frieben ("Mount Doom"!), Dr. Christian Heinzemann, Renate Löffler, as well as the original "s-lab automotive office E1.111 crew" (piggy bank for stupid jokes and later for finished PhD theses ftw!) Dr. Jan Meyer, Dr. Matthias Schnelte, and Christian Nawratil for bearing me for better or for worse.

The RE Expert Group That is, I thank Markus, David, and Thorsten Koch for valuable discussions and the joint elaboration of concepts, publications, and examples in the context of RE as well as joint work in diverse industrial and research projects.

The Software Engineering Group / Department Particularly, I thank Dr. Uwe Pohlmann (afterwork beer!), Dr. Stefan Dziwok, Dr. Marie Christin Platenius-Mohr, David Schubert, Christopher Gerking, Johannes Geismann, Dr. Matthias Meyer, Dr. Johannes Späth, Sven Merschjohann, Andreas Dann, Lars Stockmann, Ingo Budde, Dr. Matthias Becker, Oliver Sudmann, Dr. Jan Rieke, Dr. Christian Heinzemann, Dr. Markus von Detten, and Prof. Dr. Matthias Tichy for the joint work, feedback, discussions, coffee breaks, and/or (early) evening drinks.

My colleagues in the it's OWL – SE project Particularly, I thank Dr.-Ing. Lydia Kaiser, Martin Rabe, Dr. Anja Schierbaum, Dr.-Ing. Arno Kühn, and Dr. Stefan Herbrechtsmeier for the interdisciplinary joint work and discussions in the context of Systems Engineering.

My colleagues at the Fraunhofer IEM Particularly, I thank Dr. Christian Tschirner, Christian Bremer, Alexander Albers, Lukas Bretz, Matthias Greinert, Dr.-Ing. Peter Ebbesmeyer, Christopher Lankeit, and Fabian Ernst for the interdisciplinary joint work and discussions in diverse industrial and research projects.

My external colleagues Particularly, I thank Assoc. Prof. Julien DeAntoni for the inter-university and unfunded joint work and for supporting me as well as my students with ideas, advices, and bugfixes regarding Chapter 4. Furthermore, I thank Dr. Ernst Sikora, Prof. Dr. Bastian Tenbergen, Marian Daun, Prof. Dr. Alexander Metzner, Dr. Eike Thaden, and Philipp Reinkemeier for the inter-university joint work and discussions on embedded systems software engineering in the German SPES2020 project.

My former student assistants and mentees Particularly, I thank Ruslan Bernijazov whose bachelor's / master's theses as well as student assistant work strongly contributed to this thesis. Furthermore, I particularly thank Sergej Japs, Marcel Sander, Kai Biermeier, and Sebastian Otutuama for their work as student assistants. Finally, I particularly thank Sergej Japs, Thorsten Koch, Dimitar Shipchanov, Simon Schwichtenberg, and Christopher Brune for their results in the context of master's / bachelor's theses and the project group.

My administrative colleagues That is, I thank Jutta Haupt, Jürgen "Sammy" Maniera, Sabine Illigen, and Vera Meyer for caring about administrative or technical issues directly or indirectly connected to my thesis or work.

Of course, I also had a social life, which influenced my thesis and vice versa. In this context, I thank my friends who accompanied me in all living situations. A large part of you was always kidding me with finishing my thesis—here it is, and now read it!☺ Furthermore, I thank my former girlfriends who accompanied me partly for long times and had to "suffer" from my thesis. Moreover, I thank my parents and the small remainder of my family for giving me all the required space and support. Finally, a special shout out to Keith Flint, who sadly passed away shortly before my oral exam and whose band I followed the last 25+ years. One line of your lyrics even made it onto my mortarboard—raise the roof and fly high!

Contents

1

Introduction

Information and communication have become a key innovation force for technical systems [GDS+15]. The software part of these technical systems drives their information and communication capabilities and hence is rising both in size [McK18; VBK10] and market value [McK16; BBH+10]. Such *software-intensive systems* [ISO17] are comprised of software as well as hardware, inter alia, and include embedded systems [HS07], mechatronic systems [Aus96; VDI04], and cyber-physical systems [aca11; Poo10; SW07]. There is a shift toward cyber-physical systems accompanied by an increasing functionality and complexity [AT16]. Furthermore, software-intensive systems often operate in safety-critical application areas. An example of a safety-critical software-intensive system is an automotive Vehicle-to-X driver assistance system, the so-called *Emergency Braking & Evasion Assistance System (EBEAS)* [*HFK+16, Chapter 4]. The EBEAS senses the vehicle's environment, coordinates its actions with other vehicles, and actively performs emergency braking or evasion maneuvers in case a potential hazard occurs.

In this thesis, we propose and evaluate an approach for improving the quality of the requirements on the message-based interactions within and between software-intensive systems. Software-intensive systems have a multitude of characteristics, of which we particularly consider the following ones in this thesis:

Multidisciplinary Development One characteristic of software-intensive systems is their multidisciplinary development [GDS+15]. That is, such systems are jointly developed by several engineering disciplines like software engineering, control engineering, electrical engineering, and mechanical engineering.

Real-time Criticality Software-intensive systems often have to operate under hard real-time constraints. That is, the correctness of the behavior of the system under development (SUD) is not only dependent on correct value computations but also on the time in which the values are produced and delivered [But11]. For example, performing emergency braking only milliseconds too late can harm the life of the passengers in the case of the EBEAS.

Distributed and Concurrent Computation The growing functionality of software-intensive systems has led to thousands of software operations distributed across hundreds of ECUs that communicate via multiple bus systems (e.g., in the case of modern vehicles [VBK10]).

The increasing complexity of software-intensive systems requires a rigorous development process. Such a process encompasses several phases, in which different approaches are applied. In this thesis, we focus on the following phases:

Systems Engineering The multidisciplinary development of software-intensive systems requires a holistic and interdisciplinary consideration of the overall system to obtain a common understanding of the SUD for all roles involved in the development. "*Systems engineering* is an interdisciplinary approach and means to enable the realization of successful systems" [INCOSE; WRF$^+$15] and aims at achieving such a common understanding.

Requirements Engineering The basis for the design and development of all systems including software-intensive systems are requirements on them. "*Requirements Engineering [(RE)]*[1] is a systematic and disciplined approach to the specification and management of requirements with [the goal] [...] to minimize the risk of delivering a system that does not meet the stakeholders' desires and needs." [PR11] RE is one of the most important phases during the development of a software-intensive system [GDS$^+$15], since errors in the requirements specification are hard and hence costly to fix in the subsequent development phases (e.g., [Boe81; Boe83; PR11]). Consequently, the requirements quality strongly influences the success or failure of development projects [KEF09; KT07]. In the development process of a software-intensive system, RE is conducted for the overall system (i.e., RE for system requirements) as well as for the tasks of the particular disciplines (e.g., RE for software requirements).

Software Design and Development After the development goals have become clear for all involved disciplines by means of systems engineering and the discipline-specific RE, the discipline-specific design and development can be approached. We distinguish the engineering of continuous (feedback-)*control behavior* and the engineering of discrete *coordination behavior* [*PHMG14].

Like in [*PHMG14; HSST13; Rie15; GRS14], we regard the discipline of software engineering as the one that designs the coordination behavior. The requirements engineering for software is regarded as a branch of systems engineering [NE00]. More specifically, we regard *software requirements engineering (SwRE)* as a sub-discipline of software engineering, that is, as RE for the coordination behavior of the overall system.

The goal of this thesis is to provide an SwRE approach for improving the quality of requirements on the coordination behavior of software-intensive systems, where the approach encompasses two techniques. First, we describe a technique for semi-automatically deriving such coordination behavior requirements from more abstract systems engineering requirements, thereby systematizing the requirements refinement in a multidisciplinary development process. Second, we describe a technique for the early simulative verification of timing-relevant platform properties against coordination behavior requirements with real-time constraints, thereby validating real-time requirements on the distributed and concurrent coordination behavior.

[1]RE encompasses the core activities elicitation, documentation, validation/negotiation, and management [PR11]. Within this thesis, we focus on the activities requirements documentation and requirements validation, which has the goal to discover errors in the documented requirements. Despite this focus, we use the abbreviated term RE for these two core activities if not otherwise stated.

1.1 Approaches for the Development of Software-intensive Systems Considered in this Thesis

A multitude of approaches are applied in the overall development process of software-intensive systems. Figure 1.1 visualizes the application of the approaches that we consider in this thesis by means of an alignment into a Vee model based on the VDI guideline 2206 [VDI04].

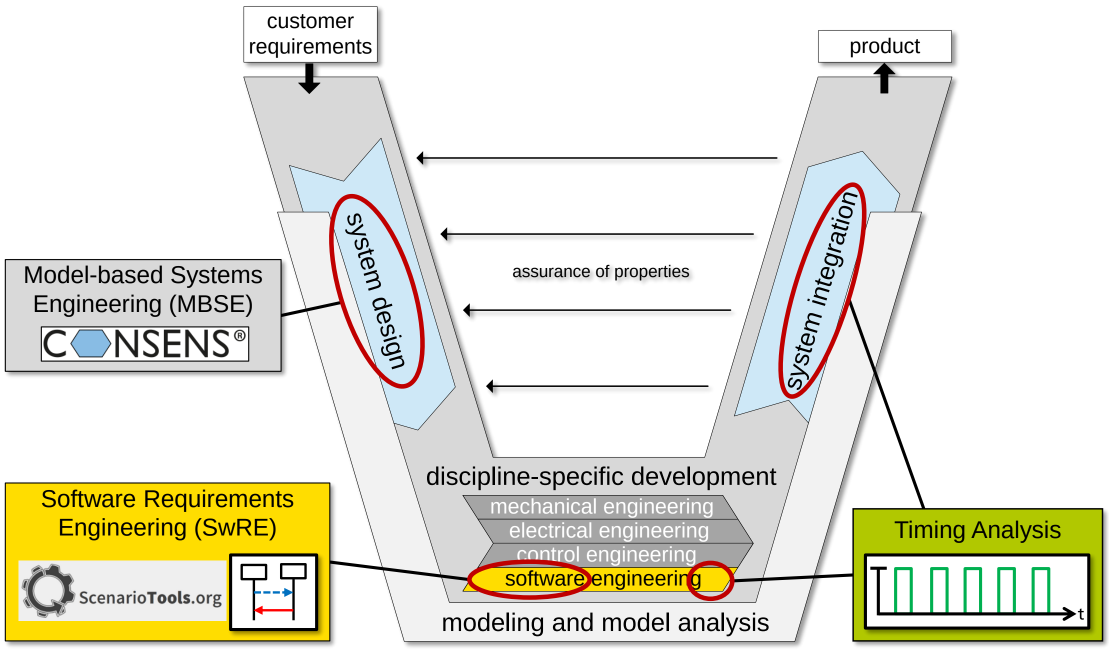


Figure 1.1: Overview of the approaches considered in this thesis (Vee model visualization based on [VDI04])

1.1.1 The Specification Technique CONSENS for Model-based Systems Engineering

According to the INCOSE Systems Engineering Vision 2025 [INC14], the current transition from traditional document-based systems engineering to *Model-Based Systems Engineering (MBSE)* will establish systems engineering as the future development paradigm for software-intensive systems. In order to follow the MBSE paradigm, we apply the model-based specification technique CONSENS (*CONceptual design Specification technique for the ENgineering of complex Systems*) [DDGI14; GFDK09; Fra06] within the interdisciplinary system design phase (cf. upper left part of Figure 1.1). This specification technique is explicitly designed to cover all aspects of software-intensive systems and to facilitate the communication between all disciplines at eye level. CONSENS encompasses a modeling language and a tailored method for the language application. The resulting CONSENS system models serve as input to the discipline-specific requirements analysis conducted in the particular engineering disciplines.

1.1.2 Modal Sequence Diagrams (MSDs) for Scenario-based Software Requirements Specification and Analysis

The use of models in SwRE for software-intensive systems is considered beneficial [STP12]. The main advantages of requirements models as documentation format are that they facilitate the understanding of requirements [NT09] by raising the abstraction level in requirements descriptions [CA09; CA07] and foster automatic analysis techniques. A well-suited notation for the requirements documentation in model-based SwRE is a scenario-based formalism. Scenarios describe sequences of events of tasks that the SUD has to accomplish [HRD10]. Scenario-based notations have an intuitive representation [HRD10] and improve the comprehension of functional requirements for people experienced in modeling [AGI$^+$13].

The modal semantics of the scenario-based formalism *Live Sequence Charts (LSCs)* [DH01] allows requirements engineers to specify which event sequences produced by the SUD may, must, or must not occur, which is crucial for a requirements language [Har01; Har00]. Greenyer [Gre11] developed an SwRE approach based on a recent LSC variant compliant to the Unified Modeling Language (UML) [OMG17b], so-called *Modal Sequence Diagrams (MSDs)* [HM08; HM06]. Greenyer conceived an MSD dialect suited to address some of the characteristics of mechatronic systems. For this purpose, he extended the original MSD formalism by modeling constructs to specify real-time requirements and assumptions on the environment, inter alia. Based on this MSD dialect, Greenyer applied and extended two complementary automatic analysis techniques enabling the early detection of requirements defects in the beginning of the software engineering phase (cf. lower left part of Figure 1.1).

1.1.3 Timing Analysis

The distributed and concurrent computation in software-intensive systems leads to a wide range of platform properties that influence the timing behavior of the SUD. This can lead to violations of the real-time requirements.

Nowadays, the timing-relevant platform properties and their effects on the timing behavior are verified against real-time requirements by means of different simulative and analytical *timing analysis* techniques. One class of these techniques applies response time analysis [SAÅ$^+$04; ABD$^+$95], which relies on computing worst-case execution time bounds for the code compiled to a specific target platform or on execution measurements [WEE$^+$08]. Thus, such techniques are applied in the end of the software engineering development phase (cf. bottom right in Figure 1.1) when the target platform is known and the software code exists [DWUL17; MNS$^+$17; MSN$^+$15]. A second class of these techniques applies Hardware-in-the-Loop simulation to execute the software on an existing target platform and hence is applied in the system integration phase (cf. middle right in Figure 1.1).

1.2 Problem Description

Even with a formal scenario-based SwRE approach that ensures consistent requirements, there remain the following problems that we address within this thesis.

1.2.1 Manual and Unsystematic Transition from MBSE to SwRE

Although MBSE endeavors to coordinate the overall development process for software-intensive systems by means of interdisciplinary system models, the transition from MBSE to SwRE is not trivial. System models contain much information that is only partly relevant to each involved discipline. Thus, the Software Requirements Engineer must carefully identify the SwRE-relevant information in the system models and transfer this information into MSD specifications.

Although CONSENS system models are amenable for automatisms to extract the SwRE-relevant information, this is a manual and thereby error-prone and time-consuming task up to now. Thus, the Software Requirements Engineers can introduce defects (e.g., incompleteness, incorrectness w.r.t. the CONSENS system models) into the MSD specifications. In such cases, the analysis techniques for MSDs can yield wrong results like false positives.

1.2.2 Late Timing Analyses

Timing analyses are nowadays conducted in late engineering phases despite the fact that coarse-grained information about the timing-relevant platform properties is mostly known in the SwRE phase from prior projects [MSN+15; HH04] or from technology decisions determined during the interdisciplinary system design. Applying validation and verification techniques early in the development process is desirable since the detection and fixing of defects in later engineering phases causes costly development iterations (e.g., [Boe81; Boe83; PR11]). Thus, the platform properties and their effects on the timing behavior should be verified as early as possible w.r.t. the real-time requirements in order to identify and resolve potential requirements violations, ideally in the SwRE phase.

Whereas we improved MSDs and their analysis techniques regarding real-time aspects [*BGH+14; *Jap15; *BBG+13], MSDs address the platform-independent requirements on the system's coordination behavior. That is, the timing and event handling abstractions in the platform-independent timed analysis techniques for MSDs are not designed to consider detailed platform-specific effects on the timing behavior emerging from the software execution in distributed systems. Hence, platform-aware timing analysis techniques verifying such timing behaviors against real-time requirements are not in the scope of the platform-independent timed MSD analysis techniques up to now.

1.3 Approach to Solution and Contributions

The goal of this thesis is to improve the quality of the requirements on the coordination behavior of software-intensive systems. In order to address the problems described in Section 1.2, we present in this thesis two techniques sketched in the remainder of this section. The first technique reduces the likelihood to introduce requirements defects during the transition from MBSE with CONSENS to SwRE with MSDs. The second technique reveals platform-induced real-time requirement violations in the early SwRE phase based on MSD specifications. We implemented all concepts as extensions to the tool suite SCENARIOTOOLS MSD [ST-MSD] and evaluated the techniques for effectiveness and efficiency.

1.3.1 Semi-automatic Technique for the Transition from MBSE to SwRE

In order to address the first problem, we present a semi-automatic and systematic technique for the transition from MBSE with CONSENS to SwRE with MSDs [*HBM+16; *HBM+15; *HBM+17; *Ber15]. We identify aspects of the system model that are relevant to SwRE with MSDs and describe how SwRE integrates into MBSE. We automate steps of the transition where possible (i.e., for the automatically processable information in CONSENS system models) to avoid error-prone and time-consuming manual tasks. This particularly includes incremental and traceability-establishing model transformations that derive initial and update existing MSD specifications from CONSENS system models. For the informal information part of CONSENS system models, we provide an semi-automatic approach for the systematic manual refinement of such initially derived or updated MSD specifications.

1.3.2 Early Timing Analyses based on MSDs

In order to enable early timing analyses based on MSDs, we present an approach for the verification of timing-relevant platform properties w.r.t. the functional and particularly the real-time requirements specified by means of timed MSDs [*Ber17]. To this end, we first propose a UML profile to provide modeling means for the timing-relevant platform properties. Second, we conceptually extend the event handling semantics of timed MSDs so that the timing analysis is able to take the delays between different timing-relevant events into account. Based on this, we third apply the GEMOC approach [LCD+15; CAL+13] to specify MSD semantics dedicated to the purpose of timing analyses. This semantics enables the consideration of the platform properties through encoding their effects on the timing behavior as well as the MSD real-time requirements in order to verify them w.r.t. the timed MSDs. Taking this information as input, GEMOC generates timed models that are executable in a simulative timing analysis tool for distributed systems.

1.4 Thesis Structure

Chapter 2 introduces the foundations that are necessary for the understanding of the remainder of this thesis. We present our transition technique from MBSE to SwRE in Chapter 3 and its supplementary material in Appendix A. Subsequently, we present our approach enabling early timing analyses based on MSDs in Chapter 4 and its supplementary material in Appendix B. Finally, we conclude this thesis and summarize its future work in Chapter 5.

2

Foundations

In Section 2.1, we present and extend terminology for model-based traceability. Section 2.2 introduces the specification technique CONSENS for MBSE, and Section 2.3 presents prior work on the automatic derivation of discipline-specifific models from CONSENS system models. In Section 2.4, we introduce foundations on MSDs. Section 2.5 present UML profile excerpts that we use and extend in this thesis. We give an overview on conventional timing analysis techniques for hard real-time systems in Section 2.6. Subsequently, we introduce a modeling language, semantics, and tool support for simulative timing analyses in Section 2.7. Finally, we sketch an approach for specifying semantics for arbitrary modeling languages dedicated to this timing analysis approach in Section 2.8.

2.1 Model-based Traceability

Traceability is the basis for many management activities throughout the development lifecycle. For example, such management activities encompass impact analyses after requirements or other work products have been changed, assessing that requirements are satisfied, deriving and associating test cases from/with the corresponding requirements, or reasoning about the "right to exist" of design artifacts. For such purposes, traceability is demanded by many standards for the development of software-intensive systems (e.g., [ASIG17; ISO18a; RTCA11]), which aim at ensuring a high quality of the development process and thereby of the resulting product.

In the following section, we introduce the foundational traceability terminology used in this thesis and extend it for model-based aspects where required. Subsequently, we introduce a model-based traceability management tool that we apply in this thesis to semi-automatically establish traceability between CONSENS system models and MSD specifications.

2.1.1 Terminology

In this section, we first introduce the foundational terminology used in this thesis. Subsequently, we add more precise definitions for the model-based aspects of traceability that we address in this thesis in order to distinguish between different kinds of traceability in the remainder of this thesis.

2.1.1.1 Foundational Terminology

In this section, we present the existing terminology introduced by the traceability literature used in this thesis. We first introduce generic terms, and we explain the distinction between implicit and explicit traceability subsequently.

Basic Terms

Gotel et al. [GCH$^+$12] present a common, generic terminology for traceability: "*Traceability* is the potential for *traces* [...] to be established (i.e., created and maintained) and used", where a trace encompasses "a *source artifact*, a *target artifact*[,] and a *trace link* associating the two artifacts". Source and target artifacts (together called *trace artifacts*) as well as trace links have a certain *trace artifact type* and *trace link type*, respectively. These types are labels that characterize those artifacts or links "that have the same or similar structure (syntax) and/or purpose (semantics)". A *trace relation* includes "all the trace links created between two sets of specified trace artifact types", where a "trace relation is the instantiation of the *trace relationship* and hence is a collection of traces". A *traceability information model* defines "the permissible trace artifact types, the permissible trace link types and the permissible trace relationships". Trace links are typically bidirectional and can be traversed in the *primary trace link direction* (i.e., from the source to the target artifact) or in the *reverse trace link direction* (i.e., from the target to the source artifact). The *trace granularity* is "the level of detail at which a trace is recorded and performed", where "the granularity of a trace is defined by the granularity of the source artifact and the target artifact." [GCH$^+$12]

Implicit vs. Explicit Traceability

Mäder et al. [MPR07] distinguish between vaguely documented information about traces and distinct, explicitly specified traces:

Implicit Traceability "Implicit traceability results from existing associations between [trace artifacts]. For example, the use of the same identifier in an analysis and a design art[i]fact expresses a dependency between both. The creation of this [kind of] traceability link does not cause any additional effort." [MPR07] Implicit traceability impedes traceability-based management activities because, for example, impact analyses have to be conducted by means of awkward and error-prone searching for the same or similar identifier or term to determine an implicit trace [*HFKS16].

Explicit Traceability "Explicit [t]raceability results from the establishing of connections between two art[i]facts during the [...] development process by a developer. [...] The creation of explicit traceability requires additional effort of the developer." [MPR07] Explicit traceability is required to conduct traceability-based management activities like impact analyses etc. in an adequate and tool-supported way [*HFKS16; MPR07]. Furthermore, explicit traceability is required if traceability is applied to enable incremental model transformations [CH06]. We focus on explicit traceability in this thesis.

2.1.1.2 Extended Terminology for Model-based Traceability

The term traceability and its definition originates from the requirements management community [GF94]. This implies the need for a broader definition of traceability to also cover traceability in the domain of model-based engineering [WP10; ANRS06]. However, the traceability literature does not define a unified terminology for model-based traceability as we apply it in our thesis until now. Thus, we define an extended terminology for model-based traceability used throughout this thesis in the following. This extended terminology enables to precisely distinguish between the different kinds of traceability that we apply in this thesis.

Intra- vs. Inter-model Traceability

The notion of *horizontal* and *vertical traceability* [RE93; GCH+12] distinguishes traceability between trace artifacts belonging to the same project phase or level of abstraction and traceability between trace artifacts belonging to different ones, respectively [WP10]. However, this distinction can be ambiguous depending on the level of detail at which the engineering process is taken into account [WP10]. For example, there is, on the one hand, vertical traceability between partial models specified in earlier and later steps of the CONSENS specification method and thereby within a CONSENS system model. On the other hand, there is vertical traceability between a CONSENS system model and an MSD specification. Hence, we distinguish the following terms to define the terminology more precisely:

Intra-model Traceability The potential to establish and use traces that do not cross the borders of a model. An *intra-model trace link* associates a source model element and a target model element, both residing in the same model. Intra-model traceability can be both horizontal and vertical.

Inter-model traceability The potential to establish and use traces where source and target artifacts are elements of different models. An *inter-model trace link* associates a source model element and a target model element, where one model element resides in one model and the other one in another model. Whereas also inter-model traceability can be both horizontal and vertical, we focus in this thesis on vertical inter-model traceability as we semi-automatically establish traceability between CONSENS system models and MSD specifications belonging to different engineering phases.

Relational vs. Referential Traceability

We distinguish between relational trace links provided as first-class citizens by a modeling language or by a traceability management tool and referential traceability provided by the abstract syntax of a modeling language:

Relational Traceability The potential to establish and use traces by means of relational trace links. A *relational trace link* associates a source model artifact with a target model artifact by means of a dedicated model element provided by the metamodel of a modeling language (with both a concrete and abstract syntax) or of a traceability management tool (e.g., by providing a dedicated trace link type in a traceability metamodel). In terms of the UML [OMG17b], such trace links typically are instances of (metaclass specializations of) the metaclass **DirectedRelationship** that has two association ends pointing to the source and to the target artifact, respectively. In terms of graphs, a relational trace link is represented by a dedicated node with two incident directed edges pointing to the source and to the target artifact node, respectively. In terms of relational databases, relational trace links are captured by dedicated relational tables with two foreign keys that represent primary keys in the source and the target artifact tables, respectively.

Referential Traceability The potential to establish and use traces by means of referential trace links. A *referential trace link* associates a source model artifact with a target model artifact by means of a direct reference that is provided by the abstract syntax of a modeling language. In terms of the UML [OMG17b], such trace links are instances of an association between two metaclasses, referencing directly from the source artifact to the target artifact. In terms of graphs, a referential trace link is a directed edge from the node representing the source artifact to the node representing the target artifact. In terms of relational databases, referential trace links are captured by one foreign key in the source artifact table; this foreign key represents the primary key in the target artifact table.

Lifecycle vs. Transformation Traceability

We distinguish between trace links as the basis for typical model management activities throughout the development lifecycle as exemplified in the beginning of Section 2.1 (e.g., impact analyses) and trace links as the basis for the "Target-Incrementality" feature [CH06] of model transformations (e.g., the correspondence model of Triple Graph Grammars [Sch95]):

Lifecycle Traceability The potential to establish and use traces as the basis for model management activities throughout the development lifecycle. A *lifecycle trace link* associates a source model artifact with a target model artifact for the purpose of enabling lifecycle-oriented management activities. To serve this purpose, the trace link associates model elements that are relevant to the conductor of the management activities. Such model elements are typically represented by the concrete syntax of the corresponding modeling languages and have a rather high granularity For example, the management activity conductor wants to investigate which software components are related to a system element. Thus, the corresponding trace granularity is analogously high. Furthermore, the trace link should be valid w.r.t. to predefined rules regarding the corresponding artifact and trace link types.

Transformation Traceability The potential to establish and use traces as the basis for the "Target-Incrementality" feature of model transformations. A *transformation trace link* associates a source model artifact with a target model artifact for the purpose of enabling the "Target-Incrementality" feature of a model transformation approach. To serve this purpose, these trace links have to associate all model elements that are covered by the model transformation. Such model elements typically include many ones that have no concrete syntax representation and that have a rather low granularity (e.g., two connector ends, each having several properties, are needed to link two structural elements via one connector). Thus, the corresponding trace granularity is analogously low. Furthermore, the trace link validity is ensured by the actual model transformation.

Valid Traceability

We distinguish between a trace link associating a source and a target artifact, and the fact whether both artifacts are indeed semantically related to each other [*FHM12]. Requirements management tools, traceability management tools, or modeling languages typically allow arbitrary linking between trace artifacts, whereas the checks they provide only determine whether a trace artifact is linked at all. This leads to the problem that a trace link can accidentally be created between actually unrelated artifacts, or that two formerly related and linked artifacts are not related anymore after one of the linked artifacts has changed.

We consider a trace link as *valid*, iff the associated source and target artifact are indeed related to each other [*FHM12]. That is, a trace link is valid, if its source and target artifacts are in a traceability relation that can be described by a set of constraints. Several such relations are discussed in literature [BLY09; EAG06; ZSPK03], and we focus on the *overlap relation* [ZSPK03]. This is a relation between two different trace artifacts, which refer to a common feature of the SUD. Trace link validity can be determined, for example, by evaluating predefined rules w.r.t. to the overlap relation based on a traceability information model [MGP09], as we have also shown in [*FHM12].

Traceability literature published at the same time or later as our definition [*FHM12] refers to trace link validity also as the correctness property of the overall *traceability quality* [GCH$^+$12] or as *trace integrity* [CGH$^+$14].

2.1.2 The Model-based Traceability Management Tool CAPRA

In this thesis, we apply the model-based traceability management tool CAPRA [MS16; CAPRA] to semi-automatically establish vertical inter-model traceability between CONSENS system models and MSD specifications. The traceability information model is specified by means of a traceability metamodel, which is freely specifiable through an extension point. The permissible trace link types can be further constrained by means of Java customizations. The traceability metamodel as well as its resulting models are accessible to automatisms. The traceability models can hence be exploited or modified by model transformations. Furthermore, the traceability models store traces external to the models containing the actual trace artifacts, which does not unnecessarily "pollute" the latter ones [PDK$^+$11; KPP06; DPFK06]. Visual impact analyses can be conducted based on the traceability models.

2.2 Model-based Systems Engineering with CONSENS

In this section, we describe the CONSENS specification technique including its modeling language and the method to apply the language. As an example, we introduce a CONSENS system model of the EBEAS that also serves as illustrative basis for the description of our transition technique in this thesis.

The CONSENS modeling language is divided into eight partial models describing different aspects of a software-intensive system: *Environment*, *Application Scenarios*, *Requirements*, *Functions*, *Active Structure*, *Shape*, *System of Objectives*, and *Behavior*. The partial model *Behavior* is furthermore subdivided into *Behavior – States* and *Behavior – Activities*.

All aspects specified by means of the CONSENS partial models are strongly interconnected by means of so-called *cross-references* or *interrelations* [DDGI14]. In terms of our extended terminology for model-based traceability (cf. Section 2.1.1.2), these cross-references are specified by means of relational, intra-model, and explicit trace links of different types between elements of different partial models. In the following, we refer to this kind of cross-references as relational trace links.

Based on the method presented in [GV14], Figure 2.1 depicts the process step order to specify the particular partial models documented by means of the Business Process Model and Notation (BPMN) [OMG14b]. The overall process is specified by means of a BPMN *private process*. Work results are specified as BPMN *data objects* (document icons), and persistent models that are subject to update and retrieval operations are specified as BPMN *data stores* (database icon). The process steps relevant in this thesis are grouped into the BPMN *sub-processes* Planning and Clarifying the Task and Conceptual Design on the System Level.

Figure 2.2 depicts excerpts of six SwRE-relevant partial models specified in CONSENS for the EBEAS. In the following, we exemplarily perform each of the process steps (cf. Figure 2.1) for this running example system model.

2.2.1 Analyze Environment

As usual in model-based RE (cf. [CHQW16; DTW12; PR11; DeM79]), the first process step in the CONSENS specification method is the analysis of the environment. It has the aim to define the scope of the SUD, its system boundaries, and its external interfaces. The *Environment* is a structural partial model and distinguishes between the *system* (i.e., the SUD) and *environment*

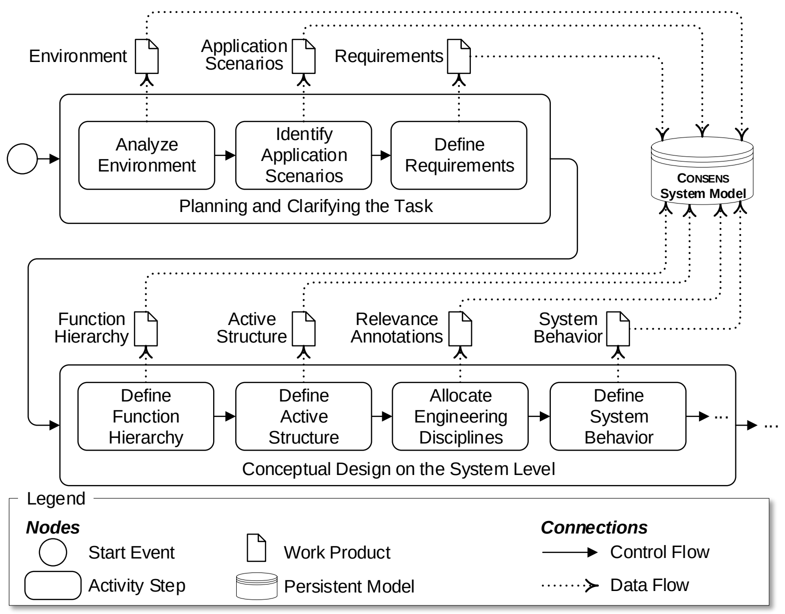


Figure 2.1: Excerpt of the CONSENS specification method (based on [GV14])

elements, where the system is viewed as a black box. The system and the environment elements represent non-physical elements like software components as well as physical elements like parts, assemblies, or modules. Different kinds of flows (i.e., material flows, energy flows, and information flows) represent the relationships between the system and the environment elements. The flows connect ports of the system and of the environment elements [Rie15].

For example, the *Environment* excerpt in the top left of Figure 2.2 embeds the SUD EBEAS into its environment. This environment encompasses other driver assistance systems and bus systems within the car. For example, the EBEAS shall continuously compute evasion maneuver trajectories and control the vehicle movement w.r.t. to these trajectories. For this purpose, the EBEAS sends trajectoryCommands to the environment element ActiveFrontSteering, which in turn sends back the current vehicle position on the trajectory via the information flow steeringInfo, resulting in a feedback loop. A FlexRay bus system delivers this logical information via the energy flow FlexRay signals. Furthermore, disturbing energy flows are indicated on the left-hand side representing interference factors that can disturb the SUD functionality (e.g., electromagnetic disturbances or temperatures). Countermeasures have to be conducted in the subsequent system design to tackle these interference factors. Figure A.7 in Appendix A.2.1 depicts the complete *Environment* for the EBEAS.

2.2.2 Identify Application Scenarios

Similarly to use cases [BS02], application scenarios represent initial assumptions of the system's behavior. They describe the most common operation modes of the system and the corresponding behavior on a coarse-grained abstraction level. Every application scenario describes a specific technical situation and the required behavior of the system by means of informal texts and sketches.

For example, the application scenario Emergency Evasion (top right of Figure 2.2) informally describes the trigger situation as well as the intended behavior for the overall emergency

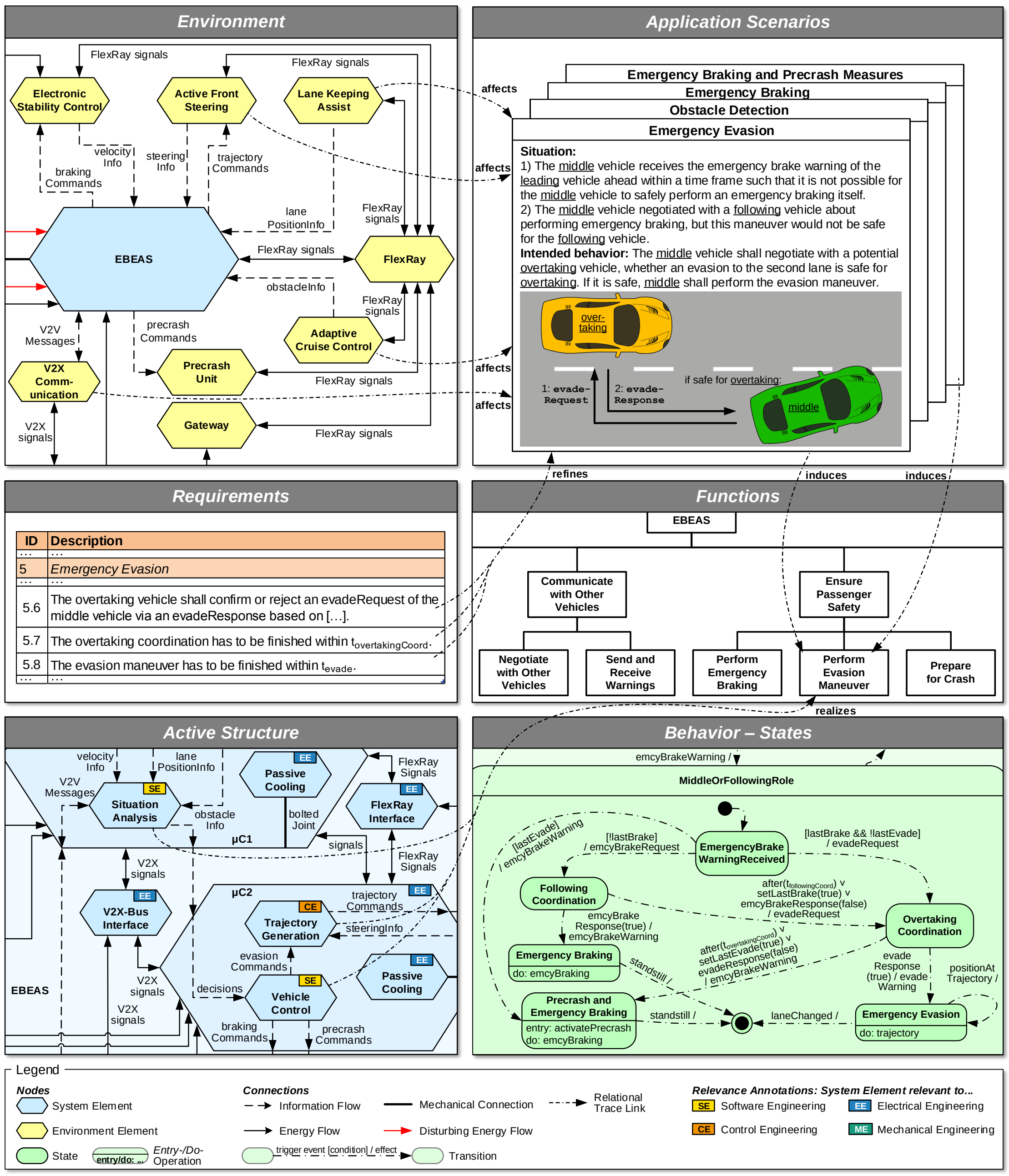

Environment
Application Scenarios
Electronic Stability Control
Active Front Steering
Lane Keeping Assist
EBEAS
FlexRay
Adaptive Cruise Control
Precrash Unit
V2X Communication
Gateway
FlexRay signals
velocity Info
steering Info
trajectory Commands
braking Commands
lane PositionInfo
obstacleInfo
precrash Commands
V2V Messages
V2X signals
affects
Emergency Braking and Precrash Measures
Emergency Braking
Obstacle Detection
Emergency Evasion
Situation:
1) The middle vehicle receives the emergency brake warning of the leading vehicle ahead within a time frame such that it is not possible for the middle vehicle to safely perform an emergency braking itself.
2) The middle vehicle negotiated with a following vehicle about performing emergency braking, but this maneuver would not be safe for the following vehicle.
Intended behavior: The middle vehicle shall negotiate with a potential overtaking vehicle, whether an evasion to the second lane is safe for overtaking. If it is safe, middle shall perform the evasion maneuver.
overtaking
1: evadeRequest
2: evadeResponse
if safe for overtaking:
middle
refines
induces
Requirements
ID
Description
5
Emergency Evasion
5.6
The overtaking vehicle shall confirm or reject an evadeRequest of the middle vehicle via an evadeResponse based on […].
5.7
The overtaking coordination has to be finished within t_overtakingCoord.
5.8
The evasion maneuver has to be finished within t_evade.
Functions
EBEAS
Communicate with Other Vehicles
Ensure Passenger Safety
Negotiate with Other Vehicles
Send and Receive Warnings
Perform Emergency Braking
Perform Evasion Maneuver
Prepare for Crash
realizes
Active Structure
Situation Analysis
Passive Cooling
FlexRay Interface
obstacle Info
bolted Joint
µC1
signals
µC2
V2X-Bus Interface
Trajectory Generation
Vehicle Control
evasion Commands
decisions
steeringInfo
SE
EE
CE
Behavior – States
emcyBrakeWarning /
MiddleOrFollowingRole
[lastEvade] / emcyBrakeWarning
[!lastBrake] / emcyBrakeRequest
EmergencyBrake WarningReceived
[lastBrake && !lastEvade] / evadeRequest
Following Coordination
after(t_followingCoord) ∨ setLastBrake(true) ∨ emcyBrakeResponse(false) / evadeRequest
emcyBrake Response(true) / emcyBrakeWarning
Overtaking Coordination
Emergency Braking
do: emcyBraking
standstill /
after(t_overtakingCoord) ∨ setLastEvade(true) ∨ evadeResponse(false) / emcyBrakeWarning
evade Response (true) / evade Warning
positionAt Trajectory /
Precrash and Emergency Braking
entry: activatePrecrash
do: emcyBraking
standstill /
laneChanged /
Emergency Evasion
do: trajectory
Legend
Nodes
System Element
Environment Element
State
entry/do: ...
Entry-/Do-Operation
Connections
Information Flow
Energy Flow
Mechanical Connection
Disturbing Energy Flow
Relational Trace Link
trigger event [condition] / effect
Transition
Relevance Annotations: System Element relevant to...
Software Engineering
Electrical Engineering
Control Engineering
Mechanical Engineering

Figure 2.2: Excerpt of the CONSENS system model of the EBEAS with partial models relevant to SwRE

evasion functionality. More concretely, the application scenario specifies that the participating vehicles shall negotiate about an evasion maneuver if there is no time for emergency braking or if emergency braking is unsafe for the following vehicle. Figure A.8 in Appendix A.2.1 depicts all application scenarios for the EBEAS.

The environment elements have relational trace links of the type affects to the application scenarios in which they are involved. For example, the environment elements ActiveFrontSteering and LaneKeepingAssist are involved in the application scenario Emergency Evasion but not in the remaining ones. Table A.1 in Appendix A.2.1 presents the full relational traceability between the partial models *Environment* and *Application Scenarios*.

2.2.3 Define Requirements

Based on the partial models *Environment* and *Application Scenarios*, the actual requirements have to be defined, specified, and managed. The partial model *Requirements* contains an organized collection of requirements that need to be fulfilled by the SUD. Requirements enable to expose what is expected from the future system. They form a milestone for the validation and verification in the subsequent development phases. In this thesis, we apply the *Requirements* partial model to refine and breaks down the application scenarios into particular, individually testable requirements.

For example, requirement 5.7 (center left of Figure 2.2) specifies that the overtaking coordination has to be finished within $t_{overtakingCoord}$.

All requirements with ID 5.x have refines trace links to the application scenario Emergency Evasion. These trace links enable that all requirements refining an application scenario can be identified from it and vice versa.

2.2.4 Define Function Hierarchy

A solution-neutral, functional view on the system is often applied in the model-based design of software-intensive systems [EAST13; VEFR12; DeM79]. A function is the general and required coherence between input and output parameters, aiming at fulfilling a task. Functions are realized by solution patterns and their concretizations. Starting with the overall function, a breakdown into subfunctions is conducted until useful technical solutions can be found for the functions.

In our example, the function hierarchy (center right in Figure 2.2) decomposes the overall functionality Ensure Passenger Safety into separate subfunctions for emergency braking, emergency evasion, and crash preparation. Figure A.9 in Appendix A.2.1 depicts the complete function hierarchy for the EBEAS.

The Systems Engineers specify a function to enable the realization of *Application Scenarios* or due to a disturbing influence in the *Environment*. They document such design decisions by means of relational traceability from the partial models *Application Scenarios* or *Environment* to the partial model *Functions*, that is, through trace links of the type induces. For example, the function Control Steering is induced from the application scenario Emergency Evasion. Tables A.2 and A.3 in Appendix A.2.1 present the full relational traceability from the partial models *Environment* and *Application Scenarios*, respectively, to the partial model *Functions*.

We furthermore provide a controlled natural language (e.g., [WAB+10]) representation of the function hierarchy by means of so-called requirement patterns [*FH15; *FHM14; *FH14; *DFHT13; *FHH+12; *FHM12; *HMD11; *HMM11; *Hol10].

2.2.5 Define Active Structure

By means of the *Active Structure*, the SUD is considered a white box. The *Active Structure* defines the internal structure of the system through *system elements* and relationships between themselves as well as from/to environment elements. As in the *Environment* (cf. Section 2.2.1), the relationships are specified by means of different kinds of flows. We call the combination of *Environment* and *Active Structure* together *system architecture*, which is one of the most important artifacts in a system model [*Mer15].

For example, the *Active Structure* (bottom left in Figure 2.2) considers the SUD EBEAS as white box and modularizes it into subsystems. It comprises the system elements µC1 and µC2 representing micro controllers that contain further system elements, which are connected to the environment elements from the *Environment* via delegation connectors. For example, the TrajectoryGeneration sends trajectoryCommands to the environment element ActiveFrontSteering and receives the steeringInfo from it (cf. Section 2.2.1). The Situation Analysis software serving as a sensor fusion component is deployed to µC1, and the Vehicle Control software controlling other driver assistance systems from the *Environment* is deployed to µC2. Furthermore, the EBEAS contains bus interfaces and micro controller cooling systems. Figure A.10 in Appendix A.2.1 depicts the complete *Active Structure* for the EBEAS.

The system elements have realizes trace links to the functions of the partial model *Functions* if they contribute to the realization of a function. In our example, the Situation Analysis, the Vehicle Control, and the TrajectoryGeneration realize the function Perform Evasion Maneuver. Table A.4 in Appendix A.2.1 presents the full relational traceability between the partial models *Active Structure* and *Functions*.

In general, the CONSENS modeling language does not distinguish between types and their actual applications in the model in the form of parts/roles or instances. This difference to object-oriented modeling languages like UMLstemming from the software engineering discipline exists due to the fact that the CONSENS modeling language stems from the mechanical engineering discipline where object-oriented concepts are only of theoretical nature. Thus, the CONSENS modeling language is more amenable to mechanical engineers, who find it difficult to understand the object-oriented concepts of the UML.The absent distinction between types and applications implies that only applications of environment and system elements are present in the structural partial models *Environment* and *Active Structure*. That is, a system element like Situation Analysis represents the actual corresponding software component in the SUD but does not define an additional, reusable type. However, the CONSENS modeling language distinguishes between environment/system element *templates* and *exemplars* for the case that reusability is required. That is, one can specify a system element template Passive Cooling with multiple exemplars if the system element occurs multiple times in the *Active Structure*.

2.2.6 Allocate Engineering Disciplines

While conceiving the *Active Structure*, the Systems Engineers determine during the discussion with the discipline-specific experts (e.g., the Software Engineers) which system elements are to be concretized in which disciplines. The Systems Engineers apply so-called *relevance annotations* [Rie15; HSST13] to specify the relevance of a system element to a certain discipline in the *Active Structure*. This information can be exploited to enable the automatic derivation of discipline-specific models from CONSENS system models (cf. Section 2.3). The relevance annotation “SE” specifies the relevance of a system element to the discipline of software engineering. That is, the system element represents a *discrete software component*, which realizes coordination behavior and hence communicates via messages and is concretized by means of state-based models. We call such system elements *SwE-relevant*.

For example, the discrete software components Situation Analysis and Vehicle Control in the *Active Structure* (bottom left in Figure 2.2) are deployed to µC1 and µC2, respectively. µC2 additionally contains the *continuous software component* TrajectoryGeneration, which is concretized in the control engineering discipline (relevance annotation "CE"). Furthermore, the micro controllers contain cooling systems and the EBEAS encompass bus interfaces. These system elements are relevant to the electrical engineering discipline. Figure A.10 in Appendix A.2.1 depicts the complete *Active Structure* including all relevance annotations for the EBEAS.

2.2.7 Define System Behavior

Software-intensive systems are characterized by different kinds of behavior. This is reflected in CONSENS by the two different behavior models *Behavior – States* and *Behavior – Activities*. The usage of the models depends on the underlying development task and on the kind of the SUD. *Behavior – Activities* specify flow-based behavior and are typically applied for the design of *transformational systems* [HP85] like production systems, for example. In this thesis, we focus on *reactive systems*, which continuously interact with their environment [HP85]. State-based behavioral models are well-suited for this system class [HP85], which are provided by the CONSENS language by means of the partial model *Behavior – States*.

We omit the description of the *Behavior – States* example excerpt in the bottom right of Figure 2.2 in this section and refer to Section 3.1.3 instead. This is due to the fact that we introduce assumptions on the way this partial model has to be specified in Chapter 3 in order to apply automatisms.

2.3 Automatic Derivation of Discipline-specific Design Models from CONSENS System Models

Based on an initial idea presented by Gausemeier et al. [GGS+07], Rieke [Rie15; Rie14; GSG+09; Rie08] exploits the relevance annotations in the *Active Structure* (cf. Section 2.2.6) for the automatic derivation of platform-independent MECHATRONICUML software engineering and control engineering design models. This enables the exploitation of the information specified in CONSENS system models and reducing the manual effort to transfer the system model information to the discipline-specific design. Furthermore, the approach ensures the consistency between the system models and the discipline-specific models, facilitating the orchestration of the development process by means of the system models. The declarative, bidirectional, and synchronizing model transformation approach Triple Graph Grammars (TGGs) [Sch95] is applied and extended for this purpose.

For example, Figure 2.3 depicts the platform-independent MECHATRONICUML software architecture that is automatically derived from the CONSENS *Active Structure* of the EBEAS (cf. Figure 2.2) using the approach of Rieke et al. It distinguishes between so-called discrete and continuous software components. Discrete software components realize the coordination behavior and are implemented by using MECHATRONICUML. The two discrete component parts (including their corresponding types) as well as their port interfaces and connectors in Figure 2.3 are derived from the equally named SwE-relevant system elements (labeled with the relevance annotation “SE”) Situation Analysis and Vehicle Control in the CONSENS *Active Structure*. Continuous software components realize the control behavior and are implemented using commercial-off-the-shelf control engineering tools like MATLAB/Simulink [Hei15; HRS13; HPR+12] or Dymola [Poh18; *PHMG14]. The continuous component part (including its corresponding type) tg: TrajectoryGeneration is derived from the corresponding system element labeled with the relevance annotation “CE”. The encapsulating structured software component type EBEAS is of hybrid nature due to the contained discrete as well as continuous component parts.

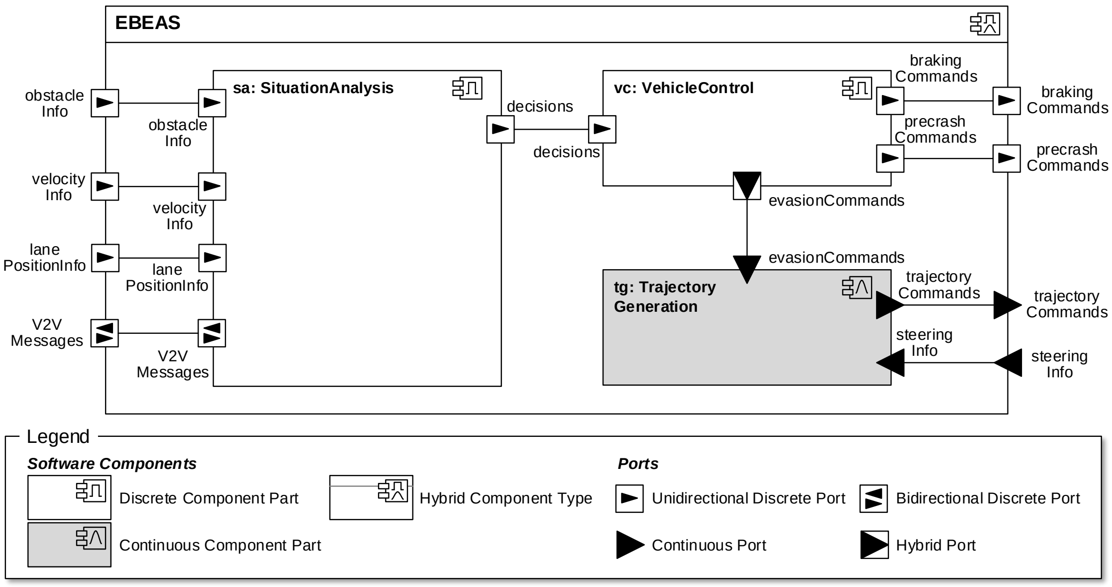


Figure 2.3: Platform-independent MECHATRONICUML software architecture automatically derived from the CONSENS *Active Structure* of the EBEAS (cf. Figure 2.2)

Beyond deriving and synchronizing structural models, Rieke specifies the behavior of the SwE-relevant system elements with the CONSENS partial model *Behavior – States*, automatically derives initial MECHATRONICUML behavioral models for the corresponding software components, and ensures the consistency of these models [RDS+12; RS12]. Furthermore, we transferred the structural part of the consistency-preserving transformation approach to SYSML4CONSENS [*PHM14] and to the automotive sector [*FHH+12; *FHM12; *HMM11; *MH11]. However, these approaches aim at deriving and refining discipline-specific design models but lack a transition to SwRE.

2.4 Modal Sequence Diagrams (MSDs)

In this section, we present foundations on Greenyer’s dialect of Modal Sequence Diagrams (MSDs) (cf. Section 1.1.2) relevant to this thesis. We refer to [*HFK+16] for the complete MSD dialect language and the method for its application. The dialect and the corresponding analysis techniques are implemented in the tool suite SCENARIOTOOLS MSD [ST-MSD], which bases on the UML modeling environment PAPYRUS [PAPYRUS]. The language of the MSD dialect is specified by means of the UML language [OMG17b] and the Modal UML profile, which we present in Section 2.5.1.

We introduce the overall structure of MSD specifications in Section 2.4.1, the MSD semantics in Section 2.4.2, and the MSD analysis techniques in Section 2.4.3.

2.4.1 Structure of MSD Specifications

An *MSD specification* is structured by means of *MSD use cases*, which encapsulate the overall requirements on the message-based coordination behavior to be provided by the system under development regarding a self-contained situation. An MSD use case encompasses a set of MSDs as well as the underlying structural elements, where the MSDs specify requirements on the inter-element coordination behavior.

For example, Figure 2.4 shows in the top an excerpt of an MSD specification, which resides at metamodel level M1. The MSD specification defines requirements on the coordination behavior of the EBEAS. Figure 2.4 depicts the MSD use case *ObstacleDetection* as part of the overall MSD specification. This MSD use case describes the requirements on the behavior that is expected in the case that an obstacle is detected by the leading vehicle in a semi-autonomous platoon.

MSD use cases specify UML classes encompassing operations as structural classifier basis for the remaining MSD specification elements. For example, the topmost class diagram ObstacleDetection contains a class **VehicleControl**, inter alia. This class encompasses an operation `enableBraking()`.

Based on the classes, UML collaborations (dashed ellipse symbol) specify the roles that participate in an MSD use case, for example, within the collaboration ObstacleDetection. These roles are typed by the UML classes, for example, the role vc: VehicleControl is typed by the class **VehicleControl**. Like CONSENS (cf. Section 2.2), MSD specifications distinguish between system and environment entities. That is, *system roles* (UML part symbols) are controlled by the system under development and *environment roles* (cloud symbols) are controlled by the environment. For example, the system roles sa: SituationAnalysis and vc: VehicleControl represent objects that are part of the system under development. The remaining roles are environment roles, which are outside the boundary of the system under development.

Furthermore, the UML collaborations encompass the actual MSDs (cf. referential trace links ownedBehavior). We distinguish MSDs into requirement MSDs (no stereotype applied) and assumption MSDs (an MSD with the stereotype «EnvironmentAssumption» applied). The former ones specify requirements on the coordination behavior of the system under development, whereas the latter ones specify assumptions on the behavior of the environment. For example, the MSD EmcyBrakingOnObstacleDetection is a requirement MSD specifying the emergency braking behavior of the EBEAS in the case the adaptive cruise detects an obstacle. The indicated assumption MSD CriticalPointsUntilCrash specifies which critical points in the environment (e.g., the last point to brake) are passed and sensed by the vehicle's driver assistance systems until a crash occurs, including their order (cf. Figure A.34 in Appendix A.2.2.3).

An MSD encompasses *MSD messages*, which are associated with a sending and a receiving lifeline as well as an operation signature. Such associations are established through referential trace links to the roles of the UML collaboration and to the class operations, respectively. For example, the receiving lifeline vc: VehicleControl of the MSD message `enableBraking` represents the equally named role in the UML collaboration. Furthermore, the MSD message `enableBraking` has the equally named operation of the class **VehicleControl** as signature. We explain the language constructs specific to MSD messages as well as other MSD ingredients and their semantics in Section 2.4.2.

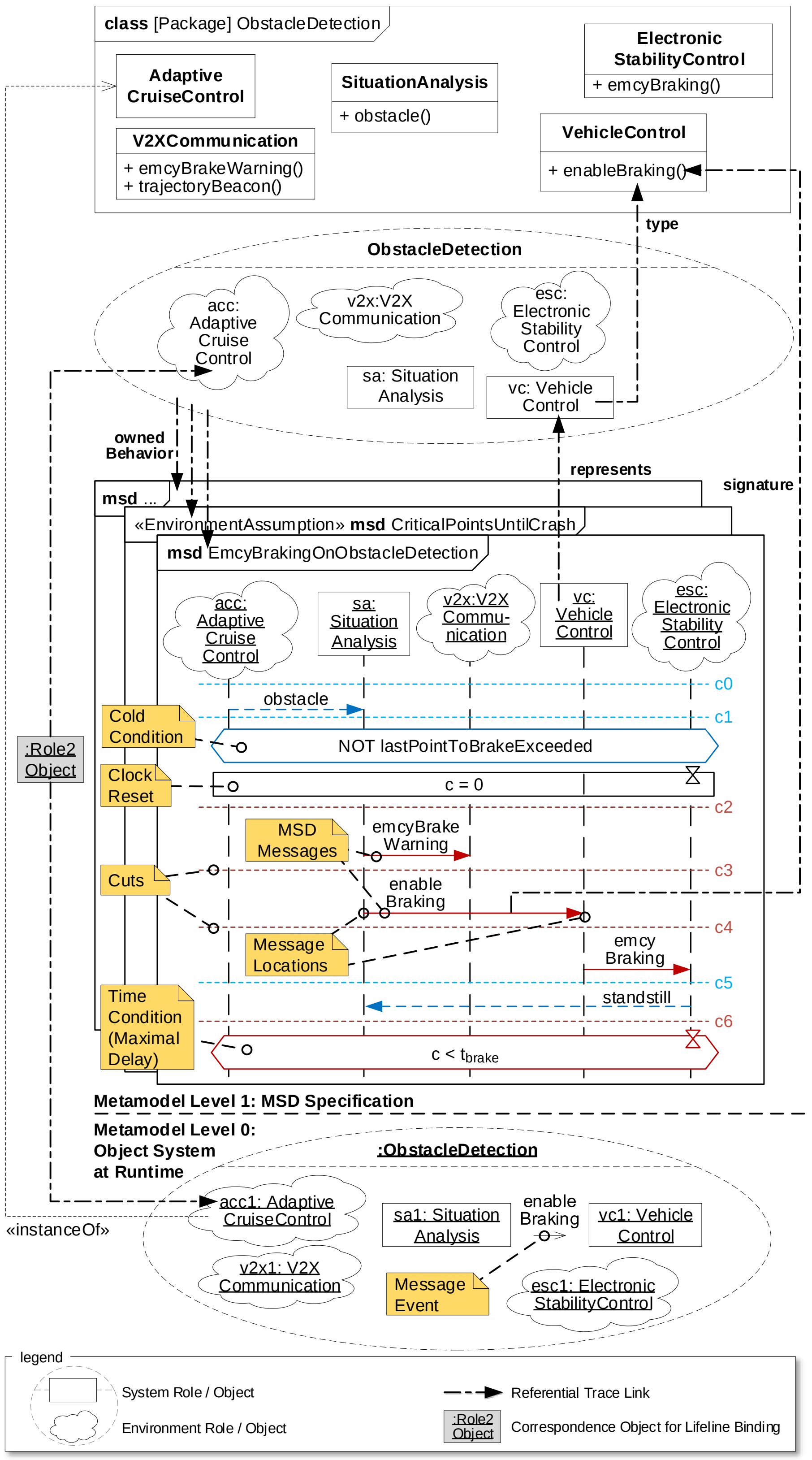


Figure 2.4: MSD use case *ObstacleDetection* in an MSD specification and at runtime

The bottom of Figure 2.4 depicts an object system at runtime (metamodel level M0), that is, during Play-out or an actual system execution. Here, the UML collaborations as well as its particular roles are instantiated. More specifically, ScenarioTools MSD Play-out derives the object system including a corresponding runtime metamodel from the structural MSD specification elements by means of model transformations (the runtime metamodel is not depicted in Figure 2.4). For example, the UML collaboration object :ObstacleDetection encompasses objects that are instances of the classes of the MSD specification. That is, the object acc1: AdaptiveCruiseControl has an «instanceOf» relationship to the class **AdaptiveCruiseControl**. In order to specify the relationship between the objects and the roles, ScenarioTools MSD Play-out provides a correspondence model that associates the objects with the roles through referential trace links. For example, the object :Role2Object associates the environment object acc1: AdaptiveCruiseControl and the corresponding environment role acc: AdaptiveCruiseControl. Via this mapping, lifelines are *bound* to objects in the object system that play the roles that the lifeline refers to. Actual message events at runtime at metamodel level M0 (e.g., the message event `enableBraking` depicted in the object system) correspond to MSD messages at metamodel level M1, as we explain in the following section.

2.4.2 MSD Semantics

Intuitively, an MSD progresses as *message events* occur in the object system at runtime as described in the MSD. The ScenarioTools MSD analysis techniques only consider *synchronous* messages where the sending and the receiving of the message together form a single message event. Each MSD message has a *temperature* and an *execution kind*, represented by its color and its line style, respectively. The temperature of a message represents its modality and can be *cold* (blue color) or *hot* (red color). The execution kind of a message can be *executed* (solid line) or *monitored* (dashed line).

If the progress reaches a monitored MSD message, the corresponding message event may or may not occur. If the MSD message is executed, the message event must eventually occur (liveness). If the MSD message is hot, no message event must occur (safety) that the scenario specifies to occur earlier or later. If such a message event occurs, this represents a hot or safety violation. If the MSD message is cold and a message event occurs that is specified to occur earlier or later, this "aborts" the progress of the MSD, representing a cold violation and a legal trace. Message events that have no corresponding MSD message specified in an MSD are ignored. That is, they do not influence the progress of the MSD, and the MSD does not impose requirements on them.

More specifically, the semantics of MSD messages and message events is as follows: A message event is *unified* with an MSD message iff the event name equals the message name (and hence the name of the associated operation signature) and the sending and receiving lifelines of the message are bound to the sending and the receiving objects. When a message event occurs in the object system that can be unified with the first message in an MSD an *active MSD* is created. Such a first message in an MSD is also called *minimal message* and is always cold and monitored. As further message events occur that can be unified with the subsequent MSD messages, the active MSD progresses. This progress is captured by the *cut*, which marks for every lifeline the *locations* of the MSD messages that were unified with the message events. If the cut reaches the end of an active MSD, the active MSD is terminated.

If the cut is in front of an MSD message on its sending and receiving lifeline, the message is *enabled*. For example, let us assume that the MSD EmcyBrakingOnObstacleDetection in

Figure 2.4 is active and in the cut c2, that is, the MSD message `emcyBrakeWarning` is enabled. If in this state the message event `emcyBrakeWarning` occurs at runtime, it is unified with the equally named MSD message, and the cut moves from c2 to c3. However, if in this state a message event `enableBraking` occurs at runtime (i.e., the sending of a warning is expected but the EBEAS omits this and directly engages the emergency braking), a hot violation occurs. In all active MSDs, the cut progresses for all enabled MSD messages on the occurrence of an unifiable message event. That is, one message event can be simultaneously unified with multiple MSD messages (defined by the sending/receiving lifeline and the operation signature) specified in several MSDs.

MSD messages and message events that are sent from environment roles and objects are called *environment messages* and *environment events*, respectively. Analogously, MSD messages and message events that are sent from system roles and objects are called *system messages* and *system events*, respectively.

2.4.2.1 Conditions

Beyond MSD messages and lifelines, MSDs can contain hot or cold *conditions*, which are represented as hexagons that cover one or more lifelines. Cold conditions are colored blue and hot conditions are colored red. Conditions can contain Object Constraint Language (OCL) [OMG14a] expressions that evaluate to a Boolean value. Conditions, if enabled, are evaluated immediately. If the expression of a hot or cold condition evaluates to true, the cut progresses beyond the condition. If the expression evaluates to false and the condition is cold, the active MSD terminates (cold violation). If a hot condition evaluates to false, the cut stays in front of the condition until it evaluates to true. If the condition never becomes true, this represents a liveness violation. If another message event occurs in such a cut that the MSD specifies to occur earlier or later or the cut reaches a hot condition with the expression `false`, this is a safety violation.

For example, the cold condition in the MSD EmcyBrakingOnObstacleDetection in Figure 2.4 contains the expression `NOT lastPointToBrakeExceeded`. That is, if the MSD is activated through an `obstacle` message event and the last point to brake is already exceeded, the subsequent message event sequence is not expected to occur anymore.

2.4.2.2 Real-time Requirements

Extending conditions, real-time requirements can be specified in MSDs by referring to *clock variables*. Clock variables are adopted from Timed Automata [AD94] relying on symbolic, dense time and represent real-value variables that increase synchronously and linearly with time. We distinguish between *clock resets* and *time conditions*. Clock resets are visualized as rectangles with an expression of the form $c = 0$ over a clock variable c. Time conditions have a temperature and define assertions w.r.t. clock variables. To this end, each time condition defines an expression of the form $c \bowtie value$, with a clock c, an operator $\bowtie \in \{<, \leq, >, \geq\}$, and an integer value $value$. For hot timed conditions, we distinguish *minimal delays* ($\bowtie \in \{>, \geq\}$) and *maximal delays* ($\bowtie \in \{<, \leq, \}$). If a minimal delay is enabled, but evaluates to false, the cut progresses as soon as it becomes true. Meanwhile the cut is hot, that is, no message that is not currently enabled in the active MSD is allowed to occur. If a maximal delay is enabled and evaluates to false, this is a liveness violation of the MSD because the MSD cannot progress. Due to the fact that once a maximal delay evaluates to false (i.e., the specified upper bound time constraint is violated) it can never evaluate to true afterward because the time linearly progresses, this also represents a safety violation.

For example, the MSD EmcyBrakingOnObstacleDetection in Figure 2.4 contains a clock reset and a maximal delay specifying that the message events unifiable with the MSD messages specified in between must occur within t_{brake} time units. This combination of a clock reset and a maximal delay forms a real-time requirement. Once the clock variable c has a value that is greater than t_{brake}, this represents a liveness and safety violation. In this thesis, we call this also a real-time requirement violation.

2.4.2.3 Existential and Universal MSDs

Typically, scenario-based specifications can be interpreted in an existential or in a universal way [SUB08]. An existential scenario specifies exemplary behavior of the system under development, where the system shall be able to produce at least one trace that fits the scenario. A universal scenario specifies requirements on all traces produced by the system under development, which is crucial for safety requirements, for example. Some scenario-based formalisms, for example Live Sequence Charts (LSCs) [DH01; HM03a], provide dedicated language constructs for both interpretations. Whereas our MSD dialect focuses on universal scenarios [Gre11; *HFK$^+$16], an existential scenario can be expressed in our dialect through an MSD containing only cold and executed system messages and cold and monitored environment messages [HM08].

2.4.3 Analysis Techniques

Based on the MSD dialect presented in the last two sections, Greenyer [Gre11; GF12; BGP13; GBC$^+$13] applied and extended two complementary automatic analysis techniques enabling the early detection of unintended behavior and inconsistencies between scenarios on requirements level. First, the *Play-out* algorithm (originally conceived for LSCs [HM03b; HM03a] and later also applied to the original MSD formalism [MH06; HKM07; HMSB10]) enables to simulatively validate untimed MSDs and thereby help the requirements engineer to understand the behavior emerging from the interplay of the scenarios [Gre11; GF12; BGP13]. Second, the synthesis of a global controller from an untimed or timed MSD specification [Gre11; GF12; GBC$^+$13] is a formal verification technique [Wan04] that enables ensuring the consistency and realizability of the requirements: An MSD specification is consistent and hence realizable iff a global controller can be synthesized from it (i.e., there exists a state-based, non-distributed implementation of the requirements).

We improved the MSD analysis techniques regarding real-time aspects by means of the Real-time Play-out approach [*BGH$^+$14; *BBG$^+$13] and a timed synthesis on top of it [*Jap15]. Real-time Play-out and thereby the timed synthesis apply the dense, symbolic time model known from Timed Automata [AD94]. However, these analysis techniques assume that the software is always fast enough to perform any finite number of steps before the occurrence of the next environment event [Gre11; HM03a]. Furthermore, MSDs address the platform-independent requirements on the system's coordination behavior. Platform-specific effects on the timing behavior have to be explicitly specified by means of minimal delays in order to analyze the timing behavior emerging from the software execution in distributed systems, resulting in awkward and cumbersome MSD specifications. Thus, we regard the timed MSD analysis techniques as *platform-independent analysis techniques*. In [*HS14; *Shi14], we presented an initial approach considering timing effects induced by target execution platforms in Real-time Play-out. However, this approach only considers static message send and task execution delays but misses more complex delay computations and dynamic timing behavior due to mutual exclusions of platform resources.

2.5 UML Profiles

The Unified Modeling Language (UML) [OMG17b] is a general-purpose and de-facto standard modeling language for the software engineering discipline. The UML provides an extension mechanism by means of so-called *profiles* for the specification of domain-specific modeling languages. In this section, we describe the UML profiles that we apply or extend in this thesis.

2.5.1 The Modal Profile

The language for our dialect of MSD specifications (cf. Section 2.4) is specified at metamodel level M2 by means of the UML language [OMG17b] and the Modal UML profile [Gre11; ST-MSD], whose original version was introduced by Harel and Maoz [HM08]. The Modal profile introduces the MSD-specific language constructs to the UML language, for example, the temperature and the execution kind of MSD messages as well as the real-time requirements and their temperature. Figure 2.5 depicts the Modal profile, and we explain its particular stereotypes in the following.

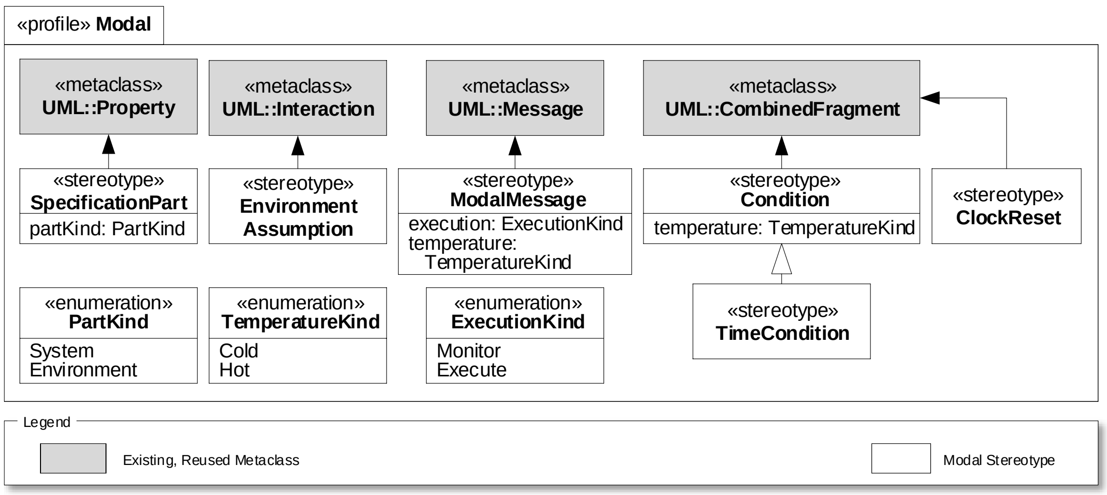

«profile» Modal
«metaclass» UML::Property
«metaclass» UML::Interaction
«metaclass» UML::Message
«metaclass» UML::CombinedFragment
«stereotype» SpecificationPart
partKind: PartKind
«stereotype» Environment Assumption
«stereotype» ModalMessage
execution: ExecutionKind
temperature: TemperatureKind
«stereotype» Condition
temperature: TemperatureKind
«stereotype» ClockReset
«enumeration» PartKind
System
Environment
«enumeration» TemperatureKind
Cold
Hot
«enumeration» ExecutionKind
Monitor
Execute
«stereotype» TimeCondition
Legend
Existing, Reused Metaclass
Modal Stereotype

Figure 2.5: The Modal profile

SpecificationPart A role as part of a UML collaboration.

> **partKind** Specifies whether the role is a system role or an environment role.

EnvironmentAssumption An environment MSD.

ModalMessage An MSD message.

> **execution** Specifies whether the MSD message is monitored or executed.
>
> **temperature** Specifies whether the MSD message is cold or hot.

Condition A condition.

> **temperature** Specifies whether the condition is cold or hot.

TimeCondition A time condition.

> **temperature** Specifies whether the time condition is cold or hot.

ClockReset A clock reset.

2.5.2 The Systems Modeling Language (SysML)

The Systems Modeling Language (SysML) [OMG17a] is a general-purpose and de-facto standard modeling language for MBSE. SysML is defined as a profile based on a subset of the UML, the UML4SysML. Figure 2.6 depicts the excerpt of the UML4SysML UML subset and the SysML profile that we use and extend in Chapter 3, and we explain the particular metaclasses and stereotypes that we extend in the following.

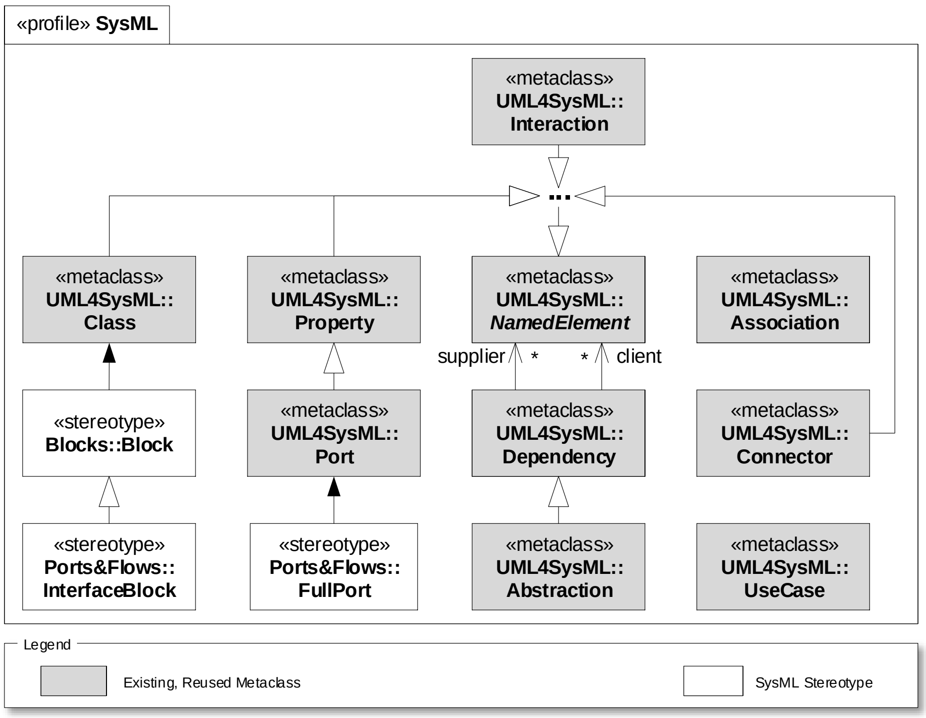

«profile» SysML
«metaclass» UML4SysML:: Interaction
...
«metaclass» UML4SysML:: Class
«metaclass» UML4SysML:: Property
«metaclass» UML4SysML:: NamedElement
«metaclass» UML4SysML:: Association
supplier
*
*
client
«stereotype» Blocks::Block
«metaclass» UML4SysML:: Port
«metaclass» UML4SysML:: Dependency
«metaclass» UML4SysML:: Connector
«stereotype» Ports&Flows:: InterfaceBlock
«stereotype» Ports&Flows:: FullPort
«metaclass» UML4SysML:: Abstraction
«metaclass» UML4SysML:: UseCase
Legend
Existing, Reused Metaclass
SysML Stereotype

Figure 2.6: Excerpt of the UML4SysML UML subset and the SysML profile used in this thesis

Block "A Block is a modular unit that describes the structure of a system or element." [OMG17a, Section 8.3.2.4]

Property "Properties are the primary structural feature of blocks. [...] Part properties are used to describe the composition hierarchy of a block and define a part in the context of its whole." [FMS12, Section 7.1]

InterfaceBlock "A [...] port is typed by an interface block which specifies the features that can be accessed via the port." [FMS12, Section 7.6]

FullPort “Full ports specify a separate element of the system from the owning block or its internal parts.” [OMG17a, Section 9.3.2.8]

Abstraction “An Abstraction is a Relationship that relates two Elements or sets of Elements that represent the same concept at different levels of abstraction or from different viewpoints.” [OMG17b, Section 7.8.1.1] It provides a modeling concept for relational traceability, because it specializes the UML metaclass **Dependency** that in turn specializes **DirectedRelationship** (cf. Section 2.1.1.2 and [OMG17b]).

Association “A link is a tuple of values that refer to typed objects. An Association classifies a set of links, each of which is an instance of the Association. Each value in the link refers to an instance of the type of the corresponding end of the Association.” [OMG17b, Section 11.8.1.1]

Connector “A Connector specifies links that enables communication between two or more instances. In contrast to Associations, which specify links between any instance of the associated Classifiers, Connectors specify links between instances playing the connected parts only.” [OMG17b, Section 11.8.10.1]

UseCase “A UseCase specifies a set of actions performed by its subjects, which yields an observable result that is of value for one or more Actors or other stakeholders of each subject.” [OMG17b, Section 18.2.5.1]

2.5.3 Modeling and Analysis of Real-Time Embedded Systems (MARTE)

The UML profile Modeling and Analysis of Real-Time Embedded Systems (MARTE) [OMG11] provides modeling means for design and analysis aspects for the embedded software part of software-intensive systems. For the most part, we cite descriptions of [OMG11], [SG14], or [OMG17b] in the following.

2.5.3.1 Subprofile Non-functional Properties Modeling (NFPs) and the Model Library MARTE_Library

In Chapter 4, we either use the UML primitive data types Boolean and Integer or more complex data types as provided by the MARTE subprofile Non-functional Properties Modeling (NFPs) [OMG11, Chapter 7/Annex F.2] and its model library MARTE_Library [OMG11, Annex D]. We introduce the used latter ones in the following.

Figure 2.7 depicts the excerpt of the NFPs subprofile that we use in Chapter 4, and we explain its particular stereotypes in the following.

Dimension “A Dimension is a relationship between a quantity and a set of base quantities in a given system of quantities.” [OMG11, Section 8.3.2.2]

symbol “This attribute represents the symbol used to designate the dimension.” [OMG11, Section 8.3.2.2]

baseDimension “This attribute represents the base dimensions by which the dimension of a derived quantity unit is created. Basic dimensions do not require this attribute.” [OMG11, Section 8.3.2.2]

Unit “Unit is a qualifier of measured values in terms of which the magnitudes of other quantities that have the same physical dimension can be stated.” [OMG11, Section 8.3.2.6]

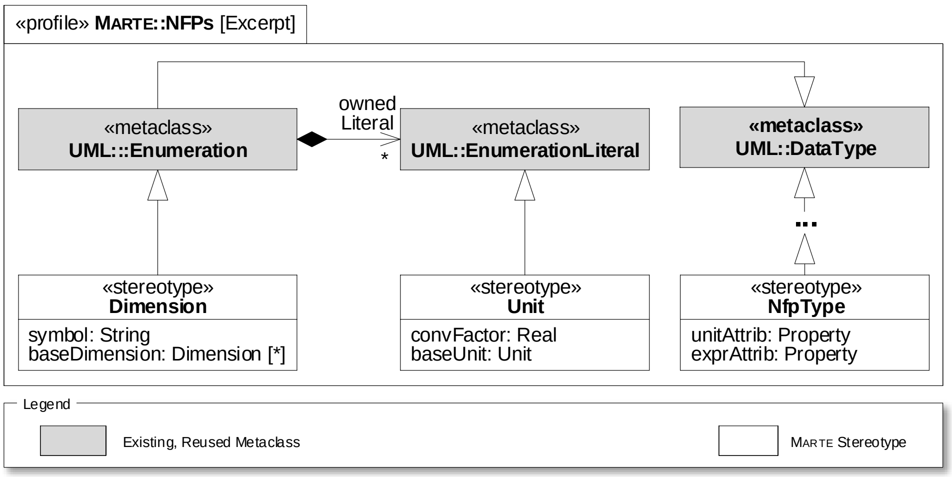


Figure 2.7: Excerpt of the MARTE NFPs subprofile used in this thesis

convFactor "This parameter allows referencing measurement units to other base units by a numerical factor." [OMG11, Section 8.3.2.6]

baseUnit "This attribute represent the base unit by which a derived measurement unit is created. Basic units do not require this attribute." [OMG11, Section 8.3.2.6]

NfpType "The actual physical values are modeled by specialized UML data types, called *NFP types*, represented by the standard MARTE stereotype, **NfpType**." [SG14, Section 3.2]

unitAttrib "measurement unit declaration that apply to all the value specifications of the NFP. Usually, it is an enumeration data type with a list of the valid measurement units." [OMG11, Section 8.3.2.4]

exprAttrib "attributes representing an expression. MARTE uses the [Value Specification (VSL)] language to define expressions." [OMG11, Section 8.3.2.4] We use this attribute specified by means of VSL expressions [OMG11, Annex B] to specify interval values.

The MARTE model library MARTE_Library [OMG11, Annex D] defines pre-defined **Dimensions** and **NfpTypes**, inter alia, for convenience. Figure 2.8 depicts the excerpt of **Dimensions** and **NfpTypes** that we use in Chapter 4, and we explain them in the following.

NFP_CommonType "This is the parent **NfpType** that contains common parameters [...] of the various **NfpTypes** defined in MARTE." [OMG11, Annex D.2.7]

expr "This attribute is bound to the exprAttrib attribute of the **NfpType** stereotype [(cf. description above)], to denote that it is the attribute that contains the value expression." [SG14, Appendix A]

NFP_Integer Specifies Integer intervals in Chapter 4.

NFP_Real "This is a general type used as a base for any physical data types whose values can be represented by real numbers." [SG14, Section 3.2] By means of this **NfpType**, we specify intervals of real-numbered values.

NFP_Duration "**NFP_Duration** is used to type elements that represent intervals in time [...]" [SG14, Section 3.3] with the unit of type **TimeUnitKind**.

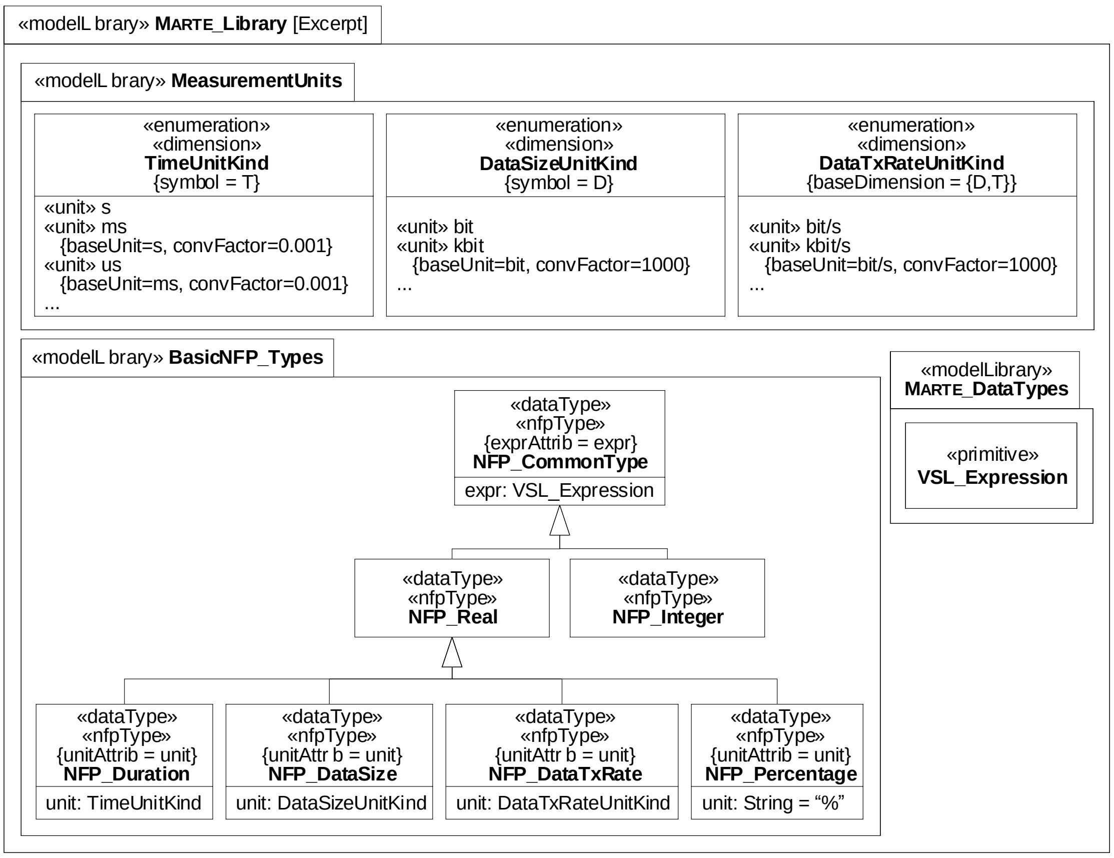

«modelLibrary» MARTE_Library [Excerpt]
«modelLibrary» MeasurementUnits
«enumeration» «dimension» TimeUnitKind {symbol = T}
«unit» s
«unit» ms {baseUnit=s, convFactor=0.001}
«unit» us {baseUnit=ms, convFactor=0.001}
...
«enumeration» «dimension» DataSizeUnitKind {symbol = D}
«unit» bit
«unit» kbit {baseUnit=bit, convFactor=1000}
...
«enumeration» «dimension» DataTxRateUnitKind {baseDimension = {D,T}}
«unit» bit/s
«unit» kbit/s {baseUnit=bit/s, convFactor=1000}
...
«modelLibrary» BasicNFP_Types
«dataType» «nfpType» {exprAttrib = expr} NFP_CommonType
expr: VSL_Expression
«dataType» «nfpType» NFP_Real
«dataType» «nfpType» NFP_Integer
«dataType» «nfpType» {unitAttrib = unit} NFP_Duration
unit: TimeUnitKind
«dataType» «nfpType» {unitAttrib = unit} NFP_DataSize
unit: DataSizeUnitKind
«dataType» «nfpType» {unitAttrib = unit} NFP_DataTxRate
unit: DataTxRateUnitKind
«dataType» «nfpType» {unitAttrib = unit} NFP_Percentage
unit: String = "%"
«modelLibrary» MARTE_DataTypes
«primitive» VSL_Expression

Figure 2.8: Excerpt of the MARTE_Library used in this thesis

NFP_DataSize By means of this **NfpType**, we specify intervals of data sizes with the unit of type **DataSizeUnitKind**.

NFP_DataTxRate By means of this **NfpType**, we specify intervals of data transmission rates with the unit of type **DataTxRateUnitKind**.

NFP_Percentage By means of this **NfpType**, we specify percentage intervals.

2.5.3.2 Subprofile Generic Resource Modeling (GRM)

The MARTE subprofile Generic Resource Modeling (GRM) [OMG11, Chapter 10/Annex F.4] provides modeling means for the specification of generic resources of execution platforms. The bottom of Figure 2.9 depicts the excerpt of the GRM subprofile that we use in Chapter 4, and we explain its particular stereotypes in the following.

Resource "[...] represents a physically or logically persistent entity that offers one or more services." [OMG11, Annex F.4.20] (cf. [OMG11, Section 10.3.2.12])

CommunicationMedia "[...] represents the means to transport information from one location to another." [OMG11, Annex F.4.7] (cf. [OMG11, Section 10.3.2.4])

capacity "Capacity of the communication element [...]" [OMG11, Section 10.3.2.4]

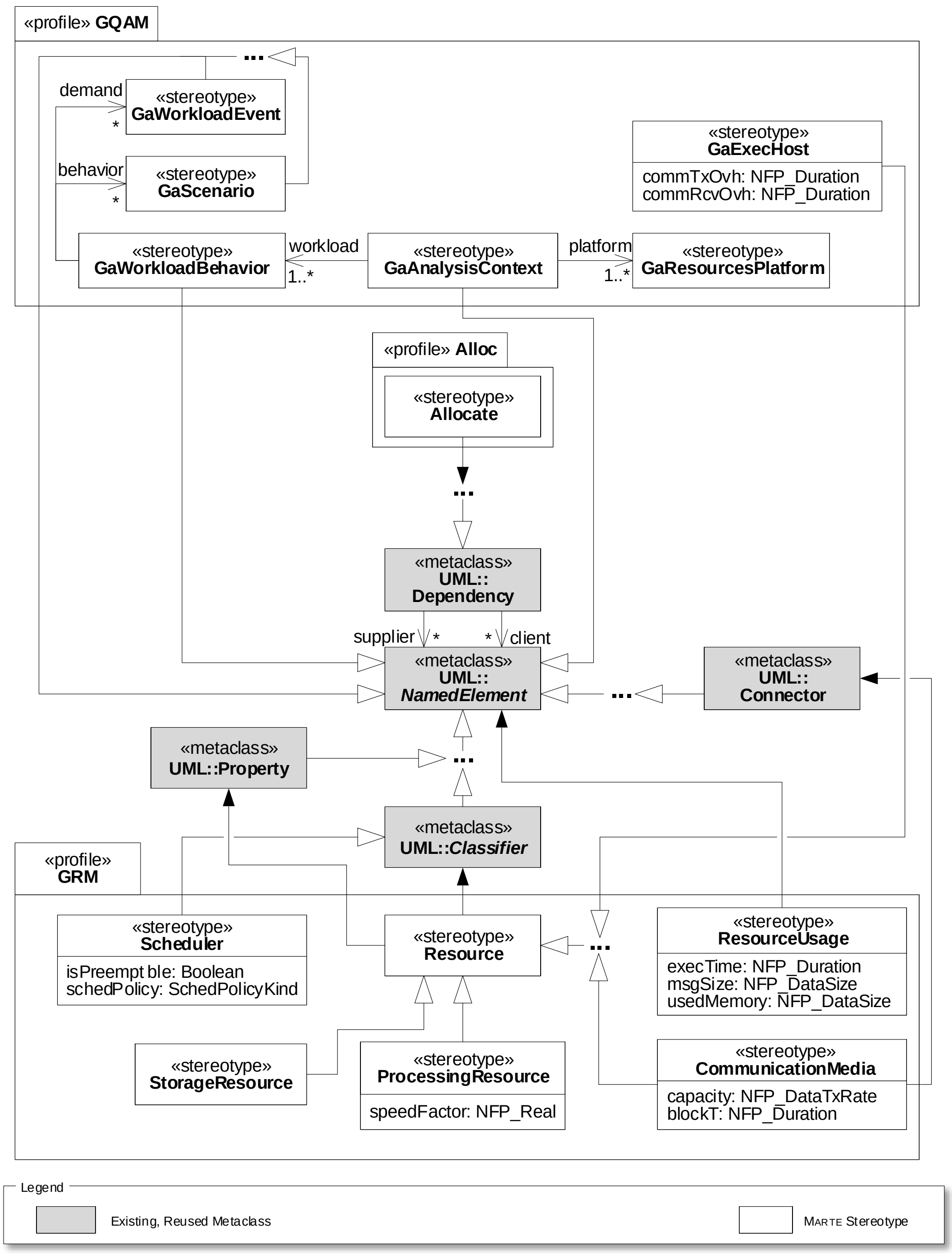

«profile» GQAM
demand
«stereotype» GaWorkloadEvent
behavior
«stereotype» GaScenario
«stereotype» GaExecHost
commTxOvh: NFP_Duration
commRcvOvh: NFP_Duration
«stereotype» GaWorkloadBehavior
workload
1..*
«stereotype» GaAnalysisContext
platform
1..*
«stereotype» GaResourcesPlatform
«profile» Alloc
«stereotype» Allocate
«metaclass» UML:: Dependency
supplier
client
«metaclass» UML:: NamedElement
«metaclass» UML:: Connector
«metaclass» UML::Property
«metaclass» UML::Classifier
«profile» GRM
«stereotype» Scheduler
isPreemptible: Boolean
schedPolicy: SchedPolicyKind
«stereotype» Resource
«stereotype» ResourceUsage
execTime: NFP_Duration
msgSize: NFP_DataSize
usedMemory: NFP_DataSize
«stereotype» StorageResource
«stereotype» ProcessingResource
speedFactor: NFP_Real
«stereotype» CommunicationMedia
capacity: NFP_DataTxRate
blockT: NFP_Duration
Legend
Existing, Reused Metaclass
MARTE Stereotype

Figure 2.9: Excerpt of the MARTE subprofiles GRM, GQAM, and Alloc used in this thesis

- **blockT** "Time the communicationMedia is blocked and cannot transmit due to the transmission of one communication quantum." [OMG11, Section 10.3.2.4]

ProcessingResource "[...] an active, protected, executing-type resource that is allocated to the execution of schedulable resources [...]." [OMG11, Section 10.3.2.10]

- **speedFactor** "[...] a relative factor for annotating the processing speed expressed as a ratio to the speed of the reference processingResource for the system under consideration." [OMG11, Section 10.3.2.10]

StorageResource "[...] represents the different forms of memory." [OMG11, Annex F.4.36] (cf. [OMG11, Section 10.3.2.17])

Scheduler "[...] brings access to [...] resources following a certain scheduling policy." [OMG11, Annex F.4.30] (cf. [OMG11, Section 10.3.2.15]).

- **isPreemptible** "Qualifies the capacity of the scheduler for preempting schedulable resources once the access to the processing capacity has been granted upon the arrival of a new situation where a different schedulable resource has to execute." [OMG11, Section 10.3.2.15]
- **schedPolicy** "Scheduling policy implemented by the scheduler." [OMG11, Section 10.3.2.15]

ResourceUsage "[...] represents the run-time mechanism that effectively requires the usage of the resource." [OMG11, Annex F.4.27]). (cf. [OMG11, Section 10.3.2.13]).

- **execTime** "Time that the resource is in use due to the usage." [OMG11, Section 10.3.2.13]
- **msgSize** "Amount of data transmitted by the resource." [OMG11, Section 10.3.2.13]
- **usedMemory** "Amount of memory that will be used from a resource but that will be immediately returned, and hence should be available while the usage is in course." [OMG11, Section 10.3.2.13]

2.5.3.3 Subprofile Generic Quantitative Analysis Modeling (GQAM)

The MARTE subprofile Generic Quantitative Analysis Modeling (GQAM) [OMG11, Chapter 15/Annex F.10] provides modeling means for the specification of generic and quantitative aspects relevant to automatic analysis techniques. The top of Figure 2.9 depicts the excerpt of the GQAM subprofile that we use in Chapter 4, and we explain its particular stereotypes in the following.

GaAnalysisContext "For a given analysis, the context identifies the model elements [...] of interest and specifies global parameters of the analysis." [OMG11, Section 15.3.2.2]

- **platform** "Logical containers for the resources used in the behavior to be analyzed." [OMG11, Annex F.10.2]
- **workload** "Logical container for the workload model and for the system-level behavior triggered by it." [OMG11, Annex F.10.2]

GaResourcesPlatform "A logical container for the resources used in an analysis context." [OMG11, Section 15.3.2.10]

GaWorkloadBehavior "A logical container for the analyzed behavior and the workload that triggers it, in an analysis context." [OMG11, Section 15.3.2.15]

- **demand** “Indicates the request event streams that are part of this container.” [OMG11, Annex F.10.19]
- **behavior** “Indicates the set of system behaviors used for analysis.” [OMG11, Annex F.10.19]

GaScenario “[...] defines the behavior in response to a request event, including the sequence of steps and their use of resources.” [OMG11, Annex F.10.3] (cf. [OMG11, Section 15.3.2.12])

GaWorkloadEvent “A stream of events that initiate system-level behavior.” [OMG11, Section 15.3.2.16]

GaExecHost “A CPU or other device that executes functional steps.” [OMG11, Annex F.10.8] (cf. [OMG11, Section 15.3.2.7])

- **commTxOvh** “[...] denotes the overhead involved in sending a message.” [SG14, Section 9.4.4]
- **commRcvOvh** “[...] denotes the overhead involved in receiving a message.” [SG14, Section 9.4.4]

2.5.3.4 Subprofile Allocation Modeling (Alloc)

The MARTE subprofile Allocation Modeling (Alloc) [OMG11, Chapter 11/Annex F.5] provides modeling means for the specification of allocations of typically logical (i.e., application software) elements to physical and technical (i.e., execution platform) elements. The middle of Figure 2.9 depicts the excerpt of the Alloc subprofile that we use in Chapter 4, and we explain its used stereotype in the following.

Allocate “[...] a mechanism for associating elements from a logical context, application model elements, to named elements described in a more physical context, execution platform model elements.” [OMG11, Section 11.3.2.1] It provides a modeling concept for relational traceability, because it extends the UML metaclass **Dependency** that in turn specializes **DirectedRelationship** (cf. Section 2.1.1.2 and [OMG17b]).

- **client** “The Element(s) dependent on the supplier Element(s).” [OMG17b, Section 7.8.4.5]
- **supplier** “The Element(s) on which the client Element(s) depend in some respect.” [OMG17b, Section 7.8.4.5]

2.6 Timing Analysis Techniques for Hard Real-time Systems

Real-time systems are distinguished into hard and soft real-time systems [But11]. The violation of hard real-time requirements may cause catastrophic consequences (e.g., people are harmed), whereas the violation of soft real-time requirements has some utility but causes a performance degradation [But11].

Hard real-time systems must be designed to tolerate worst-case conditions [JP86]. In general, *schedulability analysis* (e.g., [But11]) for hard real-time systems investigates whether jobs with each an activation time, a processing time, and a deadline w.r.t. the activation time can be scheduled on resources so that always all deadlines are met.

In the following two sections, we present two concrete techniques for the schedulability analysis of hard real-time systems.

2.6.1 Response Time Analysis

Response time analysis [SAÅ+04; ABD+95] is a well established a-priori analysis technique to check the schedulability of hard real-time systems. It calculates upper bounds on the *response times* of all jobs and checks whether all response times fulfill the corresponding deadlines. In simplified terms, the response time of a job is defined as its activation time plus its processing time plus the sum of potential preemption times by other jobs. A job can be a task to be executed on a processing unit resource or a message to be transmitted via a communication medium resource.

In the case of tasks, the processing time is the execution time that the processing unit needs to execute the task. *Worst-case execution times* (WCETs) of the tasks are input to *task response time analyses* [PB00]. For this purpose, WCETs have to be measured or upper bounds have to be computed in *WCET analyses*, which requires the final platform-specific code or a very detailed model of the system, respectively [WEE+08].

In the case of messages, the processing time is the transmission time that the communication medium needs to transmit the message. For the corresponding *message response time analyses*, the properties of the physical medium and of the communication protocol influencing the transmission time are typically known. However, the activation time also encompasses a queuing jitter that is inherited from the worst-case response time of the sending task [THW94]. Thus, the results of message response time analyses also depend on the final platform-specific code.

2.6.2 End-to-End Response Time Analysis

Whereas response time analyses as described in the last section enable determining the schedulability of individual tasks and messages, determining the overall schedulability of distributed systems requires a more holistic view on the overall system [Kop11]. That is, distributed real-time systems encompass *event chains* [AUTOSAR; EAST13; *KHD14; *Koc13; *Tee12] starting with an initial system stimulus and involving multiple software components that may be deployed at different ECUs until the system provides an externally observable response event. The timing behavior of event chains converges from the occurrence of task start and completion events as well as of different events involved in the message transmission. The *end-to-end response time* of an event chain is defined as the amount of time elapsed between the arrival of an event at the first task and the production of the response by the last task in the chain [MNS+17].

High-level real-time requirements are formulated w.r.t. such event chains. That is, they impose timing constraints on the event chains between an initial system stimulus and an externally observable response event. In order to verify whether an overall system meets its high-level real-time requirements, *end-to-end response time analyses* [TC94; FRNJ08; MMS13; MMS14] determine whether the aggregated response times of the individual tasks and messages of an event chain fulfill the real-time requirements. Since techniques and tools providing such analyses rely on the response times of the individual jobs (cf. last section), they require the final platform-specific implementation [DWUL17; MNS+17; MSN+15]. Thus, they can also be only applied in late development phases like the techniques and tools for the analysis of the individual response times.

2.7 Clock Constraint Specification Language (CCSL)

The MARTE time domain model [OMG11, Chapter 9] defines three different classes of time abstraction. The most abstract of these three classes relies on logical time [Lam78; Fid91], which is designed for distributed and concurrent systems and treats time as partial orders of causally and temporally related event occurrences. In the logical time interpretation of the MARTE time domain model, a timed event occurrence refers to one *instant*, where discrete-time *logical clocks* give access to such instants.

In order to provide modeling means for the symbolic specification of partial order sets on the instants of the clocks, MARTE defines the textual concrete syntax and the informal semantics of the declarative *Clock Constraint Specification Language* (CCSL) [OMG11, Annex C.3]. André [And09] formalizes the semantics of CCSL, thereby enabling the simulation of CCSL models in the tool suite TIMESQUARE [DM12a; T^2]. TIMESQUARE has particularly been applied for the timing analysis (cf. Section 2.6) of software-intensive systems (e.g., [GDM$^+$15; GDPM13; PD11; XJMZ11; MAD09; MPA09]).

Section 2.7.1 sketches the semantics of CCSL and its realization in TIMESQUARE. Subsequently, we explain the pre-defined CCSL constraints that we apply in Chapter 4. Finally, we explain an extension of CCSL that enables to specify user-defined constraints.

2.7.1 CCSL Semantics and its Realization in TIMESQUARE

André [And09] defines the semantics of CCSL with respect to a *time system* TS, which consists of a clock model $M = (C, S)$ and an initial configuration X^0. From a static point of view, a clock model $M = (C, S)$ is specified by a *CCSL model* and encompasses a finite set of discrete clocks C as well as a constraint specification S for these clocks. That is, S defines constraints on the clocks in C by means of clock expressions and relations (cf. Section 2.7.2). The initial configuration X^0 defines the initial instants for all clocks in C.

From a dynamic point of view, a time system TS changes its state by *firing* clocks, where firing a clock means that the clock ticks. A set of simultaneous clock firings is named a *step*. In order to compute a step, all *enabled clocks* that are fireable are determined in a set $E \subset C$. A subset $F \subset E$ of *simultaneously fireable clocks* is identified according to the constraint specification S and fired during the transition to the next step. A particular execution of TS is called a *run*, which is a possibly infinite sequence of steps.

In order to compute the set of simultaneously fireable clocks F, André [And09] presents a mapping from CCSL models to a Boolean expression on $\mathscr{C}$, where $\mathscr{C}$ is a set of Boolean variables in bijection with C. For any $c \in \mathscr{C}$, the corresponding clock ticks iff c is evaluated to true. Such a Boolean expression representing a CCSL model is encoded as a Reduced Ordered Binary Decision Diagram (ROBDD) [Bry86; Ake78; Lee59], so that a BDD solver can efficiently compute the particular CCSL specification steps. This approach is implemented in the simulation tool TIMESQUARE [DM12a; T^2], which automatically transforms CCSL models into ROBDDs and computes the steps with a BDD solver. A step that cannot be computed results in a deadlock of the simulation.

TIMESQUARE stores the resulting traces of particular runs in Value Change Dump (VCD) [IEC04] files. VCD files originate from the domain of electronic design automation and are originally intended to store signal traces from logic simulation tools to predict the behavior of digital circuits and hardware description languages. Such VCD files storing signal traces can be used to visualize the particular signal values by means of waveforms. TIMESQUARE reinterprets VCD files and their waveform visualization by not storing and visualizing signal values but ticks of the CCSL specification clocks. That is, a clock ticks (value 1) or does not tick (value 0) at a point in time. This leads to a visualization of CCSL specification runs that is similar to UML timing diagrams [OMG17b].

If the state space is bounded, TIMESQUARE allows also a complete exploration of the CCSL model state space as well as a controller synthesis on this basis [YTB+11], similarly to the MSD analysis techniques (cf. Section 2.4.3).

2.7.2 Pre-defined CCSL Constraints

The constraint specification S on the clock set C (cf. Section 2.7.1) encompasses the two CCSL constraint kinds clock expressions and clock relations, which apply to the clocks' particular instant sets. André [And09] formalizes a set of pre-defined CCSL constraints, which TIMESQUARE provides as a model library. Thereby, these CCSL constraints can conveniently be used during the specification and simulation of CCSL models. In the following, we explain the clock expressions and relations that we apply in Chapter 4 and Appendix B.

2.7.2.1 Clock Expressions

Clock expressions define a new clock based on other clocks and possibly extra parameters. Figure 2.10 depicts exemplary TIMESQUARE simulation runs of the clock expressions that we use in Chapter 4 and Appendix B. We describe these expressions in the following.

Union (clocks: Set(Clock))[1] The clock specified by this expression ticks whenever one of the clocks in its parameter set clocks ticks (cf. Figure 2.10(a)).

Intersection (clocks: Set(Clock))[1] The clock specified by this expression ticks whenever all of the clocks in its parameter set clocks tick (cf. Figure 2.10(b)).

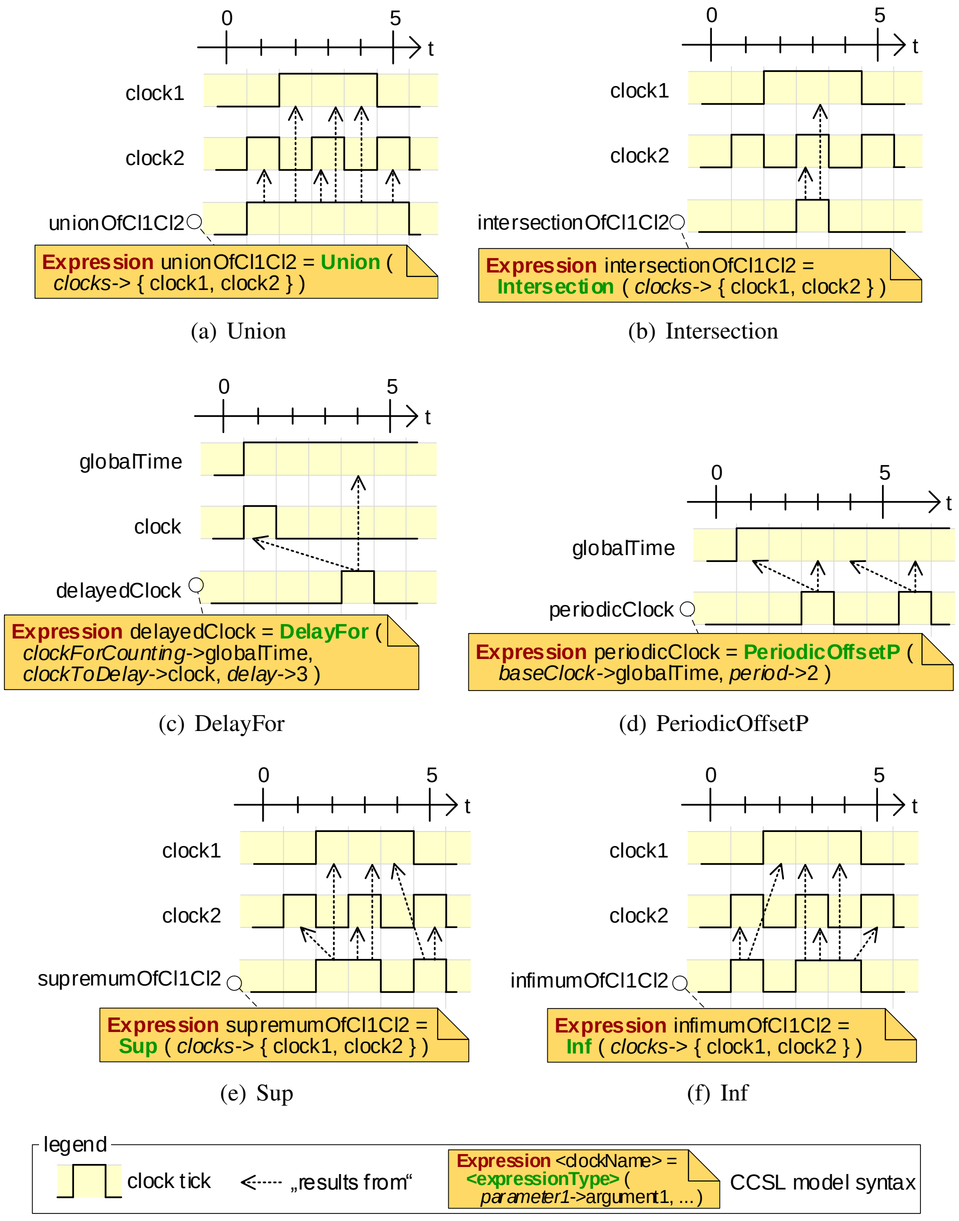
DelayFor (clockForCounting: Clock, clockToDelay: Clock, delay: Integer) This clock expression delays any tick of the clock clockToDelay by delay ticks of the clock clockForCounting. Note that we define in this thesis an always ticking clock globalTime as argument for clockForCounting so that the clock defined by this expression simply ticks delay ticks after clockToDelay (cf. Figure 2.10(c)).

PeriodicOffsetP (baseClock: Clock, period: Integer) The clock specified by this expression ticks any periodth tick of the baseClock. Note that we define in this thesis an always ticking clock globalTime as argument for baseClock so that the clock defined by this expression simply ticks any periodth tick (cf. Figure 2.10(d)).

Sup (clocks: Set(Clock))[1] The clock specified by this expression ticks with a clock in the parameter set that does not precede the other clocks, that is, it specifies the supremum of the particular instant sets (cf. Figure 2.10(e)).

Inf (clocks: Set(Clock))[1] The clock specified by this expression ticks with a clock in the parameter set that precedes the other clocks, that is, it specifies the infimum of the particular instant sets (cf. Figure 2.10(f)).

[1]For the sake of simplicity, we describe clock expressions and relations that have two clock parameters but can be arbitrarily chained as expressions with a clock set parameter.

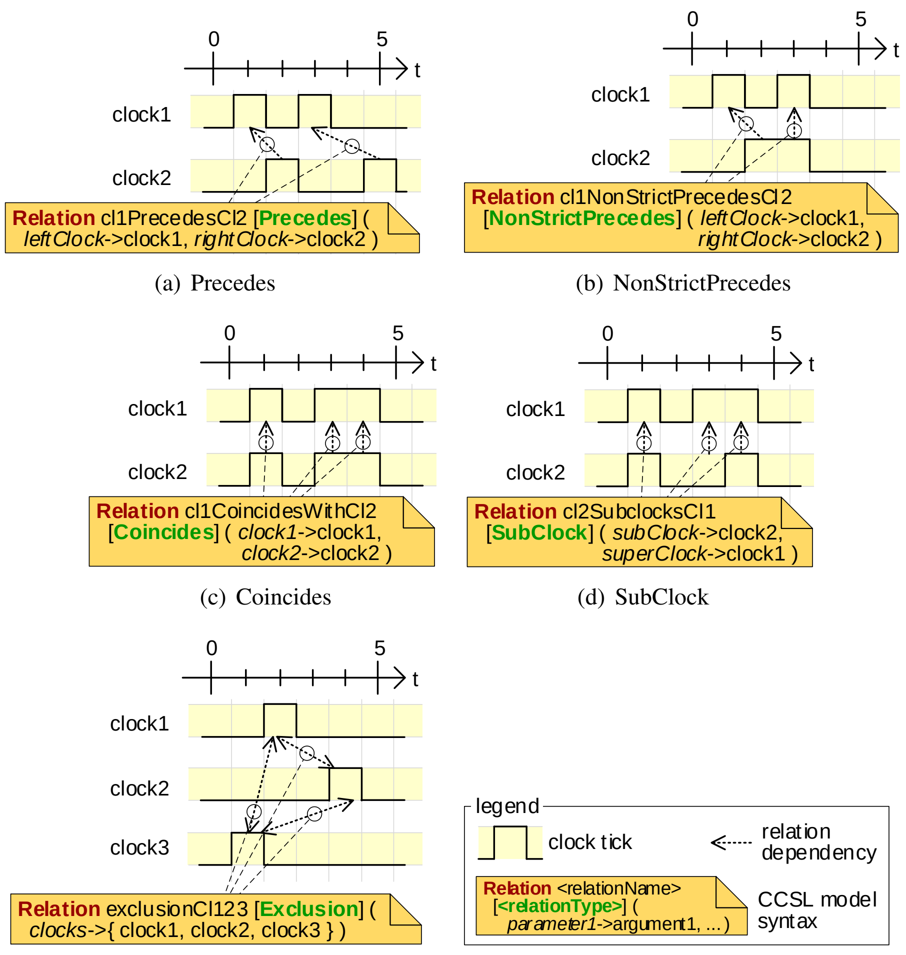

0
5
t
clock1
clock2
unionOfCl1Cl2
Expression unionOfCl1Cl2 = Union (clocks-> { clock1, clock2 })
(a) Union
intersectionOfCl1Cl2
Expression intersectionOfCl1Cl2 = Intersection (clocks-> { clock1, clock2 })
(b) Intersection
globalTime
clock
delayedClock
Expression delayedClock = DelayFor (clockForCounting->globalTime, clockToDelay->clock, delay->3)
(c) DelayFor
periodicClock
Expression periodicClock = PeriodicOffsetP (baseClock->globalTime, period->2)
(d) PeriodicOffsetP
supremumOfCl1Cl2
Expression supremumOfCl1Cl2 = Sup (clocks-> { clock1, clock2 })
(e) Sup
infimumOfCl1Cl2
Expression infimumOfCl1Cl2 = Inf (clocks-> { clock1, clock2 })
(f) Inf
legend
clock tick
„results from“
Expression <clockName> = <expressionType> (parameter1->argument1, ...)
CCSL model syntax

(g) Legend

Figure 2.10: Exemplary simulation runs of CCSL clock expressions

2.7.2.2 Clock Relations

Clock relations between clocks impose ordering constraints between the clocks' particular instant sets. Figure 2.11 depicts exemplary TIMESQUARE simulation runs of the clock relations that we use in Chapter 4 and Appendix B. We describe these relations in the following.



Relation cl1PrecedesCl2 [Precedes] (leftClock->clock1, rightClock->clock2)
Relation cl1NonStrictPrecedesCl2 [NonStrictPrecedes] (leftClock->clock1, rightClock->clock2)
Relation cl1CoincidesWithCl2 [Coincides] (clock1->clock1, clock2->clock2)
Relation cl2SubclocksCl1 [SubClock] (subClock->clock2, superClock->clock1)
Relation exclusionCl123 [Exclusion] (clocks->{ clock1, clock2, clock3 })
legend
clock tick
relation dependency
Relation <relationName> [<relationType>] (parameter1->argument1, ...)
CCSL model syntax

(a) Precedes (b) NonStrictPrecedes (c) Coincides (d) SubClock (e) Exclusion (f) Legend

Figure 2.11: Exemplary simulation runs of CCSL clock relations

Precedes (leftClock: Clock, rightClock: Clock) This relation constrains the k^{th} instant of leftClock to precede the k^{th} instant of rightClock $\forall k \in \mathbb{N}$. That is, the event represented by leftClock always occurs before rightClock (cf. Figure 2.11(a)).

NonStrictPrecedes (leftClock: Clock, rightClock: Clock) This non-strict version of the Precedes relation constrains the k^{th} instant of leftClock to coincide with or precede the k^{th} instant of rightClock $\forall k \in \mathbb{N}$. That is, the events can also occur simultaneously (cf. Figure 2.11(b)).

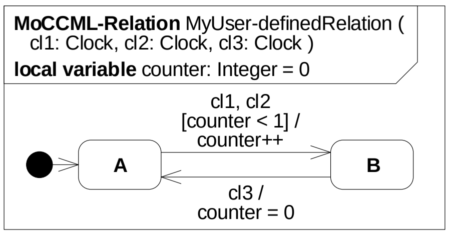
Coincides (clock1: Clock, clock2: Clock) This relation constrains all instants of clock1 and clock2 to coincide. That is, the events represented by clock1 and clock2 must occur simultaneously (cf. Figure 2.11(c)).

SubClock (subClock: Clock, superClock: Clock) This relation constrains all ticks of subClock to coincide with a tick of superClock but not vice versa. That is, subClock can only tick when superClock ticks, but it does not have to (cf. Figure 2.11(d)).

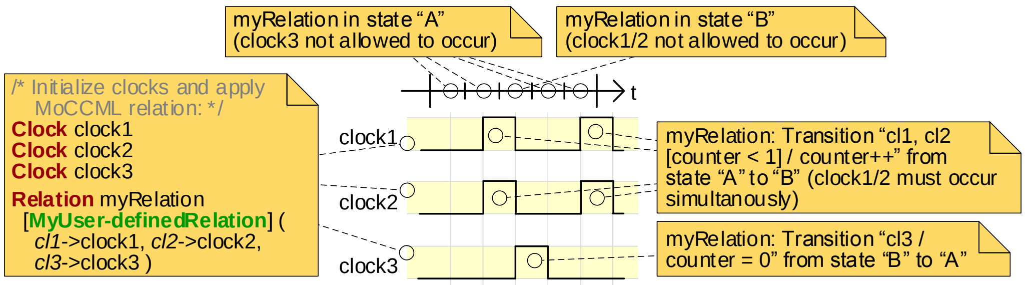
Exclusion (clocks: Set(Clock))[1] This relation constrains the parameter clocks to tick in mutual exclusion to each other. That is, the instants of any parameter clocks must not coincide (cf. Figure 2.11(e)).

2.7.3 User-defined Constraints

In order to enable a convenient specification of user-defined CCSL constraints based on automata, DeAntoni et al. [DDC⁺14] extend CCSL with the *Model of Concurrency and Communication Modeling Language* (MoCCML). MoCCML allows to specify user-defined clock relations by means of *MoCCML relations*, which can be simulated in TIMESQUARE and stored in user-defined model libraries. Figure 2.12 depicts an exemplary MoCCML relation whose concepts we use in this thesis and a corresponding exemplary TIMESQUARE simulation run, which we explain in the following.


MoCCML-Relation MyUser-definedRelation (
cl1: Clock, cl2: Clock, cl3: Clock)
local variable counter: Integer = 0
cl1, cl2
[counter < 1] /
counter++
A
B
cl3 /
counter = 0

(a) Exemplary MoCCML relation


myRelation in state "A"
(clock3 not allowed to occur)
myRelation in state "B"
(clock1/2 not allowed to occur)
t
/* Initialize clocks and apply
MoCCML relation: */
Clock clock1
Clock clock2
Clock clock3
Relation myRelation
[MyUser-definedRelation] (
cl1->clock1, cl2->clock2,
cl3->clock3)
clock1
clock2
clock3
myRelation: Transition "cl1, cl2
[counter < 1] / counter++" from
state "A" to "B" (clock1/2 must occur
simultanously)
myRelation: Transition "cl3 /
counter = 0" from state "B" to "A"

(b) Exemplary simulation run

Figure 2.12: Exemplary MoCCML relation and simulation run

Figure 2.12(a) depicts the MoCCML relation MyUser-definedRelation, which has three clock parameters and a local Integer variable counter that is initialized with zero. Figure 2.12(b) depicts an exemplary simulation run that initializes three clocks and applies the clock relation myRelation typed by the MoCCML relation on them. The automaton specifies two states A and B as well as each a transition from one state to the other. Such transitions allow to specify possibly coincident parameter clock triggers, guards, and effects. For example, the transition from A to B fires when both the clock parameters cl1 and cl2 (i.e., the clock arguments clock1 and clock2 in the simulation run) tick simultaneously and additionally the guard [counter < 1] holds. When it fires, the transition effect counter++ increments the counter. The transition from state B to A fires on the tick of the clock parameter cl3 (i.e., the clock clock3 in the simulation run) and sets the counter back to zero.

The triggers of the transitions leaving a state specify in which state which clocks are allowed to tick. For example, the clock parameter cl3 is not allowed to tick in state A, and the clock parameters cl1 and cl2 are not allowed to tick in B. Furthermore, cl1 and cl2 are not allowed to tick independently but must tick simultaneously in order to fire the transition from state A to B.

2.8 Specifying Modeling Language Semantics with GEMOC

According to Harel and Rumpe [HR04], a modeling language consists of an *abstract syntax* specifying the language concepts and their relations, a *semantic domain* describing the language meaning, and a *semantic mapping* relating the language concepts to the semantic domain elements. The GEMOC approach [CAL+13; LCD+15] enables to flexibly specify (potentially different) semantics for a modeling language following these definitions.

Figure 2.13 depicts an overview of the GEMOC approach. Specifically, the semantic domain is specified by means of a *Model of Concurrency and Communication (MoCC)*. This MoCC is defined by semantic constraints in the form of pre-defined CCSL constraints (cf. Section 2.7.2.1) as well as user-defined MoCCML constraints (cf. Section 2.7.3. The MoCC defines the concurrency, the synchronizations, and the possibly timed way the elements of a program interact during an execution. The semantic mapping is specified by the declaration of *Domain-Specific Events* (DSEs), which associate the abstract syntax and the MoCC. The DSEs are specified by means of the declarative *Event Constraint Language* (ECL) [DM12b]. ECL is an extension of the Object Constraint Language [OMG14a], augmented with the notion of DSEs as well as behavioral invariants that use CCSL and MoCCML constraints.

The approach is implemented in the modeling language workbench GEMOC Studio [BDV+16] for building and composing executable modeling languages. GEMOC Studio takes a language metamodel, an ECL mapping specification, and semantic constraints specified through a MoCC as inputs and automatically derives a modeling workbench with simulation and debugging facilities. Specifically, it derives a dedicated QVT-O model transformation [OMG16]. This model transformation takes an instance of the language metamodel as input and generates a dedicated CCSL model that parametrizes an execution engine based on TIMESQUARE. The model transformation maps the associated DSEs to CCSL clocks based on the ECL mapping specification and applies the semantic constraints from the behavioral invariants on these clocks.

For example, Listing 2.1 specifies ECL code that describes the causal/temporal behavior of asynchronous message events. For this purpose, it references the UML metaclass **MessageEvent** by setting the context to it. Within this context, two DSEs msgSendEvt and msgReceiveEvt are specified. Given a UML model containing any model element :MessageEvent, the derived QVT-O Transformation each generates two clocks corresponding to the DSEs as part of the CCSL model (cf. metamodel level M1 in Figure 2.13). The invariant asynchronousMessageSending specifies the semantic constraints between the DSEs. For this purpose, it references the CCSL clock relation Precedes (cf. Section 2.7.2.2) from the model library Pre-defined CCSL Constraints (cf. metamodel level M2 in Figure 2.13). Given a UML model containing any model element :MessageEvent, the derived QVT-O Transformation each generates a clock relation typed by Precedes using the two clocks as arguments. During the execution of the resulting CCSL model, TIMESQUARE computes a simulation trace similar to the one depicted

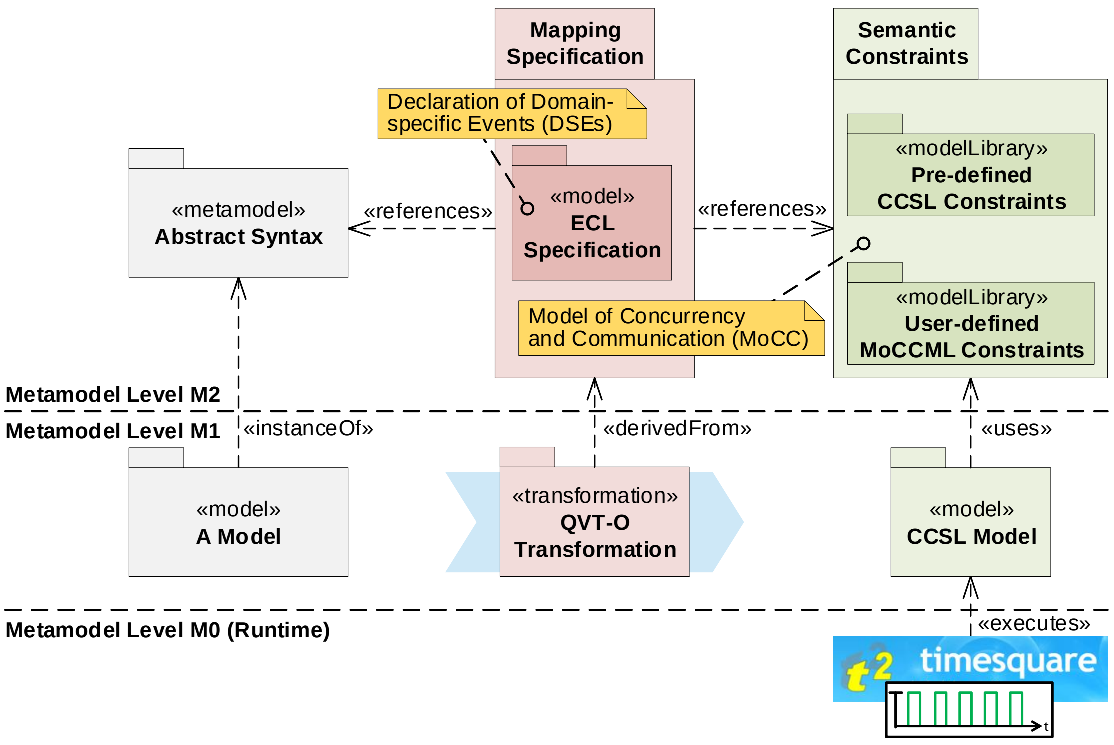

Mapping Specification
Declaration of Domain-specific Events (DSEs)
«model» ECL Specification
Model of Concurrency and Communication (MoCC)
Semantic Constraints
«modelLibrary» Pre-defined CCSL Constraints
«modelLibrary» User-defined MoCCML Constraints
«metamodel» Abstract Syntax
«references»
«references»
Metamodel Level M2
Metamodel Level M1
«instanceOf»
«derivedFrom»
«uses»
«model» A Model
«transformation» QVT-O Transformation
«model» CCSL Model
Metamodel Level M0 (Runtime)
«executes»
t2 timesquare

Figure 2.13: Specifying semantics for executable modeling languages with GEMOC

in Figure 2.11(a) where the msgSendEvt clock ticks always precede the msgReceiveEvt clock ticks. Likewise, user-defined MoCCML relations or clock expressions for the definitions of auxiliary clocks can be referenced from ECL.

Listing 2.1: ECL code specifying the causal/temporal behavior of asynchronous message events

```
/* referencing a UML metaclass */
context UML::MessageEvent
 /* declaration of two DSEs */
 def: msgSendEvt:Event
 def: msgReceiveEvt:Event

 /* specification of semantic constraints on the DSEs */
 inv asynchronousMessageSending:
  /* referencing the predefined CCSL clock relation Precedes */
  Relation Precedes( leftClock→self.msgSendEvt, rightClock→self.msgReceiveEvt )
```

Note that GEMOC does not consider a runtime metamodel for the CCSL model execution in TIMESQUARE as SCENARIOTOOLS MSD Play-out (cf. Section 2.4.1) or the Dynamic Meta Modeling approach [EHHS00; Hau05; Sol13]. This implies that if such modeling concepts that explicitly describe the dynamic runtime behavior [LBTA11] are required in the resulting CCSL models, dedicated abstract syntax elements have to be provided in order to specify the corresponding DSEs and their semantic constraints.

3

Integrated Systems Engineering and Software Requirements Engineering

In this chapter, we present a technique for the transition from MBSE with CONSENS to model-based SwRE with MSDs [*HBM$^+$16; *HBM$^+$15; *HBM$^+$17; *Ber15] (cf. Figure 3.1). This technique applies automatisms and systematization to make the transition more effective and efficient. That is first, we make the transition more effective through reducing the likelihood to introduce defects into the software requirements. That is second, we make the transition more efficient through reducing the effort for the Software Requirements Engineers. Similar techniques for the transition from system models to discipline-specific models support the transition to software design but not requirements models [Rie15; Thr10; OMG12; CLP11], do not support behavioral models [BHH$^+$14; Thr10], and/or do not provide automatisms [BHH$^+$14; AGD$^+$12].

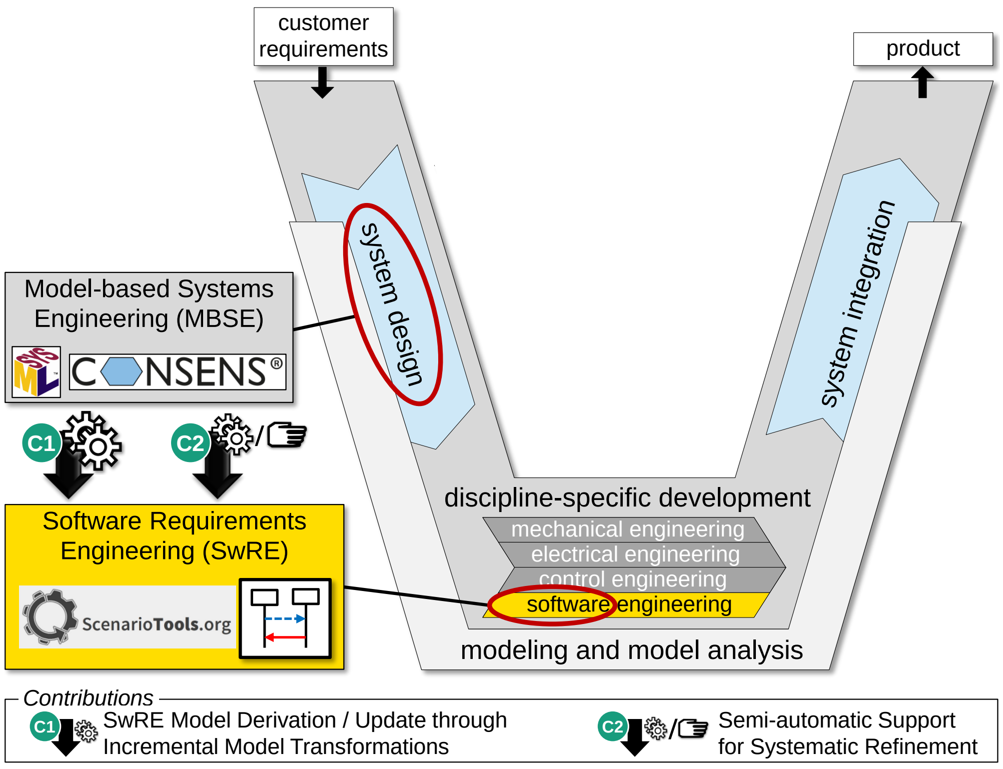

customer requirements
product
system design
system integration
Model-based Systems Engineering (MBSE)
CONSENS®
C1
C2
Software Requirements Engineering (SwRE)
ScenarioTools.org
discipline-specific development
mechanical engineering
electrical engineering
control engineering
software engineering
modeling and model analysis
Contributions
SwRE Model Derivation / Update through Incremental Model Transformations
Semi-automatic Support for Systematic Refinement

Figure 3.1: The transition from model-based systems engineering to software requirements engineering (based on [*HBM$^+$17; VDI04])

Figure 3.1 sketches the main contributions of our transition technique:

- We automate steps of the transition where possible by means of incremental and traceability-establishing model transformations (C1) (cf. Sections 3.4, 3.7, and 3.8.2). This reduces error-prone and time-consuming manual tasks through automatically deriving initial and updating existing MSD specifications from CONSENS system models.
- Not all transition steps are automatable due to informal information in the system models. We provide a systematic refinement approach (C2) that encompasses semi-automatic tool support complemented by a set of informal guidelines (cf. Sections 3.5 and 3.8.3 and Appendix A.2.2.2). This further supports the Software Requirements Engineers during the manual refinement of automatically derived or updated MSD specifications.

Furthermore, we introduce the following supporting contributions:

- We present a SysML-based CONSENS language variant by means of a SysML profile called SYSML4CONSENS [*KDHM13; IKDN13] and transfer the relevance annotations of [Rie15] to a further SysML profile [*Ber15] (cf. Section 3.9.1.1). This abstract syntax language specification based on SysML [OMG17a] enables us conceiving automatisms to automatically process SYSML4CONSENS models as part of our transition technique. Furthermore, the transfer of the conventional CONSENS modeling language to SysML provides Systems Engineers a mature tool support for CONSENS by means of the widespread UML/SysML modeling tools.
- The other way round, we introduce SysML concepts into the conventional CONSENS modeling language (cf. Section 3.1). This includes language means to thoroughly specify system architecture interfaces as well as to establish referential traceability from the behavioral to the structural partial models. Furthermore, we introduce a new variant of the partial model *Behavior – Sequences* to specify discipline-spanning, sequentially ordered actions to specify existential interaction behavior for refining *Application Scenarios* or to define test cases. We consolidate these aspects and several former publications to a CONSENS language specification, for which we refer to [*Wör14].
- We extend the MSD modeling language to reflect hierarchical software component architectures encompassing components, ports, connectors, and hierarchies [*HM13; *BBG+13] (cf. Section 3.2). This aligns the MSD modeling language more closely to CONSENS as well as other modeling languages for software-intensive systems and thereby enables a smooth transition from CONSENS system models to MSD specifications.
- We identify aspects of the CONSENS system models that are relevant to SwRE and document these artifact dependencies in a process description including role responsibilities (cf. Section 3.3). This process description guides the Systems Engineers and the Software Requirements Engineers to perform their particular steps in a systematic way, where the task assignment to roles clarifies the distribution of their respective responsibilities.
- We present a model transformation approach that borrows concepts of the imperative model transformation approach QVT-O [OMG16] and the traceability-establishing model transformation approach Triple Graph Grammars (TGGs) [Sch95] (cf. Section 3.8.2.1). On the one hand, the imperative logic of QVT-O enables to determine and process the SwRE-relevant information scattered across several CONSENS partial models in a compact and efficient way. On the other hand, the traceability-establishing concepts of TGGs enable incremental updates through transformation traceability as well as the establishment of lifecycle traceability for the purpose of model management activities.

- We present a novel terminology for model-based traceability extending the generic terminology from traceability literature (cf. Section 2.1.1). This enables us precisely distinguishing between the different kinds of traceability that we apply throughout this chapter.

3.1 Extensions to the CONSENS Specification Technique

Every model has a certain purpose, inter alia [Sta73]. In the automatic part of our transition technique presented in this chapter, we exploit CONSENS system models with the purpose of automatically deriving MSD specifications. This purpose requires a more detailed and formalized system model than an initial, informal system model for the purpose of system requirements elicitation or the clarification of a coarse-grained system architecture, for which CONSENS is typically applied for [Tsc16; TDBG15]. In terms of the MBSE concept classification of Tschirner et al. [Tsc16; TDBG15], the system model should be specified for our modeling purpose according to the concept of “Mechatronic Systems Modeling”. This MBSE concept requires more modeling effort than concepts with lightweight modeling purposes like interdisciplinary communication.

Our modeling purpose imposes certain preconditions on the CONSENS system models. First, we expect that the behavioral partial models reference the underlying system architecture in terms of the contents that the particular interfaces of environment/system elements allow to produce or consume. This is due to the fact that executable MSD specifications require referential traceability from the actual MSDs to their structural basis, and consequently we need this information also in CONSENS system models for the automatic parts of our transition techniques. Second, we extend and rigorously define the partial model *Behavior – Sequences* in order to reflect sequential behavior similar to MSDs. This is due to the fact that up to now this partial model is introduced in a very restricted way [Ana15; AGD$^+$12] or is even only mentioned [DDGI14; GFDK09]. However, in industrial projects there is often the need for a discipline-spanning description of partially ordered interactions between few system and environment elements [*Wör14]. Furthermore, an adequate partial model for the description of such discipline-spanning interactions would be a well-suited basis for the transition to MSD specifications.

For both the rigorous specification of the behavioral descriptions on the basis of system architectures and the definition of *Behavior – Sequences*, we apply modeling means based on SysML. This is because SysML provides adequate modeling language means for these purposes and is a well-established, de-facto standard language for systems modeling. Consequently, our actual implementation bases on a SysML profile called SYSML4CONSENS (cf. Section 3.9.1.1). We present the SysML modeling means for a rigorous specification of CONSENS system models for our purposes in this section and summarize a complete list of the preconditions for CONSENS models in Section 3.8.1.

In Section 3.1.1, we introduce port specifications as basis for the referential traceability from the behavioral partial models to the architecture. We introduce our version of the partial model *Behavior – Sequences* and its referential traceability to the port specifications in Section 3.1.2. Finally, we explain how we specify the partial model *Behavior – States* with language concepts of UML/SysML State Machines [OMG17b] including referential traceability to the port specifications in Section 3.1.3.

3.1.1 Port Specifications

We exemplarily introduce interfaces for the particular environment/system elements as structural basis for the behavioral partial models in this section and refer to [*Wör14] for a full language definition. Figure 3.2 depicts excerpts of the partial models relevant in this and the subsequent section. In contrast to the interdisciplinary view on the system model in Figure 2.2 in Section 2.2, we focus in Figure 3.2 on the *logical view* of the *Environment* and *Active Structure*. The logical view considers only logical software components and information flows, which are realized by the disciplines of software and control engineering (cf. [EMV12; Kru95])[1].

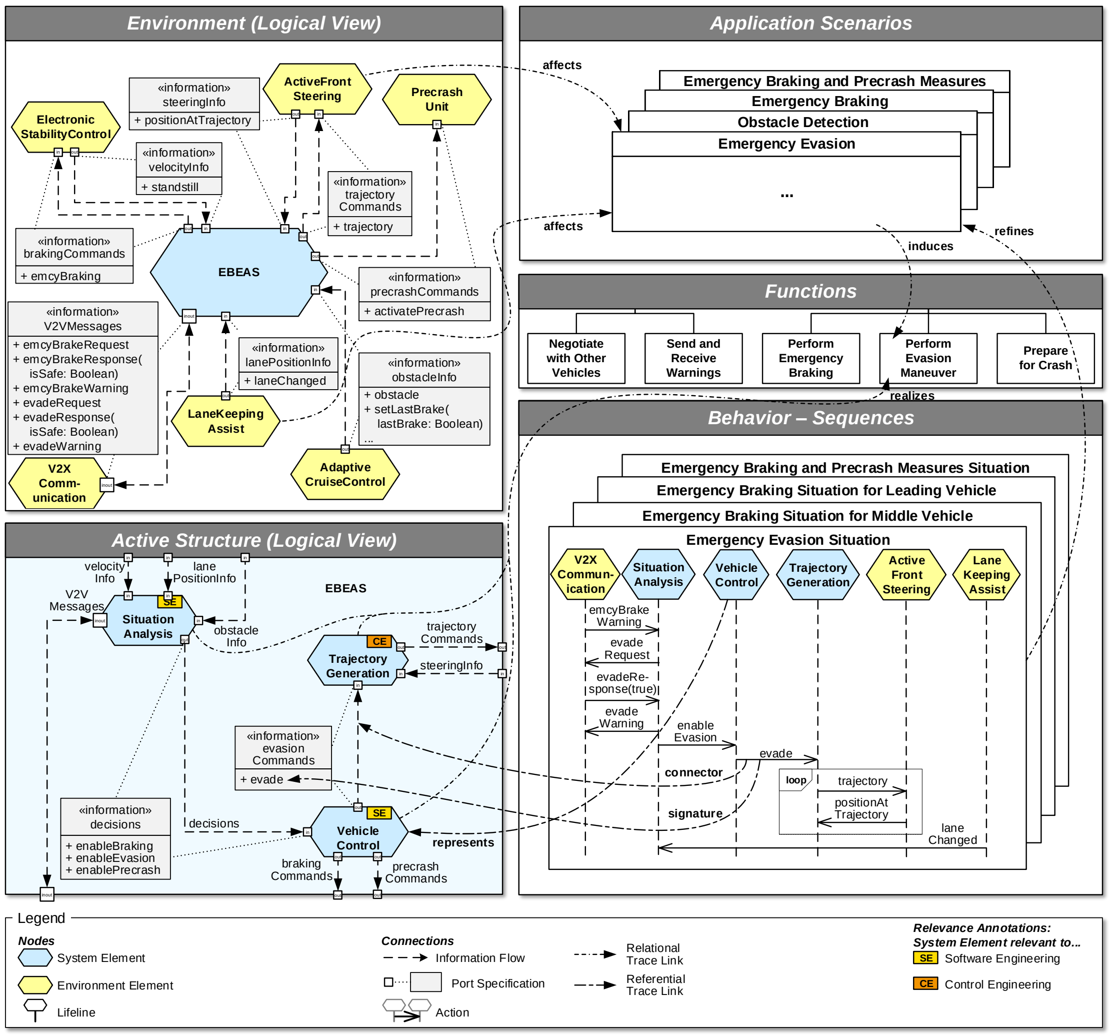


Figure 3.2: Logical view on the partial models *Environment* and *Active Structure* with port specifications as well as excerpt of the partial model *Behavior – Sequences* for the EBEAS

[1]The term logical view also exists in the MBSE community as part of the Requirements/Functional/Logical/Physical (RFLP) view/viewpoint approach (e.g., [EGZ12]). All system elements responsible for realizing functions—without a restriction to a certain discipline—are considered logical elements to distinguish them from requirements, functions, and physical elements in this terminology.

Similarly to SysML interfaces, we apply *port specifications* to specify the messages and signals sent via the particular information flows (as well as energy and material transferred via energy and material flows, respectively). This facilitates the reuse of interfaces, for example, across multiple hierarchy levels. For example, the port specification trajectoryCommands is an *information flow specification*. It defines that the *information flow item* trajectory can be sent via the corresponding information flows from the port of the system element TrajectoryGeneration via the port of the system EBEAS to the port of the environment element ActiveFrontSteering (cf. top and bottom left of Figure 3.2). All port specifications in the logical view of Figure 3.2 define that the corresponding flows represent information flows due the stereotype «information», where the kind of information can be of discrete or continuous nature.

3.1.2 *Behavior – Sequences*

We exemplarily introduce our new variant of the partial model *Behavior – Sequences* in this section and refer to [*Wör14] for a full language definition, which borrows the main concepts of UML/SysML Interactions. In contrast to MSDs, the partial model *Behavior – Sequences* focuses on discipline-spanning actions, is semi-formal, and provides no means for specifying modalities, execution kinds, or real-time requirements. Furthermore, we assume in this thesis that *Behavior – Sequences* are used to specify exemplary behavior (e.g., for refining one or two positive cases of an *Application Scenario*). Thus, in contrast to MSD specifications describing the universal behavior partitioned across multiple MSD use cases, *Behavior – Sequences* exemplarily specify excerpts of this behavior, that is, existential behavior in terms of MSDs (cf. Section 2.4.2.3).

The bottom right of Figure 3.2 depicts the *Behavior – Sequence* Emergency Evasion Situation and indicates three other ones. Similar to other scenario-based modeling languages, *Behavior – Sequences* specify lifelines and partially ordered interactions between them. The traceability to the remainder of the CONSENS system model is established by a combination of referential and relational trace links. A lifeline represents a system element or environment element in the *Active Structure* and the *Environment*, respectively (e.g., the lifeline Vehicle Control represents the equally named system element). An interaction between lifelines is specified by means of an *action*. Such an action represents a discrete message, a continuous signal, an energetic actuation, or a mechanical movement depending on the content of a port specification it refers to. This reference is specified by means of the referential trace link signature. For example, the action evade has the equally named signature of the port specification evasionCommands and represents a discrete message. Furthermore, each *Behavior – Sequence* has a relational trace link of the type refines to an *Application Scenario* (e.g., the *Behavior – Sequence* Emergency Evasion Situation refines the *Application Scenario* Emergency Evasion). Finally, combined fragments as known from the UML [OMG17b] can be applied within *Behavior – Sequences*. For example, the continuous interactions between the lifelines Trajectory Generation and ActiveFrontSteering are embedded into a loop fragment, representing the steady signal interchange between the corresponding system and environment elements until the message laneChanged occurs.

3.1.3 *Behavior – States*

We assume in this thesis, that the partial model *Behavior – States* specifies the state-based behavior of the overall system (i.e., EBEAS) but not, as in [Rie15], for its particular subsystems (e.g., SituationAnalysis). The latter specification step is subject to the discipline-specific design as we experienced in industrial projects. Thus, *Behavior – States* specify requirements on the particular state-based subsystems of the SUD, whose individual and interconnected behaviors have to yield the overall system behavior. The Software Requirements Engineer defines these semi-formal requirements more concretely by means of MSDs through adding system-internal messages between the particular software components. Hence, the Systems Engineer has to specify the *Behavior – States* in terms of an input/output (I/O) automaton [TL89] for the overall SUD, where the I/O automaton has no internal actions.

We use the specification means of UML/SysML Behavior StateMachines [OMG17b] for a thorough specification of the partial model *Behavior – States*. That is, we distinguish between transition triggers occurring at the in-ports of the SUD, and transition effects or state operations performing actions on the environment elements. The referential traceability from the *Behavior – States* to the *Environment* is established through the corresponding UML/SysML referential trace links.

For example, Figure 3.3 shows on the left-hand side an excerpt of the logical view of the partial model *Environment* with port specifications and on the right-hand side an excerpt of the partial model *Behavior – States*. A message trigger as `laneChanged` is associated to a call event. This call event has a referential trace link operation to an equally named operation of a port specification, that is, lanePositionInfo. Furthermore, such a message trigger has a referential trace link port to the port of the SUD that the corresponding event occurs at.

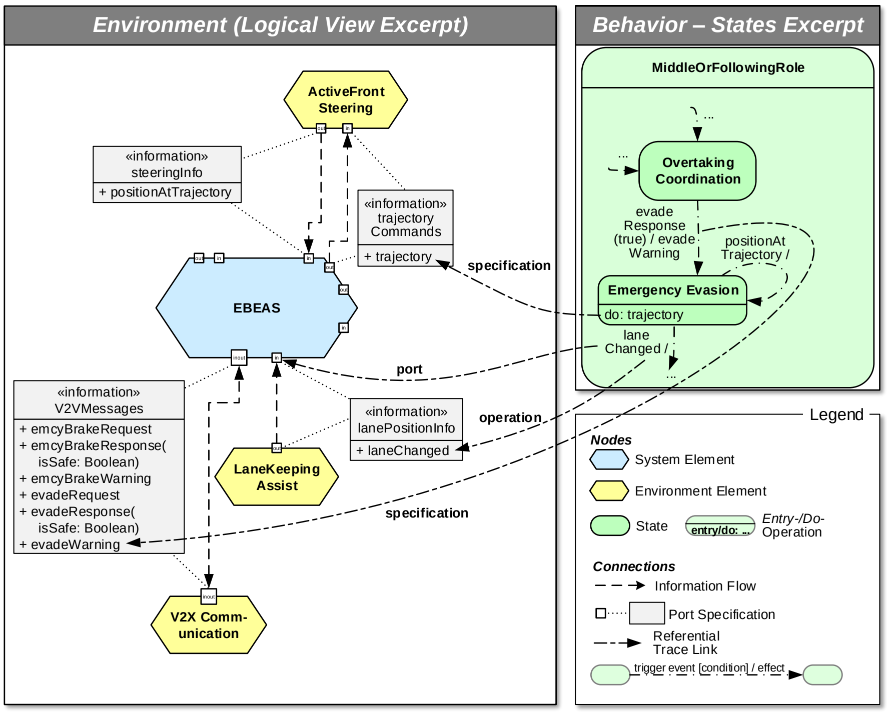


Figure 3.3: Logical view excerpt of the partial model *Environment* with port specifications as well as excerpt of the partial model *Behavior – States* for the EBEAS

Actions performed by the SUD on the environment elements are either specified by means of transition effects or the entry-/do-/exit-operations of a state. Both a transition effect and a state operation are specified by means of a referential trace link specification from a transition effect (e.g., `evadeWarning`) or a state operation (e.g., the do-operation trajectory) to an equally named information flow item of a port specification (i.e., the corresponding information flow items in the port specifications V2VMessages and trajectoryCommands, respectively).

Furthermore, we use the concepts of hierarchical state machines as originally introduced as Statecharts by Harel [Har87]. That is, we distinguish between simple states that are atomic and composite states encompassing sub states and/or regions. For example, the state MiddleOrFollowingRole is a composite state because it contains further sub states. In contrast, these sub states are not further decomposed and hence are simple states. Figure A.12 in Appendix A.2.1 depicts the complete *Behavior – States* for the EBEAS. This partial model contains at the top-most level the composite state EBEAS System Behavior that is divided into the two regions Main Behavior and Critical Points Notification, which are concurrently active at the same time.

Finally, we use transition guards that have to evaluate to the Boolean value `true` so that a transition enabled by an event can fire. The bottom right in Figure 2.2 in Section 2.2.7 depicts a larger *Behavior – States* excerpt. The sub state machine MiddleOrFollowingRole gets active when the trigger event emcyBrakeWarning occurs. If, for example, this warning occurs at a point in time such that an emergency braking is unsafe because the last point to brake is exceeded (first part `lastBrake` of the guard condition of the transition between the states EmergencyBrakeWarningReceived and OvertakingCoordination) but the last point to evade is not exceeded (second part `!lastEvade` of the guard condition), the overtaking coordination takes place. If this negotiation yields that an emergency evasion is safe for the overtaking vehicle (trigger event evadeResponse(true)), the state EmergencyEvasion gets active. The trajectory is continuously updated based on the current vehicle position at the trajectory until the trigger event laneChanged occurs.

3.2 Component-based MSD Specifications

In this section, we present *component-based MSD specifications* [*HM13; *BBG+13]. These extend conventional MSD specifications (cf. Section 2.4) with hierarchical component architectures encompassing ports, interfaces and directed connectors to foster component and interface reusability as well as encapsulation. Like conventional MSDs, the modeling language for component-based MSD specifications bases on a subset of modeling constructs of the UML [OMG17b] and extends these constructs by means of the Modal profile (cf. Section 2.5.1). Component-based MSD specifications are the target models of our technique for the transition from MBSE with CONSENS to SwRE with MSDs described in this chapter, and we use them as modeling basis for platform-specific timing analyses in Chapter 4.

We partition component-based MSD specifications into three different view types [GBB12] that reflect different stakeholder-specific viewpoints [ISO11] and adhere to different UML [OMG17b] metamodel parts. "A view type defines rules according to which views of the respective type are created", and "a view can [...] be considered an instance of a view type" [GBB12]. In the following, we hence use the term view type for general aspects common to all component-based MSD specifications and the term view for concrete models or diagrams. Figure 3.4 depicts a conceptual overview of the ingredients of component-based MSD specifications and their assignments to the view types. Figure 3.5 depicts an example component-based MSD specification based on the example conventional MSD specification depicted in Figure 2.4 of Section 2.4.

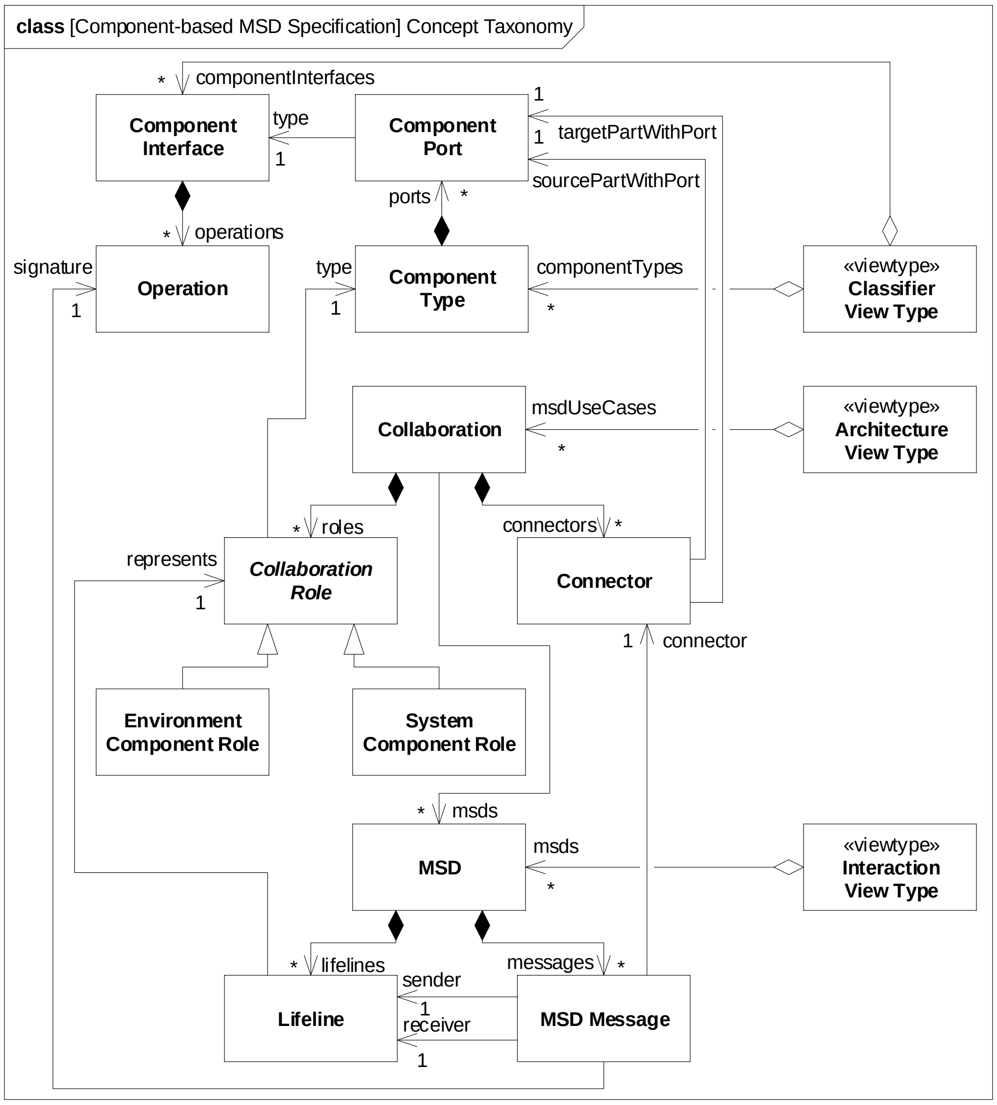


Figure 3.4: Concept taxonomy for component-based MSD specifications

The *classifier view type* provides reusable types (cf. topmost part of Figure 3.4). First, it contains interfaces defining operations. Second, this view type encompasses software component types owning ports that are typed by the interfaces.

For example, the Classifier View in the top of Figure 3.5 encompasses interfaces and component types for the EBEAS specification. The topmost class diagram depicts excerpts of the package Obstacle Detection Interfaces, which contains the interface **Decisions** defining the operation `enableBraking()` (UML classes with the stereotype «interface»). The class diagram below depicts excerpts of the package Obstacle Detection Types, which contains the software component types **SituationAnalysis** and **VehicleControl** (UML component symbols). The former one owns a port that uses the interface **Decisions** as required interface, whereas the latter one owns a port that uses the interface as provided interface.

The *architecture view type* contains the architectural part of MSD use cases (cf. middle part of Figure 3.4). Like in conventional MSD specifications, we use UML collaborations to specify the

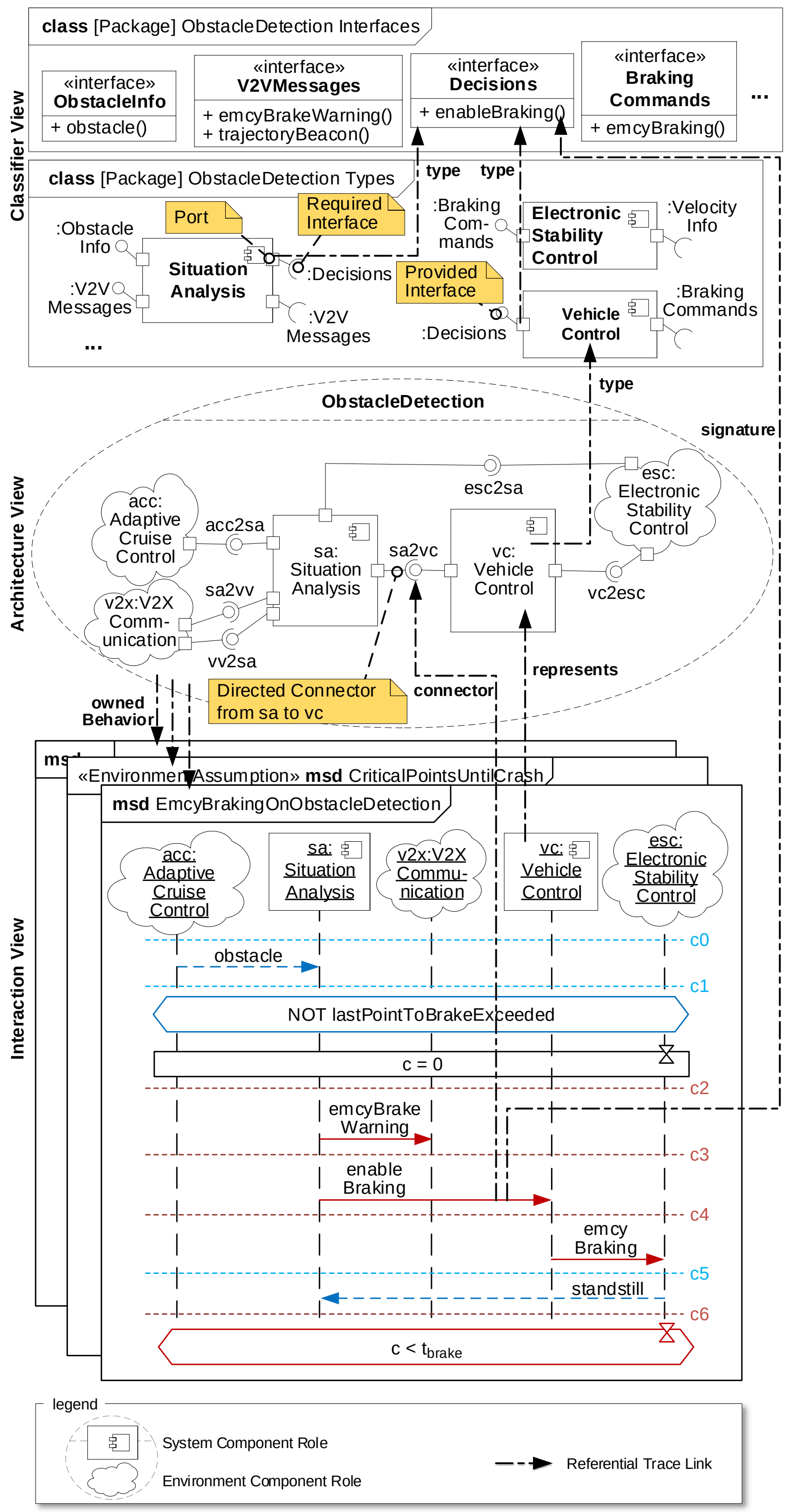


Figure 3.5: Example of a component-based MSD specification

structural basis for the actual MSDs. For component-based MSD specifications, this structural basis is specified by means of software component architectures. The collaborations define the participants of an MSD use case by means of roles classified by the software component types in the classifier view type. We distinguish between software components of the SUD (called *system component roles* in the following) and *environment component roles*. Beyond conventional MSD specifications, the collaborations contain directed communication links between these roles specified by connectors that interconnect the ports of the software component roles.

For example, the Architecture View in the middle of Figure 3.5 depicts the UML collaboration for the MSD use case *ObstacleDetection*. This collaboration specifies the software component architecture for the MSD use case. The system component role sa: SituationAnalysis can send messages to the system component role vc: VehicleControl via the directed connector sa2vc, but not vice versa. The communication direction is specified through the usage kind of the port interfaces of the corresponding component types in the classifier view type. That is, the port :Decisions of the software component type **VehicleControl** uses its classifying interface as provided interface and hence can only receive but not send the messages defined by the interface; this is vice versa for the corresponding port of **SituationAnalysis**. The remaining roles in the architecture are environment component roles, which are visualized by means of cloud symbols.

Each use case contains a set of MSDs, which are specified in the *interaction view type* (cf. bottommost part of Figure 3.4). Each lifeline in the MSD represents a role within the collaboration of a specific use case. The messages correspond to the operations of the interfaces in the classifier view type and are associated to a directed connector in the architecture view type.

For example, the Interaction View in the bottom of Figure 3.5 depicts an MSD EmcyBraking-OnObstacleDetection and indicates further MSDs. The contents are similar to the example of conventional MSD specifications depicted in Figure 2.4. However, component-based lifelines represent system component roles or environment component roles. Furthermore, a message is sent via a connector and has a signature referencing an operation defined in an interface as specified through referential trace links. For example, the message `enableBraking` is sent from the lifeline representing the system component role sa: SituationAnalysis to the lifeline representing the system component role vc: VehicleControl via the connector sa2vc. Furthermore, its signature references the equally named operation as part of the interface **Decisions**.

The static semantics of component-based MSD specifications are described by means of OCL constraints [OMG14a] in [*BBG$^+$13, Appendix B]. They specify, for example, that MSD messages may only be sent in the direction that the connector and the interfaces allow.

3.3 Process Description

This section describes the overall process for applying our technique for the transition from MBSE with CONSENS to SwRE with MSDs. This process description has the purpose to clarify which engineering roles conduct which steps and produce which work products including specifying interactions between the roles. Particularly, the process description specifies which artifacts of CONSENS system models are input to which step of our transition technique, and which parts of MSD specifications the respective steps of our transition technique produce.

Figure 3.6 depicts our idealized overall process for the application of our transition technique. The process is specified by means of a BPMN *collaboration* describing the interplay between MBSE and SwRE. The main contribution of this chapter is emphasized with gray tasks and artifacts. The steps that we could automate are visualized by means of BPMN *service tasks* (cogwheel in the upper left corner of the task). We visualize manual steps by means of BPMN *manual tasks* (hand in the upper left corner of the task) and tool-supported steps by means of a BPMN *user tasks* (person in the upper left corner of the task). Work products are specified as BPMN *data objects* (document icons), and persistent models that are subject to update and retrieval operations are specified as BPMN *data stores* (database icon). Multiple equally named data store occurrences represent a BPMN *data store reference*, that is, a reference to the same data store.

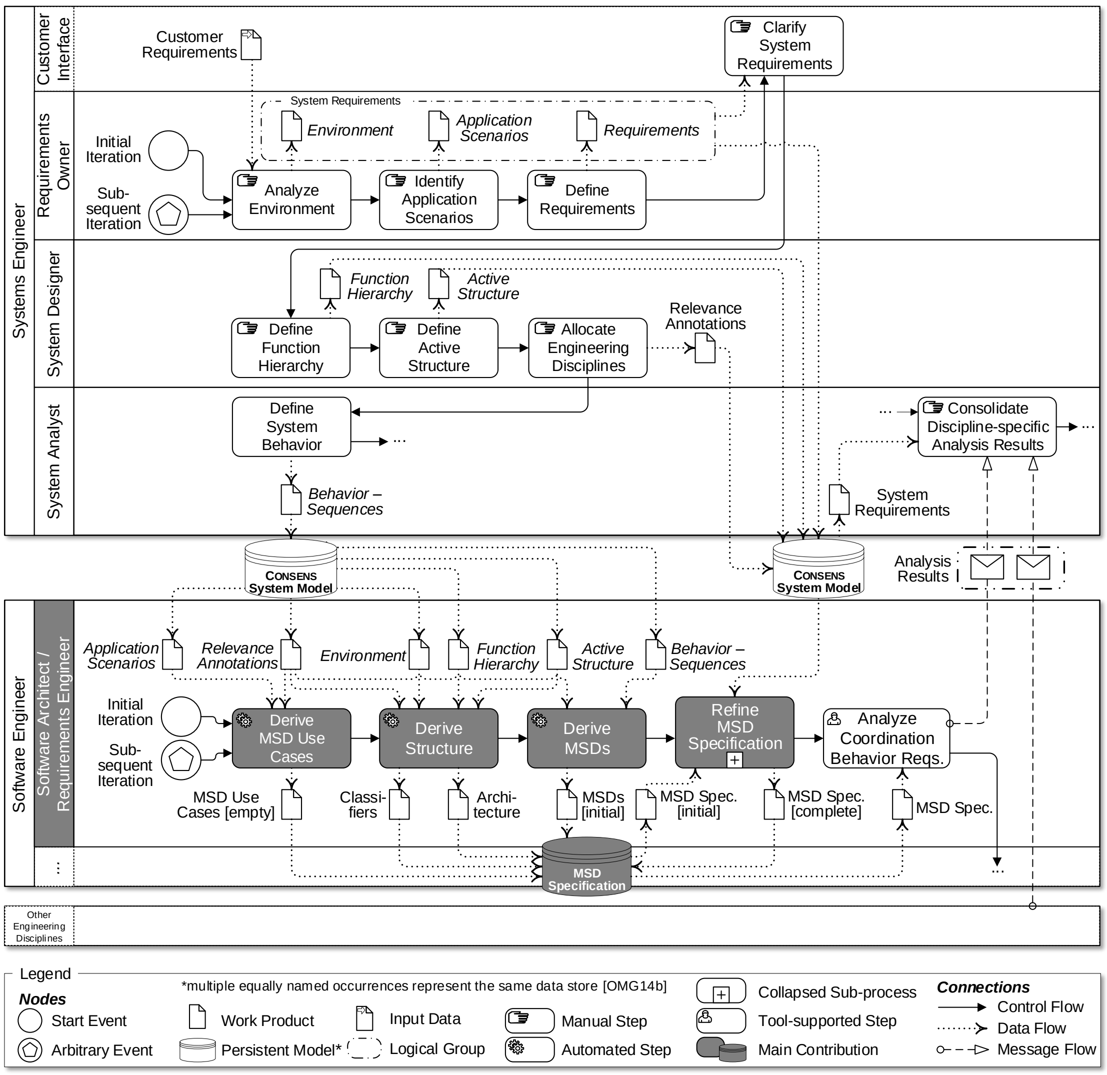


Figure 3.6: Idealized overall process excerpt for the integration of MBSE with CONSENS and SwRE with MSDs (based on [*HBM+16; *HBM+15])

In the following, we sketch the particular steps that are conducted by the different engineering roles. For this purpose, we introduce the systems engineering and software engineering roles that are relevant to our transition technique and are specified in Figure 3.6 as BPMN *pools* and *lanes*. Whereas the coarse-grained roles *Systems Engineer* and *Software Engineer* are specified by means of BPMN *pools*, the fine-grained, specialized roles introduced in the following are specified by means of BPMN *lanes*.

Within the systems engineering domain, we stick to the CONSENS specification method as presented in Figure 2.1. However, we introduce different systems engineering roles and partially additional process steps to this method in order to clarify the collaboration with the software engineering discipline. These roles base on Kaiser [Kai14], who defines which systems engineering roles according to Sheard [She96] participate in MBSE and which aspects of system models are of interest for them.

The *Customer Interface* as part of the Systems Engineer is the "face to the customer" and responsible for eliciting the right customer requirements. One of its tasks is to elicit and collect the Customer Requirements (cf. BPMN *data input* in Figure 3.6) from the external stakeholders and to analyze the environment jointly with the Requirements Owners (whose tasks are described in the following). In this task, the Customer Interface represents the market issues (e.g., external stakeholders like customers). Another task of the Customer Interface is the clarification of the system requirements, which we introduce in our transition process as explicit process step. We regard the CONSENS partial models *Environment*, *Application Scenarios*, and *Requirements* altogether as System Requirements (cf. BPMN *group* in Figure 3.6), which propose a technically oriented solution of the SUD functionality as demanded by the Customer Requirements. These system requirements have to be clarified by the Customer Interface with the external stakeholders before the actual system design starts.

The *Requirements Owners* start with the definition of technical requirements on system level (system requirements) based on given customer requirements. In the task of analyzing the *Environment* jointly with the Customer Interface, the Requirements Owners represent all technical issues and are responsible for the correct usage of the method. Furthermore, the Requirements Owners are responsible for the identification of the *Application Scenarios*, but is supported by other stakeholders, for example, from manufacturing and especially validation and verification. They have to consolidate all *Application Scenarios* and evaluate their significance. Finally, they are responsible for the definition and management of the *Requirements*.

Based on the system requirements, the *System Designer* creates the high-level, discipline-spanning system architecture. This usually includes the definition of system functions, the selection of adequate top-level components, and their allocation to engineering disciplines. In contrast to [She96], we define in this thesis that the System Designers instead of the Requirements Owners are responsible for specifying the function hierarchy. This is due to the fact that the System Designers know the technologies available at the company or at suppliers for realizing the functions. Furthermore, they conduct the definition of the function hierarchy together with the Requirements Owners as proposed by [She96].

The *System Analysts* ensure that the SUD meets the system requirements through providing an optimal starting point for the discipline-specific issues. Beyond the specification of the behavioral CONSENS partial models, the System Analyst continuously consolidates discipline-specific analysis results and work products to ensure a reasonable system design. We introduce this task in our transition process since it is not part of the CONSENS specification method (cf. Figure 2.1).

Within the software engineering discipline, we focus in this chapter on the roles *Software Requirements Engineer* and *Software Architect*. The former role is responsible to document and validate the requirements on the coordination behavior and negotiate them with the System Analyst. The latter role is responsible for the specification of the software architectures as part of component-based MSD specifications (cf. Section 3.2).

Our transition technique semi-automatically supports these roles in conceiving MSD specifications based on CONSENS system models. The technique is divided into a fully automatic model transformation part that derives initial or updates existing MSD specifications (cf. *service tasks* in Figure 3.6) and a manual part, in which the MSD specifications are systematically refined (cf. *collapsed sub-process* Refine MSD Specification in Figure 3.6). We provide an overview of the model transformation rules in Section 3.4 and describe them more formally in Section 3.8.2.2. We support the manual refinement by a set of guidelines and further semi-automatic means, which we describe in Section 3.5. The tool-supported step Analyze Coordination Behavior Requirements is conducted by means of the MSD analysis techniques (Real-time) Play-out and (timed) synthesis (cf. Section 2.4.3).

Note that despite the task assignment to roles might indicate a strict separation of responsibilities, we encourage the involvement of all disciplines in the specification of the system requirements and design since issues resulting from a strict separation of responsibilities can cause major problems [Boe00]. Furthermore, we assume that the Systems Engineer has a "T-shaped" competency profile [Gue91] with deep knowledge in one discipline as well as basic knowledge in the remaining disciplines, as also proposed by Pyster et al. [PAA+15]. This enables the Systems Engineer to understand and respect the needs of all discipline experts like the Software Engineer.

In contrast to [Rie15], our transition technique is intentionally unidirectional. Based on our experiences from multidisciplinary discussions, we follow the MBSE idea envisioning the system model to orchestrate the discipline-specific models. Thus, changes in a discipline-specific model shall not influence the system model. This implies that the Software Requirements Engineers must not directly manipulate the automatically derived parts of an MSD specification, which could introduce inconsistencies between system model and MSD specification. Instead, they have to contact the Systems Engineers in order to trigger changes on the CONSENS system model that influence the automatically derived part of an MSD specification.

3.4 Model Transformation Rules Overview

This section presents the functional principle of our model transformation rules in a coarse-grained way. Section 3.8.2.2 concretizes them by means of pseudocode algorithms.

Our model transformation rules use the system elements with relevance annotations (cf. Section 2.2.6) as the central basis to determine *SwRE-relevant elements* in CONSENS system models and to derive the corresponding elements of component-based MSD specifications from them. Figure 3.7 depicts on the left-hand side the concept taxonomy of *SwRE-relevant structural elements* of CONSENS system models and on the right-hand side the concept taxonomy of structural elements of component-based MSD specifications (cf. Figure 3.4) as well as the mappings in between.

Regarding the SwRE-relevant structural elements of CONSENS system models (left-hand side of Figure 3.7), we distinguish SwRE-relevant structural elements of CONSENS system models into *SwRE-relevant system elements* and *SwRE-relevant environment elements* (cf. left-hand side of Figure 3.7). Similarly to the transition from CONSENS to the platform-independent MECHATRONICUML software design (cf. Section 2.3), we further distinguish SwRE-relevant system elements into *discrete software components* and *SwRE-relevant continuous software components*.

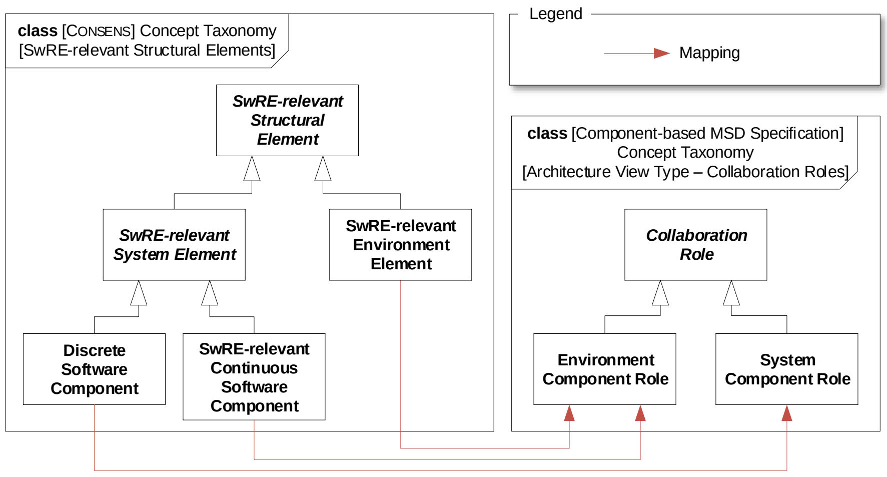

class [CONSENS] Concept Taxonomy [SwRE-relevant Structural Elements]
SwRE-relevant Structural Element
SwRE-relevant System Element
SwRE-relevant Environment Element
Discrete Software Component
SwRE-relevant Continuous Software Component
Legend
Mapping
class [Component-based MSD Specification] Concept Taxonomy [Architecture View Type – Collaboration Roles]
Collaboration Role
Environment Component Role
System Component Role

Figure 3.7: Mappings between SwRE-relevant structural elements of CONSENS system models and structural elements of component-based MSD specifications

Discrete software components realize the SUD's coordination behavior, communicate via messages, and are concretized by means of state-based models in the discipline of software engineering (cf. Section 2.2.6). Thus, they are an inherent part of MSD specifications documenting requirements on the coordination behavior. We consider all system elements with a relevance annotation "SE" discrete software components (cf. Section 2.2.6), and we define in which cases we consider the remaining CONSENS system model element kinds SwRE-relevant in the following. Furthermore, we sketch how our model transformation rules exploit discrete software components to determine other SwRE-relevant CONSENS system model elements and how these are mapped to component-based MSD specification elements.

3.4.1 Derive MSD Use Cases

The partial model *Application Scenarios* (cf. Section 2.2.2) in CONSENS system models provides a system-level structuring of the desired SUD functionality that is similar to the structuring by means of use cases in MSD specifications. In order to reuse this structuring, we create an empty MSD use case for each SwRE-relevant application scenario in the system model.

We consider an application scenario SwRE-relevant, if at least one discrete software component realizes one of the functions that are induced by the application scenario (cf. Figure 3.8). For any SwRE-relevant application scenario <appScenName>, our model transformations each derive (cf. Figure 3.8):

- A package MSD Use Case <appScenName>,
- an empty collaboration <appScenName> as part of the package MSD Use Case <appScenName>,
- an empty package <appScenName> Interfaces as part of the package MSD Use Case <appScenName>,
- and an empty package <appScenName> Types as part of the package MSD Use Case <appScenName>.

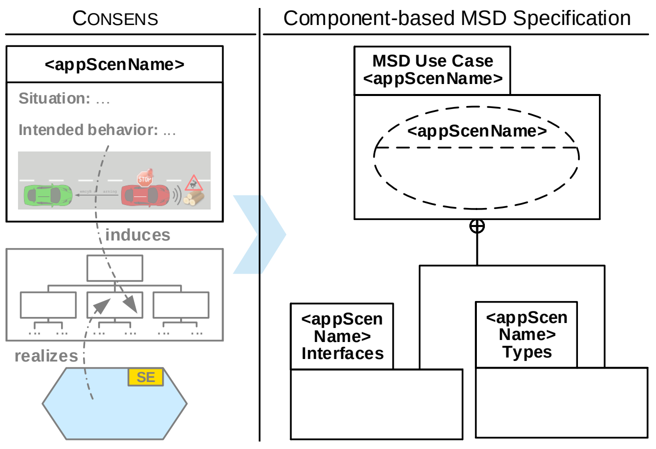


Figure 3.8: Derive MSD use cases—mapping of SwRE-relevant application scenarios

3.4.2 Derive Structure

The CONSENS partial models *Environment* (cf. Section 2.2.1) and *Active Structure* (cf. Section 2.2.5) together specify the SUD's system architecture in a discipline-spanning manner, thereby providing the basis for the software architectures of MSD use cases. In order to derive such software architectures, we again exploit the relational trace links within the system models to determine the SwRE-relevant structural elements realizing an application scenario. We distinguish between the following kinds of SwRE-relevant structural elements and its particular mappings (cf. Figure 3.7 for a coarse-grained overview of these mappings).

3.4.2.1 Derive System Component Roles from Discrete Software Components

We consider a system element a discrete software component and thereby SwRE-relevant if it has a relevance annotation "SE". For any discrete software component <sysElemName> realizing at least one of the functions that are induced by an application scenario <appScenName>, our model transformations each derive (cf. Figure 3.9):

- A component type **<sysElemName>** as part of the package <appScenName> Types
- and a system component role :<sysElemName> as part of the collaboration <appScenName>, where the system component role has the component type as classifier.

3.4.2.2 Derive Environment Component Roles from Environment Elements

Environment elements are elements outside the SUD's boundary that the SUD interacts with (cf. Section 2.2.1). Not all of them are SwRE-relevant, because some of them only interact with SwRE-irrelevant system elements. However, we consider the direct communication between discrete software components and environment elements via information flows SwRE-relevant as it imposes requirements on the interactions between the SUD's coordination behavior part and its environment. In terms of MSD specifications, we hence treat SwRE-relevant environment elements as environment component roles (cf. Figure 3.7).

More precisely, we consider an environment element SwRE-relevant if it is connected via an information flow with a discrete software component. For any SwRE-relevant environment

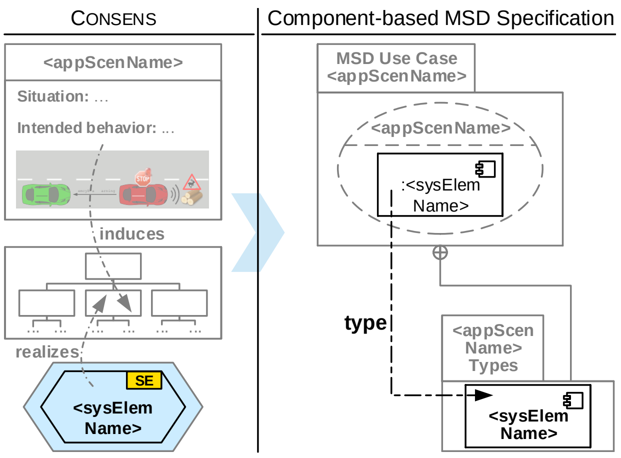


Figure 3.9: Derive structure—mapping of discrete software components

element <envElemName> affecting an SwRE-relevant application scenario <appScenName> and connected via an information flow with a discrete software component that is associated via functions with <appScenName>, our model transformations each derive (cf. Figure 3.10):

- A component type **<envElemName>** as part of the package <appScenName> Types
- and an environment component role :<envElemName> as part of the collaboration <appScenName>, where the environment component role has the component type as classifier.

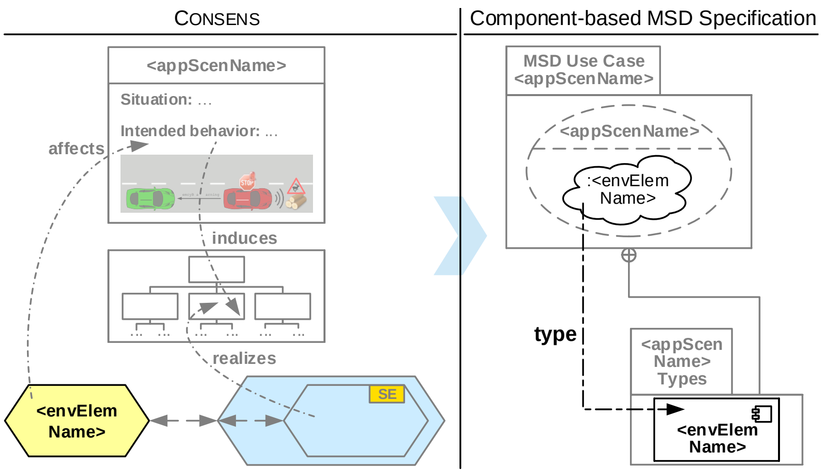


Figure 3.10: Derive structure—mapping of SwRE-relevant environment elements

3.4.2.3 Derive Environment Component Roles from Continuous Software Components

Continuous software components realize the SUD's control behavior and are concretized in the control engineering discipline (cf. Sections 2.2.6 and 2.3). Not all of them are SwRE-relevant, because some of them only interact with SwRE-irrelevant system and environment elements. However, we consider the direct communication between discrete and continuous software components via information flows SwRE-relevant as it imposes requirements on the interactions between the SUD's coordination and control behavior parts. In terms of MSD specifications, we hence treat SwRE-relevant continuous software components as environment component roles (cf. Figure 3.7).

More precisely, we consider a continuous software component (i.e., a system element with the relevance annotation "CE") SwRE-relevant if it is connected via an information flow with a discrete software component. For any SwRE-relevant continuous software component <contSysElemName> adjacent with a discrete software component, where both software components are associated via functions with an SwRE-relevant application scenario <appScenName>, our model transformations each derive (cf. Figure 3.11):

- A component type **<contSysElemName>** as part of the package <appScenName> Types
- and an environment component role :<contSysElemName> as part of the collaboration <appScenName>, where the environment component role has the component type as classifier.

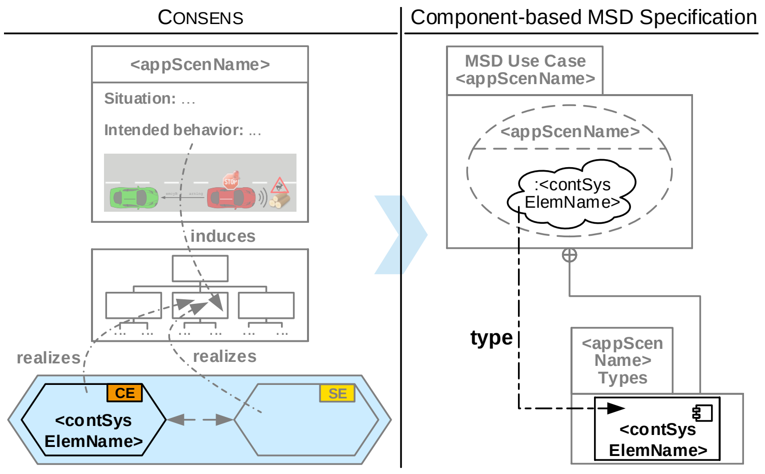


Figure 3.11: Derive structure—mapping of SwRE-relevant continuous software components

3.4.2.4 Derive Interfaces, Ports, and Connectors

We consider an information flow SwRE-relevant if it connects ports of SwRE-relevant structural elements. For any SwRE-relevant information flow between ports of structural elements associated with an application scenario <appScenName>, the ports each having the same information flow specification <portSpecName> that encompasses one or several information flow item(s) <informationItemName>, our model transformations each derive (cf. Figure 3.12):

- An interface **<portSpecName>** as part of the package <appScenName> Interfaces,
- one or several operations <informationItemName>() as part of the interface **<portSpecName>**,
- a port as part of the sending component type using **<portSpecName>** as required interface,
- a port as part of the receiving component type using **<portSpecName>** as provided interface,

- and an assembly connector :<portSpecName> connecting the required and provided interface of the roles' ports as part of the collaboration <appScenName>.

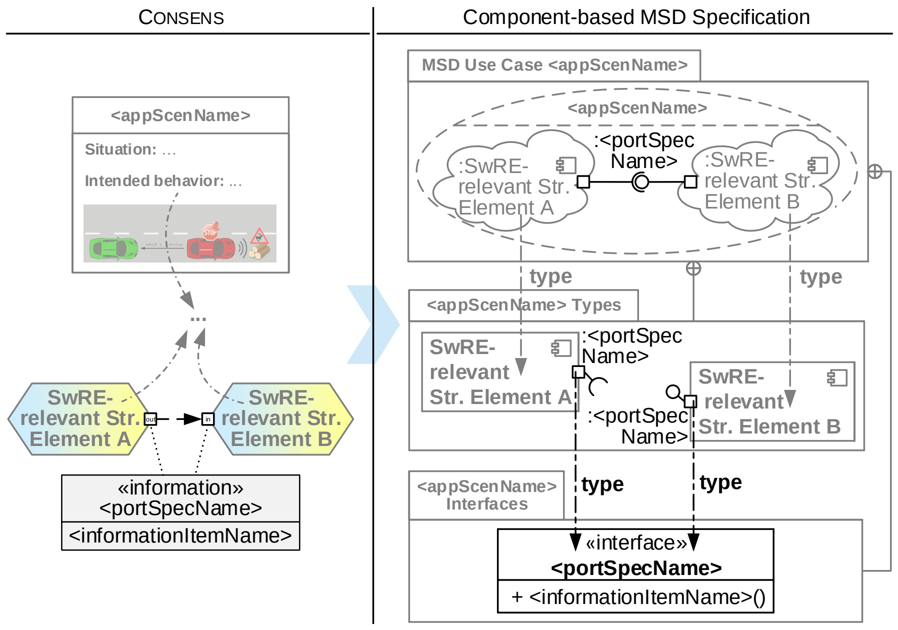

CONSENS
Component-based MSD Specification
<appScenName>
Situation: ...
Intended behavior: ...
SwRE-relevant Str. Element A
SwRE-relevant Str. Element B
«information» <portSpecName>
<informationItemName>
MSD Use Case <appScenName>
<appScenName>
:<portSpec Name>
:SwRE-relevant Str. Element A
:SwRE-relevant Str. Element B
type
type
<appScenName> Types
SwRE-relevant Str. Element A
SwRE-relevant Str. Element B
:<portSpec Name>
:<portSpec Name>
<appScenName> Interfaces
type
type
«interface» <portSpecName>
+ <informationItemName>()

Figure 3.12: Derive structure—mapping of SwRE-relevant information flows

3.4.3 Derive MSDs

Similarly to MSDs and scenario-based formalisms in general, the partial model *Behavior – Sequences* (cf. Section 3.1) specifies partially ordered sequences of actions. Therefore, we reuse the information contained in this partial model to derive initial MSDs for the MSD specification.

We consider a *Behavior – Sequence* SwRE-relevant, if it refines an SwRE-relevant application scenario and encompasses both SwRE-relevant lifelines and actions. We consider a *Behavior – Sequence* lifeline SwRE-relevant, if it represents a SwRE-relevant structural element. We consider a *Behavior – Sequence* action SwRE-relevant, if it is incident to two SwRE-relevant lifelines, associates an information flow between the SwRE-relevant structural elements that the lifelines represent, and associates an item of an information flow port specification used by the SwRE-relevant structural elements. For any SwRE-relevant *Behavior – Sequence* <behSeqName> refining an SwRE-relevant application scenario <appScenName> and its SwRE-relevant lifelines <lifelineNameA>/<lifelineNameB> and actions <actionName>, where the lifelines represent elements associated with <appScenName> and the action associates an SwRE-relevant information flow as well as an information item <informationItemName> as part of an information flow specification <portSpecName>, our model transformations each derive (cf. Figure 3.13):

- An MSD <behSeqName> as part of the collaboration <appScenName>,
- lifelines <lifelineNameA>/<lifelineNameB> as part of the MSD <behSeqName>, where the lifelines represent the collaboration roles corresponding to the SwRE-relevant structural CONSENS elements,

- and MSD messages `<actionName>` between <lifelineNameA> and <lifelineNameB>, where the MSD messages associate the connector :<portSpecName> and the operation <informationItemName>() as part of the interface **<portSpecName>**.

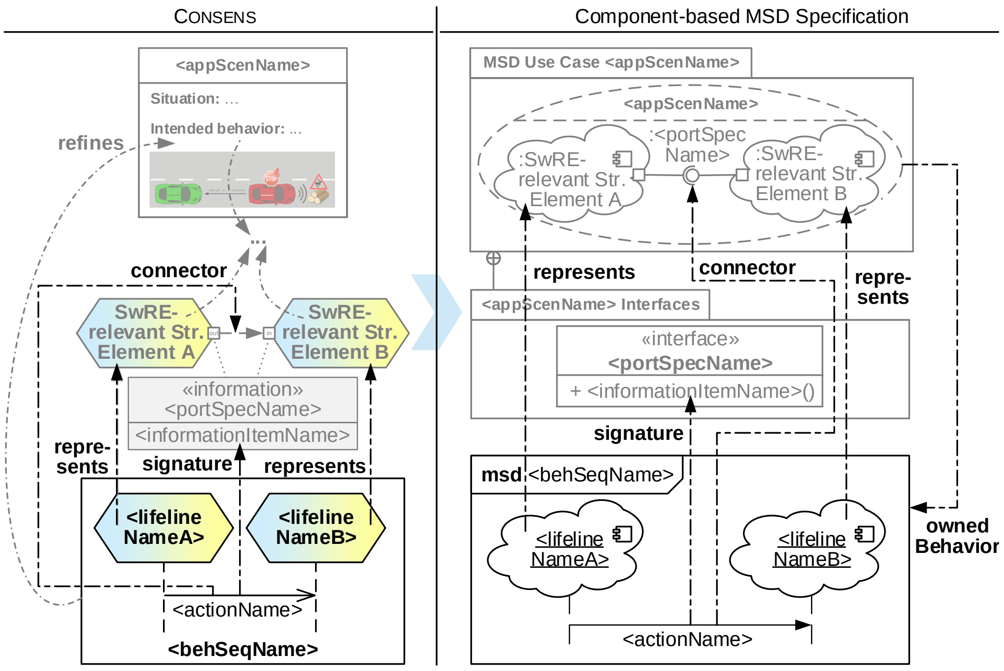


Figure 3.13: Derive MSDs—mapping of SwRE-relevant *Behavior – Sequences*

3.5 Support for Manual Refinement of MSD Specifications

Our model transformations (cf. Section 3.4) do not derive any MSD-specific modeling constructs specifying the modality and the execution kind of messages nor conditional behavior. This is due to the fact that this information is specified in the CONSENS system models informally by means of text and graphics as part of the partial models *Application Scenarios* and *Requirements*. Thus, we cannot extract this information automatically by means of model transformations.

Furthermore, the way of specifying the CONSENS partial model *Behavior – Sequences* is different than the way of specifying MSDs (cf. Section 3.1.2 and Section 2.4.2.3, respectively). That is on the one hand, *Behavior – Sequences* exemplarily describe existential behavior that refine whole situations of *Application Scenarios* in a self-contained manner. On the other hand, MSD specifications describe the universal behavior partitioned across multiple MSD use cases and fine-granular MSDs. Another input to MSD specifications are CONSENS *Behavior – States*, which semi-formally describe the overall, discipline-spanning behavior of the SUD in a state-based instead of a scenario-based way (cf. Section 3.1.3). MSD specifications shall cover the SwRE-relevant parts of this partial model. Such semantic translations require cognitive and creative effort and cannot be automated by model transformations.

Thus, the initially derived MSD specifications have to be manually refined by the Software Requirements Engineers. In this section, we present different means that support them in this task. Figure 3.14 shows the expanded BPMN *sub-process* Refine MSD Specification, which is collapsed in the overall process depicted in Figure 3.6. The sub-process encompasses four manual steps and two tool-supported steps, which we explain in the following. First, we support the manual steps by means of nine informal and constructive guidelines (cf. Section 3.5.1). Second, we provide an automatic check of static coverage rules between MSD specifications and the CONSENS partial model *Behavior – States* (cf. Section 3.5.2). Third, for any *Behavior – Sequence*, we derive each an existential MSD whose traces have to be producible by any system satisfying the MSD specification (cf. Section 3.5.3).

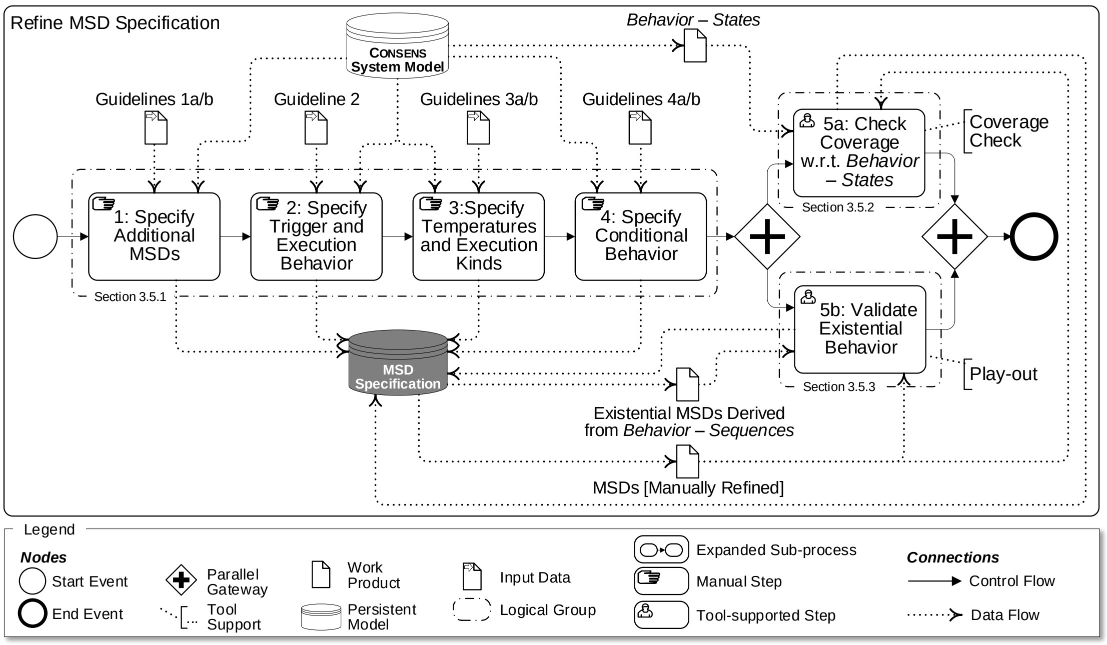


Figure 3.14: Expanded BPMN *sub-process* Refine MSD Specification (cf. Figure 3.6)

3.5.1 Informal Guidelines

The guidelines encompass common patterns for the manual refinement of each aspect of a derived or updated MSD specification w.r.t. a CONSENS system model, like the most likely source of information (e.g., the part "Situation" of *Application Scenarios*) in CONSENS or certain keywords (e.g., "if" or "shall"). We partitioned all guidelines in four parts with the purpose of a uniform presentation. Furthermore, each guideline compactly includes all relevant information on one page. Both the uniform and compact presentation aid the Software Requirements Engineers in quickly finding the information that is required for their specific needs.

For example, Figure 3.15 presents the guideline for specifying additional assumption MSDs. Part I of this guideline explain its overall purpose, part II sketches the semantics of the considered MSD modeling elements, part III presents concrete proposals for the procedure of the refinement, and part IV illustrates the guideline by means of examples. We present the remaining guidelines in Appendix A.1.

In the following, we sketch the particular four manual guideline-supported steps depicted in Figure 3.14. We exemplarily conduct these steps in Appendix A.2.2.2.

1. Specify Additional MSDs

a) Add Assumption MSDs

I. Goal

Software-intensive systems depend on their environment, that is, the behavior of external systems and of physical processes. Assumptions on this environment behavior have to be considered and documented in a requirements specification (cf. [ISO18b]). The Software Requirements Engineer can specify restrictions on the events that occur in the environment by means of assumption MSDs. In many cases, only such environment assumptions enable that the software requirements are realizable at all. The goal of this guideline is the manual specification of such environment assumptions.

II. Description of Elements

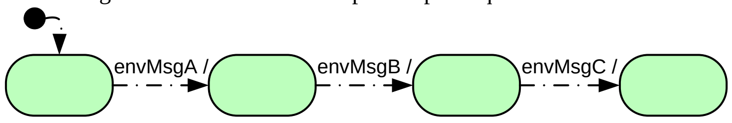
An environment assumption is specified by means of a dedicated assumption MSD, which has the stereotype «EnvironmentAssumption» applied.

An assumption MSD specifies the following aspects:
- Fixed order of environment events*
- Conditional behavior
  - Environment events may occur iff certain conditions hold
  - Assumed real-time restrictions on the environment**
- Forbidden environment events or event sequences (cf. Guideline 4a)

III. Guidelines

General:
- Partial models with informal/semi-formal information relevant to this refinement step: *Behavior – States*, *Application Scenarios*, and *Requirements*.
- Investigate *Application Scenarios* for a dedicated section "Environment assumptions" (cf. [Gre11]).
- Investigate *Requirements* for requirements with attribute "Assumption" (cf. [ISO18b]).
- Often, the execution kinds of the messages in contrast to the requirement MSDs have to be reversed since an assumption MSD takes a different perspective than a requirement MSD (cf. [BGP13]).

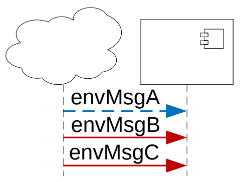
* Determine information about fixed environment event sequences:
- Investigate *Behavior – States* for plain input sequences:



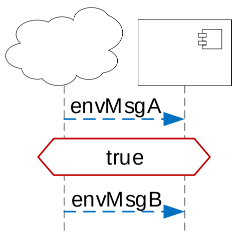
- Investigate *Application Scenarios* for sequential environment behavior (keywords: "after", "afterward", "subsequently", "following").

IV. Examples

* Plain fixed order (cf. [Gre11]):



* Environment reaction envMsgA must occur before environment event envMsgB (cf. [BGP13]):



** Environment event envMsgA may occur at most every minTime time units (cf. [*HFK+16]):

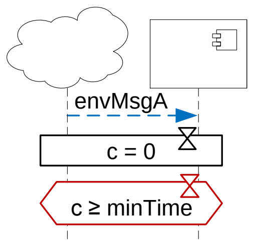


Figure 3.15: Guideline 1a—Specify additional MSDs: Add assumption MSDs

Step 1: Specify Additional MSDs

Before adding further requirement MSDs, the Software Requirements Engineers has to understand the behavior of the environment that the SUD interacts with. Thus, they should start with specifying environment assumptions. Guideline 1a (cf. Figure 3.15 supports the Software Requirements Engineers in determining the relevant information for this refinement step.

Afterward, the Software Requirements Engineer has to investigate the CONSENS system model for SwRE-relevant system behavior that is not specified as part of the exemplary *Behavior – Sequences* (cf. Guideline 1b in Figure A.1 in Appendix A.1). This behavior has to be specified manually by means of additional requirement MSDs due to the fact that we automatically derive initial MSDs only from exemplary *Behavior – Sequences*. That is, behavior that is not specified as part of this partial model is not automatically derived.

Step 2: Specify Trigger and Execution Behavior

The behavior of an application scenario and an MSD use case, respectively, is often triggered by different situations. Thus, an MSD use case behavior can often be distinguished into a *trigger behavior* and the actual *execution behavior*. In this case, the Software Requirements Engineer should divide these behaviors accordingly to avoid the redundant specification of the same execution behavior intertwined with n trigger behaviors in n MSDs. This also includes the consideration of the partial model *Behavior – States*, which specifies the complete system behavior. The Software Requirements Engineer should formalize the remaining paths leading to the *Behavior – States* state(s) corresponding to the initially generated MSD by means of other MSDs. Thereby, the redundant specification of several trigger behaviors is avoided. Typically, this refinement step leads to a rearrangement of the message sequences that are initially derived from the *Behavior – Sequences*. We provide the corresponding Guideline 2 in Figure A.2 in Appendix A.1.

Step 3: Specify Temperatures and Execution Kinds

The third step in refining an MSD specification is to investigate the informal information of the partial models *Application Scenarios* and *Requirements* in terms of the temperature and the execution kind for the particular messages of the initially generated MSDs, which may have been modified in step 2. We provide the corresponding Guidelines 3a for adding temperatures and 3b for adding execution kinds in Figure A.3 and Figure A.4 in Appendix A.1, respectively.

Step 4: Specify Conditional Behavior

The fourth step in refining an MSD specification is to investigate the informal/semi-formal information of the partial models *Application Scenarios*, *Requirements*, and *Behavior – States* in terms of conditional behavior. We provide Guideline 4a for adding conditions and Guideline 4b for adding real-time requirements in Figure A.5 and Figure A.6 in Appendix A.2, respectively.

3.5.2 Automatic Coverage Check

Step 5a: Check Coverage w.r.t. the Partial Model *Behavior – States*

During the manual refinement, the Software Requirements Engineers can overlook particular message sequences indicated by the CONSENS system model despite both the automatic derivation and the guideline-supported refinement of MSD specifications. Furthermore, they can introduce message sequences that are not intended by the CONSENS system model specification. The CONSENS partial model *Behavior – States* provides a universal and discipline-spanning behavior description for the overall system and hence provides an adequate basis to validate the completeness and conciseness of MSD specifications. Thus, we provide an automatic coverage check between MSD specifications and the CONSENS partial model *Behavior – States*, which we sketch in Figure 3.16.

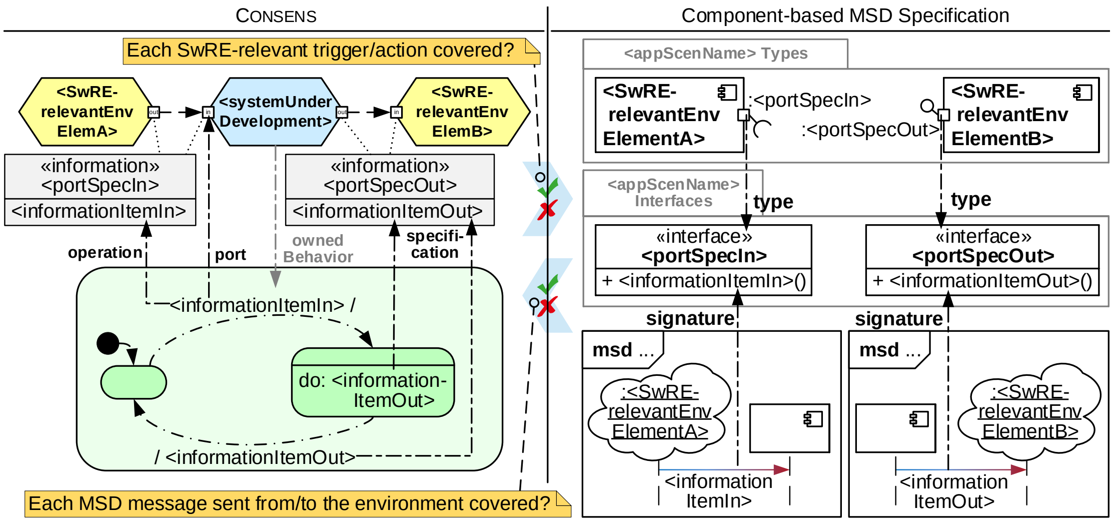


Figure 3.16: Automatic coverage check between MSD specifications and *Behavior – States*

The check statically checks two sets of coverage rule sets. The first rule set aims at checking the completeness of MSD specifications w.r.t. the CONSENS partial model *Behavior – States*. It encompasses two rules for checking whether each SwRE-relevant *Behavior – States* trigger and action is covered by the MSD specification through an environment message and a system message sent to the environment, respectively. If the rule set is not fulfilled, some interactions with the environment that the partial model *Behavior – States* specifies are missing in the MSD specification. The second rule set aims at checking the conciseness of MSD specifications w.r.t. the CONSENS partial model *Behavior – States*. It encompasses two rules for checking whether each environment message and a system message sent to the environment in the MSD specification is covered by an SwRE-relevant *Behavior – States* trigger and action, respectively. If the rule set is not fulfilled, either the MSD specification contains superfluous interactions with the environment or the partial model *Behavior – States* is incomplete. We describe the particular coverage rules more formally in Section 3.8.3.

3.5.3 Automatic Derivation of Existential MSDs

Step 5b: Validate Existential Behavior

The CONSENS partial model *Behavior – Sequences* provides exemplary existential scenarios of whole self-contained situations, whereas MSD specifications describe universal scenarios partitioned across several fine-granular MSDs and multiple MSD use cases (cf. Section 2.4.2.3). Thus, the Software Requirements Engineer typically rearranges the contents of MSDs initially derived from *Behavior – Sequences*, particularly in the guideline-supported step 2 (cf. Section 3.5.1). However, the *Behavior – Sequences* are an important input to the manual refinement since the MSD specifications have to fulfill the exemplary scenarios as described by the *Behavior – Sequences*.

Existential MSDs describe exemplary behavior, where the overall MSD specification shall be able to produce at least one trace that fulfills this exemplary behavior (cf. Section 2.4.2.3). Existential MSDs can be expressed through MSDs containing only cold and executed system messages and cold and monitored environment messages (cf. Section 2.4.2.3). By means of Play-out, the Software Requirements Engineers can check whether there is a trace in the overall state space of an MSD specification that fulfills an existential MSD. That is, they simulate the MSD specification and checks whether there is an event sequence that is able to activate and terminate such an existential MSD, which thereby serves as a test oracle.

Thus, we derive for any SwRE-relevant *Behavior – Sequence* and its SwRE-relevant contents each an existential MSD. The Software Requirements Engineers should check by means of Play-out whether their manual refinement is complete w.r.t. the *Behavior – Sequences*. A more automated approach could try to synthesize a controller that fulfills the existential scenario or perform a reachability analysis on the MSD specification state space.

More specifically, for any SwRE-relevant *Behavior – Sequence* <behSeqName> refining an SwRE-relevant application scenario <appScenName> and its SwRE-relevant lifelines <lifelineNameA>/<lifelineNameB> and actions <actionName>, where the lifelines represent elements associated with <appScenName> and the action associates an SwRE-relevant information flow as well as an information item <informationItemName> as part of an information flow specification <portSpecName>, our model transformations each derive (cf. Figures 3.17 and 3.18):

- An MSD Existential<behSeqName> as part of the collaboration <appScenName>,
- lifelines <lifelineNameA>/<lifelineNameB> as part of the MSD <behSeqName>, where the lifelines represent the corresponding collaboration roles,
- if the sending lifeline of <actionName> represents an SwRE-relevant environment element or an SwRE-relevant continuous system element (cf. Figure 3.17): cold and monitored environment messages <actionName> between <lifelineNameA> and <lifelineNameB>, where the environment messages each associate the operation <informationItemName>() and the connector :<portSpecName>,
- if the sending lifeline of <actionName> represents a discrete software component (cf. Figure 3.18): cold and executed system messages <actionName> between <lifelineNameA> and <lifelineNameB>, where the system messages each associate the operation <informationItemName>() and the connector :<portSpecName> (cf. Figure 3.18).

3.6 Exemplary Application of the Transition Technique

In this section, we exemplarily apply our semi-automatic transition technique. For this purpose, we assume that the Systems Engineers specified a CONSENS system model for the EBEAS as shown in excerpts in Figures 2.2, 3.2, and 3.3, which is input to our transition technique. In the following, we show extracts of the initially derived and manually refined component-based MSD specifications. We provide the complete CONSENS system model and MSD specification in Appendix A.2.1 and Appendix A.2.2, respectively.

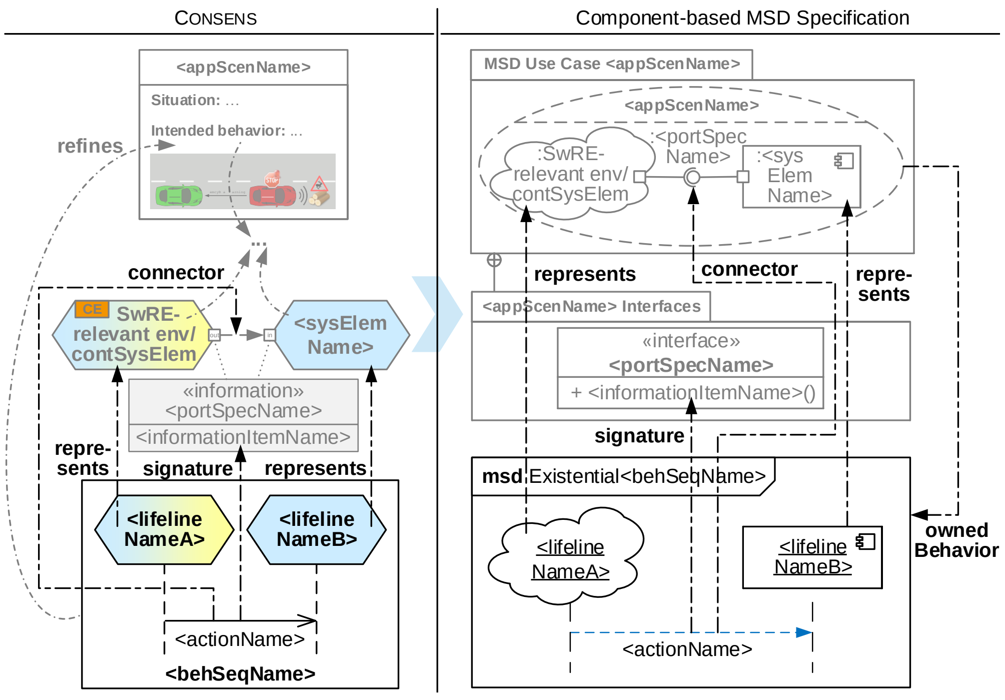


Figure 3.17: Derive environment messages for existential MSDs

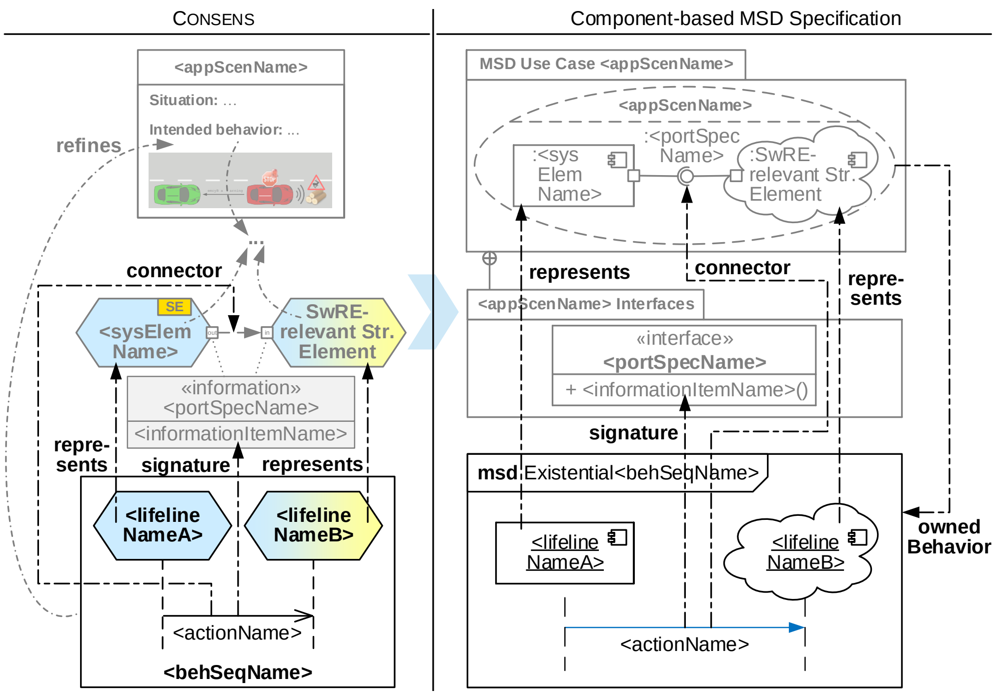


Figure 3.18: Derive system messages for existential MSDs

In Section 3.6.1, we exemplarily perform an initial transition from CONSENS to MSDs. In Section 3.6.2, we exemplarily change the initial CONSENS system model and incrementally update the MSD specification.

3.6.1 Initial Process Iteration

As sketched in the overview of the model transformation rules (cf. Section 3.4), a key aspect of the model transformation part is the identification of information in the system models that is relevant to SwRE. For this purpose, the model transformation rules exploit the relevance annotations in the CONSENS *Active Structure* (cf. Section 2.2.6 and Section 2.3) to identify the SwRE-relevant system elements. Based on these system elements, the model transformation follows the trace links between the different partial models and the information flows to identify further SwRE-relevant system model elements. Thereby, information that is not relevant to SwRE is filtered, and the model transformations derive MSD specification elements from the SwRE-relevant system model elements.

The target model of our transition technique are component-based MSD specifications (cf. Section 3.2). Figure 3.19 shows the initial component-based MSD specification that is automatically derived from a SYSML4CONSENS model as depicted in excerpts in Figure 3.29 in Section 3.9.1.1. This SYSML4CONSENS model encompasses the contents of the CONSENS system model depicted in excerpts in Figures 2.2, 3.2, and 3.3.

The classifier view of the MSD specification is depicted on the left of Figure 3.19. First, it contains interfaces defining operations that are depicted in the class diagrams named ...Interfaces. Second, it encompasses software component types owning ports that are typed by the interfaces in the class diagrams named ...Types.

The architecture view is depicted in the top right of Figure 3.19. It encompasses UML collaborations specifying the particular software component architectures of each MSD use case as the structural basis for the actual MSDs. These architectures define the participants of an MSD use case by means of roles typed by the software component types in the classifier view.

Each use case contains a set of MSDs, which are specified in the interaction view depicted in the bottom right of Figure 3.19. The MSDs formally specify requirements on the coordination behavior of the use case participants (i.e., of the particular software component roles). Each lifeline in the MSD represents a role within the collaboration of a specific use case. The messages correspond to the operations of the interfaces in the classifier view and associate directed connectors in architecture view.

In the following, we exemplarily perform and explain each of the process steps of the Software Requirements Engineer and the Software Architect depicted in the process description in Figure 3.6.

3.6.1.1 Derive MSD Use Cases

Our model transformations create an empty MSD use case for each SwRE-relevant application scenario in the system model (cf. Section 3.4.1). We call an application scenario SwRE-relevant, if at least one SwRE-relevant system element realizes one of the functions that are induced by this scenario. We use the bidirectional relational trace links in the system model to determine the SwRE-relevance of an application scenario and to identify the system elements that are used to realize the functionality specified in this scenario. Furthermore, we use the relevance

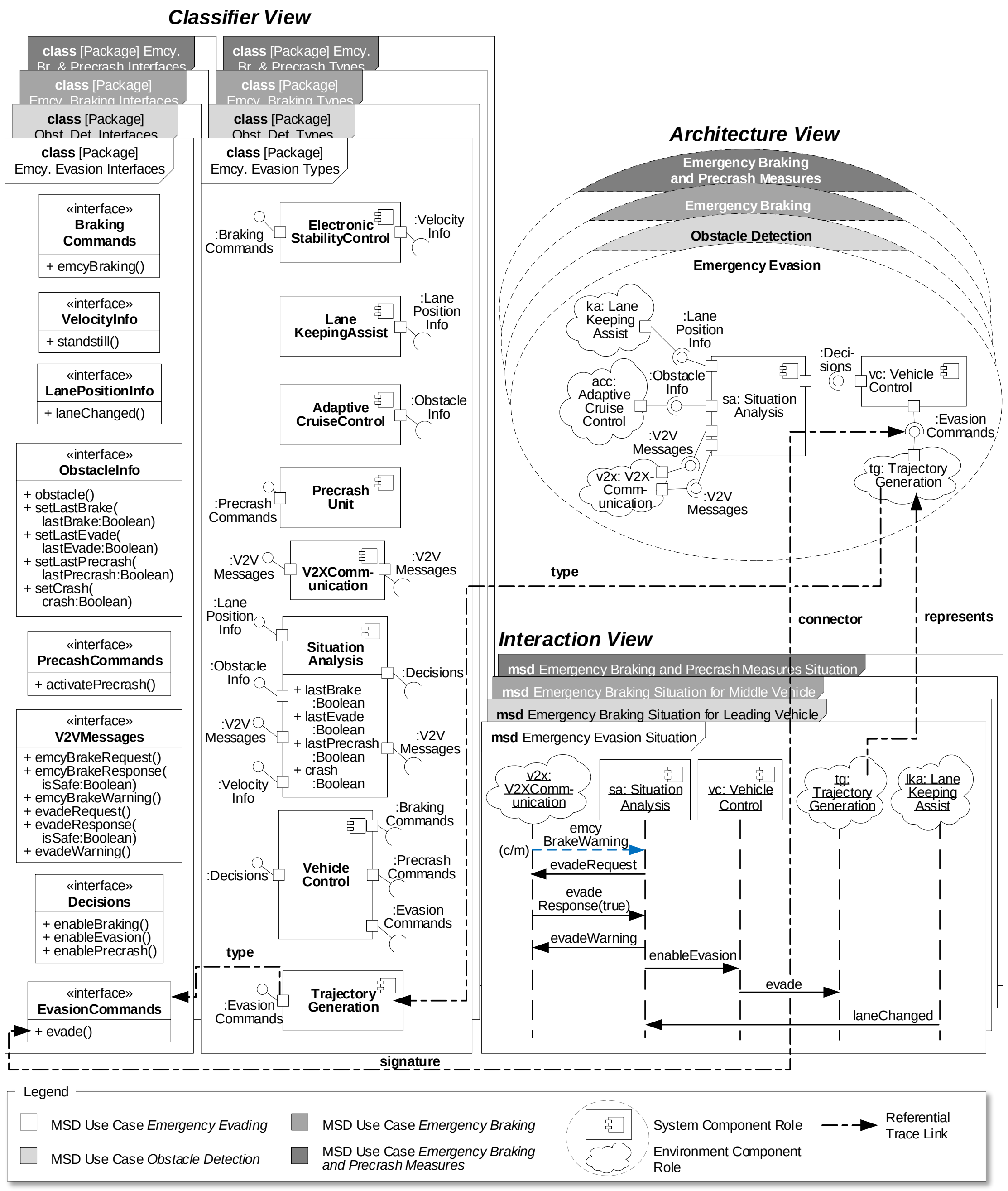

Classifier View
class [Package] Emcy. Br. & Precrash Interfaces
class [Package] Emcy. Br. & Precrash Types
class [Package] Emcy. Braking Interfaces
class [Package] Emcy. Braking Types
class [Package] Obst. Det. Interfaces
class [Package] Obst. Det. Types
class [Package] Emcy. Evasion Interfaces
class [Package] Emcy. Evasion Types
«interface» BrakingCommands + emcyBraking()
«interface» VelocityInfo + standstill()
«interface» LanePositionInfo + laneChanged()
«interface» ObstacleInfo + obstacle() + setLastBrake(lastBrake:Boolean) + setLastEvade(lastEvade:Boolean) + setLastPrecrash(lastPrecrash:Boolean) + setCrash(crash:Boolean)
«interface» PrecashCommands + activatePrecrash()
«interface» V2VMessages + emcyBrakeRequest() + emcyBrakeResponse(isSafe:Boolean) + emcyBrakeWarning() + evadeRequest() + evadeResponse(isSafe:Boolean) + evadeWarning()
«interface» Decisions + enableBraking() + enableEvasion() + enablePrecrash()
«interface» EvasionCommands + evade()
Electronic StabilityControl
Lane KeepingAssist
Adaptive CruiseControl
Precrash Unit
V2XComm-unication
Situation Analysis + lastBrake :Boolean + lastEvade :Boolean + lastPrecrash :Boolean + crash :Boolean
Vehicle Control
Trajectory Generation
type
signature
Architecture View
Emergency Braking and Precrash Measures
Emergency Braking
Obstacle Detection
Emergency Evasion
ka: Lane Keeping Assist
acc: Adaptive Cruise Control
v2x: V2X-Comm-unication
sa: Situation Analysis
vc: Vehicle Control
tg: Trajectory Generation
connector
represents
Interaction View
msd Emergency Braking and Precrash Measures Situation
msd Emergency Braking Situation for Middle Vehicle
msd Emergency Braking Situation for Leading Vehicle
msd Emergency Evasion Situation
(c/m) emcy BrakeWarning
evadeRequest
evade Response(true)
evadeWarning
enableEvasion
evade
laneChanged
Legend
MSD Use Case Emergency Evading
MSD Use Case Obstacle Detection
MSD Use Case Emergency Braking
MSD Use Case Emergency Braking and Precrash Measures
System Component Role
Environment Component Role
Referential Trace Link

Figure 3.19: Initially generated MSD specification

annotations of these system elements: If one of the system elements is annotated with “SE” and hence is SwRE-relevant, then also the application scenario is SwRE-relevant.

For example, the application scenario Emergency Evasion in Figure 2.2 in Section 2.2 is SwRE-relevant since the system element Vehicle Control is annotated with “SE” and realizes the function Control Steering induced by this application scenario. Thus, our model transformations add the empty MSD use case *Emergency Evasion* to the MSD specification, which is depicted by means of the equally named UML collaboration in the architecture view of Figure 3.19. The same principle holds for the remaining MSD use cases.

3.6.1.2 Derive Structure

Derive System Component Roles from Discrete Software Components

For any discrete software component associated with an application scenario, our model transformations derive each a system component role as part of the MSD use case collaboration and a corresponding component type in the classifier view type (cf. Section 3.4.2.1).

For example, the system elements Situation Analysis and Vehicle Control realize the application scenario Emergency Evasion and are discrete software components due to their “SE” relevance annotations (cf. Figure 2.2). Thus, the transformation derives the system component roles sa: SituationAnalysis and vc: VehicleControl as part of collaboration *Emergency Evasion* (cf. architecture view in Figure 3.19). Furthermore, the transformation derives their corresponding component types **SituationAnalysis** and **VehicleControl** as part of the package Emcy. Evasion Types (cf. classifier view in Figure 3.19).

Derive Environment Component Roles from SwRE-relevant Environment Elements

We consider all CONSENS environment elements that are connected via information flows with discrete software components SwRE-relevant (cf. Section 3.4.2.2).

For example, this encompasses all ECUs like Active Front Steering or Lane Keeping Assist (cf. Figure 2.2). Bus systems like FlexRay are disregarded since they are connected to the EBEAS via energy flows and are of no interest to the Software Requirements Engineer caring only about the logical view. Furthermore, we use the relational trace links of the type affects to determine the environment elements involved in an application scenario to derive the environment component roles of the corresponding use case. For example, we derive the environment component roles lka: LaneKeepingAssist and acc: AdaptiveCruiseControl for the use case *Emergency Evasion* but neglect the environment element Precrash Unit for this use case since it only participates in the application scenario Emergency Braking and Precrash Measures. Besides deriving the environment component roles within the collaborations in the architecture view type, we also generate the corresponding component types in the classifier view type.

Derive Environment Component Roles from SwRE-relevant Continuous System Elements

We consider all continuous software components that are connected via information flows with discrete software components and affect the corresponding application scenario SwRE-relevant (cf. Section 3.4.2.3).

For example, the system element Trajectory Generation is only relevant to the discipline of control engineering since it is tagged with the relevance annotation “CE” (cf. Figure 3.2). However, the Trajectory Generation is activated by the SwRE-relevant system element Vehicle Control via the information flow evasionCommands. Thus, we consider the system element Trajectory Generation SwRE-relevant. We derive environment component roles from continuous software components, since the Software Requirements Engineer focuses on discrete software components and leaves the design of continuous software components to the Control Engineer. Thus, the transformation derives an environment component role tg: TrajectoryGeneration as part of the MSD use case *Emergency Evasion*. Furthermore, it derives the corresponding component type.

Derive Interfaces, Ports, and Connectors

Finally, we consider information flows between SwRE-relevant structural elements in the structural CONSENS partial models SwRE-relevant (cf. Section 3.4.2.4). This includes information flows between discrete software components as well as information flows between discrete software components and SwRE-relevant environment elements or SwRE-relevant continuous software components. For any SwRE-relevant flow and its adjacent two ports and their port specification, we derive a corresponding connector, its adjacent two ports, and an interface used one time as provided and one time as required interface by the two ports.

For example, there is an information flow decisions from one port of the discrete software component SituationAnalysis to one port of the discrete software component VehicleControl (cf. Figure 3.2). The ports associate the port specification decisions, encompassing three information flow items. Thus, the model transformations derive an interface **Decisions** encompassing three operations derived from the three information flow items as part of the package Emcy. Evasion Interfaces (cf. classifier view in Figure 3.19). Furthermore, it derives each a port :Decisions as part of the component types **SituationAnalysis** and **VehicleControl**, using the interface as required and provided, respectively (cf. classifier view in Figure 3.19). Finally, it derives the connector :Decisions between the system component roles sa: SituationAnalysis and vc: VehicleControl as part of the collaboration *Emergency Evasion* (cf. architecture view in Figure 3.19).

Another example is the system element Trajectory Generation, which has two ingoing information flows as well as one outgoing information flow (cf. Figure 3.2). The two information flows trajectoryCommands and steeringInfo describe signal-based communication with the environment element ActiveFrontSteering, covering the continuous control behavior of the SUD. As motivated before, the Software Requirements Engineer is mainly interested in the SUD's coordination behavior. Thus, the model transformation determines from these three information flows only the flow evasionCommands between the SwRE-relevant structural elements as SwRE-relevant. From this information flow, it derives the connector :EvasionCommands between the ports of vc: VehicleControl and tg: TrajectoryGeneration in the MSD use case *Emergency Evasion* (cf. Figure 3.19). Furthermore, the transformation derives the corresponding ports and interface.

3.6.1.3 Derive MSDs

For any *Behavior – Sequence* refining an SwRE-relevant application scenario in the system model, our model transformations derive each one MSD in the MSD specification (cf. Section 3.4.3).

For example, our model transformations derive the MSDs in the lower right of Figure 3.19 from the *Behavior – Sequences* in Figure 3.2. We map an initially derived MSD to an MSD use case if the source *Behavior – Sequence* has a relational trace link of the type refines to the corresponding application scenario (cf. shades of gray for MSDs and MSD use cases in Figure 3.19). Thus, our model transformations derive the MSD Emergency Evasion Situation from the equally named *Behavior – Sequence* refining the application scenario Emergency Evasion. The SwRE-relevant *Behavior – Sequence* lifelines are mapped to lifelines in the corresponding MSD, and the SwRE-relevant actions are mapped to MSD messages between the corresponding MSD lifelines. Thus, our model transformations derive the lifelines vc: VehicleControl and tg: TrajectoryGeneration as well as the MSD message `evade` in between as part of the MSD Emergency Evasion Situation. Furthermore, our model transformations derive the corresponding referential trace links of these elements as depicted in the figure.

As it is the case for the structural partial models, the partial model *Behavior – Sequences* contains discipline-spanning information that can be irrelevant to SwRE. For example, the *Behavior – Sequence* Emergency Evasion Situation specifies looping actions like the continuous signal-based interaction between the system element Trajectory Generation and the environment element Active Front Steering. Since such actions are not part of the message-based coordination behavior between discrete software components, our model transformations filter such information. That is, they determine the lifeline Active Front Steering SwRE-irrelevant, because it represents a SwRE-irrelevant environment element. Consequently, the model transformations determine the also incident actions SwRE-irrelevant.

3.6.1.4 Refine MSD Specification

The initially derived MSDs have no MSD-specific modeling constructs specifying the modality and the execution kind of messages or the conditional behavior since the information for this is specified informally in the partial models *Application Scenarios* and *Requirements*. Furthermore, the way of specifying *Behavior – Sequences* and *Behavior – States* is different than the way of specifying MSDs (cf. Sections 3.1.2 and 3.1.3, respectively). That is on the one hand, *Behavior – Sequences* describe exemplary existential behavior that refine whole situations of *Application Scenarios* in a self-contained manner, and *Behavior – States* describe the discipline-spanning overall system behavior in a state-based manner. On the other hand, an MSD specification describes the universal behavior partitioned across multiple MSD use cases and fine-granular MSDs.

Thus, the initially derived MSD specification has to be refined. CONSENS system models provide information relevant to this refinement within the partial models *Application Scenarios*, *Requirements*, *Behavior – Sequences*, and *Behavior – States*. Since the information contained in the *Application Scenarios* and *Requirements* is informal text and graphics and *Behavior – Sequences* and *Behavior – States* semi-formally describe the system behavior in different way than MSD specifications, we cannot automate this step by means of model transformations. Thus, the Software Requirements Engineer has to perform it manually. Section 3.5 presents different means that support the Software Requirements Engineer in this task, and we exemplarily conduct this refinement step in Appendix A.2.2.2.

3.6.1.5 Analyze Coordination Behavior Requirements

The formal semantics of MSDs enable different, techniques for the tool-supported analysis of the requirements on the coordination behavior of the SUD (cf. Section 2.4.3). The Software Requirements Engineer is able to identify unintended behavior (e.g., the unintended triggering of execution behavior [*HBM+15]) and scenario inconsistencies (e.g., [*Jap15]) on requirements level.

3.6.1.6 Consolidate Discipline-specific Analysis Results

Beyond the specification of the behavioral partial models in CONSENS, the System Analysts consolidate discipline-specific work products to ensure a reasonable system design. The Software Requirements Engineers as well as the requirements engineers and designers from the other engineering disciplines hand over their respective analysis results to the System Analysts. They consolidate the different discipline-specific analysis results w.r.t. to the System Requirements and decide how to proceed.

Furthermore, our transition technique is intentionally unidirectional as outlined in Section 3.3. Thus, the Software Requirements Engineers do not only hand over their analysis results but also trigger change requests to the System Analysts. These requests encompass changes on the CONSENS system model that influence the automatically derived part of MSD specifications (e.g., changes on interfaces, on the software architecture, or on MSDs that are derived from *Behavior – Sequences*).

3.6.2 Subsequent Process Iterations

In this section, we exemplarily present how changes in the system model are automatically handled by our approach. For this purpose, we assume a change request that introduces an additional system element to the *Active Structure*. Such a change conduct implicates a broad range of changes to several of the remaining partial models. Applying these changes to the initially derived and refined MSD specification (cf. last section) manually would be very tedious and error-prone.

We apply the graph transformation formalism *(Component) Story Diagrams* [FNTZ00; *THHO08; *HT08; *Hol08] to visualize manual changes to the CONSENS system model as well as changes automatically propagated to the MSD specification in terms of the concrete syntax. We distinguish between two basic modification operations of (Component) Story Diagrams: Adding objects/links to our models (visualized by green outlines and the additional label "++") and deleting objects/links (visualized by red outlines and the additional label "--").

In the following section, we exemplarily illustrate the system model changes conducted by the Systems Engineer in detail. In Section 3.6.2.2, we present how these changes are applied to the MSD specification automatically in terms of an incremental update mechanism. We present excerpts of these changes to the source and target models in terms of the abstract syntax in Section 3.7.2.

3.6.2.1 Manual Changes to the CONSENS System Model

All *Application Scenarios* of the EBEAS are safety-critical since they describe functionality that actively intervenes in the vehicle's braking and steering systems. Safety standards like [ISO18a; IEC10; RTCA11] demand dedicated and expensive safety measures in the development of system elements participating in such application scenarios. The system element Situation Analysis realizes many of the *Functions* induced by these *Application Scenarios*. However, it is also responsible for *Functions* that are not as safety-critical as the intervention in the vehicle's braking and steering. For example, the function Send and Receive Warnings is less safety-critical than the functions decomposed from the function Ensure Passenger Safety (cf. Figure 2.2 in Section 2.2) since the vehicle can determine obstacles and braking/evasion maneuvers through sensors. Thus, Send and Receive Warnings describes redundant functionality and has a lower safety level than other functions.

The automotive safety standard ISO 26262 [ISO18a] proposes to allocate functions with different safety levels to dedicated architecture elements, thereby separating the functionality with different safety levels in the realizing system/software architecture. This enables to approach the development of the particular architectural elements with different levels of rigor. Thus, we assume a change request to reduce the effort spent on safety measures in the overall development of the system element Situation Analysis. In this change request, the Safety Engineer[2] decides to encapsulate the part of Situation Analysis realizing the less safety-critical function Send and Receive Warnings into a new, dedicated system element Warning Communication.

Changes to the Partial Model *Environment*

In the context of the change request, the Safety Engineers decide to change port specifications and information flows in the *Environment* (cf. upper left in Figure 3.20) so that the functionality related to sending/receiving warnings is separated from the functionality related to negotiate with other vehicles. For this purpose, they move the information flow items emcyBrakeWarning and evadeWarning from the port specification V2VMessages to the new information flow port specification V2VWarnings. Moreover, the Safety Engineers rename the generic port specification name V2VMessages to V2VNegotiation to reflect this change. They add two new inout ports typed by V2VWarnings to the environment element V2X Communication and the system EBEAS, respectively. Finally, they add an additional, bidirectional information flow between these ports.

Changes to the Partial Model *Active Structure*

The Safety Engineers add the new system element Warning Communication to the *Active Structure* and connects it with the *Environment* (i.e., with the environment element V2X Communication) by means of a delegation connector connecting ports that are typed by V2VWarnings. Furthermore, they add a new port specification warningTranslation as well as a bidirectional information flow between Situation Analysis and Warning Communication. By means of this port specification and the corresponding information flow, the Warning Communication shall translate warning send requests by Situation Analysis into warning messages to the V2X Communication and translate incoming warning messages from V2X Communication into meaningful interpretations for Situation Analysis. Finally, the Safety Engineers move the realizes trace link to the function Send and Receive Warnings from the system element Situation Analysis to the newly added system element Warning Communication.

Changes to the Partial Model *Behavior – Sequences*

Based on the changes in the *Environment* and *Active Structure*, the Safety Engineers add a new lifeline Warning Communication that represents the equally named system element. They move the receiving and sending lifeline of the messages emcyBrakeWarning and evadeWarning, respectively, from Situation Analysis to Warning Communication. Furthermore, they adapt the signature referential trace links of these messages according to the movement of the corresponding information flow items emcyBrakeWarning and evadeWarning to the new port specification V2VWarnings. Finally, the Safety Engineers introduce the new messages leadingVehicleBrakes and communicateEvasion between the lifelines Warning Communication and Situation Analysis, and they set the signature referential trace link of these messages to the corresponding information flow items of the newly added port specification warningTranslation.

[2]The Safety Engineer is not explicitly included in our process roles (cf. Section 3.3) since we focus only on the roles that are influenced directly by our transition technique.

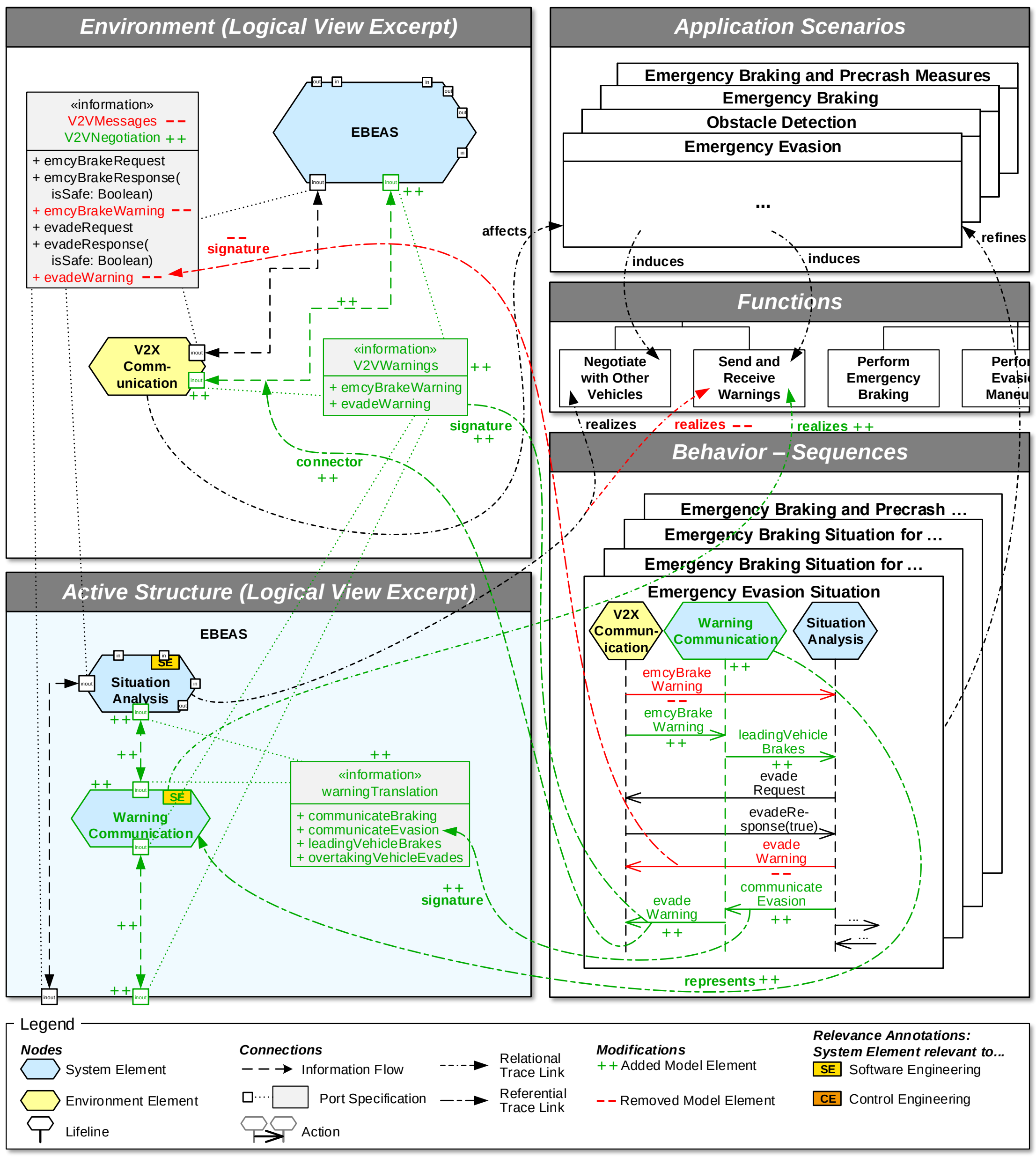


Figure 3.20: Changes in the CONSENS system model

3.6.2.2 Automatic Incremental Update of the MSD Specification

Everything that is subject to the initial automatic transformation (cf. Sections 3.4 and 3.6.1.1 to 3.6.1.3) is incrementally updated when changes to the system model occur. Particularly, no manually added information (cf. Sections 3.5 and 3.6.1.4) gets lost in this case. We apply an intermediate transformation traceability model to enable this incremental update mechanism (cf. Section 3.9.1.2). We present the details of its functional principle in terms of the abstract syntax of the source and target models in Section 3.7.2.

As explained in the last section, the newly added system element Communication Warning fulfills the tasks of sending/receiving warnings formerly conducted by the system element Situation Analysis after a second development process iteration. To reflect this task change, the system element Communication Warning is assigned to the function Send and Receive Warnings induced by the application scenario Emergency Evasion (cf. Figure 3.20). These manual changes to the CONSENS system model influence all three views of the initially derived and refined MSD specification. In the following, we present the changes for the MSD specification in terms of Figure 3.21.

Impact on the Classifier View

The incremental update of our transition technique adapts the contents of the classifier view (cf. left of Figure 3.21) in the following way. In the package Emcy. Evasion Interfaces, it adds the new interfaces **V2VWarnings** and **WarningTranslation**. Furthermore, the interface **V2VMessages** is renamed to **V2VNegotiation** and the operations `emcyBrakeWarning()` and `evadeWarning()` move from this interface to the newly added interface **WarningTranslation**. In the package Emcy. Evasion Types, it adds a component type **WarningCommunication** including four ports using the two new interfaces each as provided and required interface. The component types **V2XCommunication** and **SituationAnalysis** get ports typed by the new interfaces **V2VWarnings** and **WarningTranslation**, respectively.

Impact on the Architecture View

In terms of the architecture view (cf. top right of Figure 3.21), the incremental update adds the system component role wc: WarningCommunication to the initially generated MSD use case *Emergency Evasion*. Furthermore, it adds adds connectors between the roles' new ports and the corresponding ports of wc: WarningCommunication. Note that the incremental update also adds/removes/modifies roles and incident connectors to other MSD use cases according to the trace links between *Active Structure*, *Functions*, and *Application Scenarios*.

Impact on the Interaction View

After the structural changes have been performed, the interaction view (cf. bottom right of Figure 3.21) can be updated. First, the lifeline wc: WarningCommunication as well as a represents referential trace link to the equally named role in the MSD use case is added to the initially generated MSD Emergency Evasion Situation. Second, the message `evadeWarning` from sa: SituationAnalysis to v2x: V2XCommunication as well as its referential trace links are removed. Third, one message `communicateEvasion` sent by sa: SituationAnalysis and one message `evadeWarning` sent to v2x: V2XCommunication are added to the new lifeline wc: WarningCommunication as received and sent message, respectively. The connector referential trace links of these messages are set to the newly added connectors in the MSD use case (visualized for `evadeWarning` in Figure 3.21). Furthermore, the signature referential

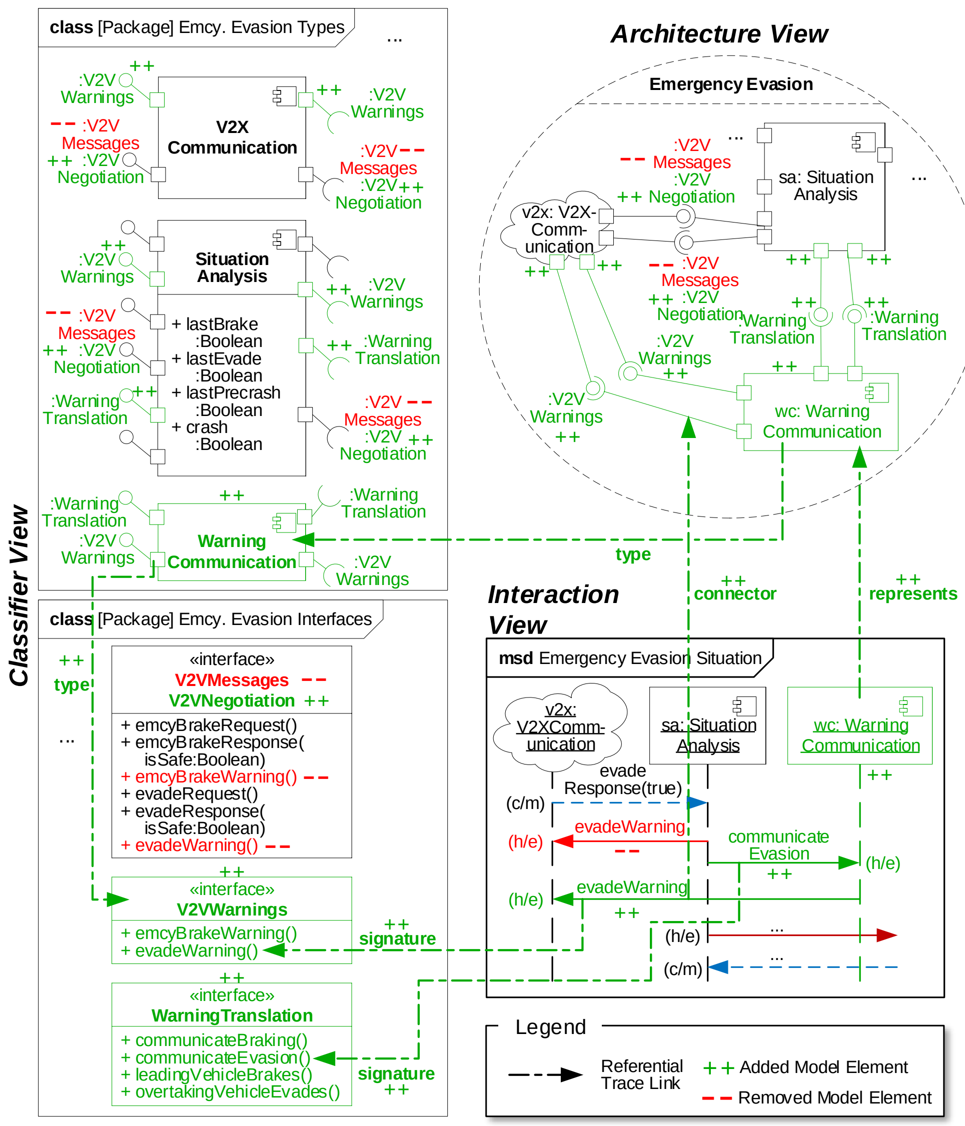


Figure 3.21: Automatically updated MSD specification

trace links are set to the corresponding, equally named operations of the newly added interfaces in the classifier view. Note that the incremental update automatically updates the message `evadeWarning` that the Software Requirements Engineer did not modify earlier, but does not restore the message `emcyBrakeWarning` that the Software Requirements Engineer moved to the MSD NoSafeEmcyBraking (cf. Appendix A.2.2.2). We explain in Section 3.7.2 in terms of the abstract syntax of the particular models how the incremental update handles these different cases.

Summary

In summary, our example shows that a rather small change in the system model can imply a multitude of changes in the MSD specification in terms of concrete and abstract syntax. Thus, the automatic incremental update of our transition technique saves a lot of manual and hence extensive as well as error-prone effort on keeping both models consistent.

3.7 Semi-automatic Establishment of Explicit Inter-model Traceability Between CONSENS System Models and MSD Specifications

Based on the semi-automatic derivation of MSD specifications from CONSENS system models, we present in this section the semi-automatic establishment of vertical, explicit inter-model traceability between both kinds of models (see also our work on this topic in the context of automotive design models [*FHM12]). We apply the traceability framework CAPRA (cf. Section 2.1.2) for this purpose. We provide each one traceability information model for lifecycle traceability and for transformation traceability that define the permissible trace link types between the particular trace artifacts of MSD specifications and CONSENS system models. The resulting traceability models store the trace links external to the MSD specifications and to the CONSENS system models, which has the advantage that these models are not "polluted" with the trace links [PDK$^+$11; DPFK06; KPP06]. We present the traceability information models in Section 3.9.1.2.

In the following section, we present the concepts for the establishment of the lifecycle traceability. In Section 3.7.2, we present the concepts for the establishment of the transformation traceability.

3.7.1 Lifecycle Traceability

For all parts of our transition technique that derive MSD specification elements automatically from CONSENS system models, we also establish (i.e., create as well as maintain) lifecycle trace links between the elements of CONSENS and of the MSD specification automatically. Such automatically established trace links are valid (cf. Section 2.1.1) by construction, because the underlying model transformation associates only semantically related trace artifacts with each other based on our mapping rules (cf. Sections 3.4 and 3.8.2.2). Analogously, the lifecycle traceability has to be established manually for the manual parts of our transition technique. We cannot guarantee the validity of such manually established trace links, but we constrain the lifecycle traceability information model in such a way that the chances of manually establishing invalid trace links are minimized (cf. Section 3.9.1.2).

For example, Figure 3.22 depicts an excerpt of the trace links between the structural elements of the EBEAS CONSENS system model and the corresponding MSD specification. Since the structural elements in the MSD specification can be derived automatically, all of the depicted trace links can also be established automatically. Our model transformations associate the interface **PrecrashCommands** to the equally named port specification by means of the trace link of the type Interface2FlowSpecification. Furthermore, they relate the software component types **AdaptiveCruiseControl** and **TrajectoryGeneration** to their corresponding environment/system element templates by means of the trace links :Component2EnvironmentElementTemplate and :Component2SystemElementTemplate, respectively. Analogously, the model transformations link the MSD use case roles to the environment/system element exemplars they are derived from.

Our model transformations only derive MSDs from CONSENS system models in the case that *Behavior – Sequences* are specified. The Software Requirements Engineers have to manually

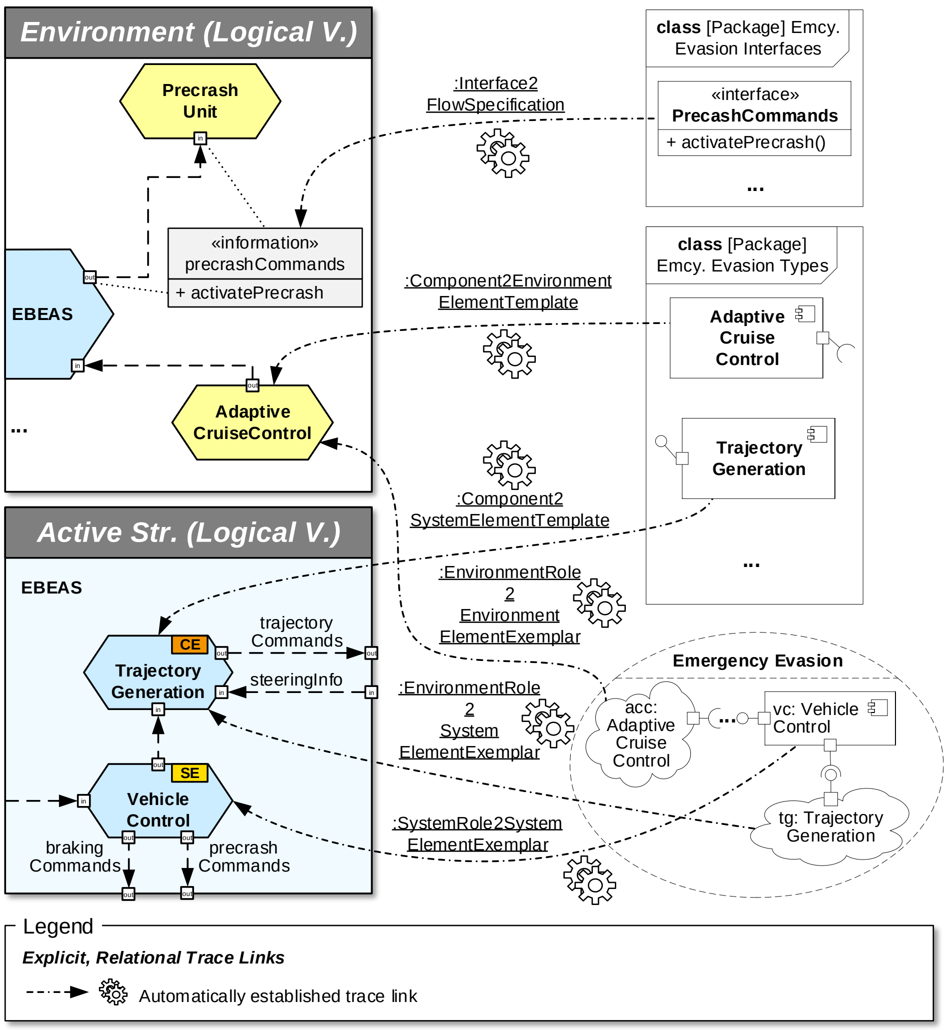

Environment (Logical V.)
Precrash Unit
«information» precrashCommands
+ activatePrecrash
EBEAS
Adaptive CruiseControl
...
Active Str. (Logical V.)
EBEAS
CE
Trajectory Generation
trajectory Commands
steeringInfo
SE
Vehicle Control
braking Commands
precrash Commands
:Interface2 FlowSpecification
:Component2Environment ElementTemplate
:Component2 SystemElementTemplate
:EnvironmentRole 2 Environment ElementExemplar
:EnvironmentRole 2 System ElementExemplar
:SystemRole2System ElementExemplar
class [Package] Emcy. Evasion Interfaces
«interface» PrecashCommands
+ activatePrecrash()
class [Package] Emcy. Evasion Types
Adaptive Cruise Control
Trajectory Generation
Emergency Evasion
acc: Adaptive Cruise Control
vc: Vehicle Control
tg: Trajectory Generation
Legend
Explicit, Relational Trace Links
Automatically established trace link

Figure 3.22: Trace link excerpt between structural elements of the CONSENS system model and of the MSD specification, established automatically

add further MSDs in order to get a complete MSD specification. For such manually added model elements, the Software Requirements Engineers have to likewise add the corresponding trace links manually.

For example, Figure 3.23 depicts an excerpt of the trace links between the non-structural elements of the EBEAS CONSENS system model and the corresponding MSD specification. Our model transformations associate the MSD use cases to the *Application Scenarios* they are derived from by means of :MSDUseCase2ApplicationScenario trace links. Analogously, they relate the MSDs that are initially derived from *Behavior – Sequences* to them by means of :MSD2BehaviorSequence trace links.

If parts of initially derived MSDs are moved to newly created MSDs, the Software Requirements Engineers have to manually establish :MSD2BehaviorSequence trace links to the corresponding *Behavior – Sequences*. For example, this is the case for the MSD NoSafeEmcyBraking in Figure 3.23 (cf. refinement step 2 in Section 3.5.1 and Appendix A.2.2.2). Likewise, they have to manually link MSDs that are newly specified based on the partial model *Requirements*

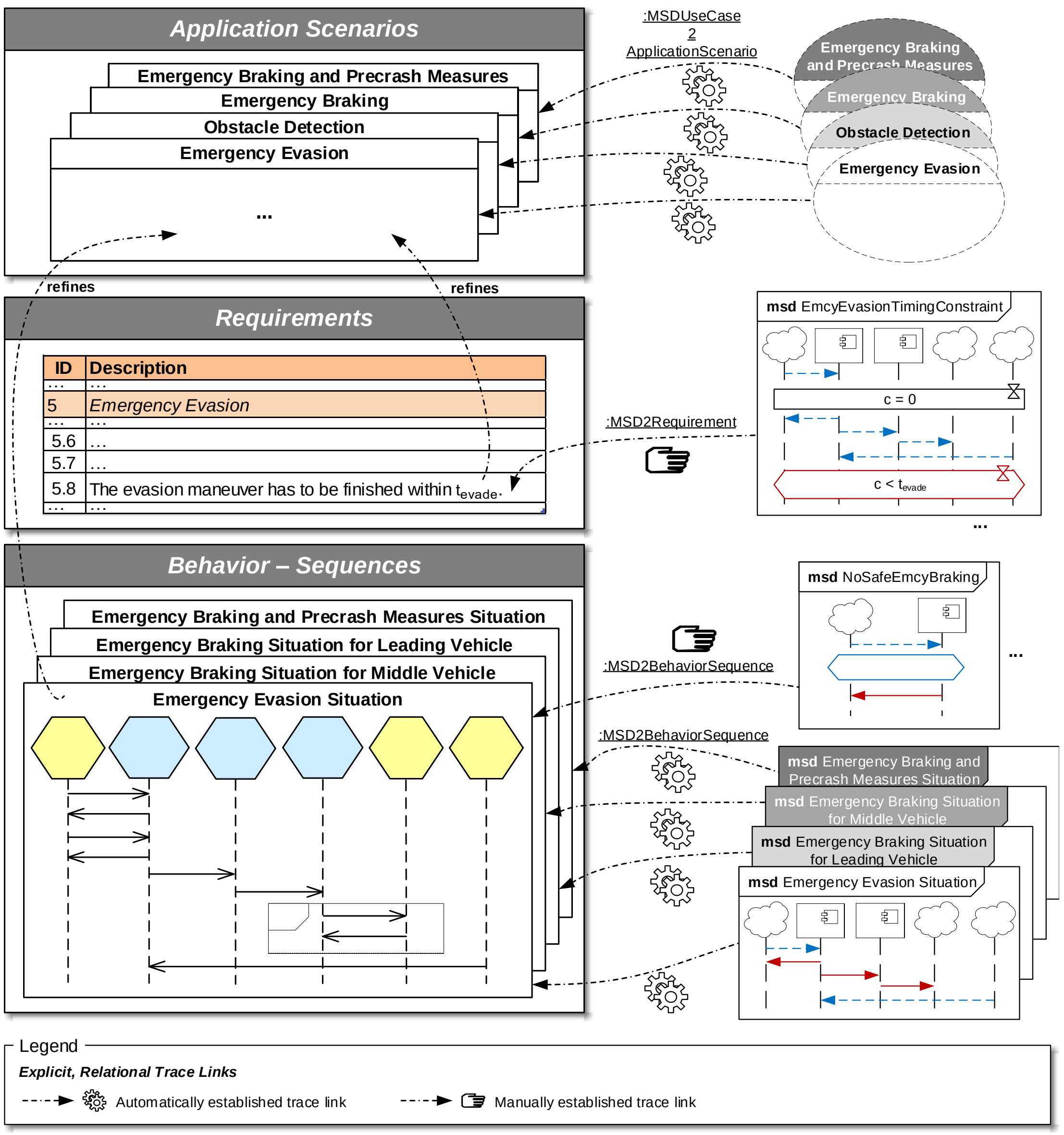

Application Scenarios
Emergency Braking and Precrash Measures
Emergency Braking
Obstacle Detection
Emergency Evasion
...
:MSDUseCase
2
ApplicationScenario
Emergency Braking and Precrash Measures
Emergency Braking
Obstacle Detection
Emergency Evasion
refines
refines
Requirements
ID
Description
...
5
Emergency Evasion
5.6
5.7
5.8
The evasion maneuver has to be finished within t_{evade}.
:MSD2Requirement
msd EmcyEvasionTimingConstraint
c = 0
c < t_{evade}
Behavior – Sequences
Emergency Braking and Precrash Measures Situation
Emergency Braking Situation for Leading Vehicle
Emergency Braking Situation for Middle Vehicle
Emergency Evasion Situation
msd NoSafeEmcyBraking
:MSD2BehaviorSequence
:MSD2BehaviorSequence
msd Emergency Braking and Precrash Measures Situation
msd Emergency Braking Situation for Middle Vehicle
msd Emergency Braking Situation for Leading Vehicle
msd Emergency Evasion Situation
Legend
Explicit, Relational Trace Links
Automatically established trace link
Manually established trace link

Figure 3.23: Trace link excerpt between non-structural elements of the CONSENS system model and of the MSD specification, established semi-automatically

to their corresponding requirement elements by means of :MSD2Requirement trace links. For example, this is the case for the MSD EmcyEvasionTimingConstraint in Figure 3.23 (see also Figure A.25 as part of the exemplary refinement step 4 in Appendix A.2.2.2).

Such manually established trace links can be potentially invalid since the Software Requirements Engineer can erroneously associate the wrong requirement with an MSD, or manually added MSDs can become obsolete and link to formerly related *Behavior – Sequences* or *Requirements*. However, we cannot ensure the validity of manually established trace links in a tool-supported way due to the informal nature of the CONSENS trace artifacts associated with MSD specifications.

As outlined in Section 2.1.1, the granularity of the lifecycle trace links is rather high in contrast to the transformation trace links (cf. Section 3.7.2), as we only associate model elements that are relevant for the management activities conducted by the Systems Engineer and Software Requirements Engineer. For example, we associate CONSENS elements with whole MSDs and not their particular lifelines, messages or their underlying model elements that have no concrete syntax. This is due to the fact that the latter ones are too fine-grained to be meaningful for management activities like impact analyses.

3.7.2 Transformation Traceability

Transformation traceability is one possibility to enable the "Target-Incrementality" feature of model transformation approaches [CH06], which we need for the automatic incremental updates of MSD specifications (cf. Sections 3.6.2.2 and 3.8.2.1). Thus, we establish transformation traceability for the automatic part of our transition technique for this purpose. Like for the lifecycle traceability, we apply CAPRA to this end.

We present the corresponding traceability information model in Section 3.9.1.2. In contrast to the lifecycle traceability information model, this metamodel is very generic since it allows to associate every arbitrary CONSENS model element with each arbitrary MSD specification model element. The model transformation algorithm (cf. Section 3.8.2.2) defines which actual model elements are associated with each other. Furthermore, the transformation trace information model defines no constraints for ensuring the trace link validity. Instead, the trace link validity is completely ensured by the model transformations in a constructive manner.

In the following section, we exemplarily explain how we exploit the transformation trace links to perform incremental updates of MSD specifications that were not subject to manual refinements. We explain how we preserve manual changes to MSD specifications in the incremental update in the subsequent section. Similarly to Section 3.6.2, we apply a variant of Story Diagrams to visualize the changes to the CONSENS system model, the MSD specification, and the transformation traceability model.

3.7.2.1 Incremental Update of not Manually Modified MSD Specifications

Figure 3.24 depicts abstract syntax excerpts of the CONSENS system model, the MSD specification, and the transformation traceability model in terms of our running EBEAS example. The figure exemplarily sketches how the transformation trace links are exploited to perform an incremental update after the CONSENS system model was changed in the case that the affected part of the MSD specification is not manually refined before.

More specifically, the figure shows the particular outcomes of the following four steps (cf. the corresponding four steps in the figure):

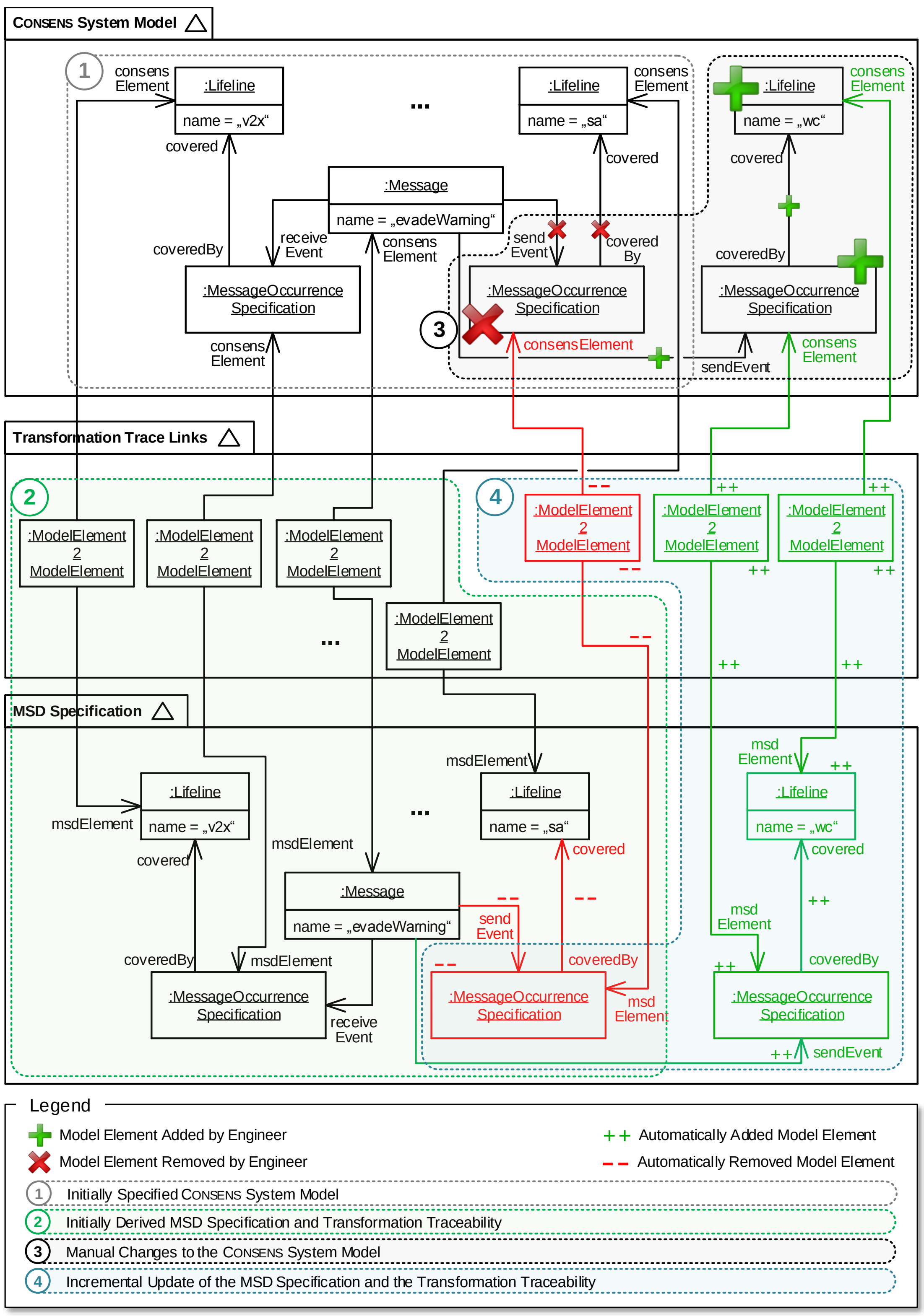

CONSENS System Model
1
consens Element
:Lifeline
name = „v2x“
covered
...
:Lifeline
name = „sa“
consens Element
covered
:Lifeline
name = „wc“
consens Element
covered
:Message
name = „evadeWarning“
coveredBy
receive Event
consens Element
send Event
covered By
coveredBy
:MessageOccurrence Specification
:MessageOccurrence Specification
:MessageOccurrence Specification
3
consens Element
consensElement
sendEvent
consens Element
Transformation Trace Links
2
4
:ModelElement 2 ModelElement
...
MSD Specification
msdElement
:Lifeline
name = „v2x“
covered
msdElement
:Message
name = „evadeWarning“
coveredBy
msdElement
:MessageOccurrence Specification
receive Event
:Lifeline
name = „sa“
covered
send Event
coveredBy
:MessageOccurrence Specification
msd Element
msd Element
:Lifeline
name = „wc“
covered
msd Element
coveredBy
:MessageOccurrence Specification
sendEvent
++
--
Legend
Model Element Added by Engineer
Model Element Removed by Engineer
++ Automatically Added Model Element
-- Automatically Removed Model Element
1 Initially Specified CONSENS System Model
2 Initially Derived MSD Specification and Transformation Traceability
3 Manual Changes to the CONSENS System Model
4 Incremental Update of the MSD Specification and the Transformation Traceability

Figure 3.24: Exploiting transformation traceability during the incremental update of the MSD Emergency Evasion Situation—updating messages and lifelines

1. The Systems Engineer specifies the initial CONSENS system model as presented in Section 2.2 and Section 3.1. The abstract syntax excerpt in the top depicts the corresponding objects and the referential traceability for the lifelines v2x and sa as well as for the message `evadeWarning` sent from sa to v2x.
2. The initial MSD specification as well as the transformation traceability model is automatically derived by our transition technique (cf. Sections 3.4 and 3.6.1.1 to 3.6.1.3). The abstract syntax excerpt in the bottom depicts model elements of this initially derived MSD specification. The abstract syntax excerpt in the center depicts the corresponding :ModelElement2ModelElement transformation trace links associating the particular source and target model elements.
3. The Systems Engineer changes the CONSENS system model as described in Section 3.6.2.1. The abstract syntax excerpt of the situation depicts the part where the Systems Engineer moves the send event of the message `evadeWarning` from the lifeline sa to the newly created lifeline wc (cf. Figure 3.20). That is, a :Lifeline object as well as an adjacent send :MessageOccurrenceSpecification object are created (plus icons), whereas the original send :MessageOccurrenceSpecification is deleted (cross icon).
4. The automatic incremental update of the MSD specification is performed (cf. Section 3.6.2.2).

   First, there are no changes regarding the lifelines v2x and sa as well as the message `evadeWarning` and its receive event in the CONSENS system model. We define the transformation trace links associating such existing source and target objects with the same, unchanged properties (e.g., the names of the lifelines) as *valid*. The incremental update does not perform any actions on the source/target objects and the transformation trace links in this case.

   Second, the send :MessageOccurrenceSpecification in the MSD specification is associated by a :ModelElement2ModelElement trace link. This trace link associates an existing MSD specification element with no corresponding CONSENS system model element due to its deletion in step 3. We define such a transformation trace link and its associated MSD specification element as *source-invalid*. In this case, the incremental update deletes the MSD specification model element as well as the associating trace link.

   Finally, there are no transformation trace links associating the CONSENS system model objects newly created in step 3. We define such source model elements as *unprocessed*. In such a case, the transformation algorithm creates the corresponding objects in their MSD representation as well as the associating transformation trace links (i.e., the :Lifeline and :MessageOccurrenceSpecification objects in the MSD specification as well as the corresponding :ModelElement2ModelElement objects in the transformation traceability model). Furthermore, the referential traceability is established according to the referential traceability in the source model (e.g., the link from the :MessageOccurrenceSpecification object to the :Lifeline object), and the properties of the target objects are set with the values of the corresponding properties of the source objects (e.g., the lifeline name v2x).

3.7.2.2 Preservation of Manual Modifications to MSD Specifications

Figure 3.25 exemplarily sketches how the transformation trace links are exploited to perform an incremental update after the affected part of the MSD specification was manually refined and the CONSENS system model is changed afterward.

More specifically, the figure shows the particular outcomes of the following five steps (cf. the corresponding five steps in the figure):

1. The Systems Engineer specifies the initial CONSENS system model as presented in Section 2.2 and Section 3.1. The abstract syntax excerpt depicts the corresponding objects and the referential traceability for the lifelines v2x and sa as well as for the message `emcyBrakeWarning` sent from v2x to sa.
2. The initial MSD specification as well as the transformation traceability model is automatically derived by our transition technique (cf. Sections 3.4 and 3.6.1.1 to 3.6.1.3). The abstract syntax excerpt in the bottom depicts model elements of this initially derived MSD specification. The abstract syntax excerpt in the center depicts the corresponding :ModelElement2ModelElement transformation trace links associating the particular source and target model elements.
3. The Software Requirements Engineer manually refines the MSD specification and, inter alia, moves the message `emcyBrakeWarning` from the MSD Emergency Evasion Situation to another MSD (cf. step 2 in Section 3.5.1 and Appendix A.2.2.2). That is, the corresponding :Message object as well as the send and receive :MessageOccurrenceSpecification objects are deleted from the depicted abstract syntax excerpt (cross icons).
4. The Systems Engineer changes the CONSENS system model (cf. Figure 3.20 as described in Section 3.6.2.1). The abstract syntax excerpt of the situation depicts the part where the Systems Engineer moves the receive event of the message `emcyBrakeWarning` from the lifeline sa to the newly created lifeline wc (cf. Figure 3.20). That is, a :Lifeline object as well as an adjacent receive :MessageOccurrenceSpecification object are created (plus icons), whereas the original receive :MessageOccurrenceSpecification is deleted (cross icon).
5. The automatic incremental update of the MSD specification is performed (cf. Section 3.6.2.2). There are three transformation trace links that associate CONSENS model elements but have no msdElement link to their corresponding MSD specification element counterpart due to the fact that the corresponding MSD specification elements were manually deleted in step 3. We define such transformation trace links as *target-invalid*. For any target-invalid transformation trace links associating MSD specification elements that are not child elements of MSDs (i.e, structural elements), the transformation algorithm restores the corresponding MSD specification elements.

   However, the incremental update does not perform any changes on target model elements that are part of an MSD (i.e., objects typed by the UML metaclasses **Lifeline**, **Message**, and **MessageOccurrenceSpecification**). This is due to the fact that the Software Requirements Engineer typically conducts manual changes on the MSDs in the course of the MSD specification refinement (cf. Sections 3.5 and 3.6.1.4 and Appendix A.2.2.2). Thus, we preserve the Software Requirements Engineer's changes for target-invalid transformation trace links associating source model elements that would be transformed to target model

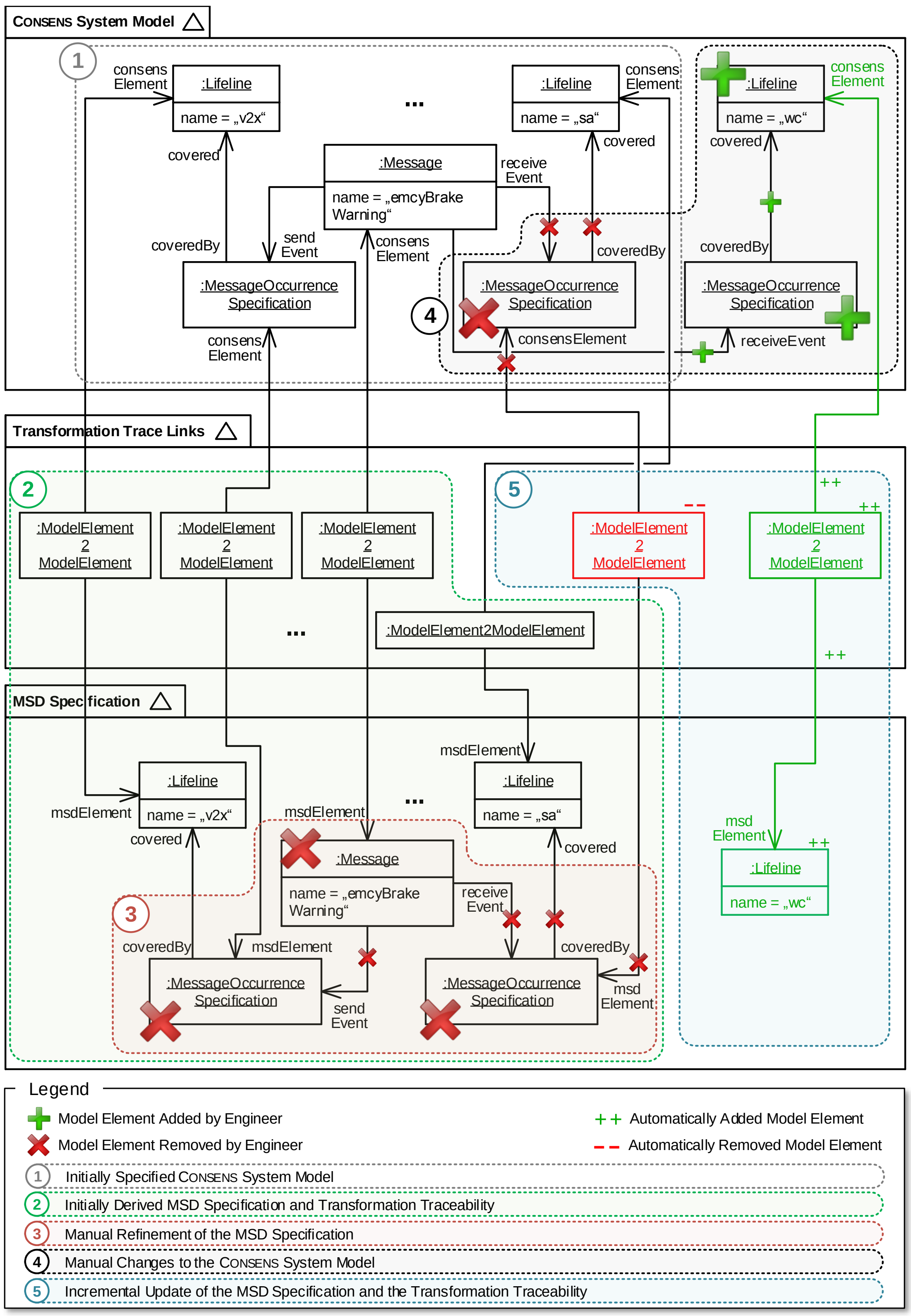

CONSENS System Model
1
consens Element
:Lifeline
name = „v2x“
...
:Lifeline
name = „sa“
consens Element
:Lifeline
name = „wc“
consens Element
covered
:Message
name = „emcyBrake Warning“
receive Event
covered
covered
coveredBy
send Event
consens Element
coveredBy
coveredBy
:MessageOccurrence Specification
4
:MessageOccurrence Specification
:MessageOccurrence Specification
consens Element
consensElement
receiveEvent
Transformation Trace Links
2
5
++
--
++
:ModelElement 2 ModelElement
:ModelElement 2 ModelElement
:ModelElement 2 ModelElement
:ModelElement 2 ModelElement
:ModelElement 2 ModelElement
...
:ModelElement2ModelElement
++
MSD Specification
msdElement
:Lifeline
name = „v2x“
msdElement
covered
...
:Lifeline
name = „sa“
msdElement
:Message
name = „emcyBrake Warning“
covered
msd Element
++
:Lifeline
name = „wc“
3
coveredBy
msdElement
receive Event
coveredBy
:MessageOccurrence Specification
send Event
:MessageOccurrence Specification
msd Element
Legend
Model Element Added by Engineer
Model Element Removed by Engineer
++ Automatically Added Model Element
-- Automatically Removed Model Element
1 Initially Specified CONSENS System Model
2 Initially Derived MSD Specification and Transformation Traceability
3 Manual Refinement of the MSD Specification
4 Manual Changes to the CONSENS System Model
5 Incremental Update of the MSD Specification and the Transformation Traceability

Figure 3.25: Exploiting transformation traceability during the incremental update of the MSD Emergency Evasion Situation—no recreation of once manually moved messages

elements as childs of MSDs (see also the feature “Preservation of User Edits in the Target” of model transformation approaches [CH06]). Hence, the transformation algorithm neither restores the MSD message `emcyBrakeWarning` nor its referenced message occurrence specifications. The algorithm neither deletes the corresponding transformation trace links so that this procedure is repeatable in further transformation executions.

Note that this procedure does not influence the handling of source-invalid transformation trace links or unprocessed source model elements. For example, the message `evadeWarning` that is not subject to manual modification by the Software Requirements Engineer is updated automatically as described in the last paragraph and in Section 3.6.2.2.

However, the transformation algorithm deletes the source-invalid <u>:ModelElement2ModelElement</u> trace link associated with the receive <u>:MessageOccurrenceSpecification</u> that was deleted in the CONSENS model in step 4. This is due to the fact that the trace link would be futile in further executions.

We also define *target-property-invalid* transformation trace links that associate an existing MSD specification element with an existing CONSENS system model element where the value of one or more source element properties has changed (e.g., a name property or an incident referential trace link). In this case, the transformation algorithm updates the obsolete property of the target model element according to the source model changes. This aspect is not part of the example in this section.

3.8 Model Transformations and Coverage Check More Formally

In this section, we describe general aspects of the automatic part of our transition technique more formally. We provide details about the design of the model transformation approach and the model transformation algorithm in Section 3.8.2. Furthermore, we concretize the rules sketched in Section 3.5.2 for the automatic coverage check between MSD specifications and the CONSENS partial model *Behavior – States* in Section 3.8.3. Both the transformation algorithm and the coverage check impose preconditions for the CONSENS system models, which we first of all explain in the following section.

3.8.1 Preconditions for the CONSENS System Model

The automatisms of our transition technique require a thoroughly specified system model in order to be executable. That is, the Systems Engineers have to specify a more detailed and formalized system model than an initial, informal system model for the purpose of system requirements elicitation or the clarification of a coarse-grained system architecture. In terms of the MBSE concept classification of Tschirner et al. [Tsc16; TDBG15], the system model should be specified according to the concept “Mechatronic Systems Modeling” that requires more modeling effort than concepts with lightweight modeling purposes like communication. With such a thoroughly specified system model with the purpose of orchestrating the particular disciplines, the manual effort on conceiving a correct MSD specification is reduced by means of our transition technique (cf. Section 3.9.2). Moreover, further automatisms to facilitate other transition steps like the transition to the software/control engineering design phase (cf. Section 2.3) can be applied. Thus, putting slightly more effort into the specification of system models can significantly reduce the effort on many manual transition steps throughout the whole development process.

We assume that the system model is present in a SYSML4CONSENS model with relevance annotations (cf. Section 3.9.1.1). However, we mainly apply the original CONSENS notation throughout this thesis for illustration purposes. The preconditions on SYSML4CONSENS system models can be transferred to the corresponding constructs of the CONSENS modeling language as we introduced it in Section 2.2 and as we extended it in Section 3.1.

In the following, we explain the preconditions for the particular CONSENS partial models and for the relational traceability between them. These preconditions have to mainly hold for the execution of the transformation algorithm and the coverage rules explained in Section 3.8.2.2 and Section 3.8.3, respectively. However, they also provide means for the Software Requirements Engineers to identify all relevant information during the manual refinement of MSD specifications.

3.8.1.1 Relational Traceability Between Partial Models

As exemplarily illustrated in Section 2.2, the following preconditions for the relational traceability between the particular partial models have to hold:

- Explicit, relational traceability has to be established between the partial models *Environment* and *Application Scenarios*, *Application Scenarios* and *Functions*, *Functions* and *Active Structure*, *Behavior – Sequences* and *Application Scenarios*. Thereby, the transformation algorithm can collect and link the system model information spread over the particular partial models in order to automatically derive an MSD specification.
- Explicit, relational traceability has to be established between the partial models *Requirements* and *Application Scenarios*. Thereby, the Software Requirements Engineer can identify the informal, more detailed *Requirements* corresponding to each particular *Application Scenario*/MSD use case during the manual refinement.

3.8.1.2 *Environment* and *Active Structure*

As exemplarily illustrated in Section 3.1.1, the following preconditions for the structural partial models *Environment* and *Active Structure* have to hold, thereby enabling the transformation algorithm to derive the structural aspects of an MSD specification from a SYSML4CONSENS system model (cf. Section 3.4.2 and Section 3.8.2.2):

- Each system element type must have relevance annotations applied, particularly system element types representing discrete and continuous software component types that shall be transferred to an MSD specification.
- Each environment/system element must be connected via ports to their incident flows, where each port's specification specifies which items can flow over the port[3].
- The port specifications encompassing contents that shall be transferred to interfaces of an MSD specification must have the stereotype «information» applied. Furthermore, the

[3]This is not obligatory for incident mechanical connections since they typically not transfer items. However, it can be meaningful to link mechanical connections with ports if there is a dedicated recess as part of the connection's incident system/environment element.

contents of these interfaces must be specified, as they are specified for the interfaces of an MSD specification.

3.8.1.3 *Behavior – Sequences*

As exemplarily illustrated in Section 3.1.2, referential traceability has to be established from the partial model *Behavior – Sequences* to the partial models *Environment* and *Active Structure*. Thereby, the transformation algorithm is able to derive the behavioral aspects of an MSD specification from a SYSML4CONSENS system model (cf. Section 3.4.3 and Section 3.8.2.2). The following concrete preconditions have to hold:

- Each lifeline has to represent a environment/system element in the partial models *Environment* and *Active Structure*, respectively.
- Each action has to refer to an interface content, thereby specifying which item is transferred or which process is triggered via the action.
- Each action has to refer to a flow, thereby specifying via which flow the action is executed.

3.8.1.4 *Behavior – States*

As explained and exemplarily illustrated in Section 3.1.3, the System Analyst has to specify the partial model *Behavior – States* as an input/output automaton [TL89] for the overall SUD, and referential traceability has to be established to the partial models *Environment* and *Active Structure*. Thereby, our static coverage checks are able to ensure coverage of an MSD specification w.r.t. the *Behavior – States* (cf. Section 3.5.2 and Section 3.8.3). The following preconditions have to hold:

- Transition triggers as well as actions have to refer to an port specification content, thereby specifying which item is transferred or which process is triggered via the trigger/action. A port that is attached to the SUD has to be typed by the encompassing port specification.
- Each transition trigger referring to an SwRE-relevant information flow item additionally has to refer to a port that is typed by the port specification encompassing the flow item. The port has to be attached to the SUD.

3.8.2 Model Transformation Approach and Algorithm

In this section, we outline general aspects on the applied model transformation approach and the model transformation algorithm. We explain the general requirements on the model transformation approach and its resulting selection and extension in the following section. Subsequently, we explain how the model transformation algorithm derives MSD specifications from CONSENS system models and provide details on the incremental update mechanism.

3.8.2.1 Selection and Extension of the Model Transformation Approach

Czarnecki and Helsen categorize different model transformation approaches according to their features [CH06]. From these, the setting for the selection of one of these model transformation approaches for our transition technique requires or implies the following features:

Unidirectional Our transition technique is intentionally unidirectional. We address changes on the system model needed by the discipline experts through the communication between the respective engineering roles in our transition process (cf. Figure 3.6 and Section 3.6.1.6). Thus, a unidirectional model transformation approach to derive MSD specifications from CONSENS system models is sufficient.

Target-Incrementality The unavoidable iterations in the development processes of software-intensive systems require a model transformation approach that supports the "Incrementality" feature. Since we do not need a bidirectional approach, "Target-Incrementality" is sufficient. That is, the model transformation approach can update an existing target model with changes that occurred in the corresponding source model. Such incremental updates are up to now only rarely supported by model transformation tools [KBC+18].

Destructive Source-Target Relationship Changes to the CONSENS system model typically do not only encompass the addition of new model elements but also their deletion (cf. Section 3.6.2.1). Thus, the model transformation approach has to support also the deletion of the corresponding model elements in the MSD specification while updating it (cf. Sections 3.6.2.2 and 3.7.2.1), which is covered by this feature.

Preservation of User Edits in the Target This feature is required because the Software Requirements Engineer is expected to manually refine an initially derived MSD specification (cf. Sections 3.5 and 3.6.1.4). Thus, such an edited target model has to be automatically updated when changes in the CONSENS system models occur without losing the manual changes to the MSD specification (cf. Sections 3.6.2.2 and 3.7.2.2).

Automatic Tracing / Separate Storage Location In order to automatically establish traceability between system models and MSD specifications, the feature "Automatic Tracing" is required. Furthermore, we argue that a separate storage location distinct from system model and MSD specification is well-suited (cf. Section 3.7). Both features are relevant to the lifecycle traceability as well as to the transformation traceability, which enables the latter three features. Built-in traceability is up to now only rarely supported by model transformation tools [KBC+18].

Imperative Logic Czarnecki and Helsen [CH06] distinguish model-to-model transformation approaches into operational and relational ones, inter alia. Rules of operational model transformation approaches are specified through imperative logic similar to programming languages, whereas rules of relational approaches are specified through declarative mappings. We favor an operational approach due to the following reasons. First, the model transformations have to collect information scattered across several CONSENS partial models to derive the particular MSD specification elements. Furthermore, we have to deal with fine-grained exceptions for the "Destructive Source-Target Relationship" feature like the non-modification of once manually moved messages (cf. Section 3.7.2.2). Thus, declarative mapping rules would be cumbersome for such complex and conditional transformations [Bud18]. Furthermore, a declarative model transformation approach has to traverse all elements of the source model (e.g., by non-deterministic graph pattern matching or constraint solving). In contrast, the mapping rules of an operational model transformation approach can be designed such that only the relevant parts of the source model are traversed, which strongly improves the execution performance of the model transformation. Finally, unidirectionality is sufficient for us, and relational model transformation approaches naturally support multidirectionality [CH06].

On the one hand, operational model transformation approaches like Query/View/Transformation-Operational (QVT-O) [OMG16] provide the features "Unidirectional" and "Imperative Logic" by default. The Eclipse implementation [QVTo] of QVT-O also supports the features "Target-Incrementality" and "Automatic Tracing" / "Separate Storage Location" as of version 3.5. However, the combination of the features "Preservation of User Edits in the Target" and "Destructive Source-Target Relationship" is only supported to a certain degree. That is, QVT-O is only capable of additively updating the target model without deleting elements that were manually added to it.

On the other hand, the features "Target-Incrementality" and "Automatic Tracing" / "Separate Storage Location" are fully supported by relational model transformation approaches like Triple Graph Grammars (TGGs) [Sch95]. However, TGGs do not provide "Destructive Source-Target Relationship" in their original form [Sch95]. Several extensions were introduced to TGGs [GW09; Wag09; GH09; Hil14; Anj14; GPR11; Rie15] to cope with the absence of this feature in the context of model-driven engineering. Likewise, the next missing feature "Preservation of User Edits in the Target" including fine-grained exceptions could be expressed in TGGs with extensions like (negative) application conditions (cf. [HHT96] for their general introduction to graph grammars and [GLO09; AVS12; LHGO12] for applications in the context of TGGs) and refinement rules [RS12; Rie15]. Finally, TGGs do not provide the feature "Imperative Logic" so that the application of TGGs in our setting would yield very complex mapping rules in terms of the abstract syntax as well as a poor execution performance.

In order to enable the technical advantages of both QVT-O and TGGs, we combine certain concepts of both approaches. That is, we use QVT-O as basic, unidirectional model transformation approach with imperative logic to specify the processing logic of CONSENS models through a compact algorithm that is executed efficiently. Furthermore, we apply a transformation traceability information model—similarly to the correspondence model of TGGs—to provide the features that are not fully supported by QVT-O (cf. Section 3.9.1.2). The transformation traceability enables to incrementally add or delete target model elements based on additions and deletions in the source model, respectively. Moreover, it allows to preserve user edits in the target model for certain target model elements. Finally, we semi-automatically establish lifecycle traceability based on a dedicated lifecycle traceability information model for the purpose of model management activities (cf. Section 3.9.1.2).

3.8.2.2 Model Transformation Algorithm

Concretizing the coarse-grained description of the model transformation rules in Section 3.4, this section presents our model transformation rules for deriving component-based MSD specifications from CONSENS system models more formally by means of pseudocode algorithms. The algorithms enable the replicability to other MBSE and SwRE approaches as pointed out in the related work section. The system models have to fulfill the preconditions described in Section 3.8.1 for the model transformations to be applicable.

The source models for the transformation algorithm are system models specified in a CONSENS language variant based on the SysML profile SYSML4CONSENS [*KDHM13; IKDN13] in the UML/SysML modeling tool PAPYRUS [PAPYRUS]. The target models are MSD specifications for hierarchical component architectures [*HM13] based on the UML Modal profile provided by the SCENARIOTOOLS MSD [ST-MSD; MUML] tool suite. Since both the source and target models make use of the UML profiling extension mechanism, the approach can also be implemented in other UML/SysML tools.

Algorithm 3.1 presents an abstract pseudocode version of the overall QVT-O transformation. This pseudocode describes in which cases the algorithm considers a CONSENS *Application Scenario* SwRE-relevant. In these cases, the algorithm derives the corresponding MSD use case and calls sub-procedures for deriving the structural MSD specification elements (cf. Algorithm 3.2) and for deriving the actual MSDs (cf. Algorithm 3.3).

Algorithm 3.1: Overall transformation from CONSENS to MSDs (based on [*HBM+16])

```
transformation DERIVEMSDSPECIFICATION (in systemModel, inout msdSpecification, inout transfTraceLinks)
forall appScenario $\in$ systemModel.applicationScenarios do
    discreteSoftwareComponents $\leftarrow$ sysElemExemplar $\in$ systemModel.activeStructure $|$ ($\exists$ funct $\in$ systemModel.functions $|$ (appScenario induces funct $\wedge$ sysElemExemplar realizes funct $\wedge$ sysElemExemplar has relevance annotation “SE”))
    if discreteSoftwareComponents $\neq \emptyset$ then                ▷ Application Scenario is SwRE-relevant
        ▷ Derive MSD use cases (cf. Algorithm 3.4, Section 3.4.1, and Section 3.6.1.1)
        msdUseCase $\leftarrow$ appScenario.deriveCollaboration(transfTraceLinks)
        SwRE-relevantStructuralElements, SwRE-relevantInformationFlows $\leftarrow \emptyset$
        ▷ Derive structure (cf. Algorithm 3.2, Section 3.4.2, and Section 3.6.1.2)
        SwRE-relevantStructuralElements, SwRE-relevantInformationFlows $\leftarrow$ systemModel.deriveStructure ( discreteSoftwareComponents, msdUseCase, transfTraceLinks)
        ▷ Derive MSDs (cf. Algorithm 3.3, Section 3.4.3, and Section 3.6.1.3)
        systemModel.deriveBehavior(SwRE-relevantStructuralElements, SwRE-relevantInformationFlows, msdUseCase, transfTraceLinks)
        msdSpecification += msdUseCase
    end
end
deleteSourceInvalidTraceLinksAndTargetObjects (transfTraceLinks, msdSpecification)        ▷ cf. Algorithm 3.5
```

There are two execution scenarios for the algorithm: The initial generation of an MSD specification and the incremental update of an existing one. The system model is passed as value for the input parameter systemModel in both application scenarios. The MSD specification is represented by the inout parameter msdSpecification, and the intermediate transformation traceability model is contained in the inout parameter transfTraceLinks. Both the MSD specification and the transformation traceability model are `null` in the first execution scenario. In the second execution scenario, msdSpecification as well as transfTraceLinks contain as input value the corresponding model from a prior process iteration before the algorithm execution and the updated corresponding model as output value afterward.

Algorithms 3.1 to 3.3 together refine the coarse-grained transformation rules as sketched in Section 3.4. These algorithms call concrete versions of the generic Algorithm 3.4 for the creation or the update of MSD specification elements. That is, for each <CONSENSElementType> and each corresponding <MSDElementType> there is a concrete algorithm that works according to the functional principle of Algorithm 3.4.

In Section 3.7.2, we exemplify properties of the transformation trace links and transformation trace artifacts for the update execution scenarios. We define these properties more detailed in the following:

- A transformation trace link is *source-invalid*, if it associates an existing MSD specification element with a non-existing CONSENS system model element. This case occurs due to the manual deletion of CONSENS system elements.
- A transformation trace link is *target-invalid*, if it associates an existing CONSENS model element with a non-existing MSD specification element. This case occurs due to the manual deletion of MSD specification elements.

Algorithm 3.2: Procedure for deriving structural elements

```
procedure Model::deriveStructure(in discreteSoftwareComponents: Set(Property), inout msdUseCase: Collaboration, inout transfTraceLinks: Set(TraceLink)) : Set(Property), Set(Connector)
otherSwRE-relevantStructuralElements ← ∅
▷ Derive system component roles from discrete software components (cf. Section 3.4.2.1)
forall discreteSoftwareComponent ∈ discreteSoftwareComponents do
    systemComponentRole ← discreteSoftwareComponent.deriveProperty(transfTraceLinks)    ▷ cf. Algorithm 3.4
    systemComponentRole.specificationKind ← Modal::SpecificationKind::System
    componentType ← discreteSoftwareComponent.type.deriveComponent(transfTraceLinks)    ▷ cf. Algorithm 3.4
    systemComponentRole.type ← componentType
    msdUseCase += systemComponentRole
    msdUseCase += componentType
    ▷ Determine and cache SwRE-relevant environment elements / continuous software components
    forall port ∈ discreteSoftwareComponent.type.ports | (port.type is SysML4CONSENS::InformationFlowSpecification) do
        connectedSwRE-relevantStructuralElement ← port.connectedElement | ((port.connectedElement ∈ self.environment ∧ port.connectedElement affects appScenario) ∨ (port.connectedElement ∈ self.activeStructure ∧ port.connectedElement has relevance annotation “CE” ∧ ∃ funct ∈ self.functions | (appScenario induces funct ∧ port.connectedElement realizes funct))
        otherSwRE-relevantStructuralElements += connectedSwRE-relevantStructuralElement
        SwRE-relevantInformationFlows += informationFlow between discreteSoftwareComponent and connectedSwRE-relevantStructuralElement
    end
end
▷ Derive environment component roles from SwRE-relevant environment elements / continuous SW components (cf. Section 3.4.2.2 and Section 3.4.2.3)
forall SwRE-relevantStructuralElemExemplar ∈ otherSwRE-relevantStructuralElements do
    environmentComponentRole ← SwRE-relevantStructuralElemExemplar.deriveProperty(transfTraceLinks)    ▷ cf. Algorithm 3.4
    environmentComponentRole.specificationKind ← Modal::SpecificationKind::Environment
    componentType ← SwRE-relevantStructuralElemExemplar.deriveComponent(transfTraceLinks)    ▷ cf. Algorithm 3.4
    environmentComponentRole.type ← componentType
    msdUseCase += environmentComponentRole
    msdUseCase += componentType
end
▷ Derive connectors from information flows between already processed SwRE-relevant structural elements (cf. Section 3.4.2.4)
forall informationFlow ∈ SwRE-relevantInformationFlows do
    forall connectorEnd ∈ informationFlow.flow.end do
        ▷ Multiple calls of <ConsensElement>.derive<MSDElement> (cf. Algorithm 3.4) on the same object return the already derived and unaltered MSD element
        alreadyDerivedComponentType ← connectorEnd.partWithPort.owner.deriveComponent(transfTraceLinks)    ▷ cf. Algorithm 3.4
        port ← connectorEnd.partWithPort.derivePort(transfTraceLinks)    ▷ cf. Algorithm 3.4
        interface ← connectorEnd.partWithPort.type.deriveInterface(transfTraceLinks)    ▷ cf. Algorithm 3.4
        forall informationFlowItem in connectorEnd.partWithPort.type do
            interface += informationFlowItem.deriveOperation(transfTraceLinks)    ▷ cf. Algorithm 3.4
        end
        port.type ← interface
        alreadyDerivedComponentType += port
        msdUseCase += interface
    end
    msdUseCase += informationFlow.deriveConnector(transfTraceLinks)    ▷ cf. Algorithm 3.4
end
return discreteSoftwareComponents ∪ otherSwRE-relevantStructuralElements, SwRE-relevantInformationFlows
```

Algorithm 3.3: Procedure for deriving MSDs

```
procedure Model::deriveBehavior(in SwRE-relevantStructuralElements: Set(Property), in SwRE-relevantInformationFlows: Set(Connector), inout msdUseCase: Collaboration, inout transfTraceLinks: Set(TraceLink))
forall behaviorSequence $\in$ self.behaviorSequences | (behaviorSequence refines appScenario) do
    msd $\leftarrow$ behaviorSequence.deriveInteraction(transfTraceLinks)    $\triangleright$ cf. Algorithm 3.4
    SwRE-relevantLifelines $\leftarrow \emptyset$
    forall behaviorSequenceLifeline $\in$ behaviorSequence.lifelines | (behaviorSequenceLifeline.represents $\in$ SwRE-relevantStructuralElements) do
        SwRE-relevantLifelines += behaviorSequenceLifeline.deriveLifeline(transfTraceLinks)    $\triangleright$ cf. Algorithm 3.4
    end
    msd += relevantLifelines
    forall behSeqMsg $\in$ behaviorSequence.messages | (behSeqMsg.sender $\in$ SwRE-relevantLifelines $\wedge$ behSeqMsg.receiver $\in$ SwRE-relevantLifelines $\wedge$ behSeqMsg.connector $\in$ SwRE-relevantInformationFlows $\wedge$ behSeqMsg.signature.owner is SysML4CONSENS::InformationFlowSpecification) do
        msd += behSeqMsg.deriveMessage(transfTraceLinks)    $\triangleright$ cf. Algorithm 3.4
    end
    msdUseCase += msd
end
```

Algorithm 3.4: Generic procedure for creating or updating an MSD specification element

```
procedure <CONSENSElementType>::derive<MSDElementName> (inout transfTraceLinks: Set(TraceLink)) : <MSDElementType>
msdElementToReturn: <MSDElementType> $\leftarrow$ null
if $\exists$ transformationTraceLink $\in$ transfTraceLinks | (transformationTraceLink.consensElement = self) then
    $\triangleright$ Source element exists already
    if transformationTraceLink.msdElement $\neq$ null then $\triangleright$ Trace link is target-invalid
        if transformationTraceLink.msdElement.type is UML::Lifeline $\vee$ UML::Message $\vee$ UML::MessageOccurrenceSpecification then $\triangleright$ MSD specification element is child of an MSD
            $\triangleright$ Do nothing, so that this algorithm returns null and the calling transformation adds nothing to the corresponding list
        end
        else $\triangleright$ Target object has to be restored
            msdElementToReturn $\leftarrow$ self.map <CONSENSElementType> 2<MSDElementType> ()
            transformationTraceLink.msdElement $\leftarrow$ msdElementToReturn
        end
    end
    else $\triangleright$ Trace link is potentially target-property-invalid
        msdElementToReturn $\leftarrow$ transformationTraceLink.msdElement
        synchronizeProperties(msdElementToReturn, self)
    end
end
else $\triangleright$ Source element is unprocessed $\Rightarrow$ create target object and link both with each other
    msdElementToReturn $\leftarrow$ self.map <CONSENSElementType> 2<MSDElementType> ()
    transformationTraceLink $\leftarrow$ new ModelElement2ModelElement()
    transformationTraceLink.msdElement $\leftarrow$ msdElementToReturn
    transformationTraceLink.consensElement $\leftarrow$ self
    transfTraceLinks += transformationTraceLink
end
return msdElementToReturn
```

- A transformation trace link is *target-property-invalid*, if it associates a CONSENS model element with an MSD specification element where at least one property of the MSD specification element is not synchronized with the corresponding CONSENS model element property. This case occurs due to the manual change of the property of MSD specification elements (e.g., a name change).
- A transformation trace source artifact is *unprocessed*, if there is no transformation trace link that associates a CONSENS system model element that is input to the transformation algorithm with an MSD specification element. This case occurs due to the manual addition of CONSENS system elements.

The functional principle of Algorithm 3.4 describes in which cases a transformation trace link is considered target-(property-)invalid and a trace source artifact is considered unprocessed. Particularly, the principle defines which actions have to be performed in these cases (cf. Section 3.7.2 for exemplary executions of the algorithm). The called `map` operations represent the QVT-O mappings for the particular source and target element types. The called operation `synchronizeProperties(<MSDElementType>, <CONSENSElementType>)` synchronizes all relevant properties of the particular source and target element types. We do not present these operations here due to their simplicity.

At the end, Algorithm 3.1 calls Algorithm 3.5 to delete all remaining source-invalid transformation trace links and target objects. The functional principle of this algorithm describes in which cases a transformation trace link is considered source-invalid. Such objects are removed from the transformation traceability model and the MSD specification, respectively (cf. Section 3.7.2 for exemplary executions of the algorithm).

Algorithm 3.5: Procedure for deleting source-invalid transformation trace links and MSD specification elements

```
procedure deleteSourceInvalidTraceLinksAndTargetObjects (inout transfTraceLinks: Set(TraceLink), inout msdSpecification: Model)
forall transformationTraceLink ∈ transfTraceLinks | (transformationTraceLink.consensElement = null) do
    ▷ Source element does not exist ⇒ transformationTraceLink is source-invalid
    if transformationTraceLink.msdElement ≠ null then ▷ Source-invalid MSD specification element exists
        msdSpecification.remove(transformationTraceLink.msdElement) ▷ Remove MSD specification element
    end
    transfTraceLinks.remove(transformationTraceLink) ▷ Remove source-invalid transformation trace link
end
```

Although we exploit the relevance annotations similarly to Rieke [Rie15] as described in Section 2.3, he points out that the mappings covering the purely structural aspects in the automatic derivation of discipline-specific design models from CONSENS system models "are rather straightforward" [Rie15]. In contrast, our transformation for this automatic part is more complex due to the facts that source and target models are structurally more different and the SwRE-relevant information in CONSENS is spread across several partial models and hence has to be collected through trace links.

3.8.3 Coverage Check between MSD Specifications and *Behavior – States*

Concretizing the coarse-grained description of the rules for automatically checking an MSD specification for coverage w.r.t. the CONSENS partial model *Behavior – States* in Section 3.5.2, we describe these rules in this section more formally. The partial model *Behavior – States* has to adhere to the preconditions for CONSENS system models as described in Section 3.8.1. We exemplarily explain the violation of one of these coverage rules and a corresponding correction in step 5a of the manual refinement of the EBEAS MSD specification in Appendix A.2.2.2.

We provide checks for static coverage rules to reveal simple coverage violations but no formal refinement verification approach (e.g., proving a simulation relationship [Mil71; HBDS15] between the MSD stategraph and the *Behavior – States*) for two reasons. First, both the System Analyst and the Software Requirements Engineer would be strongly constrained in their particular specification freedom if they had to conceive their respective models in such a way that the stategraph resulting from the MSD specification fulfills a formal refinement relationship w.r.t the *Behavior – States*. Second, the partial model *Behavior – States* describes the interdisciplinary behavior of the overall system, and hence only certain parts of it are SwRE-relevant (e.g., the state Emergency Evasion including its do-operation and its self-transition is only relevant to the Control Engineer). This would strongly complicate a formal refinement check since the overall automaton could be partitioned into several disjoint SwRE-relevant parts.

Thus, our static checks provide no guarantee that the MSD specification correctly refines the SwRE-relevant parts of the *Behavior – States*. In fact, checking the coverage rules reveals whether there is at least one corresponding model element in the MSD specification for certain SwRE-relevant model elements in the *Behavior – States* and vice versa. Ensuring that there is at least one corresponding model element representation for each model element counterpart in the other model is optimistic, that is, there are potentially false positives. Nevertheless, information is missing or superfluous in the MSD specification for sure if one of the coverage rules is violated. Thus, the static checks improve the completeness and conciseness of the respective models.

In the following, we present the two sets of static coverage rules. The first one has the aim to improve the completeness of the MSD specification w.r.t. to the *Behavior – States*. The second one aims at improving the completeness of the *Behavior – States* w.r.t. the MSD specification and the conciseness of the MSD specification w.r.t. the *Behavior – States*. All coverage rules are specified by means of a simplified pseudocode variant of the OCL [OMG14a].

3.8.3.1 Rule Set 1: Check Whether Each SwRE-relevant Trigger/Effect in the *Behavior – States* is Represented in any Requirement MSD

This rule set has the aim to improve the completeness of the MSD specification w.r.t. to the *Behavior – States*. It checks whether there is at least one corresponding model element in the MSD specification for certain SwRE-relevant model elements in the *Behavior – States*.

Algorithm 3.6 presents the coverage rule for checking whether there is at least one corresponding environment message in any requirement MSD for each SwRE-relevant trigger in the *Behavior – States*. We consider a *Behavior – States* trigger SwRE-relevant iff it has a referential trace link operation to an operation defined in an information flow specification as well as a referential trace link port to a system template's port typed by the information flow specification, where the port is connected to an SwRE-relevant environment element (cf. Section 3.1.3).

Algorithm 3.7 presents the coverage rule for checking whether there is at least one corresponding system message sent to the environment in any requirement MSD for each SwRE-relevant effect on an SwRE-relevant environment element in the *Behavior – States*. We consider a *Behavior – States* effect of a transition or the entry-/do-/exit-operation of a state SwRE-relevant iff it has a referential trace link specification to an operation defined in an information flow

Algorithm 3.6: Coverage rule for ensuring that each SwRE-relevant trigger in the *Behavior – States* is represented by at least one environment message in any requirement MSD

▷ The set of SwRE-relevant environment message triggers contains triggers with the following properties:

- The trigger's event is a UML4SysML::CallEvent.
- The corresponding operation is defined in a SysML4CONSENS::InformationFlowSpecification.
- The corresponding port is owned by the system template and connected to an SwRE-relevant environment element exemplar.

For any of such SwRE-relevant environment message triggers in the *Behavior – States*, there must be an environment message in any requirement MSD with the following properties:

- The message corresponds to an operation named equally to the corresponding operation of any of the triggers defined above.
- This operation has to be defined within an interface that is derived from the InformationFlowSpecification encompassing the trigger's operation.
- The MSD message has to be sent by an environment role that is derived from an environment element exemplar causing the message event trigger.

```
context UML4SysML::Trigger inv:
self.owner is UML4SysML::Transition ∧ self.event is UML4SysML::CallEvent
  ∧ ∃ flowSpecification ∈ systemModel.flowSpecifications | (((UML4SysML::CallEvent)
  self.event).operation.owner = flowSpecification ∧ flowSpecification is
  SysML4CONSENS::InformationFlowSpecification
  ∧ self.port.type = flowSpecification ∧ self.port.owner is SysML4CONSENS::SystemTemplate
  ∧ self.port.owner is connected to envElem ∈ systemModel.environment)
  implies
  ∃ message ∈ msdSpecification.MSDs.messages | (message is environment message ∧
  message.owner not is Modal::EnvironmentAssumption ∧
  message.signature.name = ((UML4SysML::CallEvent) self.event).operation.name ∧
  ∃ :Interface2FlowSpecification trace link from message.signature.owner to flowSpecification ∧
  ∃ :EnvironmentRole2EnvironmentElementExemplar trace link from message.connector.connectedSourceElement
  to self.port.connectorEnd.connector.connectedSourceElement)
```

specification that types a port of an environment template, where the port is connected to an SwRE-relevant environment element (cf. Section 3.1.3).

We do not define a coverage rule specifying that each trigger with a relative time event `after(`*<time>*`)` in the *Behavior – States* is represented by a cold time condition with the expression *clockname* > *<time>*. For example, the transition in Figure 2.2 leading from the state Following Coordination to Overtaking Coordination has a trigger with the relative time event `after(`$t_{followingCoord}$`)`. This is due to the fact that triggers with time events are not specific to any engineering discipline and have no referential traceability to the system architecture (e.g., a port). Thus, their SwRE-relevance cannot be determined.

3.8.3.2 Rule Set 2: Check Whether Each MSD Message Sent from/to the Environment in a Requirement MSD is Represented in the *Behavior – States*

This rule set aims at improving the completeness of the *Behavior – States* w.r.t the MSD specification and the conciseness of the MSD specification w.r.t. the *Behavior – States* (i.e., to detect superfluous aspects in the MSD specification). It checks whether there is at least one corre-

Algorithm 3.7: Coverage rule for ensuring that each SwRE-relevant action on an SwRE-relevant environment element in the *Behavior – States* is represented by at least one system message sent to the environment in any requirement MSD

▷ The set of *Behavior – States* SwRE-relevant actions on the environment contains opaque behaviors with the following properties:

- The opaque behavior belongs to a *Behavior – States* transition or state operation and references a UML4SysML::Operation.
- This operation is defined in a SysML4CONSENS::InformationFlowSpecification.
- The port typed by this flow specification is owned by an SwRE-relevant environment element template and connected to the system exemplar.

For any of such SwRE-relevant actions on the environment in the *Behavior – States*, there must be a system message in any requirement MSD with the following properties:

- The message corresponds to an operation named equally to the operation referenced by the opaque behavior.
- This operation has to be defined within an interface that is derived from the InformationFlowSpecification encompassing the operation referenced by the opaque behavior.
- The MSD message has to be sent to an environment role that is derived from an environment element exemplar providing the operation referenced by the opaque behavior.

```
context UML4SysML::OpaqueBehavior inv:
(self.owner is UML4SysML::Transition ∨ self.owner is UML4SysML::State)
  ∧ self.specification is UML4SysML::Operation ∧ ∃ flowSpecification ∈ systemModel.flowSpecifications |
  (self.specification.owner = flowSpecification ∧
  flowSpecification is SysML4CONSENS::InformationFlowSpecification ∧
  ∃ port ∈ systemModel.environment | (port.type = flowSpecification ∧
  port.owner is SysML4CONSENS::EnvironmentElementTemplate ∧
  port is connected to the systemModel.systemExemplar))
  implies
  message ∈ msdSpecification.MSDs.messages | (message is system message ∧
  message.owner not is Modal::EnvironmentAssumption ∧
  message.signature.name = self.specification.name ∧
  ∃ :Interface2FlowSpecification trace link from message.signature.owner to flowSpecification) ∧
  ∃ :EnvironmentRole2EnvironmentElementExemplar trace link from
  message.connector.connectedTargetElement to port.owner
```

sponding SwRE-relevant trigger or action in the *Behavior – States* for environment messages or system messages sent to the environment in the MSD specification.

If the *Behavior – States* are not specified, the checks return a positive result so that the *Behavior – States* specification is optional. Furthermore, the *Behavior – States* specify the behavior w.r.t. inputs and outputs between the system under development and the environment (cf. Section 3.1.3). Since we also derive environment roles from system-internal SwRE-relevant continuous software components, we have to ensure in this rule set that we exclude these and only consider environment roles derived from SwRE-relevant environment elements.

Algorithm 3.8 presents the coverage rule for checking whether there is at least one SwRE-relevant trigger in the *Behavior – States* for each environment message in any requirement MSD. Algorithm 3.9 presents the coverage rule for checking whether there is at least one SwRE-relevant effect on a SwRE-relevant environment element in the *Behavior – States* for each system message sent to the environment in any requirement MSD.

Algorithm 3.8: Coverage rule for ensuring that each environment message in any requirement MSD is represented by at least one SwRE-relevant environment message trigger in the *Behavior – States*

▷ For any environment message not sent from an environment role derived from a continuous software component (i.e., the sender is derived from an SwRE-relevant environment element) in any requirement MSD, there must be an SwRE-relevant environment message trigger in the *Behavior – States* (if specified) with the following properties:

- The trigger's event is a UML4SysML::CallEvent.
- The trigger corresponds to an operation named equally to the corresponding operation of any of the environment messages in a requirement MSD.
- This operation has to be defined within an interface, from which an InformationFlowSpecification encompassing the message's operation is derived.
- The trigger has to be sent by an SwRE-relevant environment element exemplar, from which an environment element exemplar sending the message is derived.

```
**context** UML::Message **inv**:
$\exists$ systemModel.behaviorStates $\wedge$ self is Modal::ModalMessage $\wedge$ message is environment message $\wedge$
  **not** message.connector.sender.transformationTraceLink.consensStructuralElement has relev. annotation “CE” $\wedge$
  **not** message.owner is Modal::EnvironmentAssumption
  **implies**
  $\exists$ trigger $\in$ systemModel.behaviorStates.transitions.triggers | (trigger.event is UML4SysML::CallEvent $\wedge$
  self.signature.name = ((UML4SysML::CallEvent) trigger.event).operation.name $\wedge$
  $\exists$ :Interface2FlowSpecification trace link from
  self.signature.owner to ((UML4SysML::CallEvent) trigger.event).operation.owner $\wedge$
  $\exists$ :EnvironmentRole2EnvironmentElementExemplar trace link from
  self.connector.connectedSourceElement to trigger.port.connectorEnd.connector.connectedSourceElement)
```

Algorithm 3.9: Coverage rule for ensuring that each system message sent to the environment in any requirement MSD is represented by at least one SwRE-relevant action on an SwRE-relevant environment element in the *Behavior – States*

▷ For any system message sent to an environment role not derived from a continuous software component (i.e., it is derived from an SwRE-relevant environment element) in any requirement MSD, there must be an SwRE-relevant action on an environment element in the form of a transition effect or a state operation in the *Behavior – States* (if specified) with the following properties:

- The action is an opaque behavior belonging to a *Behavior – States* transition or state operation and references a UML4SysML::Operation.
- This operation is named equally to the message's signature.
- This operation has to be defined within a SysML4CONSENS::InformationFlowSpecification, from which an interface encompassing the message's signature is derived.

```
**context** UML::Message **inv**:
$\exists$ systemModel.behaviorStates $\wedge$ self is Modal::ModalMessage $\wedge$ self is system message $\wedge$
  message.owner **not** is Modal::EnvironmentAssumption $\wedge$
  message.connector.receiver is environment role $\wedge$
  **not** message.connector.receiver.transformationTraceLink.consensStructuralElement has relev. annotation “CE”
  **implies**
  $\exists$ behavior $\in$ (systemModel.behaviorStates.transitions.effects $\cup$ systemModel.behaviorStates.states.operations) |
  (behavior is UML4SysML::OpaqueBehavior $\wedge$ behavior.specification **not** is null $\wedge$
  behavior.specification is UML4SysML::Operation $\wedge$ self.signature.name = behavior.specification.name $\wedge$
  $\exists$ :Interface2FlowSpecification trace link from self.signature.owner to behavior.specification.owner) $\wedge$
  $\exists$ port $\in$ systemModel.environment | ($\exists$ :EnvironmentRole2EnvironmentElementExemplar trace link from
  self.connector.connectedTargetElement to port.owner $\wedge$ port.type = behavior.specification.owner))
```

3.9 Realization and Evaluation

We present the implementation aspects for the concepts described throughout this chapter in Section 3.9.1 and describe the conduct of a case study to evaluate the concepts in Section 3.9.2.

3.9.1 Implementation

Figure 3.26 depicts the coarse-grained software architecture that realizes the concepts described in this chapter. The architecture visualization encompasses the components and UML profiles newly implemented in the course of this thesis, the existing and hence reused frameworks, tool suites, and UML profiles, as well as the dependencies between these components. The overall implementation bases on the Eclipse Modeling Framework (EMF) [EMF] and hence applies the component **EMF** as root component.

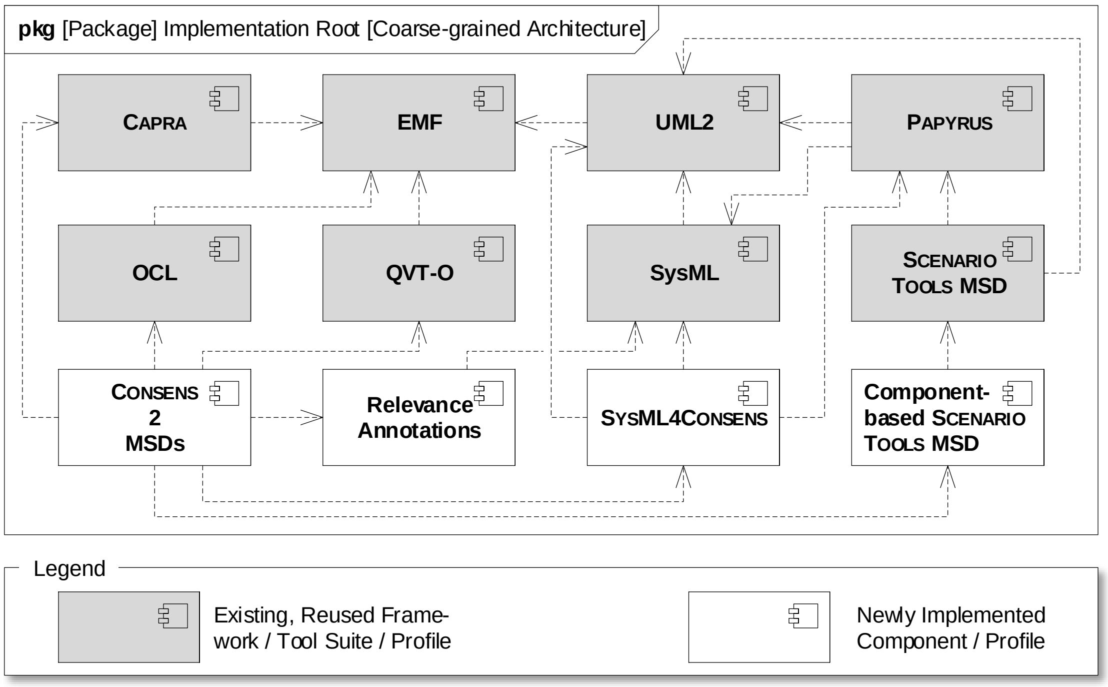

pkg [Package] Implementation Root [Coarse-grained Architecture]
CAPRA
EMF
UML2
PAPYRUS
OCL
QVT-O
SysML
SCENARIO TOOLS MSD
CONSENS 2 MSDs
Relevance Annotations
SYSML4CONSENS
Component-based SCENARIO TOOLS MSD
Legend
Existing, Reused Framework / Tool Suite / Profile
Newly Implemented Component / Profile

Figure 3.26: Coarse-grained architecture of the implementation and the reused components

The profiles **Relevance Annotations** and **SysML4Consens** that we present in Section 3.9.1.1 both extend the existing UML profile **SysML** and hence depend on it. **SysML4Consens** has further dependencies to **Papyrus** due to its contributions to the user interface and to the diagrammatic visualization as well as to **UML2** due to metaclass extensions. The component **Component-based ScenarioTools MSD** extends the tool suite **ScenarioTools MSD** with hierarchical software components as structural basis for MSDs.

The component **Consens2MSDs** implements the transformation algorithm as described in Section 3.8.2.2 through a set of QVT-O mappings including Java black-box libraries. Furthermore, it encompasses the Capra traceability information models (cf. Section 3.7) that we present in Section 3.9.1.2. The transformation algorithm takes SysML4Consens models with relevance annotations as well as potential Capra trace links as inputs, and it outputs component-based MSD specifications as well as newly created or updated Capra trace links.

Thus, the component has dependencies to the three newly implemented components described in the last paragraph, to the **QVT-O** framework [QVTo], and to the traceability management tool **Capra** [Capra]. The static coverage checks between MSD specifications and the Consens partial model *Behavior – States* as described in Section 3.8.3 by means of pseudocode are currently not yet realized. One option for their realization is the implementation by means of the **OCL** framework, resulting in the dependency to this component.

In Section 3.9.1.1, we present the SysML profiles **SysML4Consens** and **Relevance Annotations**. We subsequently present the Capra traceability information models as part of the component **Consens2MSDs** in Section 3.9.1.2.

3.9.1.1 SysML Profiles

In this section, we present the SysML profiles that we introduce in this thesis to provide a solid basis for the automatic part of our transition technique presented in this chapter. We implemented them in the UML/SysML open source modeling tool Papyrus.

In the following, we present the SysML profile SysML4Consens and subsequently the SysML profile for the relevance annotations. Finally, we exemplify the application of both profiles.

SysML4Consens

The SysML4Consens profile depicted in Figure 3.27 is a *language profile* [SG14], which specifies a stand-alone language by augmenting the SysML language profile (cf. Section 2.5.2) with concepts of the Consens modeling language.

The main part of the profile focuses on extending the SysML language w.r.t. the structural modeling elements, that is, elements that define the partial models *Environment* and *Active Structure*. In terms of these structural modeling elements, SysML and its base language UML explicitly distinguish between types/classifiers and their roles/parts. The UML subset UML4SysML, which is the basis for the actual SysML profile, captures these concepts by means of the metaclasses **UML4SysML::Property** and **UML4SysML::Class**, respectively. SysML further specializes the latter metaclass by extending it through the stereotype **SysML::Blocks::Block**, which represents a generic structural element type.

The Consens modeling language captures types and roles by means of templates and exemplars, respectively (cf. Section 2.2.5). Furthermore, Consens distinguishes between the SUD, its system elements, and its environment elements (cf. Section 2.2.1 and Section 2.2.5). To specialize SysML with these concepts, we hence refine the stereotype **SysML::Blocks::Block** and extend the metaclass **UML4SysML::Property** with stereotypes representing this terminology. First, we refine **SysML::Blocks::Block** with the stereotypes **SystemTemplate**, **SystemElementTemplate**, and **EnvironmentElementTemplate** to provide modeling means for templates of the system, of system elements, and of environment elements, respectively. Second, we extend **UML4SysML::Property** with the stereotypes **SystemExemplar**, **SystemElementExemplar**, and **EnvironmentElementExemplar** to provide modeling means for exemplars of the system, of system elements, and of environment elements, respectively. By specializing these structural modeling constructs of SysML, we also inherit their modeling means for ports without defining dedicated stereotypes for them. That is, the user of the profile can attach ports to the system as well as to system and environment elements. We ensure that the particular exemplar kinds have the corresponding template kind as block type by means of OCL constraints.

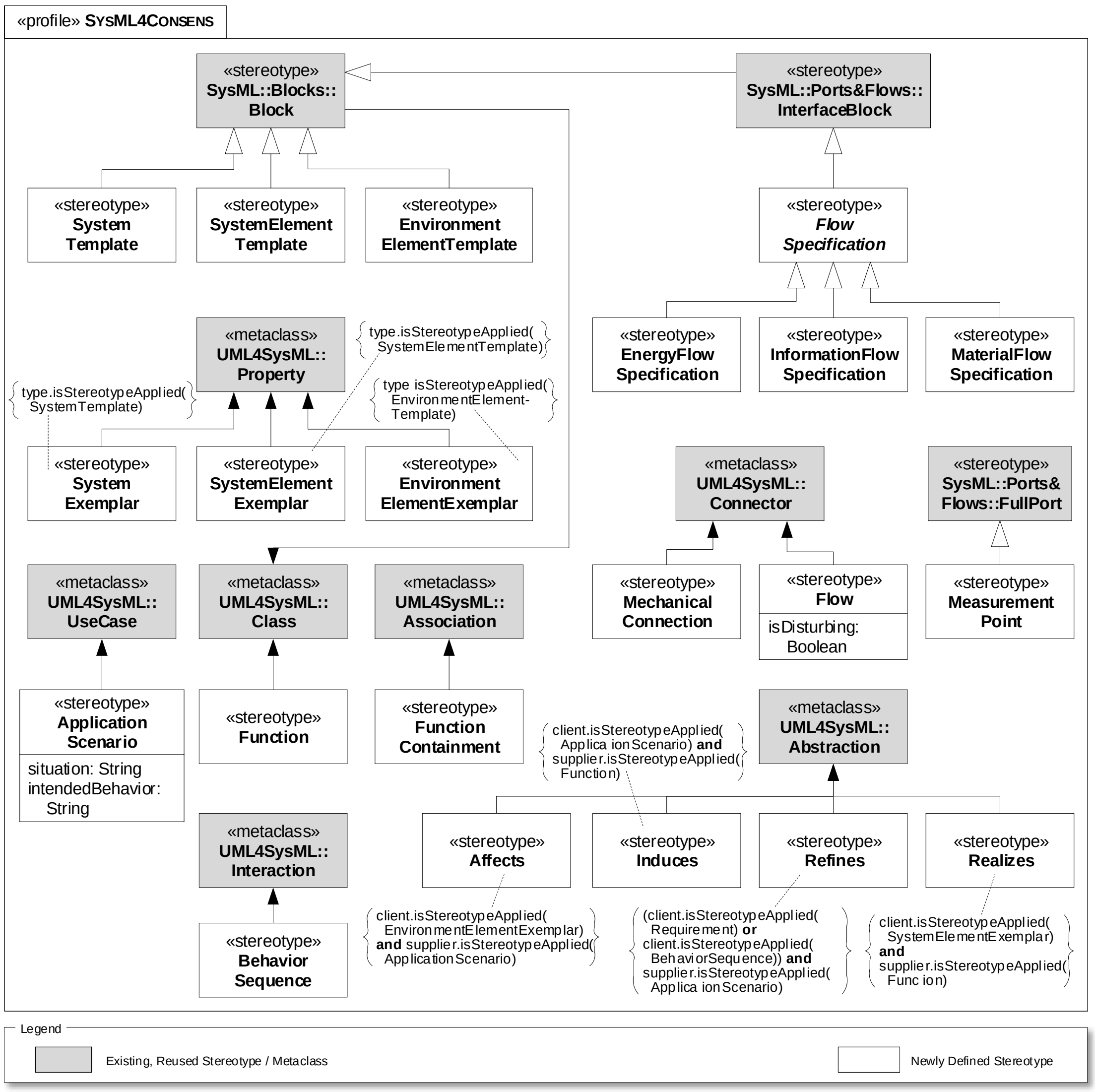

«profile» SysML4Consens
«stereotype» SysML::Blocks:: Block
«stereotype» SysML::Ports&Flows:: InterfaceBlock
«stereotype» System Template
«stereotype» SystemElement Template
«stereotype» Environment ElementTemplate
«stereotype» Flow Specification
«metaclass» UML4SysML:: Property
type.isStereotypeApplied(SystemElementTemplate)
type.isStereotypeApplied(SystemTemplate)
type isStereotypeApplied(EnvironmentElement- Template)
«stereotype» EnergyFlow Specification
«stereotype» InformationFlow Specification
«stereotype» MaterialFlow Specification
«stereotype» System Exemplar
«stereotype» SystemElement Exemplar
«stereotype» Environment ElementExemplar
«metaclass» UML4SysML:: Connector
«stereotype» SysML::Ports& Flows::FullPort
«metaclass» UML4SysML:: UseCase
«metaclass» UML4SysML:: Class
«metaclass» UML4SysML:: Association
«stereotype» Mechanical Connection
«stereotype» Flow
isDisturbing: Boolean
«stereotype» Measurement Point
«stereotype» Application Scenario
situation: String
intendedBehavior: String
«stereotype» Function
«stereotype» Function Containment
client.isStereotypeApplied(ApplicationScenario) and supplier.isStereotypeApplied(Function)
«metaclass» UML4SysML:: Abstraction
«metaclass» UML4SysML:: Interaction
«stereotype» Affects
«stereotype» Induces
«stereotype» Refines
«stereotype» Realizes
«stereotype» Behavior Sequence
client.isStereotypeApplied(EnvironmentElementExemplar) and supplier.isStereotypeApplied(ApplicationScenario)
(client.isStereotypeApplied(Requirement) or client.isStereotypeApplied(BehaviorSequence)) and supplier.isStereotypeApplied(ApplicationScenario)
client.isStereotypeApplied(SystemElementExemplar) and supplier.isStereotypeApplied(Function)
Legend
Existing, Reused Stereotype / Metaclass
Newly Defined Stereotype

Figure 3.27: The SysML4Consens profile

Furthermore, the Consens modeling language distinguishes different kind of flows, that is, energy flows, information flows, and material flow (cf. Section 2.2.1 and Section 2.2.5). In SysML, such flows can be represented by connectors between ports. However, the interface of the ports defines the connector's kind in SysML. This facilitates the reuse of interfaces, for example, across multiple hierarchy levels. Thus, we introduce the distinction between the different flow kinds to these port interfaces. SysML defines port interfaces by means of the stereotype **SysML::Ports&Flows::InterfaceBlock**, which refines the generic **SysML::Blocks::Block**. We specialize this stereotype with the abstract stereotype **FlowSpecification**, which we further subdivide into the particular stereotypes for the corresponding Consens flow kinds. We do not constrain the compatibility of the flow specifications of ports connected with each other, because the static semantics of UML/SysML are sufficient to ensure this interface compatibility.

Since mechanical connections simply connect mechanical elements and hence do not adhere to interfaces that specify which elements can flow over a connector, we extend the metaclass

UML4SysML::Connector with the corresponding stereotype **MechanicalConnection**. We also introduce the stereotype **Flow** as extension to the metaclass **UML4SysML::Connector** to represent disturbing flows of the partial model *Environment* with a corresponding Boolean attribute. Finally, we refine the **SysML::Ports&Flows::FullPort** with the stereotype **MeasurementPoint** to provide modeling means for the equally named CONSENS modeling concept.

We introduce further stereotypes for the integration of the CONSENS partial models *Application Scenarios* and *Functions* into SysML. First, we specialize SysML use cases by extending the metaclass **UML4SysML::UseCase** with the stereotype **ApplicationScenario** that introduces additional String attributes for the textual differentiation between the trigger situation and the intended behavior of an *Application Scenario*. Second, we extend the metaclass **UML4SysML::Class** with the stereotype **Function** and the metaclass **UML4SysML::Association** with the stereotype **FunctionContainment** to provide modeling means to specify function hierarchies.

Finally, we introduce the relational trace link types of CONSENS to SysML. In our extended terminology for model-based traceability (cf. Section 2.1.1), we define relational trace links in terms of the UML as dedicated modeling elements that are instances of the UML metaclass **DirectedRelationship** or of its metaclass specializations. The metaclass **UML4SysML::Abstraction** is such a specialization, where "[a]n Abstraction is a [directed] Relationship that relates two Elements or sets of Elements that represent the same concept at different levels of abstraction or from different viewpoints" [OMG17b]. Thus, we extend this metaclass with the corresponding stereotypes. We ensure that the relational trace links unidirectionally connect only the intended partial model elements by means of OCL constraints.

Relevance Annotations

The profile for relevance annotations depicted in Figure 3.28 is an *annotation profile* [SG14], which attaches supplementary information to a model. Thereby, the relevance annotation profile can be applied to any SysML-based language since the relevance annotations refine the basic structural SysML stereotype **SysML::Blocks::Block**.

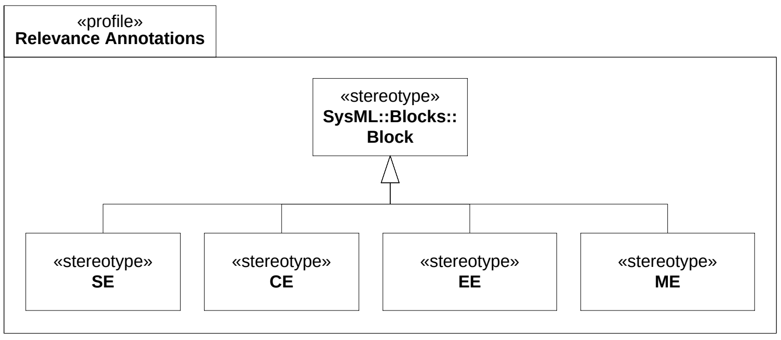


Figure 3.28: Profile for relevance annotations

In terms of terminology, we define it analogously to the relevance annotations of Rieke [Rie15]. That is, we use the stereotype **SE** to annotate that a system element is relevant to the discipline of software engineering, **CE** for control engineering, **EE** for electrical engineering, and **ME** for mechanical engineering.

By refining the SysML stereotype **SysML::Blocks::Block**, which represents a type (cf. last paragraph), we enable the user of the profile to non-redundantly apply the stereotype on system element templates. We define the semantics of a relevance annotation applied to a system element template that each system element exemplar typed by this template inherits the relevance annotation. Other profile design options, like allowing to apply relevance annotations to system element exemplars or both to system element exemplars and templates, would lead to redundancies in the resulting SysML4Consens models and to additional modeling effort.

Exemplary Application of the Profiles

We exemplify the application of the SysML4Consens profile as well as of the SysML profile for specifying the relevance annotations in this paragraph.

Figure 3.29 depicts the combined application of SysML4Consens and the SysML profile for the relevance applications with an example excerpt for the µC1 of the EBEAS (cf. Figure 2.2 in Section 2.2). Similarly to Section 3.1, we focus on the logical view within the µC1 in this figure.

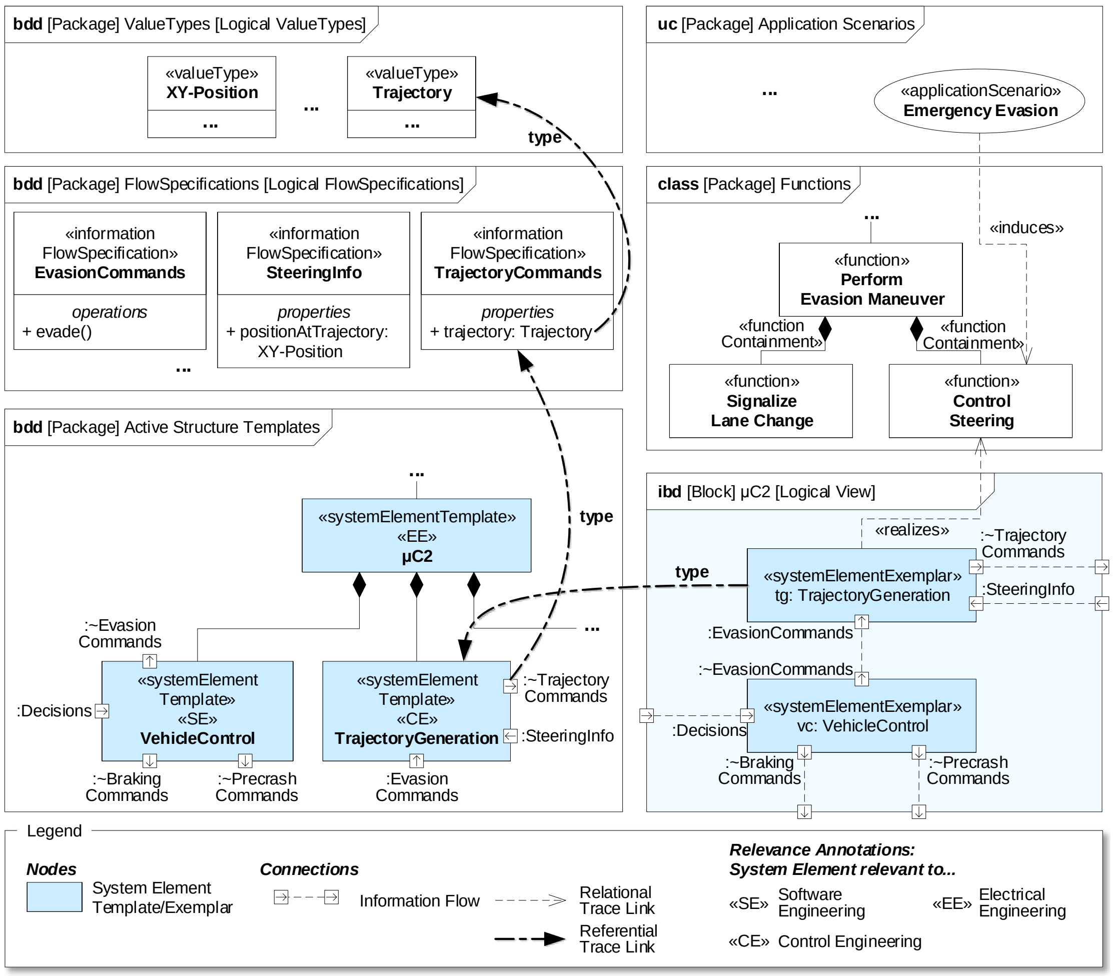

bdd [Package] ValueTypes [Logical ValueTypes]
«valueType» XY-Position
«valueType» Trajectory
type
uc [Package] Application Scenarios
«applicationScenario» Emergency Evasion
bdd [Package] FlowSpecifications [Logical FlowSpecifications]
«information FlowSpecification» EvasionCommands
operations
+ evade()
«information FlowSpecification» SteeringInfo
properties
+ positionAtTrajectory: XY-Position
«information FlowSpecification» TrajectoryCommands
properties
+ trajectory: Trajectory
class [Package] Functions
«induces»
«function» Perform Evasion Maneuver
«function Containment»
«function» Signalize Lane Change
«function» Control Steering
bdd [Package] Active Structure Templates
«systemElementTemplate» «EE» µC2
type
:~Evasion Commands
:Decisions
«systemElement Template» «SE» VehicleControl
:~Braking Commands
:~Precrash Commands
«systemElement Template» «CE» TrajectoryGeneration
:~Trajectory Commands
:SteeringInfo
:Evasion Commands
ibd [Block] µC2 [Logical View]
«realizes»
:~Trajectory Commands
:SteeringInfo
«systemElementExemplar» tg: TrajectoryGeneration
:EvasionCommands
:~EvasionCommands
«systemElementExemplar» vc: VehicleControl
:Decisions
:~Braking Commands
:~Precrash Commands
Legend
Nodes
System Element Template/Exemplar
Connections
Information Flow
Relational Trace Link
Referential Trace Link
Relevance Annotations: System Element relevant to...
«SE» Software Engineering
«EE» Electrical Engineering
«CE» Control Engineering

Figure 3.29: Excerpt of the SysML4Consens system model for the µC1 of the EBEAS

The right-hand side of the figure depicts an excerpt of the partial models *Application Scenarios*, *Functions*, and *Active Structure*. The relational trace links are specified by means of stereotyped links between the corresponding partial model elements. For example, the application scenario Emergency Evasion (i.e., a UML4SysML use case with the stereotype «applicationScenario») has a relational trace link with the stereotype «induces» to the function Control Steering (i.e., a UML4SysML class with the stereotype «function»). The system element exemplar tg: TrajectoryGeneration (i.e., a UML4SysML property with the stereotype «systemElementExemplar») in turn has a link with the stereotype «realizes» to the aforementioned function.

The left-hand side of Figure 3.29 depicts information that is needed in SysML to thoroughly specify the types of the system element exemplars (depicted in the block definition diagram Active Structure Templates), their port interfaces (i.e., *flow specifications* depicted in the block definition diagram Logical FlowSpecifications), and the actual *value types* (depicted in the block definition diagram Logical ValueTypes) used in the port interfaces. For example, the system element exemplar tg: TrajectoryGeneration is typed by the system element template **TrajectoryGeneration** (i.e., a SysML block with the stereotype «systemElementTemplate»). This system element template has a conjugated port of the information flow specification **TrajectoryCommands** (i.e., a SysML interface block with the stereotype «informationFlowSpecification»), meaning that it can send the contents of the flow specification via this port. The information flow specification **TrajectoryCommands** contains the flow property trajectory that is typed by the value type **Trajectory**, which is expressed with modeling means of the conventional SysML.

In the conventional CONSENS modeling language (cf. Section 2.2), the connector kind (i.e., information flow, energy flow, or material flow) as well as the contents that can flow via a connector are specified at the connector level. In contrast, the flow specification kind of the ports that a connector links determines the connector kind in SYSML4CONSENS. For example, the connector linking the ports of vc: VehicleControl and tg: TrajectoryGeneration represents an information flow since both ports are typed by the same information flow specification **EvasionCommands**.

As described in the last paragraph, we specify the relevance annotations on the type level for the system element templates. For example, the system element template **µC2** is relevant to the electrical engineering discipline (stereotype «EE»), **TrajectoryGeneration** is relevant to control engineering (stereotype «CE»), and **VehicleControl** is relevant to software engineering (stereotype «SE») in the block definition diagram Active Structure Templates.

3.9.1.2 CAPRA Traceability Information Models

This section presents the traceability information models defining the permissible trace link types between the permissible trace artifact types of an MSD specification and a CONSENS system model (cf. Section 3.7). We distinguish between a lifecycle and a transformation traceability information model (cf. Section 2.1.1). Both traceability information models are specified by means of the traceability metamodel specification capabilities of the traceability management tool CAPRA (cf. Section 2.1.2).

The following paragraph presents the lifecycle traceability information model, whereas the subsequent paragraph presents the transformation traceability information model.

Lifecycle Traceability Information Model

For readability reasons, we divide the lifecycle traceability information model into concepts regarding the classifier view type, the architecture view type, and the interaction view type of component-based MSD specifications (cf. Section 3.2). We conceptually constrain the metamodel with OCL constraints [OMG14a] to improve the validity of the particular trace links (cf. Section 2.1.1), since it is intended that the Software Requirements Engineer partially establishes corresponding traces in a manual manner. We implemented these conceptual OCL constraints by means of a Java implementation within the Traceability Metamodel extension point of CAPRA. One CONSENS model element typically results in several elements spread across multiple MSD use cases within an MSD specification. Thus, most multiplicities for the trace link type references to trace artifact types of CONSENS system models and of MSD specifications are 1:* if not mentioned otherwise in the following.

Figure 3.30 presents the metamodel for the lifecycle traces between the classifier view type of component-based MSD specifications and SYSML4CONSENS system models. It relates component types of an MSD specification with environment/system element templates of a SYSML4CONSENS system model as well as interfaces of an MSD specification with information flow specifications of a SYSML4CONSENS system model.

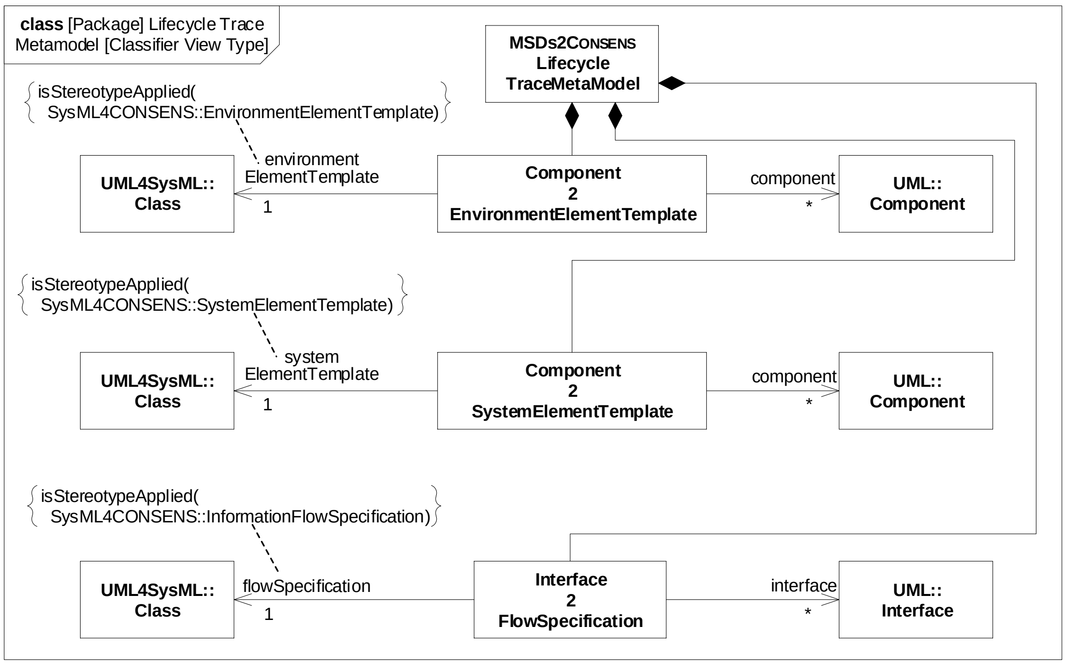

class [Package] Lifecycle Trace Metamodel [Classifier View Type]
MSDs2CONSENS Lifecycle TraceMetaModel
isStereotypeApplied(SysML4CONSENS::EnvironmentElementTemplate)
environment ElementTemplate
1
UML4SysML:: Class
Component 2 EnvironmentElementTemplate
component
*
UML:: Component
isStereotypeApplied(SysML4CONSENS::SystemElementTemplate)
system ElementTemplate
1
UML4SysML:: Class
Component 2 SystemElementTemplate
component
*
UML:: Component
isStereotypeApplied(SysML4CONSENS::InformationFlowSpecification)
flowSpecification
1
UML4SysML:: Class
Interface 2 FlowSpecification
interface
*
UML:: Interface

Figure 3.30: Metamodel for the lifecycle traces between the classifier view type of component-based MSD specifications and SYSML4CONSENS system models

Figure 3.31 presents the metamodel for the lifecycle traces between the architecture view type of component-based MSD specifications and SYSML4CONSENS system models. It relates use cases of an MSD specification with *Application Scenarios* of a SYSML4CONSENS system model as well as environment/system component roles of an MSD specification with environment/system element exemplars of a SYSML4CONSENS system model. Since exactly one MSD use case is generated out of a CONSENS application scenario, the multiplicities of the association ends of the trace link type **MSDUseCase2ApplicationScenario** are 1:1. The OCL constraints for the trace link types between the artifact types representing environment/system component roles of an MSD specification and environment/system element exemplars in SYSML4CONSENS constrain the resulting trace links to be instantiable according to parts of the mapping rules from SYSML4CONSENS to MSD specifications (cf. Sections 3.4 and 3.8.2.2): A **SystemRole2SystemElementExemplar** trace link can be instantiated between an MSD system component role and an CONSENS system element exemplar with relevance annotation “SE”, an **EnvironmentRole2SystemElementExemplar** trace link can be instantiated between an MSD environment component role and a CONSENS system element exemplar with relevance annotation “CE”, and an **EnvironmentRole2EnvironmentElementExemplar** trace link can be be instantiated between an MSD environment component role and a CONSENS environment exemplar.

Figure 3.32 presents the metamodel for the lifecycle traces between the interaction view type of component-based MSD specifications and SYSML4CONSENS system models. It relates MSDs with SYSML4CONSENS *Requirements* as well as with *Behavior–Sequences*. Since an MSD can emerge from arbitrary many requirements and one requirement can be reflected by multiple MSDs, the multiplicities of the association ends of the trace link type **MSD2Requirement** are *:*.

Transformation Traceability Information Model

Figure 3.33 presents the transformation traceability information model, which enables the incremental updates of our transition technique through establishing fine-grained transformation trace links between the corresponding trace artifacts. We provide concrete examples on the model level in Section 3.7.2.

In contrast to the lifecycle traceability information models, the transformation traceability information model is more generic in the sense that it allows to associate arbitrary UML/-SysML trace artifacts with each other (i.e., the trace artifact type **Element**) via the trace link type **ModelElement2ModelElement**. The transformation algorithm automatically ensures the trace link validity—constraints at the metamodel level are not necessary since no manual trace establishment is intended. The multiplicities of the association ends of the trace link type are 1:* since partially multiple MSD specification elements have to be associated with one CONSENS system model element (e.g., a software component derived from one system element can participate in multiple MSD use cases).

3.9.2 Case Study

We conduct a case study based on the guidelines by Kitchenham et al. [KPP95] and by Runeson et al. [RHAR12; RH08] for the evaluation of our transition technique. In our case study, we investigate the usefulness of our approach within the domain of software-intensive systems.

3.9.2.1 Case Study Context and Cases

The objective of our case study is to evaluate whether our transition technique is useful for the Software Requirements Engineers. For this purpose, we evaluate the following questions:

Evaluation Question 1 (EQ1) Does the transition technique derive reasonable MSD specifications as a basis for the manual refinement?

Evaluation Question 2 (EQ2) Does the transition technique reduce the engineering effort for conceiving MSD specifications based on CONSENS system models?

We do not aim at generalizing the case study conclusions to all possible CONSENS system models and MSD specifications but conduct the study for the following two cases:

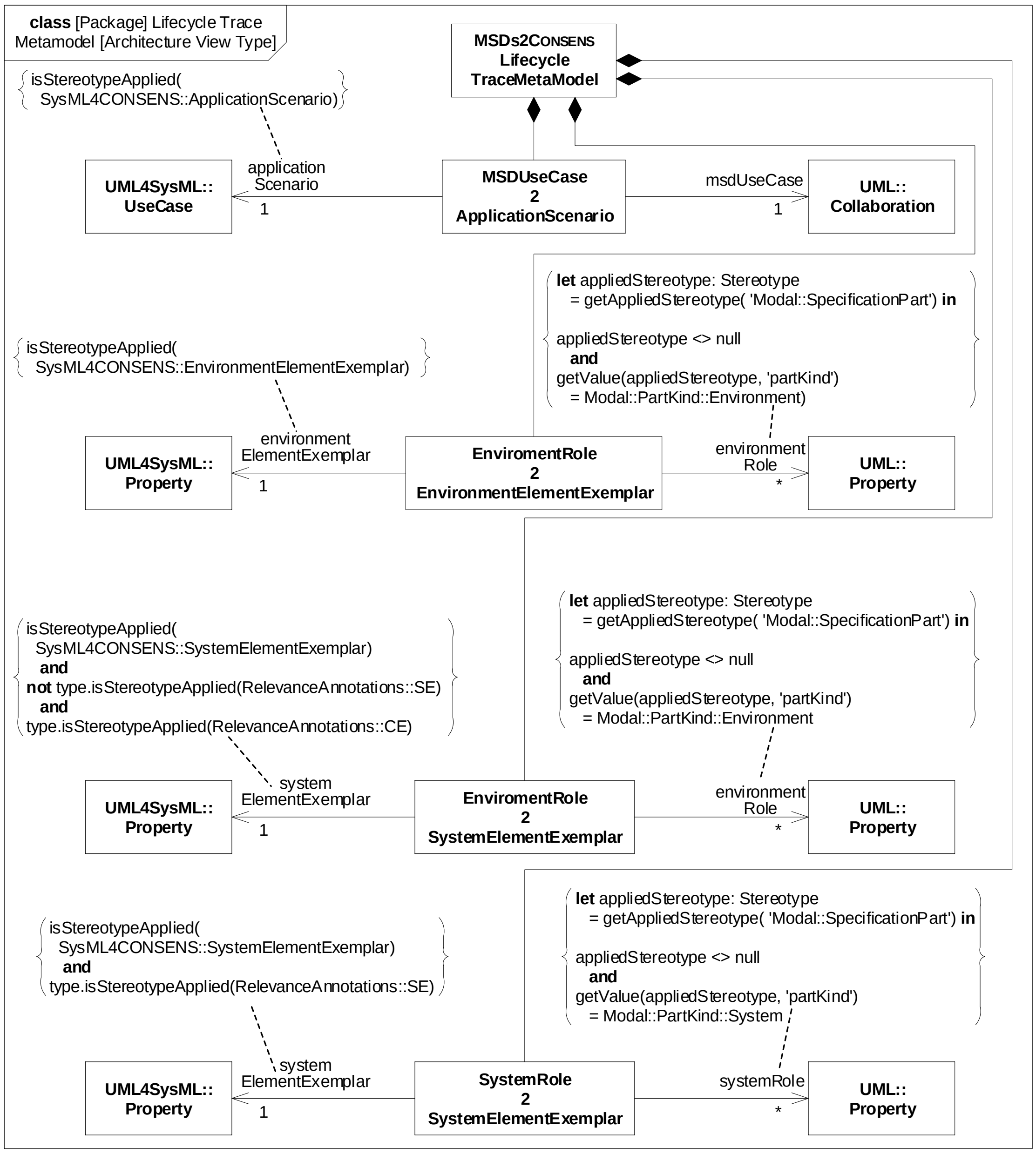

class [Package] Lifecycle Trace Metamodel [Architecture View Type]
MSDs2CONSENS Lifecycle TraceMetaModel
{isStereotypeApplied(SysML4CONSENS::ApplicationScenario)}
UML4SysML:: UseCase
application Scenario
1
MSDUseCase 2 ApplicationScenario
msdUseCase
1
UML:: Collaboration
{let appliedStereotype: Stereotype = getAppliedStereotype('Modal::SpecificationPart') in appliedStereotype <> null and getValue(appliedStereotype, 'partKind') = Modal::PartKind::Environment)}
{isStereotypeApplied(SysML4CONSENS::EnvironmentElementExemplar)}
UML4SysML:: Property
environment ElementExemplar
1
EnviromentRole 2 EnvironmentElementExemplar
environment Role
*
UML:: Property
{let appliedStereotype: Stereotype = getAppliedStereotype('Modal::SpecificationPart') in appliedStereotype <> null and getValue(appliedStereotype, 'partKind') = Modal::PartKind::Environment}
{isStereotypeApplied(SysML4CONSENS::SystemElementExemplar) and not type.isStereotypeApplied(RelevanceAnnotations::SE) and type.isStereotypeApplied(RelevanceAnnotations::CE)}
UML4SysML:: Property
system ElementExemplar
1
EnviromentRole 2 SystemElementExemplar
environment Role
*
UML:: Property
{let appliedStereotype: Stereotype = getAppliedStereotype('Modal::SpecificationPart') in appliedStereotype <> null and getValue(appliedStereotype, 'partKind') = Modal::PartKind::System}
{isStereotypeApplied(SysML4CONSENS::SystemElementExemplar) and type.isStereotypeApplied(RelevanceAnnotations::SE)}
UML4SysML:: Property
system ElementExemplar
1
SystemRole 2 SystemElementExemplar
systemRole
*
UML:: Property

Figure 3.31: Metamodel for the lifecycle traces between the architecture view type of component-based MSD specifications and SYSML4CONSENS system models

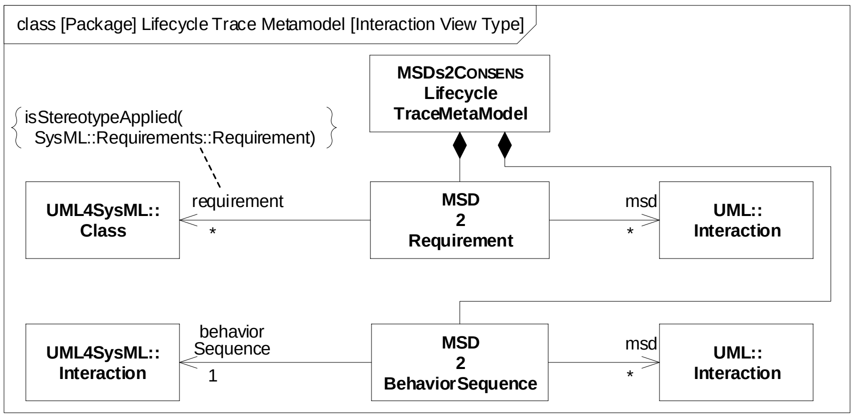

class [Package] Lifecycle Trace Metamodel [Interaction View Type]
MSDs2CONSENS Lifecycle TraceMetaModel
{ isStereotypeApplied(SysML::Requirements::Requirement) }
UML4SysML:: Class
requirement
*
MSD 2 Requirement
msd
*
UML:: Interaction
UML4SysML:: Interaction
behavior Sequence
1
MSD 2 BehaviorSequence
msd
*
UML:: Interaction

Figure 3.32: Metamodel for the lifecycle traces between the interaction view type of component-based MSD specifications and SYSML4CONSENS system models

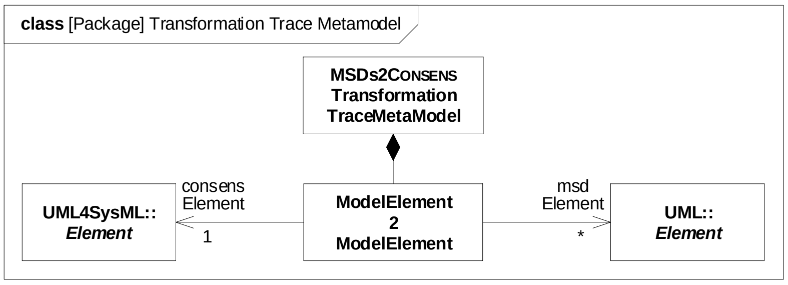

class [Package] Transformation Trace Metamodel
MSDs2CONSENS Transformation TraceMetaModel
UML4SysML:: Element
consens Element
1
ModelElement 2 ModelElement
msd Element
*
UML:: Element

Figure 3.33: Metamodel for the transformation traces enabling incremental updates

Emergency Braking & Evasion Assistance System (EBEAS) The EBEAS used throughout this thesis as a running example. In contrast to the second case, the EBEAS does not involve mechanical elements fulfilling concrete functional requirements, and its models are specified in more detail.

Car Access System (CAS) A car access system encompassing a mechatronic door lock system and the central locking functionality of a car's body control module ECU, which we presented in [*HBM$^+$16; *HBM$^+$15] based on the example in [*Ber15]. Despite being software-intensive, the system has in contrast to the EBEAS a more mechatronic character where all disciplines are equally involved.

3.9.2.2 Setting the Hypotheses

Based on the aforementioned case study objective and evaluation questions, we define the following evaluation hypotheses:

Hypothesis H1 Our transition technique derives and updates correct MSD specifications w.r.t. to initial and changed CONSENS system models, respectively (cf. evaluation question EQ1).

For evaluating H1, we give the two cases to each one of two students. The respective student qualitatively evaluates for the case he is responsible for whether all SwRE-relevant and whether no SwRE-irrelevant elements of the CONSENS model are present in the MSD specification.

We consider H1 fulfilled if the students judge that their respective generated/updated MSD specifications encompass all SwRE-relevant and no SwRE-irrelevant elements from the SYSML4CONSENS models.

Hypothesis H2 Our transition technique reduces the engineering effort for conceiving and modeling MSD specifications based on the respective CONSENS system models (cf. evaluation question EQ2).

For evaluating H2, we determine for any case the amounts of the overall and the SwRE-relevant model elements in the initial CONSENS system model. Furthermore, we determine the amounts of model elements that are initially derived and manually refined in the respective MSD specifications.

We consider H2 fulfilled if in average $\geq$20% of SwRE-relevant system model elements are identified and $\geq$50% of the final MSD specification is initially generated.

3.9.2.3 Data Collection Preparation

Besides ourselves, we employ two different students *student-1* and *student-2* to support the evaluation. Student-1 has approximately three years experience in modeling SYSML4CONSENS system models as well as in modeling and simulating MSD specifications during the case study conduct. Furthermore, he conceived and implemented the automatic derivation part of our transition technique [*Ber15]. Student-2 has little experience with CONSENS and approximately one year of experience in modeling and simulating MSD specifications during the case study conduct.

For both cases, student-1 prepares each an initial as well as a changed SYSML4CONSENS model. For this purpose, he conducts the steps allocated to the Systems Engineer in the process described in Section 3.3. We present the initial SYSML4CONSENS model for the EBEAS case as CONSENS variant in Appendix A.2.1 and describe its changed version in Section 3.6.2.1. We present the initial as well as the changed SYSML4CONSENS model for the CAS case as CONSENS variant in [*HBM+16].

The SYSML4CONSENS system models have to be complete and correct w.r.t. the transformation algorithm (cf. Section 3.8.2.2) in order to be subject to a successful model transformation. For example, the relational traceability as shown in Appendix A.2.1 has to be specified, all ports with incident information flows have to be typed by the corresponding port specifications, and the relevance annotations have to be specified. For this purpose, student-1 specifies the SYSML4CONSENS system models in such a way that they adhere to the preconditions described in Section 3.8.1.

3.9.2.4 Data Collection Procedure

In the following, we describe the procedure for the data collection for hypothesis H1 and subsequently the procedure for the data collection for hypothesis H2.

Hypothesis H1

For evaluating H1, we give the initial and changed SYSML4CONSENS models for the EBEAS case to student-1 and the initial and changed SYSML4CONSENS models for the CAS case to student-2. Both students conducts the following procedure for their respective case:

1. They derive an initial MSD specification from the initial SYSML4CONSENS model.

2. They qualitatively evaluate whether all SwRE-relevant and whether no SwRE-irrelevant elements from the SysML4Consens model are present in the initial MSD specification.
3. They manually refine the initially derived MSD specification.
4. They update the initial MSD specification based on the changed SysML4Consens model.
5. They qualitatively evaluate whether all SwRE-relevant and whether no SwRE-irrelevant elements from the SysML4Consens model are present in the updated MSD specification.

The evaluation of H1 yields for both cases that all SwRE-relevant elements and no superfluous SwRE-irrelevant elements are present in the initially generated as well as in the updated MSD specification. For the incremental update, each student moreover observes that none of his manual refinements from the particular initial process iterations is lost.

Hypothesis H2

For evaluating hypothesis H2, both students follow the same procedure as described for the evaluation of H1. That is, for both cases they initially derive and update an MSD specification and refine it afterward.

Beyond that, student-1 determines for both cases the amounts of the overall (variable H2.1) and the SwRE-relevant (variable H2.2) model elements in the initial SysML4Consens system model. Furthermore, student-1 determines for both cases the amounts of model elements that are initially derived (variable H2.3, cf. Section 3.6.1.1 to Section 3.6.1.3) and manually refined (variable H2.4, cf. Section 3.6.1.4 and Appendix A.2.2.2) in the respective MSD specifications.

Student-1 determines the model element amount variables H2.1, H2.3, and H2.4 automatically based on the respective models. In contrast, student-1 determines H2.2 partially in a manual way since the identification of SwRE-relevant model elements is hidden in the transformation. That is, there is no corresponding intermediate SwRE-relevant SysML4Consens system model that can be input to an automatism. However, ourselves determine a part of H2.2 by means of OCL [OMG14a] queries where reasonable.

On the counting of model elements, we orientate toward the manual modeling effort within Papyrus since the granularity level w.r.t. the question "what is a distinct model element?" is very arguable. That is, we count every model operation that has to be actively conducted by the Software Requirements Engineer in the modeling tool. For example, we count the specification of an MSD message as one action because the Software Requirements Engineer only has to connect the message to two lifelines, whereas the two associated send and receive message occurrence specifications are created automatically by the modeling tool. Accordingly, we count the establishment of the referential traceability (e.g., the referential trace links connector and signature of an MSD message to elements of the other view types) and setting mandatory name attributes as dedicated model operations, because the Software Requirements Engineer has to actively conduct the respective operation (cf. detailed model element variable amounts in Appendix A.3).

We follow the same procedure for the changed SysML4Consens system models and the corresponding updated MSD specifications but focus on the changed model elements. For this purpose, we define the variables H2.1-update for the amount of system model elements that were manually changed and H2.3-update for the amount of MSD specification model elements that were automatically updated due to the system model changes. We determine these amounts manually, because simply comparing the respective overall model element amounts before and after a change would ignore potential relevant changes (e.g., a model element movement is not counted as change to the model in terms of the overall model element amount).

The determination of the model element amounts for H2 yields the results as listed in Table 3.1. The table lists the results for H2 for the EBEAS case (column # EBEAS) as well as the car access system case (column # CAS). Additionally, the table provides percentage-wise relations between certain model element amounts as well as the average of these percentage values (column ∅ %).

Table 3.1: Results of model element amount variables for hypothesis H2

ID	Description	# EBEAS	# CAS	∅ %
H2.1	Overall system model elements	1008	748	
H2.2	Automatically identified SwRE-relevant system model elements	342	296	
H2.2 ÷ H2.1	Percentage of identified SwRE-relevant system model elements	~34%	~40%	~37%
H2.3	Automatically generated MSD specification model elements	703	359	
H2.4	Overall MSD specification elements after manual refinement	1307	403	
H2.3 ÷ H2.4	Percentage of final MSD specification automatically generated	~54%	~89%	~67%
H2.1-update	Changed system model elements	153	65	
H2.3-update	Automatically updated MSD specification elements	273	51	

The H2 results for the case of the EBEAS are as follows. In terms of relationships between the amounts for the initial generation, ~34% (H2.2 ÷ H2.1) SwRE-relevant elements are identified in the SYSML4CONSENS system model and ~54% (H2.3 ÷ H2.4) of the final MSD specification is generated. The amounts for the incremental update yield that manually changing 153 system model elements (H2.1-update) results in automatically updating 273 model elements in the MSD specification (H2.3-update). In contrast to the car access system case, the amount of updated MSD specification model elements is higher than the amount of changed system model elements since the system model change influences all four automatically derived MSD use cases and hence all their contained model elements, which are partly redundant.

The H2 results for the case of the car access system slightly differ from the ones presented in [*HBM^{+}16] due to an updated transformation algorithm and a more detailed model element amount determination method, and they are are as follows. In terms of relationships between the amounts for the initial generation, ~40% (H2.2 ÷ H2.1) SwRE-relevant elements are identified in the SYSML4CONSENS system model and ~89% (H2.3 ÷ H2.4) of the final MSD specification is generated. There are more MSD specification elements than SwRE-relevant system model elements due to the fact that several MSD use cases typically encompass redundant model elements, which is also true for the EBEAS case. The amounts for the incremental update yield that manually changing 65 system model elements (H2.1-update) results in automatically updating 52 model elements in the MSD specification (H2.3-update). The lesser amount of the updated MSD specification model elements in comparison to the amount of the system model elements is due to two reasons. First, it arises from the flattening of the architecture hierarchies in the transformation to the MSD specification. Second, the system model change influences only one MSD use case so that there are no redundancies.

There is only a little difference between the percentage of identified SwRE-relevant system model elements for the EBEAS case (~34%) and the car access system case (~40%), respectively. This indicates that the kind of the system (software-focused EBEAS vs. mechatronic car access system) seems not to strongly influence the amounts of identified system model elements.

In contrast, the values on the automatically generated part of the final MSD specification (~54% for the EBEAS and ~89% for the car access system) indicate that more parts of the MSD specification have to be manually specified or updated if more software is involved in the SUD. This is reasonable due to the higher degree of software-intensive complexity of the EBEAS compared with the car access system. That is, we specified only the most relevant, exemplary *Behavior – Sequence* for each SwRE-relevant *Application Scenario*, which is common in the practice of specifying coarse-grained CONSENS system models. However, it is necessary to consider all possible situations in SwRE in order to gain a complete and realizable MSD specification. Thus, the Software Requirements Engineer has to specify many more MSDs manually for the EBEAS than for the car access system to cover the whole state space (cf. the *Behavior – States* in Figure A.12 in Appendix A.2.1 as well as the differences between the initially generated MSD specification in Appendix A.2.2.1 and the final MSD specification in Appendix A.2.2.3).

Summarizing, the averaged percentage relations between the particular model element amounts yield that ~37% of the system model is automatically identified (i.e., is SwRE-relevant) and ~67% of the final MSD specification is automatically generated.

3.9.2.5 Interpreting the Results

The results for H1 show that the initially derived as well as the updated MSD specifications are correct in the sense that they only encompass SwRE-relevant modeling elements and no superfluous SwRE-irrelevant modeling elements. Thus, we consider our hypothesis H1 fulfilled.

The results for H2 show three aspects. First, a large part of the system model is automatically identified such that the Software Requirements Engineer does not have to care about the filtered SwRE-irrelevant model elements. Second, more than the half of the MSD specification is automatically generated, and only the remainder has to be specified manually. Third, the results for the incremental update indicate that the ratio between manually changed system model elements and automatically updated MSD specification model elements depends on the amount of MSD use cases impacted by the system model change. All such automatically conducted activities would have been performed completely manually without our transition technique. Since 20% of the SwRE-relevant system model elements are identified and more than 50% of the final MSD specification is initially generated, we also consider H2 fulfilled.

In summary, the fulfilled hypotheses yield that our transition technique guides the Software Requirements Engineers with reasonable MSD specifications and reduces their manual effort. This gives rise to the assumption that the approach is indeed useful for the Software Requirements Engineers.

3.9.2.6 Threats to Validity

The threats to validity in our case study (structured according to the taxonomy of Runeson et al. [RHAR12; RH08]) are as follows.

Construct Validity

- Student-1 and ourselves conceived the CONSENS system models for both cases by ourselves, knowing the functional principle of our transition technique. Thus, the case study would be more significant if other CONSENS experts would have conceived the system models.

However, we through extensive discussions with other researchers as well as industry experts. Furthermore, the system models base on examples that ourselves conceived in other contexts. That is first, we conceived the EBEAS case based on real-world advanced driver assistance systems for the purpose of exemplifying the MSD language together with other researchers from our research institute and our university's research group [*HFK+16]. Afterward, we further discussed the example with external researchers and industry experts in the European research project "AMALTHEA4Public" [A4P]. Second, the car access system case bases on artifacts that we conceived together with an automotive tier-1 supplier based on a real-world product from them in the automotive application project of the German research project "SPES 2020" [SPES2020] (e.g., [*MHNM10; *DFHT13]). Furthermore, we discussed these artifacts with external "SPES 2020" researchers and enhanced these artifacts in the context of the cross-sectional project "Systems Engineering" [itsOWL-SE] of the leading-edge cluster "it´s OWL" [itsOWL] (e.g., [*PHM14]) as well as in industry projects. These procedures assure that the CONSENS system models base on reasonable, internally as well as externally reviewed, and partly real-world examples for software-intensive systems.

- Student-1 who conceived the initial transition technique [*Ber15] evaluated H1 for the EBEAS case and H2 for both cases. Thus, he could have been biased toward the approach under investigation.

  However, we mitigate this threat by employing student-2 for the evaluation of H1 with the car access system case and by automating the evaluation of H2 as far as possible.

- There might be better metrics for quantifying the engineering effort of model manipulations than counting the conducted model element operations. For example, we only measure the actual specification effort in PAPYRUS (i.e., setting a referential trace link or specifying a name attribute) but do not measure the cognitive effort that is needed to prepare the actual tooling operation (i.e., thinking about which model element has to be traced and about how the model element should be named, respectively).

  Nevertheless, the results are easily reproducable, are determined mainly automatically, and do not depend on the general cognitive abilities or the current cognitive state of the model-manipulating person. Moreover, we mitigate this threat by using our own modeling experience with PAPYRUS so that we yield realistic results by, for example, not counting optional model operations (cf. the details about the counting method in Appendix A.3). Furthermore, Durisic et al. [DSTH17] report that using such simple and atomic metrics, like the number of changes or of model elements, is "a good indicator" for predicting the effort though having certain threats to validity as every metric.

Internal Validity

We judge in two steps (i.e., H2.2 $\div$ H2.1 and H2.3 $\div$ H2.4) via the model element amount variables for H2 about a causal relation between model sizes and the reduction of the effort for the Software Requirements Engineer. The ratio between SwRE-relevant system elements and system elements relevant to other disciplines results in different model element amounts that we apply as basis for the judgment on the effort reduction.

On the one hand, the higher the portion of SwRE-relevant system elements the lower the portion of filtered (i.e., SwRE-irrelevant) system elements (H2.2 $\div$ H2.1). Thus, a higher ratio of SwRE-relevant system elements reduces the effectiveness of our approach for this first step. On the other hand, the higher the portion of SwRE-relevant system elements the higher the amount of automatically derived MSD specification elements and at the same time the higher the amount of manually refined MSD specification elements (H2.3 ÷ H2.4). Thus, a higher ratio of SwRE-relevant system elements partly increases the effectiveness of our approach for this second step. All in all, both effects should neutralize each other. However, other factors not taken into account can influence these observations.

External Validity
We only considered two cases, and both cases stem from the automotive sector. Furthermore, exemplary case studies in general cannot ensure external validity. Thus, we cannot generalize the conclusions to all possible CONSENS system models and MSD specifications, other types of software-intensive systems, or software-intensive systems in other industry sectors. Nevertheless, the examples are typical for software-intensive systems, and we hence do not expect large deviations for other types of systems.

Reliability

- Student-1 judging H1 for the car access system case has little experience with CONSENS and hence could have judged incorrectly whether only SwRE-relevant elements are present in the MSD specification. Furthermore, student-2 might also have incorrectly judged H1 for the EBEAS case. Anyhow, both could judge the correct execution by investigating the SYSML4CONSENS model with relevance annotations and reenacting the model transformation algorithm (cf. Section 3.8.2.2).
- The partially manual counting of model elements in the context of H2 might be incorrect. However, we do not expect a large discrepancy between actual and counted elements so that the derived conclusion regarding H2 is not affected.

3.10 Related Work

In this section, we investigate related work on the transition from MBSE to discipline-specific models (cf. Section 3.10.1), on system modeling languages with discipline-specific information (cf. Section 3.10.2), on component-based scenario notations (cf. Section 3.10.3), and on semi-automatic traceability establishment (cf. Section 3.10.4).

3.10.1 Transition from MBSE to Discipline-specific Models

Greenyer [Gre11] exemplarily uses the information in CONSENS system models as a basis to manually conceive MSD specifications. However, a systematic method or an automatism for this task is not in the scope of his thesis. Beyond that, two approaches exist for the transition from MBSE with CONSENS to software engineering. As the first one, Heinzemann et al. [HSST13] present a systematic development process for the software comprising control and coordination behavior. This process utilizes automatisms to derive initial discipline-specific design models and keep them consistent with the system model afterward [Rie15; Rie14; RDS+12; GSG+09; Rie08; GGS+07]. Furthermore, we transferred this approach to SYSML4CONSENS [*PHM14] and to the automotive sector by deriving purely structural software design models from system models in a consistency-preserving manner [*FHH+12; *FHM12; *HMM11; *MH11]. However, these approaches aim at deriving and refining design models but lack a transition to SwRE. As the second one, Anacker et al. [AGD+12] present a systematic process for the transition from MBSE with CONSENS to SwRE with MSDs. However, this approach provides no automatic support. Furthermore, it relies on reusable solution patterns (i.e., there must be prior projects in which these patterns have been successfully developed) and thereby does not support greenfield development.

Similar to our work, Böhm et al. [BHH+14] present a model-based method for the transition from systems engineering to software engineering that is compliant to established process models. They also address that these process models demand a software requirements analysis phase prior to the software design: A system element to be concretized in software engineering is input to the context model of the SPES requirements viewpoint [DTW12]. This context model is comparable to the CONSENS partial model *Environment*. However, they do not provide automatisms to support the transition. Furthermore, they only consider structural but no behavioral models, which are crucial for the specification of functional requirements.

Regarding the derivation of behavioral models from system models, the OMG developed a bidirectional transformation between SysML and Modelica [OMG12]. Similarly, Cao et al. present an integration between SysML and MATLAB/Simulink models based on a bidirectional model transformation [CLP11]. Such approaches focus on the derivation of discipline-specific design and analysis models. In contrast to our work, the derivation or refinement of discipline-specific requirements from multidisciplinary system requirements is not addressed.

Thramboulidis [Thr10] presents an interdisciplinary mechatronic view concept specified by means of SysML stereotypes for discipline-specific system elements and interfaces, which is very similar to our combination of discipline-specific relevance annotations and port specifications. However, the approach aims at deriving several discipline-specific structural models and at synchronizing them with the purely structural mechatronic view. The transition to SwRE, the consideration of behavioral models, and engineering support by means of automatisms are not considered or regarded as future work.

3.10.2 System Modeling Languages and Methods with Discipline-specific Information

The SysML-based MBSE methods Object-Oriented Systems Engineering Method (OOSEM) [FMS12, Chapter 17] and SYSMOD [Wei16] serve the same purpose as CONSENS. Both methods only propose to refine the system requirements within a typical software engineering process, but the actual transition to the software requirements is out of their scope. Furthermore, both methods provide each a SysML profile introducing method-specific language concepts, including language concepts similar to our relevance annotations. Regarding discipline-specific relevance annotations, OOSEM only distinguishes between software and hardware, and SYSMOD additionally considers mechanical elements. In contrast, distinguishing between discrete and continuous software components is crucial for our transition technique. Nevertheless, the basic principle of our transition technique could also be transferred to OOSEM and SYSMOD by introducing this distinction into to the particular profiles and adapting our algorithms to them. Furthermore, our semi-automatic support for the manual refinement has to be adapted to the particular methods.

Vogel-Heuser et al. [KFV18; KV13; BKFV14; FKV14] present a systems engineering approach based on the SysML extension SysML4*Mechatronics*. They describe a coarse-grained design process and introduce stereotypes for system elements to be concretized in their respective engineering disciplines. These stereotypes are similar to our relevance annotations. However, the approach aims on analyzing discipline-spanning change influences and not on the transition to discipline-specific models. Furthermore, the discipline-specific information in the system model is not exploited to facilitate automated tool support.

3.10.3 Component-based Scenario Notations

Combes et al. [CHK08] present a case study that applies LSCs to a component-based architecture for telecommunication systems. These models are validated by means of Play-out and formally verified for the non-satisfiability of anti-scenarios using Smart Play-out [HKMP02; HKMP03; HKP04]. Similar to our view-based approach, they divide their modeling approach into a structural, a dynamical, and an extra-functional view. The structural view of the system encompasses components, ports, port interfaces containing the messages that can be exchanged between the ports, and connectors specifying communication channels between the component ports. The dynamic view consists of LSCs that are specified based on the structural view. The extra-functional view adds timing constraints to the LSCs. However, there are only scenarios that describe interactions between a component and its ports as well as between ports of different components on the same hierarchy level. Thus, there are no means to specify or refine scenarios across component hierarchy levels.

Atir et al. [AHKM08] adapt the original MSD Play-out approach [MH06; HKM07; HMSB10] to hierarchical object compositions. For this purpose, the authors syntactically and semantically extend MSDs by the PartDecomposition concept of the UML [OMG17b]. To adapt the Play-out approach to be compatible with the PartDecomposition concept, trees consisting of several part scenarios that together reflect the object composition hierarchy including consistency rules are conceived. A composition algorithm integrates this scenario hierarchy into overall flattened MSDs that are executable in Play-out. Although the approach is able to traverse hierarchies, only class models are supported. Thus, the typical concepts of components like ports, interfaces, and directed connectors are not considered.

Other approaches like [LM09] and [SDP10] reflect hierarchies in combination with scenario-based specifications in a simple way, but do not intend to provide any tool support for automatic analysis techniques. The UML [OMG17b] as well as its profiles SysML [OMG17a] and MARTE [OMG11] in general provide the means to specify hierarchical components including ports and directed connectors as well as scenarios describing their dynamic behavior. However, the absence of formal semantics in the UML introduces semantic ambiguities and thereby prevents the application of automatic analysis techniques [HM08; HM06].

3.10.4 Semi-automatic Establishment of Explicit Lifecycle Traceability

Winkler and von Pilgrim survey and classify (semi-)automatic traceability approaches [WP10]. In contrast to our transition technique, most establishment approaches of these surveyed ones establish traceability a posteriori between already existing artifacts. Since in this case only trace link candidates can be proposed, such trace links are usually more imprecise [WP10].

In the category of fully automated inter-model traceability creation, Jouault presents an approach using model transformations [Jou05]. Like in our approach, the trace link creation is specified as part of the transformation rules so that the trace links are created together with the target model, where the admissible trace link types are specified in a traceability information model. However, the maintenance of traceability is out of his scope, because updates of the target models are not supported.

Drivalos-Matragkas et al. [DKPF10] present an approach considering inter-model traceability maintenance and trace link validity. Like our approach, they use a traceability information model to define which trace artifact types can be associated by which trace link types. Based on that, they semi-automatically analyze an existing trace link set for well-formedness and trace validity. For invalid traces, a traceability engineer has to select the semantically valid trace links. In some cases, the approach can automatically repair broken or invalid trace links by means of fuzzy matching. However, the approach does not consider the creation of traceability.

Zisman et al. [ZSPK03] automatically establish traceability between three different abstraction levels of requirements specifications, where the two most abstract ones are expressed in natural language and the most concrete one is expressed in UML. They use a natural language parser for automatically processing the natural language specifications and establish the trace links mainly by name comparison. Thus, trace link validity is only considered in a limited way. Furthermore, they do not generate initial models from the natural language requirements but establish the trace links a posteriori from existing specifications.

3.11 Summary

In this chapter, we presented a technique for the semi-automatic and systematic transition from MBSE with CONSENS to SwRE with MSDs. Particularly, the transition technique applies incremental model transformations to automatically derive initial and update existing MSD specifications based on CONSENS system models. The model transformations are complemented by a semi-automatic part, which supports the Software Requirements Engineers in the manual refinement of the derived or updated MSD specifications. This support encompasses informal guidelines for the systematic refinement of MSD specifications, automatic checks for coverage of refined MSD specifications w.r.t. CONSENS *Behavior – States*, and the generation of existential MSDs from CONSENS *Behavior – Sequences* serving as a test oracle for refined MSD specifications. As a basis for the automatic part, we extend existing modeling languages by providing the SYSML4CONSENS profile as well as the SysML relevance annotation profile, and we apply MSD specifications to component-bases software architectures. Our extension to the QVT-O model transformation approach enables the model transformations incrementally updating MSD specifications due to changes in CONSENS system models while preserving information added during the manual refinement. Furthermore, the model transformations automatically establish lifecycle traceability between CONSENS system models and the automatically derived parts of the MSD specifications. Our traceability information model further enables the Software Requirements Engineer to manually add lifecycle traces between CONSENS system models and the manually added parts of MSD specifications. We provide a process description that clarifies the role responsibilities between the particular disciplines, the artifact inputs to the particular process steps, and the application of the transition technique. Using a case study, we evaluate the transition technique by means of two cases from the automotive sector.

The transition technique supports Software Requirements Engineers in effectively and efficiently conceiving MSD specifications that are correct, complete, and concise w.r.t. CONSENS system models. That is first, it increases the effectiveness of the transition by reducing error-prone cases in which the Software Requirements Engineers overlook or misinterpret information in the CONSENS system models. That is second, it increases the efficiency of the Software Requirements Engineers' tasks by reducing their time-consuming manual work. The coverage checks address the completeness and conciseness of MSD specifications w.r.t. CONSENS *Beha-*

vior – States, and the generation of existential MSDs addresses the correctness of refined MSD specification w.r.t. CONSENS *Behavior – Sequences*. Furthermore, the resulting MSD specifications do not depend completely on the way the particular persons fulfilling the role of the Software Requirements Engineer proceed during the transition process anymore. Our extensions to QVT-O allow us to apply its efficient and compact imperative logic combined with the traceability establishment concepts of TGGs. These extensions enable user-edit-preserving incremental updates, which allow the application of the model transformation part in iterative settings. The semi-automatic establishment of explicit lifecycle traceability between CONSENS system models and MSD specifications enables all stakeholders conducting traceability-based model management activities. The algorithms enable the replicability to other MBSE and SwRE approaches, and the case study indicates the effectiveness and the efficiency of the technique for the Software Requirements Engineers.

4

Early Timing Analysis based on Software Requirements Specifications

In this chapter, we present a technique for simulative end-to-end response time analyses (cf. Section 2.6.2) based on MSDs [*Ber17] (cf. Figure 4.1). This technique enables Timing Analysts verifying timing-relevant execution platform properties against real-time requirements as part of MSD specifications. Thereby, platform-induced real-time requirement violations can be revealed so that the real-time requirements or the execution platform can be adapted during the early SwRE phase. Similar techniques enabling early timing analyses consider real-time requirements insufficiently [Fri17; Mey15; MH11], do not enable interactive simulations and provide insufficient platform modeling means [Has15; WT04; HY12], or require more detailed platform models close to the final implementation [LBD$^+$10; LK01; MNS$^+$17; MNL$^+$16; NIE$^+$17].

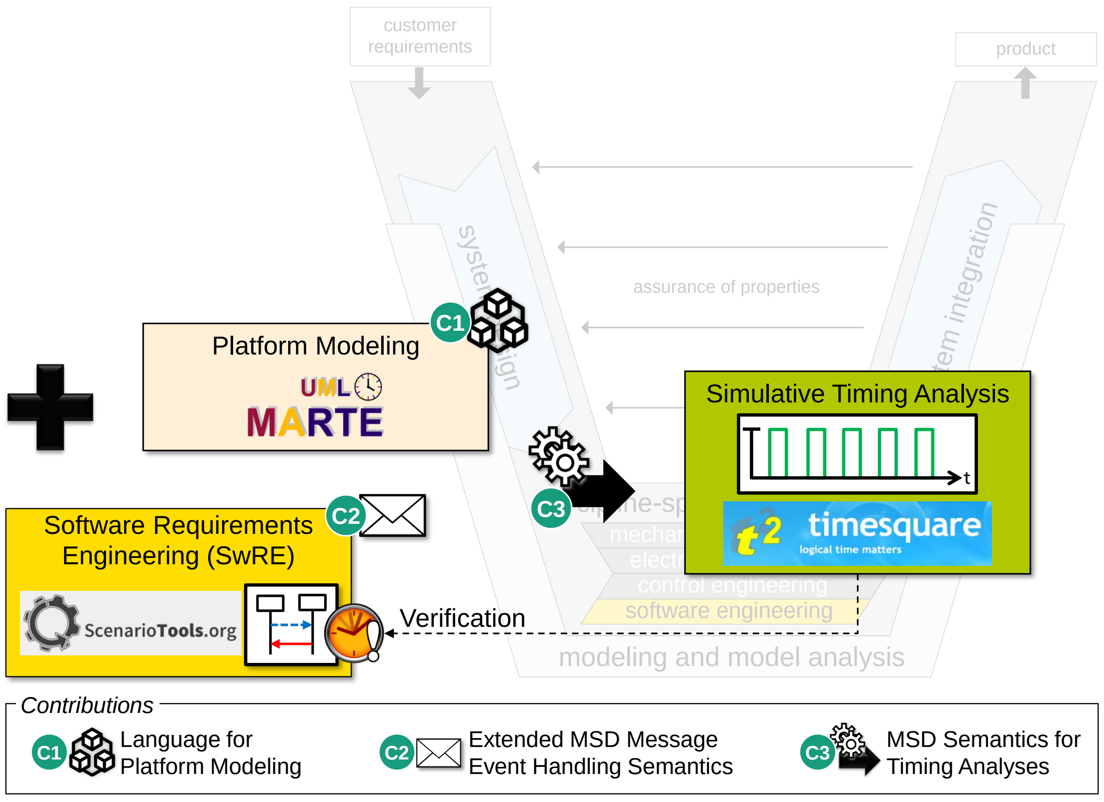


Figure 4.1: Early Timing Analysis based on Software Requirements Specifications

The main contribution of our technique is the specification of platform-aware MSD semantics dedicated to timing analyses (C3 in Figure 4.1, see Section 4.4). This semantics encompasses a subset of the conventional MSD semantics, an extended MSD message event handling semantics, and the platform properties' effects on the timing behavior. We apply the GEMOC approach (cf. Section 2.8) to declaratively specify the semantics, which makes the platform-induced timing effect behavior of MSD specifications explicit. Based on the semantics specification, GEMOC automatically derives models that are input to the timing simulation tool TIMESQUARE [DM12a; T^2].

Furthermore, we introduce the following supporting contributions:

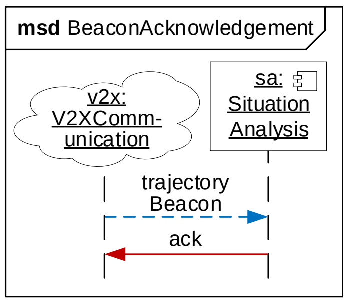
- We conceptually extend the message event handling semantics of MSDs by introducing additional event kinds that occur during the software execution on a target platform (C2 in Figure 4.1, see Section 4.3). This enables our end-to-end response time analyses to take the delays between different timing-relevant events into account.
- We propose a MARTE-based UML profile to enable platform modeling for MSDs (C1 in Figure 4.1, see Section 4.1). This encompasses specification means for platform models, for the allocation of component-based MSD specifications to platform models, and for annotating the resulting platform-specific models with timing-relevant properties. The separation of the platform models from the platform-independent MSD specifications enables to initially analyze the coordination behavior requirements and to subsequently validate the results by additionally considering the platform in a minimally invasive manner.
- We provide a process description including artifact dependencies and role responsibilities for the specification of the platform-independent and -specific parts of MSD specifications and for the conduct of timing analyses (cf. Section 4.2). This process description guides the particular engineering roles to perform their respective steps in a systematic way, where the task assignment to roles clarifies the distribution of their corresponding responsibilities.

4.1 Platform-specific MSD Specifications

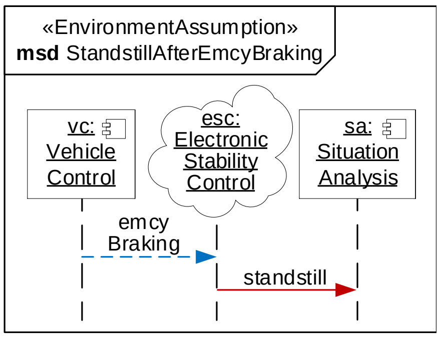
In Section 3.2, we introduced component-based MSD specifications in terms of platform-independent software requirements specifications. That is, the software architecture specified in the architecture view type has no correlation to any concrete target execution platform.

For example, Figure 4.2 depicts the interaction view of such a platform-independent MSD specification as a deliberately small variant of the EBEAS MSD specification. In this EBEAS variant, the vehicles exchange information about their particular trajectories among each other through trajectory beacons beyond the coordination of the emergency maneuvers. Figure 4.2(a) depicts an MSD specifying the v2x: V2XCommunication to send a `trajectoryBeacon` message to the sa: SituationAnalysis, which shall acknowledge the reception by sending back an `ack` message. Figure 4.2(c) depicts an MSD specifying the vc: VehicleControl to engage the esc: ElectronicStabilityControl for an emergency braking within 50 time units after the sa: SituationAnalysis was notified about an obstacle. The assumption MSD depicted in Figure 4.2(b) specifies the esc: ElectronicStabilityControl to notify the sa: SituationAnalysis when the vehicle has stopped through a `standstill` message after its engagement for an emergency braking.

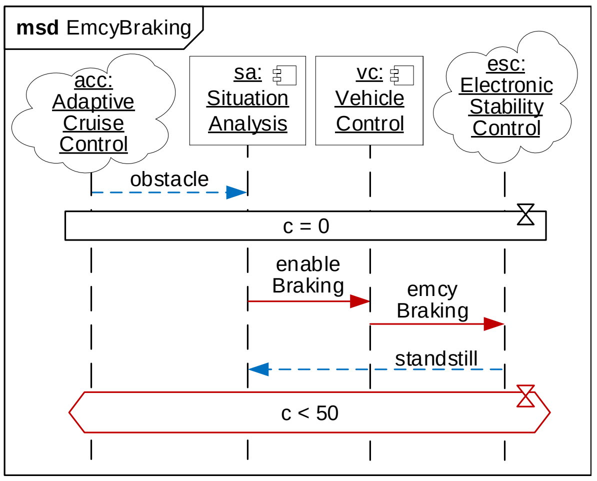
In this chapter, we want to consider the timing behavior emerging from the allocation of such component-based MSD specifications to concrete target execution platforms, which we



(a) MSD specifying an reception acknowledgment of a trajectory beacon



(b) Environment assumption specifying a standstill after an emergency braking



(c) MSD specifying an emergency braking engagement after an obstacle detection

Figure 4.2: Interaction view of an EBEAS MSD specification variant for a timing analysis

together call *platform-specific MSD specifications*. An end-to-end response time analysis has to determine whether such platform-specific MSD specifications can fulfill their high-level real-time requirements (cf. Section 2.6.2). One key question of such timing analyses is whether the resources provided by the platform have a sufficient performance to execute the application software consuming the resources. For example, the processing resources executing sa: SituationAnalysis and vc: VehicleControl may not be fast enough to process some operations in time, or the latency of the communication media could be too large to deliver some messages in time. Answering this question is exaggerated when dynamic situations with a high workload occur. For example, several messages like `obstacle` and `trajectoryBeacon` can arrive at sa: SituationAnalysis within a small time frame so that the receiving software component has to process them concurrently. Another example is the delivery of several messages via a communication medium at the same time.

In order to provide a modeling basis for our timing analysis approach in terms of a language for platform-specific MSD specifications, we present in this section the most important concepts of our Timing Analysis Modeling (TAM) UML profile. Furthermore, we illustrate it through an exemplary EBEAS platform-specific MSD specification as basis for the semantics specification in Section 4.4 and for an overall timing analysis example in Section 4.5.

The TAM profile enables to specify platform models, allocations of logical software components to the platform elements, and the specification of the platform properties that have to be considered in our timing analysis approach. The concepts introduced in the profile stem from a literature review on platform properties that influence the timing behavior of software-intensive systems on an abstraction level that is suitable during SwRE [*Ber17, Chapter 3].

The platform properties induce effects on the dynamic timing behavior of the system, and we encode these effects in the MSD semantics for timing analyses (cf. Section 4.4). The TAM profile extends the MARTE UML profile [OMG11] (cf. Section 2.5.3). Figure 4.3 depicts an excerpt of the TAM profile encompassing the most important concepts that we exemplarily use throughout this chapter. The profile is partitioned into the subprofiles ApplicationSoftware, Platform, and AnalysisContext. The Platform profile in turn is subdivided into the subprofiles ControlUnit, Communication, and OperatingSystem. We explain the particular stereotypes in the remainder of this section as we apply them to the example models, and we present the full profile definition in Section 4.6.1.

In Figure 4.4, we exemplarily apply the TAM profile for the specification of a target execution platform and its properties for the EBEAS. Thereby, we add platform-specific information to the variant of the platform-independent MSD specification derived from the CONSENS system model in the last chapter as depicted in Figure 4.2.

4.1.1 Specifying Execution Platforms

Beyond the three view types presented for component-based MSD specifications (cf. Section 3.2), we additionally introduce the *platform view type* for platform-specific MSD specifications. This view type is specified by means of the Platform subprofile of TAM (cf. Figure 4.3).

4.1.1.1 Specifying the Hardware

The TAM subprofile Platform::ControlUnit provides means to specify hardware elements of an execution platform (cf. Figure 4.3). For example, the Platform View in Figure 4.4 contains the micro controllers :µC1 and :µC2. The stereotype «TamECU» is applied to these modeling elements, which serves as a container for further hardware and operating system elements.

One important hardware element of a «TamECU» is the «TamProcessingUnit», which describes properties of the actual processing unit of an ECU or a micro controller. One of these properties is the amount of cores that the processing unit provides, specified by means of the tagged value numCores. For example, both the processing units :PUµC1 and :PUµC2 have one core. Another property and thereby tagged value is the speedFactor, which describes the relative speed w.r.t. to the normalized speed of a reference processing unit (cf. Section 2.5.3.2). For example, the processing unit :PUµC2 is two times faster than :PUµC1.

4.1.1.2 Specifying the Real-time Operating System

The TAM subprofile Platform::OperatingSystem provides means to specify aspects of real-time operating systems, which run on ECUs and provide services for the application software (cf. Figure 4.3). The stereotype «TamRTOS» as part of a «TamECU» describes properties of the

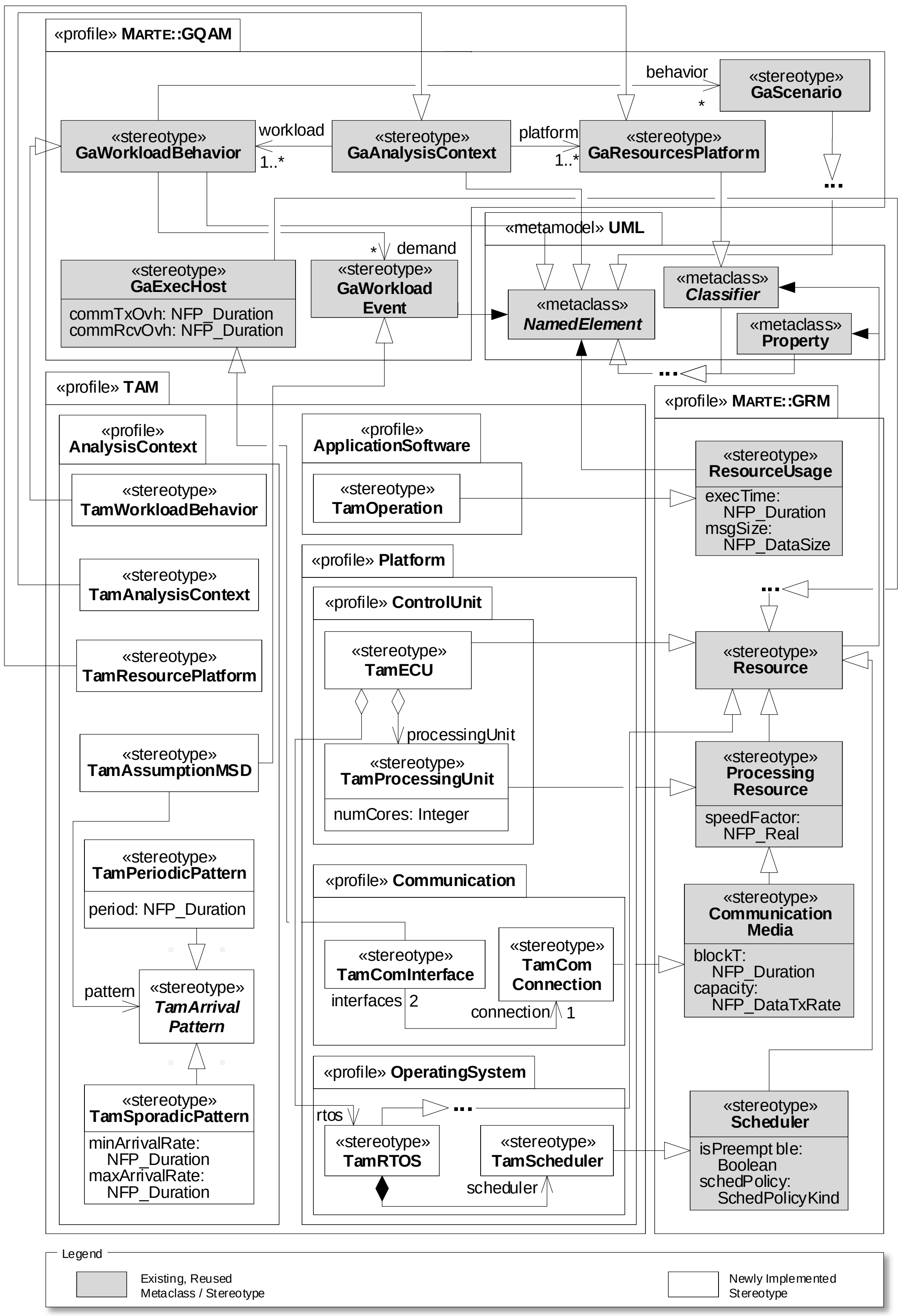

«profile» MARTE::GQAM
behavior
«stereotype» GaScenario
*
workload
platform
«stereotype» GaWorkloadBehavior
1..*
«stereotype» GaAnalysisContext
1..*
«stereotype» GaResourcesPlatform
...
«metamodel» UML
* demand
«stereotype» GaExecHost
commTxOvh: NFP_Duration
commRcvOvh: NFP_Duration
«stereotype» GaWorkload Event
«metaclass» NamedElement
«metaclass» Classifier
«metaclass» Property
...
«profile» TAM
«profile» MARTE::GRM
«profile» AnalysisContext
«profile» ApplicationSoftware
«stereotype» ResourceUsage
execTime: NFP_Duration
msgSize: NFP_DataSize
«stereotype» TamWorkloadBehavior
«stereotype» TamOperation
«profile» Platform
«stereotype» TamAnalysisContext
«profile» ControlUnit
...
«stereotype» TamECU
«stereotype» Resource
«stereotype» TamResourcePlatform
processingUnit
«stereotype» TamAssumptionMSD
«stereotype» TamProcessingUnit
numCores: Integer
«stereotype» Processing Resource
speedFactor: NFP_Real
«stereotype» TamPeriodicPattern
period: NFP_Duration
«profile» Communication
«stereotype» Communication Media
blockT: NFP_Duration
capacity: NFP_DataTxRate
pattern
«stereotype» TamArrival Pattern
«stereotype» TamComInterface
«stereotype» TamCom Connection
interfaces 2
connection 1
«profile» OperatingSystem
«stereotype» TamSporadicPattern
minArrivalRate: NFP_Duration
maxArrivalRate: NFP_Duration
rtos
...
«stereotype» TamRTOS
«stereotype» TamScheduler
scheduler
«stereotype» Scheduler
isPreemptble: Boolean
schedPolicy: SchedPolicyKind
Legend
Existing, Reused Metaclass / Stereotype
Newly Implemented Stereotype

Figure 4.3: Excerpt of the TAM profile (cf. full profile definition in Section 4.6.1)

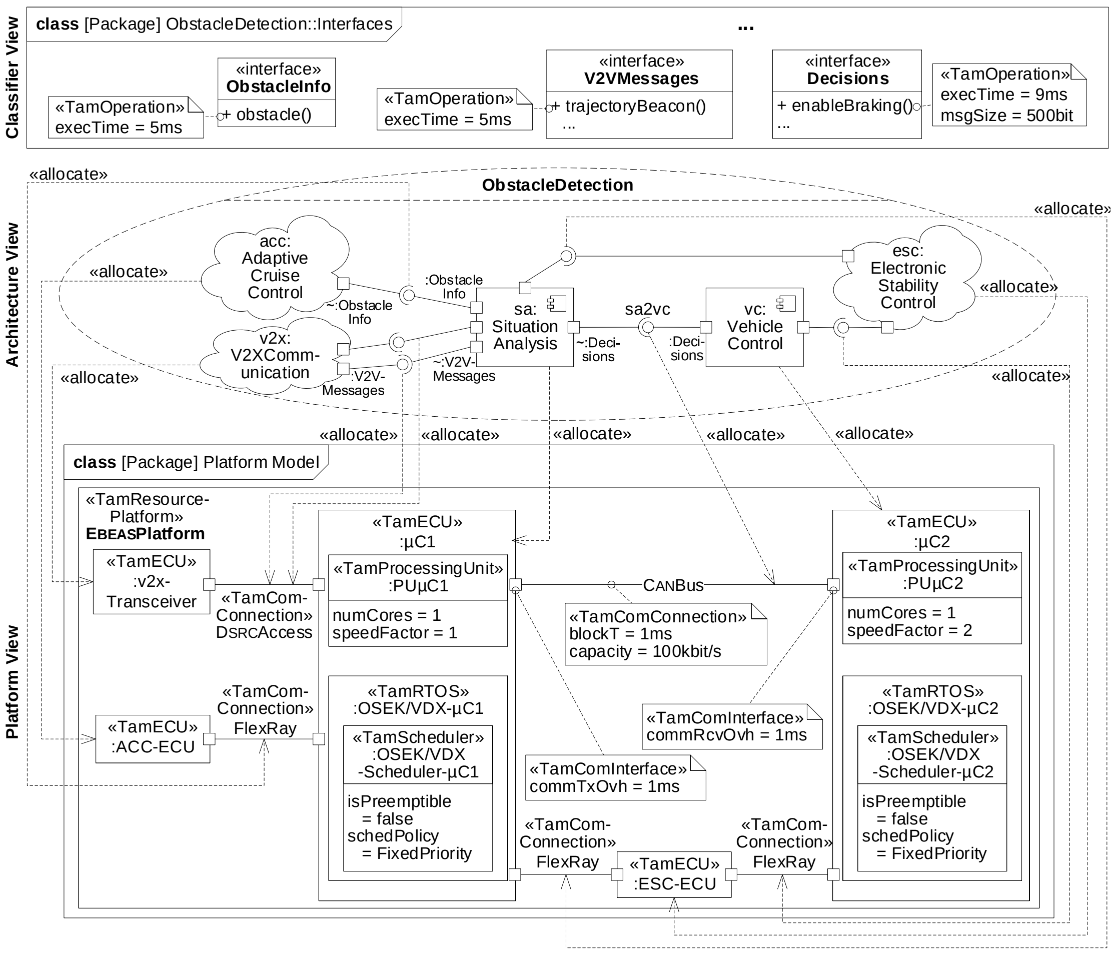


Figure 4.4: Platform-specific MSD specification excerpt for the EBEAS

real-time operating system of an ECU or a micro controller. We focus on its applied scheduler, specified through the stereotype «TamScheduler». The most important scheduler properties defining its strategy are the scheduling policy (tagged value schedPolicy) and whether the scheduler can preempt executing tasks (tagged value isPreemptible).

For example, the schedulers of both :PUμC1 and :PUμC2 implement the most prominent [NMH08; DB08; SAÅ+04] real-time operating system scheduling policy, FixedPriority. In this policy, all tasks have fixed priorities so that the scheduler dispatches the highest priority task of all tasks ready to execute after a task has finished. Furthermore, the property isPreemptible with the value false specifies that an already executing task cannot be preempted by another task. This scheduling strategy is supported by the widespread real-time operating systems of AUTOSAR [AUTOSAR] and OSEK/VDX [ISO05], for example.

4.1.1.3 Specifying Communication Facilities

The TAM subprofile Platform::Communication provides means to specify aspects of the communication facilities applied in an execution platform (cf. Figure 4.3). We provide the stereotype «TamComConnection» to specify the properties of a communication medium. The first im-

]

portant property of a communication medium is its latency, specified through the tagged value blockT. For example, the connector CANBus between both micro controllers has a latency of 1ms. The second important property of a communication medium is its throughput, specified through the tagged value capacity. For example, the throughput of the CANBus connector is 100kbit/s.

Furthermore, the network interfaces between a «TamECU» and a «TamComConnection» need time to encode messages from their logical application software representation to a representation suitable for the transport via a communication medium and vice versa. Such properties are captured as part of the stereotype «TamComInterface» for ports of TamECUs, inter alia. For example, the tagged value commTxOvh of the «TamComInterface» of :µC1's port connecting the CANBus specifies the timing overhead for encoding a message with 1ms. Analogously, the tagged value commRcvOvh of the «TamComInterface» of :µC2's port connecting the CANBus specifies the timing overhead for decoding a message with the same value.

4.1.2 Specifying Allocations

The MARTE subprofile Alloc provides means to allocate logical application software elements to resources of execution platforms (cf. Section 2.5.3.4). We apply the «allocate» stereotype to specify such allocations.

For example, the micro controllers :µC1 and :µC2 execute the software components sa: SituationAnalysis and vc: VehicleControl, respectively. This fact is expressed by means of allocation links from the logical software components to the micro controllers. Analogously, logical connectors between the software components in the architecture view type are allocated to «TamComConnection» links in the platform view type. For example, the logical connector sa2vc between sa: SituationAnalysis and vc: VehicleControl is allocated to the CANBus connecting :µC1 and :µC2.

4.1.3 Annotating the Application Software

The TAM subprofile ApplicationSoftware provides means to annotate information about the estimated resource consumption of the application software to the classifier view type of component-based MSD specifications (cf. Figure 4.3). Its most important element is the «TamOperation» stereotype, which serves for annotating the operations used as MSD message signatures with platform-specific properties. One of its properties is the execution time of an operation, specified by the tagged value execTime. For example, both the `obstacle` and `trajectoryBeacon` operations have an execution time of 5ms, whereas the `enableBraking` operation has an execution time of 9ms. These execution times are normalized w.r.t. a reference processing unit, where the speed of the particular processing units is specified as relative factor in relation to the speed of the reference processing unit (cf. Section 4.1.1.1). The second important property is the size of the corresponding message, specified by the tagged value msgSize. For example, the `enableBraking` operation has a message size of 500bit.

The direct annotation of the application software with platform-specific stereotypes is intended by MARTE. The alternative would be an allocation specification that contains the platform-specific information to further separate platform-independent and -specific aspects. However, this would result in cumbersome allocation specifications, in which, for example, any operation has to be allocated to a platform element separately so that the platform-specific information can be specified as part of the allocation.

4.1.4 Specifying Analysis Contexts

The TAM subprofile AnalysisContext provides means to specify concrete trigger situations for our timing analyses in terms of TIMESQUARE simulation runs (cf. Figure 4.3). More generally, such simulation scenarios are called *analysis contexts* [SG14, Chapter 9] (cf. Section 2.5.3.3).

For example, Figure 4.5 depicts an example of such an analysis context. The entry point for the specification of an analysis context is the «TamAnalysisContext» (e.g., the EBEASAnalysisContext). It references a platform (e.g., the EBEASPlatform depicted in Figure 4.4) as well as the concrete workload (e.g., EBEASWorkload) for the analysis context.

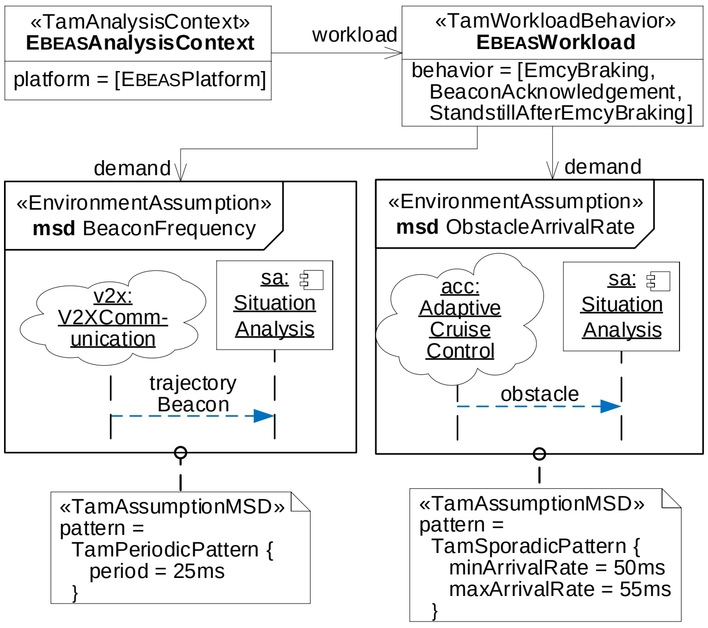


Figure 4.5: Analysis context example

The workload is specified through the stereotype «TamWorkloadBehavior». This stereotype references behavior elements specifying the system behavior to be considered in the analysis context and demand elements describing specific trigger scenarios (cf. Section 2.5.3.3). For example, the EBEASWorkload references the behavior MSDs depicted in Figure 4.2. Furthermore, the workload references demand scenarios, which are specified with the stereotype «TamAssumptionMSD». A «TamAssumptionMSD» specifies a scenario with typically one environment message triggering the system behavior, where the timing of the environment message is constrained by an arrival pattern. This arrival pattern is specified by the tagged value pattern, which references the abstract stereotype «TamArrivalPattern».

We support periodic and sporadic arrival patterns. A periodic arrival pattern, specified by the stereotype «TamPeriodicPattern» refining «TamArrivalPattern», constrains the environment message to occur periodically every period time units. For example, the «TamAssumptionMSD» BeaconFrequency depicted in Figure 4.5 specifies the `trajectoryBeacon` message to occur periodically every 25ms. A sporadic arrival pattern, specified by the stereotype «TamSporadicPattern» refining «TamArrivalPattern», constrains the environment message to occur sporadically between a minArrivalRate and/or a maxArrivalRate. For example, the «TamAssumptionMSD» ObstacleArrivalRate depicted in Figure 4.5 specifies the `obstacle` message to occur sporadically at some instant between any 50ms and 55ms.

4.2 Process Description

In this section, we provide an application view on our approach by means of a process description. This process description has the purpose to clarify which MSD analysis technique is applied in which development process phase and which information is needed by which engineering role as input for applying our timing analysis approach. In terms of MSD analysis techniques, we distinguish between the platform-independent MSD analysis techniques (cf. Section 2.4.3) and the timing analysis technique presented in this chapter. In terms of the needed information, the process describes which kind of information has to be provided by means of which part of our TAM profile (cf. Section 4.1) by which engineering role.

Figure 4.6 depicts the process for the conduct of our timing analysis approach by means of a BPMN diagram. The process has correlations to the integrated MBSE and SwRE process description depicted in Figure 3.6 in Section 3.3 and partially bases on the platform modeling and allocation engineering processes described by Pohlmann et al. [Poh18; PH18; PMDB14].

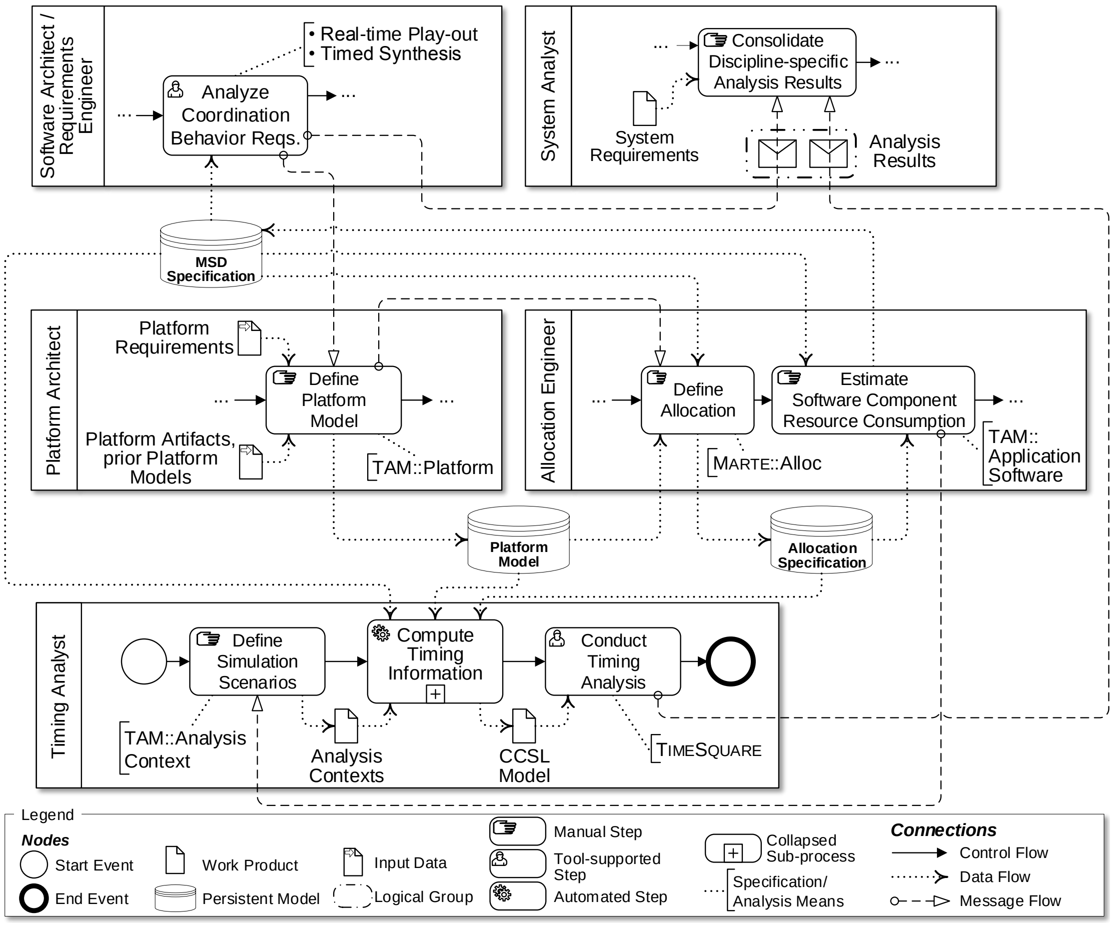


Figure 4.6: Process for conducting timing analyses based on MSDs (cf. Figure 3.6 in Section 3.3, partially based on [Poh18; PH18; PMDB14])

The process starts with the analysis of the requirements on the coordination behavior, which is also depicted in the integrated MBSE and SwRE process description in Figure 3.6 in Section 3.6. This step is conducted by the Software Requirements Engineer by means of the platform-independent MSD analysis techniques Real-time Play-out [*BGH$^+$14; *BBG$^+$13] and timed synthesis [*Jap15] (cf. Sections 2.4.3 and 3.6.1.5). Thus, we assume that the timed coordination behavior requirements that are input to our approach have a high quality from a platform-independent point of view and are, for example, consistent and realizable.

The *Platform Architects* are responsible for the overall platform, which mainly includes computing resources like ECUs, communication media like bus systems, and their topology. Thus, they participate also intensively in the definition of the *Active Structure* conducted by the System Designer role (cf. Figure 3.6 in Section 3.3). In order to describe the platform, the Platform Architects define a Platform Model. One input for this step are Platform Requirements, which include technological considerations (e.g., performance, compatibility topics, best practice experiences) and economic requirements (e.g., customer requirements, pricing, availabilities). Another input encompasses Platform Artifacts (i.e., the particular ECUs and bus systems as well as their respective properties) and practice-proven prior Platform Models from previously conducted development projects. We provide the subprofile TAM::Platform for the specification of the Platform Model (cf. Section 4.1.1 for an introduction and Section 4.6.1.1 for the complete profile definition).

The *Allocation Engineers* are responsible for allocating the particular application software components to the particular ECUs as well as for allocating the logical software component interconnections to buses. They have the knowledge and experience about the resource consumption of the software architecture parts and about the available resources of the platform and hence are able to plan an adequate allocation and estimate the resulting resource consumption. The Allocation Engineers first conduct the step Define Allocation, which has the application software components and their logical interconnections defined in the MSD Specification as well as the ECUs and communication media defined in the Platform Model as input. The step outputs an Allocation Specification, which is a separate model importing elements of the MSD Specification and of the Platform Model and defines allocations between these imported elements. The Allocation Engineers apply the MARTE::Alloc subprofile for this step (cf. Sections 2.5.3 and 4.1.1).

Based on the allocations, the Allocation Engineers can estimate the resource consumption of the software components w.r.t. the available resources of the ECUs and the communication media in the subsequent step. The output is a set of annotations for the software component types and interfaces as part of the classifier view type of the MSD Specification. These annotations are specified by means of the subprofile TAM::ApplicationSoftware (cf. Sections 4.1.3 and 4.6.1.1).

The *Timing Analysts* are responsible for conducting timing analyses. For this purpose, they define in the first step simulation scenarios, which describe concrete trigger situations for TIMESQUARE simulation runs. These simulation scenarios are more generally called Analysis Contexts [SG14, Chapter 9] (cf. Section 2.5.3.3) and are specified by means of the subprofile TAM::AnalysisContext (cf. Sections 4.1.4 and 4.6.1.1).

The Analysis Contexts and the models specified in the previous steps (i.e., the MSD Specification, the Platform Model, and the Allocation Specification) are input to the collapsed *sub-process* Compute Timing Information. This step is fully automatic and stems from our specification of semantics for timing analyses in GEMOC (cf. Sections 2.8 and 4.4). We expand this *sub-process* and explain it with more implementation details in Section 4.6.1.

Finally, the Timing Analysts perform the step Conduct Timing Analysis by means of the simulation tool TIMESQUARE. As it is the case for other discipline-specific results, the Timing Analysts hand over and discuss their Analysis Results with the System Analysts as part of the

Systems Engineer role (cf. Section 3.3). We provide details on the timing analysis conduct in the overall simulation example in Section 4.5 and the evaluation of our approach in Section 4.6.2.

4.3 Extension of MSD Message Event Handling Semantics

As explained in Section 2.4.2, message events are synchronously unified with complete MSD messages in the platform-independent semantics of MSDs and thereby in the MSD analysis techniques. This abstraction is well-suited to initially consider an idealized system but not adequate for detailed end-to-end response time analyses (cf. Section 2.6.2). Such analyses require the consideration of several events per message occurring during the execution on a platform in order to take the particular delays in between into account.

Thus, we introduce in this section additional event kinds for the purpose of platform-aware timing analyses based on MSDs. We associate each event kind with an equally named MSD lifeline location kind (cf. Section 2.4), which leads to a more fine-grained cut progression. Furthermore, we argue which delays between the event kinds we have to take into account in timing analyses based on the four components of real-time communication end-to-end delays defined by Tinkell et al. [TBW95]. We present a coarse-grained computation of these delays in Section 4.4.2.1 and the detailed computation in Section 4.6.1.2.

We exemplarily visualize the location kinds, the fine-grained cuts, and the delays for the MSD messages `enableBraking` and `emcyBraking` in Figure 4.7 and explain them in the remainder of this section.

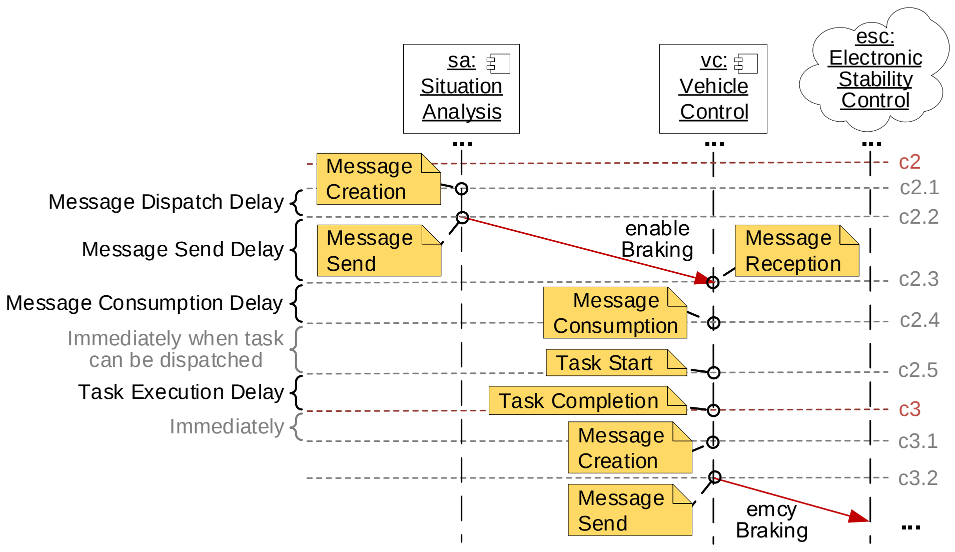


Figure 4.7: Additional events for MSD messages

4.3.1 Asynchronous Messages

The semantics for platform-independent MSD specifications only considers synchronous message events, where the sending and reception of a message in the object system at runtime happen simultaneously (cf. Section 2.4.2). However, the used communication media and communication protocols (e.g., the properties of the connector CANBus in Figure 4.4) cause a *mes-*

sage send delay, which must be taken into account by a timing analysis [TBW95]. Thus, there is the need for the consideration of asynchronous message events, which distinguish between the sending and the reception of messages.

In order to consider such events, we apply the concept of Harels original Play-out semantics [HM03a] for Live Sequence Charts (LSCs) [DH01], which distinguishes synchronous and asynchronous messages. That is, we introduce asynchronous messages by distinguishing between *message send events* and *message reception events*. These event kinds are not synchronously unified with a whole MSD message but with the message send and the message reception location of the corresponding MSD message, respectively. Figure 4.7 visualizes these location kinds exemplarily for the MSD message `enableBraking`. Here, the cut c2.2 marks that the message is sent but not yet received whereas the cut c2.3 marks that the message is received.

4.3.2 Message Creation and Consumption

The notion of message reception in scenario-based formalisms is ambiguous, because it is not clear whether a message reception is the instant when the message arrives at the receiver communication interface or the instant when the receiver application software consumes the message [HHRS05]. The distinction between message reception and message consumption is necessary in order to take *message consumption delays* into account [TBW95]. Such delays occur due to the decoding of network messages from a representation suitable for the transport via a network to a logical representation suitable to be processed by the application software. Similarly, there is a *message dispatch delay* between the instant when a message is created by the sending software component and the instant when it is actually dispatched to the network by its network interface [TBW95]. Such delays occur due to the encoding of a logical representation into a representation suitable for the transport via a communication medium.

In order to distinguish between message creation and message sending as well as between message reception and message consumption, we introduce two additional message event kinds, namely the *message creation event* and the *message consumption event*. These event kinds capture the instant when a message is created by the sending software component and consumed by the receiving software component, respectively. Likewise, we introduce two additional location kinds for each MSD message, namely the message creation location and the message consumption location. Figure 4.7 visualizes these locations exemplarily for the MSD message `enableBraking`. We define the message creation location to be positioned on the sending lifeline directly before the message sending location (cf. cut c2.1). Similarly, the message consumption location is positioned on the receiving lifeline directly after the message reception location (cf. cut c2.4).

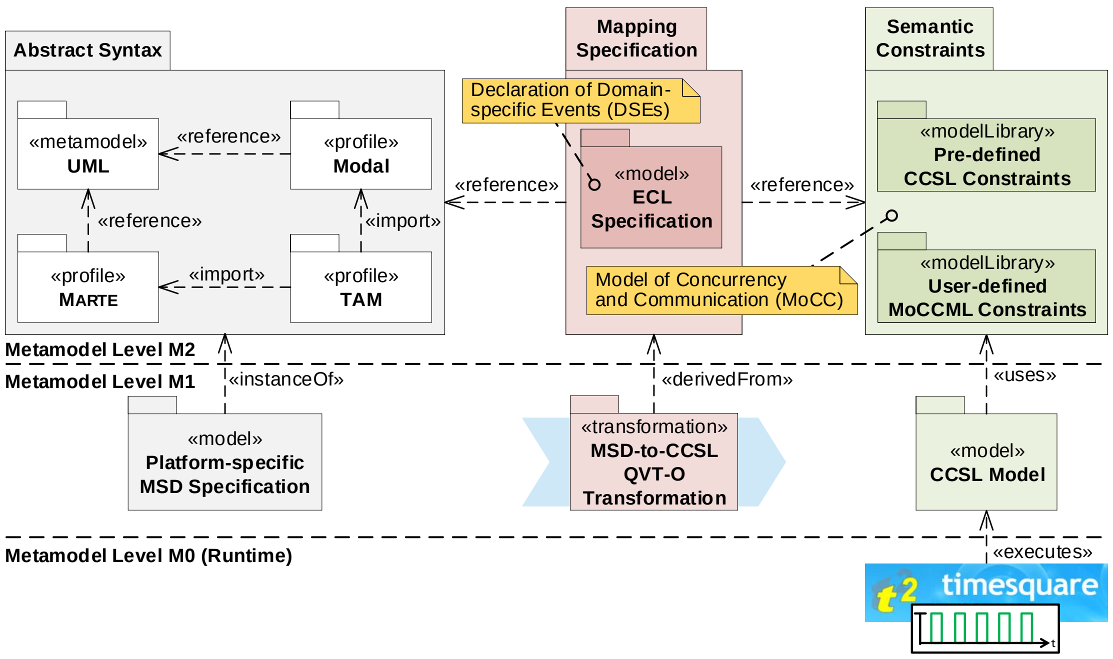
4.3.3 Task Processing

The semantics for platform-independent MSD specifications focuses on the message exchange between software components. However, it neglects internal procedures (i.e., tasks) that are executed by the software components to process consumed messages and to create the messages to be sent. Task execution interferences lead to additional *task execution delays* that affect the timing behavior of the system [TBW95] (cf. the execution times of the particular software operations in Figure 4.4).

In order to consider such effects, we do not specify explicit task models but simplifying define that each message is associated with exactly one task that is executed upon the consumption of the message by the receiving software component. That is, we introduce the two new event kinds *task start event* and *task completion event*. These event kinds represent the start and the end of the execution of the task that processes a consumed message and creates a message to be sent. Furthermore, we define the corresponding two equally named location kinds. Figure 4.7 visualizes these location kinds exemplarily for the MSD message `enableBraking`. We define the task start location to be positioned on the receiving lifeline directly after the message consumption location (cf. cut c2.5). Similarly, the task completion location is positioned on the receiving lifeline directly after the task start location, representing also the cut for the next MSD message (cf. cut c3). The next location is the message creation location (cf. cut 3.1), and so on.

4.4 MSD Semantics for Timing Analyses

Our goal is to derive CCSL models (cf. Section 2.7) from platform-specific MSD specifications (cf. Section 4.1) with extended event handling semantics (cf. Section 4.3) in order to enable simulative timing analyses in TIMESQUARE. To this end, we apply the GEMOC approach (cf. Section 2.8) in order to specify MSD semantics dedicated to timing analyses in terms of CCSL. Figure 4.8 gives an overview of our application of the GEMOC approach.



Abstract Syntax
«metamodel» UML
«reference»
«profile» Modal
«reference»
«import»
«profile» MARTE
«import»
«profile» TAM
Mapping Specification
Declaration of Domain-specific Events (DSEs)
«reference»
«model» ECL Specification
«reference»
Model of Concurrency and Communication (MoCC)
Semantic Constraints
«modelLibrary» Pre-defined CCSL Constraints
«modelLibrary» User-defined MoCCML Constraints
Metamodel Level M2
Metamodel Level M1
«instanceOf»
«derivedFrom»
«uses»
«model» Platform-specific MSD Specification
«transformation» MSD-to-CCSL QVT-O Transformation
«model» CCSL Model
Metamodel Level M0 (Runtime)
«executes»
timesquare

Figure 4.8: Specifying MSD semantics for timing analyses with GEMOC

The abstract syntax of platform-specific MSD specifications is defined at the metamodel level (M2) by several parts of the UML metamodel and the Modal profile (cf. Section 2.5.1). We extend this abstract syntax by our TAM profile introduced in Section 4.1 and presented in detail in Section 4.6.1.1, which bases on parts of the MARTE profile (cf. Section 2.5.3). The declaration of domain-specific events (DSEs) and their mapping between the abstract syntax and the MoCC are specified by means of the Event Constraint Language (ECL) (cf. Section 2.8). The MoCC is specified through a set of semantic constraints, encompassing pre-defined CCSL constraints (cf. Section 2.7.2) as well as user-defined MoCCML constraints (cf. Section 2.7.3). These constraints are referenced by the ECL specification and are used to constrain the DSEs defined in the context of metaclasses from the abstract syntax (cf. Section 2.8).

The GEMOC studio automatically derives a QVTo [OMG16] model transformation from our MoCC. This model transformation takes platform-specific MSD specifications as inputs and encodes the functional and real-time requirements as well as the timing-relevant platform properties into constrained timing effects as part of a CCSL model. The Timing Analyst can simulate such CCSL models in TIMESQUARE in order to reveal potential real-time requirement violations.

We do not intend to encode the full MSD semantics into our timing analysis semantics, because this would result in a very fine-grained analog of Real-time Play-out in TIMESQUARE. Such a low level Real-time Play-out would be cumbersome and inefficient due to the high amount of the more fine-grained event kinds (cf. Section 4.3) occurring in the platform-specific simulation. Instead, we cover only the MSD semantics subset that is required to conduct timing analyses w.r.t. the real-time requirements as part of the MSD specifications. This subset covers real-time requirements constraining the unification of message events (cf Section 2.4.2) and, consequently, the distinction of message events and hot and cold MSD messages for encoding the unification concept. Thus, we foster a process in which the coordination behavior requirements are validated for safety and liveness on a higher abstraction level by means of the platform-independent MSD analysis techniques before our timing analysis approach is applied (cf. Section 4.2).

In the following, we overview how we specify the MSD semantics dedicated to timing analyses. For reasons of comprehensibility, we describe the most important concepts of this semantics through illustrating the relationships between abstract syntax, mapping specification, and semantic constraints at metamodel level M2. Furthermore, we illustrate the implications on metamodel level M1 and provide corresponding exemplary CCSL simulation runs at metamodel level M0. We present illustrations of further concepts in Appendix B.1 and the complete semantics in Appendix B.2.

Section 4.4.1 describes how we encode the message event unification concept of MSDs with extended message event handling semantics in terms of CCSL. Section 4.4.2 describes how we encode the timing effects induced by the platform properties. Section 4.4.3 describes how we encode real-time requirements on these effects and how we encode concrete timing analysis contexts.

4.4.1 Encoding of Additional Event Kinds and their Unification

In order to enable timing analyses based on MSD specifications, we have to encode a subset of the basic semantics of MSDs in terms of CCSL. This subset encompasses the unification of events with hot and cold MSD messages, enabling the separation of the actual timed system behavior and the requirements imposed on it. Particularly, we encode the additional event kinds introduced in Section 4.3 in our semantics for timing analyses in order to consider the delays between these event kinds.

In order to encode the unification concept of the MSD semantics, we have to distinguish between MSD messages and message events. As noted in Section 2.8, GEMOC requires to distinguish the respective abstract syntax elements to specify dedicated semantics for both model levels. Thus, we distinguish between the *occurrences of a unification* of message events with MSD messages and the occurrences of the triggering message events. Whereas the MSD language provides MSD messages as an abstract syntax anchor for unification occurrences, there is no abstract syntax element dedicated to message events since the object system including metamodel is derived as a separate runtime model in SCENARIOTOOLS MSD Play-out (cf. Section 2.4.1). Thus, we introduce an explicit abstract syntax element for this purpose.

In Section 4.4.1.1, we present the encoding of occurrences of a unification of the particular event kinds with the corresponding MSD message locations. Subsequently, we present the abstract syntax element dedicated to the particular event kind occurrences and its relation to the unification occurrences in Section 4.4.1.2. Note that the timing of the event occurrences does not matter for this semantics part, so that we assume arbitrary instants throughout this section.

4.4.1.1 Unification Occurrences

In this section, we present how we encode occurrences of the unification of the particular event kinds with the corresponding message location kinds (cf. Section 4.3). That is, we explicitly define DSEs and constraints that define in which cases and in which order unifications may occur (e.g., the unification of a send event with the send location of the corresponding MSD message, of a reception event with a reception location, etc.). The consecutive order of unification occurrences represents the fine-grained cut progression w.r.t. the particular MSD message locations (cf. Figure 4.7).

Figure 4.9 depicts a specification excerpt of the semantics for unification occurrences at metamodel level M2 (upper part of the figure) as well as corresponding example models at metamodel level M1 (lower part of the figure). Figure 4.10 depicts an exemplary corresponding CCSL run at metamodel level M0. We exemplarily focus on the semantics for hot messages. We provide specification excerpts, example models, and example CCSL runs for cold unification occurrences in Appendix B.1 (cf. Figures B.2, B.3, and B.6) and the complete semantics specification in Appendix B.2.

Metamodel Level M2

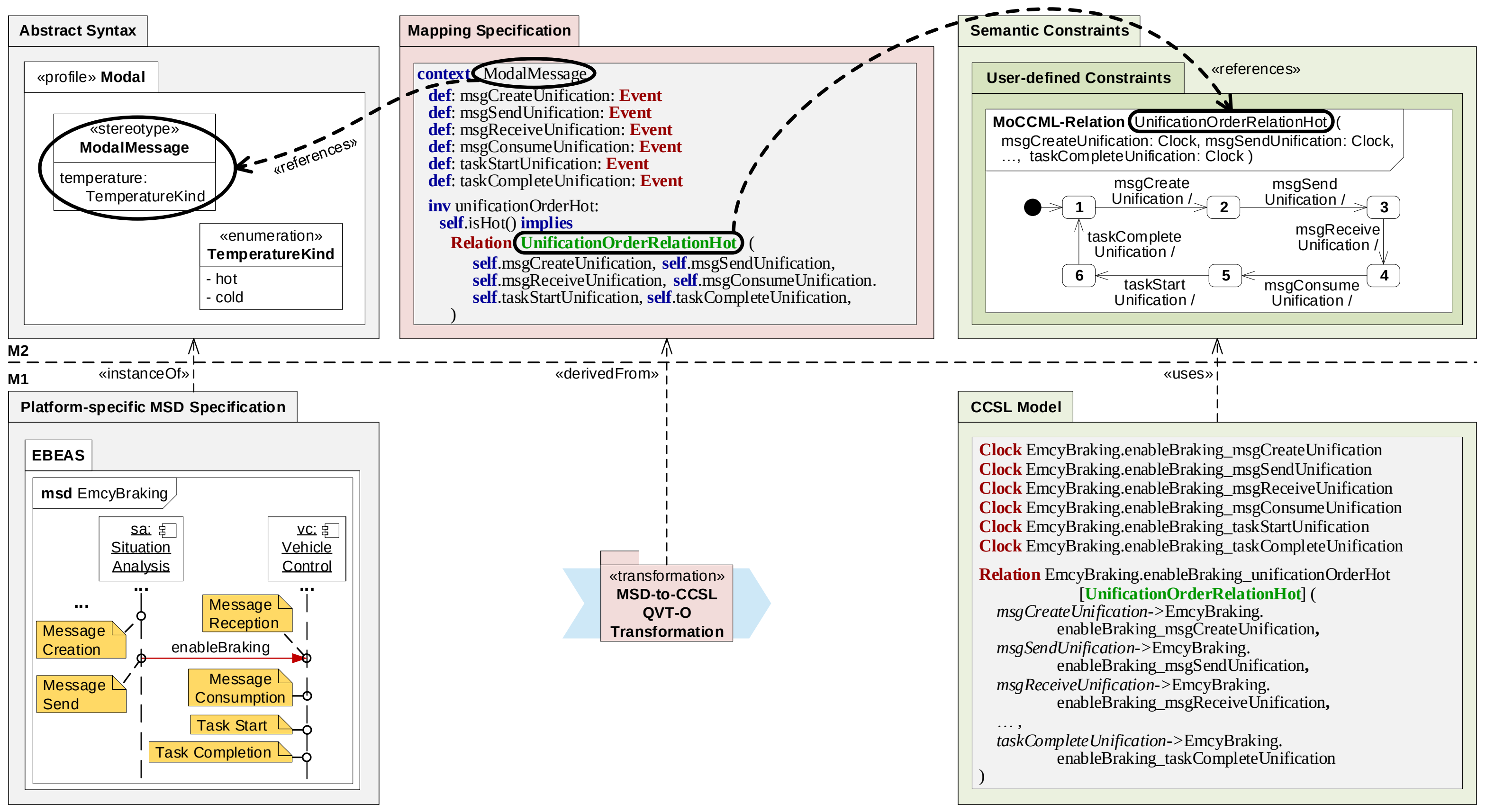
The upper part of Figure 4.9 depicts excerpts of the Modal profile, of the ECL Mapping Specification, and of the applied semantic constraints.

The ECL Mapping Specification in the middle upper part of Figure 4.9 defines DSEs and an invariant for the Modal stereotype «ModalMessage». The DSEs introduce potential occurrences of the unification for the particular event kinds. The invariant unificationOrderHot specifies the allowed order of these unification occurrences by means of a reference to the user-defined MoCCML relation UnificationOrderRelationHot. Its arguments are all unification occurrence DSEs in the order from the unification of message creation events to the unification of task completion events.

Analogously, the parameters of the MoCCML relation unificationOrderHot are clocks representing the unification occurrences for the particular message event kinds. The constraint automaton of the relation defines the allowed order of these parameter clocks. That is, it specifies that the particular event kinds must be unified with the corresponding hot message location kinds in the order message creation, message sending, message reception, message consumption, task start, and task completion.

Metamodel Level M1

The lower part of Figure 4.9 exemplarily depicts excerpts of the Platform-specific MSD Specification and of the corresponding CCSL Model generated through the automatically derived MSD-to-CCSL QVT-O Transformation.



```
context ModalMessage
  def: msgCreateUnification: Event
  def: msgSendUnification: Event
  def: msgReceiveUnification: Event
  def: msgConsumeUnification: Event
  def: taskStartUnification: Event
  def: taskCompleteUnification: Event

  inv unificationOrderHot:
    self.isHot() implies
      Relation UnificationOrderRelationHot (
        self.msgCreateUnification, self.msgSendUnification,
        self.msgReceiveUnification, self.msgConsumeUnification.
        self.taskStartUnification, self.taskCompleteUnification,
      )
```

```
MoCCML-Relation UnificationOrderRelationHot(
  msgCreateUnification: Clock, msgSendUnification: Clock,
  …, taskCompleteUnification: Clock )
```

```
Clock EmcyBraking.enableBraking_msgCreateUnification
Clock EmcyBraking.enableBraking_msgSendUnification
Clock EmcyBraking.enableBraking_msgReceiveUnification
Clock EmcyBraking.enableBraking_msgConsumeUnification
Clock EmcyBraking.enableBraking_taskStartUnification
Clock EmcyBraking.enableBraking_taskCompleteUnification

Relation EmcyBraking.enableBraking_unificationOrderHot
              [UnificationOrderRelationHot] (
  msgCreateUnification->EmcyBraking.
          enableBraking_msgCreateUnification,
  msgSendUnification->EmcyBraking.
          enableBraking_msgSendUnification,
  msgReceiveUnification->EmcyBraking.
          enableBraking_msgReceiveUnification,
  … ,
  taskCompleteUnification->EmcyBraking.
          enableBraking_taskCompleteUnification
)
```

Figure 4.9: Specification excerpt of semantics for the order of unification occurrences for hot messages with additional event kinds

The MSD excerpt in the left lower part shows the MSD message `enableBraking` and its particular event kinds as part of the MSD EmcyBraking.

As indicated in the generated CCSL Model excerpt in the right lower part of Figure 4.9, the derived transformation translates any MSD message of all MSDs to each six clock variables. These clock variables represent the potential unification occurrences of the particular event kinds with the particular locations of any MSD message `enableBraking` of all MSDs. For example, the MSD message `enableBraking` of the MSD EmcyBraking is translated to six clock variables ranging from EmcyBraking.enableBraking_msgCreateUnification to EmcyBraking.enableBraking_taskCompleteUnification.

Furthermore, the transformation generates for any MSD message of all MSDs each a CCSL model clock relation using the MoCCML relation UnificationOrderRelationHot specified at metamodel level M2. This clock relation gets the six clock variables representing the potential unification occurrences as arguments. For example, the transformation translates the MSD message `enableBraking` of the MSD EmcyBraking to the CCSL model relation EmcyBraking.enableBraking_unificationOrderHot using the MoCCML relation UnificationOrderRelationHot. This relation gets the arguments EmcyBraking.enableBraking_msgCreateUnification for the parameter msgCreateUnification, EmcyBraking.enableBraking_msgSendUnification for the parameter msgSendUnification, and so on.

Metamodel Level M0

Figure 4.10 depicts an exemplary CCSL run resulting from the CCSL model excerpt depicted in the right lower part of Figure 4.9. This CCSL run simulates the order of unification occurrences of the particular event kinds with the corresponding locations of the MSD message `enableBraking` of the MSD EmcyBraking.

All rows depict ticks of the particular clocks representing the unification occurrences for the corresponding MSD message locations. They are ordered from the top to bottom, where the topmost row represents the unification of a message creation event and the bottommost row represents the unification of a task completion event. The ticks correspond to the transitions in the MoCCML relation UnificationOrderRelationHot., and the We assume that at instant 0 the MoCCML relation is in state 1, so that due to the tick of the clock $\text{EmcyBraking.enableBraking}_{\text{msgCreateUnification}}$ the state is changed to 2. After the subsequent tick of the clock representing the message send unification occurrence at instant 2, the relation is in state 3 for the next 3 instants, and so on. The arrows visualize the causal/temporal dependencies between the clock ticks.

The concrete instants are chosen arbitrarily since the exact timing of the unification occurrences does not matter for this semantics part, which only considers causal/temporal event dependencies.

4.4.1.2 Unification of Message Events with MSD Message Locations

Based on the last section describing the encoding of the allowed orders of unification occurrences, we describe in this section how we encode the actual unification of the particular event kind with the corresponding MSD message location kinds. That is, we describe how we encode the relationship between the actual system behavior event occurrences and the unification occurrences that represent the cut progression of an MSD.

Figure 4.12 depicts a specification excerpt of the semantics for event unification at metamodel level M2 (upper part of the figure) as well as corresponding example models at metamodel level

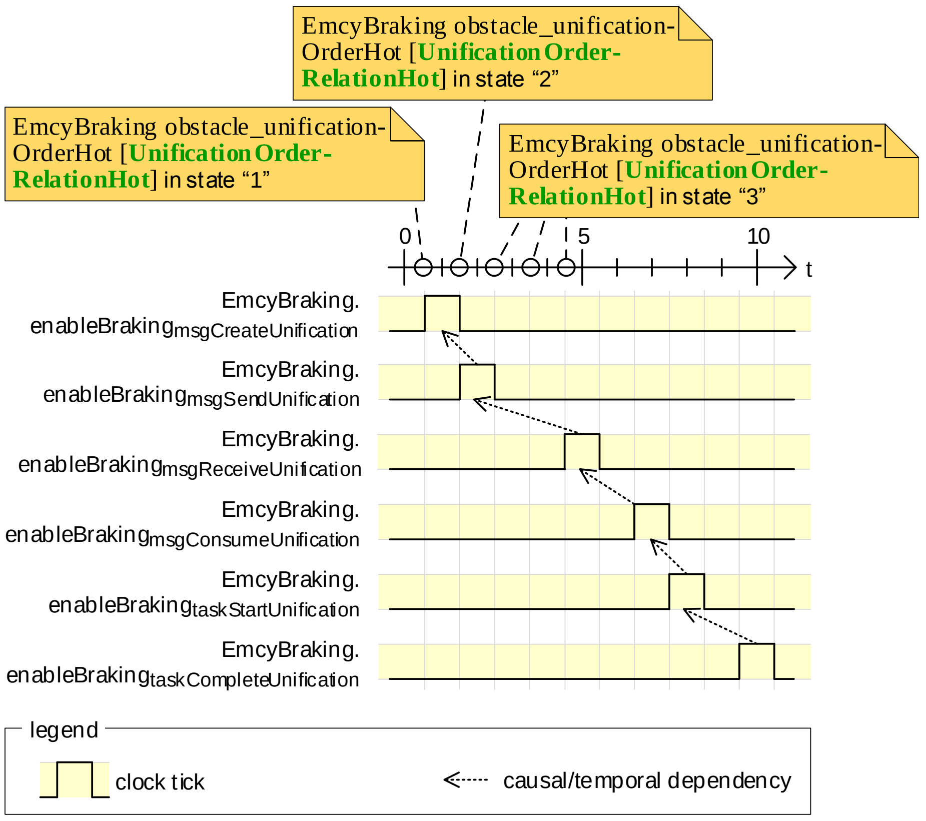

EmcyBraking obstacle_unification-OrderHot [UnificationOrder-RelationHot] in state "2"
EmcyBraking obstacle_unification-OrderHot [UnificationOrder-RelationHot] in state "1"
EmcyBraking obstacle_unification-OrderHot [UnificationOrder-RelationHot] in state "3"
0
5
10
t
EmcyBraking.enableBraking msgCreateUnification
EmcyBraking.enableBraking msgSendUnification
EmcyBraking.enableBraking msgReceiveUnification
EmcyBraking.enableBraking msgConsumeUnification
EmcyBraking.enableBraking taskStartUnification
EmcyBraking.enableBraking taskCompleteUnification
legend
clock tick
causal/temporal dependency

Figure 4.10: Example—order of message unification occurrences in CCSL, enforced by the CCSL model clock relation EmcyBraking.enableBraking_unificationOrderHot (cf. CCSL model in the lower right of Figure 4.9)

M1 (lower part of the figure). Figure 4.13 depicts an exemplary corresponding CCSL run at metamodel level M0. We exemplarily focus on the unification of send and receive events with hot send and receive locations. We provide specification excerpts, example models, and example CCSL runs for the unification of send and receive events with cold send and receive locations in Appendix B.1 (cf. Figures B.4 to B.6). However, we first introduce a new abstract syntax element in the following.

Metamodel Level M2

The Play-out algorithm for MSDs requires no explicit modeling concept for the specification of message events. That is, message events occurring at runtime (i.e., metamodel M0) in the object system are directly unified with MSD messages at metamodel level M1. However, the GEMOC approach requires an abstract syntax concept in order to define semantic constraints for such runtime events at metamodel level M2 (cf. Section 2.8). Thus, we introduce such a dedicated abstract syntax concept to enable the definition of the semantics for the unification of runtime events.

For this purpose, we follow the definitions of Harel's and Marelly's unification concept for Live Sequence Charts (LSCs) [HM03a]. Beyond LSC messages and system events (analogous to MSD messages and message events, respectively), they define *system messages*. System messages associate a sender and receiver object in the actual system (analogous to the object system for MSD specifications) and define which system events can occur between sender and receiver at runtime. A system message is associated by an LSC message, where multiple LSC messages in different LSCs can associate the same system message. Likewise, a system event associates a system message.

For the application of the LSC system message concept as explicit abstract syntax element in GEMOC, we introduce the *object system message* as stereotype in the TAM profile. Figure 4.11 depicts an abstract overview of the additional TAM subprofile SimulationExtensions, which defines the corresponding stereotype «ObjectSystemMessage» (cf. the full subprofile definition in Section 4.6.1.1). This stereotype extends the UML metaclass **MessageEvent**, where UML events do not describe an actual event occurrence at runtime "but can be considered a classification of its occurrences" [OMG17b, Section 13.3.1]. Thus, UML message events fit to our definition of object system messages. An object system message associates a sender role, a receiver role, a connector between sending and receiving role, an interface signature, and multiple MSD messages. We determine the object system messages and its associations automatically from the MSD specifications in a preprocessing step before generating the CCSL models (cf. Section 4.6.1.2).

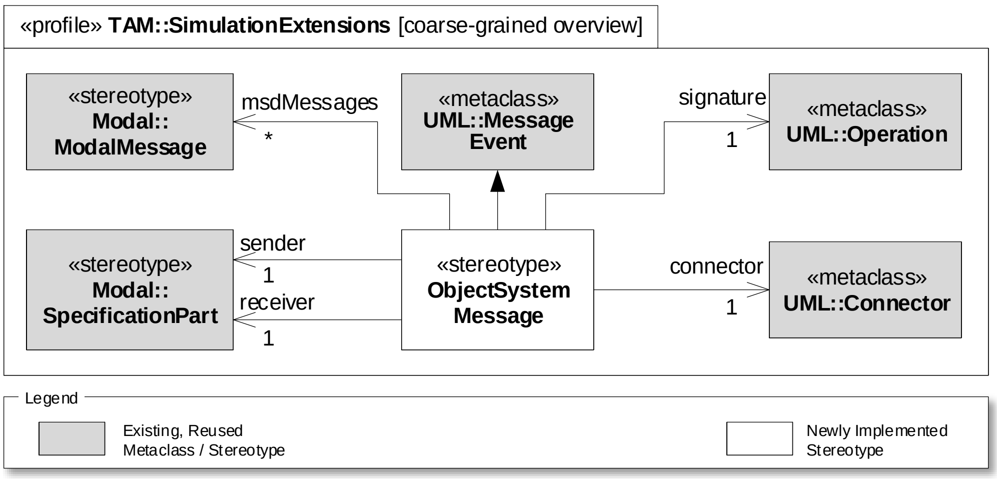
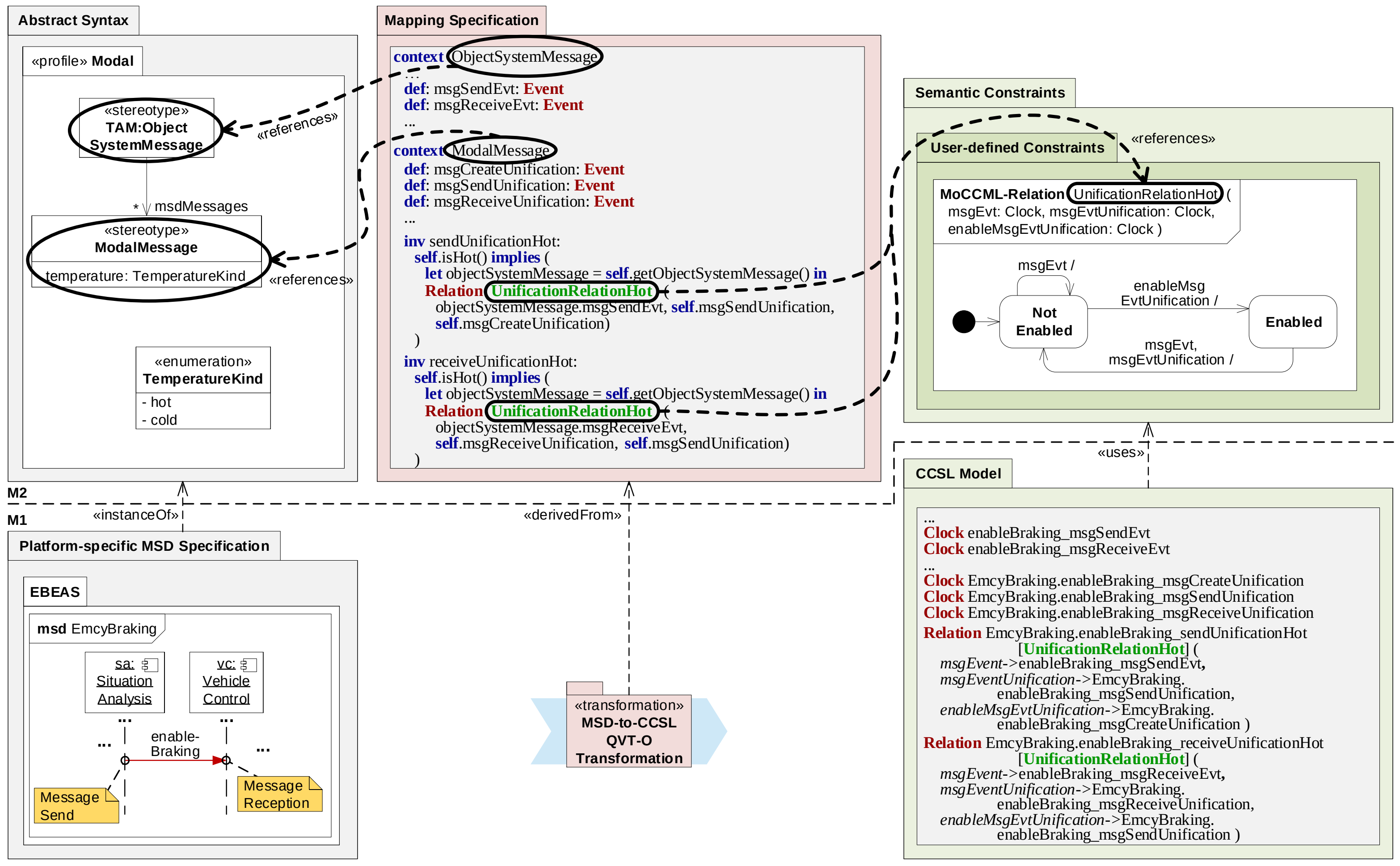

«profile» TAM::SimulationExtensions [coarse-grained overview]
«stereotype» Modal:: ModalMessage
msdMessages
*
«metaclass» UML::Message Event
signature
1
«metaclass» UML::Operation
«stereotype» Modal:: SpecificationPart
sender
1
receiver
1
«stereotype» ObjectSystem Message
connector
1
«metaclass» UML::Connector
Legend
Existing, Reused Metaclass / Stereotype
Newly Implemented Stereotype

Figure 4.11: The object system message as part of the TAM subprofile SimulationExtensions

Based on the introduction of the «ObjectSystemMessage» stereotype, the upper part of Figure 4.12 depicts excerpts of the TAM profile, the ECL Mapping Specification, and of the applied semantic constraints.

The ECL Mapping Specification in the middle upper part of Figure 4.12 defines DSEs for the TAM stereotype «ObjectSystemMessage» introduced above. Analogously to the concept of Harel's and Marelly's LSC system messages defining which LSC system events can occur at runtime, these «ObjectSystemMessage» DSEs define message events that can occur at runtime during a CCSL run. Particularly, we define one event DSE for each message event kind introduced in Section 4.3. For example, the ECL Mapping Specification excerpt defines the DSEs msgSendEvt and msgReceiveEvt representing a message send and a message reception event, respectively.

Furthermore, the ECL Mapping Specification defines DSEs and invariants for the Modal stereotype «ModalMessage». We explained the DSEs representing the unification occurrences and their semantic constraints in Section 4.4.1.1. We define an invariant for the unification of



```
context ObjectSystemMessage
  ...
  def: msgSendEvt: Event
  def: msgReceiveEvt: Event
  ...
context ModalMessage
  def: msgCreateUnification: Event
  def: msgSendUnification: Event
  def: msgReceiveUnification: Event
  ...
  inv sendUnificationHot:
    self.isHot() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      Relation UnificationRelationHot (
        objectSystemMessage.msgSendEvt, self.msgSendUnification,
        self.msgCreateUnification)
    )
  inv receiveUnificationHot:
    self.isHot() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      Relation UnificationRelationHot (
        objectSystemMessage.msgReceiveEvt,
        self.msgReceiveUnification,  self.msgSendUnification)
    )
```

```
...
Clock enableBraking_msgSendEvt
Clock enableBraking_msgReceiveEvt
...
Clock EmcyBraking.enableBraking_msgCreateUnification
Clock EmcyBraking.enableBraking_msgSendUnification
Clock EmcyBraking.enableBraking_msgReceiveUnification
Relation EmcyBraking.enableBraking_sendUnificationHot
                  [UnificationRelationHot] (
   msgEvent->enableBraking_msgSendEvt,
   msgEventUnification->EmcyBraking.
              enableBraking_msgSendUnification,
   enableMsgEvtUnification->EmcyBraking.
              enableBraking_msgCreateUnification )
Relation EmcyBraking.enableBraking_receiveUnificationHot
                  [UnificationRelationHot] (
   msgEvent->enableBraking_msgReceiveEvt,
   msgEventUnification->EmcyBraking.
              enableBraking_msgReceiveUnification,
   enableMsgEvtUnification->EmcyBraking.
              enableBraking_msgSendUnification )
```

Figure 4.12: Specification excerpt of semantics for unification of hot messages including example models

each message event kind with the corresponding message location kind. For this purpose, the invariant relates each kind of object system message event DSE with the corresponding modal message unification occurrence DSEs.

For example, the «ModalMessage» invariant sendUnificationHot first determines the corresponding object system message in the case of a hot temperature of the context MSD message. Subsequently, it specifies the behavior of the unification of the send events of the object system message with the send unification location of the context MSD message with a reference to the user-defined MoCCML relation UnificationRelationHot. Its arguments are the send event of the object system message, the send unification occurrence of the context MSD message, and the message creation unification occurrence of the context MSD message. Analogously, the invariant receiveUnificationHot also uses the MoCCML relation UnificationRelationHot but with different arguments that are relevant for the unification of message receive events.

The parameters of the MoCCML relation UnificationRelationHot are msgEvt for the object system message event to be unified, msgEvtUnification for the occurrence of the event unification, and enableMsgEvtUnification for the occurrence of the preceding event unification that enables the parameter msgEvtUnification. In the initial state Not Enabled, ticks of the parameter clock msgEvt have no effect due to the self-transition. If the parameter clock enableMsgEvt-Unification ticks, the transition to the state Enabled is fired. In this state, the parameter clocks msgEvt and msgEvtUnification are allowed to tick simultaneously, which represents the actual event unification. In this case, the transition back to the state Not Enabled is fired.

Metamodel Level M1

The lower part of Figure 4.12 depicts excerpts of the Platform-specific MSD Specification and of the corresponding CCSL Model generated through the automatically derived MSD-to-CCSL QVT-O Transformation.

The MSD excerpt in the left lower part shows the MSD message `enableBraking` with the focus on its Message Send and Message Reception events.

The derived transformation translates any object system message and any MSD message of all MSDs to each six clock variables representing the particular event kinds and unification occurrences for the message location kinds. From these clock variables, the generated CCSL Model excerpt in the right lower part of Figure 4.12 depicts five ones. For example, the derived transformation translates the object system message `enableBraking` to the two clock variables enableBraking_msgSendEvt and enableBraking_msgReceiveEvt. These clock variables represent the message send and message reception event of the object system message, respectively (cf. the corresponding DSEs for the «ObjectSystemMessage» at M2). Furthermore, it translates the equally named MSD message to the corresponding three unification occurrence clock variables (cf. Section 4.4.1.1).

Moreover, the transformation generates for any MSD message of all MSDs and for any «ModalMessage» ECL invariant referencing the user-defined MoCCML relation each a corresponding CCSL model relation using the referenced relation. For example, the transformation translates the MSD message `enableBraking` of the MSD EmcyBraking to the CCSL model relation EmcyBraking.enableBraking_sendUnificationHot using the MoCCML UnificationRelationHot relation at metamodel level M2. This relation gets the arguments enableBraking_-msgSendEvt for the parameter msgEvent, EmcyBraking.enableBraking_msgSendUnification for msgEventUnification, and EmcyBraking.enableBraking_msgCreateUnification for enable-MsgEvtUnification. With this argument set, the MoCCML relation switches to the state Enabled when EmcyBraking.enableBraking_msgCreateUnification occurs, and the unification takes place when enableBraking_msgSendEvt and EmcyBraking.enableBraking_msgSendUnification tick simultaneously in this state. Furthermore, the translation translates the second depicted «ModalMessage» ECL invariant to the CCSL model relation EmcyBraking.enableBraking_receiveUnificationHot with an analogue set of arguments.

Metamodel Level M0

Figure 4.13 depicts an exemplary CCSL run resulting from the CCSL model excerpt depicted in the right lower part of Figure 4.12. This CCSL run simulates a situation in which object system message send/receive event clocks are synchronized with the send/receive unification occurrence clocks.

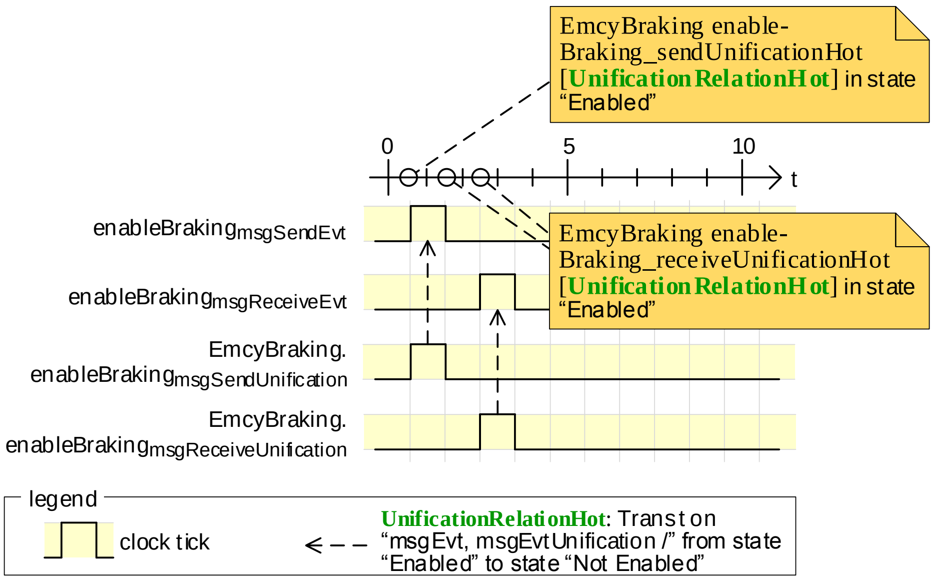

EmcyBraking enable-Braking_sendUnificationHot [UnificationRelationHot] in state "Enabled"
0
5
10
t
EmcyBraking enable-Braking_receiveUnificationHot [UnificationRelationHot] in state "Enabled"
enableBraking msgSendEvt
enableBraking msgReceiveEvt
EmcyBraking. enableBraking msgSendUnification
EmcyBraking. enableBraking msgReceiveUnification
legend
clock tick
UnificationRelationHot: Transition "msgEvt, msgEvtUnification /" from state "Enabled" to state "Not Enabled"

Figure 4.13: Example—message event unification in CCSL simulation for hot messages

The two topmost rows depict ticks of clocks representing the actual object message events. Here, we focus on the clocks that represent the message send event and the message reception event unifiable with the corresponding location types for the message `enableBraking`. We assume that the CCSL model relation enableBraking_sendUnificationHot is in the state Enabled at instant 0. If an event occurs in this state (i.e., the clock $\text{enableBraking}_{\text{msgSendEvent}}$ ticks at instant 1), we enforce the corresponding enabled unification occurrence clocks to tick simultaneously by means of the transition to the state Not Enabled. Thus, the clock $\text{EmcyBraking.enableBraking}_{\text{msgSendUnification}}$ also ticks at instant 1. The unification of the message reception event at instant 3 follows the same functional principle.

Figure B.1 in Appendix B.1 depicts an exemplary CCSL run where object system message send/receive event clocks are synchronized with multiple send/receive unification occurrence clocks representing the locations from MSD messages in different MSDs.

4.4.2 Encoding of Timing Effects Induced by Platform Properties

In this section, we present how we encode in our semantics the timing effects that are induced by the properties of the software execution platform. We support two general classes of timing behavior effects. The first class encompasses the different kinds of static delays between the particular event kinds discussed in Section 4.3. The second class encompasses delays that dynamically emerge from the blocking of resources when different software components have to access the same resource (e.g., the processor for the task execution, peripheral hardware, or an operating system service) at the same time. This blocking of resources is called mutual exclusion of resources.

Section 4.4.2.1 presents an example for the encoding of the static delay effects. Section 4.4.2.2 presents an example for the encoding of mutual resource exclusion, which induces dynamic delays.

4.4.2.1 Static Delays Between Message Event Kinds

As discussed in Section 4.3, message-based communication involves multiple events during the actual execution on a target platform, and static delays occur between each kind of such events. In this section, we present how we encode and compute these static delays. Whereas the most important input to response time analyses is the WCET (cf. Section 2.6), the best case times for the operation execution and the message transmission is also of high interest [BEGL05]. Thus, we present in this section abstract computations of intervals of both the minimum and maximum values of the delays. After we present the full TAM profile definition in Section 4.6.1.1, we refine these computations exploiting in detail each TAM platform property in Section 4.6.1.2.

In our semantics, we support the following four kinds of static delays inferred from Section 4.3 and based on the four components of real-time communication end-to-end delays defined by Tinkell et al. [TBW95] (cf. Figure 4.7):

Message Dispatch Delays occur between message creation and message send event (cf. Section 4.3.2). Message dispatch delays encompass the time to gain write access to a communication channel as well as the time to encode a message from its logical representation to a format suitable for the transfer via the communication channel. This encoding time depends on the overall message size (i.e., net size plus potential overheads) in relation to the encoding rate of the communication channel. Thus, we compute the message dispatch delay as

$$\begin{aligned}
&\left[msg{::}msgDispatchDelay_{min}, msg{::}msgDispatchDelay_{max}\right] \qquad (4.1)\\
&= \left[msg.connector.supplier{::}{<}dispatchOverhead{>}_{min}, msg.connector.supplier{::}{<}dispatchOverhead{>}_{max}\right]\\
&\quad + \left[\frac{msg.signature{::}{<}overallMsgSize{>}_{min}}{msg.connector.supplier{::}{<}overallEncodeRate{>}_{max}}, \frac{msg.signature{::}{<}overallMsgSize{>}_{max}}{msg.connector.supplier{::}{<}overallEncodeRate{>}_{min}}\right],
\end{aligned}$$

where $msg.connector$ is a **UML::Connector** associated by the MSD message msg,
$msg.connector.supplier$ is a **TamComConnection** or a **TamOSComChannel** that the connector is allocated to,
and $msg.signature$ is a **TamOperation** associated by the MSD message msg

Message Send Delays occur between message send and message reception event (cf. Section 4.3.1). Message send delays encompass the overall propagation latency (i.e., net latency plus potential overheads) of the applied communication channel as well as the time to transmit the message. This transmission time depends on the overall message size in relation to the overall throughput (i.e., gross throughput minus potential overhead deductions) of the communication channel. Thus, we compute the message send delay as

$$
\begin{aligned}
&\left[msg{::}msgSendDelay_{min}, msg{::}msgSendDelay_{max}\right] \qquad (4.2)\\
&= \left[msg.connector.supplier{::}{<}overallLatency{>}_{min}, msg.connector.supplier{::}{<}overallLatency{>}_{max}\right]\\
&\quad + \left[\frac{msg.signature{::}{<}overallMsgSize{>}_{min}}{msg.connector.supplier{::}{<}overallThroughput{>}_{max}}, \frac{msg.signature{::}{<}overallMsgSize{>}_{max}}{msg.connector.supplier{::}{<}overallThroughput{>}_{min}}\right],\\
&\quad \text{where } msg.connector \text{ is a } \textbf{UML::Connector} \text{ associated by the MSD message } msg,\\
&\quad msg.connector.supplier \text{ is a } \textbf{TamComConnection} \text{ or a } \textbf{TamOSComChannel} \text{ that the connector is allocated to,}\\
&\quad \text{and } msg.signature \text{ is a } \textbf{TamOperation} \text{ associated by the MSD message } msg
\end{aligned}
$$

Message Consumption Delays occur between message reception and message consumption event (cf. Section 4.3.2). Analogously to message dispatch delays, message consumption delays encompass the time to gain read access to a communication channel as well as the time to decode a message from the communication channel format to its logical representation. The decoding time depends on the overall message size in relation to the decoding rate of the communication channel. Thus, we compute the message consumption delay as

$$
\begin{aligned}
&\left[msg{::}msgConsumptionDelay_{min}, msg{::}msgConsumptionDelay_{max}\right] \qquad (4.3)\\
&= \left[msg.connector.supplier{::}{<}consumptionOverhead{>}_{min}, msg.connector.supplier{::}{<}consumptionOverhead{>}_{max}\right]\\
&\quad + \left[\frac{msg.signature{::}{<}overallMsgSize{>}_{min}}{msg.connector.supplier{::}{<}overallDecodeRate{>}_{max}}, \frac{msg.signature{::}{<}overallMsgSize{>}_{max}}{msg.connector.supplier{::}{<}overallDecodeRate{>}_{min}}\right],\\
&\quad \text{where } msg.connector \text{ is a } \textbf{UML::Connector} \text{ associated by the MSD message } msg,\\
&\quad msg.connector.supplier \text{ is a } \textbf{TamComConnection} \text{ or a } \textbf{TamOSComChannel} \text{ that the connector is allocated to,}\\
&\quad \text{and } msg.signature \text{ is a } \textbf{TamOperation} \text{ associated by the MSD message } msg.
\end{aligned}
$$

Task Execution Delays occur between task start and task completion event (cf. Section 4.3.3). Task execution delays encompass the normalized overall execution time (i.e., net execution time plus potential overheads) required to process a message in relation to the relative speed factor of the executing processing unit (cf. Section 4.1) as well as the times for accessing memory and resources. Thus, we compute the task execution delay as

$$
\begin{aligned}
&\left[msg{::}taskExecutionDelay_{min}, msg{::}taskExecutionDelay_{max}\right]\\
&= \left[\frac{msg.signature{::}{<}normalizedOverallExecTime{>}_{min}}{msg.connector[receiver].supplier.processingUnit{::}speedFactor_{max}},\right.\\
&\quad \left.\frac{msg.signature{::}{<}normalizedOverallExecTime{>}_{max}}{msg.connector[receiver].supplier.processingUnit{::}speedFactor_{min}}\right]\\
&\quad + \left[msg{::}{<}overallMemoryAccessTime{>}_{min}, msg{::}{<}overallMemoryAccessTime{>}_{max}\right]\\
&\quad + \left[msg{::}{<}overallResourceAccessTime{>}_{min}, msg{::}{<}overallResourceAccessTime{>}_{max}\right],\\
&\quad \text{where } msg.signature \text{ is a } \textbf{TamOperation} \text{ associated by the MSD message } msg,\\
&\quad msg.connector \text{ is a } \textbf{UML::Connector} \text{ associated by the MSD message } msg,\\
&\quad msg.connector[receiver] \text{ is a } \textbf{Modal::SpecificationPart} \text{ representing the receiving software component,}\\
&\quad msg.connector[receiver].supplier \text{ is a } \textbf{TamECU} \text{ that the receiving software component is allocated to,}\\
&\quad \text{and } msg.connector[receiver].supplier.processingUnit \text{ is the } \textbf{TamProcessingUnit} \text{ of the ECU.}
\end{aligned}
\qquad (4.4)
$$

We define a task to start immediately after it has consumed its corresponding message if the scheduler can dispatch it. If the scheduler cannot dispatch it immediately, a dynamic delay occurs (cf. Section 4.4.2.2). Furthermore, we define that after a component completed a task, it creates a potential subsequent message immediately afterward. Thus, we do not consider explicit static delays between such message consumption and task start events as well as between task completion and message creation events. Instead, we define the subsequent corresponding event to occur one tick after the occurrence of the corresponding preceding event in the case that no dynamic delay occurs (cf. Figure 4.7).

For the computation of message dispatch, send, and consumption delays, it is relevant whether the communicating components are allocated to different ECUs (i.e., the components are distributed) or to the same ECU. That is, other and typically more platform properties have to be considered for a distributed communication in contrast to an internal communication. Thus, we distinguish in the refined computations in Section 4.6.1.2 between distributed and internal message dispatch/send/consumption delays.

Figure 4.14 exemplarily depicts a specification excerpt of the semantics for task execution delays at metamodel level M2 (upper part of the figure) as well as corresponding example models at metamodel level M1 (lower part of the figure). Figure 4.15 depicts an exemplary corresponding CCSL run at metamodel level M0. We provide specification excerpts, example models, and example CCSL runs for the remaining static delays in Appendix B.1 (cf. Figures B.7 to B.14).

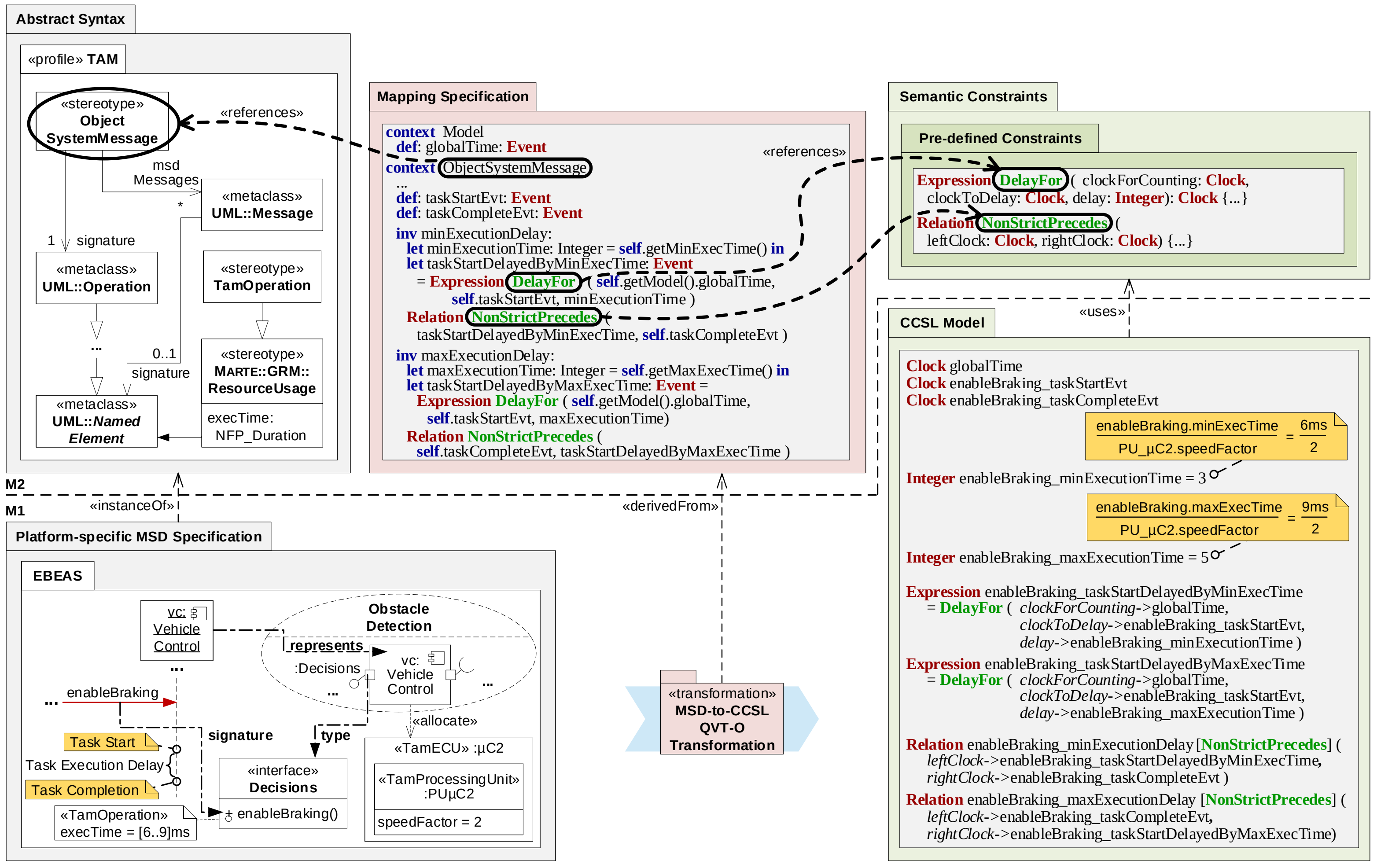
Metamodel Level M2

The upper part of Figure 4.14 depicts excerpts of the TAM profile, the ECL Mapping Specification, and of the applied semantic constraints.

In order to consider timing behavior, we have to keep track of the global time progress. For this purpose, we introduce a DSE globalTime for each Model in the ECL Mapping Specification in the middle upper part of Figure 4.14. We do not specify any semantic constraints for this DSE so that the corresponding CCSL clock always ticks. We utilize this clock as a reference clock for other clocks whose time differences we need to determine w.r.t. to globalTime, like the delay clocks that we introduce in the following.

The resolution in terms of the time unit of this discrete reference clock influences the accuracy as well as the performance of the timing analysis. That is, a too coarse-grained time unit may yield imprecise analysis results, and a too fine-grained time unit may produce many unnecessary steps of the CCSL run. We exemplarily apply milliseconds as the normalized time unit throughout this thesis.

Furthermore, the ECL Mapping Specification defines DSEs and invariants for the TAM stereotype «ObjectSystemMessage». The task processing of an object system message takes place in between occurrences of the «ObjectSystemMessage» DSEs taskStartEvt and taskCompleteEvt. The invariant minExecutionDelay defines the timing behavior taking minimum delays of task executions for the message processing into account. To this end, we first determine the minimum execution time of the object system message (cf. Equation (4.4)). We subsequently define a new event taskStartDelayedByMinExecTime by means of the pre-defined CCSL expression DelayFor that delays the taskStartEvt DSE by the minimum execution time. Finally, we specify that the taskCompleteEvt DSE can occur not earlier than this delayed event by means of the pre-defined CCSL relation NonStrictPrecedes. The invariant maxExecutionDelay defines the timing behavior taking maximum delays of task executions for the message processing into account and restricts the taskCompleteEvt DSE to occur not later than the maximum execution delay.



```
context  Model
  def: globalTime: Event
context ObjectSystemMessage
  ...
  def: taskStartEvt: Event
  def: taskCompleteEvt: Event
  inv minExecutionDelay:
    let minExecutionTime: Integer = self.getMinExecTime() in
    let taskStartDelayedByMinExecTime: Event
      = Expression DelayFor ( self.getModel().globalTime,
          self.taskStartEvt, minExecutionTime )
    Relation NonStrictPrecedes (
      taskStartDelayedByMinExecTime, self.taskCompleteEvt )
  inv maxExecutionDelay:
    let maxExecutionTime: Integer = self.getMaxExecTime() in
    let taskStartDelayedByMaxExecTime: Event =
      Expression DelayFor ( self.getModel().globalTime,
        self.taskStartEvt, maxExecutionTime)
    Relation NonStrictPrecedes (
      self.taskCompleteEvt, taskStartDelayedByMaxExecTime )
```

```
Expression DelayFor ( clockForCounting: Clock,
  clockToDelay: Clock, delay: Integer): Clock {...}
Relation NonStrictPrecedes (
  leftClock: Clock, rightClock: Clock) {...}
```

```
Clock globalTime
Clock enableBraking_taskStartEvt
Clock enableBraking_taskCompleteEvt
                        enableBraking.minExecTime  =  6ms
                        PU_µC2.speedFactor            2
Integer enableBraking_minExecutionTime = 3
                        enableBraking.maxExecTime  =  9ms
                        PU_µC2.speedFactor            2
Integer enableBraking_maxExecutionTime = 5

Expression enableBraking_taskStartDelayedByMinExecTime
    = DelayFor (  clockForCounting->globalTime,
                  clockToDelay->enableBraking_taskStartEvt,
                  delay->enableBraking_minExecutionTime )
Expression enableBraking_taskStartDelayedByMaxExecTime
    = DelayFor (  clockForCounting->globalTime,
                  clockToDelay->enableBraking_taskStartEvt,
                  delay->enableBraking_maxExecutionTime )

Relation enableBraking_minExecutionDelay [NonStrictPrecedes] (
    leftClock->enableBraking_taskStartDelayedByMinExecTime,
    rightClock->enableBraking_taskCompleteEvt )
Relation enableBraking_maxExecutionDelay [NonStrictPrecedes] (
    leftClock->enableBraking_taskCompleteEvt,
    rightClock->enableBraking_taskStartDelayedByMaxExecTime)
```

Figure 4.14: Specification excerpt of semantics for task execution delays including example models

Metamodel Level M1

The lower part of Figure 4.14 depicts excerpts of the Platform-specific MSD Specification and of the corresponding CCSL Model generated through the automatically derived MSD-to-CCSL QVT-O Transformation.

The MSD excerpt in the left lower part shows the MSD message `enableBraking` with the focus on its Task Start and Task Completion events. The MSD message has a referential trace link to the equally named «TamOperation» with a minimum execution time of 6ms and a maximum execution time of 9ms. The lifeline vc: VehicleControl represents the equally named component role that is allocated to the «TamECU» :µC2. This ECU contains a «TamProcessing-Unit» with the speed factor 2.

As indicated by the generated CCSL Model excerpt in the right lower part of Figure 4.14, the derived transformation generates for any MSD specification each the clock variable globalTime representing the always ticking reference clock. Furthermore, the transformation translates any object system message to each six clock variables in the generated CCSL Model that represent the particular event kinds (cf. Section 4.4.1.2). From these clock variables, the generated CCSL Model excerpt depicts the task start and task completion event clocks. Moreover, the transformation generates for any object system message each an Integer variable specifying the minimum and maximum task execution delay values. As defined by Equation (4.4), these delay values are determined by the minimum and maximum execution times of the associated «TamOperation» in relation to the relative speed factor of the processing unit, inter alia. For example, the minimum and maximum task execution delay values of the object system message are stored in Integer variables with the values $\frac{trajectoryBeacon::minExecTime}{PU\mu C2::speedFactor} = \frac{6ms}{2} = 3\text{ms}$ and $\frac{trajectoryBeacon::maxExecTime}{PU\mu C2::speedFactor} = \frac{9ms}{2} \approx 5\text{ms}$, respectively.

Furthermore, the transformation generates for any object system message each two clock variables delaying the task start clock by the minimum and maximum execution time, respectively. For example, the object system message `enableBraking` is translated to two CCSL expressions DelayFor defining the new clock variables enableBraking_taskStartDelayedByMin-ExecTime and enableBraking_taskStartDelayedByMaxExecTime, respectively. These expressions delay the task start event clock enableBraking_taskStartEvt by the corresponding minimum and maximum task execution delays w.r.t. the globalTime clock.

Finally, the transformation restricts the task completion event clock to tick not earlier than the minimum task execution delay clock and not later than the maximum task execution delay clock. For example, the transformation generates the CCSL model relations enableBraking_minExecu-tionDelay and enableBraking_maxExecutionDelay using the CCSL relation NonStrictPrecedes at metamodel level M2. Both relations enforce the clock enableBraking_taskCompleteEvt to tick in between the ticks of the clocks enableBraking_taskStartDelayedByMinExecTime and enableBraking_taskStartDelayedByMaxExecTime.

Metamodel Level M0

Figure 4.15 depicts an exemplary CCSL run resulting from the CCSL model excerpt depicted in the right lower part of Figure 4.14. This CCSL run simulates the task execution delay of the `enableBraking` message w.r.t. to the minimum and maximum execution time of its «TamOperation».

The topmost row depicts the ticks of the reference clock globalTime that always ticks. The row below depicts the tick of the clock $\text{enableBraking}_{\text{taskStartEvt}}$ at instant 1. This clock tick is delayed by 3 and 5 instants with the amount of ticks of the clocks

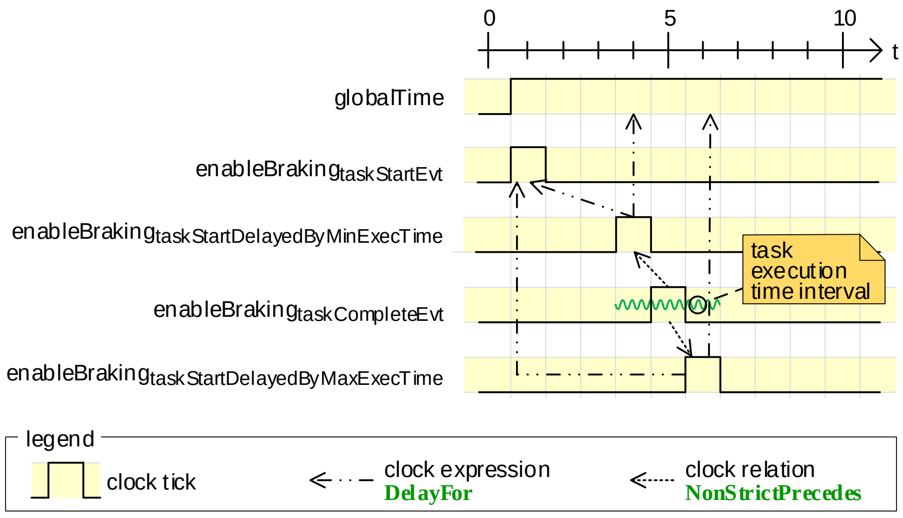

0
5
10
t
globalTime
enableBraking_taskStartEvt
enableBraking_taskStartDelayedByMinExecTime
enableBraking_taskCompleteEvt
enableBraking_taskStartDelayedByMaxExecTime
task execution time interval
legend
clock tick
clock expression DelayFor
clock relation NonStrictPrecedes

Figure 4.15: Exemplary CCSL run simulating delays due to task execution

enableBraking$_{\text{taskStartDelayedByMinExecTime}}$ and enableBraking$_{\text{taskStartDelayedByMaxExecTime}}$, respectively. The two NonStrictPrecedes clock relations enforce the clock enableBraking$_{\text{taskCompleteEvt}}$ to tick at some instant between 4 and 6, and it ticks at instant 5 in the example.

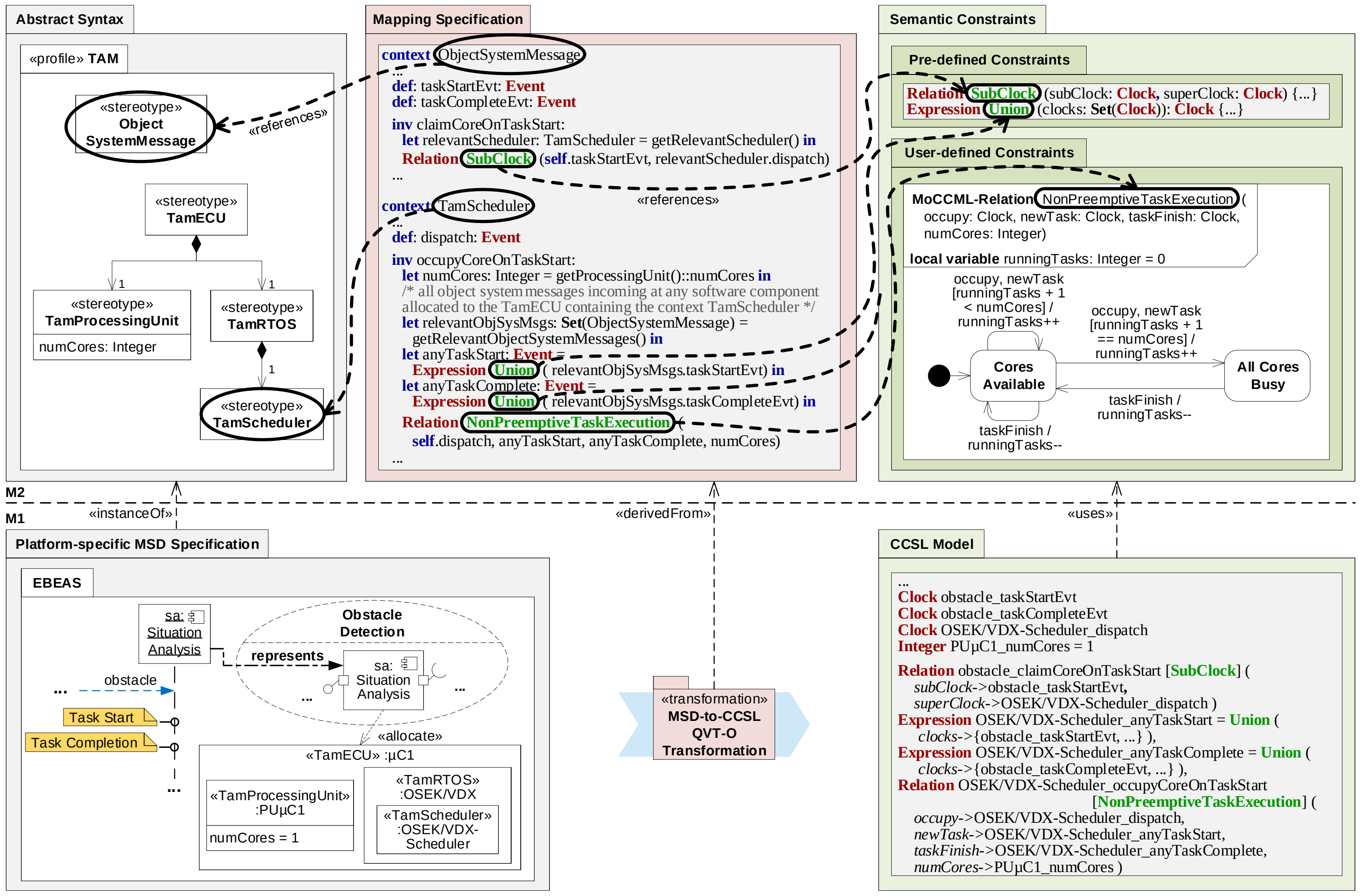
4.4.2.2 Dynamic Delays due to Mutual Exclusion of Resources

Target execution platforms of software-intensive systems have restricted resources, which cannot be completely used by different parts of the application software at the same time. For example, at one instant only a certain application software part can be processed by a core of the processing unit the application software is deployed to, and at one instant a communication medium channel can only transfer a certain amount of message bits. Middleware and operating system services take care of the management of these restricted resources for the competing software parts. That is, such services provide mechanisms ensuring that the resources are used by the software in a mutually exclusive manner. For example, schedulers select according to a scheduling algorithm at certain instants a task for the execution of an application software operation on a processing unit. Analogously, the communication services of the middleware or of the operating system manage the message scheduling for the communication media. In our semantics, we support delays that dynamically emerge from the mutual exclusion of processing units, communication media, peripherals, and operating system resources.

Figure 4.16 depicts a specification excerpt of the semantics for the scheduling of tasks on processing units at metamodel level M2 (upper part of the figure) as well as corresponding example models at metamodel level M1 (lower part of the figure). Figure 4.17 depicts an exemplary corresponding CCSL run at metamodel level M0. Furthermore, we provide specification excerpts, example models, and example CCSL runs for the message scheduling for communication media as well as shared operating system resources in Appendix B.1 (cf. Figures B.15 to B.18).

Metamodel Level M2

The upper part of Figure 4.16 depicts excerpts of the TAM profile, the ECL Mapping Specification, and of the applied semantic constraints.



```
context ObjectSystemMessage
  ...
  def: taskStartEvt: Event
  def: taskCompleteEvt: Event
  inv claimCoreOnTaskStart:
    let relevantScheduler: TamScheduler = getRelevantScheduler() in
    Relation SubClock (self.taskStartEvt, relevantScheduler.dispatch)
  ...
context TamScheduler
  ...
  def: dispatch: Event
  inv occupyCoreOnTaskStart:
    let numCores: Integer = getProcessingUnit()::numCores in
    /* all object system messages incoming at any software component
    allocated to the TamECU containing the context TamScheduler */
    let relevantObjSysMsgs: Set(ObjectSystemMessage) =
      getRelevantObjectSystemMessages() in
    let anyTaskStart: Event =
      Expression Union ( relevantObjSysMsgs.taskStartEvt) in
    let anyTaskComplete: Event =
      Expression Union ( relevantObjSysMsgs.taskCompleteEvt) in
    Relation NonPreemptiveTaskExecution (
      self.dispatch, anyTaskStart, anyTaskComplete, numCores)
  ...
```

```
Relation SubClock (subClock: Clock, superClock: Clock) {...}
Expression Union (clocks: Set(Clock)): Clock {...}
```

```
MoCCML-Relation NonPreemptiveTaskExecution (
  occupy: Clock, newTask: Clock, taskFinish: Clock,
  numCores: Integer)
local variable runningTasks: Integer = 0
```

```
...
Clock obstacle_taskStartEvt
Clock obstacle_taskCompleteEvt
Clock OSEK/VDX-Scheduler_dispatch
Integer PUµC1_numCores = 1
Relation obstacle_claimCoreOnTaskStart [SubClock] (
   subClock->obstacle_taskStartEvt,
   superClock->OSEK/VDX-Scheduler_dispatch )
Expression OSEK/VDX-Scheduler_anyTaskStart = Union (
    clocks->{obstacle_taskStartEvt, ...} ),
Expression OSEK/VDX-Scheduler_anyTaskComplete = Union (
    clocks->{obstacle_taskCompleteEvt, ...} ),
Relation OSEK/VDX-Scheduler_occupyCoreOnTaskStart
                              [NonPreemptiveTaskExecution] (
   occupy->OSEK/VDX-Scheduler_dispatch,
   newTask->OSEK/VDX-Scheduler_anyTaskStart,
   taskFinish->OSEK/VDX-Scheduler_anyTaskComplete,
   numCores->PUµC1_numCores )
```

Figure 4.16: Specification excerpt of semantics for task scheduling including example models

The ECL Mapping Specification in the middle upper part of Figure 4.16 defines DSEs and invariants for the TAM stereotypes «ObjectSystemMessage» and «TamScheduler». The execution of a task that processes an object system message takes place in between the «ObjectSystemMessage» DSEs taskStartEvt and taskCompleteEvt. We define a DSE dispatch for the «TamScheduler», which represents the point in time when the scheduler selects a task for the execution on a processing unit (i.e., the scheduler *dispatches* the task). We express the relation between the «ObjectSystemMessage» DSE taskStartEvt and the «TamScheduler» DSE dispatch by means of the «ObjectSystemMessage» invariant claimCoreOnTaskStart. This invariant uses the pre-defined CCSL relation SubClock to associate the taskStartEvt DSE with the dispatch DSE. The SubClock relation prevents the subClock argument clock taskStartEvt from ticking in the case that the superClock argument clock dispatch cannot tick. We determine whether the superClock argument clock dispatch can tick through the «TamScheduler» invariant occupyCoreOnTaskStart, which we explain below. By doing so, we prevent that a scheduler dispatches a task that processes an object system message when the corresponding processing unit is busy with the execution of other tasks.

We determine whether the scheduler is able to dispatch a task on the processing unit through the «TamScheduler» invariant occupyCoreOnTaskStart. First, we determine the amount of cores of the corresponding processing unit. Subsequently, we determine all object system messages incoming at any software component allocated to the «TamECU» that contains the context «TamScheduler». From these object system messages, we define through the pre-defined CCSL expression Union the DSEs anyTaskStart and anyTaskComplete that represent the union of all taskStartEvt and taskCompleteEvt DSEs, respectively. By doing so, we determine whether any task is ready to be started or any task gets completed.

The actual behavior of the «TamScheduler» invariant occupyCoreOnTaskStart is specified by means of the user-defined MoCCML relation NonPreemptiveTaskExecution. Its arguments are the dispatch DSE for the parameter clock occupy, the clocks representing the union of all relevant taskStartEvt and taskCompleteEvt DSEs for the newTask and the taskFinish clock, respectively, and the amount of cores of the corresponding processing unit for the Integer parameter numCores. Furthermore, we define a local variable runningTasks that captures the amount of tasks currently running on the processing unit. The initial state of the MoCCML relation is Cores Available, which defines that the scheduler is able to dispatch new tasks because the processing unit is not busy with processing other tasks. When a new task is ready to be dispatched in this state and hence the occupy parameter clock as well as the newTask parameter clock tick simultaneously, the variable runningTasks is incremented if the guard [runningTasks + 1 $<$ numCores] holds. Analogously, when any task running on the processing unit is finished in this state and hence the parameter clock taskFinish ticks, the variable runningTasks is decremented. If the amount of currently running tasks equals the amount of cores in this state, the transition to the state All Cores Busy is fired. In this state, the dispatching of new tasks is not allowed. When any task running on the processing unit gets completed in this state and hence the parameter clock taskFinish ticks, the variable runningTasks is decremented and the transition to the initial state is fired.

Our semantics support multiple software components allocated to one processing unit, multiple cores per processing unit, and different task priorities. However, it currently only supports non-preemptive scheduling, where tasks cannot be preempted by other tasks and hence run to completion. Preemptive scheduling is more dynamic and can be specified in future work through the GEMOC concept of domain-specific actions [LCD+15; DCB+15]. We specify the actual scheduling policy, which the scheduler applies to select and dispatch tasks, through the object system message invariants nonPreemptiveFixedPriorityPolicy_noInterferenceWithHigherPrioTasks and nonPreemptiveFixedPriorityPolicy_syncWithLowerPrioTasks in Listing B.2 in Appendix B.2. Currently, our semantics only supports the policy fixed-priority scheduling, which is the predominant scheduling policy used for real-time systems [NMH08; DB08; SAÅ+04] due to enabling dynamic as well as a predictable behavior [ABD+95]. Further scheduling policies can be flexibly added through adding the corresponding invariants.

Metamodel Level M1

The lower part of Figure 4.16 depicts excerpts of the Platform-specific MSD Specification and of the corresponding CCSL Model generated through the automatically derived MSD-to-CCSL QVT-O Transformation.

The MSD excerpt in the left lower part shows the MSD message `obstacle` with the focus on its Task Start and Task Completion events. The lifeline has a referential trace link to the component role sa: SituationAnalysis, which is allocated to the «TamECU» :µC1. This element encompasses a «TamProcessingUnit» :PUµC1 and a «TamRTOS» with a «TamScheduler» :OSEK/VDX-Scheduler-µC1.

The derived transformation translates any object system message to each six clock variables in the generated CCSL Model that represent the particular event kinds (cf. Section 4.4.1.2). From these clock variables, the generated CCSL Model excerpt in the right lower part of Figure 4.16 depicts the task start and task completion event clocks. Furthermore, the transformation generates for any «TamScheduler» each a scheduler dispatch clock variable. For example, the transformation generates the clock variable OSEK/VDX-Scheduler_dispatch for the :OSEK/VDX-Scheduler. Moreover, the transformation translates the tagged value numCores of any «TamProcessingUnit» to each a corresponding Integer variable. For example, the tagged value numCores of the processing unit :PUµC1 is translated to the Integer variable PUµC1_numCores.

Furthermore, the transformation translates any object system message to a SubClock clock relation that restricts the object system message task start event clock to tick only when the scheduler dispatch clock can tick simultaneously. For example, the object system message `obstacle` is translated to the CCSL model relation obstacle_claimCoreOnTaskStart using the SubClock relation at metamodel level M2. This relation gets the arguments obstacle_taskStart and OSEK/VDX-Scheduler_dispatch for the parameters subClock and superClock, respectively.

For any «TamScheduler», the transformation generates each two clock variables defined by Union clock expressions that determine the union of potential ticks of all task start and task completion event clocks. For example, the scheduler :OSEK/VDX-Scheduler is translated to the clock variables OSEK/VDX-Scheduler_anyTaskStart and OSEK/VDX-Scheduler_anyTaskComplete. These expressions get the clock variables obstacle_taskStartEvt and obstacle_taskCompleteEvt as a set argument, along with other task start and task completion events.

Finally, the transformation generates for any «TamScheduler» a CCSL model relation that uses the user-defined MoCCML relation NonPreemptiveTaskExecution at metamodel level M2. For example, the transformation translates the scheduler :OSEK/VDX-Scheduler to the CCSL model relation OSEK/VDX-Scheduler_occupyCoreOnTaskStart. This relation gets the argument OSEK/VDX-Scheduler_dispatch for the parameter occupy, OSEK/VDX-Scheduler_anyTaskStart for newTask, OSEK/VDX-Scheduler_anyTaskComplete for taskFinish, and PUµC1_numCores for numCores.

Metamodel Level M0

Figure 4.17 depicts an exemplary CCSL run resulting from the CCSL model excerpt depicted in the right lower part of Figure 4.16. This CCSL run simulates a situation in which two different computations have to be performed concurrently by the software component sa: SituationAnalysis allocated to the «TamECU» :µC1 (cf. MSD specification excerpt in left lower part of Figure 4.16).

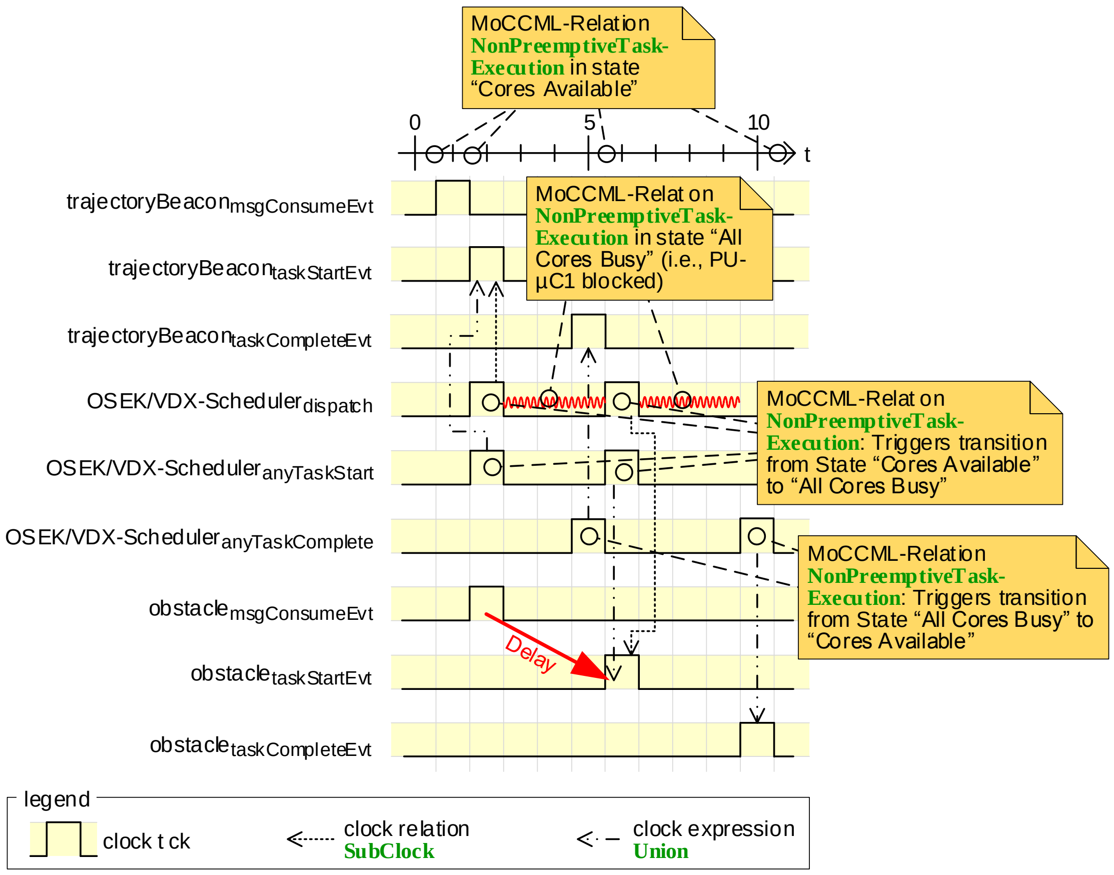

MoCCML-Relation NonPreemptiveTask-Execution in state "Cores Available"
0
5
10
t
trajectoryBeacon msgConsumeEvt
trajectoryBeacon taskStartEvt
trajectoryBeacon taskCompleteEvt
OSEK/VDX-Scheduler dispatch
OSEK/VDX-Scheduler anyTaskStart
OSEK/VDX-Scheduler anyTaskComplete
obstacle msgConsumeEvt
obstacle taskStartEvt
obstacle taskCompleteEvt
MoCCML-Relation NonPreemptiveTask-Execution in state "All Cores Busy" (i.e., PU-µC1 blocked)
MoCCML-Relation NonPreemptiveTask-Execution: Triggers transition from State "Cores Available" to "All Cores Busy"
MoCCML-Relation NonPreemptiveTask-Execution: Triggers transition from State "All Cores Busy" to "Cores Available"
Delay
legend
clock tick
clock relation SubClock
clock expression Union

Figure 4.17: Exemplary CCSL run simulating task scheduling

The MSD BeaconAcknowledgement in Figure 4.2(a) specifies that sa: SituationAnalysis is responsible for processing information about the trajectories of other vehicles via a message `trajectoryBeacon`. The three topmost rows in Figure 4.17 depict the ticks of the clocks representing the corresponding message consume, task start, and task completion events. The processing unit of :µC1 is not busy with other computations at instant 1 when the message is consumed. That is, the MoCCML relation TaskExecution used by the CCSL model relation OSEK/VDX-Scheduler_occupyCoreOnTaskStart is in the state Cores Available (cf. Figure 4.16). Hence, the clock OSEK/VDX-Scheduler$_{\text{dispatch}}$ is allowed to tick, meaning that the scheduler of :µC1 is able to dispatch the corresponding task. Since this clock is able to tick, the clock relation SubClock allows the clock trajectoryBeacon$_{\text{taskStartEvent}}$ to tick simultaneously at instant 2. Consequently, the clock OSEK/VDX-Scheduler$_{\text{anyTaskStart}}$ ticks at this instant as defined by the clock expression Union. Analogously, the clock OSEK/VDX-Scheduler$_{\text{anyTaskComplete}}$ ticks at instant 5 due to the tick of trajectoryBeacon$_{\text{taskStartComplete}}$.

The message `obstacle` is consumed at instant 2, resulting in the tick of the clock obstacle$_{\text{msgConsumeEvt}}$ at this instant. Due to the fact that :PUμC1 has only one core and due to the dispatching of the `trajectoryBeacon` processing task, the MoCCML relation NonPreemptiveTaskExecution is in the state All Cores Busy from instant 3 to instant 5. Thus, :PUμC1 is blocked in this time period so that the task processing of the `obstacle` message can be dispatched at instant 6 at the earliest, causing the clock obstacle$_{\text{taskStartEvt}}$ to tick at this instant. This causes a dynamic delay of 3 time units between the consumption and the actual processing of the message `obstacle`.

4.4.3 Encoding of Real-time Requirements and Timing Analysis Contexts

In this section, we present the encoding of aspects in our semantics that are crucial for the timing analysis results and the timing analysis setup. In Section 4.4.3.1, we explain how we encode MSD clock resets and hot time conditions in terms of real-time requirements in CCSL, which the TIMESQUARE timing analysis determines as fulfilled or violated by the timing behavior of the system. In Section 4.4.3.2, we explain how we encode analysis contexts defined by the Timing Analysts to investigate the simulation scenarios that they are interested in.

4.4.3.1 Clock Resets and Time Conditions

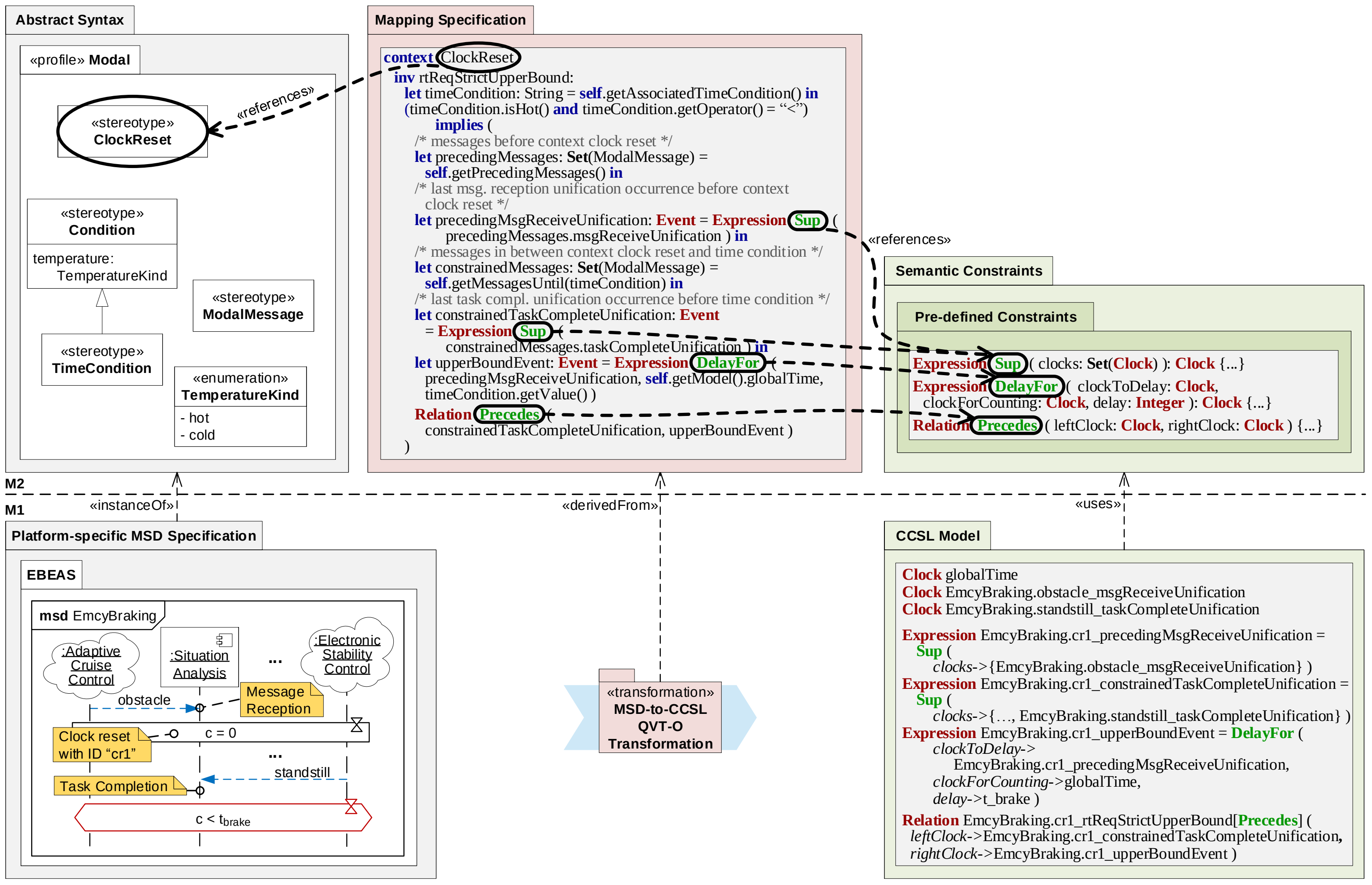
The combination of a clock reset and a hot time condition in an MSD represents a real-time requirement. A violation of such real-time requirements can lead to hazards in the case of safety-critical systems. Our timing analysis approach intends to consider a detailed timing behavior of the system incorporating a target execution platform model in order to reveal potential real-time requirement violations in an early development phase. In this section, we present how we encode combinations of clock resets and hot time conditions in our semantics.

Figure 4.18 depicts a specification excerpt of the semantics for clock resets combined with maximal delays at metamodel level M2 (upper part of the figure) as well as corresponding example models at metamodel level M1 (lower part of the figure). Figure 4.19 depicts an exemplary corresponding CCSL at metamodel level M0. We exemplarily focus on the semantics for maximal delays with a strict upper bound in this section. Furthermore, we provide the complete ECL pseudocode specification for non-strict upper bound maximal delays as well as minimal delays (both strict and non-strict lower bounds) in Listing B.7 in Appendix B.2.

Metamodel Level M2

The upper part of Figure 4.18 depicts excerpts of the Modal profile, the ECL Mapping Specification, and of the applied semantic constraints.

The ECL Mapping Specification in the middle upper part of Figure 4.9 the invariant rtReqStrictUpperBound for the Modal stereotype «ClockReset». For strict upper bound maximal delays, we define that a real-time requirement constrains the time between the message reception event occurrence prior to a clock reset and the task completion event occurrence prior to the maximal delay. Thus, the invariant constrains the task completion unification of the message prior to a hot time condition to occur before the upper bound value of the time condition w.r.t. the message reception unification of the message prior to a clock reset. To this end, the invariant initially determines the time condition following the clock reset. If the time condition is hot and its operator defines a strict upper bound (i.e., the operator equals "<"), the implication becomes true and hence the invariant is relevant to the context clock reset.



```
context ClockReset
  inv rtReqStrictUpperBound:
    let timeCondition: String = self.getAssociatedTimeCondition() in
    (timeCondition.isHot() and timeCondition.getOperator() = "<")
          implies (
      /* messages before context clock reset */
      let precedingMessages: Set(ModalMessage) =
        self.getPrecedingMessages() in
      /* last msg. reception unification occurrence before context
        clock reset */
      let precedingMsgReceiveUnification: Event = Expression Sup (
            precedingMessages.msgReceiveUnification ) in
      /* messages in between context clock reset and time condition */
      let constrainedMessages: Set(ModalMessage) =
        self.getMessagesUntil(timeCondition) in
      /* last task compl. unification occurrence before time condition */
      let constrainedTaskCompleteUnification: Event
        = Expression Sup (
            constrainedMessages.taskCompleteUnification ) in
      let upperBoundEvent: Event = Expression DelayFor (
        precedingMsgReceiveUnification, self.getModel().globalTime,
        timeCondition.getValue() )
      Relation Precedes (
        constrainedTaskCompleteUnification, upperBoundEvent )
    )
```

```
Expression Sup( clocks: Set(Clock) ): Clock {...}
Expression DelayFor( clockToDelay: Clock,
  clockForCounting: Clock, delay: Integer ): Clock {...}
Relation Precedes ( leftClock: Clock, rightClock: Clock ) {...}
```

```
Clock globalTime
Clock EmcyBraking.obstacle_msgReceiveUnification
Clock EmcyBraking.standstill_taskCompleteUnification

Expression EmcyBraking.cr1_precedingMsgReceiveUnification =
  Sup (
      clocks->{EmcyBraking.obstacle_msgReceiveUnification} )
Expression EmcyBraking.cr1_constrainedTaskCompleteUnification =
  Sup (
      clocks->{..., EmcyBraking.standstill_taskCompleteUnification} )
Expression EmcyBraking.cr1_upperBoundEvent = DelayFor (
      clockToDelay->
          EmcyBraking.cr1_precedingMsgReceiveUnification,
      clockForCounting->globalTime,
      delay->t_brake )
Relation EmcyBraking.cr1_rtReqStrictUpperBound[Precedes] (
  leftClock->EmcyBraking.cr1_constrainedTaskCompleteUnification,
  rightClock->EmcyBraking.cr1_upperBoundEvent )
```

Figure 4.18: Specification excerpt of semantics for clock resets and hot time conditions including example models

If the invariant is relevant, we determine all message reception unification DSEs of all MSD messages prior to the context clock reset. From these, we determine the last message reception unification DSE precedingMsgReceiveUnification directly prior to the context clock reset by means of the pre-defined CCSL clock expression Sup (cf. Section 2.7.2.1). This expression defines a new clock that is the fastest among all clocks in a given parameter clock set. Subsequently, we determine the last task completion unification DSE constrainedTaskCompleteUnification of all MSD messages prior to the associated time condition in an analogous manner. We define a new clock upperBoundEvent representing the upper bound of the time condition through the clock expression DelayFor, which delays precedingMsgReceiveUnification by the upper bound value. Finally, we constrain the ticks of constrainedTaskCompleteUnification to occur before the ticks of upperBoundEvent by means of the clock expression Precedes.

Metamodel Level M1

The lower part of Figure 4.18 exemplarily depicts excerpts of the Platform-specific MSD Specification and of the corresponding CCSL Model generated through the automatically derived MSD-to-CCSL QVT-O Transformation.

The left lower part of the figure shows an excerpt of the MSD EmcyBraking encompassing the MSD message `obstacle` prior to the clock reset with the identifier cr1 as well as the MSD message `standstill` prior to the hot time condition $\texttt{c < t}_{\texttt{brake}}$. The focus is on the message reception location for `obstacle` and on the task completion location for standstill.

The derived transformation generates the CCSL Model as indicated by the excerpt in the right lower part of Figure 4.18. As explained in Section 4.4.2.1, the derived transformation generates a generic clock variable globalTime that keeps track of the overall time progress. Furthermore, it translates any MSD message to each a clock variable representing the occurrence of message reception as well as task completion unifications, inter alia (cf. Section 4.4.1.1). For example, a clock variable EmcyBraking.obstacle_msgReceiveUnification is generated for the MSD message `obstacle` of the MSD EmcyBraking, and a clock variable EmcyBraking.standstill_-taskCompleteUnification is generated for the message `standstill`.

Furthermore, the transformation generates for any clock reset of all MSDs three CCSL model clock expressions using the pre-defined clock expressions referenced at metamodel level M2. That is first, a new clock is defined that represents the occurrence of the message reception unification of the last MSD message prior to the clock reset. In terms of the example model, the clock EmcyBraking.obstacle_msgReceiveUnification serves as argument for the parameter set clocks of the clock expression Sup. Since this is the only clock in the parameter set due to the fact that the MSD message `obstacle` is the only message prior to the clock reset with the identifier cr1, this clock is newly defined as EmcyBraking.cr1_precedingMsgReceiveUnification. Second, a new clock is defined that represents the occurrence of the task completion unification of the last MSD message prior to the time condition. For example, the clock EmcyBraking.standstill_-taskCompleteUnification together with other task completion unification clocks serve as arguments for the parameter set clocks of the Sup expression. From the parameter set, this clock is chosen because it is the fastest among the argument clock set and hence the last task completion unification prior to the time condition. The clock is newly defined in the variable EmcyBraking.cr1_constrainedTaskCompleteUnification. Third, the transformation generates a clock expression using DelayFor that delays the message reception unification clock stemming from the last MSD message prior to the clock reset by the value of the time condition. For example, the expression defines a new clock EmcyBraking.cr1_upperBoundEvent that is delayed by t_brake time units w.r.t. the clock EmcyBraking.cr1_precedingMsgReceiveUnification.

Finally, the transformation generates for each clock reset a CCSL model clock relation using the pre-defined clock relation Precedes referenced at metamodel level M2. This relation enforces the task completion unification clock stemming from the last MSD message prior to the time condition to tick before the delayed clock representing the upper bound value of the time condition. For example, the relation EmcyBraking.cr1_rtReqStrictUpperBound applies the clock EmcyBraking.standstill_constrainedTaskCompleteUnification as argument for leftClock and the clock EmcyBraking.cr1_upperBoundEvent as argument for rightClock.

Metamodel Level M0

Figure 4.19 depicts an exemplary CCSL run resulting from the CCSL model excerpt depicted in the right lower part of Figure 4.18. This CCSL run simulates an situation, where the time condition value placeholder $\texttt{t}_{\texttt{brake}}$ has each two exemplary concrete values, leading one time to the fulfillment and another time to the violation of the real-time requirement.

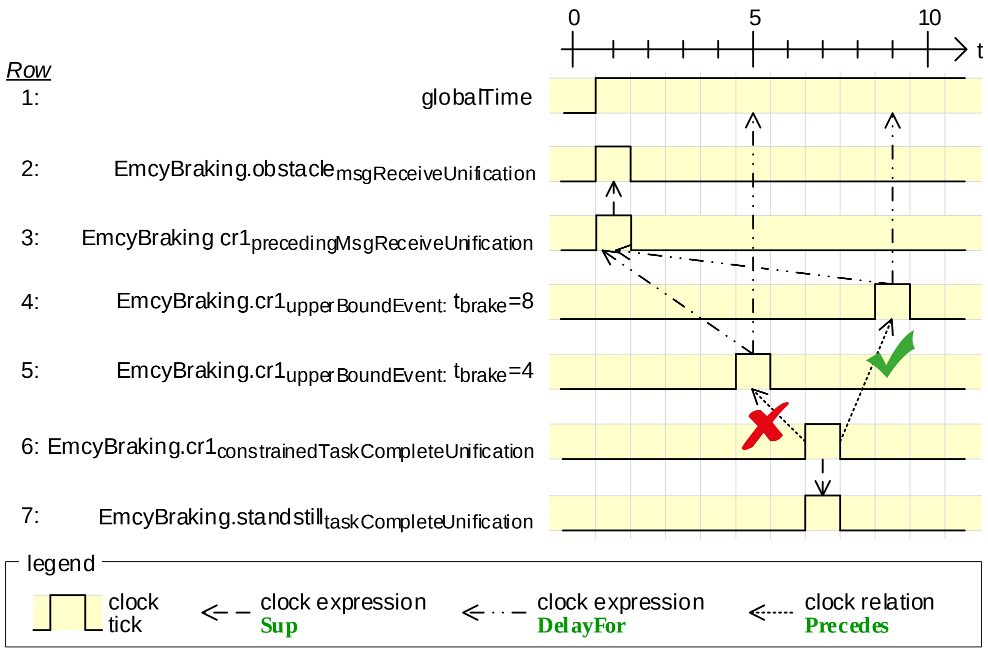

0
5
10
t
Row
1: globalTime
2: EmcyBraking.obstacle$_{\text{msgReceiveUnification}}$
3: EmcyBraking cr1$_{\text{precedingMsgReceiveUnification}}$
4: EmcyBraking.cr1$_{\text{upperBoundEvent}}$: t_{brake}=8
5: EmcyBraking.cr1$_{\text{upperBoundEvent}}$: t_{brake}=4
6: EmcyBraking.cr1$_{\text{constrainedTaskCompleteUnification}}$
7: EmcyBraking.standstill$_{\text{taskCompleteUnification}}$
legend
clock tick
clock expression Sup
clock expression DelayFor
clock relation Precedes

Figure 4.19: Exemplary CCSL run simulating a real-time requirement fulfillment and violation

The ticks of the clocks EmcyBraking.obstacle$_{\text{msgReceiveUnification}}$ (second topmost row) and EmcyBraking.standstill$_{\text{taskCompleteUnification}}$ (row 7) represent the occurrences of the message reception and task completion unification of the MSD messages `obstacle` and `standstill`, respectively. Based on both clocks, the clock expression Sup defines each a new clock EmcyBraking.cr1$_{\text{precedingMsgReceiveUnification}}$ (row 3) and EmcyBraking.cr1$_{\text{constrainedTaskCompleteUnification}}$ (row 6). Furthermore, the clock expression DelayFor delays the clock EmcyBraking.cr1$_{\text{precedingMsgReceiveUnification}}$ by $\texttt{t}_{\texttt{brake}}$ time units, defining the new clock EmcyBraking.cr1$_{\text{upperBoundEvent}}$.

This clock is depicted in row 4 and 5 with each two exemplary concrete values for $\texttt{t}_{\texttt{brake}}$. The clock relation Precedes enforces the tick of EmcyBraking.cr1$_{\text{constrainedTaskCompleteUnification}}$ to occur before the tick of EmcyBraking.cr1$_{\text{upperBoundEvent}}$. In the example situation, EmcyBraking.cr1$_{\text{constrainedTaskCompleteUnification}}$ ticks at the instant 7. This clock tick fulfills the Precedes relation if the value $\texttt{t}_{\texttt{brake}}$ is 8 so that EmcyBraking.cr1$_{\text{upperBoundEvent}}$ ticks at the instant 9 (row 4). This means that the software execution on the specified platform fulfills the real-time requirement for the given analysis context. However, if the value $\texttt{t}_{\texttt{brake}}$ is 4 (row 5), the tick of EmcyBraking.cr1$_{\text{constrainedTaskCompleteUnification}}$ cannot fulfill the Precedes relation because the real-time requirement is too tight. That is, the BDD solver of TIMESQUARE cannot solve the underlying Boolean expression, and the simulation stops with a deadlock (cf. Section 2.7.1). This situation represents a real-time requirement violation.

4.4.3.2 Timing Analysis Contexts

In order to conduct a particular timing analysis, the Timing Analysts have to specify the concrete simulation scenario that they want to investigate (cf. process step Define Simulation Scenarios in the lower left of Figure 4.6). Such an analysis scenario is known as *analysis context* [SG14; OMG11]. For the specification of such timing analysis contexts, we provide the TAM subprofile AnalysisContext (cf. Section 4.1). This profile enables the Timing Analysts annotating assumption MSDs with timing specifications for the environment events triggering a simulation scenario, and it provides further specification means for the simulation configuration. The timing specifications define how often and at which instants an environment events triggering the system behavior can occur, and like MARTE [OMG11] we refer to them as *arrival patterns*.

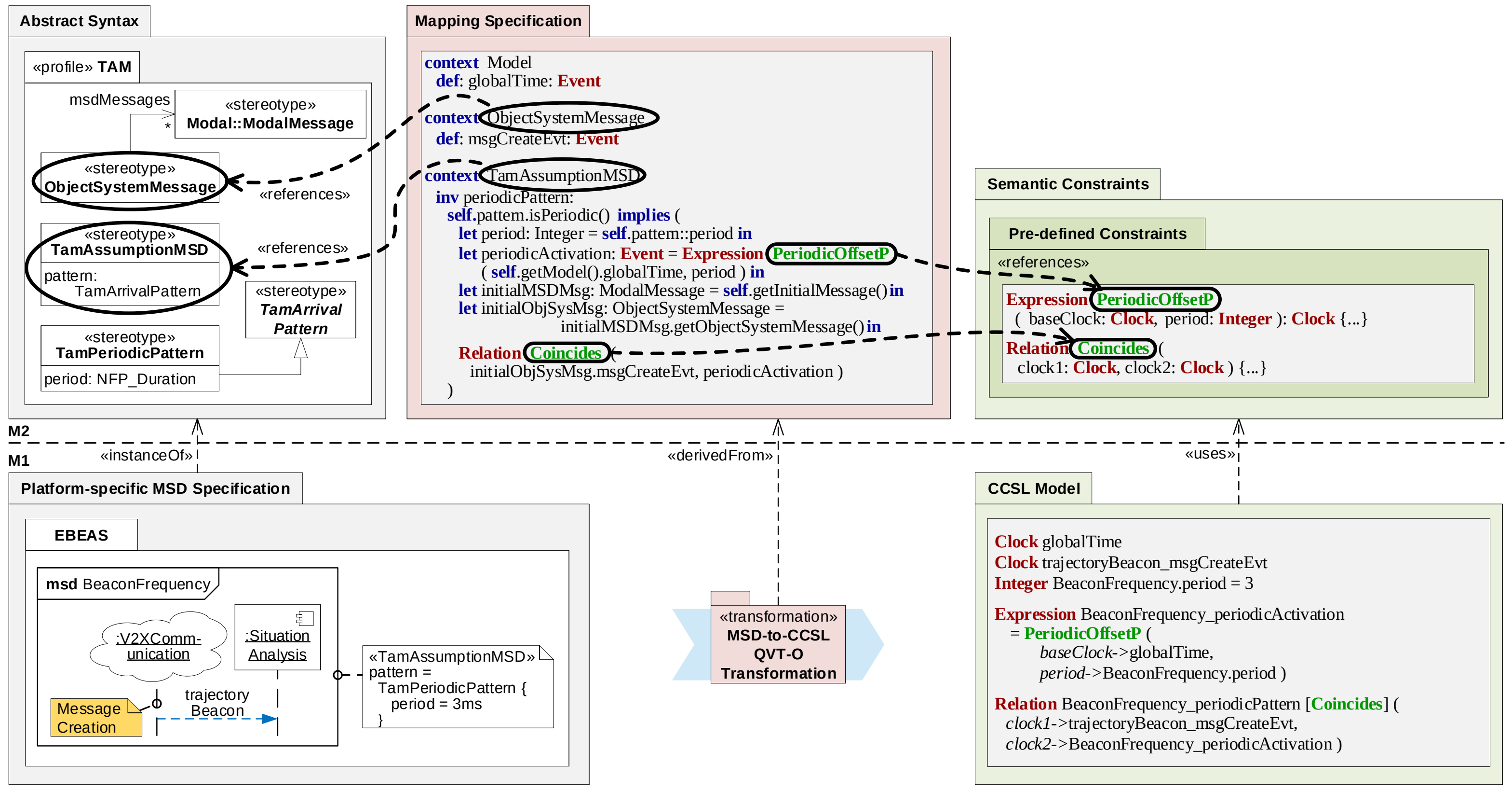
We support periodic as well as sporadic arrival patterns in our semantics. Whereas periodic arrival patterns specify the triggering of environment events that occur repeatedly with a fix period, sporadic arrival patterns specify the triggering of environment events that occur sporadically with certain restrictions. These restrictions encompass a minimum arrival rate before an event may occur, a maximum arrival rate until an event has to occur, and combinations of both.

Figure 4.20 depicts a specification excerpt of the semantics for the definition of analysis context scenarios with a periodic arrival rate at metamodel level M2 (upper part of the figure) as well as corresponding example models at metamodel level M1 (lower part of the figure). Figure 4.21 depicts an exemplary corresponding CCSL run at metamodel level M0. We provide specification excerpts, example models, and example CCSL runs for sporadic arrival patterns in Appendix B.1 (cf. Figures B.19 to B.24), and the complete ECL pseudocode specification for analysis contexts in Listing B.8 in Appendix B.2.

Metamodel Level M2

The upper part of Figure 4.20 depicts excerpts of the TAM profile, the ECL Mapping Specification, and of the applied semantic constraints.

The ECL Mapping Specification in the middle upper part of Figure 4.20 defines the invariant periodicPattern for the Tam stereotype «TamAssumptionMSD», inter alia. This invariant enforces the occurrence of message creation events associated with the initial MSD message of a «TamAssumptionMSD» to occur periodically. To this end, we first determine the period of the «TamPeriodicPattern» associated with the «TamAssumptionMSD». We define a new clock periodicActivation that ticks every period ticks by means of the pre-defined CCSL clock expression PeriodicOffsetP. Its arguments are the globalTime clock and the period value. Finally, we determine the object system message associated with the initial MSD message of the context TamAssumptionMSD. We enforce the msgCreateEvt DSE of this object system message to tick simultaneously with the periodicActivation clock through the pre-defined CCSL relation Coincides.


Abstract Syntax
«profile» TAM
msdMessages
«stereotype» Modal::ModalMessage
*
«stereotype» ObjectSystemMessage
«references»
«stereotype» TamAssumptionMSD
pattern: TamArrivalPattern
«references»
«stereotype» TamArrival Pattern
«stereotype» TamPeriodicPattern
period: NFP_Duration
Mapping Specification
Semantic Constraints
Pre-defined Constraints
«references»
M2
M1
«instanceOf»
«derivedFrom»
«uses»
Platform-specific MSD Specification
EBEAS
msd BeaconFrequency
:V2XComm-unication
:Situation Analysis
Message Creation
trajectory Beacon
«TamAssumptionMSD» pattern = TamPeriodicPattern { period = 3ms }
«transformation» MSD-to-CCSL QVT-O Transformation
CCSL Model

```
context Model
  def: globalTime: Event
context ObjectSystemMessage
  def: msgCreateEvt: Event
context TamAssumptionMSD
  inv periodicPattern:
    self.pattern.isPeriodic() implies (
      let period: Integer = self.pattern::period in
      let periodicActivation: Event = Expression PeriodicOffsetP
          ( self.getModel().globalTime, period ) in
      let initialMSDMsg: ModalMessage = self.getInitialMessage() in
      let initialObjSysMsg: ObjectSystemMessage =
                  initialMSDMsg.getObjectSystemMessage() in
      Relation Coincides(
        initialObjSysMsg.msgCreateEvt, periodicActivation )
    )
```

```
Expression PeriodicOffsetP
  ( baseClock: Clock, period: Integer ): Clock {...}
Relation Coincides(
  clock1: Clock, clock2: Clock ) {...}
```

```
Clock globalTime
Clock trajectoryBeacon_msgCreateEvt
Integer BeaconFrequency.period = 3

Expression BeaconFrequency_periodicActivation
   = PeriodicOffsetP (
        baseClock->globalTime,
        period->BeaconFrequency.period )

Relation BeaconFrequency_periodicPattern [Coincides] (
  clock1->trajectoryBeacon_msgCreateEvt,
  clock2->BeaconFrequency_periodicActivation )
```

Figure 4.20: Specification excerpt of semantics for periodic arrival patterns including example models

Metamodel Level M1

The lower part of Figure 4.20 exemplarily depicts excerpts of the Platform-specific MSD Specification and of the corresponding CCSL Model generated through the automatically derived MSD-to-CCSL QVT-O Transformation.

The MSD excerpt in the left lower part shows the «TamAssumptionMSD» BeaconFrequency. It specifies the MSD message `trajectoryBeacon` to be sent from the environment lifeline :V2XCommunication to the :SituationAnalysis. The contents of the stereotype «TamAssumptionMSD» define that this happens periodically every 3ms.

As indicated in the generated CCSL Model excerpt in the right lower part of Figure 4.20, the derived transformation translates the period of any «TamAssumptionMSD» with a periodic arrival pattern to each an Integer variable. For example, the tagged value period of the «TamPeriodicPattern» of the MSD BeaconFrequency is translated to the Integer variable BeaconFrequency.period with the value 3.

Furthermore, the transformation generates for any «TamAssumptionMSD» with a periodic arrival pattern each a CCSL model clock expression using the expression PeriodicOffsetP predefined at metamodel level M2. This CCSL model clock expression takes the globalTime as argument for the baseClock parameter and the Integer variable representing the period value as argument for the period clock parameter. For example, the transformation generates the PeriodicOffsetP expression, where the Integer variable BeaconFrequency.period is applied as argument for period. This expression defines the new clock BeaconFrequency_periodicActivation.

Finally, the transformation generates for any «TamAssumptionMSD» with a periodic arrival pattern each a CCSL model clock relation using the relation Coincides pre-defined at metamodel level M2. This clock relation gets the clock representing the message creation event of the object system message associated with the initial MSD message of the «TamAssumptionMSD» as argument for the parameter clock1. Furthermore, it gets the newly defined clock representing the periodic activation of the «TamAssumptionMSD» as argument for the parameter clock2. For example, the transformation generates the relation BeaconFrequency_periodicPattern of the type Coincides with the clocks trajectoryBeacon_msgCreateEvt and BeaconFrequency_-periodicActivation as arguments. This relation enforces the trajectoryBeacon_msgCreateEvt clock to tick every 3 instants, meaning that the message creation event of the object system message associated with the initial MSD message `trajectoryBeacon` occurs every 3ms.

Metamodel Level M0

Figure 4.21 depicts an exemplary CCSL run resulting from the CCSL model excerpt depicted in the right lower part of Figure 4.20. This CCSL run simulates the periodic occurrence of the message creation event of the object system message associated with the initial MSD message `trajectoryBeacon` in the «TamAssumptionMSD» BeaconFrequency.

The topmost row depicts the ticks of the reference clock globalTime. The middle row depicts the ticks of the clock $\text{BeaconFrequency}_{\text{periodicActivation}}$, which ticks every 3th tick of the globalTime clock. The bottommost row depicts the ticks of the clock $\text{trajectoryBeacon}_{\text{msgCreateEvt}}$, where the Coincides relation enforces this clock to tick simultaneously with the clock $\text{BeaconFrequency}_{\text{periodicActivation}}$.

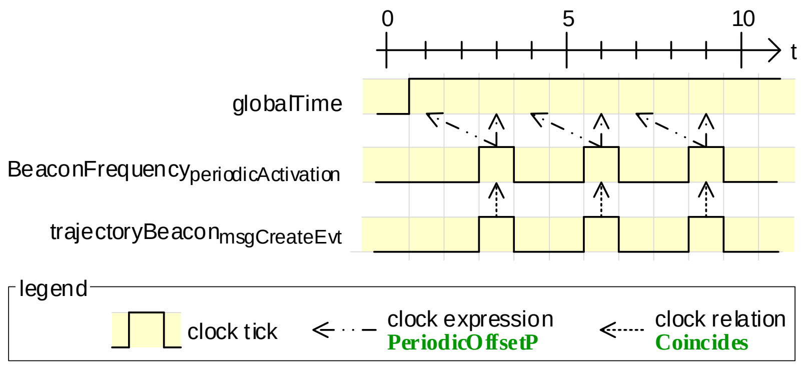


Figure 4.21: Examplary CCSL run simulating a periodic arrival pattern

4.5 Exemplary Timing Analysis

In this section, we illustrate how the different aspects of the semantics as presented in Section 4.4 work together. For this purpose, we explain an exemplary execution of a CCSL model automatically generated from the platform-specific MSD specification example presented in Section 4.1.

Figure 4.22 depicts an excerpt of the CCSL simulation run, where we only show the most important clocks. We refer to the particular semantics aspects for the particular complete underlying sets of clock expressions and relations as exemplified in Section 4.4 and Appendix B.1. Note that TIMESQUARE likewise allows to select only the clocks of interest in the VCD visualization although the BDD solver in the background considers the complete clock model (cf. Section 2.7.1). Figure B.30 in Appendix B.3 depicts the corresponding VCD screenshot resulting from the simulation in TIMESQUARE.

The topmost row depicts the tick of the clock $\text{trajectoryBeacon}_{\text{msgCreateEvt}}$ at instant 50. This tick stems from the environment message `trajectoryBeacon` in the «TamAssumptionMSD» BeaconFrequency (cf. Figure 4.5). The arrival pattern of this MSD specifies that the corresponding message event occurs periodically any 25ms. As explained in Section 4.4.3.2, the corresponding semantics enforce the message create unification (cf. Section 4.4.1.1) to occur not at other instants than this period. Thus, the message creation event $\text{trajectoryBeacon}_{\text{msgCreateEvt}}$ unifiable with the corresponding location (cf. Section 4.4.1.2) also occurs any 25ms, where the figure excerpt depicts its second tick in the overall run.

The MSD message `trajectoryBeacon` is sent via the logical connector from v2x: V2XCommunication to sa: SituationAnalysis. The ports of the TamECUs that these components are allocated to have no «TamComInterface» applied, and the «TamComConnection» that the logical connector is applied to has no quality-of-service information specified (cf. Figure 4.4). Thus, the not depicted message send, reception, and consumption events occur in the example run at the instants 51, 52, and 53, respectively. Row 2 depicts the subsequent tick of the clock $\text{trajectoryBeacon}_{\text{taskStartEvt}}$ at instant 54.

As described in Section 4.4.2.1, the static task execution delay until the following tick of the clock $\text{trajectoryBeacon}_{\text{taskCompleteEvt}}$ depicted in row 3 is computed as follows. The operation signature of `trajectoryBeacon` is a «TamOperation» having an execTime with the value 5ms (cf. Figure 4.4). The corresponding receiving component role sa: SituationAnalysis is allocated to the «TamECU» :µC1, whose «TamProcessingUnit» :PUµC1 has a speedFactor with the value 1 (cf. Figure 4.4). Thus, the task executing the operation needs $\frac{5\text{ms}}{1} = 5\text{ms}$ for the processing, and $\text{trajectoryBeacon}_{\text{taskCompleteEvt}}$ ticks at instant 59.

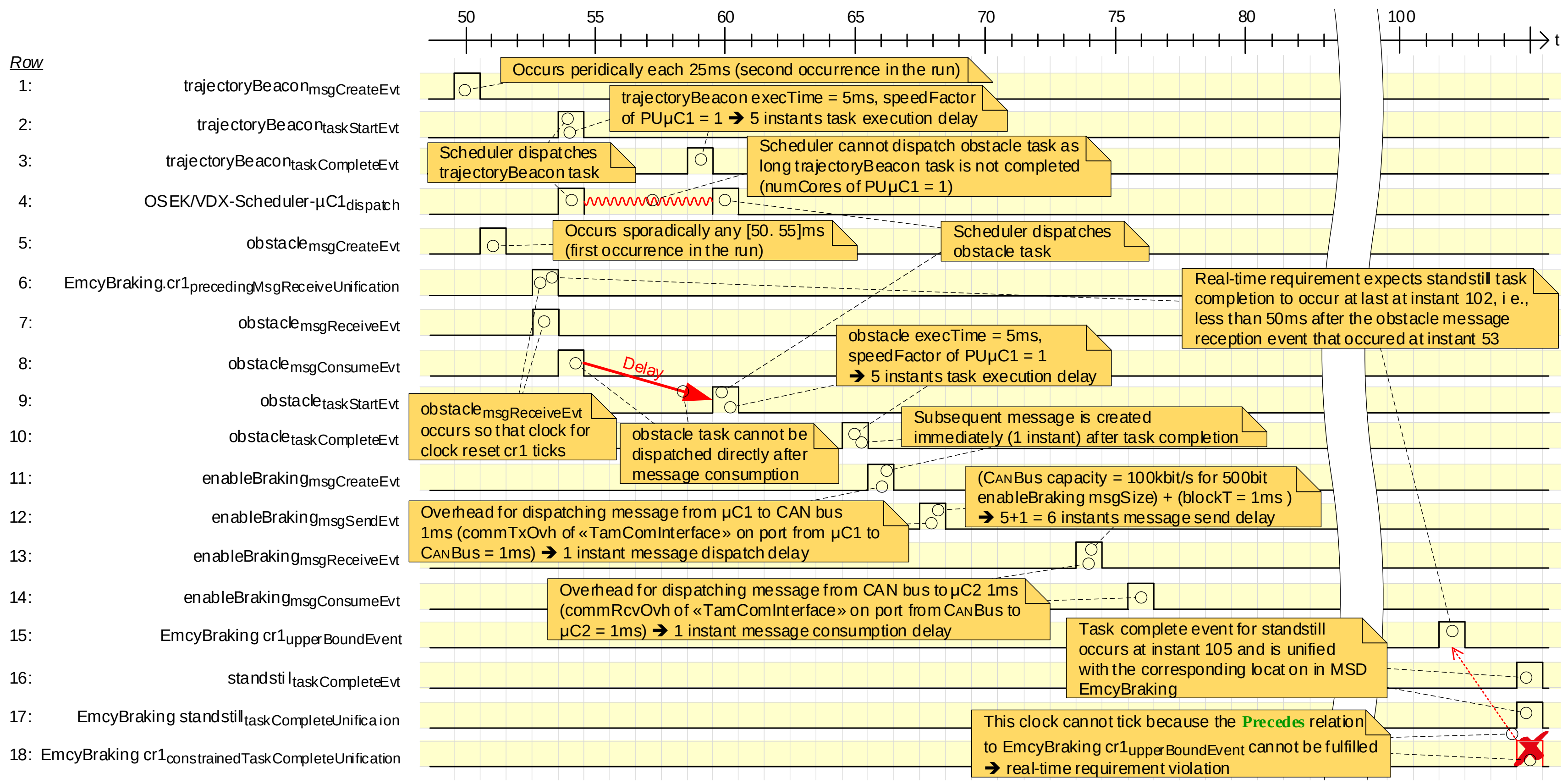

50
55
60
65
70
75
80
100
t
Row
1: trajectoryBeacon$_{msgCreateEvt}$
2: trajectoryBeacon$_{taskStartEvt}$
3: trajectoryBeacon$_{taskCompleteEvt}$
4: OSEK/VDX-Scheduler-µC1$_{dispatch}$
5: obstacle$_{msgCreateEvt}$
6: EmcyBraking.cr1$_{precedingMsgReceiveUnification}$
7: obstacle$_{msgReceiveEvt}$
8: obstacle$_{msgConsumeEvt}$
9: obstacle$_{taskStartEvt}$
10: obstacle$_{taskCompleteEvt}$
11: enableBraking$_{msgCreateEvt}$
12: enableBraking$_{msgSendEvt}$
13: enableBraking$_{msgReceiveEvt}$
14: enableBraking$_{msgConsumeEvt}$
15: EmcyBraking cr1$_{upperBoundEvent}$
16: standstill$_{taskCompleteEvt}$
17: EmcyBraking standstill$_{taskCompleteUnification}$
18: EmcyBraking cr1$_{constrainedTaskCompleteUnification}$
Occurs peridically each 25ms (second occurrence in the run)
trajectoryBeacon execTime = 5ms, speedFactor of PUµC1 = 1 ➔ 5 instants task execution delay
Scheduler dispatches trajectoryBeacon task
Scheduler cannot dispatch obstacle task as long trajectoryBeacon task is not completed (numCores of PUµC1 = 1)
Occurs sporadically any [50. 55]ms (first occurrence in the run)
Scheduler dispatches obstacle task
Real-time requirement expects standstill task completion to occur at last at instant 102, i e., less than 50ms after the obstacle message reception event that occured at instant 53
Delay
obstacle execTime = 5ms, speedFactor of PUµC1 = 1 ➔ 5 instants task execution delay
obstacle$_{msgReceiveEvt}$ occurs so that clock for clock reset cr1 ticks
obstacle task cannot be dispatched directly after message consumption
Subsequent message is created immediately (1 instant) after task completion
(CANBus capacity = 100kbit/s for 500bit enableBraking msgSize) + (blockT = 1ms) ➔ 5+1 = 6 instants message send delay
Overhead for dispatching message from µC1 to CAN bus 1ms (commTxOvh of «TamComInterface» on port from µC1 to CANBus = 1ms) ➔ 1 instant message dispatch delay
Overhead for dispatching message from CAN bus to µC2 1ms (commRcvOvh of «TamComInterface» on port from CANBus to µC2 = 1ms) ➔ 1 instant message consumption delay
Task complete event for standstill occurs at instant 105 and is unified with the corresponding locat on in MSD EmcyBraking
This clock cannot tick because the Precedes relation to EmcyBraking cr1$_{upperBoundEvent}$ cannot be fulfilled ➔ real-time requirement violation

Figure 4.22: Exemplary simulation run excerpt for the CCSL model automatically generated from the platform-specific MSD specification described in Section 4.1 (only selected clocks depicted)

Row 4 depicts the ticks of the clock OSEK/VDX-Scheduler-µC1$_{\text{dispatch}}$. As described in Section 4.4.2.2, this clock can only tick if the corresponding processing unit for the execution of a requested task dispatching has a free core. In our example, this is the case at instant 54, so that also the clock trajectoryBeacon$_{\text{taskStartEvt}}$ depicted in the row above is allowed to tick as mentioned above. However, OSEK/VDX-Scheduler-µC1$_{\text{dispatch}}$ must not tick at the following 5 instants. This is because the task for processing `obstacle` is not completed until then and the «TamProcessingUnit» :PUµC1 has numCores of 1 (cf. Figure 4.4).

Row 5 depicts the tick of the clock obstacle$_{\text{msgCreateEvt}}$. This tick stems from the environment message `obstacle` in the «TamAssumptionMSD» ObstacleArrivalRate (cf. Figure 4.5). The arrival pattern of this MSD specifies that the corresponding environment event occurs sporadically between any 50 and 55ms. As sketched in Figures B.23 and B.24 in Appendix B.1, the semantics enforce the corresponding message creation event to occur not at other instants outside this interval. Thus, the message creation event obstacle$_{\text{msgCreateEvt}}$ unifiable with the corresponding location (cf. Section 4.4.1.2) of the initial MSD message obstacle also occurs within this instant interval. In our example run, it occurs at instant 51, where the figure excerpt depicts its first tick in the overall run.

The row below depicts the tick of the clock EmcyBraking.cr1$_{\text{precedingMsgReceiveUnification}}$ at instant 53. This clock stems from the clock reset (assuming that it has the identifier cr1) in the MSD EmcyBraking (cf. Figure 4.2(c)), which is specified directly below the environment message `obstacle`. As explained in Section 4.4.3.1, the corresponding semantics enforce EmcyBraking.cr1$_{\text{precedingMsgReceiveUnification}}$ to tick on the last message receive unification occurrence before the clock reset. Thus, this clock ticks on the tick of the not depicted clock EmcyBraking.obstacle$_{\text{msgReceiveUnification}}$, which in turn ticks due to the tick of obstacle$_{\text{msgReceiveEvt}}$ (row 7, cf. Section 4.4.1.2).

Row 8 depicts the tick of the clock obstacle$_{\text{msgConsumeEvt}}$, which ticks at instant 54. As described in Section 4.4.2.2, the scheduler tries to dispatch the following task processing `obstacle` directly at the following instant. However, the executing «TamProcessingUnit» :PUµC1 has numCores of 1 (cf. Figure 4.4), and the task processing `trajectoryBeacon` has not finished yet (cf. rows 3 and 4). Thus, a dynamic delay occurs since the scheduler can dispatch the `obstacle` processing task not earlier than after the tick of the clock trajectoryBeacon$_{\text{taskCompleteEvt}}$ (cf. row 3). This results in simultaneous ticks of the clocks obstacle$_{\text{taskStartEvt}}$ and OSEK/VDX-Scheduler-µC1$_{\text{dispatch}}$ at instant 60.

The `obstacle` task is completed with the tick of the clock obstacle$_{\text{taskCompleteEvt}}$ at instant 65 (cf. row 10), where the task execution delay is computed analogously as described above for `trajectoryBeacon`. The next MSD message in the MSD EmcyBraking is `enableBraking` sent by the lifeline sa: SituationAnalysis in reaction to the environment message `obstacle` (cf. Figure 4.2(c)). As mentioned in Section 4.4.2.1, we do not consider an explicit delay between task completion and message creation but define the corresponding events to occur immediately consecutive for two subsequent messages covering the same lifeline. Thus, enableBraking$_{\text{msgCreateEvt}}$ occurs at instant 66 (cf. row 11).

In the following three paragraphs, we describe the timing effects and computation of the static message dispatch, send, and consumption delays (cf. Section 4.4.2.1 and Appendix B.1), respectively. The MSD message `enableBraking` is a message sent between distributed components, that is, sa: SituationAnalysis and vc: VehicleControl are allocated to different «TamECU»s (cf. Figure 4.4). As mentioned in Section 4.4.2.1, our semantics consider different and typically more platform properties for the resulting distributed message dispatch/send/consumption delays than for internal ones (see also Section 4.6.1.2).

Row 12 depicts the tick of the clock $\mathsf{enableBraking}_{\mathsf{msgSendEvt}}$. As sketched in Figures B.7 and B.8 in Appendix B.1, we compute the message dispatch delay between the preceding tick of the clock $\mathsf{enableBraking}_{\mathsf{msgCreateEvt}}$ and this tick by considering the underlying distributed platform communication properties. The sending software component as: SituationAnalysis is allocated to the «TamECU» :µC1, and the logical connector sa2vc sending the MSD message is allocated to the «TamComConnection» CANBus (cf. Figure 4.4). The port connecting :µC1 with the CANBus is a «TamComInterface» with a commTxOvh of 1ms. This specifies a duration overhead for encoding the message from its logical representation into a technical representation suitable for sending it via the CAN bus. Thus, the distributed message dispatch delay is one instant, and $\mathsf{enableBraking}_{\mathsf{msgSendEvt}}$ ticks at instant 68.

Row 13 depicts the tick of the clock $\mathsf{enableBraking}_{\mathsf{msgReceiveEvt}}$. As sketched in Figures B.9 and B.10 in Appendix B.1, we compute the message send delay between the preceding tick of the clock $\mathsf{enableBraking}_{\mathsf{msgSendEvt}}$ and this tick by considering the underlying distributed platform communication properties. The operation signature of the MSD message `enableBraking` has a msgSize of 500bit (cf. tagged value of the «TamOperation» `enableBraking()` in Figure 4.4). This message is sent via the «TamComConnection» CANBus, which has a throughput of 100kbit/s (cf. tagged value capacity in Figure 4.4). Furthermore, it is blocked for sending one message for 1ms (cf. tagged value blockT). Thus, the distributed message send delay is $\frac{500\text{bit}}{100\text{kbit/s}} + 1\text{ms} = \frac{500\text{bit}}{100\text{bit/ms}} + 1\text{ms} = 5\text{ms} + 1\text{ms}$, and $\mathsf{enableBraking}_{\mathsf{msgReceiveEvt}}$ ticks at instant 74.

Row 14 depicts the tick of the clock $\mathsf{enableBraking}_{\mathsf{msgConsumeEvt}}$. As sketched in Figures B.13 and B.14 in Appendix B.1, we compute the message consumption delay between the preceding tick of the clock $\mathsf{enableBraking}_{\mathsf{msgReceiveEvt}}$ and this tick by considering the underlying distributed platform communication properties. The receiving software component vc: VehicleControl is allocated to the «TamECU» :µC2, and the logical connector sa2vc sending the MSD message is allocated to the «TamComConnection» CANBus (cf. Figure 4.4). The port connecting the CANBus with :µC2 is a «TamComInterface» with a commRcvOvh of 1ms. This specifies a duration overhead for decoding the message from its technical bus representation into a logical representation suitable for the application software component vc: VehicleControl. Thus, the distributed message consumption delay is one instant, and $\mathsf{enableBraking}_{\mathsf{msgConsumeEvt}}$ ticks at instant 76.

The computation of the static delays and the determination of the dynamic delays for the particular events of the remaining MSD messages follows the same principle. Thus, we skip their description and focus at last on the clocks that are generated from the elements specified at the end of the MSD EmcyBraking.

Row 15 depicts the tick of the clock $\mathsf{EmcyBraking.cr1}_{\mathsf{upperBoundEvent}}$. As explained in Section 4.4.3.1, our semantics use this clock to represent the maximal delay $\mathtt{c} < \mathtt{50}$ w.r.t. the clock reset in the MSD EmcyBraking (cf. Figure 4.2(c)). As explained above, the clock reset is represented by the clock $\mathsf{EmcyBraking.cr1}_{\mathsf{precedingMsgReceiveUnificiation}}$ in row 6. In order to represent the maximal delay value 50, $\mathsf{EmcyBraking.cr1}_{\mathsf{upperBoundEvent}}$ ticks at instant 103, that is, 50 instants after the tick of $\mathsf{EmcyBraking.cr1}_{\mathsf{precedingMsgReceiveUnificiation}}$ at instant 53.

Due to the static and dynamic delays between the event occurrences before, the clock $\mathsf{standstill}_{\mathsf{taskCompleteEvt}}$ representing the final task completion event for the message `standstill` ticks at instant 105 (row 16). As explained in Section 4.4.1.2, this event is unified with corresponding MSD message location. This is represented by the clock $\mathsf{EmcyBraking.standstill}_{\mathsf{taskCompleteUnification}}$ ticking at the same instant (row 17). As explained in Section 4.4.3.1, this represents the last unification of a task completion event before a time condition and is captured by the clock EmcyBraking.cr1$_{\text{constrainedTaskCompleteUnification}}$ ticking at the same instant (row 18).

For maximal delays in terms of MSD real-time requirements, our semantics enforce the last task complete unification clock to tick before the upper bound event clock (cf. Section 4.4.1.2). However, due to the occurred platform-induced timing effects in terms of dynamic and static delays this is not the case for the clock EmcyBraking.cr1$_{\text{constrainedTaskCompleteUnification}}$. Thic clock ticks at instant 105 after EmcyBraking.cr1$_{\text{upperBoundEvent}}$, which ticks at instant 103. Thus, the BDD solver of TIMESQUARE (cf. Section 2.7.1) cannot solve the underlying Boolean expression, and the simulation stops with a deadlock. This represents a real-time requirement violation for the analysis context in which the `trajectoryBeacon` and `obstacle` are almost simultaneously received by the software component sa: SituationAnalysis.

The detection of such a real-time requirement violation typically opens up a variety of potential countermeasures to fix the defect. One possible countermeasure would be to speed up the «TamECU» :µC1 that the software component is allocated to: A speedFactor of 2 would allow to process `trajectoryBeacon` and `obstacle` within each 3 instants. This would reduce the end-to-end response time until the task completion of `standstill` by altogether 4 instants, thereby fulfilling the real-time requirement. Other countermeasures are the addition of an additional core to :µC1 enabling the concurrent processing of the two messages, an exchange of the communication media between the two TamECUs improving the message transmission times, the relaxation of the real-time requirement in communication with all stakeholders, etc.

4.6 Realization and Evaluation

We sketch the implementation aspects for the concepts described throughout this chapter in Section 4.6.1 and describe the conduct of a case study to evaluate the concepts in Section 4.6.2.

4.6.1 Implementation

Figure 4.23 depicts the expanded BPMN *sub-process* Compute Timing Information, which is collapsed in the overall process description depicted in Figure 4.6 (Section 4.2). This process encompasses the automatic process steps that the Timing Analysts conduct by means of our implementation, including the particular work product inputs and outputs. First, they determine the particular object system messages and compute their static delays in the automatic step Preprocessing, which we implemented through a QVT-O [OMG16; QVTo] model transformation. These object system messages are added to the Platform-specific MSD Specification. Subsequently, the Timing Analysts generate the CCSL model by means of GEMOC Studio [GEMOC] based on the object system messages and the pre-computed static delays. For this purpose, GEMOC Studio takes the Platform-specific MSD Specification, our ECL Specification, GEMOC's CCSL library (pre-defined CCSL constraints), and our MoCCML library (user-defined MoCCML constraints) as inputs.

Figure 4.24 depicts the coarse-grained software architecture that realizes the concepts described in this chapter. The architecture visualization encompasses the components and UML profiles newly implemented in the course of this thesis, the existing and hence reused frameworks, tool suites, and UML profiles, as well as the dependencies between these components. The overall implementation bases on the Eclipse Modeling Framework (EMF) [EMF] and hence applies the component EMF as root component.

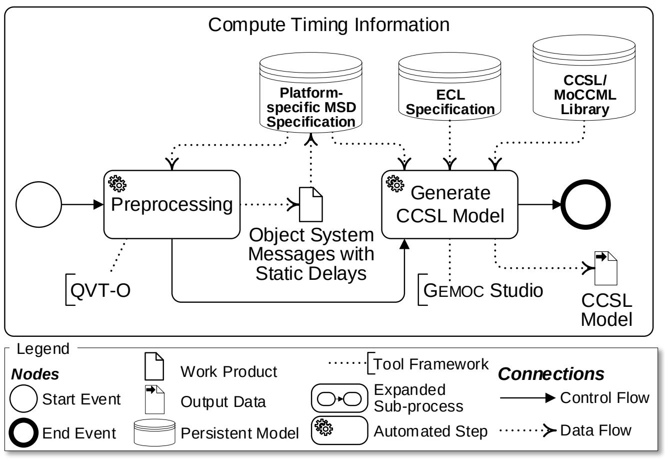


Figure 4.23: Expanded BPMN *sub-process* Compute Timing Information (cf. Figure 4.6)

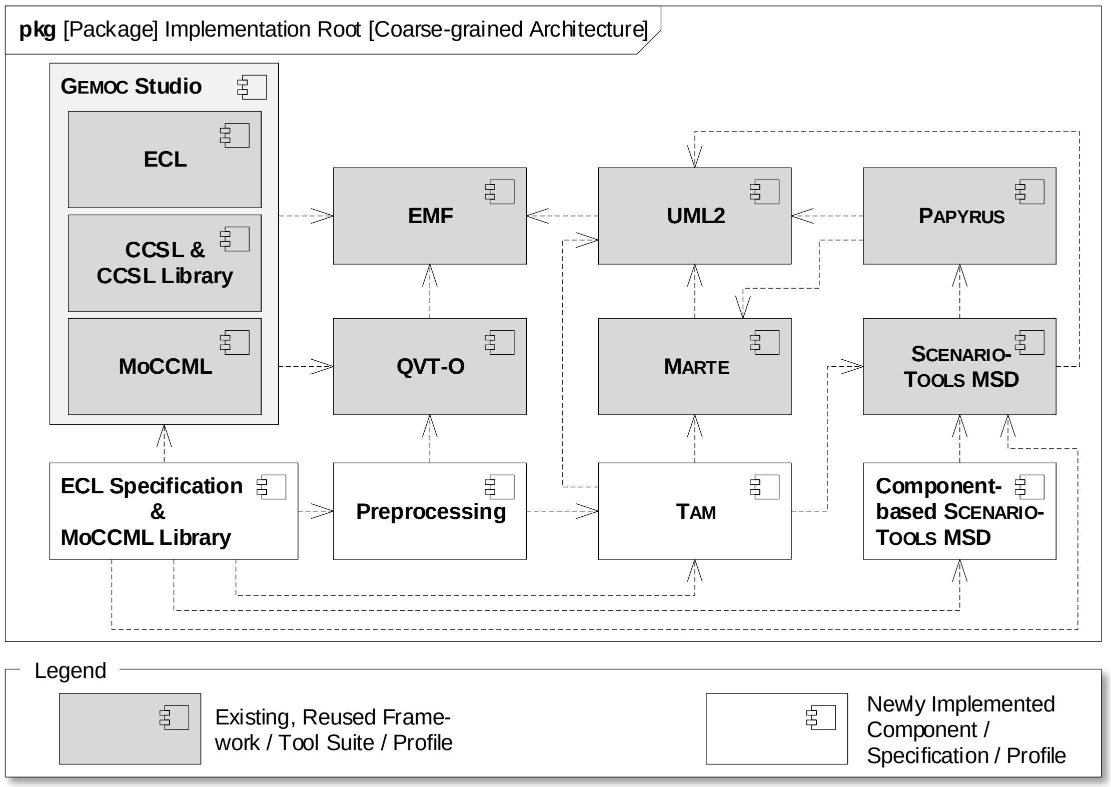

pkg [Package] Implementation Root [Coarse-grained Architecture]
GEMOC Studio
ECL
CCSL & CCSL Library
MoCCML
EMF
UML2
PAPYRUS
QVT-O
MARTE
SCENARIO-TOOLS MSD
ECL Specification & MoCCML Library
Preprocessing
TAM
Component-based SCENARIO-TOOLS MSD
Legend
Existing, Reused Frame-work / Tool Suite / Profile
Newly Implemented Component / Specification / Profile

Figure 4.24: Coarse-grained architecture of the implementation and the reused components

The TAM profile (cf. Section 4.1), which we present in detail in Section 4.6.1.1, as part of the component **Tam** extends both the MARTE profile as part of the component **Marte** and the Modal profile as part of **ScenarioTools MSD**. Furthermore, the profiles TAM, MARTE, and Modal extend the UML metamodel as part of the component **UML2** so that the respective components containing these profiles depend on **UML2**. The component **Component-based ScenarioTools MSD** extends the tool suite **ScenarioTools MSD** with hierarchical software components as structural basis for MSDs (cf. Sections 3.2 and 3.9.1). The component **Preprocessing** provides a set of QVT-O mappings and queries including Java black-box libraries for automatically setting up the particular object system messages and computing their static delays. Thus, it depends on the **QVT-O** framework [QVTo] as well as the **Tam** component, whose TAM profile defines the object system messages.

The component **ECL Specification & MoCCML Library** implements our MSD semantics dedicated to timing analyses (cf. Section 4.4 and Appendix B.2). Thus, it encompasses our ECL specification for the declaration of DSEs and our user-defined semantic constraints for the MoCC specified by means of MoCCML. The component thereby depends on the component **Gemoc Studio** [GEMOC]. This component provides the languages ECL, CCSL, and MoCCML in its corresponding subcomponents. Beyond the CCSL language, the component **CCSL & CCSL Library** provides also the pre-defined semantic constraints for the MoCC by means of a CCSL library, which we reference in our ECL specification.

In the following section, we present the TAM profile in detail. Subsequently, we present the concrete computation of the particular static delay kinds exploiting the TAM profile details and refining the abstract computations in Section 4.4.2.1 as part of the component **Preprocessing**.

4.6.1.1 The Timing Analysis Modeling (TAM) Profile in Detail

Figure 4.25 depicts the detailed overview of our Timing Analysis Modeling (TAM) profile. It imports the subprofiles GRM and GQAM as well as the model library MARTE_Library from MARTE (cf. Section 2.5.3). Furthermore, it imports the Modal profile (cf. Section 2.5.1). We divide TAM itself into the subprofiles Platform, ApplicationSoftware, AnalysisContext, and SimulationExtensions. Platform is further subdivided into the profiles ControlUnit, Communication, and OperatingSystem. We describe these subprofiles in the following.

Subprofile AnalysisContext

Figure 4.26 depicts the TAM subprofile AnalysisContext. It provides stereotypes for describing the analysis context of a concrete timing analysis in TIMESQUARE. The profile imports and specializes stereotypes of the MARTE subprofile GQAM (cf. Section 2.5.3.3). We describe the depicted stereotypes in the following.

TamAnalysisContext The analysis context for a concrete timing analysis in TIMESQUARE.

The OCL invariant restricts the properties derived from **GaAnalysisContext**. That is, it ensures that there is exactly one platform property that has the stereotype «TamResourcePlatform» applied and that the workload properties have the stereotype «TamWorkloadBehavior» applied.

TamResourcePlatform The root container for the platform model of a platform-specific MSD specification.

TamWorkloadBehavior A container for the MSDs to be analyzed for a concrete timing analysis in TIMESQUARE.

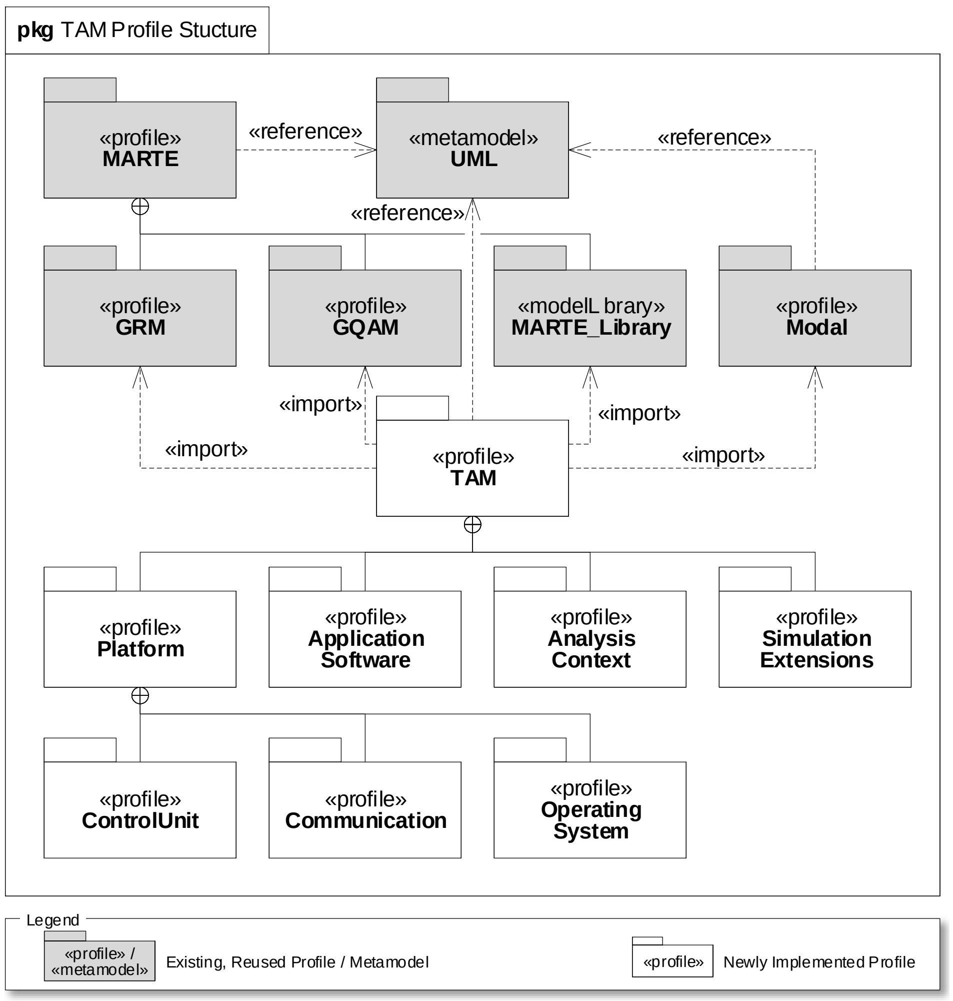


Figure 4.25: Detailed overview of the TAM subprofiles

The OCL invariant restricts the properties derived from **GaWorkloadBehavior**. That is, it ensures that the demand properties have the stereotype «TamAssumptionMSD» applied and that the behavior properties have the type UML **Interaction** and have not the stereotype «TamAssumptionMSD» applied.

TamAssumptionMSD An assumption MSD associating an arrival pattern.

TamArrivalPattern An abstract arrival pattern.

Note that MARTE also defines a multitude of arrival patterns with each a variety of fine-grained setting options including periodic and sporadic arrival patterns. However, we define our own stereotypes that are supported by our semantics for the sake of modeling language usability.

TamPeriodicPattern A periodic arrival pattern.

period The events unifiable with the MSD messages of the associated **TamAssumption-MSD** occur every periodth time unit.

TamSporadicPattern A sporadic arrival pattern.

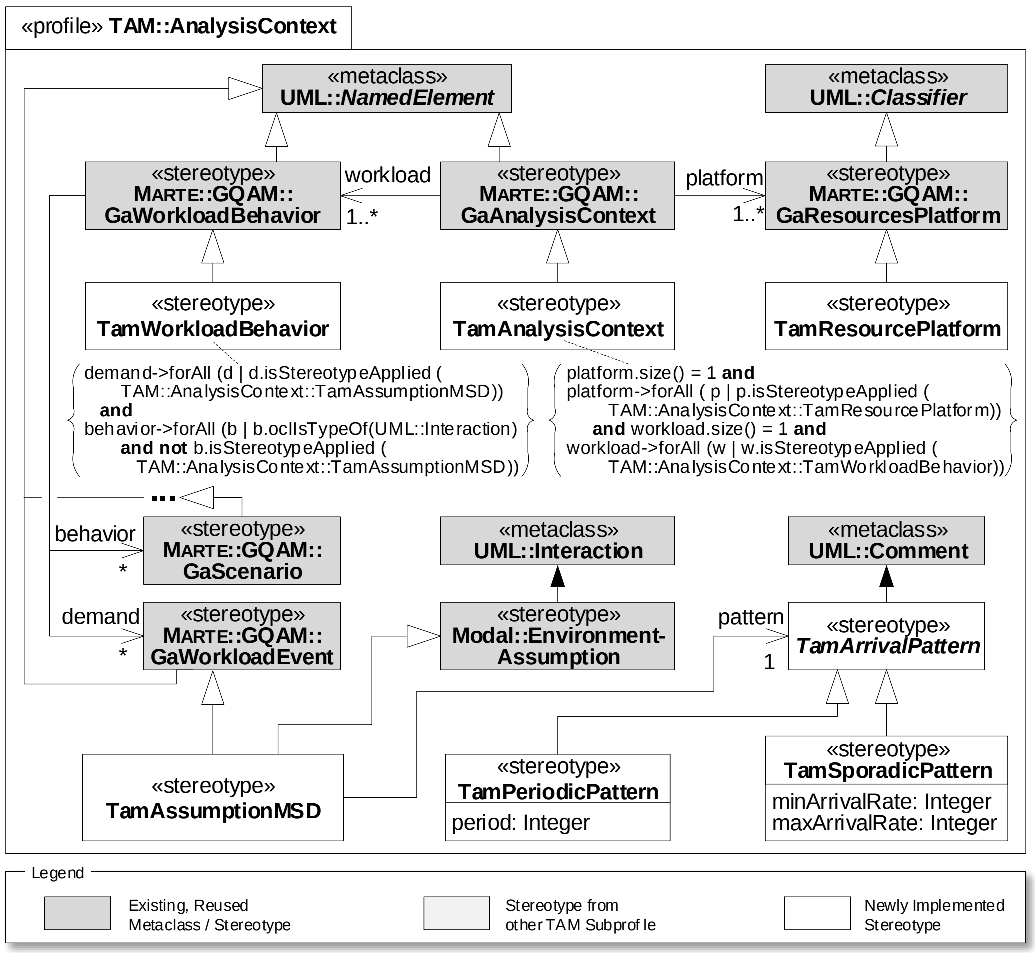

«profile» TAM::AnalysisContext
«metaclass» UML::NamedElement
«metaclass» UML::Classifier
«stereotype» MARTE::GQAM::GaWorkloadBehavior
workload
1..*
«stereotype» MARTE::GQAM::GaAnalysisContext
platform
1..*
«stereotype» MARTE::GQAM::GaResourcesPlatform
«stereotype» TamWorkloadBehavior
«stereotype» TamAnalysisContext
«stereotype» TamResourcePlatform
demand->forAll (d | d.isStereotypeApplied (TAM::AnalysisContext::TamAssumptionMSD)) and behavior->forAll (b | b.oclIsTypeOf(UML::Interaction) and not b.isStereotypeApplied (TAM::AnalysisContext::TamAssumptionMSD))
platform.size() = 1 and platform->forAll (p | p.isStereotypeApplied (TAM::AnalysisContext::TamResourcePlatform)) and workload.size() = 1 and workload->forAll (w | w.isStereotypeApplied (TAM::AnalysisContext::TamWorkloadBehavior))
behavior
*
«stereotype» MARTE::GQAM::GaScenario
«metaclass» UML::Interaction
«metaclass» UML::Comment
demand
*
«stereotype» MARTE::GQAM::GaWorkloadEvent
«stereotype» Modal::Environment-Assumption
pattern
1
«stereotype» TamArrivalPattern
«stereotype» TamAssumptionMSD
«stereotype» TamPeriodicPattern
period: Integer
«stereotype» TamSporadicPattern
minArrivalRate: Integer
maxArrivalRate: Integer
Legend
Existing, Reused Metaclass / Stereotype
Stereotype from other TAM Subprofile
Newly Implemented Stereotype

Figure 4.26: The TAM subprofile AnalysisContext

minArrivalRate The events unifiable with the MSD messages of the associated **TamAssumptionMSD** occur sporadically but at least every minArrivalRateth time unit.

maxArrivalRate The events unifiable with the MSD messages of the associated **TamAssumptionMSD** occur sporadically but at most every maxArrivalRateth time unit.

Subprofile Platform::Communication

Figure 4.27 depicts the TAM subprofile Platform::Communication. It provides stereotypes for describing the communication system of a platform for distributed software components. The profile imports and specializes stereotypes of the MARTE subprofiles GRM and GQAM (cf. Section 2.5.3.2, respectively). We describe the depicted stereotypes in the following.

TamComInterface The interface of an ECU with a communication system, encompassing hardware and driver properties. Applicable only to ports of **TamECUs** (cf. OCL constraint).

In contrast to the use of the super stereotype **GaExecHost** as intended by MARTE, we apply **TamComInterface** to ports of ECUs due to the adequate tagged values of the super stereotype (cf. Section 2.5.3.3). One **TamComConnection** connects two such stereotyped ports of different ECUs.

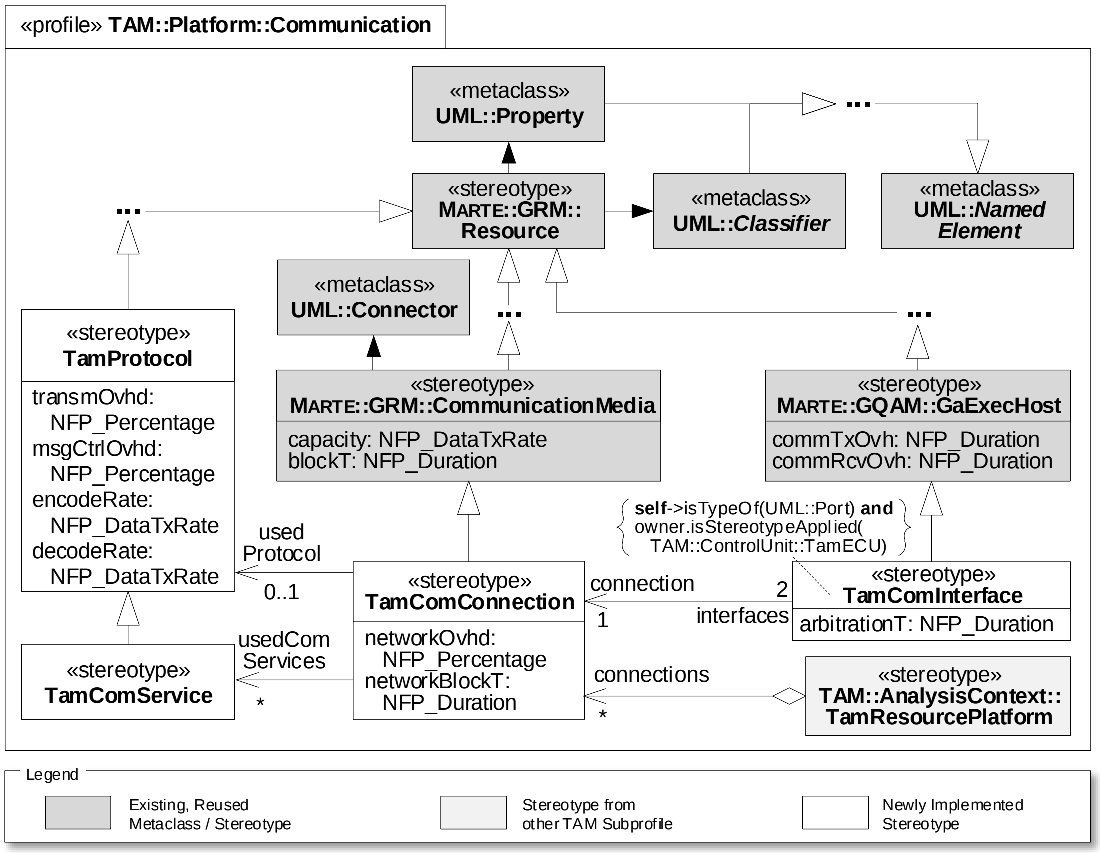

«profile» TAM::Platform::Communication
«metaclass» UML::Property
«stereotype» MARTE::GRM::Resource
«metaclass» UML::Classifier
«metaclass» UML::Named Element
«metaclass» UML::Connector
«stereotype» TamProtocol
transmOvhd: NFP_Percentage
msgCtrlOvhd: NFP_Percentage
encodeRate: NFP_DataTxRate
decodeRate: NFP_DataTxRate
«stereotype» MARTE::GRM::CommunicationMedia
capacity: NFP_DataTxRate
blockT: NFP_Duration
«stereotype» MARTE::GQAM::GaExecHost
commTxOvh: NFP_Duration
commRcvOvh: NFP_Duration
self->isTypeOf(UML::Port) and owner.isStereotypeApplied(TAM::ControlUnit::TamECU)
used Protocol
0..1
«stereotype» TamComConnection
networkOvhd: NFP_Percentage
networkBlockT: NFP_Duration
connection
2
1
interfaces
«stereotype» TamComInterface
arbitrationT: NFP_Duration
usedCom Services
*
«stereotype» TamComService
connections
*
«stereotype» TAM::AnalysisContext:: TamResourcePlatform
Legend
Existing, Reused Metaclass / Stereotype
Stereotype from other TAM Subprofile
Newly Implemented Stereotype

Figure 4.27: The TAM subprofile Platform::Communication

commTxOvh ECU -> communication system encoding duration at the sender.

commRcvOvh Communication system -> ECU decoding duration at the receiver.

arbitrationT Average waiting duration for gaining access of the sending communication node to the communication medium, additional to the ECU -> communication system encoding duration at the sender. This waiting duration is induced by the applied media access policy, which arbitrates the communication medium between all participating communication nodes.

TamComConnection A physical communication channel (e.g., a bus system) between two ECUs, as part of a **TamResourcePlatform**.

capacity The gross throughput of the communication channel.

blockT Net propagation delay duration of the communication channel.

networkOvhd Percentage network management overhead, reducing the gross throughput.

networkBlockT Network latency duration overhead, increasing the net propagation delay duration.

TamProtocol Transmission protocol applied by a communication channel (e.g., a bus protocol).

transmOvhd The percentage transmission protocol overhead additional to the net message size.

- **msgCtrlOvhd** The percentage message control overhead (e.g., induced by checksums) of the particular transmission protocol, increasing the net message size.
- **encodeRate** The transmission protocol's rate for encoding messages into the transmission protocol format at the sender.
- **decodeRate** The transmission protocol's rate for decoding messages from the transmission protocol format at the receiver.

TamComService A communication service provided by a middleware. A communication channel can apply several communication services.

- **transmOvhd** The percentage middleware communication service transmission overhead (e.g., induced by message buffers), reducing the gross throughput of a physical communication channel.
- **msgCtrlOvhd** The percentage message control overhead (e.g., induced by checksums) of the particular middleware communication service, increasing the net message size.
- **encodeRate** The middleware communication service's rate for encoding messages at the sender.
- **decodeRate** The middleware communication service's rate for decoding messages at the receiver.

Subprofile Platform::ControlUnit

Figure 4.28 depicts the TAM subprofile Platform::ControlUnit. It provides stereotypes for describing control units as part of an execution platform. The profile imports and specializes stereotypes of the MARTE subprofile GRM (cf. Section 2.5.3.2). We describe the depicted stereotypes in the following.

TamECU An ECU or a microcontroller, as part of a **TamResourcePlatform**.

TamProcessingUnit The actual processing unit of an ECU.

- **speedFactor** The processing speed factor relative to the speed of a reference processing unit.
- **numCores** The amount of processing cores.
- **coreSyncOvhd** The task execution overhead for synchronizing multiple processing cores, increasing the net operation execution time. The overhead is [0..0] in the case of $numCores = 1$ (cf. OCL constraint).

TamAccessibleResource An abstract accessible resource.

TamPeripheryUnit An abstract periphery entity.

TamSensor A sensor periphery entity.

- **accessDelay** The delay duration for accessing the sensor.

TamActuator An actuator periphery entity.

- **accessDelay** The delay duration for accessing the actuator.

TamIO An I/O periphery entity (e.g., human machine interfaces or interfaces for connections with external devices).

- **accessDelay** The delay duration for accessing the I/O device.

TamMemoryUnit A memory unit.

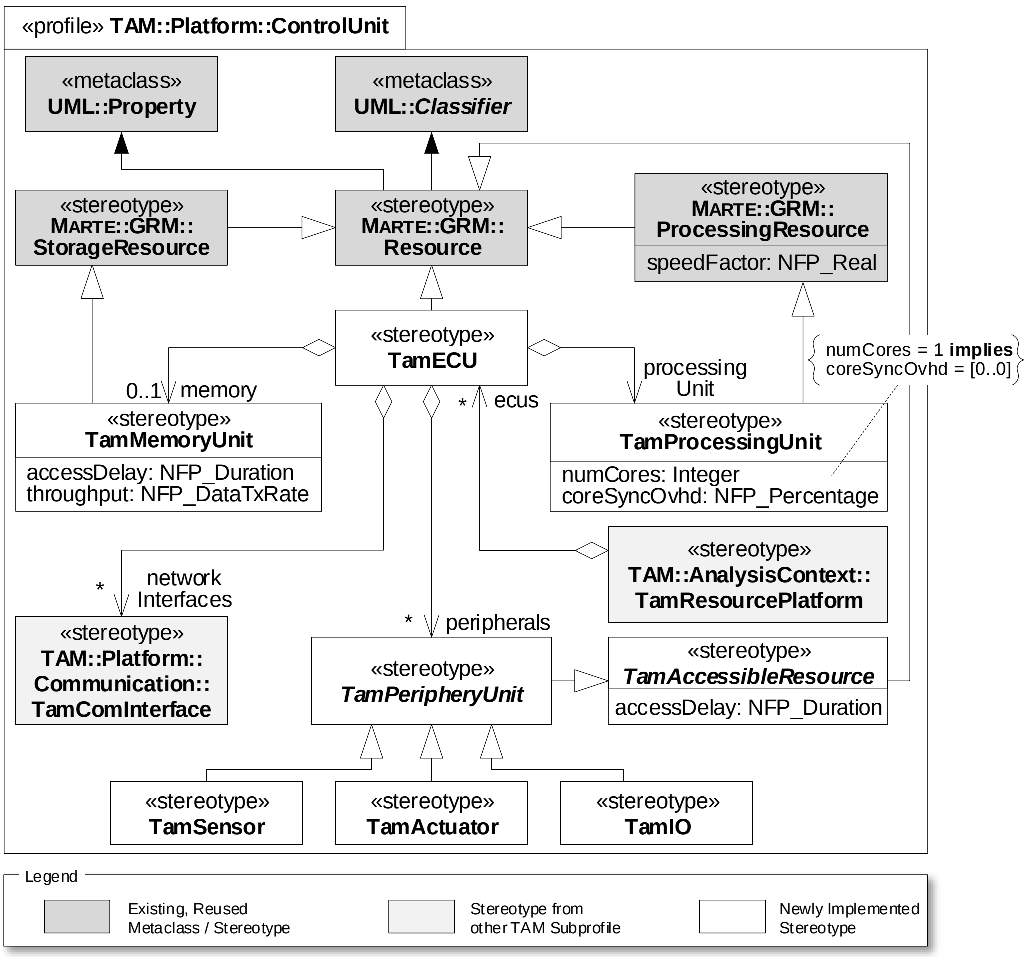

«profile» TAM::Platform::ControlUnit
«metaclass» UML::Property
«metaclass» UML::Classifier
«stereotype» MARTE::GRM::StorageResource
«stereotype» MARTE::GRM::Resource
«stereotype» MARTE::GRM::ProcessingResource
speedFactor: NFP_Real
«stereotype» TamECU
0..1 memory
processing Unit
* ecus
numCores = 1 implies coreSyncOvhd = [0..0]
«stereotype» TamMemoryUnit
accessDelay: NFP_Duration
throughput: NFP_DataTxRate
«stereotype» TamProcessingUnit
numCores: Integer
coreSyncOvhd: NFP_Percentage
«stereotype» TAM::AnalysisContext::TamResourcePlatform
* network Interfaces
«stereotype» TAM::Platform::Communication::TamComInterface
* peripherals
«stereotype» TamPeripheryUnit
«stereotype» TamAccessibleResource
accessDelay: NFP_Duration
«stereotype» TamSensor
«stereotype» TamActuator
«stereotype» TamIO
Legend
Existing, Reused Metaclass / Stereotype
Stereotype from other TAM Subprofile
Newly Implemented Stereotype

Figure 4.28: The TAM subprofile Platform::ControlUnit

accessDelay The delay duration for accessing the memory unit.

throughput The memory unit access throughput.

Subprofile Platform::OperatingSystem

Figure 4.29 depicts the TAM subprofile Platform::OperatingSystem. It provides stereotypes for describing the operating systems applied in a platform. The profile imports and specializes stereotypes of the MARTE subprofiles GRM and GQAM (cf. Sections 2.5.3.2 and 2.5.3.3, respectively). We describe the depicted stereotypes in the following.

TamOSResource An abstract operating system resource.

TamRTOS A real-time operating system as part of a **TamECU**.

backgroundUtilization The percentage task execution overhead induced by the real-time operating system, increasing the net operation execution time.

TamOSService A service provided by the real-time operating system.

backgroundUtilization The percentage task execution overhead induced by the service, increasing the net operation execution time.

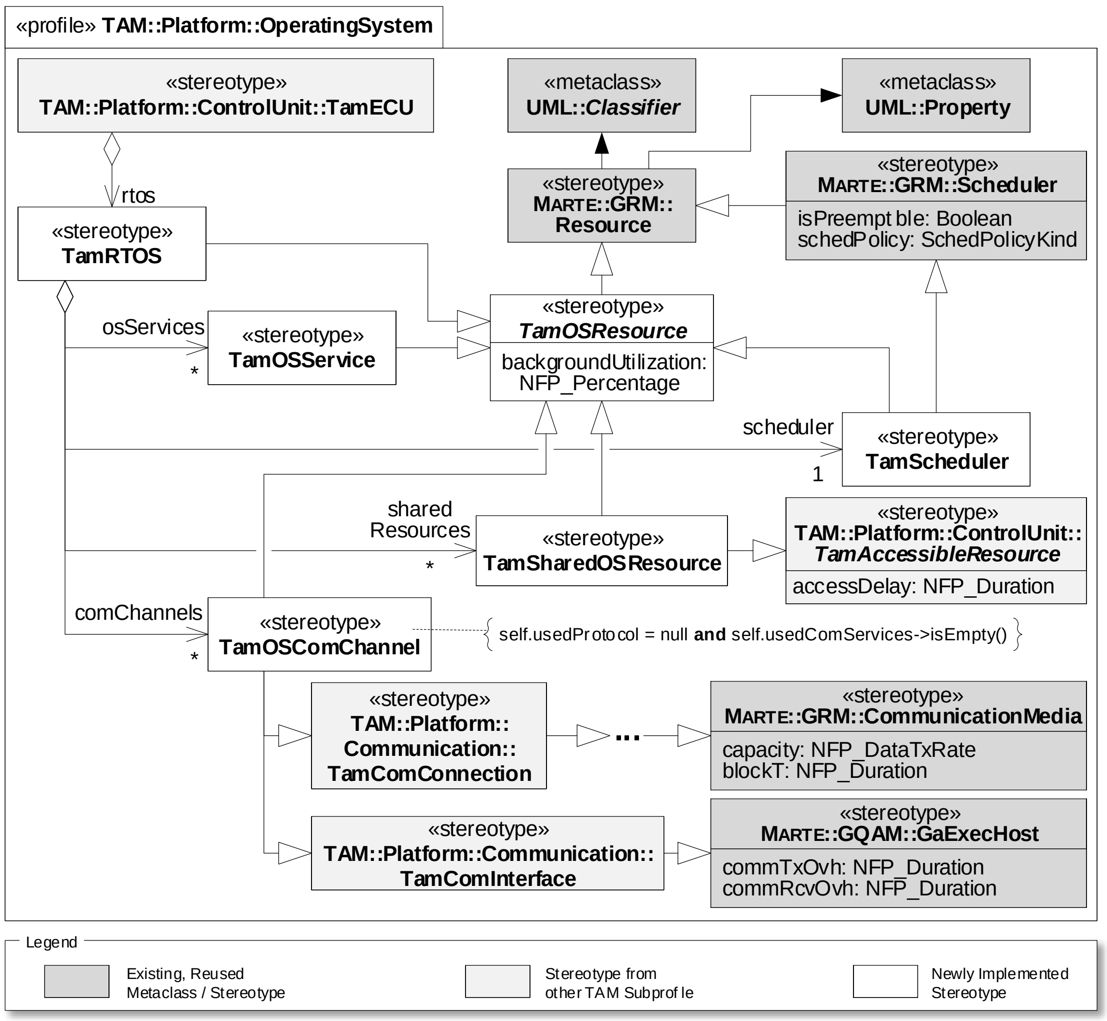

«profile» TAM::Platform::OperatingSystem
«stereotype» TAM::Platform::ControlUnit::TamECU
«metaclass» UML::Classifier
«metaclass» UML::Property
«stereotype» MARTE::GRM::Resource
«stereotype» MARTE::GRM::Scheduler
isPreemptible: Boolean
schedPolicy: SchedPolicyKind
rtos
«stereotype» TamRTOS
osServices
*
«stereotype» TamOSService
«stereotype» TamOSResource
backgroundUtilization: NFP_Percentage
scheduler
1
«stereotype» TamScheduler
shared Resources
*
«stereotype» TamSharedOSResource
«stereotype» TAM::Platform::ControlUnit::TamAccessibleResource
accessDelay: NFP_Duration
comChannels
*
«stereotype» TamOSComChannel
{ self.usedProtocol = null and self.usedComServices->isEmpty() }
«stereotype» TAM::Platform::Communication::TamComConnection
...
«stereotype» MARTE::GRM::CommunicationMedia
capacity: NFP_DataTxRate
blockT: NFP_Duration
«stereotype» TAM::Platform::Communication::TamComInterface
«stereotype» MARTE::GQAM::GaExecHost
commTxOvh: NFP_Duration
commRcvOvh: NFP_Duration
Legend
Existing, Reused Metaclass / Stereotype
Stereotype from other TAM Subprofile
Newly Implemented Stereotype

Figure 4.29: The TAM subprofile Platform::OperatingSystem

TamSharedOSResource A resource provided by the real-time operating system, where the service is shared with multiple applications.

backgroundUtilization The percentage task execution overhead induced by the resource, increasing the net operation execution time.

accessDelay The delay duration for accessing the shared operating system resource.

TamScheduler A scheduler as part of a real-time operating system of an ECU.

isPreemptible Specifies whether the scheduling algorithm is preemptive or not. Currently, our semantics only supports non-preemptive scheduling.

schedPolicy "Scheduling policy implemented by the scheduler." [OMG11, Section 10.3.2.15] Currently, our semantics only supports fixed-priority scheduling, which is the predominant scheduling policy for real-time systems [NMH08; DB08; SAÅ$^+$04].

backgroundUtilization The percentage task execution overhead induced by the scheduler, increasing the net operation execution time.

TamOSComChannel A communication channel provided by the real-time operating system (OS) for communicating software components that are deployed to the same ECU (e.g., a shared memory area). Such an internal communication channel must not use any **TamProtocol** nor any **TamComService** (cf. OCL constraint), which are applied only for distributed communication channels.

capacity The gross throughput of the OS communication channel.

blockT The net propagation delay duration of the OS communication channel.

backgroundUtilization The percentage task synchronization overhead induced by the OS communication channel, increasing the net message size.

commTxOvh The OS communication channel dispatching delay duration at the sender (e.g., the latency for writing to a shared memory area).

commRcvOvh The OS communication channel consumption delay duration at the receiver (e.g., the latency for reading from a shared memory area).

Subprofile ApplicationSoftware

Figure 4.30 depicts the TAM subprofile ApplicationSoftware. It provides stereotypes for describing the timing-relevant aspects of the application software. The profile imports and specializes stereotypes of the MARTE subprofile GRM (cf. Section 2.5.3.2). We describe the depicted stereotypes in the following.

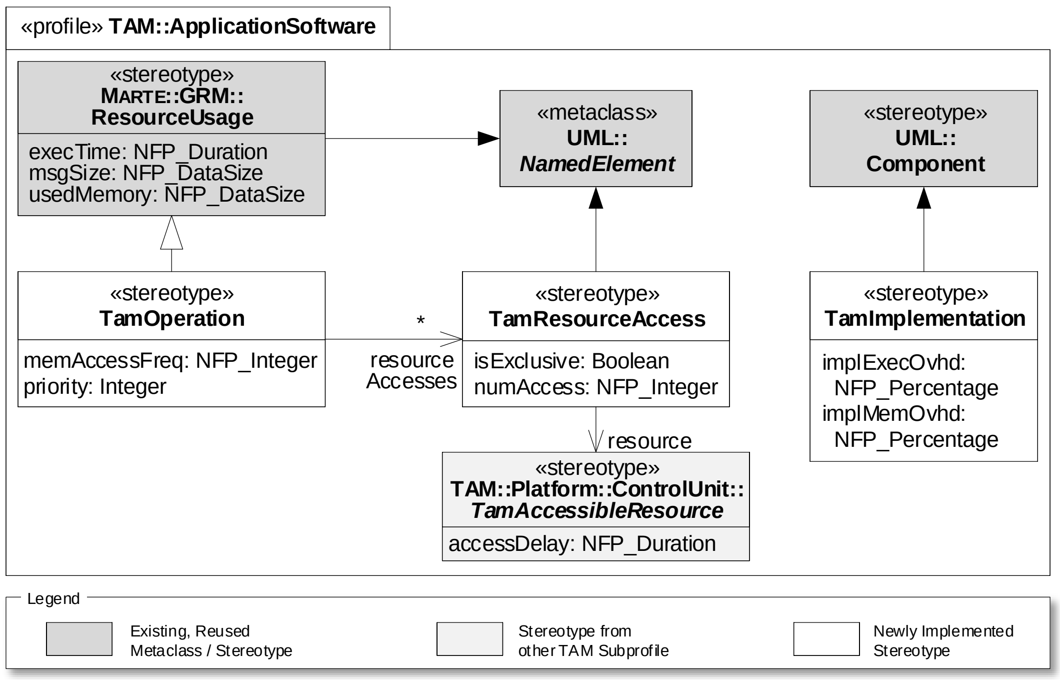

«profile» TAM::ApplicationSoftware
«stereotype» MARTE::GRM::ResourceUsage
execTime: NFP_Duration
msgSize: NFP_DataSize
usedMemory: NFP_DataSize
«metaclass» UML::NamedElement
«stereotype» UML::Component
«stereotype» TamOperation
memAccessFreq: NFP_Integer
priority: Integer
*
resourceAccesses
«stereotype» TamResourceAccess
isExclusive: Boolean
numAccess: NFP_Integer
resource
«stereotype» TAM::Platform::ControlUnit::TamAccessibleResource
accessDelay: NFP_Duration
«stereotype» TamImplementation
implExecOvhd: NFP_Percentage
implMemOvhd: NFP_Percentage
Legend
Existing, Reused Metaclass / Stereotype
Stereotype from other TAM Subprofile
Newly Implemented Stereotype

Figure 4.30: The TAM subprofile ApplicationSoftware

TamOperation An operation associated by an MSD message.

execTime Net operation execution time that a task needs for processing the associating MSD message.

msgSize Net size of the MSD message.

usedMemory Memory occupied on the memory unit of an ECU.

memAccessFreq Amount of memory accesses.

priority Priority for the task that processes the associated MSD message.

TamResourceAccess An access by an associating operation to an accessible resource.

isExclusive Specifies whether other operations may access the associated resource at the same time.

numAccess Amount of accesses to the associated resource.

TamImplementation Implementation details for a software component type.

implExecOvhd Percentage task execution overhead due to the programming language choice, compiler options, etc, increasing the net operation execution time.

implMemOvhd Percentage task execution overhead due to implementation-specific memory consumption, increasing the net operation execution time.

Subprofile SimulationExtensions

In Section 4.4.1.2, we motivated and introduced object system messages. In Figure 4.11 we defined that an object system message associates MSD messages, sender and receiver software component, the operation signature, and the logical connector between sender and receiver component.

Refining Figure 4.11, Figure 4.31 depicts the detailed TAM subprofile SimulationExtensions that defines further convenience information for the stereotype **ObjectSystemMessage**. This additional information encompasses associations to the platform-specific elements **TamProcessingUnit**, **TamSharedOSResource**, and **TamPeripheryUnit**. Furthermore, the additional information also encompasses for any static delay kind between the particular message event kinds as introduced in Section 4.4.2.1 each the minimum and maximum value as tagged value of **ObjectSystemMessage**. We present in Section 4.6.1.2 how these values are computed by means of a preprocessing step.

The additional information is also directly accessible in the ECL specification. However, accessing this information would lead to cumbersome ECL statements, particularly for the delay computation. Thus, we preprocess this information and store it as part of object system messages for convenience purposes.

4.6.1.2 Preprocessing

In Section 4.4.2.1, we presented abstract computations for the different static delay kinds between the particular message event kinds. In Section 4.6.1.1, we presented the detailed TAM profile including all stereotypes and tagged values. In the following, we present the detailed computations of the different static delay kinds between the particular message event kinds, exploiting the detailed TAM profile information. The computed delay values are stored as part of the object system messages (cf. Figure 4.31) before generating the CCSL model.

For the computation of all delay kinds except the task execution delay, we have to distinguish whether the communicating software components are deployed to the same ECU (i.e., intra-ECU communication) or to different ECUs (i.e., inter-ECU communication). In the case of intra-ECU communication, a communication channel provided by the operating system is applied for the message sending (e.g., a shared memory area). In the case of inter-ECU communication, a physical communication channel with a dedicated communication system is applied

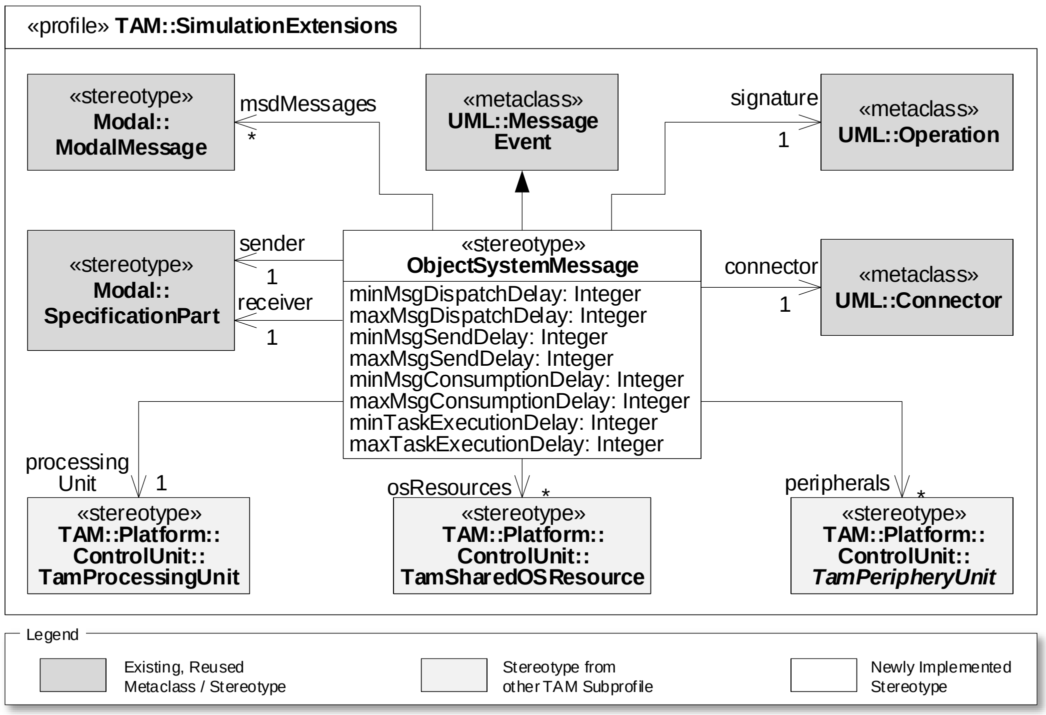

«profile» TAM::SimulationExtensions
«stereotype» Modal:: ModalMessage
msdMessages
*
«metaclass» UML::Message Event
signature
1
«metaclass» UML::Operation
«stereotype» Modal:: SpecificationPart
sender
1
receiver
1
«stereotype» ObjectSystemMessage
minMsgDispatchDelay: Integer
maxMsgDispatchDelay: Integer
minMsgSendDelay: Integer
maxMsgSendDelay: Integer
minMsgConsumptionDelay: Integer
maxMsgConsumptionDelay: Integer
minTaskExecutionDelay: Integer
maxTaskExecutionDelay: Integer
connector
1
«metaclass» UML::Connector
processing Unit
1
osResources
*
peripherals
*
«stereotype» TAM::Platform:: ControlUnit:: TamProcessingUnit
«stereotype» TAM::Platform:: ControlUnit:: TamSharedOSResource
«stereotype» TAM::Platform:: ControlUnit:: TamPeripheryUnit
Legend
Existing, Reused Metaclass / Stereotype
Stereotype from other TAM Subprofile
Newly Implemented Stereotype

Figure 4.31: The TAM subprofile SimulationExtensions

for the message sending (e.g., a bus system). The computation of the latter *distributed message delays* for inter-ECU communication considers more characteristics like network properties than the computation of the former *internal message delays* for intra-ECU communication.

Computation of Message Dispatch Delays

Message dispatch delays occur between message create events and message send events (cf. Sections 4.3.2 and 4.4.2.1).

In the case of intra-ECU communication, the internal message dispatch delay only encompasses the latency for writing to the communication channel. Refining Equation (4.1) in Section 4.4.2.1, we hence compute the internal minimum message dispatch delay (internal maximum message dispatch delay analogously) as

$$\begin{aligned}&msg::internalMsgDispatchDelay_{min}\\&\quad = msg.connector.supplier::commTxOvh_{min},\\&\qquad \text{where } msg.connector \text{ is a } \textbf{UML::Connector} \text{ associated by the MSD message } msg,\\&\qquad \text{and } msg.connector.supplier \text{ is a } \textbf{TamOSComChannel} \text{ that the connector is allocated to.}\end{aligned} \tag{4.5}$$

In the case of inter-ECU communication, the distributed message dispatch delay is determined by several characteristics. That is, the latency for accessing the communication channel, the arbitration time for gaining access to the overall communication system, and the time for encoding the message from its logical format into a format suitable for the communication system. The time for encoding messages for the communication between distributed software components depends on the overall message size in relation to the encode rate of the applied transmission protocol and of the applied middleware communication services. Refining Equation (4.1) in Section 4.4.2.1, we hence compute the distributed minimum message dispatch delay (distributed maximum message dispatch delay analogously) as

$$
\begin{aligned}
&msg{::}distributedMsgDispatchDelay_{min}\\
&\quad = msg.connector.supplier.interfaces[sender]{::}commTxOvh_{min}\\
&\qquad + msg.connector.supplier.interfaces[sender]{::}arbitrationT_{min}\\
&\qquad + \frac{msg{::}{<}overallDistributedMsgSize_{min}{>}}{msg.connector.supplier.usedProtocol{::}encodeRate_{max}}\\
&\qquad + \sum_{\substack{tamComService \in\\ msg.connector.supplier.usedComServices}} \frac{msg{::}{<}overallDistributedMsgSize_{min}{>}}{tamComService{::}encodeRate_{max}},
\end{aligned}
\tag{4.6}
$$

where $msg.connector$ is a **UML::Connector** associated by the MSD message msg,
$msg.connector.supplier$ is a **TamComConnection** that the connector is allocated to,
$msg.connector.supplier.interfaces$ are **TamComInterface**s of ECUs connected by the **TamComConnection**,
$msg.connector.supplier.interfaces[sender]$ is a **TamComInterface** of the sending ECU,
$msg{::}{<}overallDistributedMsgSize_{min}{>}$ is computed in Equation (4.7),
$msg.connector.supplier.usedProtocol$ is a **TamProtocol** of the **TamComConnection**,
and $msg.connector.supplier.usedComServices$ are the **TamComService**s of the **TamComConnection**.

(4.6)

In the case of distributed communication, the overall message size encompasses the net message size plus the percentage message control overheads (e.g., checksums) of the applied transmission protocol and of the applied middleware communication services. Thus, we compute the overall minimum size (overall maximum size analogously) for messages sent between two distributed software components (i.e., they are deployed to different ECUs) via a communication system as

$$
\begin{aligned}
&msg{::}{<}overallDistributedMsgSize_{min}{>}\\
&\quad = msg.signature{::}msgSize_{min}\\
&\qquad * (1 + msg.connector.supplier.usedProtocol{::}msgCtrlOvhd_{min})\\
&\qquad * \prod_{\substack{tamComService \in\\ msg.connector.supplier.usedComServices}} (1 + tamComService{::}msgCtrlOvhd_{min}),
\end{aligned}
$$

where $msg.signature$ is a **TamOperation** associated by the MSD message msg,
$msg.connector$ is a **UML::Connector** associated by the MSD message msg,
$msg.connector.supplier$ is a **TamComConnection** that the connector is allocated to,
$msg.connector.supplier.usedProtocol$ is a **TamProtocol** of the **TamComConnection**,
and $msg.connector.supplier.usedComServices$ are the **TamComService**s of the **TamComConnection**.

(4.7)

Computation of Message Send Delays

Message send delays occur between message send events and message reception events (cf. Sections 4.3.1 and 4.4.2.1).

In the case of intra-ECU communication, the internal message send delay encompasses the propagation delay and the net message size plus an operating system background utilization (e.g., a resource management overhead) in relation to the throughput of the operating system communication channel. Refining Equation (4.2) in Section 4.4.2.1, we hence compute the internal minimum message send delay (internal maximum send delay analogously) as

$$
\begin{aligned}
&msg{::}internalMsgSendDelay_{min} \\
&\quad = msg.connector.supplier{::}blockT_{min} \\
&\qquad + \frac{msg.signature{::}msgSize_{min} * (1 + msg.connector.supplier{::}backgroundUtilization_{min})}{msg.connector.supplier{::}capacity_{max}},
\end{aligned} \tag{4.8}
$$

where $msg.connector$ is a **UML::Connector** associated by the MSD message msg, $msg.connector.supplier$ is a **TamOSComChannel** that the connector is allocated to, and $msg.signature$ is a **TamOperation** associated by the MSD message msg.

In the case of inter-ECU communication, the distributed message send delay encompasses the latency of the physical communication channel, the latency of the overall network, and the time for the actual sending. The time for the actual sending depends on the overall message size in relation to the overall throughput of the underlying communication channel. Refining Equation (4.2) in Section 4.4.2.1, we hence compute the distributed minimum message send delay (distributed maximum send delay analogously) as

$$
\begin{aligned}
&msg{::}distributedMsgSendDelay_{min} \\
&\quad = msg.connector.supplier{::}blockT_{min} \\
&\qquad + msg.connector.supplier{::}networkBlockT_{min} \\
&\qquad + \frac{msg{::}\langle overallDistributedMsgSize_{min}\rangle}{msg.connector.supplier{::}\langle overallThroughput_{max}\rangle},
\end{aligned} \tag{4.9}
$$

where $msg.connector$ is a **UML::Connector** associated by the MSD message msg, $msg.connector.supplier$ is a **TamComConnection** that the connector is allocated to, $msg{::}\langle overallDistributedMsgSize_{min}\rangle$ is computed in Equation (4.7), and $msg.connector.supplier{::}\langle overallThroughput_{max}\rangle$ is computed in Equation (4.10).

The overall throughput of a physical communication channel encompasses its net throughput minus the percentage transmission overhead of the applied transmission protocol and of the applied middleware communication services. Thus, we compute the overall maximum throughput (overall minimum throughput analogously) of a message's connector allocated to a phyiscal communication channel connecting two different ECUs as

$$
\begin{aligned}
&msg.connector.supplier{::}\langle overallThroughput_{max}\rangle = msg.connector.supplier{::}capacity_{max} \\
&\quad * (1 - msg.connector.supplier{::}networkOvhd_{min}) \\
&\quad * (1 - msg.connector.supplier.usedProtocol{::}transmOvhd_{min}) \\
&\quad * \prod_{\substack{tamComService \in \\ msg.connector.supplier.usedComServices}} (1 - tamComService{::}transmOvhd_{min}),
\end{aligned} \tag{4.10}
$$

where $msg.connector$ is a **UML::Connector** associated by the MSD message msg, $msg.connector.supplier$ is a **TamComConnection** that the connector is allocated to, $msg.connector.supplier.usedProtocol$ is the **TamProtocol** of the **TamComConnection**, and $msg.connector.supplier.usedComServices$ are the **TamComService**s of the **TamComConnection**.

Computation of Message Consumption Delays

Message consumption delays occur between message reception events and message consumption events (cf. Sections 4.3.2 and 4.4.2.1).

In the case of intra-ECU communication, the internal message dispatch delay only encompasses the latency for reading from the applied operating system communication channel. Refining Equation (4.3) in Section 4.4.2.1, we hence compute the internal minimum message consumption delay (internal maximum message consumption delay analogously) as

$$\begin{aligned}&msg{::}internalMsgConsumptionDelay_{min}\\&\quad= msg.connector.supplier{::}commRcvOvh_{min},\\&\qquad\text{where } msg.connector \text{ is a } \textbf{UML::Connector} \text{ associated by the MSD message } msg,\\&\qquad\text{and } msg.connector.supplier \text{ is a } \textbf{TamOSComChannel} \text{ that the connector is allocated to.}\end{aligned} \tag{4.11}$$

In the case of inter-ECU communication, the distributed message dispatch delay encompasses the latency for accessing the communication channel and the time for decoding the message from a format suitable for the communication system into its logical representation. The time for decoding messages depends on the overall message size in relation to the decode rate of the applied transmission protocol and of the applied middleware communication services. Refining Equation (4.3) in Section 4.4.2.1, we hence compute the distributed minimum message consumption delay (distributed maximum message consumption delay analogously) as

$$\begin{aligned}&msg{::}distributedMsgConsumptionDelay_{min}\\&\quad= msg.connector.supplier.interfaces[receiver]{::}commRcvOvh_{min}\\&\quad+ \frac{msg{::}{<}overallDistributedMsgSize_{min}{>}}{msg.connector.supplier.usedProtocol{::}decodeRate_{max}}\\&\quad+ \sum_{\substack{tamComService\in\\ msg.connector.supplier.usedComServices}} \frac{msg{::}{<}overallDistributedMsgSize_{min}{>}}{tamComService{::}decodeRate_{max}},\\&\quad\text{where } msg.connector \text{ is a } \textbf{UML::Connector} \text{ associated by the MSD message } msg,\\&\quad msg.connector.supplier \text{ is a } \textbf{TamComConnection} \text{ that the connector is allocated to,}\\&\quad msg.connector.supplier.interfaces \text{ are } \textbf{TamComInterface}\text{s of ECUs connected by the } \textbf{TamComConnection},\\&\quad msg.connector.supplier.interfaces[receiver] \text{ is a } \textbf{TamComInterface} \text{ of the receiving ECU,}\\&\quad msg.connector.supplier.usedProtocol \text{ is the } \textbf{TamProtocol} \text{ of the } \textbf{TamComConnection},\\&\quad msg.connector.supplier.usedComServices \text{ are the } \textbf{TamComService}\text{s of the } \textbf{TamComConnection},\\&\quad\text{and } msg{::}{<}overallDistributedMsgSize_{min}{>} \text{ is computed in Equation (4.7).}\end{aligned} \tag{4.12}$$

Computation of Task Execution Delays

Task delays occur between message send events and message reception events (cf. Sections 4.3.3 and 4.4.2.1). They are determined by the normalized overall operation execution time required to process a message in relation to the processing power of the executing processing unit plus the overall times to access memory units as well as other resources. Refining Equation (4.4) in Section 4.4.2.1, we hence compute the minimum task execution delay (maximum task execution delay analogously) as

$$\begin{aligned}
&\mathit{msg}::\mathit{taskExecutionDelay}_{min}\\
&\quad= \frac{\mathit{msg}::<\mathit{normalizedOverallOperationExecTime}_{min}>}{\mathit{msg.connector}[\mathit{receiver}].\mathit{supplier.processingUnit}::\mathit{speedFactor}_{max}}\\
&\qquad+ \mathit{msg}::<\mathit{overallMemoryAccessTime}_{min}>\\
&\qquad+ \mathit{msg}::<\mathit{overallResourceAccessTime}_{min}>,
\end{aligned}$$

where $\mathit{msg}::<\mathit{normalizedOverallOperationExecTime}_{min}$ is computed in Equation (4.14),
$\mathit{msg.connector}$ is a **UML::Connector** associated by the MSD message msg,
$\mathit{msg.connector}[\mathit{receiver}]$ is a **Modal::SpecificationPart** representing the receiving software component,
$\mathit{msg.connector}[\mathit{receiver}].\mathit{supplier}$ is a **TamECU** that the receiving software component is allocated to,
$\mathit{msg.connector}[\mathit{receiver}].\mathit{supplier.processingUnit}$ is the **TamProcessingUnit** of the ECU,
$\mathit{msg}::<\mathit{overallMemoryAccessTime}_{min}$ is computed in Equation (4.16),
and $\mathit{msg}::<\mathit{overallResourceAccessTime}_{min}$ is computed in Equation (4.17).

(4.13)

The normalized overall operation execution time encompasses the net execution time plus overheads induced by the implementation, by a synchronization of potentially multiple cores, and by the overall background utilization of the real-time operating system. Thus, we compute the minimum normalized overall operation execution time (maximum normalized overall operation execution time analogously) as

$$\begin{aligned}
&\mathit{msg}::<\mathit{normalizedOverallOperationExecTime}_{min}>\\
&= \mathit{msg.signature}::\mathit{execTime}_{min}\\
&\quad * (1 + \mathit{msg.connector}[\mathit{receiver}].\mathit{type}::\mathit{implExecOvhd}_{min})\\
&\quad * (1 + \mathit{msg.connector}[\mathit{receiver}].\mathit{supplier.processingUnit}::\mathit{coreSyncOvhd}_{min})\\
&\quad * (1 + \mathit{msg.connector}[\mathit{receiver}].\mathit{supplier.rtos}::<\mathit{overallBackgroundUtilization}_{min}>),
\end{aligned}$$

where $\mathit{msg.signature}$ is a **TamOperation** associated by the MSD message msg,
$\mathit{msg.connector}$ is a **UML::Connector** associated by the MSD message msg,
$\mathit{msg.connector}[\mathit{receiver}]$ is a **Modal::SpecificationPart** representing the receiving software component,
$\mathit{msg.connector}[\mathit{receiver}].\mathit{type}$ is a **TamImplementation** of the receiving software component type,
$\mathit{msg.connector}[\mathit{receiver}].\mathit{supplier}$ is a **TamECU** that the receiving software component is allocated to,
$\mathit{msg.connector}[\mathit{receiver}].\mathit{supplier.processingUnit}$ is the **TamProcessingUnit** of the ECU,
$\mathit{processingUnit}::\mathit{coreSyncOvhd}_{min} = 0$ if $\mathit{processingUnit}::\mathit{numCores} = 1$ (cf. OCL constraint in Figure 4.28),
$\mathit{msg.connector}[\mathit{receiver}].\mathit{supplier.rtos}$ is the **TamRTOS** of the ECU,
and $\mathit{rtos}::<\mathit{overallBackgroundUtilization}_{min}>$ is computed in Equation (4.15).

(4.14)

The overall background utilization of the real-time operating system encompasses its own background utilization as well as the background utilization of its scheduler, of its communication channels, of its shared resources, and of its services. Thus, we compute the minimum overall background utilization (maximum overall background utilization analogously) as

$$\begin{aligned}
&msg.connector[receiver].supplier.rtos::<overallBackgroundUtilization_{min}> \\
&= msg.connector[receiver].supplier.rtos::backgroundUtilization_{min} \\
&\quad + msg.connector[receiver].supplier.rtos.scheduler::backgroundUtilization_{min}) \\
&\quad + \sum_{\substack{tamComChannel\in \\ msg.connector[receiver].supplier.rtos.comChannels}} (tamComChannel::backgroundUtilization_{min}) \\
&\quad + \sum_{\substack{tamOSResource\in \\ msg.connector[receiver].supplier.rtos.sharedResources}} (tamOSResource::backgroundUtilization_{min}) \\
&\quad + \sum_{\substack{tamOSService\in \\ msg.connector[receiver].supplier.rtos.osServices}} tamOSService::backgroundUtilization_{min}),
\end{aligned} \tag{4.15}$$

where $msg.connector$ is a **UML::Connector** associated by the MSD message msg,
$msg.connector[receiver]$ is a **Modal::SpecificationPart** representing the receiving software component,
$msg.connector[receiver].supplier$ is a **TamECU** that the receiving software component is allocated to,
$msg.connector[receiver].supplier.rtos$ is the **TamRTOS** of the ECU,
$msg.connector[receiver].supplier.rtos.scheduler$ is the **TamScheduler** of the RTOS,
$msg.connector[receiver].supplier.rtos.comChannels$ are **TamComChannel**s of the RTOS,
$msg.connector[receiver].supplier.rtos.sharedResources$ are **TamSharedOSResource**s of the RTOS,
and $msg.connector[receiver].supplier.rtos.osServices$ are **TamOSService**s of the RTOS.

The overall time for accessing a memory unit encompasses the frequency of memory accesses multiplied with the particular access delays as well as the used net memory plus a implementation-specific overhead in relation to the memory throughput. Thus, we compute the minimum overall memory access time (maximum overall memory access time analogously) as

$$\begin{aligned}
&msg::<overallMemoryAccessTime_{min}> \\
&\quad = (msg.signature::memAccessFreq_{min} * msg.connector[receiver].supplier.memory::accessDelay_{min}) \\
&\quad + \frac{msg.signature::usedMemory_{min} * (1 + msg.connector[receiver].type::implMemOvhd_{min})}{msg.connector[receiver].supplier.memory::throughput_{max}},
\end{aligned}$$

where $msg.signature$ is a **TamOperation** associated by the MSD message msg,
$msg.connector$ is a **UML::Connector** associated by the MSD message msg,
$msg.connector[receiver]$ is a **Modal::SpecificationPart** representing the receiving software component,
$msg.connector[receiver].type$ is a **TamImplementation** of the receiving software component type,
$msg.connector[receiver].supplier$ is a **TamECU** that the receiving software component is allocated to,
$msg.connector[receiver].supplier.memory$ is the **TamMemoryUnit** of the ECU.

(4.16)

The overall time for accessing periphery and operating system resources is determined by the number of accesses multiplied with the particular access delays, summed up over all resource accesses. Thus, we compute the minimum overall resource access time (maximum overall resource access time analogously) as

$$\begin{aligned}
&msg::<overallResourceAccessTime_{min}> \\
&= \sum_{\substack{tamResourceAccess\in \\ msg.signature.resourceAccesses}} tamResourceAccess::numAccess_{min} * tamResourceAccess.resource::accessDelay_{min},
\end{aligned} \tag{4.17}$$

where $msg.signature$ is a **TamOperation** associated by the MSD message msg,
$msg.signature.resourceAccesses$ are **TamResourceAccess**es associated by the operation,
and $tamResourceAccess.resource$ is a **TamAccessibleResource** associated by the resource access.

4.6.2 Case Study

We conduct a case study based on the guidelines by Kitchenham et al. [KPP95] and by Runeson et al. [RHAR12; RH08] for the evaluation of our timing analysis approach. In our case study, we investigate the applicability of our approach within the domain of software-intensive systems.

4.6.2.1 Case Study Context and Cases

The objective of our case study is to evaluate whether our timing analysis approach is useful for the Timing Analysts. For this purpose, we evaluate the following questions:

Evaluation Question EQ1 Does our timing analysis approach generate syntactically and semantically correct CCSL models?

Evaluation Question EQ2 Does our timing analysis approach reduce the engineering effort for conceiving and specifying CCSL models?

We conduct the case study with the case of the EBEAS. In order to answer the evaluation questions above, we apply different variants of the platform-independent EBEAS MSD specification introduced in the last chapter. These variants are complemented by platform-specific information as exemplarily shown in extracts throughout this chapter (cf. particularly Section 4.1), resulting in platform-specific MSD specifications.

4.6.2.2 Setting the Hypotheses

Based on the aforementioned case study objective and evaluation questions, we define the following evaluation hypotheses:

Hypothesis H1 Our MSD semantics for timing analyses correctly encodes the timing effects that are induced by the platform properties provided as modeling means by our TAM profile (cf. evaluation question EQ1).

For evaluating H1, two different students prepare a set of platform-specific MSD specifications that jointly cover all platform properties that are provided as modeling means by our TAM profile. Afterward, they investigate whether any of the platform properties induces each the expected timing effect with the expected delay duration.

We consider H1 fulfilled if at least 90% of the platform properties specifiable with our TAM profile are covered by our semantics, and from these $\geq$90% each induced timing effect is observed as expected (i.e., 100% test coverage).

Hypothesis H2 A platform-specific MSD specification is more compact than the corresponding derived CCSL model (cf. evaluation question EQ2).

For evaluating H2, two different students conceive three different platform-specific MSD specifications with each different model element amounts and generate CCSL models from them. Afterward, we count for any platform-specific MSD specification and the corresponding CCSL model each the model elements and compare their respective amounts.

We consider H2 fulfilled if at least as many CCSL model elements as MSD specification elements are generated.

Hypothesis H3 Automatically deriving CCSL models from platform-specific MSD specifications with our timing analysis approach is more efficient conceiving and specifying them manually (cf. evaluation question EQ2).

For evaluating H3, a student measures the time for the automatic derivation of the CCSL models from the three platform-specific MSD specifications conceived for hypothesis H2. Furthermore, we take the CCSL model element amounts determined for H2 into account.

We consider H3 fulfilled if at least one CCSL model element is generated per second.

Hypothesis H4 The generated CCSL models are syntactically correct (cf. evaluation question EQ1).

For evaluating H4, all generated CCSL models used for the evaluation of H1-H3 are opened in the CCSL model editor and simulated in TIMESQUARE.

We consider H4 fulfilled if all CCSL models generated during the evaluation of H1-H3 can be opened in the CCSL model editor and can be simulated in TIMESQUARE without the occurrence of any error.

4.6.2.3 Data Collection Preparation

Besides ourselves, we employ two different students *student-1* and *student-2* to support the evaluation. Student-1 has approximately four years experience in modeling and simulating MSD specifications as well as one year experience with the GEMOC approach during the case study conduct. Furthermore, he conceived and implemented the initial version of our timing analysis approach [*Ber17]. Student-2 has approximately one year experience with modeling and simulating MSD specifications as well as with the GEMOC approach during the case study conduct.

As a basis for evaluating H1, the students prepare a set of platform-specific MSD specifications that jointly cover all platform properties that are supported by our semantics.

A large part of these platform properties is covered by a platform-specific MSD specification that is presented in [*Ber17, Section 7.2] as a proof of concept by student-1, which we call *MSD-spec-1*. In this proof of concept, typical use cases in the course of a timing analysis are constructed, where the Timing Analysts determine several real-time requirement violations through the timing analysis and adapt the specification multiple times until the real-time requirements are fulfilled.

For any of the remaining platform properties that are not covered by MSD-spec-1, student-2 specifies each a dedicated model (altogether 17 further models) that covers the respective platform property to reproduce the corresponding induced timing effect. For this purpose, he initially creates a base model adapted from MSD-spec-1. Subsequently, for any platform property to investigate he creates each a dedicated copy of the base model and adds the property.

MSD-spec-1 has only one system-internal TamComConnection between two system-internal TamECUs and each two requirement MSDs and TamAssumptionMSDs. As a basis for evaluating H2 and H3, student-2 copies MSD-spec-1, completes the platform model to five TamComConnections connecting five TamECUs, and allocates the software architecture to it. We present the resulting model in Figure 4.4 and call it *MSD-spec-2*. Furthermore, student-1 specifies a variant of the platform-independent MSD specification introduced in the last chapter encompassing one merged MSD use case with 24 MSDs (cf. Appendix A.2.2.3) and allocates it to the platform model of MSD-spec-1, which we call *MSD-spec-3*. Finally, student-2 copies the variant of the platform-independent MSD specification introduced in the last chapter, extends the platform model of MSD-spec-2 (i.e., he adds ECUs as well as bus connections for the lane keeping assist and the precrash unit), and adds the allocation specification. We call the resulting model *MSD-spec-4*.

4.6.2.4 Data Collection Procedure

We describe the respective procedures for the data collection for evaluating the four hypotheses in the following four paragraphs.

Hypothesis H1
For evaluating hypothesis H1 with MSD-spec-1, student-1 conducts the specification and timing analysis process described in Section 4.2:

1. He specifies a platform-independent MSD specification in terms of a variant of the EBEAS MSD specification as introduced in the last chapter, playing the role of the Software Requirements Engineer.
2. He specifies a platform model based on information about real-world platforms, playing the role of the Platform Architect.
3. He specifies an allocation from the MSD specification to the platform model and annotates software component resource consumption properties, playing the role of the Allocation Engineer.
4. He specifies analysis contexts and iteratively conducts the timing analysis in TIMESQUARE, playing the role of the Timing Analyst. In the course of the timing analysis, he iteratively encounters platform-induced real-time requirement violations, determines their respective causes, and adapts the platform properties until all real-time requirements are fulfilled. This procedure enables him to reenact every timing effect induced by a platform property that is both considered by our semantics and specified in the proof of concept model. Particularly, he simulatively determines whether for any specified platform property each the expected timing effect occurs.

In order to evaluate hypothesis H1 for the remaining platform properties not covered by MSD-spec-1, student-2 proceeds for any dedicated model specific to a platform property as follows:

1. He specifies the platform property with one value each so that the expected induced timing effect in one case fulfills a real-time requirement and in the other case violates the same real-time requirement. For this purpose, he reenacts the semantics for the corresponding platform property to conceive a property value so that the desired timing effect for each the real-time requirement fulfillment and violation should occur according to his expectation. In the case of static delays, he reenacts the respective delay computation formula (cf. Section 4.6.1.2). One example for dynamic delays is the construction of a runtime situation in which two software components concurrently access one resource.
2. He conducts the timing analysis in TIMESQUARE. For the static delays, he already determines in the intermediate preprocessed model (cf. Section 4.6.1) whether the corresponding delay changes according to his expectations. For both dynamic and static delays, he simulatively determines whether the corresponding timing effect as well as the real-time requirement fulfillment or violation for the platform property under investigation occurs as expected.

Table 4.1 lists the test results for all platform properties that induce static timing effects as covered by our MSD semantics for timing analyses. It documents how the particular platform property (column Platform Property) induces the respective timing effect (column Induced Timing Effect) and how this timing effect sums up to which kind of static delay (column Static Delay). Furthermore, the table documents which platform-specific MSD specification covers the platform property (column Platform-specific MSD specification) and whether the student observes the static delay as expected (column Test Result).

Table 4.2 lists the test results for all platform properties that induce dynamic timing effects as covered by our MSD semantics for timing analyses. It documents how the particular runtime platform property (column Platform Property at Runtime) induces the respective timing effect (column Induced Timing Effect). Furthermore, the table documents which platform-specific MSD specification covers the platform property (column Platform-specific MSD specification) and whether the student observes the dynamic effect as expected (column Test Result).

Hypothesis H2

In order to evaluate hypothesis H2, ourselves count the model elements of MSD-spec-1, MSD-spec-2, MSD-spec-3, and MSD-spec-4. For this purpose, we use the automatic counting method used in the evaluation of the last chapter (cf. Section 3.9.2.4). The specification process described in Section 4.2 assumes that several roles add platform-specific aspects to an already existing platform-independent MSD specification. Thus, we distinguish between platform-independent and platform-specific MSD specification elements during counting to get an impression how much effort for adding these platform-specific aspects is required. Table 4.3 summarizes the model element amounts (cf. Table B.1 and Table B.2 in Appendix B.4 for the detailed model element amounts).

Afterward, we count the elements of the corresponding generated CCSL models *CCSL-model-1*, *CCSL-model-2*, *CCSL-model-3*, and *CCSL-model-4*. We distinguish between Integer variables, clock variables, clock expressions, and clock relations. The CCSL model editor automatically supports this counting method by displaying the amount of selected elements. Table 4.4 documents the results.

Hypothesis H3

For evaluating hypothesis H3, student-2 performs several times the transformation from the platform-specific MSD specifications MSD-spec-1, MSD-spec-2, and MSD-spec-3 to the CCSL models CCSL-model-1, CCSL-model-2, and CCSL-model-3, respectively. As explained in Section 4.6.1, this transformation consists of a self-developed preprocessing step and the actual MSD-to-CCSL transformation that is automatically derived by GEMOC Studio. For measuring the particular times of both transformation steps, student-2 instruments the particular QVT-O transformations so that timestamps are generated. He measures the preprocessing step six times per model and the MSD-to-CCSL transformation four times per model. These measurements are conducted on a HP EliteDesk 800 G3 TWR with an Intel Core i7-6700 CPU, which runs at a speed of 3.4 GHz. The personal computer has 8GB RAM and a Micron SDD with a capacity of 512GB, and it applies Windows 7 Enterprise (64 Bit) as operating system. Table 4.5 summarizes the averaged execution times (cf. Table B.3 in Appendix B.4 for the detailed measurements).

Hypothesis H4

During the evaluation of H1, H2, and H3, the students open every generated CCSL model in the CCSL model editor and simulate it in TIMESQUARE. These models encompass CCSL-model-1,

Table 4.1: Test results static delays for hypothesis H1

Platform Property	**TAM Subprofile**	**Modeling Means**	**Induced Timing Effect**	**Static Delay**	**Platform-specific MSD Specification**	**Test Result**
Communication transmission overhead	Platform::Communication	TamComInterface::commTxOvhd	ECU -> comm system encoding delay	distr message dispatch delay	MSD-spec-1	✓
Communication reception overhead	Platform::Communication	TamComInterface::commRcvOvhd	comm system -> ECU decoding delay	distr msg consumption delay	MSD-spec-1	✓
Media access policy arbitration time	Platform::Communication	TamComInterface::arbitrationT	arbitration delay at sender	distr message dispatch delay	MSD-spec-1	✓
Transmission protocol encoding at sender	Platform::Communication	TamProtocol::encodeRate	encoding delay	distr message dispatch delay	MSD-spec-1	✓
Transmission protocol decoding at receiver	Platform::Communication	TamProtocol::decodeRate	decoding delay	distr msg consumption delay	MSD-spec-1	✓
Transmission protocol overhead	Platform::Communication	TamProtocol::transmOvhd	distributed throughput reduction	distributed message send delay	MSD-spec-1	✓
Transmission protocol msg ctrl overhead	Platform::Communication	TamProtocol::msgCtrlOvhd	throughput reduction	distributed message send delay	MSD-spec-1	✓
Transmission protocol msg ctrl overhead	Platform::Communication	TamProtocol::msgCtrlOvhd	encoding delay	distr message dispatch delay	MSD-spec-1	✓
Transmission protocol msg ctrl overhead	Platform::Communication	TamProtocol::msgCtrlOvhd	decoding delay	distr msg consumption delay	MSD-spec-1	✓
Middleware encoding at sender	Platform::Communication	TamComService::encodeRate	encoding delay	distr message dispatch delay	dedicated middleware encoding delay model	✓
Middleware decoding at receiver	Platform::Communication	TamComService::decodeRate	decoding delay	distr msg consumption delay	dedicated middleware decoding delay model	✓
Middleware overhead	Platform::Communication	TamComService::transmOvhd	throughput reduction	distributed message send delay	dedicated middleware overhead model	✓
Middleware message control overhead	Platform::Communication	TamComService::msgCtrlOvhd	throughput reduction	distributed message send delay	dedicated middleware msg ctrl ovhd model	✓
Middleware message control overhead	Platform::Communication	TamComService::msgCtrlOvhd	encoding delay	distr message dispatch delay	dedicated middleware msg ctrl ovhd model	✓
Middleware message control overhead	Platform::Communication	TamComService::msgCtrlOvhd	decoding delay	distr msg consumption delay	dedicated middleware msg ctrl ovhd model	✓
Communication medium latency	Platform::Communication	TamComConnection::blockT	propagation delay	distributed message send delay	MSD-spec-1	✓
Communication medium capacity	Platform::Communication	TamComConnection::capacity	distributed gross throughput	distributed message send delay	MSD-spec-1	✓
Network latency	Platform::Communication	TamComConnection::networkBlockT	network delay	distributed message send delay	dedicated network delay model	✓
Network management overhead	Platform::Communication	TamComConnection::networkOvhd	distributed throughput reduction	distributed message send delay	dedicated network management ovhd model	✓
Processing power	Platform::ControlUnit	TamProcessingUnit::speedFactor	processing time	task execution delay	MSD-spec-1	✓
Core synchronization overhead	Platform::ControlUnit	TamProcessingUnit::coreSyncOvhd	core synchronization delay	task execution delay	dedicated core synchronization ovhd model	✓
Memory access latency	Platform::ControlUnit	TamMemoryUnit::accessDelay	memory access delay	task execution delay	dedicated memory delay model	✓
Memory access throughput	Platform::ControlUnit	TamMemoryUnit::throughput	memory access throughput	task execution delay	dedicated memory delay model	✓
Sensor latency	Platform::ControlUnit	TamSensor::delay	PeripheryUnit delay	task execution delay	dedicated periphery delay model	✓
Actuator latency	Platform::ControlUnit	TamActuator::delay	PeripheryUnit delay	task execution delay	dedicated periphery delay model	✓
I/O latency	Platform::ControlUnit	TamIO::delay	PeripheryUnit delay	task execution delay	dedicated periphery delay model	✓
Communication medium latency	Platform::OperatingSystem	TamOSComChannel::blockT	internal propagation delay	internal message send delay	dedicated internal msg send delay model	✓
Task communication delay	Platform::OperatingSystem	TamOSComChannel::commTxOvhd	internal propagation delay overhead	internal message send delay	dedicated internal msg send delay model	✓
Task communication delay	Platform::OperatingSystem	TamOSComChannel::commTxOvhd	internal dispatch delay	internal message dispatch delay	dedicated internal msg send delay model	✓
Task communication delay	Platform::OperatingSystem	TamOSComChannel::commRcvOvhd	internal comsumption delay	internal msg consumption delay	dedicated internal msg send delay model	✓
Task communication capacity	Platform::OperatingSystem	TamOSComChannel::capacity	internal gross throughput	internal message send delay	dedicated internal msg send delay model	✓
Task communication overhead	Platform::OperatingSystem	TamOSComChannel::backgroundUtilization	internal throughput reduction	internal message send delay	dedicated internal msg send delay model	✓
RTOS overhead	Platform::OperatingSystem	TamRTOS::backgroundUtilization	RTOS delay	task execution delay	MSD-spec-1	✓
Resource access time	Platform::OperatingSystem	TamSharedOSResource::accessDelay	access delay	task execution delay	MSD-spec-1	✓
Resource management overhead	Platform::OperatingSystem	TamOSResource::backgroundUtilization	resource management delay	task execution delay	dedicated resource management delay model	✓
Scheduling overhead	Platform::OperatingSystem	TamScheduler::backgroundUtilization	scheduling management delay	task execution delay	dedicated scheduling mgmt delay model	✓
Background utilization	Platform::OperatingSystem	TamOSService::backgroundUtilization	execution delay	task execution delay	MSD-spec-1	✓
Implementation execution overhead	Platform::OperatingSystem	TamImplementation::implExecOvhd	implementation induced execution delay	task execution delay	dedicated impl induced delay model	✓
Implementation memory overhead	Platform::OperatingSystem	TamImplementation::implMemOvhd	implementation induced memory delay	task execution delay	dedicated impl induced delay model	✓
Message size	ApplicationSoftware	TamOperation::msgSize	throughput communication channel	distributed message send delay	MSD-spec-1	✓
Message size	ApplicationSoftware	TamOperation::msgSize	throughput internal ECU communication	internal message send delay	MSD-spec-1	✓
Message size	ApplicationSoftware	TamOperation::msgSize	encoding delay	distr message dispatch delay	MSD-spec-1	✓
Message size	ApplicationSoftware	TamOperation::msgSize	decoding delay	distr msg consumption delay	MSD-spec-1	✓
Execution time	ApplicationSoftware	TamOperation::execTime	processing time	task execution delay	MSD-spec-1	✓
Memory consumption	ApplicationSoftware	TamOperation::usedMemory	memory access throughput reduction	task execution delay	dedicated memory delay model	✓
Amount memory accesses	ApplicationSoftware	TamOperation::memAccessFreq	memory delay	task execution delay	dedicated memory delay model	✓
Amount resource accesses	ApplicationSoftware	TamResourceAccess::numAccess	resource access delay	task execution delay	MSD-spec-1	✓

Table 4.2: Test results dynamic timing effects for hypothesis H1

Platform Property at Runtime	**TAM Subprofile**	**Modeling Means**	**Induced Timing Effect**	**Platform-specific MSD Specification**	**Test Result**
concurrent communication channel access	Platform::Communication	TamComConnection	mutual exclusion of accessing tasks	dedicated communication channel access model	✓
concurrent periphery resource access	Platform::ControlUnit	TamPeripheryUnit, TamResourceAccess::isExclusive, TamResourceAccess::peripheryResource	mutual exclusion of accessing tasks	dedicated periphery resource access model	✓
concurrent OS resource access	Platform::OperatingSystem	TamOSSharedResource, TamResourceAccess::isExclusive, TamResourceAccess::osResource	mutual exclusion of accessing tasks	dedicated OS resource access model	✓
task scheduling on processing unit cores	Platform::OperatingSystem	TamScheduler::isPreemptible, TamScheduler::schedPolicy, TamOperation::schedParameters, TAMProcessingUnit: numCores	task dispatch delay	dedicated task scheduling model	✓

Table 4.3: Summarized model element amounts of the platform-specific MSD specifications for H2 (cf. Table B.1 and Table B.2 for detailed model element amounts)

	# MSD-spec-1	# MSD-spec-2	# MSD-spec-3	# MSD-spec-4
Platform-independent model elements	193	193	883	883
Platform-specific model elements	88	172	88	201
Overall model elements	**281**	**365**	**971**	**1,084**

Table 4.4: Model element amounts of the generated CCSL models for hypothesis H2

	# CCSL-model-1	# CCSL-model-2	# CCSL-model-3	# CCSL-model-4
Integer variables	19	29	31	99
Clock variables	85	114	746	754
Clock expressions	92	171	1,410	1,648
Clock relations	159	241	1,280	1,747
Overall model elements	**355**	**555**	**3,467**	**4,248**

CCSL-model-2, CCSL-model-3, CCSL-model-4, and the 17 further CCSL models dedicated to certain platform properties (cf. Table 4.1 and Table 4.2). The students can open and simulate all models without the occurrence of any error.

4.6.2.5 Interpreting the Results

Both Table 4.1 and Table 4.2 document that our semantics encode the timing effects induced by the platform properties as expected. Furthermore, all modeling means as provided by our TAM profile are considered by the semantics. Thus, we consider our hypothesis H1 fulfilled.

Interpreting the results for hypothesis H2, we observe that the CCSL-model-1 model element amount is ~126% of the MSD-spec-1 model element amount, # CCSL-model-2 is ~152% of # MSD-spec-2, # CCSL-model-3 is ~357% of # MSD-spec-3, and # CCSL-model-4 is ~391% of # MSD-spec-4. These observations indicate that the size of the generated CCSL models is proportionally increasing w.r.t. the size of the input platform-specific MSD specification. This assumption is consistent with the fact that in general for any MSD specification model element each several CCSL model elements are generated, as exemplified in Section 4.4. Summarizing, we consider our hypothesis H2 fulfilled because the sizes of the examined CCSL models are >100% of the sizes of the respective platform-specific MSD specification counterparts.

For interpreting the results for hypothesis H3, we relate the CCSL model sizes determined for hypothesis H2 with the measured averaged execution times. By doing so, we observe that per second ~36 CCSL model elements are generated from MSD-spec-1, ~51 CCSL model elements from MSD-spec-2, ~80 CCSL model elements from MSD-spec-3, and ~80 CCSL model elements from MSD-spec-4. These observations indicate that the relative transformation speed initially increases with the size of the generated CCSL models and stabilizes after a certain

Table 4.5: Averaged transformation execution times for deriving CCSL models from platform-specific MSD specifications for H3 (cf. Table B.3 for individual measurements)

	MSD-spec-1	MSD-spec-2	MSD-spec-3	MSD-spec-4
∅ Preprocessing runs	434 ms	369 ms	4,948 ms	4,769 ms
∅ MSD-to-CCSL Transformation runs	9,427 ms	10,443 ms	38,528 ms	48,523 ms
∅ Overall transformation execution time	**9,860 ms**	**10,812 ms**	**43,476 ms**	**53,292 ms**

CCSL model size is reached. We consider hypothesis H3 fulfilled because more than one CCSL model element is generated per second.

Finally, we consider hypothesis H4 fulfilled because 100% of the 21 CCSL models generated during the evaluation of H1-H3 were opened in the CCSL model editor and simulated in TIMESQUARE without the occurrence of any error.

The fulfilled hypotheses indicate a positive answer to our evaluation questions. That is, our timing analysis approach generates syntactically and semantically CCSL models and reduces the engineering effort for conceiving and specifying them. The effort is even less because the specification process described in Section 4.2 assumes that the platform-specific aspects are added to an already existing platform-independent MSD specification. Summarizing, the fulfilled hypotheses give rise to the assumption that our timing analysis approach is indeed useful for the Timing Analysts.

4.6.2.6 Threats to Validity

The threats to validity in our case study (structured according to the taxonomy of Runeson et al. [RHAR12; RH08]) are as follows.

Construct Validity

- Student-1 conceived the initial timing analysis approach [*Ber17] as well as MSD-spec-1, and the other platform-specific MSD specifications are variants of it. Thus, he knew the functional principle of the approach and could have been biased toward it.

  However, we complementary employed student-2 for conceiving the other platform-specific MSD specifications as well as evaluating the hypotheses, and ourselves had comprehensive discussions with him. Furthermore, the platform-specific MSD specifications base on a platform-independent MSD specification and a CONSENS system model for the EBEAS (cf. last chapter) that we extensively discussed with other both internal and external researchers as well as industry experts (cf. Section 3.9.2.6).

- Regarding the evaluation of H1, the platform properties provided as modeling means by our TAM profile might not be extensive enough, might not be useful, or might not be applicable during the early development phase of SwRE.

  However, we argue that the considered platform properties represent typical timing-relevant ones at an adequate abstraction level due to their systematic determination by means of a literature review [*Ber17, Chapter 3]. This literature review considered scientific as well as industrial-grade publications and investigated which platform properties influence the timing behavior of software-intensive systems and which concrete effects on the timing behavior they induce. Furthermore, it ensured the applicability during SwRE by excluding publications describing platform properties that have a too detailed abstraction level for a timing analysis during this early development phase.

Internal Validity

Regarding the evaluation of H3, we set the amount of the generated CCSL model elements in ratio with the transformation execution time and conclude that the generation of CCSL models is more efficient than a manual specification. This causal relation might be incorrect. However, it is obvious that manually specifying a CCSL model cannot outperform an automatic CCSL model generation.

External Validity

We only considered one case, and the platform-specific MSD specifications are variants of the automotive EBEAS. Furthermore, exemplary case studies in general cannot ensure external validity. Thus, we cannot generalize the conclusions to all possible platform-independent MSD specifications, other types of software-intensive systems, or software-intensive systems in other industry sectors. Nevertheless, the examples are typical for software-intensive systems, and we hence do not expect large deviations for other types of systems.

Reliability

Regarding the evaluation of H1, the students could have judged incorrectly whether the platform-induced timing effects occur as expected. However, we mitigate this threat by employing two students, whose judgments complement each other.

4.7 Related Work

We present early timing analysis approaches based on system models in Section 4.7.1, for scenario-based formalisms in Section 4.7.2, and for component-based architectures in Section 4.7.3.

4.7.1 Timing Analyses based on System Models

The approaches described in the following base on system models to enable simulative performance or timing analyses. They have in common that much information only relevant to the Platform Architects, Allocation Engineers, and Timing Analysts (e.g., B/WCETs, task models, latencies, etc.) has to be specified in the system model to apply the simulations. Thus, the interdisciplinary system model is overloaded with information specific to the particular discipline-specific analysis goals. The approaches hence are applicable in their restricted domains but not in general in multidisciplinary settings. Furthermore, both approaches do not consider functional behavior requirements.

Frieben et al. [Fri17; FHMB13; FH12] present an approach for the early simulative validation of factors influencing the performance of software-intensive systems in the automation sector. The simulation is based on a fine-grained automation model encompassing system properties influencing the timing behavior such as hardware, operating system, and scheduling properties. However, they focus on performance analysis, which applies stochastic approaches. Such approaches are not suitable for hard real-time systems as they cannot guarantee the correct working of a system under worst-case conditions [JP86; Sta88] (cf. Section 2.6). Furthermore, the approach is restricted to the automation sector.

Meyer et al. [Mey15; *MHM11; NMK10] present an approach enabling early real-time simulations to validate system properties influencing the timing behavior and partly to verify them against real-time requirements for software-intensive systems in the automotive sector. The real-time simulation is applied in the system design as well as in the design part of the software engineering phase (cf. Figure 1.1) and requires information from a system and a software design model as input. However, the approach bases on design models and does not focus on requirements.

4.7.2 Scenario-based Timing Analyses

Most approaches described in this section annotate scenario-based requirements formalisms with real-time requirements as well as timing-relevant effects and provide different non-simulative techniques for their respective timing analyses. However, the outputs of these timing analysis techniques are plain yes/no results and partly logged information about the processing times. In contrast, our timing analysis of the generated CCSL time models in TIMESQUARE enables to comprehensively detect real-time requirement violations straightaway by means of interactive simulation.

Furthermore, the approaches only allow to describe the timing effects (typically static delays) induced by the system in a very generic way. However, they do not allow to describe the causes of the effects, that is, a platform property inducing the particular timing effect. In contrast, we provide modeling means based on MARTE that enable to annotate a component-based MSD specification allocated to a platform model with timing-relevant platform properties. We encode the induced timing effects in terms of the simulative execution in TIMESQUARE by means of our semantics for timing analyses. The separation of the platform property specification from the timing effects that they induce enables to use the platform properties also for other purposes (e.g., for design reviews or as requirements for the subsequent design phases).

Hassine [Has15; Has09] annotates the scenario notation of Timed Use Case Maps (TUCM) [HRD06] and its underlying architecture with annotations about timing-relevant effects. The annotated TUCM model is transformed into an Abstract State Machine model [BS03] that is simulated in an external tool, similarly to our approach. However, the simulation tool is not capable of interpreting the annotations. Instead, it is instrumented so that execution traces are generated and persisted in a text file. These execution traces potentially contain log messages about real-time requirement violations and have to be inspected manually to verify platform properties against real-time requirements. In contrast, we generate CCSL specifications that we directly simulate in TIMESQUARE, where we detect and comprehend potential real-time requirement violations straightaway. Furthermore, the approach only allows to specify delays (i.e., an excerpt of our effects) induced by the platform in a very generic way, but not the causes of the effects. In contrast, we provide modeling means based on MARTE that enable annotating a component-based MSD specification allocated to a platform model with timing-relevant platform properties, where the induced timing effects are encoded in our semantics.

Wang and Tsai [WT04] apply Message Sequence Charts [ITU11] to specify functional requirements and the Specification and Description Language [ITU16] to specify the underlying architecture. They annotate these models with a task model including real-time requirements and with timing-relevant effects, respectively. They use algorithms to first compute an allocation of tasks to processing resources and subsequently perform a schedulability analysis to verify the effects against real-time requirements, yielding a plain yes/no result. In contrast, we explicitly specify the allocation of software components to processing resources in our platform-specific MSD specifications and simulate the resulting CCSL specifications, where the simulation facilitates to comprehend potential real-time requirement violations. Again, the approach does not distinguish between platform properties and the timing effects that they induce.

Han and Youn [HY12] apply Interval Timed Colored Petri Nets [Bou08] to specify delays for the execution of event sequences and annotate these models with real-time requirements. They present algorithms for the computation of event sequence processing delays and for the verification of the delays w.r.t. the real-time requirements. Similarly to the approaches mentioned above, the outputs of these algorithms are plain yes/no results as well as the logged processing times. Thus, our simulative approach again enables a better comprehension of real-time requirement violations. Furthermore, the approach only allows to specify static delays in terms of timing effects.

Larsen et al. [LBD$^+$10; LLNP09; LLNP10] present an approach to formally verify real-time design behavior specified through Timed Automata (TA) [AD94] against scenario-based functional and real-time requirements specified by means of time-enriched Live Sequence Charts (LSCs) [DH01]. For this purpose, the LSC requirements are translated to observer TA that are composed with the design behavior TA that encompass timing effects. The resulting TA network is verified against reachability properties on the observer TA in a model checking tool. However, the need for detailed intra-component design models encompassing timing effects impedes the application of the approach in SwRE.

Lettrari and Klose [LK01] simulatively verify real-time design models against scenario-based functional and real-time requirements that are specified by means of time-constrained UML 1.3 Sequence Diagrams augmented with concepts from LSCs. For this purpose, they generate instrumented code from the design models so that timestamps are recorded in the simulative code execution. These timestamps are used to check whether the implementation fulfills the real-time requirements specified in the scenarios. However, the need for executable software code generated from detailed design models impedes the application of the approach in SwRE.

Maoz et al. [MH11; MKH07] present an approach to visualize and inspect traces of system executions in correspondence with their functional requirements specified by means of Maoz' and Harel's original MSD language [HM08]. Amongst other things, they provide a time-based view to visualize the concrete timestamps of the traces to address timing aspects. However, the approach does not aim at verification, and the MSDs do not specify real-time requirements.

4.7.3 Architecture-based Timing Analyses

The approaches described in the following enable implementation-level timing analyses w.r.t. real-time requirements specified at a higher abstraction level based solely on component-based architectures, or reuse architecture-based knowledge from prior projects. However, the approaches do not consider functional requirements and require conducting timing analyses on the final system implementation like typical timing analysis tools (i.e., based on executed platform code or models of it). In contrast, we provide early timing analyses based on functional requirements on the implementation.

Mubeen et al. [MNS$^+$17] present an approach to annotate real-time requirements to component-based architectures (both specified on a coarse-grained, high abstraction level) and to automatically refine the abstract real-time requirements into concrete real-time requirements for a low-level implementation model (that is also automatically derived from the high-level architecture). However, their actual timing analysis still requires the final system implementation.

Another line of research of Mubeen et al. [MNL$^+$16; MSN$^+$15] focuses on enabling early timing analyses based on legacy systems. To this end, they provide differently precise timing analysis techniques for different kinds of components: Precise timing analysis for white-box components whose internal architectures were developed in prior projects, less precise timing analysis for gray-box components whose internals have to be adapted, and imprecise timing analysis for black-box components whose internals are unknown and have to be newly conceived.

Noyer et al. [NIE$^+$17] provide a tool chain that enables to specify informal real-time requirements, to specify a UML/MARTE design, to generate source code, to extract traces from the executed implementation, and to conduct timing analyses in a commercial-off-the-shelf tool. They focus on the stepwise refinement and on the traceable data exchange from the informal requirements down to the actual timing analysis, but their timing analysis requires the final system implementation like typical timing analysis tools.

4.8 Summary

In this chapter, we presented an approach that enables end-to-end response time analyses based on MSD specifications encompassing real-time requirements. For this purpose, we introduced the MARTE-based TAM profile providing modeling means for platforms, their timing-relevant properties, and the allocation of component-based MSD specifications to the platform models. This profile enables the participating engineering roles to model platform-specific MSD specifications. Furthermore, we conceptually extended the event handling semantics of MSDs by introducing additional event kinds for the consideration of static and dynamic delays in between that occur during the software execution on a target platform. As major contribution, we specified the semantics of platform-specific MSD specifications with extended event handling for the purpose of conducting timing analyses. This semantics encompasses a subset of the conventional MSD semantics extended by the additional event types, the platform properties' effects on the timing behavior in terms of the static and dynamic delays, and the encoding of the MSD real-time requirements as well as timing analysis contexts. To this end, we applied the GEMOC approach that enables the automatic derivation of CCSL models from platform-specific MSD specifications based on our semantics specification. These CCSL models are executable in the simulative timing analysis tool TIMESQUARE. We provided a process description that clarifies the artifact dependencies and role responsibilities for the specification of the particular platform-independent and -specific aspects and the application of the approach. Using a case study, we evaluated the transition technique by means of the automotive EBEAS example and outlined the timing problems that we are able to identify on this abstraction level.

The timing analysis approach provides Software Requirement Engineers and Timing Analysts means for the platform-aware validation of the platform-independent MSD analysis results during the early phase of SwRE. The TAM profile provides comprehensive modeling means to add timing-relevant platform-specific aspects to MSD specifications at an abstraction level suitable for SwRE. The extended event handling semantics for MSDs enables a more realistic consideration of the particular event occurrences during the observation of the message-based coordination behavior of the system under development. The specification of the MSD semantics dedicated to timing analyses encodes a subset of the conventional MSD semantics, the extended MSD event handling, and the platform properties' effects on the timing behavior in terms of CCSL. Furthermore, the declarative semantics specification with GEMOC allows the flexible encoding of additional platform properties' timing effects or the adaption to other scenario-based formalisms. The model transformation generation feature of GEMOC Studio takes this declarative specification as input and thereby reduces the effort of moving from MSDs to the CCSL formalism. The conducted case study indicates the effectiveness and the efficiency of the approach.

5 Conclusion

Section 5.1 summarizes this thesis, and Section 5.2 outlines future work.

5.1 Summary

The software part of software-intensive systems has become their key innovation force [GDS^{+}15] and consequently provides more and more functionality. Thus, the software part of these systems is increasing both in size [VBK10; MF10; PBKS07] and market value [BBH^{+}10]. Requirements are the basis for all engineering tasks in the further development process and hence must have a high quality, particularly for the increasingly complex software. The contributions of this thesis enable Software Requirements Engineers to improve the quality of their requirements specifications for software-intensive systems. Our contributions focus on the SwRE approach applying the scenario-based modeling language MSDs [*HFK^{+}16]. However, the results are transferable to other scenario-based SwRE approaches. We implemented all contributions on top of the tool suite SCENARIOTOOLS MSD [ST-MSD].

As our first contribution, we provide a technique for the semi-automatic and systematic transition from MBSE with CONSENS to SwRE with MSDs. This transition technique reduces the likelihood of defects in MSD specifications that can be introduced by the Software Requirements Engineers during the transition. Furthermore, it reduces the manual effort for conceiving MSD specifications based on CONSENS system models. Particularly, the transition technique applies incremental model transformations to automatically derive initial and update existing MSD specifications based on CONSENS system models. Furthermore, these model transformations semi-automatically establish lifecycle traceability between CONSENS system models and the MSD specifications. The model transformations are complemented by a semi-automatic part, which supports the Software Requirements Engineers in the manual refinement of the initially derived MSD specifications. Using a case study, we evaluate that our transition technique provides effective and efficient support for the Software Requirements Engineers by means of two cases from the automotive sector.

As our second contribution, we provide an approach that enables end-to-end response time analyses based on MSD specifications. This timing analysis approach identifies platform-induced real-time requirement violations in MSD specifications that could otherwise be revealed only in late engineering phases through conventional timing analysis techniques. Thereby, we provide Timing Analysts means for the platform-aware validation of the platform-independent MSD analysis results during the early phase of SwRE. Particularly, we present an MSD semantics dedicated to timing analyses, which we specify declaratively by means of the GEMOC approach. This approach automatically generates simulative timing analysis models from MSD specifications with timing-relevant platform properties based on the semantics specification. As a basis for the semantics, we introduce a MARTE-based profile providing modeling means for adding the timing-relevant platform properties to MSD specifications. Furthermore, we extend the MSD message event handling semantics by introducing additional event kinds and the timing effect delays in between for the consideration in the semantics and thereby in the timing analyses. Using a case study, we evaluate that our timing analysis approach provides effective and efficient support for the Timing Analysts by means of an example from the automotive sector.

In combination, our contributions improve the quality of software requirements specifications for software-intensive real-time systems that are developed in a multidisciplinary manner and deployed to distributed and concurrent execution platforms. We achieve this quality improvement by reducing the likelihood to introduce defects during the transition from the interdisciplinary MBSE and by early revealing platform-induced real-time requirement violations.

5.2 Future Work

Consolidation with other CONSENS and MECHATRONICUML Integrations Our thesis integrates the CONSENS specification technique and the MECHATRONICUML SwRE method. However, a variety of other approaches exist that integrate aspects of CONSENS and MECHATRONICUML. First, Anacker et al. [AGD+12] present an approach for the transition from MBSE with CONSENS to SwRE with MSDs in the context of brownfield development. Second, Rieke [Rie15] and Heinzemann et al. [HSST13] present approaches for the transition from MBSE with CONSENS to the software design with MECHATRONICUML. Finally, Bröggelwirth et al. [*BBG+13] and Basak et al. [BBB+16] present early results for the transition from the SwRE method to the software design method of MECHATRONICUML. These integration approaches should be consolidated with our transition technique in order to combine them to a seamless and holistic method for the design of software-intensive systems.

Discipline-spanning, Tool-supported Requirements Validation Our thesis focuses on the documentation and the tool-supported validation of the requirements on the coordination behavior part of software-intensive systems. However, the coordination behavior engages control behavior, which actuates electronic elements that trigger physical actions. Thus, the MSD analysis techniques should be combined with tool-supported requirements validation approaches from other disciplines (e.g., [HLMR13; GKS00] for formal languages for the specification of requirements on the control behavior). Such a combination would enable tool support that allows to analyze the interactions between the disciplines on requirements level.

Further Support for Conceiving MSD Specifications Our thesis provides semi-automatic means to support the manual refinement of initially derived MSD specifications based and informal or semi-formal information in CONSENS system models. However, we partially contributed to other complementary approaches that can be utilized to constructively assure a high quality of MSD specifications or to interpret the informal system model information. First, we present initial results of an approach that uses evolutionary algorithms to complete missing parts of underspecified MSD specifications [*SGH17]. Second, we provide MSD requirement patterns [*FHKS18; *FHKS17] that provide reusable and established building blocks to assemble high-quality MSD specifications. Third, the informal textual information in system models can be formalized trough controlled natural language (e.g., [WAB$^+$10]) so that this information is interdisciplinarily understandable and automatically processable at the same time, as we showed in [*FH15; *FHM14; *FH14; *DFHT13; *FHH$^+$12; *FHM12; *HMD11; *HMM11; *Hol10]. Finally, natural language processing can extract a representation amenable to automatisms from the informal textual information in system models. Combining such approaches would further support the Software Requirements Engineers to conceive or refine MSD specifications by means of automatisms.

Allocation Engineering across Different Engineering Phases Planning, specifying, and analyzing allocations of software components to execution platforms is part of the allocation engineering, which is regarded as a part of the software engineering discipline (e.g., [ŠCV13; Poh18]. However, allocation engineering is obviously interdisciplinary due to its hardware dependencies and hence should be already considered in the system design phase. Thus, our CONSENS system models contain coarse-grained allocations of software components to execution hardware elements, and our platform-specific MSD specifications contain more fine-grained allocations of software architectures to execution platforms as basis for our timing analysis approach during SwRE. Furthermore, recent improvements of the MECHATRONICUML design method enable a constructive schedulability assurance through computing feasible allocations based on even more fine-grained software design and platform models [GHK$^+$18; Poh18; PH18; PH15] in a later software engineering phase. Future work should thoroughly define at which abstraction levels allocations shall be specified in system models and software/platform models w.r.t. to the particular modeling purposes [Sta73] and how the transitions and refinements across the different engineering phases shall look like (see also [JVTM17] and [FMS12, Section 14.4.4]).

Coupled Platform-independent and -specific Requirements Validation The MSD analysis techniques address safety and liveness properties of functional and real-time platform-independent requirements. For this purpose, they apply a dense time model and abstract event handling semantics. TIMESQUARE as instrumented by our semantics addresses the verification of platform-induced timing effects w.r.t. the real-time requirements, applying a discrete logical time model. For this purpose, our approach considers a subset of the complete MSD semantics regarding functional requirements but refines this subset for platform-specific timing aspects with more fine-grained event handling semantics. Thus, both approaches complement each other and induce a process, in which initially timed MSD specifications are analyzed for general realizability and are subsequently enriched with platform properties that are verified w.r.t. the real-time requirements. The coupling of both approaches could enable a synchronized simulation in both Real-time Play-out and TIMESQUARE, directly tracing back real-time requirement violations detected in TIMESQUARE to MSD specifications. Furthermore, the TIMESQUARE simulation could be guided by timed controllers synthesized from the MSD specification, so that only the realizable state space paths are considered in the timing analyses.

Generalizing the MBSE to SwRE Model Transformation Approach Our model transformation approach for the transition from MBSE to SwRE combines the efficient and compact imperative logic of QVT-O with the user-edit-preserving target-incrementality and the destructive source-target relationship known from TGGs. However, we realized the TGG features by means of hard-coded rules for the particular source and target model element types as part of our model transformation algorithm. Future work should combine the advantages of operational and relational model transformation approaches in a more general way. This can be achieved through generalizing the hard-coded rules to parameterizable features of QVT-O or through identifying/conceiving a hybrid (i.e., mixed operational and relational) transformation approach [CH06; KBC^{+}18] that fulfills all of our requirements.

Bibliography

The publication keys of all literature that I contributed to (i.e., my publications/theses as well as the theses supervised by me) have the prefix * to identify them easily within this thesis. From these, I do not only list literature that was directly conducive to my PhD thesis but all literature that I contributed to during my time as PhD student.

Own Peer-reviewed Publications

[*BGH+14] CHRISTIAN BRENNER; JOEL GREENYER; JÖRG HOLTMANN; GRISCHA LIEBEL; GERALD STIEGLBAUER; MATTHIAS TICHY: "ScenarioTools Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study". In: *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014).* Vol. 67. Electronic Communications of the EASST. European Association for the Study of Science and Technology (EASST), 2014. DOI: 10.14279/tuj.eceasst.67.948.

[*FH14] MARKUS FOCKEL; JÖRG HOLTMANN: "A Requirements Engineering Methodology Combining Models and Controlled Natural Language". In: *Proceedings of the 4th International Model-Driven Requirements Engineering Workshop (MoDRE 2014).* Ed. by ANA MOREIRA; PABLO SÁNCHEZ; GUNTER MUSSBACHER; JOÃO ARAÚJO. Piscataway, USA: IEEE, 2014, pp. 67–76. ISBN: 978-1-4799-6343-0. DOI: 10.1109/MoDRE.2014.6890827.

[*FH15] MARKUS FOCKEL; JÖRG HOLTMANN: "ReqPat: Efficient Documentation of High-quality Requirements using Controlled Natural Language." In: *Proceedings of the 23rd IEEE International Requirements Engineering Conference (RE), Posters and Tool Demos Track.* Los Alamitos, USA: IEEE, 2015, pp. 280–281. ISBN: 978-1-4673-6905-3. DOI: 10.1109/RE.2015.7320438.

[*FHKS18] MARKUS FOCKEL; JÖRG HOLTMANN; THORSTEN KOCH; DAVID SCHMELTER: "Formal, Model- and Scenario-based Requirement Patterns". In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018).* Setúbal, Portugal: SCITEPRESS, 2018, pp. 311–318. ISBN: 978-989-758-283-7. DOI: 10.5220/0006554103110318.

[*FHM12] MARKUS FOCKEL; JÖRG HOLTMANN; JAN MEYER: "Semi-automatic Establishment and Maintenance of Valid Traceability in Automotive Development Processes". In: *Proceedings of the 2nd International Workshop on Software Engineering for Embedded Systems (SEES).* Piscataway, USA: IEEE, 2012, pp. 37–43. ISBN: 978-1-4673-1852-5. DOI: 10.1109/SEES.2012.6225489.

[*GTH16] Matthias Greinert; Christian Tschirner; Jörg Holtmann: "Anwendung von Methoden der Produktentstehung auf Basis des Systemmodells mechatronischer Systeme". In: *Tag des Systems Engineering (TdSE 2016)*. Ed. by Sven-Olaf Schulze; Christian Tschirner; Rüdiger Kaffenberger; Sascha Ackva. München, Germany: Hanser, 2016, pp. 77–86. isbn: 978-3-446-45126-1 (Print), 978-3-446-45141-4 (Online). doi: 10.3139/9783446451414.008.

[*HBM$^+$15] Jörg Holtmann; Ruslan Bernijazov; Matthias Meyer; David Schmelter; Christian Tschirner: "Integrated Systems Engineering and Software Requirements Engineering for Technical Systems". In: *Proceedings of the 2015 International Conference on Software and System Process (ICSSP)*. Best Full Paper. New York, USA: ACM, 2015, pp. 57–66. isbn: 978-1-4503-3346-7. doi: 10.1145/2785592.2785597.

[*HBM$^+$16] Jörg Holtmann; Ruslan Bernijazov; Matthias Meyer; David Schmelter; Christian Tschirner: "Integrated and iterative systems engineering and software requirements engineering for technical systems". In: *Journal of Software Evolution and Process (Special Issue on Best Papers of International Conference on Software and Systems Process 2015)* 28:9 (2016). Ed. by Dietmar Pfahl; Marco Kuhrmann; Reda Bendraou; Richard Turner, pp. 722–743. issn: 2047-7481. doi: 10.1002/smr.1780.

[*HM13] Jörg Holtmann; Matthias Meyer: "Play-out for Hierarchical Component Architectures". In: *Proceedings of the 11th Workshop on Automotive Software Engineering*. Ed. by Matthias Horbach. Vol. P-220. GI-Edition – Lecture Notes in Informatics (LNI). Bonn, Germany: Köllen, 2013, pp. 2458–2472. isbn: 978-3-88579-614-5.

[*HMD11] Jörg Holtmann; Jan Meyer; Markus von Detten: "Automatic Validation and Correction of Formalized, Textual Requirements". In: *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Los Alamitos, USA: IEEE, 2011, pp. 486–495. isbn: 978-1-4577-0019-4. doi: 10.1109/ICSTW.2011.17.

[*HMM11] Jörg Holtmann; Jan Meyer; Matthias Meyer: "A Seamless Model-Based Development Process for Automotive Systems". In: *ENVISION 2020 – Zweiter Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme*. Ed. by Ralf Reussner; Alexander Pretschner; Stefan Jähnichen. Vol. P-184. GI-Edition – Lecture Notes in Informatics (LNI). Bonn, Germany: Köllen, 2011, pp. 79–88. isbn: 978-3-88579-278-9.

[*HMSN10] Jörg Holtmann; Jan Meyer; Wilhelm Schäfer; Ulrich Nickel: "Eine erweiterte Systemmodellierung zur Entwicklung von softwareintensiven Anwendungen in der Automobilindustrie". In: *ENVISION 2020 – Erster Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme*. Ed. by Gregor Engels; Markus Luckey; Alexander Pretschner; Ralf Reussner. Vol. P-160. GI-Edition – Lecture Notes in Informatics (LNI). Bonn, Germany: Köllen, 2010, pp. 149–158. isbn: 978-3-88579-254-3.

[*HS14] JÖRG HOLTMANN; DIMITAR SHIPCHANOV: "Considering Architectural Properties in Real-time Play-out". In: *Proceedings of the 12th Workshop on Automotive Software Engineering*. Ed. by ERHARD PLÖDEREDER; LARS GRUNSKE; ERIC SCHNEIDER; DOMINIK ULL. Vol. P-232. GI-Edition – Lecture Notes in Informatics (LNI). Bonn, Germany: Köllen, 2014, pp. 2169–2180. ISBN: 978-3-88579-626-8.

[*HT08] JÖRG HOLTMANN; MATTHIAS TICHY: "Component Story Diagrams in Fujaba4Eclipse". In: *Proceedings of the 6th International Fujaba Days*. Ed. by UWE ASSMANN; JENDRIK JOHANNES; ALBERT ZÜNDORF. 2008, pp. 44–47.

[*KDHM13] LYDIA KAISER; ROMAN DUMITRESCU; JÖRG HOLTMANN; MATTHIAS MEYER: "Automatic Verification of Modeling Rules in Systems Engineering for Mechatronic Systems". In: *Proceedings of the ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (ASME IDETC/CIE)*. New York, USA: American Society of Mechanical Engineers (ASME), 2013. ISBN: 978-0-7918-5586-7. DOI: 10.1115/DETC2013-12330.

[*KHD14] THORSTEN KOCH; JÖRG HOLTMANN; JULIEN DEANTONI: "Generating EAST-ADL Event Chains from Scenario-Based Requirements Specifications". In: *Proceedings of the 8th European Conference on Software Architecture (ECSA)*. Ed. by PARIS AVGERIOU; UWE ZDUN. Vol. 8627. Lecture Notes in Computer Science (LNCS). Cham, Switzerland: Springer, 2014, pp. 146–153. ISBN: 978-3-319-09969-9 (Print), 978-3-319-09970-5 (Online). DOI: 10.1007/978-3-319-09970-5_14.

[*KHL17] THORSTEN KOCH; JÖRG HOLTMANN; TIMO LINDEMANN: "Flexible Specification of STEP Application Protocol Extensions and Automatic Derivation of Tool Capabilities". In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*. Ed. by LUÍS FERREIRA PIRES; SLIMANE HAMMOUDI; BRAN SELIC. Setúbal, Portugal: SCITEPRESS, 2017, pp. 53–64. ISBN: 978-989-758-210-3. DOI: 10.5220/0006137400530064.

[*KHSL16] THORSTEN KOCH; JÖRG HOLTMANN; DAVID SCHUBERT; TIMO LINDEMANN: "Towards Feature-based Product Line Engineering of Technical Systems". In: *Proceedings of the 3rd International Conference on System-Integrated Intelligence – New Challenges for Product and Production Engineering*. Vol. 26. Procedia Technology. Elsevier, 2016, pp. 447–454. DOI: 10.1016/j.protcy.2016.08.057.

[*MFH15] JAN MEYER; MARKUS FOCKEL; JÖRG HOLTMANN: "Systementwurf unter Einbeziehung funktionaler Sicherheit bei automobilen Steuergeräten". In: *Tag des Systems Engineering (TdSE 2015)*. Ed. by SVEN-OLAF SCHULZE; CHRISTIAN MUGGEO. München, Germany: Hanser, 2015, pp. 365–374. ISBN: 978-3-446-44729-5 (Print), 978-3-446-44728-8 (Online). DOI: 10.3139/9783446447288.036.

[*MH11] JAN MEYER; JÖRG HOLTMANN: "Eine durchgängige Entwicklungsmethode von der Systemarchitektur bis zur Softwarearchitektur mit AUTOSAR". In: *Tagungsband des 7. Dagstuhl-Workshops Modellbasierte Entwicklung eingebetteter Systeme (MBEES VII)*. Ed. by HOLGER GIESE; MICHAELA HUHN; JAN PHILIPPS; BERNHARD SCHÄTZ. München, Germany: fortiss, 2011, pp. 21–30.

[*MHKM15] JAN MEYER; JÖRG HOLTMANN; THORSTEN KOCH; MATTHIAS MEYER: "Generierung von AUTOSAR-Modellen aus UML-Spezifikationen". In: *10. Paderborner Workshop Entwurf mechatronischer Systeme*. Ed. by JÜRGEN GAUSEMEIER; ROMAN DUMITRESCU; FRANZ-JOSEF RAMMIG; WILHELM SCHÄFER; ANSGAR TRÄCHTLER. Vol. 343. HNI-Verlagsschriftenreihe. Paderborn, Germany: Heinz Nixdorf Institut, 2015, pp. 159–172. ISBN: 978-3-942647-62-5.

[*MHM11] JAN MEYER; JÖRG HOLTMANN; MATTHIAS MEYER: "Formalisierung von Anforderungen und Betriebssystemeigenschaften zur frühzeitigen Simulation von eingebetteten, automobilen Systemen". In: *8. Paderborner Workshop Entwurf mechatronischer Systeme*. Ed. by JÜRGEN GAUSEMEIER; FRANZ-JOSEF RAMMIG; WILHELM SCHÄFER; ANSGAR TRÄCHTLER. Vol. 294. HNI-Verlagsschriftenreihe. Paderborn, Germany: Heinz Nixdorf Institut, 2011, pp. 203–215. ISBN: 978-3-942647-13-7.

[*PHMG14] UWE POHLMANN; JÖRG HOLTMANN; MATTHIAS MEYER; CHRISTOPHER GERKING: "Generating Modelica Models from Software Specifications for the Simulation of Cyber-physical Systems". In: *Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2014, pp. 191–198. DOI: `10.1109/SEAA.2014.18`.

[*SGH17] DAVID SCHMELTER; JOEL GREENYER; JÖRG HOLTMANN: "Toward Learning Realizable Scenario-Based, Formal Requirements Specifications". In: *Proceedings of the 4th International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*. 2017, pp. 372–378. DOI: `10.1109/REW.2017.14`.

[*SNH⁺10] HELLA SEEBACH; FLORIAN NAFZ; JÖRG HOLTMANN; JAN MEYER; MATTHIAS TICHY; WOLFGANG REIF; WILHELM SCHÄFER: "Designing Self-healing in Automotive Systems". In: *Proceedings of the 7th International Conference on Autonomic and Trusted Computing (ATC 2010)*. Ed. by BING XIE; JUERGEN BRANKE; S. MASOUD SADJADI; DAQING ZHANG; XINGSHE ZHOU. Vol. 6407. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 2010, pp. 47–61. ISBN: 978-3-642-16575-7 (Print), 978-3-642-16576-4 (Online). DOI: `10.1007/978-3-642-16576-4_4`.

[*THHO08] MATTHIAS TICHY; STEFAN HENKLER; JÖRG HOLTMANN; SIMON OBERTHÜR: "Component Story Diagrams – A Transformation Language for Component Structures in Mechatronic Systems". In: *Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER4)*. Ed. by MATTHIAS GEHRKE; HOLGER GIESE; JOACHIM STROOP. Vol. 236. HNI-Verlagsschriftenreihe. Paderborn, Germany: Heinz Nixdorf Institut, 2008, pp. 27–38. ISBN: 978-3-939350-55-2.

Own Non-peer-reviewed Publications

[*DFHT13] MARIAN DAUN; MARKUS FOCKEL; JÖRG HOLTMANN; BASTIAN TENBERGEN: *Goal-Scenario-Oriented Requirements Engineering for Functional Decomposition with Bidirectional Transformation to Controlled Natural Language – Case Study "Body Control Module"*. ICB Research Report no. 55. University of Duisburg-Essen, 2013.

[*FHH$^+$12] MARKUS FOCKEL; PETER HEIDL; JÖRG HOLTMANN; WILFRIED HORN; JENS HÖFFLINGER; HARALD HÖNNINGER; JAN MEYER; MATTHIAS MEYER; JÖRG SCHÄUFFELE: "Application and Evaluation in the Automotive Domain". In: *Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology*. Ed. by KLAUS POHL; HARALD HÖNNINGER; REINHOLD E. ACHATZ; MANFRED BROY. Berlin/Heidelberg, Germany: Springer, 2012. Chap. 12, pp. 157–175. ISBN: 978-3-642-34613-2 (Print), 978-3-642-34614-9 (Online). DOI: `10.1007/978-3-642-34614-9_12`.

[*FHKS17] MARKUS FOCKEL; JÖRG HOLTMANN; THORSTEN KOCH; DAVID SCHMELTER: *Model-based Requirement Pattern Catalog*. Tech. rep. tr-ri-17-354. Version 1.0. Paderborn, Germany: Software Engineering Department, Fraunhofer IEM, 2017.

[*FHM14] MARKUS FOCKEL; JÖRG HOLTMANN; MATTHIAS MEYER: "Mit Satzmustern hochwertige Anforderungsdokumente effizient erstellen". In: *OBJEKTspektrum* RE/2014 (Online Themenspecial Requirements Engineering) (2014).

[*HBM$^+$17] JÖRG HOLTMANN; RUSLAN BERNIJAZOV; MATTHIAS MEYER; DAVID SCHMELTER; CHRISTIAN TSCHIRNER: "Integrated and Iterative Systems Engineering and Software Requirements Engineering for Technical Systems (Précis)". In: *Software Engineering 2017*. Ed. by JAN JÜRJENS; KURT SCHNEIDER. Vol. P-267. GI-Edition – Lecture Notes in Informatics (LNI). Bonn, Germany: Köllen, 2017, pp. 109–110. ISBN: 978-3-88579-661-9.

[*HFK$^+$16] JÖRG HOLTMANN; MARKUS FOCKEL; THORSTEN KOCH; DAVID SCHMELTER; CHRISTIAN BRENNER; RUSLAN BERNIJAZOV; MARCEL SANDER: *The MechatronicUML Requirements Engineering Method – Process and Language*. Tech. rep. tr-ri-16-352. Version 1.0. Paderborn, Germany: Software Engineering Department, Fraunhofer IEM and Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, 2016. DOI: `10.13140/RG.2.2.33223.29606`.

[*HFKS16] JÖRG HOLTMANN; MARKUS FOCKEL; THORSTEN KOCH; DAVID SCHMELTER: "Requirements Engineering – Zusatzaufgabe oder Kernkompetenz?" In: *OBJEKTspektrum* RE/2016 (Online Themenspecial Requirements Engineering) (2016).

[*Hol10] JÖRG HOLTMANN: "Mit Satzmustern von textuellen Anforderungen zu Modellen". In: *OBJEKTspektrum* RE/2010 (Online Themenspecial Requirements Engineering) (2010).

[*KHL18] THORSTEN KOCH; JÖRG HOLTMANN; TIMO LINDEMANN: "Model-Driven STEP Application Protocol Extensions Combined with Feature Modeling Considering Geometrical Information". In: *Revised Selected Papers of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*. Ed. by LUÍS FERREIRA PIRES; SLIMANE HAMMOUDI; BRAN SELIC. Vol. 880. Communications in Computer and Information Science. Cham: Springer, 2018, pp. 173–197. ISBN: 978-3-319-94763-1 (Print), 978-3-319-94764-8 (Online). DOI: 10.1007/978-3-319-94764-8_8.

[*MHNM10] JAN MEYER; JÖRG HOLTMANN; ULRICH NICKEL; MATTHIAS MEYER: *Beschreibung der Fallstudie "Komfortsteuergerät"*. SPES 2020: Project-internal Case Study Description. Version 1.1. Lippstadt/Paderborn, Germany: Hella KGaA Hueck & Co. and Paderborn University, 2010.

[*PHM14] CLAUDIA PRIESTERJAHN; JÖRG HOLTMANN; MATTHIAS MEYER: "Smarte Entwicklung für smarte Systeme – Softwareentwicklung im Kontext des Gesamtsystems". In: *Tagungsband Embedded Software Engineering Kongress 2014*. 2014, pp. 619–627. ISBN: 978-3-8343-2409-2.

[*PHM16] UWE POHLMANN; JÖRG HOLTMANN; MATTHIAS MEYER: "Das Erwachen der Macht – automatische Softwareverteilung". In: *Tagungsband Embedded Software Engineering Kongress 2016*. 2016, pp. 587–592. ISBN: 978-3-8343-2504-4.

Supervised and Own Theses

[*BBG+13] JANA BRÖGGELWIRTH; CHRISTOPHER BRUNE; FAEZEH GHASSEMI; VINAY AKKASETTY GOPAL; ARGYRIS KOLLIAS; SIJIA LI; SVEN MERSCHJOHANN; SIMON SCHWICHTENBERG; DIMITAR SHIPCHANOV: "Project Group SafeBots III". Co-supervised by STEFAN DZIWOK and OLIVER SUDMANN. Final Documentation. Paderborn, Germany: Paderborn University, 2013.

[*Ber15] RUSLAN BERNIJAZOV: "Systems and Software Requirements Engineering for Cyber-Physical Systems". Bachelor's Thesis. Paderborn, Germany: Paderborn University, 2015.

[*Ber17] RUSLAN BERNIJAZOV: "Early Timing Analysis of Scenario-based Software Requirements". Master's Thesis. Paderborn, Germany: Paderborn University, 2017.

[*Edl12] FABIAN EDLING: "Debugging von simulierter AUTOSAR-Software auf Modellebene". Co-supervised by MARKUS FOCKEL. Master's Thesis. Paderborn, Germany: Paderborn University, 2012.

[*Gre15] MATTHIAS GREINERT: "Vorgehen zur Unterstützung von Managementaufgaben mit dem Systemmodell mechatronischer Systeme". Co-supervised by CHRISTIAN TSCHIRNER. Master's Thesis. Paderborn, Germany: Paderborn University, 2015.

[*Hol08] JÖRG HOLTMANN: "Graphtransformationen für komponentenbasierte Softwarearchitekturen". Diploma Thesis. Paderborn, Germany: Paderborn University, 2008.

[*Jap15] SERGEJ JAPS: “Synthese globaler Controller aus szenariobasierten Spezifikationen unter Berücksichtigung von Echtzeitanforderungen”. Bachelor’s Thesis. Paderborn, Germany: Paderborn University, 2015.

[*Koc13] THORSTEN KOCH: “Combining Scenario-based and Architecture-based Timing Requirements”. Master’s Thesis. Paderborn, Germany: Paderborn University, 2013.

[*Mer15] SIMON MERS: “Dedizierte Werkzeugunterstützung für Anwendungsfälle des Model-Based Systems Engineering”. Co-supervised by CHRISTIAN BREMER. Master’s Thesis. Paderborn, Germany: Paderborn University, 2015.

[*Sch13] DANIEL SCHOLZ: “Refinement of Requirement Specifications for Automotive Systems”. Master’s Thesis. Paderborn, Germany: Paderborn University, 2013.

[*Shi14] DIMITAR SHIPCHANOV: “Considering Message Delays in Timed Play-Out”. Master’s Thesis. Paderborn, Germany: Paderborn University, 2014.

[*Tee12] ALEXANDER TEETZ: “Werkzeuggestützter Übergang von der plattformunabhängigen zur plattformabhängigen Modellebene für eingebettete Systeme im Automobilbereich”. Master’s Thesis. Paderborn, Germany: Paderborn University, 2012.

[*Wör14] MAX WÖRDEHOFF: “Spezifikation der Sprache CONSENS”. Co-supervised by CHRISTIAN BREMER. Master’s Thesis. Paderborn, Germany: Paderborn University, 2014.

Preliminary Work

[AGD+12] HARALD ANACKER; JÜRGEN GAUSEMEIER; ROMAN DUMITRESCU; STEFAN DZIWOK; WILHELM SCHÄFER: “Solution Patterns of Software Engineering for the System Design of Advanced Mechatronic Systems”. In: *MECATRONICS REM*. Piscataway, USA: IEEE, 2012, pp. 101–108. ISBN: 978-1-4673-4771-6. DOI: `10.1109/MECATRONICS.2012.6450994`.

[Ana15] HARALD ANACKER: “Instrumentarium für einen lösungsmusterbasierten Entwurf fortgeschrittener mechatronischer Systeme”. PhD thesis. Paderborn, Germany: Paderborn University, 2015.

[BBB+16] AINDRILA BASAK; RUSLAN BERNIJAZOV; PAUL BÖRDING; HENDRIK EIKERLING; PATRICK ERNSTE; ANDREAS FLOHRE; CONRAD NEUMANN; FLORIAN STOLTE: “Project Group Aramid”. Final Documentation. Paderborn, Germany: Paderborn University, 2016.

[BGP13] CHRISTIAN BRENNER; JOEL GREENYER; PANZICA LA MANNA, VALERIO: “The ScenarioTools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions”. In: *Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*. Vol. 58. Electronic Communications of the EASST. European Association for the Study of Science and Technology (EASST), 2013. DOI: `10.14279/tuj.eceasst.58.856`.

[Bud18] INGO BUDDE: "Verfeinerung deklarativer Mappingmodelle durch imperative Modelltransformationen". Bachelor's Thesis. Paderborn, Germany: Paderborn University, 2018.

[DDGI14] RAFAŁ DOROCIAK; ROMAN DUMITRESCU; JÜRGEN GAUSEMEIER; PETER IWANEK: "Specification Technique CONSENS for the Description of Self-Optimizing Systems". In: *Design Methodology for Intelligent Technical Systems – Develop Intelligent Technical Systems of the Future*. Ed. by JÜRGEN GAUSEMEIER; FRANZ-JOSEF RAMMIG; WILHELM SCHÄFER. Lecture Notes in Mechanical Engineering. Berlin/Heidelberg, Germany: Springer, 2014. Chap. 4.1, pp. 119–127. ISBN: 978-3-642-45434-9 (Print), 978-3-642-45435-6 (Online).

[EHHS00] GREGOR ENGELS; JAN HENDRIK HAUSMANN; REIKO HECKEL; STEFAN SAUER: "Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML". In: *Proceedings of the 3rd International Conference on the Unified Modeling Language (≪UML≫ 2000—The Unified Modeling Language: Advancing the Standard)*. Ed. by ANDY EVANS; STUART KENT; BRAN SELIC. Vol. 1939. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2000, pp. 323–337. ISBN: 978-3-540-41133-8 (Print), 978-3-540-40011-0 (Online). DOI: 10.1007/3-540-40011-7_23.

[FH12] JENS FRIEBEN; HENNING HEUTGER: "Case Study: Palladio-based Modular System for Simulating PLC Performance". In: *Proceedings of the Palladio Days 2012*. Ed. by STEFFEN BECKER; JENS HAPPE; ANNE KOZIOLEK; RALF REUSSNER. Vol. 2012,21. Karlsruhe Reports in Informatics. Karlsruhe, Germany: Karlsruhe Institute of Technology, 2012, pp. 27–35.

[FHMB13] JENS FRIEBEN; HENNING HEUTGER; MATTHIAS MEYER; STEFFEN BECKER: "Modulare Leistungsprognose von Kompaktsteuerungen". In: *9. Paderborner Workshop Entwurf mechatronischer Systeme*. Ed. by JÜRGEN GAUSEMEIER; ROMAN DUMITRESCU; FRANZ-JOSEF RAMMIG; WILHELM SCHÄFER; ANSGAR TRÄCHTLER. Vol. 310. HNI-Verlagsschriftenreihe. Paderborn, Germany: Heinz Nixdorf Institute, 2013, pp. 147–160. ISBN: 978-3-942647-29-8.

[FNTZ00] THORSTEN FISCHER; JÖRG NIERE; LARS TORUNSKI; ALBERT ZÜNDORF: "Story Diagrams – A new Graph Rewrite Language based on the Unified Modeling Language". In: *Theory and Application of Graph Transformations*. Ed. by HARTMUT EHRIG; GREGOR ENGELS; HANS-JÖRG KREOWSKI; GRZEGORZ ROZENBERG. Vol. 1764. Lecture Notes in Computer Science (LNCS). Berlin/-Heidelberg, Germany: Springer, 2000, pp. 157–167. ISBN: 978-3-540-67203-6 (Print), 978-3-540-46464-8 (Online). DOI: 10.1007/978-3-540-46464-8_21.

[Foc16] MARKUS FOCKEL: "ASIL Tailoring on Functional Safety Requirements". In: *Computer Safety, Reliability, and Security – Proceedings of the SAFECOMP 2016 Workshops, ASSURE, DECSoS, SASSUR, and TIPS*. Ed. by AMUND SKAVHAUG; JÉRÉMIE GUIOCHET; ERWIN SCHOITSCH; FRIEDEMANN BITSCH. Vol. 9923. Lecture Notes in Computer Science (LNCS). Cham, Switzerland:

Springer, 2016, pp. 298–310. ISBN: 978-3-319-45479-5 (Print), 978-3-319-45480-1 (Online). DOI: `10.1007/978-3-319-45480-1_24`.

[Fra06] URSULA FRANK: "Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme". PhD thesis. Paderborn, Germany: Paderborn University, 2006. ISBN: 978-3935433846.

[Fri17] JENS FRIEBEN: "Early Performance Analysis of Automation Systems Based on Systems Engineering Models". PhD thesis. Paderborn, Germany: Paderborn University, 2017.

[GBB12] THOMAS GOLDSCHMIDT; STEFFEN BECKER; ERIK BURGER: "Towards a Tool-Oriented Taxonomy of View-Based Modelling". In: *Modellierung 2012*. Ed. by ELMAR J. SINZ; ANDY SCHÜRR. Vol. P-201. GI-Edition – Lecture Notes in Informatics (LNI). Bonn, Germany: Köllen, 2012, pp. 59–74. ISBN: 978-3-88579-295-6.

[GBC+13] JOEL GREENYER; CHRISTIAN BRENNER; MAXIME CORDY; PATRICK HEYMANS; ERIKA GRESSI: "Incrementally Synthesizing Controllers from Scenario-based Product Line Specifications". In: *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Ed. by BERTRAND MEYER; LUCIANO BARESI. New York, USA: ACM, 2013, pp. 433–443. ISBN: 978-1-4503-2237-9. DOI: `10.1145/2491411.2491445`.

[GDS+15] JÜRGEN GAUSEMEIER; ROMAN DUMITRESCU; DANIEL STEFFEN; ANJA CZAJA; OLGA WIEDERKEHR; CHRISTIAN TSCHIRNER: *Systems Engineering in Industrial Practice*. Paderborn, Germany: Heinz Nixdorf Institute, Fraunhofer IEM, and Unity AG, 2015.

[GF12] JOEL GREENYER; JENS FRIEBEN: "Consistency Checking Scenario-Based Specifications of Dynamic Systems by Combining Simulation and Synthesis". In: *Proceedings of the 4th Workshop on Behaviour Modelling – Foundations and Applications*. New York, USA: ACM, 2012, pp. 1–9. ISBN: 978-1-4503-1187-8. DOI: `10.1145/2325276.2325278`.

[GFDK09] JÜRGEN GAUSEMEIER; URSULA FRANK; JÖRG DONOTH; SASCHA KAHL: "Specification technique for the description of self-optimizing mechatronic systems". In: *Research in Engineering Design* 20:4 (2009), pp. 201–223. ISSN: 0934-9839 (Print), 1435-6066 (Online). DOI: `10.1007/s00163-008-0058-x`.

[GGS+07] JÜRGEN GAUSEMEIER; HOLGER GIESE; WILHELM SCHÄFER; BJÖRN AXENATH; URSULA FRANK; STEFAN HENKLER; SEBASTIAN POOK; MATTHIAS TICHY: "Towards the Design of Self-Optimizing Mechatronic Systems: Consistency Between Domain-Spanning and Domain-Specific Models". In: *Proceedings of the 16th International Conference of Engineering Design (ICED'07)*. Ed. by JEAN-CLAUDE BOCQUET. Design Society, 2007.

[GHK+18] JOHANNES GEISMANN; ROBERT HÖTTGER; LUKAS KRAWCZYK; UWE POHLMANN; DAVID SCHMELTER: "Automated Synthesis of a Real-Time Scheduling for Cyber-Physical Multi-core Systems". In: *Revised Selected Papers of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*. Ed. by LUÍS FERREIRA PIRES; SLIMANE HAMMOUDI; BRAN SELIC. Communications in Computer and Information Science. Cham: Springer, 2018, pp. 72–93. ISBN: 978-3-319-94763-1 (Print), 978-3-319-94764-8 (Online). DOI: `10.1007/978-3-319-94764-8_4`.

[GPR11] JOEL GREENYER; SEBASTIAN POOK; JAN RIEKE: "Preventing Information Loss in Incremental Model Synchronization by Reusing Elements". In: *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011)*. Ed. by ROBERT B. FRANCE; JOCHEN M. KUESTER; BEHZAD BORDBAR; RICHARD F. PAIGE. Vol. 6698. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2011, pp. 144–159. ISBN: 978-3-642-21469-1 (Print), 978-3-642-21470-7 (Online). DOI: `10.1007/978-3-642-21470-7_11`.

[Gre11] JOEL GREENYER: "Scenario-based Design of Mechatronic Systems". PhD thesis. Paderborn, Germany: Paderborn University, 2011.

[GRS14] JÜRGEN GAUSEMEIER; FRANZ-JOSEF RAMMIG; WILHELM SCHÄFER: *Design Methodology for Intelligent Technical Systems – Develop Intelligent Technical Systems of the Future*. Lecture Notes in Mechanical Engineering. Berlin/Heidelberg, Germany: Springer, 2014. ISBN: 978-3-642-45434-9 (Print), 978-3-642-45435-6 (Online). DOI: `10.1007/978-3-642-45435-6`.

[GSG+09] JÜRGEN GAUSEMEIER; WILHELM SCHÄFER; JOEL GREENYER; SASCHA KAHL; SEBASTIAN POOK; JAN RIEKE: "Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems". In: *Proceedings of the 17th International Conference on Engineering Design (ICED'09)*. Ed. by MARGARETA NORELL BERGENDAHL; MARTIN GRIMHEDEN; LARRY LEIFER; PHILIPP SKOGSTAD; UDO LINDEMANN. Design Society, 2009, pp. 1–12. ISBN: 978-1-904670-10-0.

[GV14] JÜRGEN GAUSEMEIER; MAREEN VASSHOLZ: "Domain-Spanning Conceptual Design". In: *Design Methodology for Intelligent Technical Systems – Develop Intelligent Technical Systems of the Future*. Ed. by JÜRGEN GAUSEMEIER; FRANZ-JOSEF RAMMIG; WILHELM SCHÄFER. Lecture Notes in Mechanical Engineering. Berlin/Heidelberg, Germany: Springer, 2014. Chap. 3.2, pp. 69–73. ISBN: 978-3-642-45434-9 (Print), 978-3-642-45435-6 (Online).

[GW09] HOLGER GIESE; ROBERT WAGNER: "From Model Transformation to Incremental Bidirectional Model Synchronization". In: *Software & Systems Modeling* 8:1 (2009), pp. 21–43. ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: `10.1007/s10270-008-0089-9`.

[Hau05] JAN HENDRIK HAUSMANN: "Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages". PhD thesis. Paderborn, Germany: Paderborn University, 2005.

[HBDS15] CHRISTIAN HEINZEMANN; CHRISTIAN BRENNER; STEFAN DZIWOK; WILHELM SCHÄFER: "Automata-based refinement checking for real-time systems". In: *Computer Science – Research and Development* 30:3 (2015), pp. 255–283. ISSN: 1865-2034. DOI: 10.1007/s00450-014-0257-9.

[Hei15] CHRISTIAN HEINZEMANN: "Verification and Simulation of Self-Adaptive Mechatronic Systems". PhD thesis. Paderborn, Germany: Paderborn University, 2015.

[HPR+12] CHRISTIAN HEINZEMANN; UWE POHLMANN; JAN RIEKE; WILHELM SCHÄFER; OLIVER SUDMANN; MATTHIAS TICHY: "Generating Simulink and Stateflow Models from Software Specifications". In: *Proceedings of the 12th International Design Conference (DESIGN 2012)*. Ed. by DORIAN MARJANOVIĆ; MARIO ŠTORGA; NEVEN PAVKOVIĆ; NENAD BOJČETIĆ. Zagreb, Croatia: Faculty of Mechanical Engineering and Naval Architecture, 2012, pp. 475–484. ISBN: 978-953-7738-17-4.

[HRS13] CHRISTIAN HEINZEMANN; JAN RIEKE; WILHELM SCHÄFER: "Simulating Self-Adaptive Component-Based Systems Using MATLAB/Simulink". In: *Proceedings of the 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2013)*. Piscataway, USA: IEEE, 2013, pp. 71–80. ISBN: 978-0-7695-5129-6. DOI: 10.1109/SASO.2013.17.

[HSST13] CHRISTIAN HEINZEMANN; OLIVER SUDMANN; WILHELM SCHÄFER; MATTHIAS TICHY: "A Discipline-Spanning Development Process for Self-Adaptive Mechatronic Systems". In: *Proceedings of the 2013 International Conference on Software and Systems Process (ICSSP)*. Ed. by JÜRGEN MÜNCH; JO ANN LAN; HE ZHANG. New York, USA: ACM, 2013, pp. 36–45. ISBN: 978-1-4503-2062-7. DOI: 10.1145/2486046.2486055.

[IKDN13] PETER IWANEK; LYDIA KAISER; ROMAN DUMITRESCU; ALEXANDER NYSSEN: "Fachdisziplinübergreifende Systemmodellierung mechatronischer Systeme mit SysML und CONSENS". In: *Tag des Systems Engineering 2013 (TdSE 2013)*. Ed. by MAIK MAURER; SVEN-OLAF SCHULZE. München, Germany: Hanser, 2013, pp. 337–346. ISBN: 978-3-446-43915-3 (Print), 978-3-446-43946-7 (Online). DOI: 10.3139/9783446439467.032.

[Kai14] LYDIA KAISER: "Rahmenwerk zur Modellierung einer plausiblen Systemstruktur mechatronischer Systeme". PhD thesis. Paderborn, Germany: Paderborn University, 2014.

[Mey15] JAN MEYER: "Eine durchgängige modellbasierte Entwicklungsmethodik für die automobile Steuergeräteentwicklung unter Einbeziehung des AUTOSAR Standards". PhD thesis. Paderborn, Germany: Paderborn University, 2015.

[NMK10] ULRICH NICKEL; JAN MEYER; TAPIO KRAMER: "Wie hoch ist die Performance?" In: *Automobil-Elektronik* 3 (2010), pp. 36–38.

[PH15] UWE POHLMANN; MARCUS HÜWE: "Model-Driven Allocation Engineering". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ed. by MYRA COHEN; LARS GRUNSKE; MICHAEL WHALEN. Los Alamitos, USA: IEEE, 2015, pp. 374–384. ISBN: 978-1-5090-0024-1. DOI: 10.1109/ASE.2015.18.

[PH18] UWE POHLMANN; MARCUS HÜWE: “Model-driven allocation engineering: specifying and solving constraints based on the example of automotive systems”. In: *Automated Software Engineering* (2018). ISSN: 0928-8910 (Print), 1573-7535 (Online). DOI: `10.1007/s10515-018-0248-3`.

[PMDB14] UWE POHLMANN; MATTHIAS MEYER; ANDREAS DANN; CHRISTOPHER BRINK: “Viewpoints and Views in Hardware Platform Modeling for Safe Deployment”. In: *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO’14)*. Ed. by COLIN ATKINSON. New York, USA: ACM, 2014, pp. 23–30. ISBN: 978-1-4503-2900-2. DOI: `10.1145/2631675.2631682`.

[Poh18] UWE POHLMANN: “A Model-driven Software Construction Approach for Cyber-physical Systems”. PhD thesis. Paderborn, Germany: Paderborn University, 2018. ISBN: 978-3-00-059657-5.

[RDS$^+$12] JAN RIEKE; RAFAŁ DOROCIAK; OLIVER SUDMANN; JÜRGEN GAUSEMEIER; WILHELM SCHÄFER: “Management of Cross-Domain Model Consistency for Behavior models of Mechatronic Systems”. In: *Proceedings of the 12th International Design Conference (DESIGN 2012)*. Ed. by DORIAN MARJANOVIĆ; MARIO ŠTORGA; NEVEN PAVKOVIĆ; NENAD BOJČETIĆ. Zagreb, Croatia: Faculty of Mechanical Engineering and Naval Architecture, 2012, pp. 1781–1790. ISBN: 978-953-7738-17-4.

[Rie08] JAN RIEKE: “Konsistenzerhaltung zwischen domänenübergreifenden und domänenspezifischen Modellen im Entwicklungsprozess mechatronischer Systeme”. Diploma Thesis. Paderborn, Germany: Paderborn University, 2008.

[Rie14] JAN RIEKE: “Automatic Model Transformation and Synchronization”. In: *Design Methodology for Intelligent Technical Systems – Develop Intelligent Technical Systems of the Future*. Ed. by JÜRGEN GAUSEMEIER; FRANZ-JOSEF RAMMIG; WILHELM SCHÄFER. Lecture Notes in Mechanical Engineering. Berlin/Heidelberg, Germany: Springer, 2014. Chap. 5.1, pp. 186–197. ISBN: 978-3-642-45434-9 (Print), 978-3-642-45435-6 (Online).

[Rie15] JAN RIEKE: “Model Consistency Management for Systems Engineering”. PhD thesis. Paderborn, Germany: Paderborn University, 2015.

[RS12] JAN RIEKE; OLIVER SUDMANN: “Specifying Refinement Relations in Vertical Model Transformations”. In: *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*. Ed. by ANTONIO VALLECILLO; JUHA-PEKKA TOLVANEN; EKKART KINDLER; HARALD STÖRRLE; DIMITRIOS S. KOLOVOS. Vol. 7349. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2012, pp. 210–225. ISBN: 978-3-642-31490-2 (Print), 978-3-642-31491-9 (Online). DOI: `10.1007/978-3-642-31491-9_17`.

[Sol13] CHRISTIAN SOLTENBORN: “Quality Assurance with Dynamic Meta Modeling”. PhD thesis. Paderborn, Germany: Paderborn University, 2013.

[SW07] WILHELM SCHÄFER; HEIKE WEHRHEIM: “The Challenges of Building Advanced Mechatronic Systems”. In: *Proceedings of the Conference on Future of Software Engineering (FOSE)*. Ed. by LIONEL C. BRIAND; ALEXANDER L. WOLF. Washington, USA: IEEE, 2007, pp. 72–84. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.28.

[TDBG15] CHRISTIAN TSCHIRNER; ROMAN DUMITRESCU; MICHAEL BANSMANN; JÜRGEN GAUSEMEIER: “Tailoring Model-Based Systems Engineering – Concepts for Industrial Application”. In: *Proceedings of the 9th Annual IEEE Systems Conference (SysCon)*. Piscataway, USA: IEEE, 2015, pp. 69–76. ISBN: 978-1-4799-5926-6. DOI: 10.1109/SYSCON.2015.7116731.

[Tsc16] CHRISTIAN TSCHIRNER: “Rahmenwerk zur Integration des modellbasierten Systems Engineering in die Produktentstehung mechatronischer Systeme”. PhD thesis. Paderborn, Germany: Paderborn University, 2016.

[Wag09] ROBERT WAGNER: “Inkrementelle Modellsynchronisation”. PhD thesis. Paderborn, Germany: Paderborn University, 2009.

Literature

[ABD+95] NEIL C. AUDSLEY; ALAN BURNS; ROBERT I. DAVIS; KEN W. TINDELL; ANDY J. WELLINGS: “Fixed priority pre-emptive scheduling: An historical perspective”. In: *Real-Time Systems* 8:2 (1995), pp. 173–198. ISSN: 0922-6443. DOI: 10.1007/BF01094342.

[aca11] ACATECH, ed.: *Cyber-Physical Systems – Driving force for innovation in mobility, health, energy and production*. acatech BEZIEHT POSITION. Berlin/Heidelberg, Germany: Springer, 2011. ISBN: 978-3-642-29090-9. DOI: 10.1007/978-3-642-29090-9.

[AD94] RAJEEV ALUR; DAVID L. DILL: “A theory of timed automata”. In: *Theoretical Computer Science* 126:2 (1994), pp. 183–235. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8.

[AGI+13] SILVIA ABRAHÃO; CARMINE GRAVINO; EMILIO INSFRAN; GIUSEPPE SCANNIELLO; GENOVEFFA TORTORA: “Assessing the Effectiveness of Sequence Diagrams in the Comprehension of Functional Requirements: Results from a Family of Five Experiments”. In: *IEEE Transactions on Software Engineering* 39:3 (2013), pp. 327–342. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.27.

[AHKM08] YORAM ATIR; DAVID HAREL; ASAF KLEINBORT; SHAHAR MAOZ: “Object Composition in Scenario-Based Programming”. In: *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Ed. by JOSÉ LUIZ FIADEIRO; PAOLA INVERARDI. Vol. 4961. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 2008, pp. 301–316. ISBN: 978-3-540-78742-6 (Print), 978-3-540-78743-3 (Online). DOI: 10.1007/978-3-540-78743-3_23.

[Ake78] SHELDON B. AKERS: “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* C-27:6 (1978), pp. 509–516. ISSN: 0018-9340. DOI: `10.1109/TC.1978.1675141`.

[And09] CHARLES ANDRÉ: *Syntax and Semantics of the Clock Constraint Specification Language (CCSL)*. Research Report RR-6925. INRIA, 2009.

[Anj14] ANTHONY ANJORIN: “Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars”. PhD thesis. Darmstadt, Germany: Darmstadt University of Technology, 2014.

[ANRS06] NETTA AIZENBUD-RESHEF; BRIAN T. NOLAN; JULIA RUBIN; YAEL SHAHAM-GAFNI: “Model Traceability”. In: *IBM Systems Journal* 45:3 (2006), pp. 515–526. ISSN: 0018-8670. DOI: `10.1147/sj.453.0515`.

[AT16] SOFIANE ACHICHE; TETSUO TOMIYAMA: “Design of Multidisciplinary Cyber Physical Systems”. In: *Journal of Integrated Design and Process Science* 19:3 (2016). Editorial, pp. 1–3. ISSN: 1092-0617 (Print), 1875-8959 (Online). DOI: `10.3233/jid-2015-0015`.

[Aus96] DAVID M. AUSLANDER: “What is Mechatronics?” In: *IEEE/ASME Transactions on Mechatronics* 1:1 (1996), pp. 5–9. ISSN: 1083-4435. DOI: `10.1109/3516.491404`.

[AVS12] ANTHONY ANJORIN; GERGELY VARRÓ; ANDY SCHÜRR: “Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques”. In: *Proceedings of the 1st International Workshop on Bidirectional Transformations (BX 2012)*. Ed. by FRANK HERRMANN; JANIS VOIGTLÄNDER. Vol. 49. Electronic Communications of the EASST. 2012. DOI: `10.14279/tuj.eceasst.49.707`.

[BBH+10] PETER BRAUN; MANFRED BROY; FRANK HOUDEK; MATTHIAS KIRCHMAYR; MARK MÜLLER; BIRGIT PENZENSTADLER; KLAUS POHL; THORSTEN WEYER: “Guiding requirements engineering for software-intensive embedded systems in the automotive industry – The REMsES approach”. In: *Computer Science – Research and Development* 29:1 (2010), pp. 21–43. ISSN: 1865-2034. DOI: `10.1007/s00450-010-0136-y`.

[BDV+16] ERWAN BOUSSE; THOMAS DEGUEULE; DIDIER VOJTISEK; TANJA MAYERHOFER; JULIEN DEANTONI; BENOÎT COMBEMALE: “Execution Framework of the GEMOC Studio (Tool Demo)”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. Ed. by TIJS VAN DER STORM; EMILIE BALLAND; DANIEL VARRO. New York, USA: ACM, 2016, pp. 84–89. ISBN: 978-1-4503-4447-0. DOI: `10.1145/2997364.2997384`.

[BEGL05] SUSANNA BYHLIN; ANDREAS ERMEDAHL; JAN GUSTAFSSON; BJÖRN LISPER: “Applying Static WCET Analysis to Automotive Communication Software”. In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS’05)*. Los Alamitos, USA: IEEE, 2005, pp. 249–258. ISBN: 0-7695-2400-1. DOI: `10.1109/ECRTS.2005.7`.

[BHH+14] WOLFGANG BÖHM; STEFAN HENKLER; FRANK HOUDEK; ANDREAS VOGELSANG; THORSTEN WEYER: "Bridging the Gap between Systems and Software Engineering by Using the SPES Modeling Framework as a General Systems Engineering Philosophy". In: *Procedia Computer Science* 28 (2014), pp. 187–194. ISSN: 1877-0509. DOI: `10.1016/j.procs.2014.03.024`.

[BKFV14] GIACOMO BARBIERI; KONSTANTIN KERNSCHMIDT; CESARE FANTUZZI; BIRGIT VOGEL-HEUSER: "A SysML Based Design Pattern for the High-Level Development of Mechatronic Systems to Enhance Re-Usability". In: *Proceedings of the 19th IFAC World Congress*. Ed. by EDWARD BOJE; XIAOHUA XIA. International Federation of Automatic Control (IFAC), 2014, pp. 3431–3437. ISBN: 978-3-902823-62-5. DOI: `10.3182/20140824-6-ZA-1003.00615`.

[BLY09] LIONEL C. BRIAND; YVAN LABICHE; TAO YUE: "Automated traceability analysis for UML model refinements". In: *Information and Software Technology* 51:2 (2009), pp. 512–527. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2008.06.002`.

[Boe00] BARRY W. BOEHM: "Unifying Software Engineering and Systems Engineering". In: *Computer* 33:3 (2000), pp. 114–116. ISSN: 0018-9162. DOI: `10.1109/2.825714`.

[Boe81] BARRY W. BOEHM: *Software Engineering Economics*. Englewood Cliffs, USA: Prentice-Hall, 1981. ISBN: 978-0138221225.

[Boe83] BARRY W. BOEHM: "Seven Basic Principles of Software Engineering". In: *Journal of Systems and Software* 3:1 (1983), pp. 3–24. ISSN: 0164-1212. DOI: `10.1016/0164-1212(83)90003-1`.

[Bou08] HANIFA BOUCHENEB: "Interval timed coloured Petri net: efficient construction of its state class space preserving linear properties". In: *Formal Aspects of Computing* 20:2 (2008), pp. 225–238. ISSN: 0934-5043 (Print), 1433-299X (Online). DOI: `10.1007/s00165-007-0050-7`.

[Bry86] RANDAL E. BRYANT: "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35:8 (1986), pp. 677–691. ISSN: 0018-9340. DOI: `10.1109/TC.1986.1676819`.

[BS02] KURT BITTNER; IAN SPENCE: *Use Case Modeling*. Amsterdam, Netherlands: Addison Wesley, 2002. ISBN: 978-0201709131.

[BS03] EGON BÖRGER; ROBERT STARK: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Berlin/Heidelberg, Germany: Springer, 2003. ISBN: 978-3-642-62116-1 (Print), 978-3-642-18216-7 (Online). DOI: `10.1007/978-3-642-18216-7`.

[But11] GIORGIO C. BUTTAZZO: *Hard real-time computing systems – Predictable scheduling algorithms and applications*. 3rd edition. New York: Springer, 2011. ISBN: 978-1-4614-0675-4 (Print), 978-1-4614-0676-1 (Online). DOI: `10.1007/978-1-4614-0676-1`.

[CA07] CHENG, BETTY H. C.; JOANNE M. ATLEE: “Research Directions in Requirements Engineering”. In: *Proceedings of the Conference on Future of Software Engineering (FOSE)*. Ed. by LIONEL C. BRIAND; ALEXANDER L. WOLF. Washington, USA: IEEE, 2007, pp. 285–303. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.17.

[CA09] CHENG, BETTY H. C.; JOANNE M. ATLEE: “Current and Future Research Directions in Requirements Engineering”. In: *Design Requirements Engineering: A Ten-Year Perspective – Revised and Invited Papers*. Ed. by KALLE LYYTINEN; PERICLES LOUCOPOULOS; JOHN MYLOPOULOS; BILL ROBINSON. Vol. 14. Lecture Notes in Business Information Processing. Berlin/Heidelberg: Springer, 2009, pp. 11–43. ISBN: 978-3-540-92965-9 (Print), 978-3-540-92966-6 (Online). DOI: 10.1007/978-3-540-92966-6_2.

[CAL$^+$13] BENOÎT COMBEMALE; JULIEN de ANTONI; MATIAS VARA LARSEN; FRÉDÉRIC MALLET; OLIVIER BARAIS; BENOIT BAUDRY; ROBERT B. FRANCE: “Reifying Concurrency for Executable Metamodeling”. In: *Proceedings of the 6th International Conference on Software Language Engineering*. Ed. by MARTIN ERWIG; RICHARD F. PAIGE; ERIC VAN WYK. Vol. 8225. Lecture Notes in Computer Science (LNCS). Cham, Switzerland: Springer, 2013, pp. 365–384. ISBN: 978-3-319-02653-4 (Print), 978-3-319-02654-1 (Online). DOI: 10.1007/978-3-319-02654-1_20.

[CGH$^+$14] JANE CLELAND-HUANG; ORLENA C. Z. GOTEL; JANE HUFFMAN HAYES; PATRICK MÄDER; ANDREA ZISMAN: “Software Traceability – Trends and Future Directions”. In: *Proceedings of the Conference on Future of Software Engineering (FOSE)*. Ed. by MATTHEW DWYER; JAMES HERBSLEB. FOSE 2014. New York, USA: ACM, 2014, pp. 55–69. ISBN: 978-1-4503-2865-4. DOI: 10.1145/2593882.2593891.

[CH06] KRZYSZTOF CZARNECKI; SIMON HELSEN: “Feature-based survey of model transformation approaches”. In: *IBM Systems Journal* 45:3 (2006), pp. 621–645. ISSN: 0018-8670. DOI: 10.1147/sj.453.0621.

[CHK08] PIERRE COMBES; DAVID HAREL; HILLEL KUGLER: “Modeling and verification of a telecommunication application using live sequence charts and the Play-Engine tool”. In: *Software & Systems Modeling* 7:2 (2008), pp. 157–175. ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: 10.1007/s10270-007-0069-5.

[CHQW16] THORSTEN CZIHARZ; PETER HRUSCHKA; STEFAN QUEINS; THORSTEN WEYER: *Handbook of Requirements Modeling According to the IREB Standard*. Version 1.2. 2016.

[CLP11] YUE CAO; YUSHENG LIU; CHRISTIAAN J. J. PAREDIS: “System-level model integration of design and simulation for mechatronic systems based on SysML”. In: *Mechatronics* 21:6 (2011), pp. 1063–1075. ISSN: 0957-4158. DOI: 10.1016/j.mechatronics.2011.05.003.

[DB08] Robert I. Davis; Alan Burns: "Response Time Upper Bounds for Fixed Priority Real-Time Systems". In: *Proceedings of the 2008 Real-Time Systems Symposium (RTSS)*. Los Alamitos, USA: IEEE, 2008, pp. 407–418. isbn: 978-0-7695-3477-0. doi: `10.1109/RTSS.2008.18`.

[DCB+15] Thomas Degueule; Benoît Combemale; Arnaud Blouin; Olivier Barais; Jean-Marc Jézéquel: "Melange: A Meta-language for Modular and Reusable Development of DSLs". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*. Ed. by Richard F. Paige; Davide Di Ruscio; Markus Völter. New York, USA: ACM, 2015, pp. 25–36. isbn: 978-1-4503-3686-4.

[DDC+14] Julien DeAntoni; Issa Papa Diallo; Joël Champeau; Benoît Combemale; Ciprian Teodorov: *Operational Semantics of the Model of Concurrency and Communication Language*. Research Report RR-8584. INRIA, 2014.

[DeM79] Tom DeMarco: *Structured Analysis and System Specification*. Upper Saddle River, USA: Prentice-Hall, 1979. isbn: 0-13-854380-1.

[DH01] Werner Damm; David Harel: "LSCs: Breathing Life into Message Sequence Charts". In: *Formal Methods in System Design* 19 (2001), pp. 45–80. issn: 0925-9856. doi: `10.1023/A:1011227529550`.

[DKPF10] Nikolaos Drivalos-Matragkas; Dimitrios S. Kolovos; Richard F. Paige; Kiran J. Fernandes: "A State-based Approach to Traceability Maintenance". In: *Proceedings of the 6th ECMFA Traceability Workshop*. Ed. by Jon Oldevik. New York, USA: ACM, 2010, pp. 23–30. isbn: 978-1-60558-993-0. doi: `10.1145/1814392.1814396`.

[DM12a] Julien DeAntoni; Frédéric Mallet: "TimeSquare: Treat Your Models with Logical Time". In: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe)*. Ed. by Carlo A. Furia; Sebastian Nanz. Vol. 7304. Berlin/Heidelberg, Germany: Springer, 2012, pp. 34–41. isbn: 978-3-642-30560-3 (Print), 978-3-642-30561-0 (Online). doi: `10.1007/978-3-642-30561-0_4`.

[DM12b] Julien DeAntoni; Frédéric Mallet: *ECL: the Event Constraint Language, an Extension of OCL with Events*. Research Report RR-8031. INRIA, 2012.

[DPFK06] Nicolas Drivalos; Richard F. Paige; Kiran J. Fernandes; Dimitrios S. Kolovos: "Towards Rigorously Defined Model-to-Model Traceability". In: *Proceedings of the 2nd ECMDA Traceability Workshop (ECMDA-TW)*. 2006.

[DSTH17] Darko Durisic; Miroslaw Staron; Matthias Tichy; Jörgen Hansson: "Assessing the impact of meta-model evolution: a measure and its automotive application". In: *Software & Systems Modeling* (2017). issn: 1619-1366 (Print), 1619-1374 (Online). doi: `10.1007/s10270-017-0601-1`.

[DTW12] Marian Daun; Bastian Tenbergen; Thorsten Weyer: "Requirements Viewpoint". In: *Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology*. Ed. by Klaus Pohl; Harald Hönninger; Reinhold E. Achatz; Manfred Broy. Berlin/Heidelberg, Germany: Springer, 2012. Chap. 4, pp. 51–68. isbn: 978-3-642-34613-2 (Print), 978-3-642-34614-9 (Online). doi: `10.1007/978-3-642-34614-9_4`.

[DWUL17] Christian Dietrich; Peter Wägemann; Peter Ulbrich; Daniel Lohmann: "SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems". In: *2017 IEEE 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Ed. by Gabriel Parmer. Piscataway, USA: IEEE, 2017, pp. 37–48. isbn: 978-1-5090-5269-1. doi: `10.1109/RTAS.2017.37`.

[EAG06] Angelina Espinoza; P. Pedro Alarcón; Juan Garbajosa: "Analyzing and Systematizing Current Traceability Schemas". In: *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*. Los Alamitos, USA: IEEE, 2006, pp. 21–32. isbn: 0-7695-2624-1. doi: `10.1109/SEW.2006.12`.

[EGZ12] Martin Eigner; Torsten Gilz; Radoslav Zafirov: "Proposal for Functional Product Description as Part of a PLM Solution in Interdisciplinary Product Development". In: *Proceedings of the 12th International Design Conference (DESIGN 2012)*. Ed. by Dorian Marjanović; Mario Štorga; Neven Pavković; Nenad Bojčetić. Vol. 70. DS. Zagreb, Croatia: Faculty of Mechanical Engineering and Naval Architecture, 2012, pp. 1667–1676. isbn: 978-953-7738-17-4.

[EMV12] Sebastian Eder; Jakob Mund; Andreas Vogelsang: "Logical Viewpoint". In: *Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology*. Ed. by Klaus Pohl; Harald Hönninger; Reinhold E. Achatz; Manfred Broy. Berlin/Heidelberg: Springer, 2012. Chap. 6, pp. 85–93. isbn: 978-3-642-34613-2 (Print), 978-3-642-34614-9 (Online). doi: `10.1007/978-3-642-34614-9_6`.

[Fid91] Colin Fidge: "Logical time in distributed computing systems". In: *Computer* 24:8 (1991), pp. 28–33. issn: 0018-9162. doi: `10.1109/2.84874`.

[FKV14] Stefan Feldmann; Konstantin Kernschmidt; Birgit Vogel-Heuser: "Combining a SysML-based Modeling Approach and Semantic Technologies for Analyzing Change Influences in Manufacturing Plant Models". In: *Procedia CIRP (Variety Management in Manufacturing — Proceedings of the 47th CIRP Conference on Manufacturing Systems)* 17 (2014). Ed. by Hoda ElMaraghy, pp. 451–456. doi: `10.1016/j.procir.2014.01.140`.

[FMS12] Sanford Friedenthal; Alan Moore; Rick Steiner: *A Practical Guide to SysML – The Systems Modeling Language*. 2nd edition. Waltham, USA: Morgan Kaufmann, 2012. isbn: 978-0-12-385206-9.

[FRNJ08] NICO FEIERTAG; KAI RICHTER; JOHAN NORDLANDER; JAN JONSSON: “A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics”. In: *Proceedings of the 1st International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*. 2008.

[GCH+12] ORLENA C. Z. GOTEL; JANE CLELAND-HUANG; JANE HUFFMAN HAYES; ANDREA ZISMAN; ALEXANDER EGYED; PAUL GRÜNBACHER; ALEX DEKHTYAR; GIULIANO ANTONIOL; JONATHAN MALETIC; PATRICK MÄDER: “Traceability Fundamentals”. In: *Software and Systems Traceability*. Ed. by JANE CLELAND-HUANG; ORLENA GOTEL; ANDREA ZISMAN. London, UK: Springer, 2012, pp. 3–22. ISBN: 978-1-4471-2238-8 (Print), 978-1-4471-2239-5 (Online). DOI: `10.1007/978-1-4471-2239-5_1`.

[GDM+15] CALIN GLITIA; JULIEN DEANTONI; FRÉDÉRIC MALLET; JEAN-VIVIEN MILLO; PIERRE BOULET; ABDOULAYE GAMATIÉ: “Progressive and explicit refinement of scheduling for multidimensional data-flow applications using UML MARTE ”. In: *Design Automation for Embedded Systems* 19:1-2 (2015), pp. 1–33. ISSN: 0929-5585 (Print), 1572-8080 (Online). DOI: `10.1007/s10617-014-9140-y`.

[GDPM13] ARDA GOKNIL; JULIEN DEANTONI; MARIE-AGNÈS PERALDI-FRATI; FRÉDÉRIC MALLET: “Tool Support for the Analysis of TADL2 Timing Constraints Using TIMESQUARE ”. In: *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS)*. Piscataway, USA: IEEE, 2013, pp. 145–154. ISBN: 978-0-7695-5007-7. DOI: `10.1109/ICECCS.2013.28`.

[GF94] ORLENA C. Z. GOTEL; ANTHONY C. W. FINKELSTEIN: “An Analysis of the Requirements Traceability Problem”. In: *Proceedings of the 1st International Conference on Requirements Engineering (RE)*. Los Alamitos, USA: IEEE, 1994, pp. 94–101. ISBN: 0-8186-5480-5. DOI: `10.1109/ICRE.1994.292398`.

[GH09] HOLGER GIESE; STEPHAN HILDEBRANDT: *Efficient Model Synchronization of Large-Scale Models*. Tech. rep. 28. Potsdam, Germany: Hasso Plattner Institute at the University of Potsdam, 2009.

[GKS00] RADU GROSU; INGOLF KRÜGER; THOMAS STAUNER: “Hybrid Sequence Charts”. In: *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*. Los Alamitos, USA: IEEE, 2000, pp. 104–111. ISBN: 0-7695-0607-0. DOI: `10.1109/ISORC.2000.839517`.

[GLO09] ESTHER GUERRA; JUAN de LARA; FERNANDO OREJAS: “Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions”. In: *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009)*. Ed. by RICHARD F. PAIGE. Vol. 5563. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2009, pp. 83–99. ISBN: 978-3-642-02407-8 (Print), 978-3-642-02408-5 (Online). DOI: `10.1007/978-3-642-02408-5_7`.

[Gue91] DAVID GUEST: "The Hunt is on for the Renaissance Man of Computing". In: *The Independent (London)* 17 (1991).

[Har00] DAVID HAREL: "From Play-In Scenarios to Code: An Achievable Dream". In: *Fundamental Approaches to Software Engineering*. Ed. by TOM MAIBAUM. Vol. 1783. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 2000, pp. 22–34. ISBN: 978-3-540-67261-6 (Print), 978-3-540-46428-0 (Online). DOI: `10.1007/3-540-46428-X_3`.

[Har01] DAVID HAREL: "From Play-In Scenarios to Code: An Achievable Dream". In: *Computer* 34:1 (2001), pp. 53–60. ISSN: 0018-9162. DOI: `10.1109/2.895118`.

[Har87] DAVID HAREL: "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8:3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: `10.1016/0167-6423(87)90035-9`.

[Has09] JAMELEDDINE HASSINE: "Early Schedulability Analysis with Timed Use Case Maps". In: *Proceedings of the 14th International SDL Forum – Design for Motes and Mobiles (SDL 2009)*. Ed. by RICK REED; ATTILA BILGIC; REINHARD GOTZHEIN. Vol. 5719. Lecture Notes in Computer Science (LNCS). Berlin/-Heidelberg, Germany: Springer, 2009, pp. 98–114. ISBN: 978-3-642-04553-0 (Print), 978-3-642-04554-7 (Online). DOI: `10.1007/978-3-642-04554-7_7`.

[Has15] JAMELEDDINE HASSINE: "Early modeling and validation of timed system requirements using Timed Use Case Maps". In: *Requirements Engineering* 20:2 (2015), pp. 181–211. ISSN: 0947-3602. DOI: `10.1007/s00766-013-0200-9`.

[HH04] NADINE HEUMESSER; FRANK HOUDEK: "Experiences in Managing an Automotive Requirements Engineering Process". In: *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE)*. Los Alamitos, CA: IEEE, 2004, pp. 322–327. ISBN: 0-7695-2174-6. DOI: `10.1109/ICRE.2004.1335690`.

[HHRS05] ØYSTEIN HAUGEN; KNUT EILIF HUSA; RAGNHILD KOBRO RUNDE; KETIL STØLEN: "Why Timed Sequence Diagrams Require Three-Event Semantics". In: *Scenarios: Models, Transformations and Tools*. Ed. by STEFAN LEUE; TARJA JOHANNA SYSTÄ. Vol. 3466. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2005, pp. 1–25. ISBN: 978-3-540-26189-6. DOI: `10.1007/11495628_1`.

[HHT96] ANNEGRET HABEL; REIKO HECKEL; GABRIELE TAENTZER: "Graph grammars with negative application conditions". In: *Fundamenta Informaticae (Special Issue)* 26:3,4 (1996). Ed. by GREGOR ENGELS; HARTMUT EHRIG; GRZEGORZ ROZENBERG, pp. 87–313. ISSN: 0169-2968 (Print), 1875-8681 (Online). DOI: `10.3233/FI-1996-263404`.

[Hil14] STEPHAN HILDEBRANDT: "On the Performance and Conformance of Triple Graph Grammar Implementations". PhD thesis. Potsdam, Germany: Hasso Plattner Institute at the University of Potsdam, 2014.

[HKM07] DAVID HAREL; ASAF KLEINBORT; SHAHAR MAOZ: “S2A: A Compiler for Multi-modal UML Sequence Diagrams”. In: *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Ed. by MATTHEW B. DWYER; ANTÓNIA LOPES. Vol. 4422. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 2007, pp. 121–124. ISBN: 978-3-540-71288-6 (Print), 978-3-540-71289-3 (Online). DOI: `10.1007/978-3-540-71289-3_11`.

[HKMP02] DAVID HAREL; HILLEL KUGLER; RAMI MARELLY; AMIR PNUELI: “Smart Play-out of Behavioral Requirements”. In: *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by MARK D. AAGAARD; JOHN W. O'LEARY. Vol. 2517. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2002, pp. 378–398. ISBN: 978-3-540-00116-4 (Print), 978-3-540-36126-8 (Online). DOI: `10.1007/3-540-36126-X_23`.

[HKMP03] DAVID HAREL; HILLEL KUGLER; RAMI MARELLY; AMIR PNUELI: “Smart Play-out”. In: *Companion Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. New York, USA: ACM, 2003, pp. 68–69. ISBN: 1-58113-751-6. DOI: `10.1145/949344.949353`.

[HKP04] DAVID HAREL; HILLEL KUGLER; AMIR PNUELI: “Smart Play-Out Extended: Time and Forbidden Elements”. In: *Proceedings of the 4th International Conference on Quality Software (QSIC)*. Ed. by HANS-DIETER EHRICH; KLAUS-DIETER SCHEWE. Los Alamitos, USA: IEEE, 2004, pp. 2–10. ISBN: 0-7695-2207-6. DOI: `10.1109/QSIC.2004.1357938`.

[HLMR13] MATS P.E. HEIMDAHL; LIAN DUAN; ANITHA MURUGESAN; SANJAI RAYADURGAM: “Modeling and Requirements on the Physical Side of Cyber-Physical Systems”. In: *2nd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*. 2013, pp. 1–7. DOI: `10.1109/TwinPeaks.2013.6614716`.

[HM03a] DAVID HAREL; RAMI MARELLY: *Come, let's play: Scenario-based programming using LSCs and the play-engine*. Berlin/Heidelberg, Germany: Springer, 2003. ISBN: 3540007873.

[HM03b] DAVID HAREL; RAMI MARELLY: “Specifying and executing behavioral requirements: the play-in/play-out approach”. In: *Software & Systems Modeling* 2:2 (2003), pp. 82–107. ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: `10.1007/s10270-002-0015-5`.

[HM06] DAVID HAREL; SHAHAR MAOZ: “Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams”. In: *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM)*. Ed. by JON WHITTLE; LEIF GEIGER; MICHAEL MEISINGER. New York, USA: ACM, 2006, pp. 13–20. ISBN: 1-59593-394-8. DOI: `10.1145/1138953.1138958`.

[HM08] David Harel; Shahar Maoz: "Assert and negate revisited: Modal semantics for UML sequence diagrams". In: *Software and Systems Modeling* 7:2 (2008), pp. 237–252. ISSN: 1619-1374. DOI: 10.1007/s10270-007-0054-z.

[HMSB10] David Harel; Shahar Maoz; Smadar Szekely; Daniel Barkan: "PlayGo: Towards a Comprehensive Tool for Scenario Based Programming". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, USA: ACM, 2010, pp. 359–360. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859075.

[HP85] David Harel; Amir Pnueli: "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt. Vol. 13. NATO ASI Series, Series F: Computer and Systems Sciences. Berlin/-Heidelberg, Germany: Springer, 1985, pp. 477–498. ISBN: 978-3-642-82455-5 (Print), 978-3-642-82453-1 (Online). DOI: 10.1007/978-3-642-82453-1_17.

[HR04] David Harel; Bernhard Rumpe: "Meaningful modeling: What's the semantics of 'semantics'?" In: *Computer* 37:10 (2004), pp. 64–72. ISSN: 0018-9162. DOI: 10.1109/MC.2004.172.

[HRD06] Jameleddine Hassine; Jürgen Rilling; Rachida Dssouli: "Timed Use Case Maps". In: *Revised Selected Papers of the 5th International Workshop on System Analysis and Modeling: Language Profiles (SAM 2006)*. Ed. by Reinhard Gotzhein; Rick Reed. Vol. 4320. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 2006, pp. 99–114. ISBN: 978-3-540-68371-1 (Print), 978-3-540-68373-5 (Online). DOI: 10.1007/11951148_7.

[HRD10] Jameleddine Hassine; Jürgen Rilling; Rachida Dssouli: "An evaluation of timed scenario notations". In: *Journal of Systems and Software* 83:2 (2010), pp. 326–350. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.09.014.

[HS07] Thomas A. Henzinger; Joseph Sifakis: "The Discipline of Embedded Systems Design". In: *Computer* 40:10 (2007), pp. 32–40. ISSN: 0018-9162. DOI: 10.1109/MC.2007.364.

[HY12] Seungwok Han; Hee Yong Youn: "Modeling and Analysis of Time-Critical Context-Aware Service Using Extended Interval Timed Colored Petri Nets". In: *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans* 42:3 (2012), pp. 630–640. ISSN: 1083-4427. DOI: 10.1109/TSMCA.2011.2170064.

[INC14] International Council on Systems Engineering (INCOSE): *A World in Motion – Systems Engineering Vision 2025*. 2014.

[INCOSE] International Council on Systems Engineering (INCOSE): *What is Systems Engineering?* http://www.incose.org/AboutSE/WhatIsSE. Last accessed: January 2018.

[Jou05] Frédéric Jouault: "Loosely Coupled Traceability for ATL". In: *Proceedings of the 1st ECMDA-FA Traceability Workshop*. 2005, pp. 29–37.

[JP86] MATHAI JOSEPH; PARITOSH PANDYA: "Finding Response Times in a Real-Time System". In: *The Computer Journal* 29:5 (1986), pp. 390–395. DOI: `10.1093/comjnl/29.5.390`.

[JVTM17] MIGUEL JIMÉNEZ; NORHA M. VILLEGAS; GABRIEL TAMURA; HAUSI A. MÜLLER: "Deployment Specification Challenges in the Context of Large Scale Systems". In: *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. Riverton, USA: IBM, 2017, pp. 220–226.

[KBC+18] NAFISEH KAHANI; MOJTABA BAGHERZADEH; JAMES R. CORDY; JUERGEN DINGEL; DANIEL VARRÓ: "Survey and classification of model transformation tools". In: *Software & Systems Modeling* (2018). ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: `10.1007/s10270-018-0665-6`.

[KEF09] ERIC KNAUSS; CHRISTIAN EL BOUSTANI; THOMAS FLOHR: "Investigating the Impact of Software Requirements Specification Quality on Project Success". In: *Proceedings of the 10th International Conference on Product-Focused Software Process Improvement (PROFES)*. Ed. by FRANK BOMARIUS; MARKKU OIVO; PÄIVI JARING; PEKKA ABRAHAMSSON. Berlin/Heidelberg: Springer, 2009, pp. 28–42. ISBN: 978-3-642-02151-0 (Print), 978-3-642-02152-7 (Online).

[KFV18] KONSTANTIN KERNSCHMIDT; STEFAN FELDMANN; BIRGIT VOGEL-HEUSER: "A model-based framework for increasing the interdisciplinary design of mechatronic production systems". In: *Journal of Engineering Design* 29:11 (2018), pp. 617–643. ISSN: 0954-4828 (Print), 1466-1837 (Online). DOI: `10.1080/09544828.2018.1520205`.

[Kop11] HERMANN KOPETZ: *Real-time systems – Design principles for distributed embedded applications*. 2nd edition. New York, USA: Springer, 2011. ISBN: 1441982361.

[KPP06] DIMITRIOS S. KOLOVOS; RICHARD F. PAIGE; FIONA A.C. POLACK: "On-Demand Merging of Traceability Links with Models". In: *Proceedings of the 2nd ECMDA Traceability Workshop (ECMDA-TW)*. 2006.

[KPP95] BARBARA KITCHENHAM; LESLEY PICKARD; SHARI LAWRENCE PFLEEGER: "Case Studies for Method and Tool Evaluation". In: *IEEE Software* 12:4 (1995), pp. 52–62. ISSN: 0740-7459. DOI: `10.1109/52.391832`.

[Kru95] PHILIPPE B. KRUCHTEN: "The 4+1 View Model of architecture". In: *IEEE Software* 12:6 (1995), pp. 42–50. ISSN: 0740-7459. DOI: `10.1109/52.469759`.

[KT07] MAYUMI ITAKURA KAMATA; TETSUO TAMAI: "How Does Requirements Quality Relate to Project Success or Failure?" In: *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE)*. Ed. by ALISTAIR SUTCLIFFE. Los Alamitos, USA: IEEE, 2007, pp. 69–78. ISBN: 978-0-7695-2935-6. DOI: `10.1109/RE.2007.31`.

[KV13] Konstantin Kernschmidt; Birgit Vogel-Heuser: "An interdisciplinary SysML based modeling approach for analyzing change influences in production plants to support the engineering". In: *2013 IEEE International Conference on Automation Science and Engineering (CASE)*. 2013, pp. 1113–1118. DOI: `10.1109/CoASE.2013.6654030`.

[Lam78] Leslie Lamport: "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21:7 (1978), pp. 558–565. ISSN: 0001-0782. DOI: `10.1145/359545.359563`.

[LBD+10] Shuhao Li; Sandie Balaguer; Alexandre David; Kim G. Larsen; Brian Nielsen; Saulius Pusinskas: "Scenario-based verification of real-time systems using Uppaal". In: *Formal Methods in System Design* 37:2 (2010), pp. 200–264. ISSN: 0925-9856 (Print), 1572-8102 (Online). DOI: `10.1007/s10703-010-0103-z`.

[LBTA11] Grzegorz Lehmann; Marco Blumendorf; Frank Trollmann; Sahin Albayrak: "Meta-modeling Runtime Models". In: *Proceedings of the 5th International Workshop on Models@run.time (Reports and Revised Selected Papers)*. Ed. by Juergen Dingel; Arnor Solberg. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2011, pp. 209–223. ISBN: 978-3-642-21209-3 (Print), 978-3-642-21210-9 (Online).

[LCD+15] Florent Latombe; Xavier Crégut; Julien DeAntoni; Marc Pantel; Benoît Combemale: "Coping with Semantic Variation Points in Domain-Specific Modeling Languages". In: *Proceedings of the 1st International Workshop on Executable Modeling (EXE'15)*. Ottawa, Canada: CEUR, 2015.

[Lee59] C. Y. Lee: "Representation of Switching Circuits by Binary-Decision Programs". In: *Bell System Technical Journal* 38:4 (1959), pp. 985–999. ISSN: 1538-7305. DOI: `10.1002/j.1538-7305.1959.tb01585.x`.

[LHGO12] Leen Lambers; Stephan Hildebrandt; Holger Giese; Fernando Orejas: "Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case". In: *Proceedings of the 1st International Workshop on Bidirectional Transformations (BX 2012)*. Ed. by Frank Herrmann; Janis Voigtländer. Vol. 49. Electronic Communications of the EASST. 2012. DOI: `10.14279/tuj.eceasst.49.706`.

[LK01] Marc Lettrari; Jochen Klose: "Scenario-Based Monitoring and Testing of Real-Time UML Models". In: *Proceedings of the 4th International Conference on the Unified Modeling Language (≪UML≫ 2001 — The Unified Modeling Language: Modeling Languages, Concepts, and Tools)*. Ed. by Martin Gogolla; Cris Kobryn. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 2001, pp. 317–328. ISBN: 978-3-540-42667-7 (Print), 978-3-540-45441-0 (Online). DOI: `10.1007/3-540-45441-1_24`.

[LLNP09] KIM G. LARSEN; SHUHAO LI; BRIAN NIELSEN; SAULIUS PUSINSKAS: "Verifying Real-Time Systems against Scenario-Based Requirements". In: *Proceedings of the 2nd World Congress on Formal Methods (FM 2009)*. Ed. by ANA CAVALCANTI; DENNIS R. DAMS. Vol. 5850. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2009, pp. 676–691. ISBN: 978-3-642-05088-6 (Print), 978-3-642-05089-3 (Online). DOI: 10.1007/978-3-642-05089-3_43.

[LLNP10] KIM G. LARSEN; SHUHAO LI; BRIAN NIELSEN; SAULIUS PUSINSKAS: "Scenario-based Analysis and Synthesis of Real-time Systems Using UPPAAL". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. 2010, pp. 447–452. ISBN: 978-3-9810801-6-2. DOI: 10.1109/DATE.2010.5457164.

[LM09] DAVID LO; SHAHAR MAOZ: "Mining Hierarchical Scenario-Based Specifications". In: *24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2009, pp. 359–370. DOI: 10.1109/ASE.2009.19.

[MAD09] FRÉDÉRIC MALLET; CHARLES ANDRÉ; JULIEN DEANTONI: "Executing AADL Models with UML/MARTE ". In: *2009 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. Piscataway, USA: IEEE, 2009, pp. 371–376. ISBN: 978-0-7695-3702-3. DOI: 10.1109/ICECCS.2009.10.

[McK16] MCKINSEY: *Automotive revolution – Perspective Towards 2030: How the convergence of disruptive technology-driven trends could transform the auto industry*. 2016.

[McK18] MCKINSEY CENTER FOR FUTURE MOBILITY: *Ready for Inspection – The Automotive Aftermarket in 2030*. 2018.

[MF10] JOHN PAUL MACDUFFIE; TAKAHIRO FUJIMOTO: "Why Dinosaurs Will Keep Ruling the Auto Industry". In: *Harvard Business Review* 88:6 (2010), pp. 23–25.

[MGP09] PATRICK MÄDER; ORLENA C. Z. GOTEL; ILKA PHILIPPOW: "Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice". In: *Proceedings of the 5th ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. Piscataway, USA: IEEE, 2009, pp. 21–25. ISBN: 978-1-4244-3741-2. DOI: 10.1109/TEFSE.2009.5069578.

[MH06] SHAHAR MAOZ; DAVID HAREL: "From Multi-modal Scenarios to Code: Compiling LSCs into AspectJ". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Ed. by MICHAL YOUNG; PREMKUMAR DEVANBU. New York, USA: ACM, 2006, pp. 219–230. ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181802.

[MH11] SHAHAR MAOZ; DAVID HAREL: "On tracing reactive systems". In: *Software & Systems Modeling* 10:4 (2011), pp. 447–468. ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: 10.1007/s10270-010-0151-2.

[Mil71] ROBIN MILNER: "An algebraic definition of simulation between programs". In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. 1971, pp. 481–489.

[MKH07] SHAHAR MAOZ; ASAF KLEINBORT; DAVID HAREL: "Towards Trace Visualization and Exploration for Reactive Systems". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. Ed. by PHILIP COX; JOHN HOSKING. Los Alamitos, USA: IEEE, 2007, pp. 153–156. ISBN: 978-0-7695-2987-5. DOI: `10.1109/VLHCC.2007.27`.

[MMS13] SAAD MUBEEN; JUKKA MÄKI-TURJA; MIKAEL SJÖDIN: "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`.

[MMS14] SAAD MUBEEN; JUKKA MÄKI-TURJA; MIKAEL SJÖDIN: "Communications-oriented development of component-based vehicular distributed real-time embedded systems". In: *Journal of Systems Architecture* 60:2 (2014), pp. 207–220. ISSN: 1383-7621. DOI: `10.1016/j.sysarc.2013.10.008`.

[MNL+16] SAAD MUBEEN; THOMAS NOLTE; JOHN LUNDBÄCK; MATTIAS GÅLNANDER; KURT-LENNART LUNDBÄCK: "Refining Timing Requirements in Extended Models of Legacy Vehicular Embedded Systems Using Early End-to-end Timing Analysis". In: *Information Technology: New Generations*. Ed. by SHAHRAM LATIFI. Vol. 448. Advances in Intelligent Systems and Computing (AISC). Cham, Switzerland: Springer, 2016, pp. 497–508. ISBN: 978-3-319-32466-1 (Print), 978-3-319-32467-8 (Online). DOI: `10.1007/978-3-319-32467-8_44`.

[MNS+17] SAAD MUBEEN; THOMAS NOLTE; MIKAEL SJÖDIN; JOHN LUNDBÄCK; KURT-LENNART LUNDBÄCK: "Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints". In: *Software & Systems Modeling* (2017). ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: `10.1007/s10270-017-0579-8`.

[MPA09] FRÉDÉRIC MALLET; MARIE-AGNÈS PERALDI-FRATI; CHARLES ANDRÉ: "MARTE CCSL to Execute East-ADL Timing Requirements". In: *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'09)*. Los Alamitos, USA: IEEE, 2009, pp. 249–253. ISBN: 978-0-7695-3573-9. DOI: `10.1109/ISORC.2009.18`.

[MPR07] PATRICK MÄDER; ILKA PHILIPPOW; MATTHIAS RIEBISCH: "Customizing Traceability Links for the Unified Process". In: *Proceedings of the 3rd International Conference on Quality of Software Architectures (QoSA) – Revised Selected Papers*. Ed. by SVEN OVERHAGE; CLEMENS A. SZYPERSKI; RALF REUSSNER; JUDITH A. STAFFORD. Berlin/Heidelberg: Springer, 2007, pp. 53–71. ISBN: 978-3-540-77617-8 (Print), 978-3-540-77619-2 (Online). DOI: `10.1007/978-3-540-77619-2_4`.

[MS16] SALOME MARO; JAN-PHILIPP STEGHÖFER: "Capra: A Configurable and Extendable Traceability Management Tool". In: *Proceedings of the 24th IEEE International Requirements Engineering Conference (RE)*. Los Alamitos, USA: IEEE, 2016, pp. 407–408. ISBN: 978-1-5090-4121-3. DOI: `10.1109/RE.2016.19`.

[MSN+15] SAAD MUBEEN; MIKAEL SJÖDIN; THOMAS NOLTE; JOHN LUNDBÄCK; MATTIAS GÅLNANDER; KURT-LENNART LUNDBÄCK: "End-to-End Timing Analysis of Black-Box Models in Legacy Vehicular Distributed Embedded Systems". In: *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications (RTSCA)*. Los Alamitos, USA: IEEE Computer Society, 2015, pp. 149–158. ISBN: 978-1-4673-7855-0. DOI: `10.1109/RTCSA.2015.24`.

[NE00] BASHAR NUSEIBEH; STEVE EASTERBROOK: "Requirements Engineering: A Roadmap". In: *Proceedings of the Conference on The Future of Software Engineering (FOSE)*. Ed. by ANTHONY C. W. FINKELSTEIN. New York, USA: ACM, 2000, pp. 35–46. ISBN: 1-58113-253-0. DOI: `10.1145/336512.336523`.

[NIE+17] ARNE NOYER; PADMA IYENGHAR; JOACHIM ENGELHARDT; ELKE PULVERMUELLER; GERT BIKKER: "A model-based framework encompassing a complete workflow from specification until validation of timing requirements in embedded software systems". In: *Software Quality Journal* 25:3 (2017), pp. 671–701. ISSN: 1573-1367. DOI: `10.1007/s11219-016-9323-9`. URL: `https://doi.org/10.1007/s11219-016-9323-9`.

[NMH08] MIKAEL NOLIN; JUKKA MÄKI-TURJA; KAJ HÄNNINEN: "Achieving Industrial Strength Timing Predictions of Embedded System Behavior". In: *Proceedings of the 2008 International Conference on Embedded Systems & Applications (ESA)*. Ed. by HAMID R. ARABNIA; MUN YOUNGSONG. CSREA Press, 2008, pp. 173–178. ISBN: 1-60132-065-5.

[NT09] JOAQUÍN NICOLÁS; AMBROSIO TOVAL: "On the Generation of Requirements Specifications from Software Engineering Models: A Systematic Literature Review". In: *Information and Software Technology* 51:9 (2009), pp. 1291–1307. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2009.04.001`.

[PAA+15] ART PYSTER; RICK ADCOCK; MARK ARDIS; ROB CLOUTIER; DEVANANDHAM HENRY; LINDA LAIRD; HAROLD 'BUD' LAWSON; MICHAEL PENNOTTI; KEVIN SULLIVAN; JON WADE: "Exploring the Relationship between Systems Engineering and Software Engineering". In: *Processings of the 2015 Conference on Systems Engineering Research*. Vol. 44. Procedia Computer Science. 2015, pp. 708–717. DOI: `10.1016/j.procs.2015.03.016`.

[PB00] PETER PUSCHNER; ALAN BURNS: "Guest Editorial: A Review of Worst-Case Execution-Time Analysis". In: *Real-Time Systems* 18:2 (2000), pp. 115–128. ISSN: 1573-1383. DOI: `10.1023/A:1008119029962`.

[PBKS07] ALEXANDER PRETSCHNER; MANFRED BROY; INGOLF KRÜGER; THOMAS STAUNER: "Software Engineering for Automotive Systems: A Roadmap". In: *Proceedings of the Conference on Future of Software Engineering (FOSE)*. Ed. by LIONEL C. BRIAND; ALEXANDER L. WOLF. Washington, USA: IEEE, 2007, pp. 55–71. ISBN: 0-7695-2829-5. DOI: `10.1109/FOSE.2007.22`.

[PD11] MARIE-AGNÈS PERALDI-FRATI; JULIEN DEANTONI: "Scheduling Multi Clock Real Time Systems: From Requirements to Implementation". In: *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'11)*. Piscataway, USA: IEEE, 2011, pp. 50–57. ISBN: 978-1-61284-433-6. DOI: `10.1109/ISORC.2011.16`.

[PDK+11] RICHARD F. PAIGE; NIKOLAOS DRIVALOS; DIMITRIOS S. KOLOVOS; KIRAN J. FERNANDES; CHRISTOPHER POWER; GORAN K. OLSEN; STEFFEN ZSCHALER: "Rigorous identification and encoding of trace-links in model-driven engineering". In: *Software & Systems Modeling* 10:4 (2011), pp. 469–487. ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: `10.1007/s10270-010-0158-8`.

[Poo10] RADHA POOVENDRAN: "Cyber-Physical Systems: Close Encounters Between Two Parallel Worlds". In: *Proceedings of the IEEE* 98:8 (2010). Point of View, pp. 1363–1366. ISSN: 0018-9219. DOI: `10.1109/JPROC.2010.2050377`.

[PR11] KLAUS POHL; CHRIS RUPP: *Requirements Engineering Fundamentals*. 1st edition. Santa Barbara, USA: Rocky Nook, 2011. ISBN: 978-1-933952-81-9.

[RE93] BALASUBRAMANIAM RAMESH; MICHAEL EDWARDS: "Issues in the Development of a Requirements Traceability Model". In: *Proceedings of the IEEE International Symposium on Requirements Engineering*. Los Alamitos, USA: IEEE, 1993, pp. 256–259. ISBN: 0-8186-3120-1. DOI: `10.1109/ISRE.1993.324849`.

[RH08] PER RUNESON; MARTIN HÖST: "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14:2 (2008), pp. 131–164. ISSN: 1382-3256 (Print), 1573-7616 (Online). DOI: `10.1007/s10664-008-9102-8`.

[RHAR12] PER RUNESON; MARTIN HÖST; RAINER AUSTEN; BJÖRN REGNELL: *Case Study Research in Software Engineering – Guidelines and Examples*. 1st edition. Hoboken, USA: Wiley, 2012. ISBN: 9781118104354.

[SAÅ+04] LUI SHA; TAREK ABDELZAHER; KARL-ERIK ÅRZÉN; ANTON CERVIN; THEODORE BAKER; ALAN BURNS; GIORGIO BUTTAZZO; MARCO CACCAMO; JOHN LEHOCZKY; ALOYSIUS K. MOK: "Real Time Scheduling Theory: A Historical Perspective". In: *Real-Time Systems* 28:2-3 (2004), pp. 101–155. ISSN: 0922-6443. DOI: `10.1023/B:TIME.0000045315.61234.1e`.

[Sch95] ANDY SCHÜRR: "Specification of Graph Translators with Triple Graph Grammars". In: *Graph-Theoretic Concepts in Computer Science*. Ed. by ERNST W. MAYR. Vol. 903. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 1995, pp. 151–163. ISBN: 3540590714. DOI: `10.1007/3-540-59071-4_45`.

[ŠCV13] IVAN ŠVOGOR; IVICA CRNKOVIĆ; NEVEN VRČEK: "An Extended Model for Multi-Criteria Software Component Allocation on a Heterogeneous Embedded Platform". In: *Journal of Computing and Information Technology* 21:4 (2013), pp. 211–222. DOI: `10.2498/cit.1002284`.

[SDP10] Ernst Sikora; Marian Daun; Klaus Pohl: “Supporting the Consistent Specification of Scenarios across Multiple Abstraction Levels”. In: *Requirements Engineering: Foundation for Software Quality (REFSQ)*. Ed. by Roel Wieringa; Anne Persson. Vol. 6182. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg: Springer, 2010, pp. 45–59. isbn: 978-3-642-14191-1 (Print), 978-3-642-14192-8 (Online). doi: 10.1007/978-3-642-14192-8_6.

[SG14] Bran Selic; Sébastien Gérard: *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE – Developing Cyber-Physical Systems*. Amsterdam, Netherlands: Elsevier, 2014. isbn: 978-0-12-416619-6.

[She96] Sarah A. Sheard: “Twelve Systems Engineering Roles”. In: *INCOSE International Symposium* 6:1 (1996), pp. 478–485. issn: 2334-5837. doi: 10.1002/j.2334-5837.1996.tb02042.x.

[Sta73] Herbert Stachowiak: *Allgemeine Modelltheorie*. Vienna, Austria: Springer, 1973. isbn: 978-3211811061.

[Sta88] John A. Stankovic: “Misconceptions About Real-Time Computing – A Serious Problem for Next-Generation Systems”. In: *Computer* 21:10 (1988), pp. 10–19. issn: 0018-9162. doi: 10.1109/2.7053.

[STP12] Ernst Sikora; Bastian Tenbergen; Klaus Pohl: “Industry Needs and Research Directions in Requirements Engineering for Embedded Systems”. In: *Requirements Engineering* 17 (2012), pp. 57–78. issn: 0947-3602. doi: 10.1007/s00766-011-0144-x.

[SUB08] German Sibay; Sebastian Uchitel; Victor Braberman: “Existential Live Sequence Charts Revisited”. In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 41–50. doi: 10.1145/1368088.1368095.

[TBW95] Ken Tindell; Alan Burns; Andy J. Wellings: “Analysis of hard real-time communications”. In: *Real-Time Systems* 9:2 (1995), pp. 147–171. issn: 1573-1383. doi: 10.1007/BF01088855.

[TC94] Ken Tindell; John Clark: “Holistic schedulability analysis for distributed hard real-time systems”. In: *Microprocessing and Microprogramming* 40:2 (1994), pp. 117–134. issn: 0165-6074. doi: 10.1016/0165-6074(94)90080-9.

[Thr10] Kleanthis Thramboulidis: “The 3+1 SysML View-Model in Model Integrated Mechatronics”. In: *Journal of Software Engineering and Applications* 3:2 (2010), pp. 109–118. doi: 10.4236/jsea.2010.32014.

[THW94] Ken W. Tindell; Hans Hansson; Andy J. Wellings: “Analysing real-time communications: controller area network (CAN)”. In: *Proceedings of the 1994 Real-Time Systems Symposium (RTTS)*. IEEE, 1994, pp. 259–263. doi: 10.1109/REAL.1994.342710.

[TL89] Mark R. Tuttle; Nancy A. Lynch: “An Introduction to Input/Output Automata”. In: *CWI Quarterly* 2:3 (1989), pp. 219–246. issn: 0922-5366.

[VBK10] K. Venkatesh Prasad; Manfred Broy; Ingolf Krüger: "Scanning Advances in Aerospace & Automobile Software Technology". In: *Proceedings of the IEEE* 98:4 (2010), pp. 510–514. ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2041835.

[VEFR12] Andreas Vogelsang; Sebastian Eder; Martin Feilkas; Daniel Ratiu: "Functional Viewpoint". In: *Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology*. Ed. by Klaus Pohl; Harald Hönninger; Reinhold E. Achatz; Manfred Broy. Berlin/Heidelberg: Springer, 2012. Chap. 5, pp. 69–83. ISBN: 978-3-642-34613-2 (Print), 978-3-642-34614-9 (Online). DOI: 10.1007/978-3-642-34614-9_5.

[WAB+10] Adam Wyner; Krasimir Angelov; Guntis Barzdins; Danica Damljanovic; Brian Davis; Norbert E. Fuchs; Stefan Hoefler; Ken Jones; Kaarel Kaljurand; Tobias Kuhn; Martin Luts; Jonathan Pool; Mike Rosner; Rolf Schwitter; John Sowa: "On Controlled Natural Languages: Properties and Prospects". In: *Controlled Natural Language*. Ed. by Norbert E. Fuchs. Vol. 5972. Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer, 2010, pp. 281–289. ISBN: 978-3-642-14417-2. DOI: 10.1007/978-3-642-14418-9_17.

[Wan04] Farn Wang: "Formal Verification of Timed Systems: A Survey and Perspective". In: *Proceedings of the IEEE* 92:8 (2004), pp. 1283–1305. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.831197.

[WEE+08] Reinhard Wilhelm; Jakob Engblom; Andreas Ermedahl; Niklas Holsti; Stephan Thesing; David Whalley; Guillem Bernat; Christian Ferdinand; Reinhold Heckmann; Tulika Mitra; Frank Mueller; Isabelle Puaut; Peter Puschner; Jan Staschulat; Per Stenström: "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools". In: *ACM Transactions on Embedded Computing Systems* 7:3 (2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389.

[Wei16] Tim Weilkiens: *SYSMOD – The Systems Modeling Toolbox (Version 4.1): Pragmatic MBSE with SysML*. 2nd edition. Victoria, Canada: Leanpub, 2016. ISBN: 978-3-9817875-9-7.

[WP10] Stefan Winkler; Jens von Pilgrim: "A Survey of Traceability in Requirements Engineering and Model-driven Development". In: *Software & Systems Modeling* 9:4 (2010), pp. 529–565. ISSN: 1619-1366 (Print), 1619-1374 (Online). DOI: 10.1007/s10270-009-0145-0.

[WRF+15] David D. Walden; Garry J. Roedler; Kevin J. Forsberg; R. Douglas Hamelin; Thomas M. Shortell, eds.: *Systems Engineering Handbook – A Guide for System Lifecycle Processes and Activities*. 4th edition. INCOSE–TP–2003–002–04. Hoboken, USA: Wiley, 2015. ISBN: 9781118999400.

[WT04] Shuhua Wang; Grace Tsai: "Specification and Timing Analysis of Real-Time Systems". In: *Real-Time Systems* 28:1 (2004), pp. 69–90. ISSN: 1573-1383. DOI: 10.1023/B:TIME.0000033379.78994.1a.

[XJMZ11] Xiaohong Chen; Jing Liu; Frédéric Mallet; Zhi Jin: "Modeling Timing Requirements in Problem Frames Using CCSL". In: *18th Asia-Pacific Software Engineering Conference (APSEC)*. Ed. by Dan Thu Tran. Piscataway, USA: IEEE, 2011, pp. 381–388. isbn: 978-1-4577-2199-1. doi: `10.1109/APSEC.2011.30`.

[YTB+11] Huafeng Yu; Jean-Pierre Talpin; Loïc Besnard; Thierry Gautier; Hervé Marchand; Paul Le Guernic: "Polychronous controller synthesis from Marte CCSL timing specifications". In: *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Piscataway, USA: IEEE, 2011, pp. 21–30. isbn: 978-1-4577-0116-0. doi: `10.1109/MEMCOD.2011.5970507`.

[ZSPK03] Andrea Zisman; George Spanoudakis; Elena Pérez-Miñana; Paul Krause: "Tracing Software Requirements Artefacts". In: *Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP)*. Ed. by Ban Al-Ani; Hamid R. Arabnia; Youngsong Mun. CSREA Press, 2003, pp. 448–455. isbn: 1-932415-20-3.

Standards and Specifications

[ASIG17] Automotive Special Interest Group / VDA QMC Working Group 13: *Automotive SPICE Process Reference and Assessment Model*. Version 3.1. 2017.

[AUTOSAR] Automotive Open System Architecture: *AUTomotive Open System ARchitecture (AUTOSAR) Standard*. url: `http://www.autosar.org`.

[EAST13] EAST-ADL Association: *EAST-ADL Domain Model Specification*. Version V2.1.12. 2013.

[IEC04] International Electrotechnical Commission (IEC) / Institute of Electrical and Electronics Engineers (IEEE): *IEC/IEEE Behavioural Languages – Part 4: Verilog Hardware Description Language (Adoption of IEEE Std 1364-2001)*. IEC 61691-4:2004(E) / IEEE 1364-2001(E). 2004. isbn: 2-8318-7675-3. doi: `10.1109/IEEESTD.2004.95753`.

[IEC10] International Electrotechnical Commission (IEC): *Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC 61508:2010. 2010.

[ISO05] International Organization for Standardization: *Road vehicles – Open interface for embedded automotive applications – Part 3: OSEK/VDX Operating System (OS)*. ISO 17356-3:2005. 2005.

[ISO11] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) / Institute of Electrical and Electronics Engineers (IEEE): *Systems and software engineering – Architecture description*. ISO/IEC/IEEE 42010:2011(E). 2011. isbn: 978-0-7381-7142-5. doi: `10.1109/IEEESTD.2011.6129467`.

[ISO17] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) / Institute of Electrical and Electronics Engineers (IEEE): *Systems and software engineering – Vocabulary*. ISO/IEC/IEEE 24765-2017(E). New York, USA, 2017. isbn: 978-1-5044-4118-6. doi: `10.1109/IEEESTD.2017.8016712`.

[ISO18a] International Organization for Standardization (ISO): *Road vehicles – Functional safety*. ISO 26262:2018. 2018.

[ISO18b] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC) / Institute of Electrical and Electronics Engineers (IEEE): *Systems and software engineering – Life cycle processes – Requirements engineering*. ISO/IEC/IEEE 29148-2018. 2018. isbn: 978-1-5044-5302-8. doi: `10.1109/IEEESTD.2018.8559686`.

[ITU11] ITU Telecommunication Standardization Sector: *ITU-T Recommendation Z.120 (02/2011): Message Sequence Chart (MSC)*. 02/2011. Z.120. 2011.

[ITU16] ITU Telecommunication Standardization Sector: *ITU-T Recommendation Z.101 (04/2016): Specification and Description Language – Basic SDL-2010*. 2016.

[OMG11] Object Management Group (OMG): *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. OMG Document Number: formal/2011-06-02. Version 1.1. 2011. url: `http://www.omg.org/spec/MARTE/1.1/`.

[OMG12] Object Management Group (OMG): *OMG SysML-Modelica Transformation (SyM)*. OMG Document Number: formal/2012-11-09. Version 1.0. 2012. url: `http://www.omg.org/spec/SyM/1.0/`.

[OMG14a] Object Management Group (OMG): *OMG Object Constraint Language (OCL)*. OMG Document Number: formal/2014-02-03. Version 2.4. 2014. url: `http://www.omg.org/spec/OCL/2.4`.

[OMG14b] Object Management Group (OMG): *Business Process Model and Notation (BPMN)*. OMG Document Number: formal/2013-12-09. Version 2.0.2. 2014. url: `http://www.omg.org/spec/BPMN/2.0.2/`.

[OMG16] Object Management Group (OMG): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. OMG Document Number: formal/2016-06-03. Version 1.3. 2016. url: `http://www.omg.org/spec/QVT/1.3/`.

[OMG17a] Object Management Group (OMG): *OMG Systems Modeling Language (OMG SysML)*. OMG Document Number: formal/2017-05-01. Version 1.5. 2017. url: `http://www.omg.org/spec/SysML/1.5/`.

[OMG17b] Object Management Group (OMG): *OMG Unified Modeling Language (OMG UML)*. OMG Document Number: formal/2017-12-05. Version 2.5.1. 2017. url: `http://www.omg.org/spec/UML/2.5.1/`.

[RTCA11] RADIO TECHNICAL COMMISSION FOR AERONAUTICS (RTCA): *Software Considerations in Airborne Systems and Equipment Certification*. DO-178C. 2011.

[VDI04] ASSOCIATION OF GERMAN ENGINEERS (VEREIN DEUTSCHER INGENIEURE, VDI): *VDI Guideline 2206 – Design Methodology for Mechatronic Systems*. Berlin, Germany: Beuth, 2004.

Research Projects

[A4P] *European research project "Amalthea4Public": An open platform project for embedded multicore systems*. Last accessed April 2019. URL: `http://www.amalthea-project.org/`.

[itsOWL] *German leading-edge cluster "Intelligent Technical Systems OstWestfalenLippe (it's OWL)"*. Last accessed April 2019. URL: `https://www.its-owl.com/`.

[itsOWL-SE] *it's OWL cross-sectional project "Systems Engineering"*. Last accessed April 2019. URL: `https://www.its-owl.com/projects/cross-sectional-projects/details/systems-engineering/`.

[SPES2020] *German research project "Software Plattform Embedded Systems 2020 (SPES 2020)"*. Last accessed April 2019. URL: `http://spes2020.informatik.tu-muenchen.de/spes-home.html`.

Tool Suites and Tool Frameworks

[CAPRA] *Capra Traceability Management Tool*. Last accessed January 2019. URL: `http://projects.eclipse.org/projects/modeling.capra`.

[GEMOC] *Eclipse GEMOC Studio*. Last accessed January 2019. URL: `https://projects.eclipse.org/projects/modeling.gemoc`.

[PAPYRUS] *Papyrus Modeling Environment*. Last accessed January 2019. URL: `http://www.eclipse.org/papyrus`.

[EMF] *Eclipse Modeling Framework (EMF)*. Last accessed January 2019. URL: `http://www.eclipse.org/modeling/emf`.

[MUML] *MechatronicUML Tool Suites*. Last accessed January 2019. URL: `http://www.mechatronicuml.org/en`.

[QVTo] *Eclipse QVT Operational*. Last accessed January 2019. URL: `http://projects.eclipse.org/projects/modeling.mmt.qvt-oml`.

[ST-MSD] *ScenarioTools MSD*. Last accessed January 2019. URL: `http://scenariotools.org/projects2/msd`.

[T^2] *TimeSquare Model Development Kit*. Last accessed January 2019. URL: `http://timesquare.inria.fr/`.

List of Figures

List of Tables

List of Algorithms

Listings

Appendices

A

Supplementary Material for the Transition Technique from MBSE to SwRE

A.1 Guidelines for Manual MSD Refinement

This appendix presents the guidelines for the manual refinement of MSDs based on informal or semi-formal information in CONSENS system models (cf. Section 3.5.1). These guidelines are exemplarily applied in Appendix A.2.2.2.

The Guidelines 1a and 1b (cf. Figure 3.15 and Figure A.1, respectively) focus on the specification of additional MSDs, whereas Guideline 2 (cf. Figure A.2) focuses on the distinct specification of trigger and execution behavior. Subsequently, the Guidelines 3a and 3b (cf. Figure A.3 and Figure A.4, respectively) present best practices on the specification of temperatures and execution kinds, which are not automatically derived from CONSENS. Finally, the Guidelines 4a and 4b (cf. Figure A.5 and Figure A.6, respectively) focus on the specification of conditional behavior like conditions and real-time requirements, which are again not subject to the automatic part of our technique for the transition from CONSENS to MSD specifications.

1. Specify Additional MSDs
b) Add Requirement MSDs

I. Goal

Requirement MSDs specify the actual requirements on the SUD. The goal of this guideline is the manual specification of additional requirement MSDs that are not automatically derived because they have no representation in the partial model *Behavior – Sequences*.

II. Description of Elements

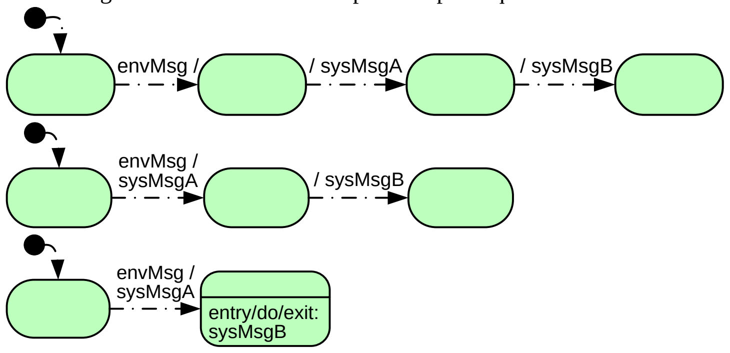
The requirements on the SUD are specified by means of requirement MSDs.

Similar to assumption MSDs but taking the perspective of the SUD, requirement MSDs specify the following aspects:
- Fixed order of system events*
- Conditional behavior: System events may occur iff certain conditions hold (cf. Guideline 4.a) or under real-time constraints (cf. Guideline 4.b)
- Forbidden system events or event sequences (cf. Guideline 4.a)

III. Guidelines

General:
- Partial models with informal/semi-formal information relevant to this refinement step: *Behavior – States*, *Application Scenarios*, and *Requirements*.
- Investigate *Behavior – States* for behavior that is not specified by means of *Behavior – Sequences* and hence is not part of automatically derived Requirement MSDs.
- Check consistency of the execution kinds of messages w.r.t. assumption MSDs (cf. Guideline 1.a).

* Determine information about fixed system event sequences:
- Investigate *Application Scenarios* for sequential system behavior, keywords: “after”, “afterward”, “subsequently”, “following”.
- Investigate *Behavior – States* for plain output sequences:



IV. Examples

* Plain fixed order:

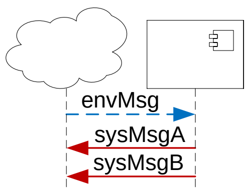


Figure A.1: Guideline 1b—Specify additional MSDs: Add requirement MSDs

2. Specify Trigger and Execution Behavior

I. Goal

The actual execution behavior of application scenarios and MSD use cases, respectively, is often triggered by different situations. The goal of this guideline is to…

1) …identify these trigger situations and formalize them by means of additional MSDs that are not automatically derived because they have no representation in the partial model *Behavior – Sequences* as well as to…
2) …rearrange the manually added trigger behavior and the combined trigger/execution behavior that is automatically derived from the exemplary *Behavior – Sequences* such that the execution behavior is not specified redundantly.

II. Description of Elements

The trigger and the execution behavior of MSD use cases is specified by means of requirement MSDs. The focus is on separating both kinds of behavior in such a way that…

- * …internal system messages of multiple requirement MSDs specifying the trigger behavior…
- ** …can be unified as minimal events in as few as possible requirement MSDs specifying the execution behavior. If such MSDs are automatically derived, this procedure results in moving messages to dedicated MSDs specifying only the trigger behavior.

III. Guidelines

- Partial models with informal/semi-formal information relevant to this refinement step: *Application Scenarios* and *Behavior – States.*
- Investigate *Application Scenarios* for multiple trigger situations resulting in the same intended behavior (trigger situations and intended behavior can also overlap between multiple *Application Scenarios*):

Application Scenario

Situation: Either envMsgA or envMsgB occurs.
Intended behavior: sysMsg has to be sent.

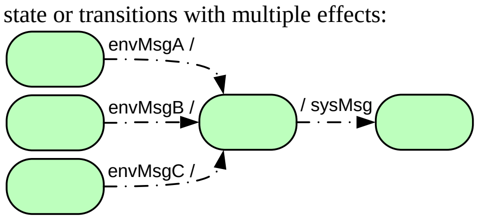
Application Scenario 1

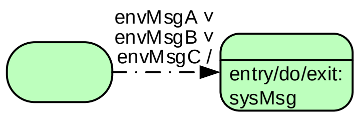
Situation: envMsgA occurs.
Intended behavior: sysMsg has to be sent.

Application Scenario 2

Situation: envMsgB occurs.
Intended behavior: sysMsg has to be sent.

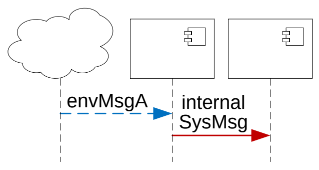
- Investigate *Behavior – States* for multiple transitions leading to the same state or transitions with multiple effects:

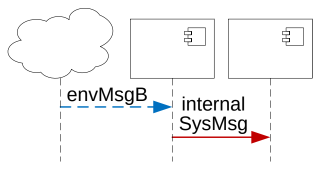


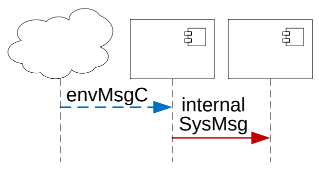


IV. Examples

* Specifying trigger behavior:







** Specifying execution behavior:

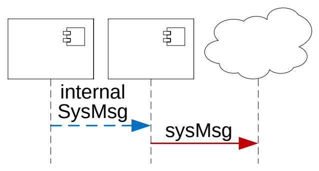


Figure A.2: Guideline 2—Specify trigger and execution behavior

3. Specify Temperatures and Execution Kinds

a) Add Temperatures

I. Goal

Modal messages differ in their temperature, which is not automatically derived by our transition technique since there is no automatically processable information about this aspect in CONSENS system models. Thus, the goal of this guideline is to determine temperatures in the course of refining automatically derived messages or in the course of specifying a message manually.

II. Description of Elements

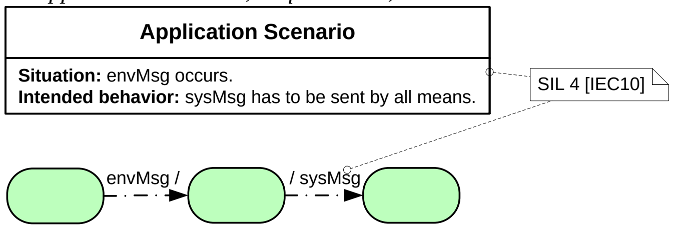
The temperature of messages is distinguished into:

- Cold: The MSD is discarded if an event occurs that is specified in the MSD to occur earlier or later.
- Hot: No message must occur that the MSD specifies to occur only earlier or later (safety).

III. Guidelines

General:

- Partial models with informal information relevant to this refinement step: *Application Scenarios*, *Requirements*, and *Behavior – States*.


Application Scenario

Situation: envMsg occurs.
Intended behavior: sysMsg has to be sent by all means.

Determine cold messages:

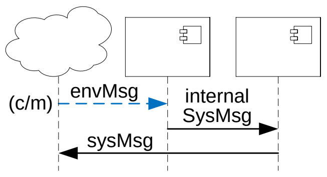
- The first message of an MSD is always cold.
- Investigate *Application Scenarios* and *Requirements* for conditional or temporally dependent events (keywords: “if”, “in case”, “when”, “once”).
- The part “Situation” of an *Application Scenario* typically can be formalized by means of cold messages (cf. envMsg in the “Situation” part of the *Application Scenario* in the figure).

Determine hot messages:

- Investigate *Application Scenarios* and *Requirements* for safety-critical events (keywords: “obligatory”, “absolutely”, “by all means”).
- The part “Intended Behavior” of an Application Scenario often can be formalized by means of hot messages (cf. sysMsg in the “Situation” part of the *Application Scenario* in the figure).
- Safety standards (e.g., [IEC10; ISO18a; RTCA11]) classify the safety-criticality by means of safety integrity levels (SILs). System modeling languages often provide means [EAST13] or are extended [Foc16] to annotate SILs to model elements, like *Application Scenarios*, *Requirements*, and *Behavior – States*. Investigate these partial models of the CONSENS system model for high SILs to determine safety-critical requirements and event sequences (cf. SIL annotation for the *Application Scenario* and the *Behavior – States* transition in the figure).

IV. Examples

Automatically derived MSD without any temperatures for messages except for the first message:



Declaring remaining messages as hot:

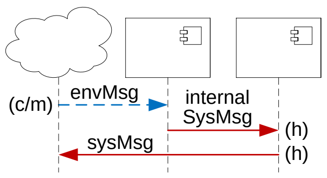


Figure A.3: Guideline 3a—Specify temperatures and execution kinds: Add temperatures

3. Specify Temperatures and Execution Kinds
b) Add Execution Kinds

I. Goal

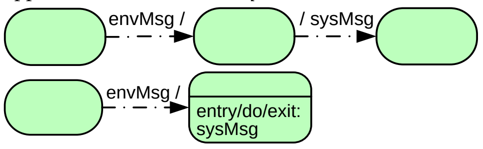
Modal messages differ in their execution kind, which is not automatically derived by our transition technique since there is no automatically processable information about this aspect in CONSENS system models. Thus, the goal of this guideline is to determine execution kinds in the course of refining automatically derived messages or in the course of specifying a message manually.

II. Description of Elements

The execution kind of messages can be distinguished into:
- Monitored: The corresponding event may or may not occur.
- Executed: The corresponding event must eventually occur (liveness).

III. Guidelines

General:
- Partial models with informal information relevant to this refinement step: *Application Scenarios*, *Requirements*, and *Behavior – States*.

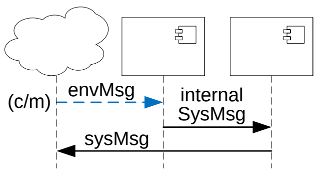


Determine monitored messages:
- The first message of an MSD is always monitored.
- Investigate *Application Scenarios* and *Requirements* for possibly occurring events (keywords: “may”, “optional”, “optionally”, “possibly”, “potentially”, “maybe”, “perhaps”, “in case of”, “as may be the case”).
- The part “Situation” of an Application Scenario typically can be formalized by means of monitored messages.
- Typically, environment messages are monitored in requirement MSDs since the SUD has no influence on the environment behavior. Consequently, trigger events in the *Behavior – States* (cf. envMsg in *Behavior – States* figures) typically result in monitored environment messages.* The consistency with the execution kind of the corresponding messages in assumption MSDs has to be checked (cf. Guideline 1.a, [BGP13]): System messages are typically monitored in assumption MSDs.
- There are MSDs that monitor messages that are executed in other MSDs to specify dedicated reactions or constraints (cf. Guideline 4b, [*HFK+16]). These messages are typically monitored, too.**

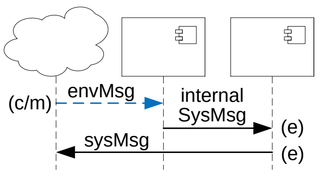
Determine executed messages:
- Investigate *Application Scenarios* and *Requirements* for mandatory events (keywords: “must”, “shall”, “has to”, “eventually”).
- The part “Intended Behavior” of an Application Scenario typically can be formalized by means of executed messages.
- Typically, for each operation there is one system message that is executed in at least one of the requirement MSDs of an MSD use case. Consequently, effect events in the *Behavior – States* (cf. sysMsg in *Behavior – States* figures) typically have to correspond with at least one executed system message to the environment.* Furthermore, the consistency with the execution kind of the corresponding messages in assumption MSDs has to be checked (cf. Guideline 1.a, [BGP13]): Environment messages are typically executed in assumption MSDs.

IV. Examples

Automatically derived MSD without any execution kinds for messages except for the first message:



* Declaring remaining messages as executed (both messages have to be executed in at least one of the requirement MSDs of the MSD use case, sysMsg is moreover a system message sent to the environment):



** Monitoring messages that are executed in another MSD to specify dedicated reactions or constraints:

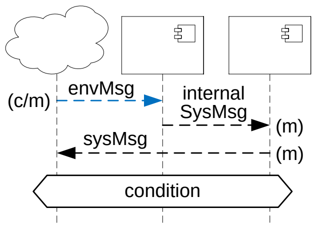


Figure A.4: Guideline 3b—Specify temperatures and execution kinds: Add execution kinds

4. Specify Conditional Behavior
a) Add Conditions

I. Goal

Modal conditions are not automatically derived by our transition technique since there is no automatically processable information about this aspect in CONSENS system models. Thus, the goal of this guideline is to determine conditions and their temperatures in the course of refining automatically derived MSDs or in the course of specifying an MSD manually (cf. Guidelines 1a/b).

II. Description of Elements

Conditions contain OCL expressions that evaluate to a Boolean value. The cut progresses beyond an enabled condition if its expression evaluates to `true`. The temperature of conditions is distinguished into:

- Cold: The MSD is discarded if the condition expression evaluates to `false`.
- Hot: A safety violation occurs if the condition expression evaluates to `false`.

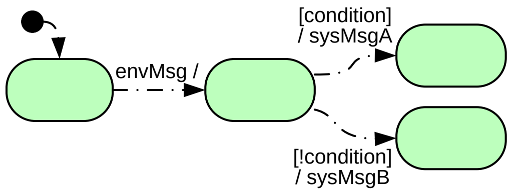
III. Guidelines

General:

- Partial models with informal information relevant to this refinement step: *Application Scenarios*, *Requirements*, and *Behavior – States*.
- If conditions refer to parameters of parameterized messages or setter messages and these parameters were not derived automatically, it is necessary to extend the corresponding operations manually (cf. [*HBM+16, *HBM+15]).

Determine cold conditions:

- Investigate *Application Scenarios* and *Requirements* for conditional behavior (keywords: “if”, “in case”) (cf. [*HBM+16, *HBM+15]).
- Investigate *Behavior – States* for transitions with guards:*

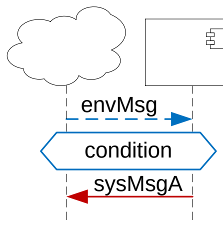


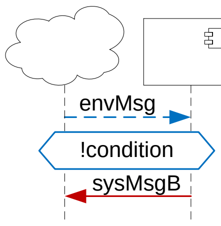
Determine hot conditions:

- Investigate *Application Scenarios* and *Requirements* for safety-critical behavior (keywords: “obligatory”, “absolutely”, “by all means”) or forbidden concrete event sequences (keywords: “must not”, “forbidden”, “not allowed to”).
- Investigate *Application Scenarios*, *Requirements*, *Behavior – States* for annotated high SILs (cf. Guideline 3a).
- Investigate *Application Scenarios* and *Requirements* for forbidden event sequences** or forbidden events***. A forbidden event sequence describes one fix event sequence leading to a violation, whereas a forbidden event describes one particular forbidden event in a certain context specifiable with a negate fragment.

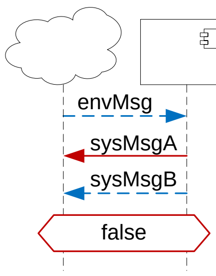
IV. Examples

* Conditional system behavior corresponding to guards in *Behavior – States* (sysMsgA has to be sent if condition holds, and sysMsgB has to be sent otherwise):





** Forbidden event sequence (sysMsgB is not allowed to occur after the occurrence of sysMsgA):



*** Forbidden events (as long as the MSD is active and the enabled message is neither sysMsgA nor sysMsgB, sysMsgA will lead to a cold violation and sysMsgB to a safety violation (cf. [*BGH+14]):

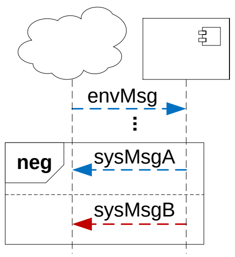


Figure A.5: Guideline 4a—Specify conditional behavior: Add conditions

4. Specify Conditional Behavior
b) Add Real-time Requirements

I. Goal

Real-time requirements (i.e., clock resets combined with time conditions) are not automatically derived by our transition technique since there is no automatically processable information about this aspect in CONSENS system models. Thus, the goal of this guideline is to determine clock resets and time conditions and their temperatures in the course of refining automatically derived MSDs or in the course of specifying an MSD manually (cf. Guidelines 1a/b).

II. Description of Elements

Time conditions contain Boolean expressions of the form $x \bullet expr$ where x is a clock variable, *expr* an expression evaluating to an Integer value, and ● is an operator $<, \leq, >, \geq$. Cold time conditions are treated like cold untimed conditions (cf. Guideline 4a), and hot time conditions are distinguished into:

- Minimal delays ($\bullet \in \{>, \geq\}$): If an enabled minimal delay evaluates to `false`, the cut progresses as soon as the condition becomes `true`.
- Maximal delay ($\bullet \in \{<, \leq\}$): If an enabled maximal delay evaluates to `false`, this is a liveness violation (which is equivalent with a safety violation in this case [Gre11]).

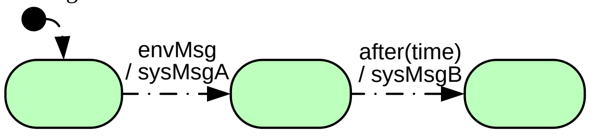
III. Guidelines

General:

- Partial models with informal information relevant to this refinement step: *Application Scenarios*, *Requirements*, and *Behavior – States*.

Determine cold time conditions:

- Investigate *Application Scenarios* and *Requirements* for conditional timing behavior (keywords: "if … more than <time>", "after <time>").
- Investigate *Behavior – States* for transitions with relative time events:*

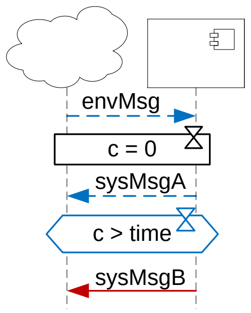


Determine hot time conditions:

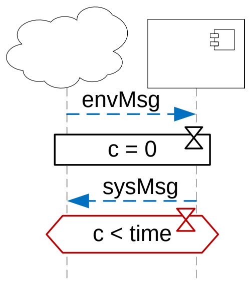
- Investigate *Application Scenarios* and *Requirements* for hard real-time requirements (keywords: "shall/must/have to … within <time>", "shall/must/have to … not longer than <time>".
- Investigate *Application Scenarios*, *Requirements*, *Behavior – States* for annotated high SILs (cf. Guideline 3a).
- Specify the messages to be constrained as monitored messages in a dedicated MSD to gain separation of concerns and atomic MSDs corresponding to atomic real-time requirements.**
- Apply minimal delays to enforce the system/environment to delay or to let it run into a safety violation otherwise [*BGH+14].***
- Apply a mix of cold time conditions and hot untimed conditions with the expression `false` to specify strict time(s) (intervals), where missing such a time (interval) shall lead to a safety violation [*BGH+14].

IV. Examples

* If sending sysMsgA lasts more than `time`, sysMsgB has to be sent:



** Constraining a monitored message with a hot timing condition:



*** Enforce sysMsg to be sent after at least time after envMsg has ben received:

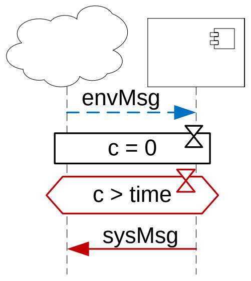


Figure A.6: Guideline 4b—Specify conditional behavior: Add real-time requirements

A.2 EBEAS Models Applied in the Transition from MBSE with CONSENS to SwRE with MSDs

A.2.1 CONSENS System Model

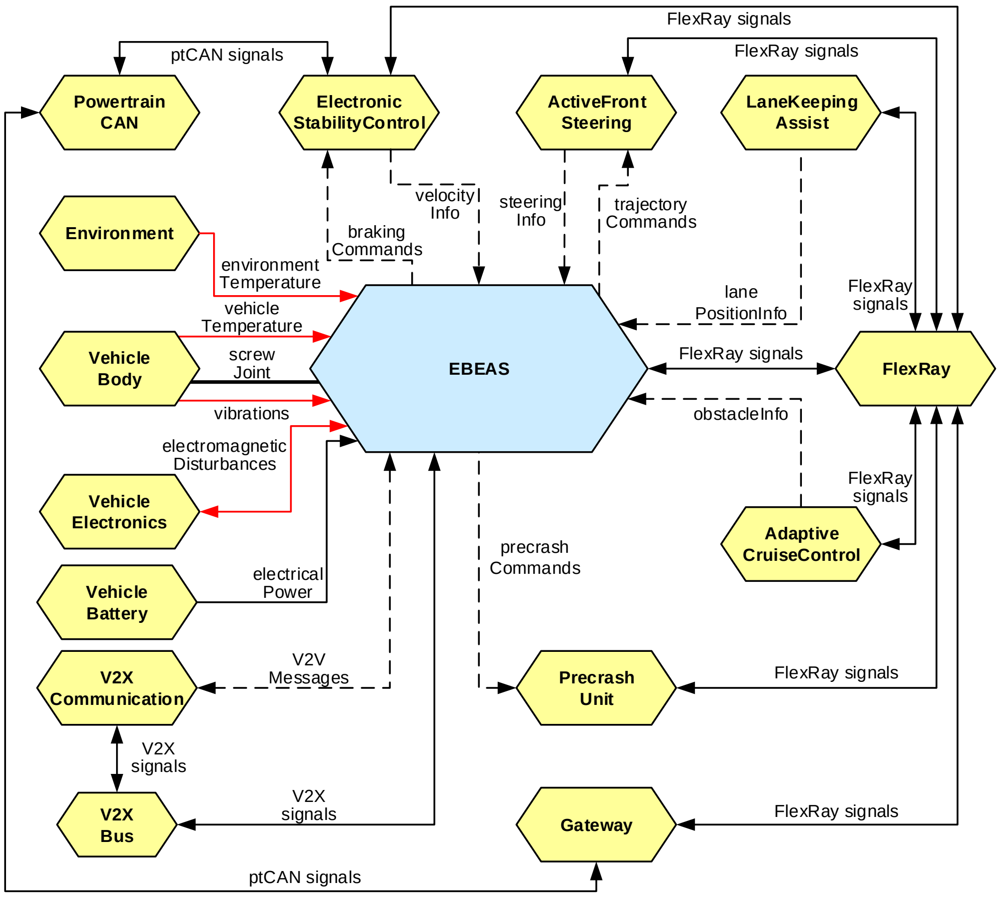
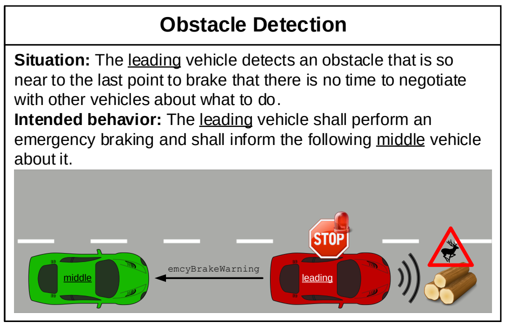


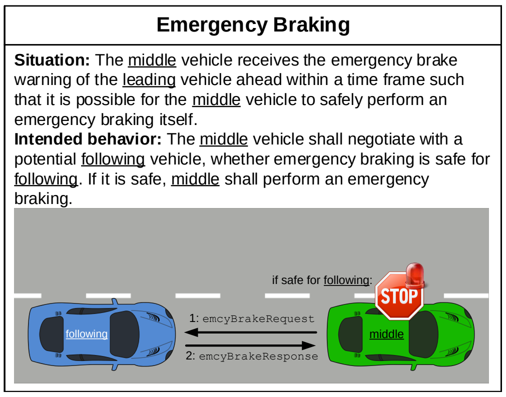
Figure A.7: Partial model *Environment*



Obstacle Detection

Situation: The leading vehicle detects an obstacle that is so near to the last point to brake that there is no time to negotiate with other vehicles about what to do.
Intended behavior: The leading vehicle shall perform an emergency braking and shall inform the following middle vehicle about it.

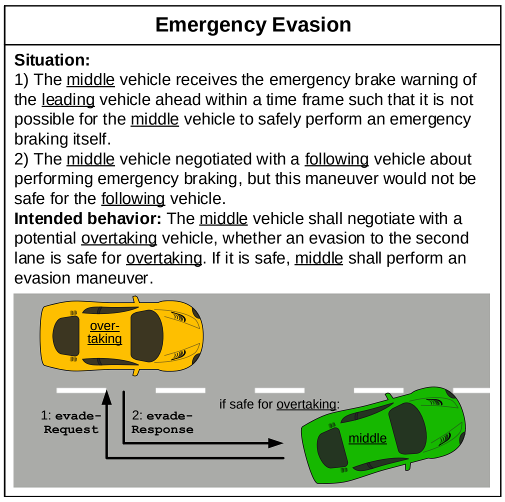
(a) Obstacle Detection



Emergency Braking

Situation: The middle vehicle receives the emergency brake warning of the leading vehicle ahead within a time frame such that it is possible for the middle vehicle to safely perform an emergency braking itself.
Intended behavior: The middle vehicle shall negotiate with a potential following vehicle, whether emergency braking is safe for following. If it is safe, middle shall perform an emergency braking.

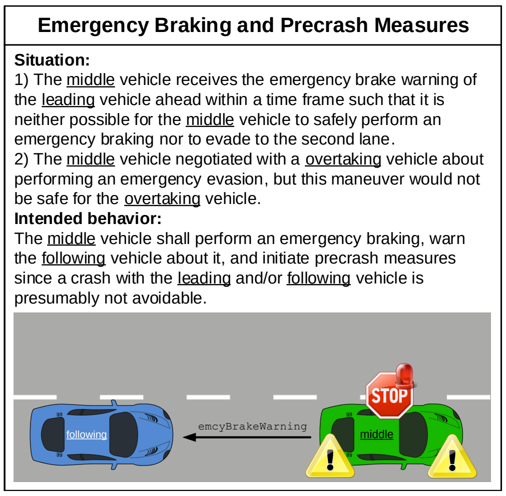
(b) Emergency Braking



Emergency Evasion

Situation:
1) The middle vehicle receives the emergency brake warning of the leading vehicle ahead within a time frame such that it is not possible for the middle vehicle to safely perform an emergency braking itself.
2) The middle vehicle negotiated with a following vehicle about performing emergency braking, but this maneuver would not be safe for the following vehicle.
Intended behavior: The middle vehicle shall negotiate with a potential overtaking vehicle, whether an evasion to the second lane is safe for overtaking. If it is safe, middle shall perform an evasion maneuver.

(c) Emergency Evasion



Emergency Braking and Precrash Measures

Situation:
1) The middle vehicle receives the emergency brake warning of the leading vehicle ahead within a time frame such that it is neither possible for the middle vehicle to safely perform an emergency braking nor to evade to the second lane.
2) The middle vehicle negotiated with a overtaking vehicle about performing an emergency evasion, but this maneuver would not be safe for the overtaking vehicle.
Intended behavior:
The middle vehicle shall perform an emergency braking, warn the following vehicle about it, and initiate precrash measures since a crash with the leading and/or following vehicle is presumably not avoidable.

(d) Emergency Braking and Precrash Measures

Figure A.8: Partial model *Application Scenarios*

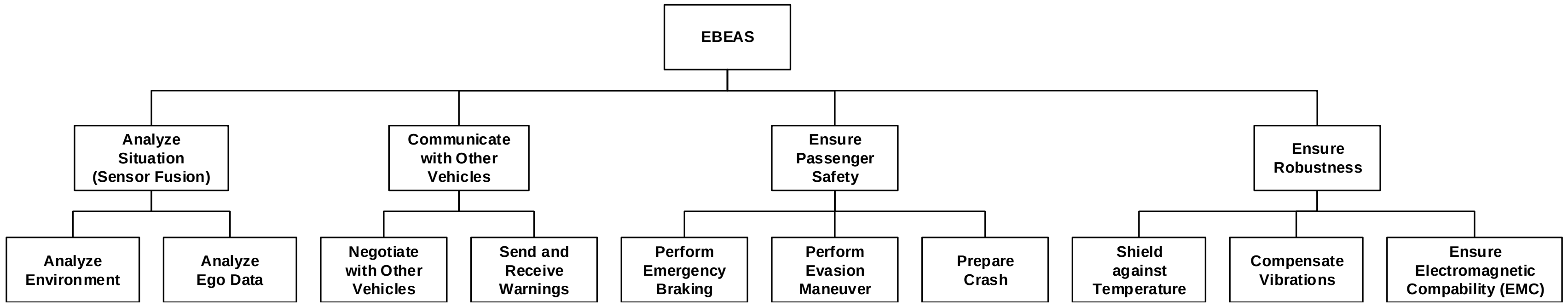

EBEAS
Analyze Situation (Sensor Fusion)
Communicate with Other Vehicles
Ensure Passenger Safety
Ensure Robustness
Analyze Environment
Analyze Ego Data
Negotiate with Other Vehicles
Send and Receive Warnings
Perform Emergency Braking
Perform Evasion Maneuver
Prepare Crash
Shield against Temperature
Compensate Vibrations
Ensure Electromagnetic Compability (EMC)

Figure A.9: Partial model *Functions*

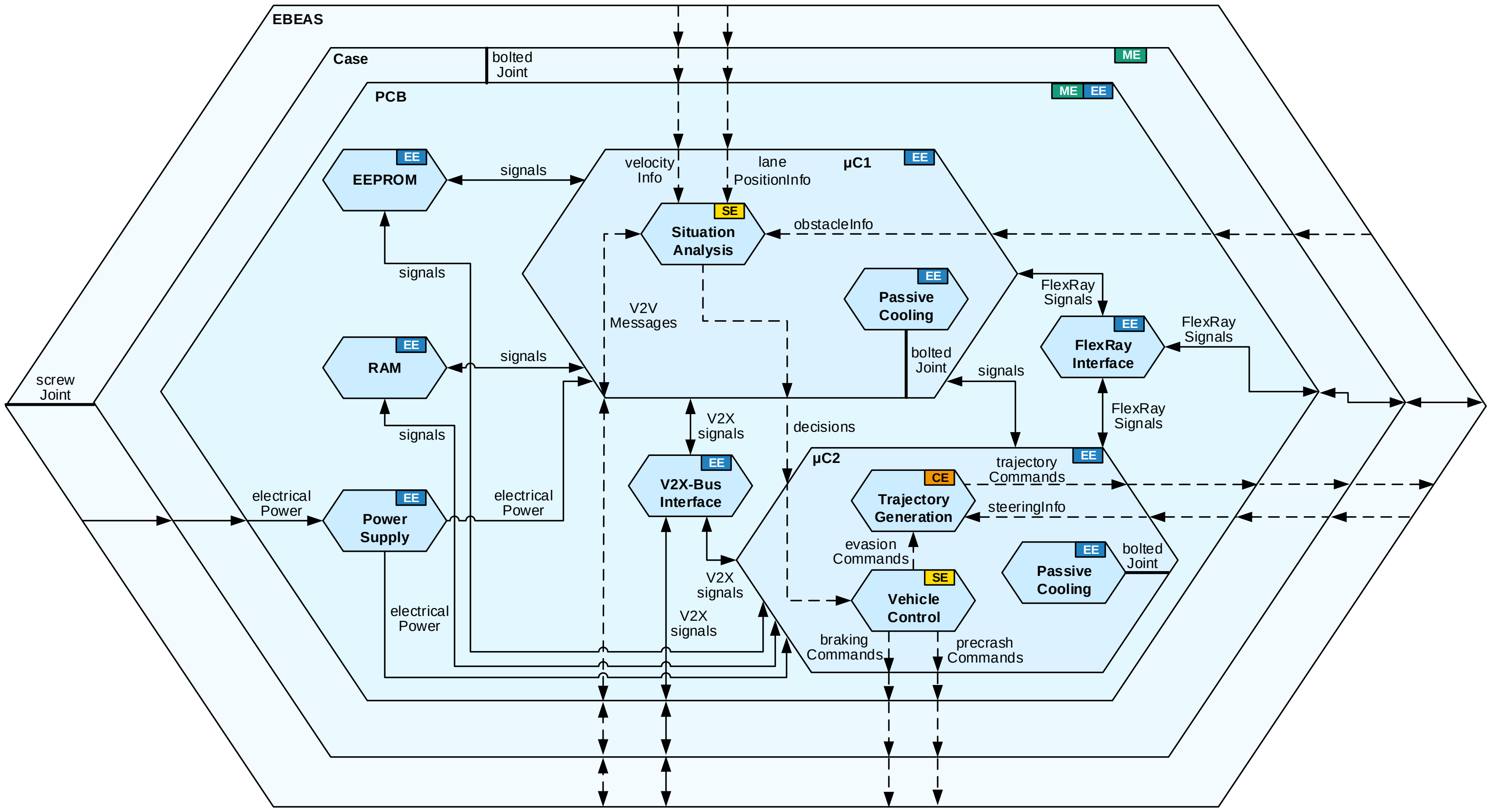
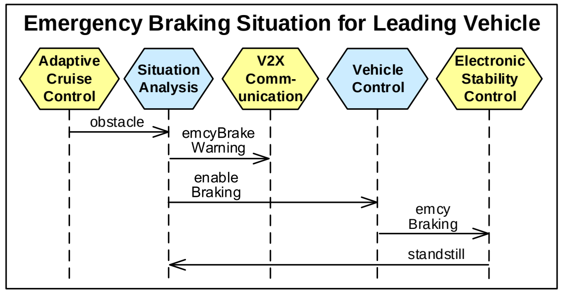
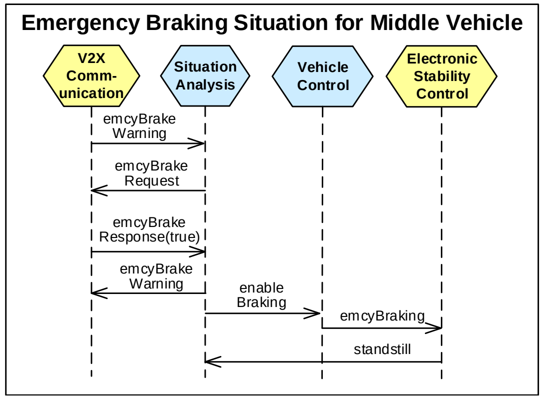


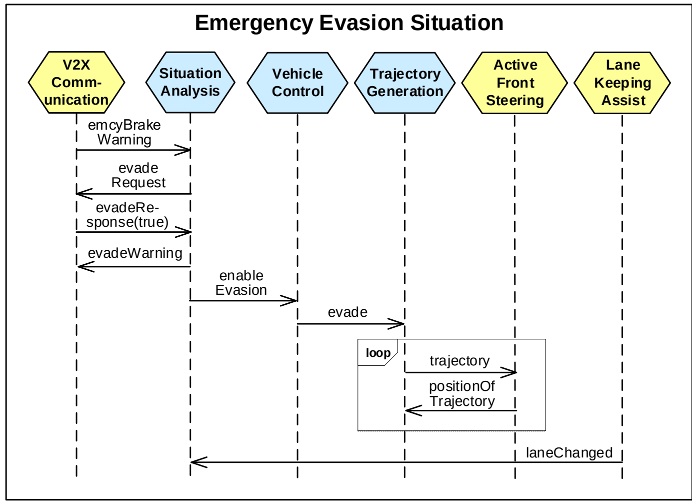
Figure A.10: Partial model *Active Structure*



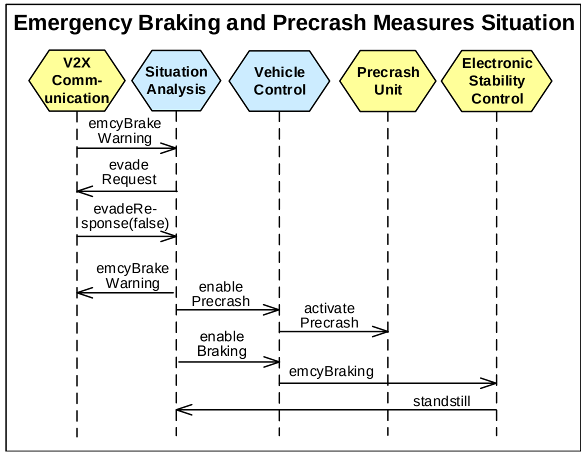
(a) Emergency Braking Situation for Leading Vehicle



(b) Emergency Braking Situation for Middle Vehicle



(c) Emergency Evasion Situation



(d) Emergency Braking and Precrash Measures Situation

Figure A.11: Partial model *Application Scenarios*

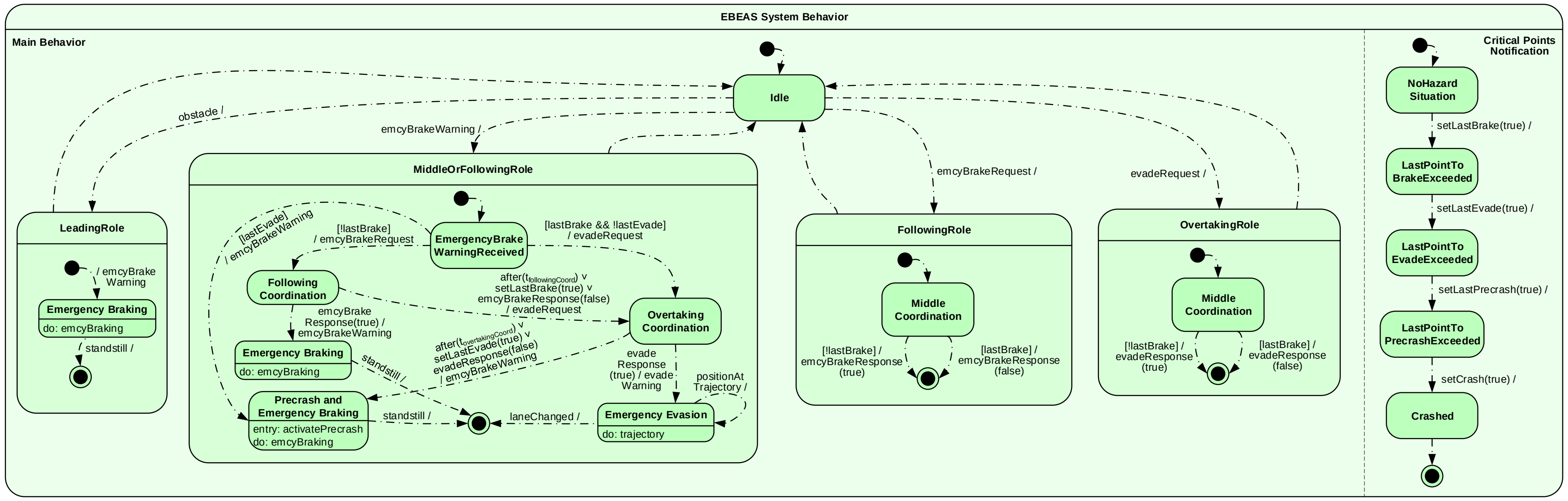

EBEAS System Behavior
Main Behavior
Idle
obstacle /
emcyBrakeWarning /
emcyBrakeRequest /
evadeRequest /
LeadingRole
/ emcyBrake Warning
Emergency Braking
do: emcyBraking
standstill /
MiddleOrFollowingRole
[lastEvade] / emcyBrakeWarning
[!lastBrake] / emcyBrakeRequest
EmergencyBrake WarningReceived
[lastBrake && !lastEvade] / evadeRequest
Following Coordination
after(t_followingCoord) ∨ setLastBrake(true) ∨ emcyBrakeResponse(false) / evadeRequest
emcyBrake Response(true) / emcyBrakeWarning
Overtaking Coordination
after(t_overtakingCoord) ∨ setLastEvade(true) ∨ evadeResponse(false) / emcyBrakeWarning
Emergency Braking
do: emcyBraking
standstill /
evade Response (true) / evade Warning
positionAt Trajectory /
Precrash and Emergency Braking
entry: activatePrecrash
do: emcyBraking
standstill /
laneChanged /
Emergency Evasion
do: trajectory
FollowingRole
Middle Coordination
[!lastBrake] / emcyBrakeResponse (true)
[lastBrake] / emcyBrakeResponse (false)
OvertakingRole
Middle Coordination
[!lastBrake] / evadeResponse (true)
[lastBrake] / evadeResponse (false)
Critical Points Notification
NoHazard Situation
setLastBrake(true) /
LastPointTo BrakeExceeded
setLastEvade(true) /
LastPointTo EvadeExceeded
setLastPrecrash(true) /
LastPointTo PrecrashExceeded
setCrash(true) /
Crashed

Figure A.12: Partial model *Behavior – States*

Table A.1: Relational traceability between *Environment* and *Application Scenarios*

Environment Element Affects Application Scenario	Obstacle Detection	Emergency Braking	Emergency Evasion	Emergency Braking and Precrash Measures
Active Front Steering			X	
Adaptive Cruise Control	X		X	
Electronic Stability Control	X	X		X
Environment				
FlexRay	X	X	X	X
Gateway	X	X	X	X
Lane Keeping Assist			X	
Powertrain CAN	X	X	X	X
Precrash Unit				X
V2X Communication	X	X	X	X
V2X Bus	X	X	X	X
Vehicle Battery				
Vehicle Body				
Vehicle Electronics				

Table A.2: Relational traceability between *Environment* and *Functions*

Environment Element Induces Function	Analyze Environment	Analyze Ego Data	Negotiate with Other Vehicles	Send and Receive Warnings	Perform Emergency Braking	Perform Evasion Maneuver	Prepare Crash	Shield Against Temperature	Compensate Vibrations	Ensure EMC
Active Front Steering										
Adaptive Cruise Control										
Electronic Stability Control										
Environment									X	
FlexRay										
Gateway										
Lane Keeping Assist										
Powertrain CAN										
Precrash Unit										
V2X Communication										
V2X Bus										
Vehicle Battery										
Vehicle Body								X	X	
Vehicle Electronics										X

Table A.3: Relational traceability between *Application Scenarios* and *Functions*

Application Scenario Induces Function	Analyze Environment	Analyze Ego Data	Negotiate with Other Vehicles	Send and Receive Warnings	Perform Emergency Braking	Perform Evasion Maneuver	Prepare Crash	Shield Against Temperature	Compensate Vibrations	Ensure EMC
Obstacle Detection	X			X	X					
Emergency Braking	X	X	X	X	X					
Emergency Evasion	X	X	X	X		X				
Emergency Braking and Precrash Measures	X	X	X	X	X		X			

Table A.4: Relational traceability between *Active Structure* and *Functions*

System Element Realizes Function	Analyze Environment	Analyze Ego Data	Negotiate with Other Vehicles	Send and Receive Warnings	Perform Emergency Braking	Perform Evasion Maneuver	Prepare Crash	Shield Against Temperature	Compensate Vibrations	Ensure EMC
Case								X	X	X
EEPROM	X	X	X	X	X	X	X			
FlexRay Interface	X	X			X	X	X			
µC1	X	X	X	X						
µC2					X	X	X			
Passive Cooling (µC1)								X		
Passive Cooling (µC2)								X		
PCB	X	X	X	X	X	X	X			
Power Supply										
RAM	X	X	X	X	X	X	X			
Situation Analysis	X	X	X	X						
Trajectory Generation						X				
V2X-Bus Interface			X	X						
Vehicle Control					X	X	X			

Table A.5: Relational traceability between *Behavior – Sequences* and *Application Scenarios*

Behavior-Sequence Refines Application Scenario	Obstacle Detection	Emergency Braking	Emergency Evasion	Emergency Braking and Precrash Measures
Emergency Braking Situation for Leading Vehicle	X			
Emergency Braking Situation for Middle Vehicle		X		
Emergency Evasion Situation			X	
Emergency Braking and Precrash Measures Situation				X

A.2.2 MSD Specification

A.2.2.1 Initially Derived MSD Specification

Whereas the initially derived MSD use case *Emergency Evasion* is depicted in Figure 3.19 in Section 3.6.1, we present the remainder of the initially derived MSD specification in the following.

MSD Use Case *Obstacle Detection*

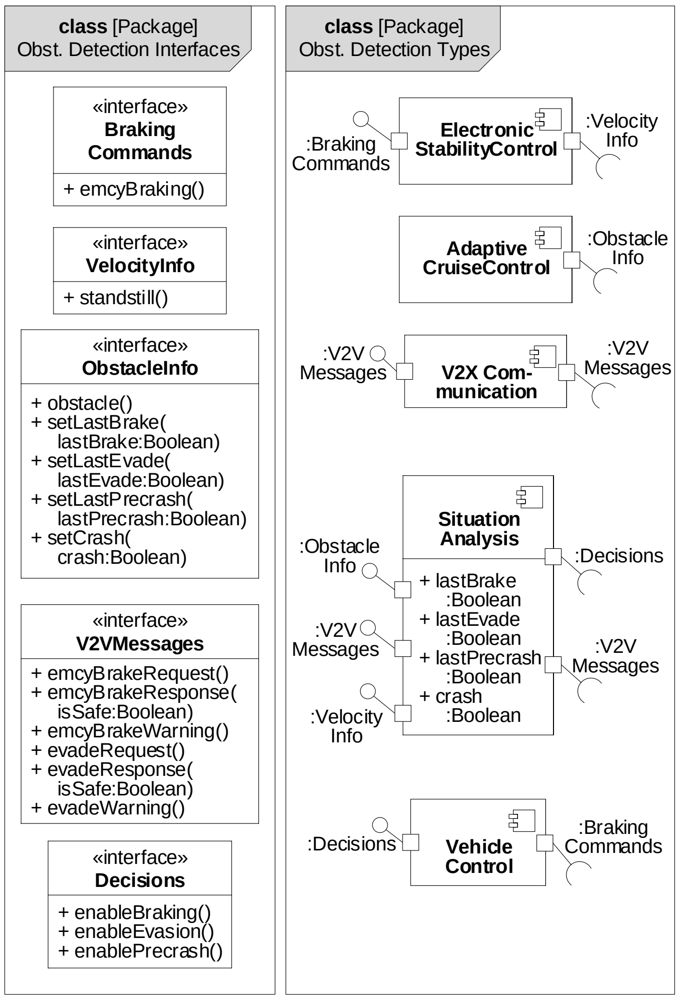


Figure A.13: MSD use case *Obstacle Detection*—classifier view

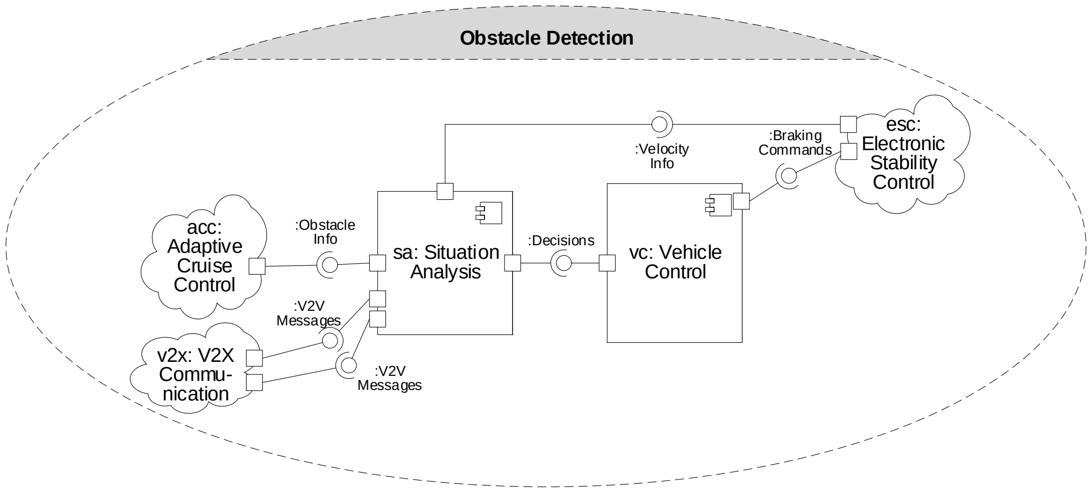


Figure A.14: MSD use case *Obstacle Detection*—architecture view

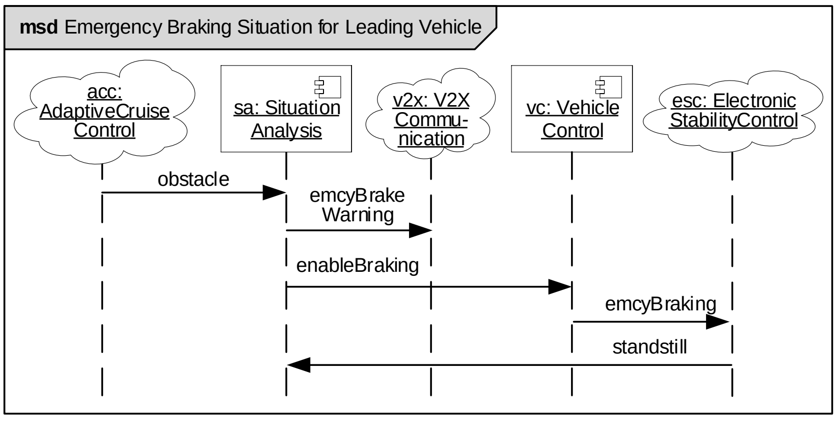


Figure A.15: MSD use case *Obstacle Detection*—interaction view

MSD Use Case *Emergency Braking*

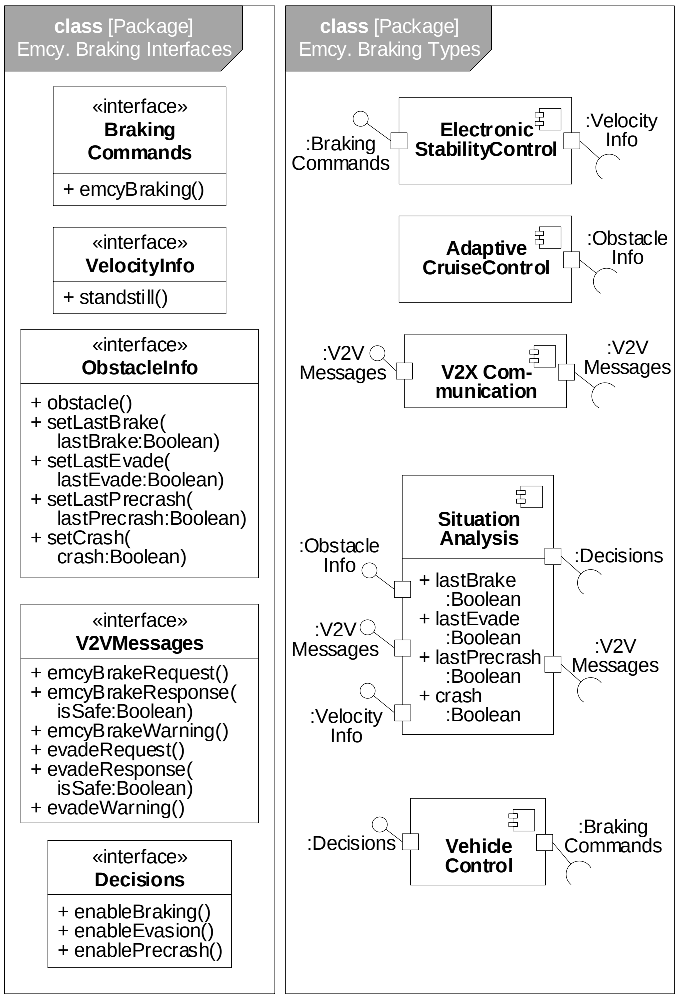


Figure A.16: MSD use case *Emergency Braking*—classifier view

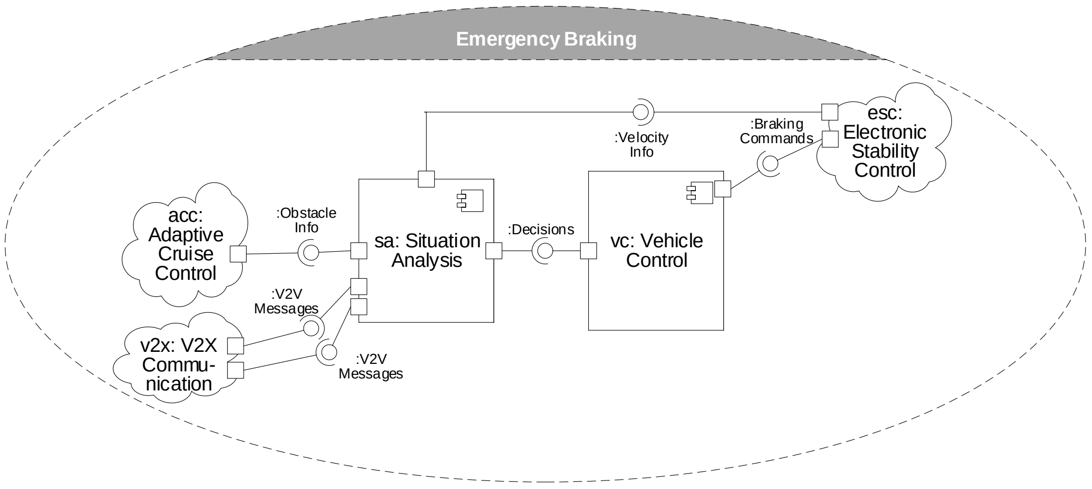


Figure A.17: MSD use case *Emergency Braking*—architecture view

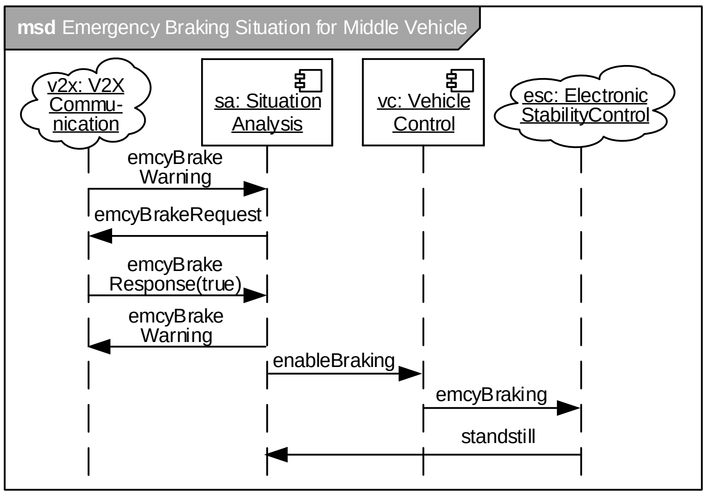


Figure A.18: MSD use case *Emergency Braking*—interaction view

MSD Use Case *Emergency Braking and Precrash Measures*

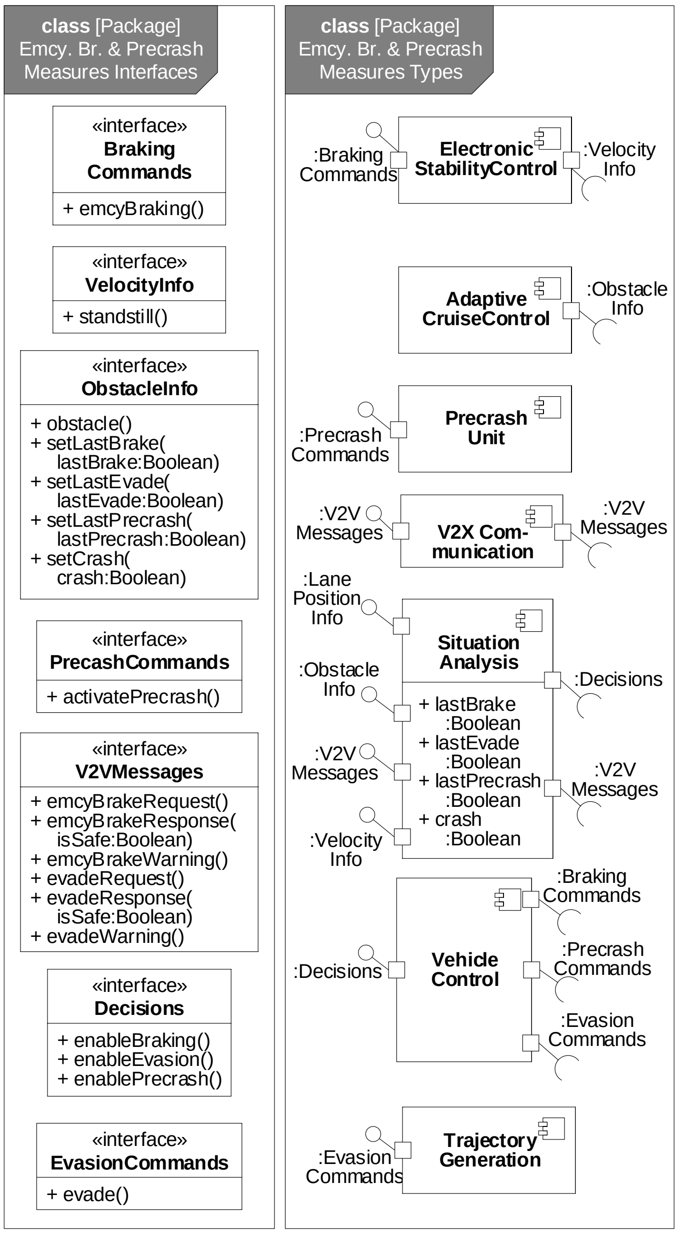


Figure A.19: MSD use case *Emergency Braking and Precrash Measures*—classifier view

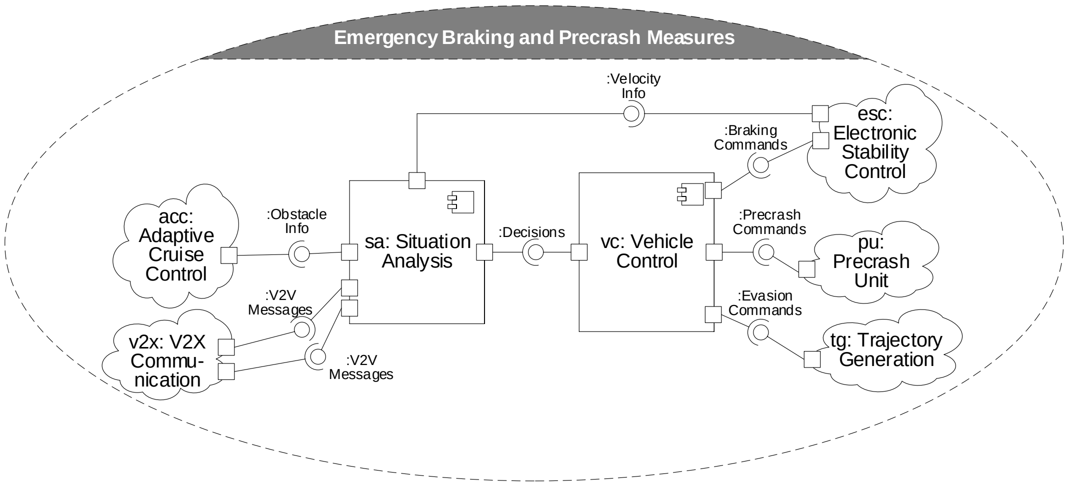


Figure A.20: MSD use case *Emergency Braking and Precrash Measures*—architecture view

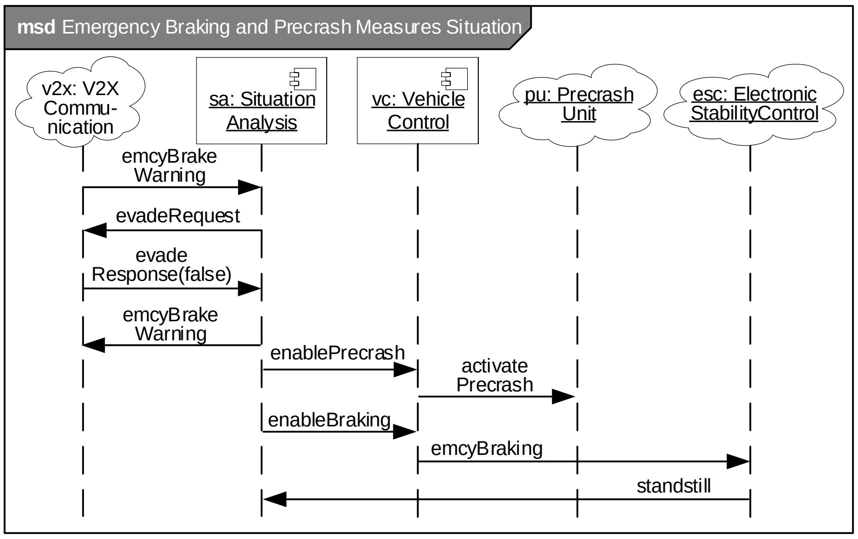


Figure A.21: MSD use case *Emergency Braking and Precrash Measures*—interaction view

A.2.2.2 Example: Manual Refinement of an Initially Derived MSD Specification

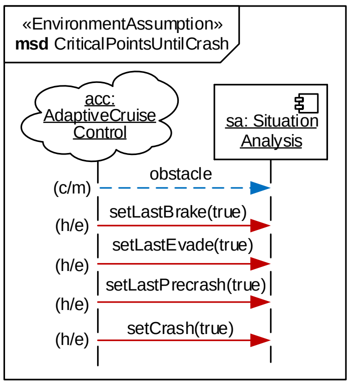
Step 1: Specify Additional MSDs

a) Add Assumption MSDs Regarding the specification of missing environment assumptions, some transitions within the composite state MiddleOrFollowingRole of the *Behavior – States* (cf. Figure A.12 in Appendix A.2) are associated with guards evaluating the Boolean variables `lastBrake` and `lastEvade` in order to react to exceeding the last point to brake or to evade, respectively. The region Critical Points Notification (cf. Figure A.12 in Appendix A.2) specifies the fixed order of these and further critical points in time until a crash occurs. According to Guideline 1a (cf. Figure 3.15), the Software Requirements Engineer has to consider this information in the MSD specification and hence specifies the assumption MSD CriticalPointsUntilCrash depicted in Figure A.22(a). The acc: AdaptiveCruiseControl determines whether the vehicle reached one of the critical points and notifies the sa: SituationAnalysis about this, one after the other. The message temperatures and execution kinds can be specified directly in this refinement step. For example, the environment component acc: AdaptiveCruiseControl is not allowed to disregard one of the messages or change their (physically fixed) order. Thus, the Software Requirements Engineer declares the corresponding messages as hot and executed.

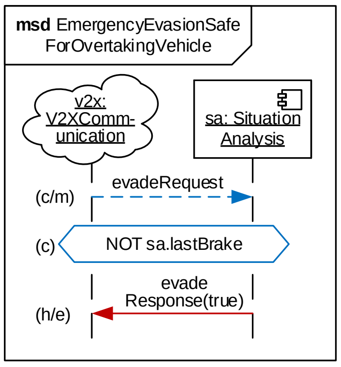
b) Add Requirement MSDs Two examples for sources of missing system behavior are the composite states FollowingRole and OvertakingRole (cf. Figure A.12 in Appendix A.2.1). We exemplarily focus on the latter one in the following: An `evadeResponse` of an overtaking vehicle can evaluate to `true` or `false` according to whether it exceeded the last point to brake w.r.t. to a potential evasion maneuver of the following vehicle (see also the requirement with ID 5.6 in Figure 2.2 in Section 2.2). The Software Requirements Engineer specifies this information by means of the requirement MSDs EmergencyEvasionSafeForOvertakingVehicle and EmergencyEvasionUnsafeForOvertakingVehicle, respectively (cf. Figure A.22(b) and Figure A.22(c)). The assignment of the Boolean variable of the `evadeResponse` depends on the valuation of the attribute lastBrake of the sa: SituationAnalysis, which may be set in the MSD CriticalPointsUntilCrash described above. If the point to last brake for the overtaking vehicle was not exceeded (cold condition `NOT sa.lastBrake` in MSD EmergencyEvasionSafeForOvertakingVehicle), then the sa: SituationAnalysis sends an `evadeResponse(true)` signalizing that it is safe for the overtaking vehicle if the middle vehicle performs an evasion maneuver. If the last point to brake was exceeded (cold condition `sa.lastBrake` in MSD EmergencyEvasionUnsafeForOvertakingVehicle), the sa: SituationAnalysis answers with an `evadeResponse(false)`.

Step 2: Specify Trigger and Execution Behavior

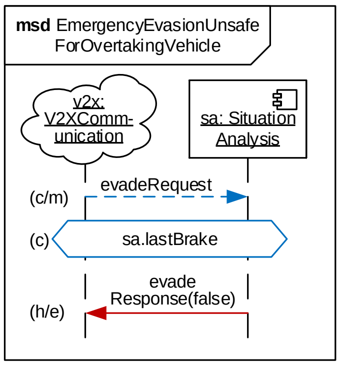
An example of an application scenario triggered by different situations is the application scenario Emergency Evasion. This application scenario describes two starting situations (cf. Figure 2.2 in Section 2.2). In the *Behavior – States*, there are even more transitions (one of them with multiple transition triggers) leading to the state Overtaking Coordination representing a part of this application scenario—each of these transitions have an `evadeRequest` effect. Only if an event `evadeResponse(true)` occurs in this state, the actual evasion maneuver is performed in the state Emergency Evasion. The trigger behavior of the MSD use case *Emergency Evasion* is defined by the different transition paths leading to the state Overtaking Coordination, which all trigger an `evadeRequest`). The actual execution behavior of the MSD use case *Emergency Evasion* is defined by the behavior of the state Emergency Evasion.



(a) Assumption MSD specifying the critical points until a crash occurs



(b) Requirement MSD specifying that an emergency evasion of the middle vehicle would be safe for the overtaking vehicle



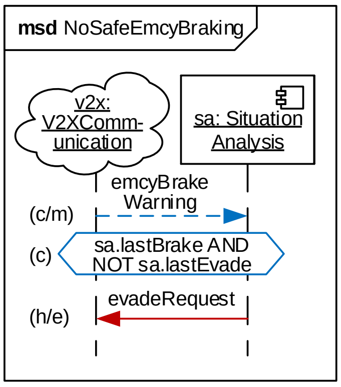
(c) Requirement MSD specifying that an emergency evasion of the middle vehicle would be unsafe for the overtaking vehicle

Figure A.22: Refinement step 1—specify additional MSDs

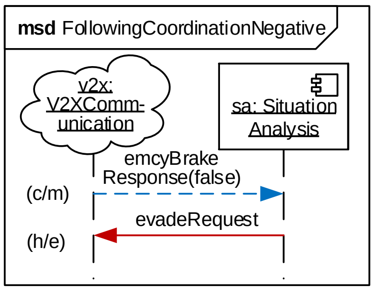
An example of a *Behavior – Sequence* describing an exemplary situation is the *Behavior – Sequence* Emergency Evasion Situation (cf. Figure 3.2 in Section 3.1.2). It incompletely (i.e., without conditional behavior) specifies the first trigger situation of the application scenario Emergency Evasion (cf. Figure 2.2 in Section 2.2), which is represented in the *Behavior – States* by the transition from the state EmergencyBrakeWarningReceived to the state Overtaking Coordination. Furthermore, the *Behavior – Sequence* Emergency Evasion Situation specifies the execution behavior of the application scenario (“intended behavior”) represented by the *Behavior – States* state Emergency Evasion.

An example for formalizing the remaining paths leading to the *Behavior – States* state(s) corresponding to the initially generated MSD by means of other MSDs is depicted in Figure A.23(a). The depicted the MSD NoSafeEmergencyBraking specifies the first trigger situation of the application scenario: If the `emcyBrakeWarning` occurs at a point in time when the last point to brake is exceeded (cold condition part `sa.lastBrake`) and the last point to evade is not exceeded (cold condition part `NOT sa.lastEvade`), the sa: SituationAnalysis has to send an `evadeRequest` to the overtaking vehicle via the v2x: V2XCommunication.

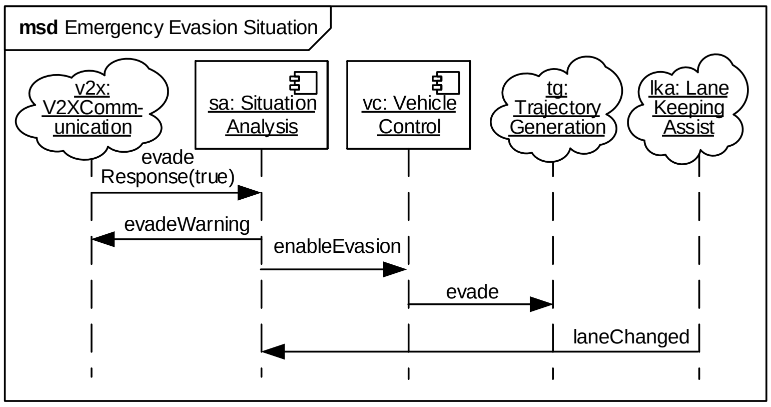
The second trigger situation of the application scenario is specified by means of the MSD FollowingCoordinationNegative in Figure A.23(b): If the following vehicle rejects an emergency braking request with the message `emcyBrakeResponse(false)`, the middle vehicle starts the overtaking coordination with the overtaking vehicle by means of the message `evadeRequest`.



(a) Requirement MSD specifying the start of the overtaking coordination after passing the last point to brake (trigger behavior)



(b) Requirement MSD specifying the start of the overtaking coordination after a rejected emergency braking maneuver (trigger behavior)



(c) Initially generated requirement MSD (cf. Figure 3.19) after removing the first two messages (execution behavior)

Figure A.23: Refinement step 2—specify trigger and execution behavior

As described above, parts of the latter message sequence are already covered incompletely by the first two messages of the initially generated MSD Emergency Evasion Situation (cf. Figure 3.19). The actual execution behavior within this MSD takes place after the message `evadeReponse(true)` (see also the transition from the *Behavior – States* state Overtaking Coordination to the state Emergency Evasion in Figure 2.2 in Section 2.2). Thus, the Software Requirements Engineer has to remove the exemplary and incomplete trigger behavior from the initially generated MSD Emergency Evasion Situation (i.e., the messages `emcyBrakeWarning` and `evadeRequest`) to restrict the MSD to the execution behavior and thereby avoid redundant specification of this execution behavior (cf. Figure A.23(c)). To be more precise, the two messages are moved to the MSD NoSafeEmcyBraking and refined afterward. Further

MSDs for the MSD use case *Emergency Evasion* specifying the trigger behavior and covering all paths to the *Behavior – States* state Overtaking Coordination can be found in Figures A.41 and A.42 in Appendix A.2.2.3. We describe in Section 3.7.2.2 how the incremental update of our transition technique handles such manual changes to MSDs.

Step 3: Specify Temperatures and Execution Kinds

Figure A.24 depicts the MSD Emergency Evasion Situation after the refinement step 3 has been conducted. The initially generated, first two messages `emcyBrakeWarning` and `evadeRequest` have been moved in the last refinement step to the MSD NoSafeEmcyBraking specifying one particular trigger behavior. The MSD-specific modeling constructs temperature and execution kind have to be added to the remaining messages in this refinement step. The now first message `evadeResponse(true)` corresponds to the "intended behavior" part of the application scenario Emergency Evasion "If [the evasion maneuver] is safe [for the overtaking vehicle] [...]" and to the transition from the *Behavior – States* state Overtaking Coordination to Emergency Evasion. Thus, this message triggers this execution behavior MSD and has to be specified as cold and monitored like every first message in an MSD.

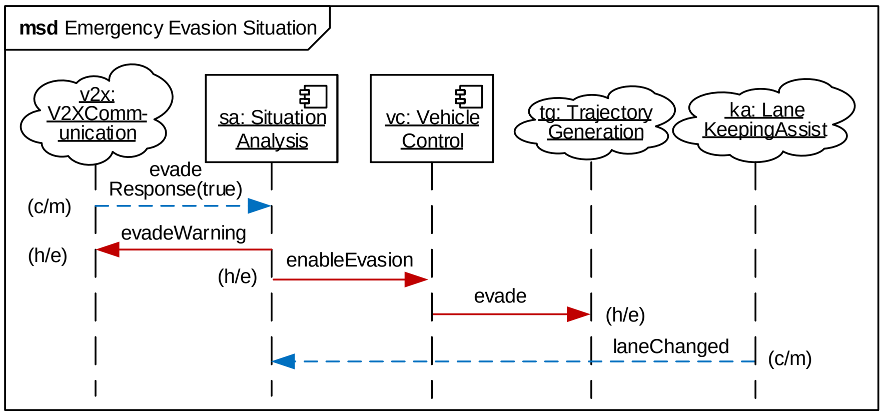


Figure A.24: Refinement step 3—MSD Emergency Evasion Situation after adding temperatures and execution kinds

The next message corresponds to the action of the transition entering the *Behavior – States* state Emergency Evasion: Before an evasion maneuver is performed, the overtaking vehicle has to be warned about it. The next two messages describe the behavior internal to the EBEAS and are thus not covered by the *Behavior – States*. However, it is obvious that these correspond to the mandatory execution behavior of the MSD use case. Thus, the Software Requirements Engineer declares all three messages as hot and executed.

The last message `laneChanged` is sent by the lka: LaneKeepingAssist when the evasion maneuver is finished. The Software Requirements Engineer declares it as cold and monitored since this environment message must neither lead to a safety violation nor a liveness violation for the SUD.

Step 4: Specify Conditional Behavior

We exemplarily focus on real-time requirements in this paragraph. For example, the requirement with ID 5.8 demands that the evasion maneuver has to be finished within the time frame t_{evade}. For this purpose, the Software Requirements Engineer copies the MSD Emergency Evasion

Situation into the MSD EmcyEvasionTimingConstraint (cf. Figure A.25), declares all messages as cold and monitored, and adds a clock reset after the minimal message as well as the maximal delay after the last message.

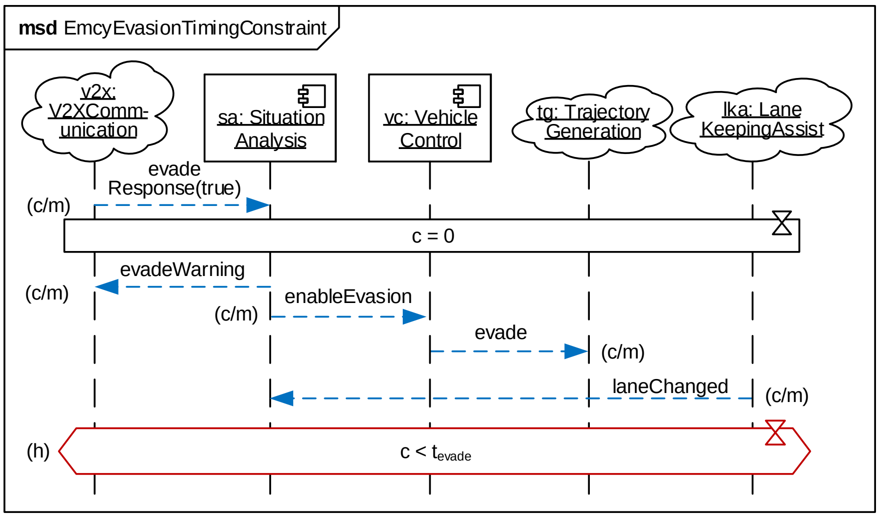


Figure A.25: Refinement step 4—MSD EmcyEvasionTimingConstraint specifying a safety-critical real-time requirement for the functional behavior in MSD Emergency Evasion Situation (cf. Figure A.24)

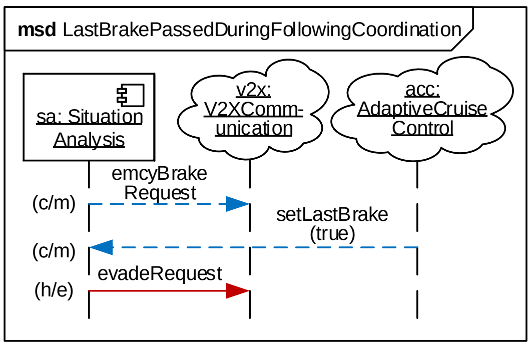
This procedure of specifying a dedicated MSD for monitoring messages under real-time requirements has the advantage of separating the concerns (cf. [*FHKS18; *FHKS17]). That is, the original MSD specifies the purely functional behavior and the newly specified MSD imposes timing constraints on it. Furthermore, the MSDs can be atomically traced to the corresponding information in the CONSENS model. That is, the Software Requirements Engineers establish a trace link from the original MSD Emergency Evasion Situation specifying the functional behavior to the equally named *Behavior – Sequence*, and they establish a trace link from the newly specified MSD EmcyEvasionTimingConstraint to the real-time requirement with ID 5.8 in the CONSENS *Requirements* (cf. Section 3.7).

Step 5a: Check Coverage w.r.t. the Partial Model *Behavior – States*

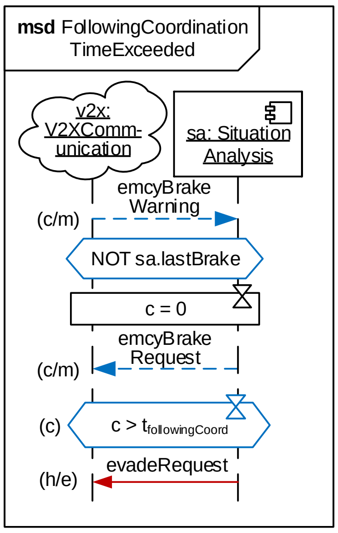
In step 2, the Software Requirements Engineers formalized the transitions leading to the state Overtaking Coordination in the *Behavior – States* (cf. Figure 2.2 in Section 2.2) by means of MSDs. That is, they formalized the transition leading from the state Emergency Brake Warning Received to Overtaking Coordination through the MSD NoSafeEmcyBraking (cf. Figure A.23(a)), and they formalized the transition leading from the state Following Coordination to Overtaking Coordination through the MSD FollowingCoordinationNegative (cf. Figure A.23(b)). However, while formalizing the latter transition, they focused on the message event trigger `emcyBrakeResponse(false)` but neglected the two remaining triggers for this transition.

In such a case, our coverage check reveals that the current state of the MSD specification is incomplete: The message event trigger `setLastBrake(true)` for the same transition, which specifies the cancellation of the following coordination due to passing the last point of brake, has no representation in any requirement MSD. The corresponding coverage rule (cf. Algorithm 3.6

in Section 3.8.3) demands that there is at least one environment message in any requirement MSD for each SwRE-relevant message event trigger in the *Behavior – States*. The only environment message `setLastBrake(true)` in the current MSD specification is present in the assumption MSD CriticalPointsUntilCrash (cf. Figure A.22(a)), but there is no requirement MSD encompassing it. Thus, the Software Requirements Engineer newly specifies the MSD LastBrakePassedDuringFollowingCoordination (cf. Figure A.26(a)).



(a) Requirement MSD specifying the cancellation of the following coordination due to passing the last point of brake (trigger behavior)



(b) Requirement MSD specifying the cancellation of the following coordination due to exceeding the maximum time (trigger behavior)

Figure A.26: Refinement step 5a—manually specified MSDs after coverage check w.r.t. the partial model *Behavior – States*

Furthermore, the Software Requirements Engineers realize during investing the transition leading from the state Following Coordination to Overtaking Coordination that they neglected the transition trigger `after(`$t_{followingCoord}$`)`. This relative time event specifies the cancellation of the following coordination due to exceeding the maximum time. Thus, the Software Requirements Engineers newly specify the MSD FollowingCoordinationTimeExceeded (cf. Figure A.26(b)).

Step 5b: Validate Existential Behavior

Beyond the initial MSDs that are subject to the manual refinement, we automatically derive from any *Behavior – Sequence* each an existential MSD. These existential MSDs serve as test oracle for the refined MSD specification, where the specified system must be able to produce at least one trace that fits to one of the existential scenarios. Figures A.27 to A.30 depict these existential MSDs for the example specification. The Software Requirements Engineer is able to simulatively produce the corresponding traces in Play-out, so that no further corrections to the MSD specification are necessary.

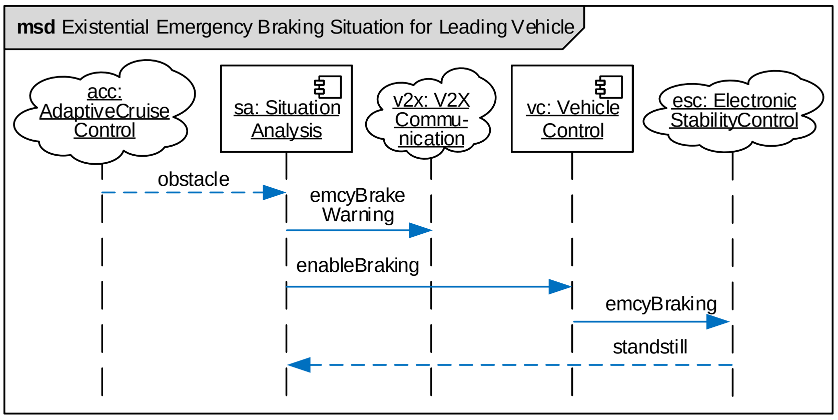


Figure A.27: Refinement step 5b—Automatically derived existential MSD Existential Emergency Braking Situation for Leading Vehicle complementing the initially derived corresponding MSD of Figure A.15

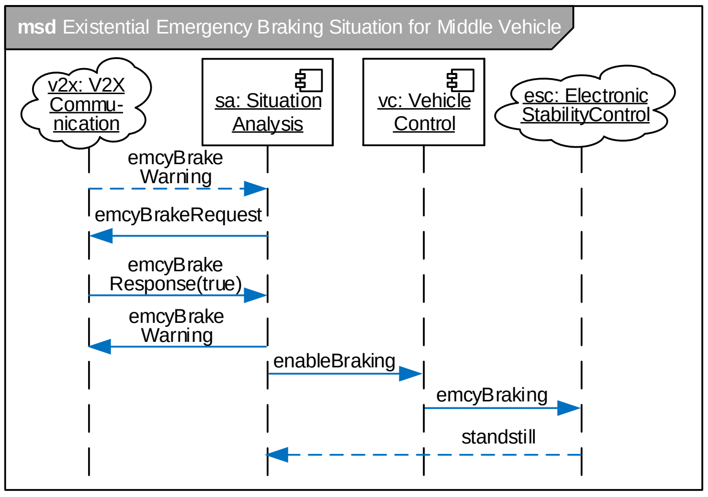


Figure A.28: Refinement step 5b—Automatically derived existential MSD Existential Emergency Braking Situation for Middle Vehicle complementing the initially derived corresponding MSD of Figure A.18

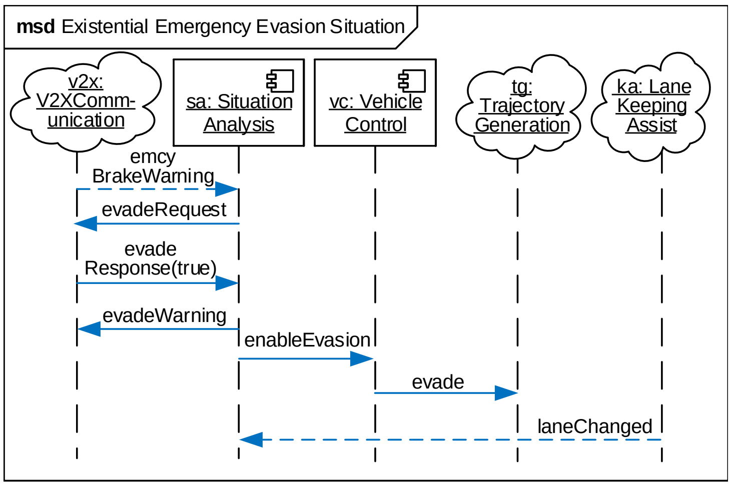


Figure A.29: Refinement step 5b—Automatically derived existential MSD Existential Emergency Evasion Situation complementing the initially derived corresponding MSD of Figure 3.19

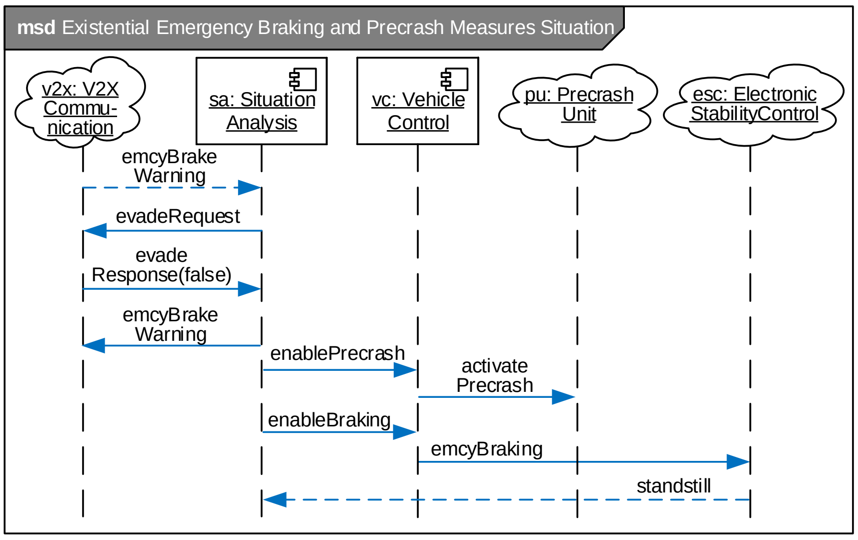


Figure A.30: Refinement step 5b—Automatically derived existential MSD Existential Emergency Braking and Precrash Measures Situation complementing the initially derived corresponding MSD of Figure A.21

A.2.2.3 MSD Specification After Manual Refinement

Figure A.31 depicts the overview of the particular MSD use cases as well as their merge relationships. Besides the initially derived MSD use cases (cf. Appendix A.2.2.1), the Software Requirements Engineer specifies an MSD use case *General Environment Assumptions* for assumption MSDs that are relevant for the remaining MSD use cases. Furthermore, the Software Requirements Engineer adds the merge relationships [Gre11; *HFK+16], which are not automatically derived.

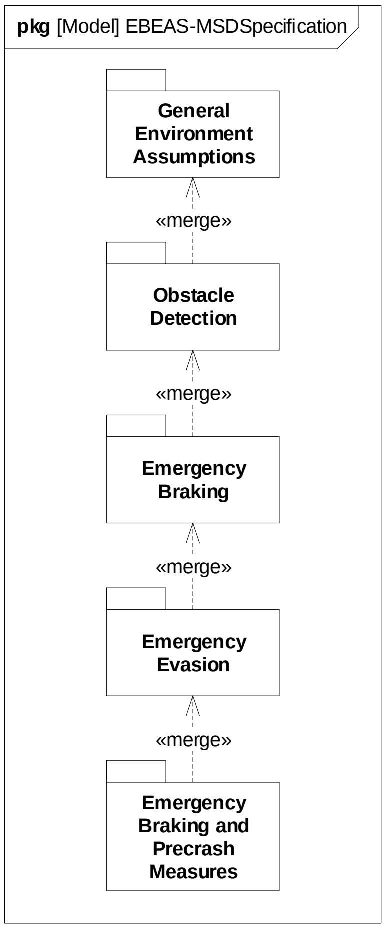

pkg [Model] EBEAS-MSDSpecification
General Environment Assumptions
«merge»
Obstacle Detection
«merge»
Emergency Braking
«merge»
Emergency Evasion
«merge»
Emergency Braking and Precrash Measures

Figure A.31: MSD use case overview

MSD Use Case *General Environment Assumptions*

The Software Requirements Engineer specified an additional assumption MSD CriticalPoints-UntilCrash during step 1 of the manual refinement of the initially derived MSD specification (cf. Figure A.22(a)). This assumption MSD is also depicted in Figure A.34 and explicitly defines the critical points in time until a crash happens as well as their order of occurrence. Furthermore, the Software Requirements Engineer specifies the minimal time bound between multiple `obstacle` event occurrences in an assumption MSD ObstacleDetectionTimeBound (cf. Figure A.35), which enables that the remaining real-time requirements do not result in a safety violation and moreover reduces the state space of the MSD specification.

The Software Requirements Engineer specifies these assumption MSDs as well as the corresponding classifier view (cf. Figure A.32) and architecture view (cf. Figure A.33) as part of the MSD use case *General Environment Assumptions* since it is relevant for all other MSD use cases.

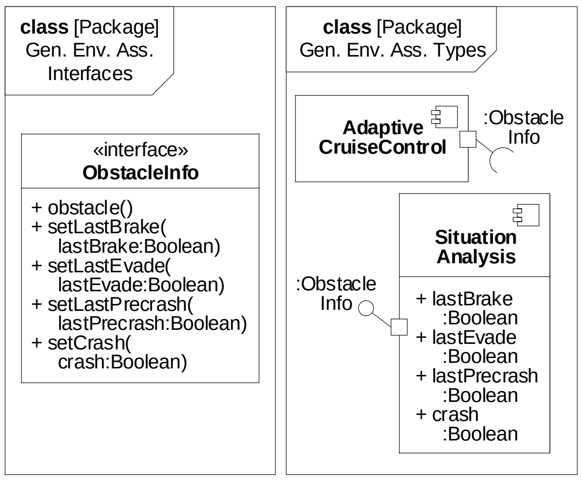

class [Package] Gen. Env. Ass. Interfaces
«interface»
ObstacleInfo
+ obstacle()
+ setLastBrake(lastBrake:Boolean)
+ setLastEvade(lastEvade:Boolean)
+ setLastPrecrash(lastPrecrash:Boolean)
+ setCrash(crash:Boolean)
class [Package] Gen. Env. Ass. Types
Adaptive CruiseControl
:Obstacle Info
Situation Analysis
:Obstacle Info
+ lastBrake :Boolean
+ lastEvade :Boolean
+ lastPrecrash :Boolean
+ crash :Boolean

Figure A.32: MSD use case *General Environment Assumptions*—classifier view

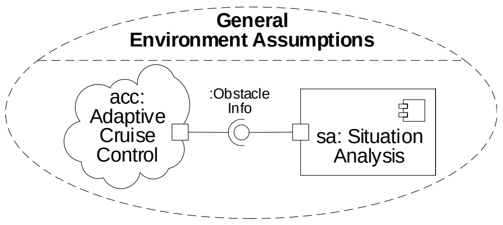


Figure A.33: MSD use case *General Environment Assumptions*—architecture view

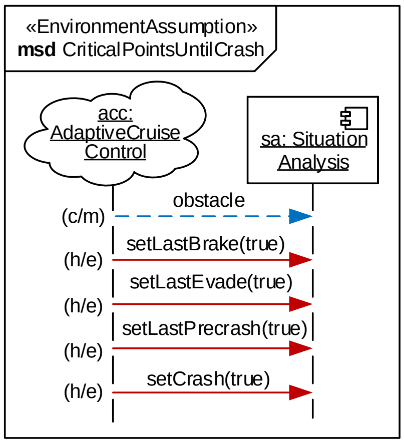


Figure A.34: MSD use case *General Environment Assumptions*—assumption MSD specifying the critical points in time until a crash occurs

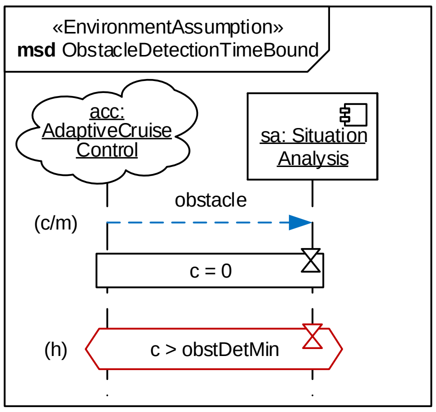


Figure A.35: MSD use case *General Environment Assumptions*—assumption MSD specifying the minimal time bound between occurrences of `obstacle`

MSD Use Case *Obstacle Detection*

The classifier view and architecture view do not differ from the ones of the initially derived MSD use case depicted in Figure A.13 and Figure A.14 (cf. Appendix A.2.2.1), respectively.

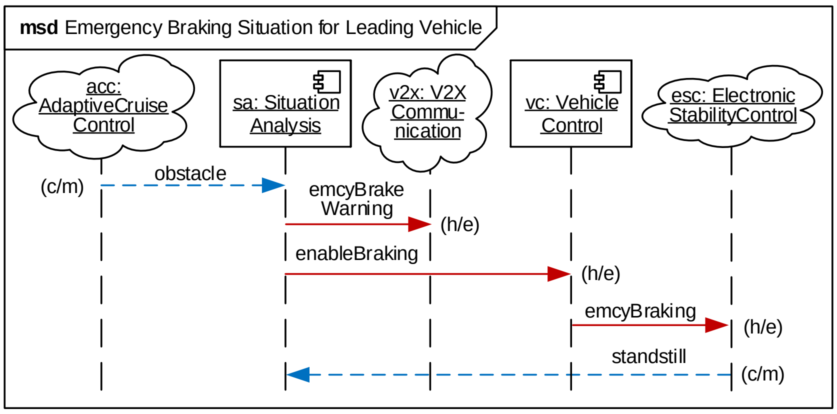


Figure A.36: MSD use case *Obstacle Detection*—requirement MSD Emergency Braking Situation for Leading Vehicle manually refined by temperatures and execution kinds (cf. initially derived MSD in Figure A.15 in Appendix A.2.2.1)

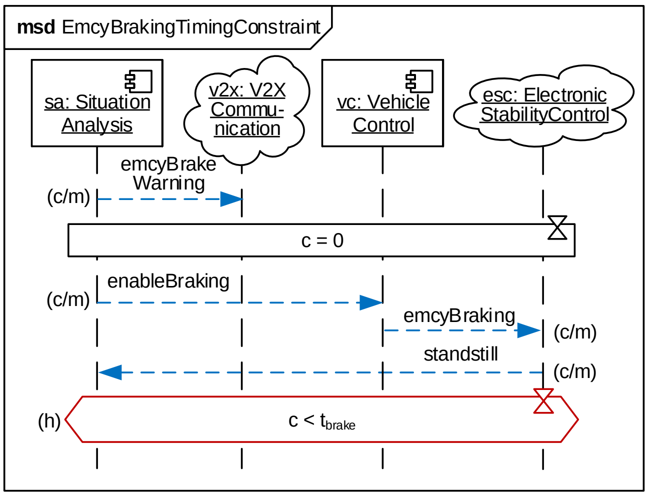


Figure A.37: MSD use case *Obstacle Detection*—requirement MSD restricting the behavior of the MSD Emergency Braking Situation for Leading Vehicle (Figure A.36), of the MSD Emergency Braking Situation for Middle Vehicle (Figure A.40), and of the MSD Emergency Braking and Precrash Measures Situation (Figure A.46) with a real-time requirement

MSD Use Case *Emergency Braking*

The classifier view and architecture view do not differ from the ones of the initially derived MSD use case depicted in Figure A.16 and Figure A.17 (cf. Appendix A.2.2.1), respectively.

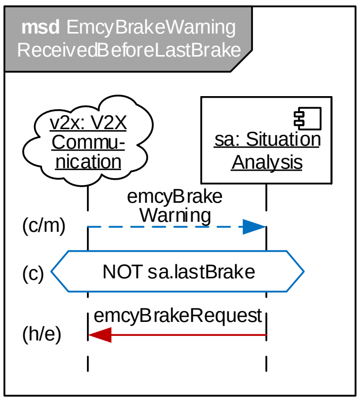
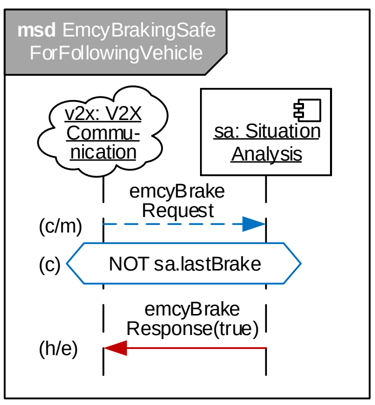
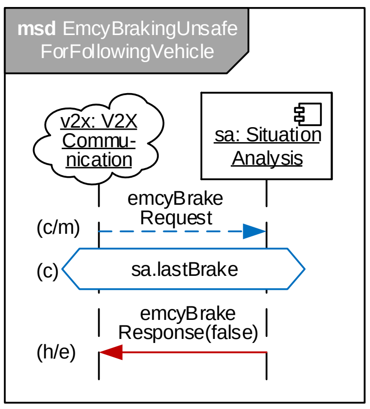


Figure A.38: MSD use case *Emergency Braking*—requirement MSD specifying that an `emcy-BrakeRequest` has to be sent if an `emcyBrakeWarning` was received from the preceding vehicle before the own last point to brake is passed (cf. transition from state EmergencyBrakeWarningReceived to state FollowingCoordination in composite state MiddleOrFollowingRole in Figure A.12 in Appendix A.2.1)

Figure A.40 depicts the manually refined version of the MSD Emergency Braking Situation for Middle Vehicle. The Software Requirements Engineer moves the first messages representing the trigger behavior of the initially derived version of this MSD (cf. Figure A.18 in Appendix A.2.2.1) to the MSDs EmcyBrakeWarningReceivedBeforeLastBrake (cf. Figure A.38) and EmcyBrakingSafeForFollowingVehicle (cf. Figure A.39(a)) in the course of step 2 of the manual refinement process (cf. Appendix A.2.2.2). Besides the temperatures and execution kinds for the messages, the Software Requirements Engineer adds the cold condition `NOT sa.lastBrake` for the case that the last point to brake is passed during the state FollowingCoordination before an `emcyBrakeResponse(true)` is received (cf. trigger event `setLastBrake(true)` of the transition from the state FollowingCoordination to the state OvertakingCoordination within the composite state MiddleOrFollowingRole in Figure A.12).



(a) Requirement MSD specifying that emergency braking is safe for the following vehicle since it has not passed the last point to brake (trigger behavior for the MSD Emergency Braking Situation for Middle Vehicle in Figure A.40)



(b) Requirement MSD specifying that emergency braking is unsafe for the following vehicle since it has passed the last point to brake (trigger behavior for the MSD FollowingCoordinationNegative) in Figure A.41(d))

Figure A.39: Trigger behavior of the MSD use cases *Emergency Braking* and *Emergency Evasion* leading to a positive or negative `emcyBrakeResponse` (cf. composite state FollowingRole in Figure A.12 in Appendix A.2.1)

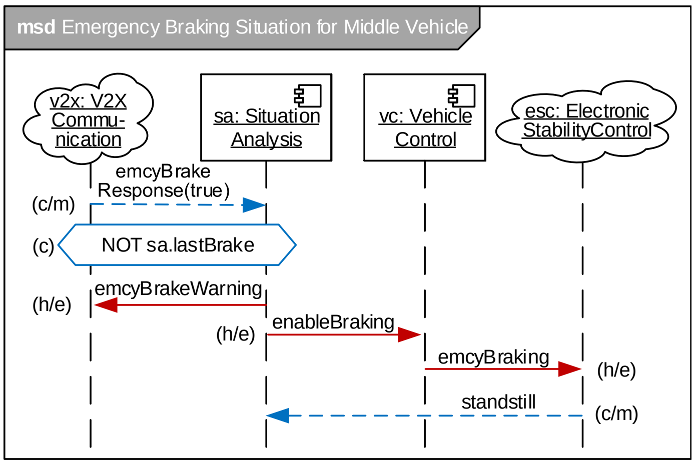
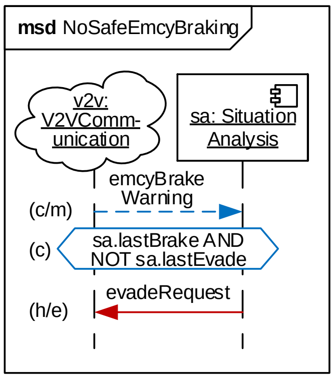


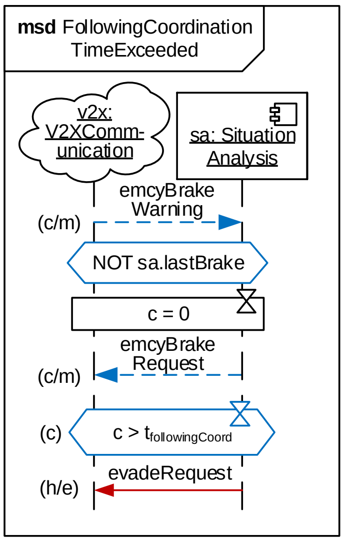
Figure A.40: Execution behavior of the MSD use case *Emergency Braking*—requirement MSD specifying the engagement of the esc: ElectronicStabilityControl to perform a braking maneuver (cf. transition from the state FollowingCoordination to the state Emergency Braking within the composite state MiddleOrFollowingRole in Figure A.12 in Appendix A.2.1)

MSD Use Case *Emergency Evasion*

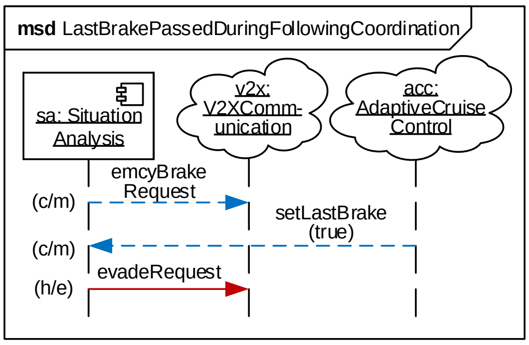
The classifier view and architecture view do not differ from the ones of the initially derived MSD use case depicted in Figure 3.19 in Section 3.6.1.



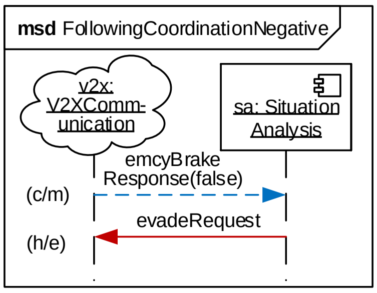
(a) Requirement MSD specifying the start of the overtaking coordination after passing the last point to brake (cf. transition from the state EmergencyBrakeWarningReceived to the state OvertakingCoordination within the composite state MiddleOrFollowingRole in Figure A.12)



(b) Requirement MSD specifying the cancellation of the following coordination due to missing a real-time requirement (cf. relative time event `after(`$t_{followingCoord}$`)` of the transition from the state FollowingCoordination to the state OvertakingCoordination within the composite state MiddleOrFollowingRole in Figure A.12)

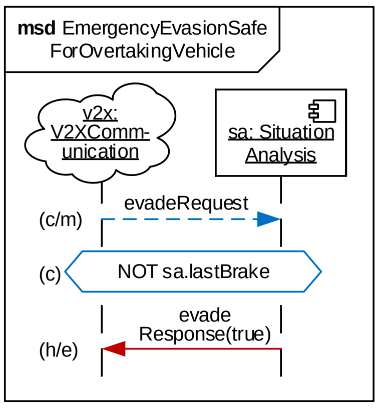


(c) Requirement MSD specifying the cancellation of the following coordination due to passing the last point to brake (cf. trigger event `setLastBrake(true)` of the transition from the state FollowingCoordination to the state OvertakingCoordination within the composite state MiddleOrFollowingRole in Figure A.12)

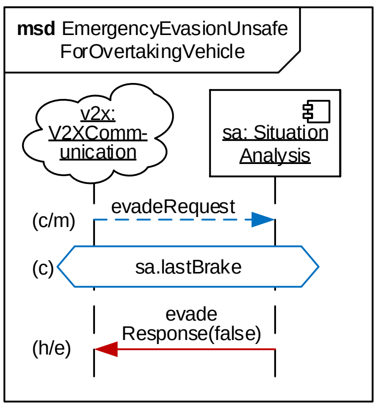


(d) Requirement MSD specifying the start of the overtaking coordination after an rejected emergency braking maneuver (cf. trigger event `emcyBrakeResponse(false)` of the transition from the state FollowingCoordination to the state OvertakingCoordination within the composite state MiddleOrFollowingRole in Figure A.12)

Figure A.41: Trigger behavior of the MSD use case *Emergency Evasion* leading to an `evadeRequest`



(a) Requirement MSD specifying that an emergency evasion of the middle vehicle would be safe for the overtaking vehicle



(b) Requirement MSD specifying that an emergency evasion of the middle vehicle would be unsafe for the overtaking vehicle

Figure A.42: Trigger behavior of the MSD use case *Emergency Evasion* leading to a positive or negative `evadeResponse` (cf. composite state OvertakingRole in Figure A.12 in Appendix A.2.1)

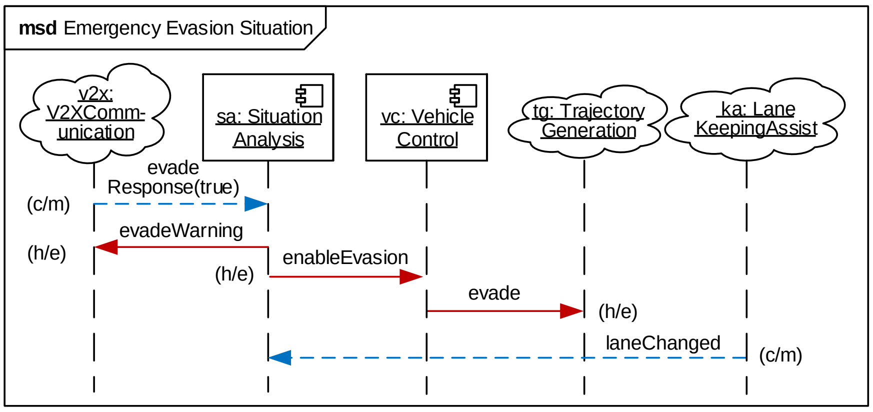


Figure A.43: Execution behavior of the MSD use case *Emergency Evasion*—requirement MSD specifying the activation of the tg: TrajectoryGeneration to perform an evasion maneuver

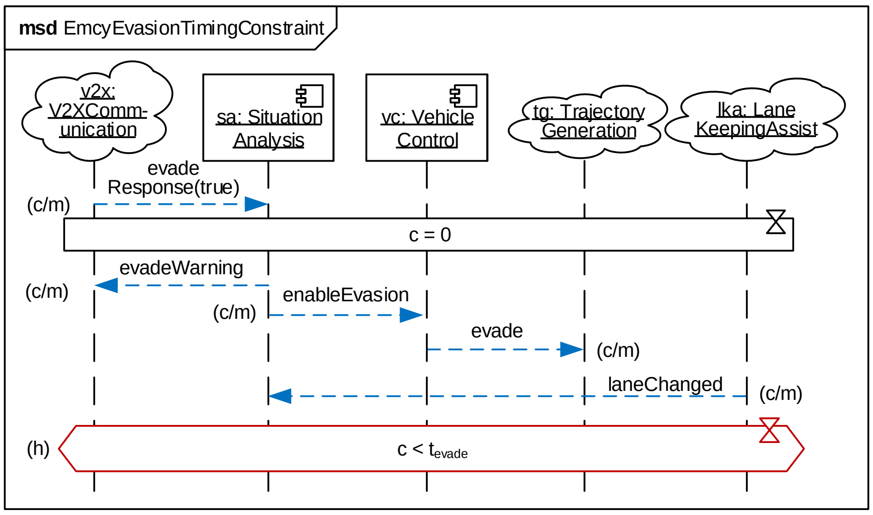
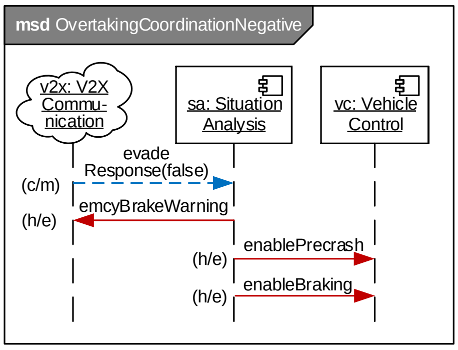


Figure A.44: Real-time requirement for execution behavior of the MSD use case *Emergency Evasion*—requirement MSD restricting the behavior of the MSD Emergency Evasion Situation (cf. Figure A.41) with a real-time requirement

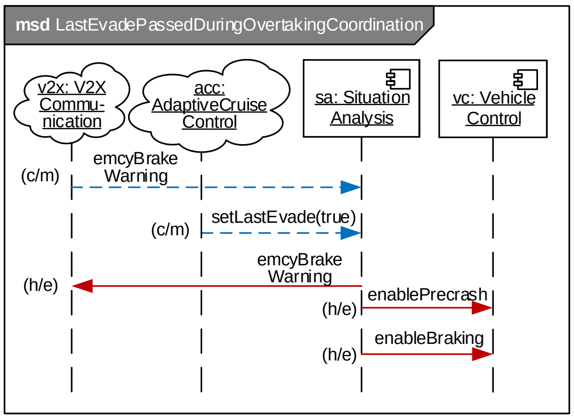
MSD Use Case *Emergency Braking and Precrash Measures*

The classifier view and architecture view do not differ from the ones of the initially derived MSD use case depicted in Figure A.19 and Figure A.20 (cf. Appendix A.2.2.1), respectively.

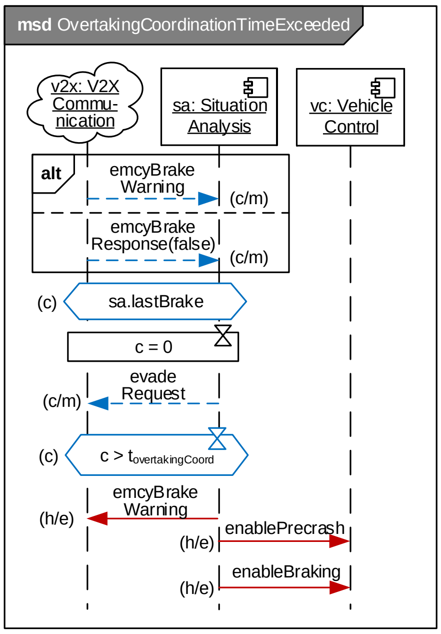
Figure A.46 depicts the manually refined version of the MSD Emergency Braking and Precrash Measures Situation. The Software Requirements Engineer moves the lifeline v2x: V2XCommunication and the first messages representing the trigger behavior of the initially derived version of this MSD (cf. Figure A.21 in Appendix A.2.2.1) to the MSDs NoSafeEmcyBraking (cf. Figure A.41(a)), EmergencyEvasionUnsafeForOvertakingVehicle (cf. Figure A.42(b)) (both part of MSD use case *Emergency Evasion*), and OvertakingCoordinationNegative (cf. Figure A.45(a)) in the course of step 2 of the manual refinement process (cf. Appendix A.2.2.2).



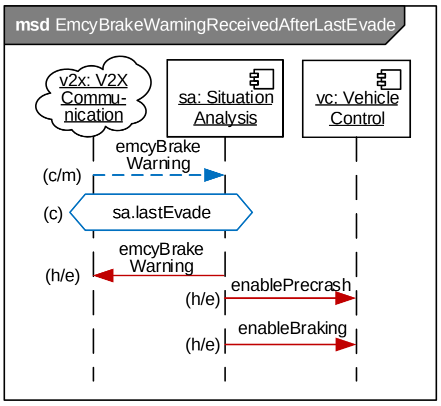
(a) Requirement MSD specifying that an `evadeRequest` is rejected (cf. trigger event `evadeResponse(false)` of the transition from the state OvertakingCoordination to the state Precrash and Emergency Braking within composite state MiddleOrFollowingRole in Figure A.12)



(b) Requirement MSD specifying that the last point to evade is passed during the overtaking coordination (cf. trigger event `setLastEvade(true)` of the transition from the state OvertakingCoordination to the state Precrash and Emergency Braking within the composite state MiddleOrFollowingRole in Figure A.12)



(c) Requirement MSD specifying that the time for the overtaking coordination is exceeded (cf. relative time event `after(`$t_{overtakingCoord}$`)` of the transition from the state OvertakingCoordination to the state Precrash and Emergency Braking within the composite state MiddleOrFollowingRole in Figure A.12)



(d) Requirement MSD specifying that the last point to evade is already passed when receiving an `emcyBrakeWarning` (cf. guard `lastEvade` of the transition from the state EmergencyBrakeWarningReceived to the state Precrash and Emergency Braking within the composite state MiddleOrFollowingRole in Figure A.12)

Figure A.45: Trigger behavior of the MSD use case *Emergency Braking and Precrash Measures*

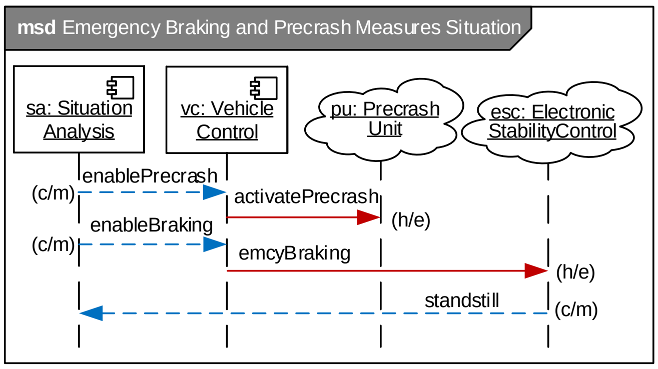


Figure A.46: Execution behavior of the MSD use case *Emergency Braking and Precrash Measures*—requirement MSD specifying the activation of precrash measures and of an emergency braking maneuver

A.3 Case Study Details: Hypothesis H2 for the Transition Technique from MBSE to SwRE

This appendix presents the detailed results of the model element variables in the context of the case study conducted for the transition technique from MBSE to SwRE (cf. Section 3.9.2). Table A.6 presents the results for the variables H2.1, H2.2, and H2.1-update, whereas Table A.7 presents the results for the variables H2.3, H2.4, and H2.3-update.

Table A.6: Detailed results of the model element amount variables H2.1, H2.2, and H2.1-update

		# EBEAS			**# CAS**		
Partial Model	**Model Element (-> Attribute)**	**H2.1**	**H2.2**	**H2.1-update**	**H2.1**	**H2.2**	**H2.1-update**
Application Scenarios	Application Scenario	4	4	0	5	3	0
	Application Scenario -> name	4	4	0	5	3	0
Functions	Function	15	7	0	28	9	0
	Function -> name	15	7	0	28	9	0
Environment / Active Structure	Environment Template	14	5	0	7	4	0
	Environment Template -> name	14	5	0	7	1	0
	Environment Element Exemplar	14	5	0	7	4	0
	Environment Element Exemplar -> name	14	5	0	7	4	0
	Environment Element Exemplar -> type	14	5	0	7	4	0
	System Element Template	14	3	1	11	3	1
	System Element Template -> name	14	3	1	11	3	1
	System Element Exemplar	14	3	1	11	3	1
	System Element Exemplar -> name	14	3	1	11	3	1
	System Element Exemplar -> type	14	3	1	11	3	1
	FlowSpecification	18	8	3	18	8	1
	FlowSpecification -> name	18	8	3	18	8	1
	Operation	21	19	8	24	11	1
	Operation -> name	21	19	8	24	11	0
	Parameter	6	6	0	0	0	0
	Parameter -> name	6	6	0	0	0	0
	Port	123	17	16	66	32	11
	Port -> name	123	17	16	66	32	11
	Port -> type	123	17	16	66	32	11
	Port -> isConjugated	54	8	8	24	12	5
	Connector	88	11	10	44	7	6
	Connector -> name	88	11	10	44	7	6
Behavior - Sequences	Behavior - Sequence	4	4	0	7	4	0
	Behavior - Sequence -> name	4	4	0	7	4	0
	Lifeline	20	19	4	35	15	0
	Lifeline -> represents	20	19	4	35	15	1
	Message	30	28	14	38	14	0
	Message -> signature	30	28	14	38	14	1
	Message -> connector	30	28	14	38	14	5
	Message -> argument	3	3	0	0	0	0
Sum		**1008**	**342**	**153**	**748**	**296**	**65**

We count everything that is subject to manual effort in the context of the Software Requirements Engineers' modeling activities to yield a realistic approximation of their effort. This is reflected by several aspects described in the following.

Table A.7: Detailed results of the model element amount variables H2.3, H2.4, and H2.3-update

		# EBEAS			# CAS		
View Type	**Model Element (-> Attribute)**	**H2.3**	**H2.4**	**H2.3-update**	**H2.3**	**H2.4**	**H2.3-update**
Classifier	Component Type	21	23	4	13	13	1
	Component Type -> name	21	23	4	13	13	1
	Port	48	50	32	26	26	7
	Port -> name	48	50	32	26	26	7
	Port -> type	48	50	32	26	26	7
	Port -> isConjugated	24	25	16	13	13	3
	Interface	20	21	4	13	13	1
	Interface -> name	20	21	8	13	13	1
	Operation	60	65	24	20	20	1
	Operation -> name	60	65	16	20	20	1
	Parameter	20	24	0	3	3	0
	Parameter -> name	20	24	0	3	3	0
	Parameter -> type	20	24	0	3	3	0
Architecture	Collaboration	4	5	0	3	3	0
	Collaboration -> name	4	5	0	3	3	0
	Component Role	21	23	4	13	13	1
	Component Role -> name	21	23	4	13	13	1
	Component Role -> type	21	23	4	13	13	1
	Component Role -> partKind	21	23	4	13	13	1
	Connector	24	25	16	13	13	4
	Connector -> name	24	25	16	13	13	4
Interaction	MSD	4	22	0	4	4	0
	MSD -> name	4	22	0	4	4	0
	Lifeline	19	68	4	15	15	0
	Lifeline -> represents	19	68	4	15	15	1
	Message	28	86	15	14	14	0
	Message -> signature	28	86	15	14	14	4
	Message -> connector	28	86	15	14	14	4
	Message -> argument	3	15	0	3	3	1
	ModalMessage -> ExecutionKind	0	86	0	0	14	0
	ModalMessage -> Temperature	0	86	0	0	14	0
	ModalCondition	0	10	0	0	2	0
	ModalCondition -> Expression	0	10	0	0	2	0
	ModalCondition -> Temperature	0	10	0	0	2	0
	ClockReset	0	7	0	0	2	0
	ClockReset -> Expression	0	7	0	0	2	0
	TimeCondition	0	7	0	0	2	0
	TimeCondition -> Expression	0	7	0	0	2	0
	TimeCondition -> Temperature	0	7	0	0	2	0
Sum		**703**	**1307**	**273**	**359**	**403**	**52**

We do not only count the creation or update of a model element, but also the creation or update of the particular attributes of a model element. That is, if the Software Requirements Engineer has to specify a name of a model element or has to set a referential trace link or an attribute, we count such modeling activities. We only count mandatory attributes and neglect optional ones. For example, we do not count the name attribute of lifelines or messages since PAPYRUS renders the display name of these model elements depends on the lifeline's represents and the message's signature referential trace link, respectively. Thus, the Software Requirements Engineer does typically not specify the name attribute for these model elements.

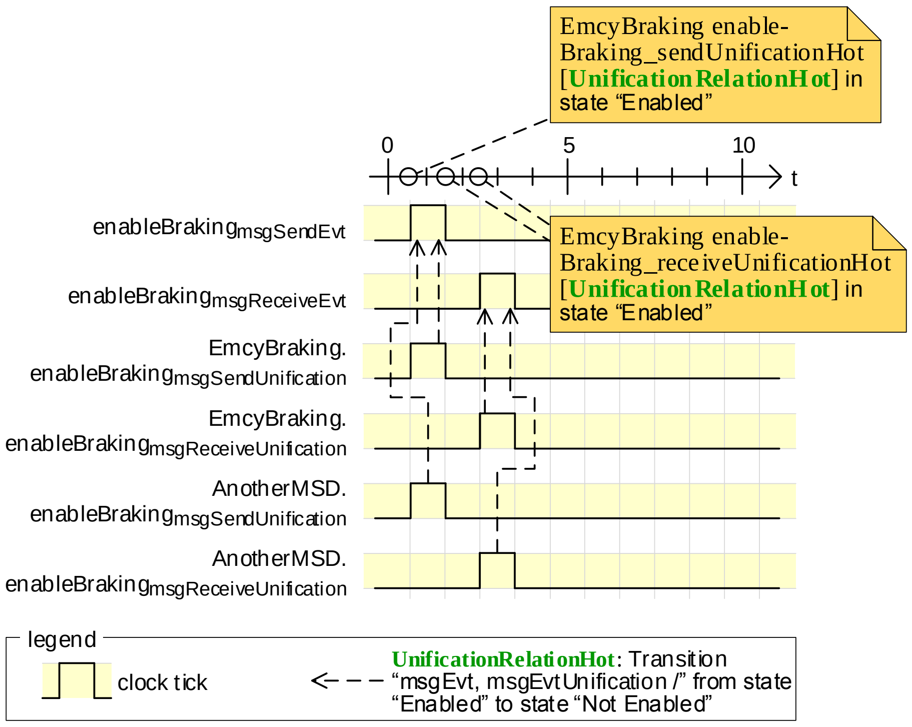
In the context of updates, we count one add and one remove activity as one activity if it can be conducted as an copy and paste activity. Furthermore, if a model element can be updated by changing one or more attributes, we count only the necessary change activities but do not assume that the Software Requirements Engineer deletes the model element and creates a new one as replacement. For instance, if the Software Requirements Engineer can update a lifeline by only changing its represents referential trace link, we do not count the model element itself or its name attribute.

We do not count the relational trace links in the system models as they do not have a corresponding representation in the MSD specifications. Furthermore, we do not count the *Behavior – States* in the system models since we do not derive parts of the MSD specifications from it but apply it as the basis for our coverage check between MSD specifications and system models.

B

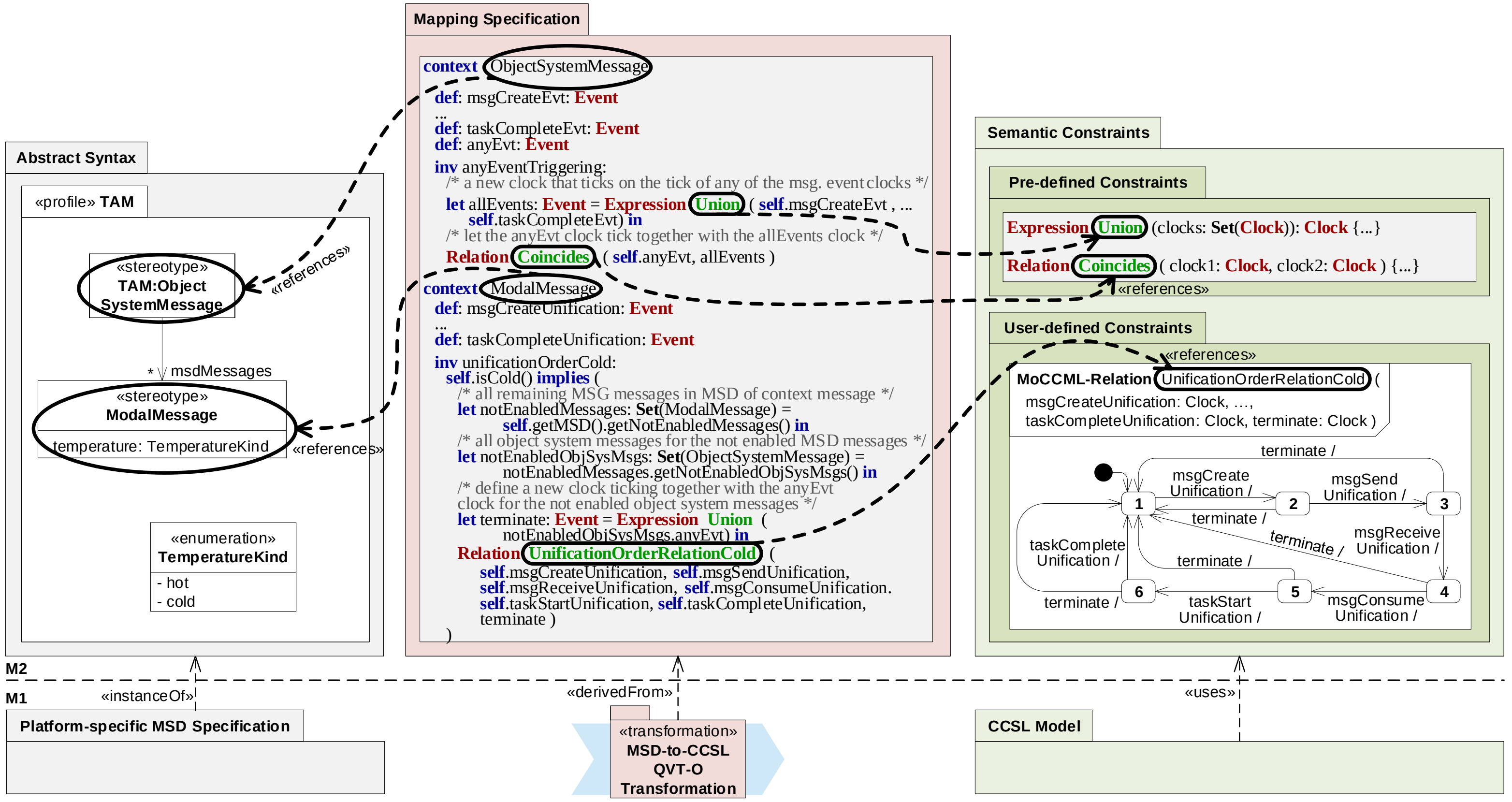
Supplementary Material on the MSD Semantics for Timing Analysis

B.1 Further Examples of the MSD Semantics for Timing Analyses



EmcyBraking enableBraking_sendUnificationHot [UnificationRelationHot] in state "Enabled"
EmcyBraking enableBraking_receiveUnificationHot [UnificationRelationHot] in state "Enabled"
0
5
10
t
enableBraking$_{msgSendEvt}$
enableBraking$_{msgReceiveEvt}$
EmcyBraking. enableBraking$_{msgSendUnification}$
EmcyBraking. enableBraking$_{msgReceiveUnification}$
AnotherMSD. enableBraking$_{msgSendUnification}$
AnotherMSD. enableBraking$_{msgReceiveUnification}$
legend
clock tick
UnificationRelationHot: Transition "msgEvt, msgEvtUnification /" from state "Enabled" to state "Not Enabled"

Figure B.1: Example—message event unification with multiple hot MSD messages in CCSL simulation

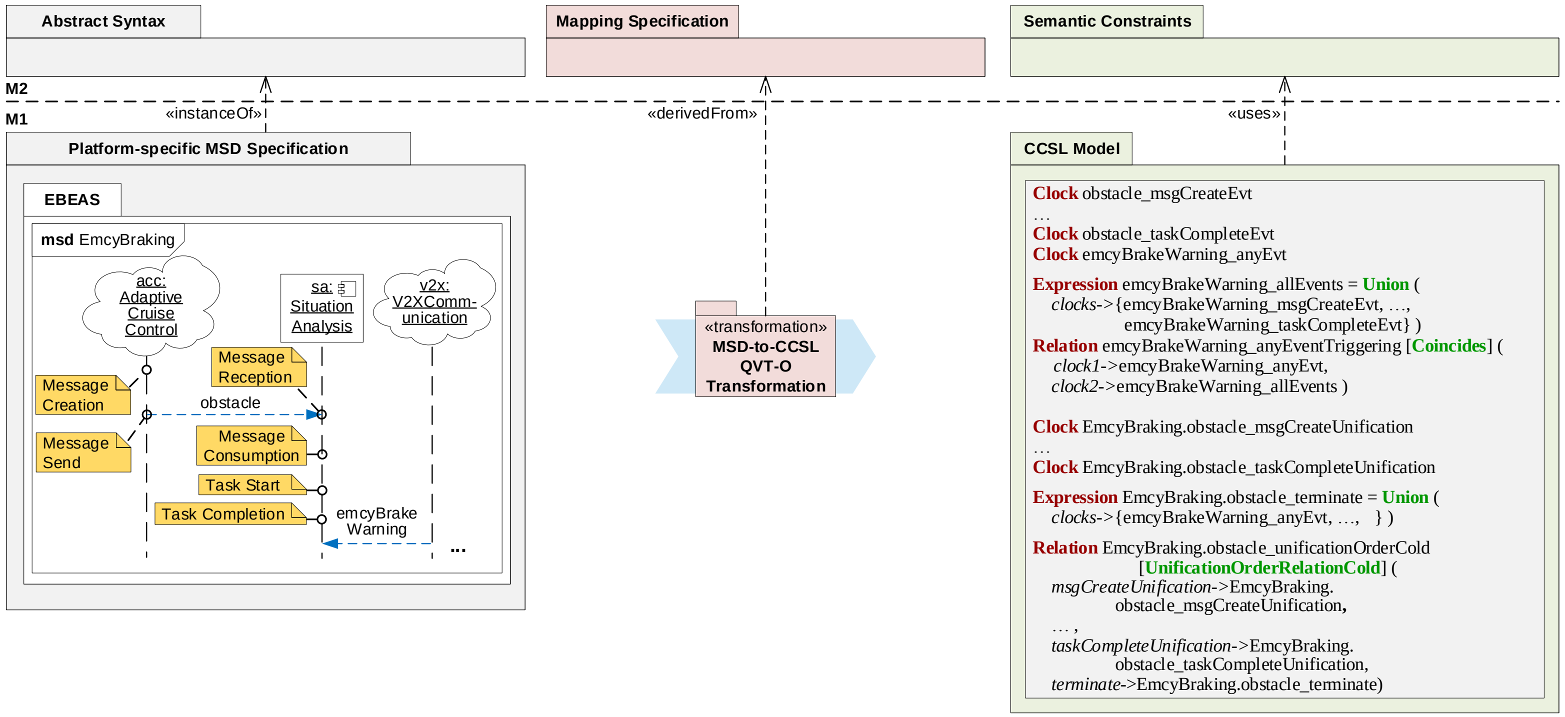


```
context ObjectSystemMessage
  def: msgCreateEvt: Event
  ...
  def: taskCompleteEvt: Event
  def: anyEvt: Event
  inv anyEventTriggering:
    /* a new clock that ticks on the tick of any of the msg. event clocks */
    let allEvents: Event = Expression Union ( self.msgCreateEvt , ...
        self.taskCompleteEvt) in
    /* let the anyEvt clock tick together with the allEvents clock */
    Relation Coincides ( self.anyEvt, allEvents )
context ModalMessage
  def: msgCreateUnification: Event
  ...
  def: taskCompleteUnification: Event
  inv unificationOrderCold:
    self.isCold() implies (
      /* all remaining MSG messages in MSD of context message */
      let notEnabledMessages: Set(ModalMessage) =
            self.getMSD().getNotEnabledMessages() in
      /* all object system messages for the not enabled MSD messages */
      let notEnabledObjSysMsgs: Set(ObjectSystemMessage) =
            notEnabledMessages.getNotEnabledObjSysMsgs() in
      /* define a new clock ticking together with the anyEvt
      clock for the not enabled object system messages */
      let terminate: Event = Expression  Union  (
            notEnabledObjSysMsgs.anyEvt) in
      Relation UnificationOrderRelationCold  (
        self.msgCreateUnification,  self.msgSendUnification,
        self.msgReceiveUnification,  self.msgConsumeUnification.
        self.taskStartUnification, self.taskCompleteUnification,
        terminate )
  )
```

```
Expression Union (clocks: Set(Clock)): Clock {...}
Relation Coincides ( clock1: Clock, clock2: Clock ) {...}
```

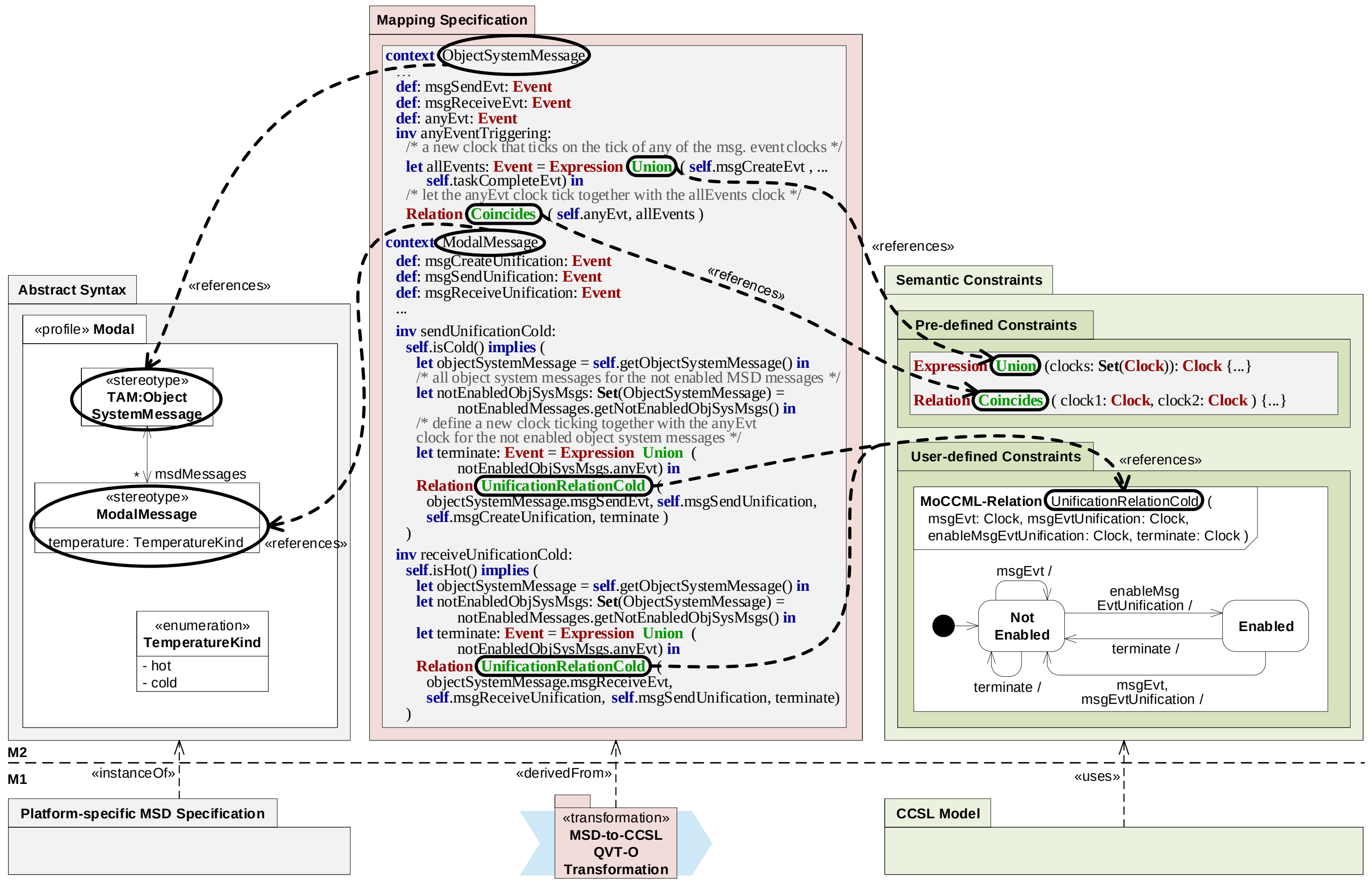
```
MoCCML-Relation UnificationOrderRelationCold (
  msgCreateUnification: Clock, ...,
  taskCompleteUnification: Clock, terminate: Clock )
```

Figure B.2: Specification excerpt of semantics for the order of unification occurrences for cold messages with additional event kinds (focus on metamodel level M2)



```
Clock obstacle_msgCreateEvt
…
Clock obstacle_taskCompleteEvt
Clock emcyBrakeWarning_anyEvt
Expression emcyBrakeWarning_allEvents = Union (
    clocks->{emcyBrakeWarning_msgCreateEvt, …,
               emcyBrakeWarning_taskCompleteEvt} )
Relation emcyBrakeWarning_anyEventTriggering [Coincides] (
    clock1->emcyBrakeWarning_anyEvt,
    clock2->emcyBrakeWarning_allEvents )

Clock EmcyBraking.obstacle_msgCreateUnification
…
Clock EmcyBraking.obstacle_taskCompleteUnification
Expression EmcyBraking.obstacle_terminate = Union (
    clocks->{emcyBrakeWarning_anyEvt, …,    } )
Relation EmcyBraking.obstacle_unificationOrderCold
                  [UnificationOrderRelationCold] (
    msgCreateUnification->EmcyBraking.
            obstacle_msgCreateUnification,
    … ,
    taskCompleteUnification->EmcyBraking.
            obstacle_taskCompleteUnification,
    terminate->EmcyBraking.obstacle_terminate)
```

Figure B.3: Example models of semantics for the order of unification occurrences for cold messages with additional event kinds (focus on metamodel level M1)

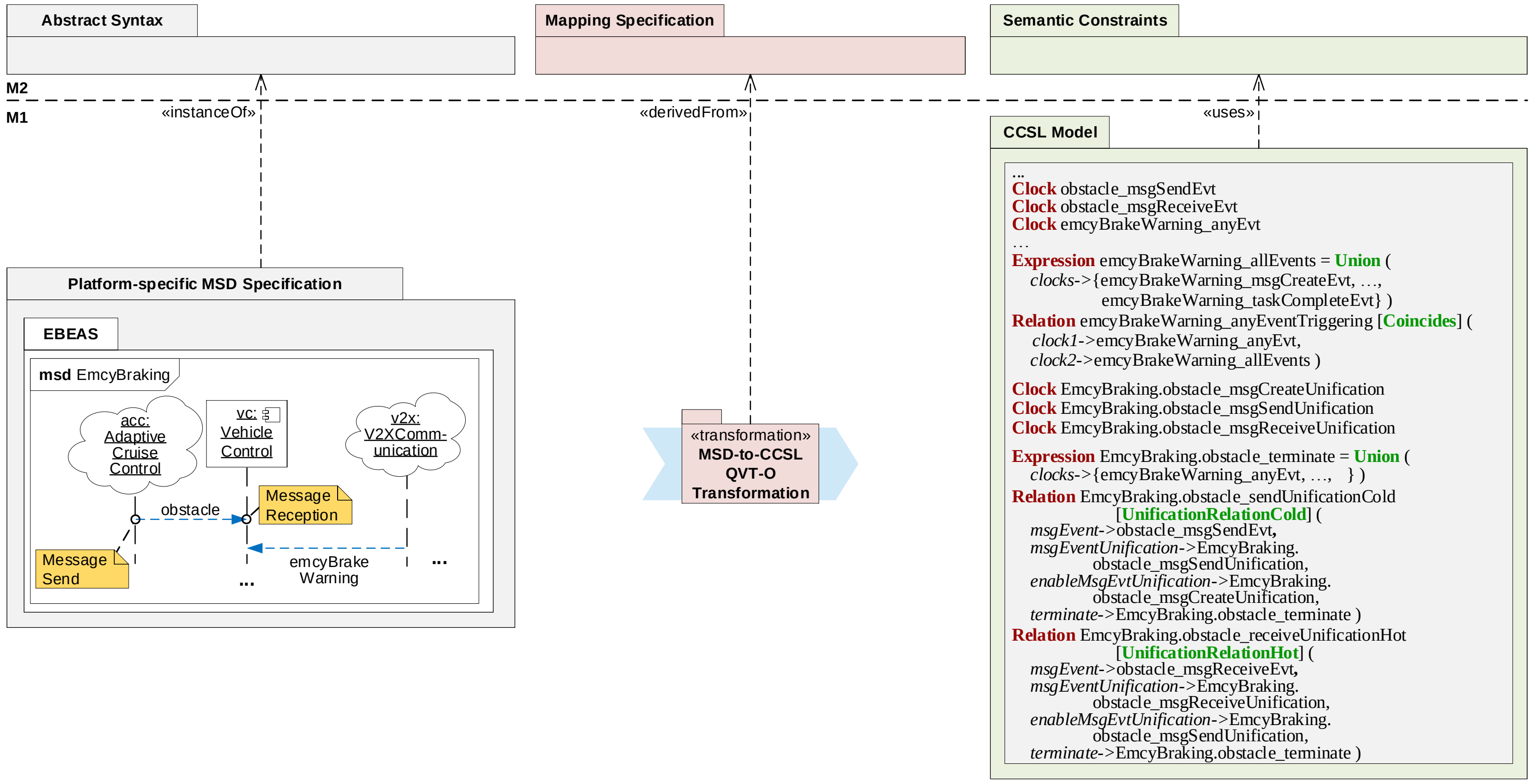


```
context ObjectSystemMessage
  ...
  def: msgSendEvt: Event
  def: msgReceiveEvt: Event
  def: anyEvt: Event
  inv anyEventTriggering:
    /* a new clock that ticks on the tick of any of the msg. event clocks */
    let allEvents: Event = Expression Union ( self.msgCreateEvt , ...
        self.taskCompleteEvt) in
    /* let the anyEvt clock tick together with the allEvents clock */
    Relation Coincides ( self.anyEvt, allEvents )
context ModalMessage
  def: msgCreateUnification: Event
  def: msgSendUnification: Event
  def: msgReceiveUnification: Event
  ...
  inv sendUnificationCold:
    self.isCold() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      /* all object system messages for the not enabled MSD messages */
      let notEnabledObjSysMsgs: Set(ObjectSystemMessage) =
              notEnabledMessages.getNotEnabledObjSysMsgs() in
      /* define a new clock ticking together with the anyEvt
      clock for the not enabled object system messages */
      let terminate: Event = Expression Union (
              notEnabledObjSysMsgs.anyEvt) in
      Relation UnificationRelationCold (
        objectSystemMessage.msgSendEvt, self.msgSendUnification,
        self.msgCreateUnification, terminate )
    )
  inv receiveUnificationCold:
    self.isHot() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      let notEnabledObjSysMsgs: Set(ObjectSystemMessage) =
              notEnabledMessages.getNotEnabledObjSysMsgs() in
      let terminate: Event = Expression Union (
              notEnabledObjSysMsgs.anyEvt) in
      Relation UnificationRelationCold (
        objectSystemMessage.msgReceiveEvt,
        self.msgReceiveUnification, self.msgSendUnification, terminate)
    )
```

```
Expression Union (clocks: Set(Clock)): Clock {...}
Relation Coincides ( clock1: Clock, clock2: Clock ) {...}
```

```
MoCCML-Relation UnificationRelationCold (
  msgEvt: Clock, msgEvtUnification: Clock,
  enableMsgEvtUnification: Clock, terminate: Clock )
```

Figure B.4: Specification excerpt of semantics for unification of cold messages (focus on metamodel level M2)



```
...
Clock obstacle_msgSendEvt
Clock obstacle_msgReceiveEvt
Clock emcyBrakeWarning_anyEvt
...
Expression emcyBrakeWarning_allEvents = Union (
   clocks->{emcyBrakeWarning_msgCreateEvt, ...,
             emcyBrakeWarning_taskCompleteEvt} )
Relation emcyBrakeWarning_anyEventTriggering [Coincides] (
   clock1->emcyBrakeWarning_anyEvt,
   clock2->emcyBrakeWarning_allEvents )

Clock EmcyBraking.obstacle_msgCreateUnification
Clock EmcyBraking.obstacle_msgSendUnification
Clock EmcyBraking.obstacle_msgReceiveUnification

Expression EmcyBraking.obstacle_terminate = Union (
   clocks->{emcyBrakeWarning_anyEvt, ...,   } )
Relation EmcyBraking.obstacle_sendUnificationCold
              [UnificationRelationCold] (
   msgEvent->obstacle_msgSendEvt,
   msgEventUnification->EmcyBraking.
          obstacle_msgSendUnification,
   enableMsgEvtUnification->EmcyBraking.
          obstacle_msgCreateUnification,
   terminate->EmcyBraking.obstacle_terminate )
Relation EmcyBraking.obstacle_receiveUnificationHot
              [UnificationRelationHot] (
   msgEvent->obstacle_msgReceiveEvt,
   msgEventUnification->EmcyBraking.
          obstacle_msgReceiveUnification,
   enableMsgEvtUnification->EmcyBraking.
          obstacle_msgSendUnification,
   terminate->EmcyBraking.obstacle_terminate )
```

Figure B.5: Example models of semantics of semantics for unification of cold messages (focus on metamodel level M2)

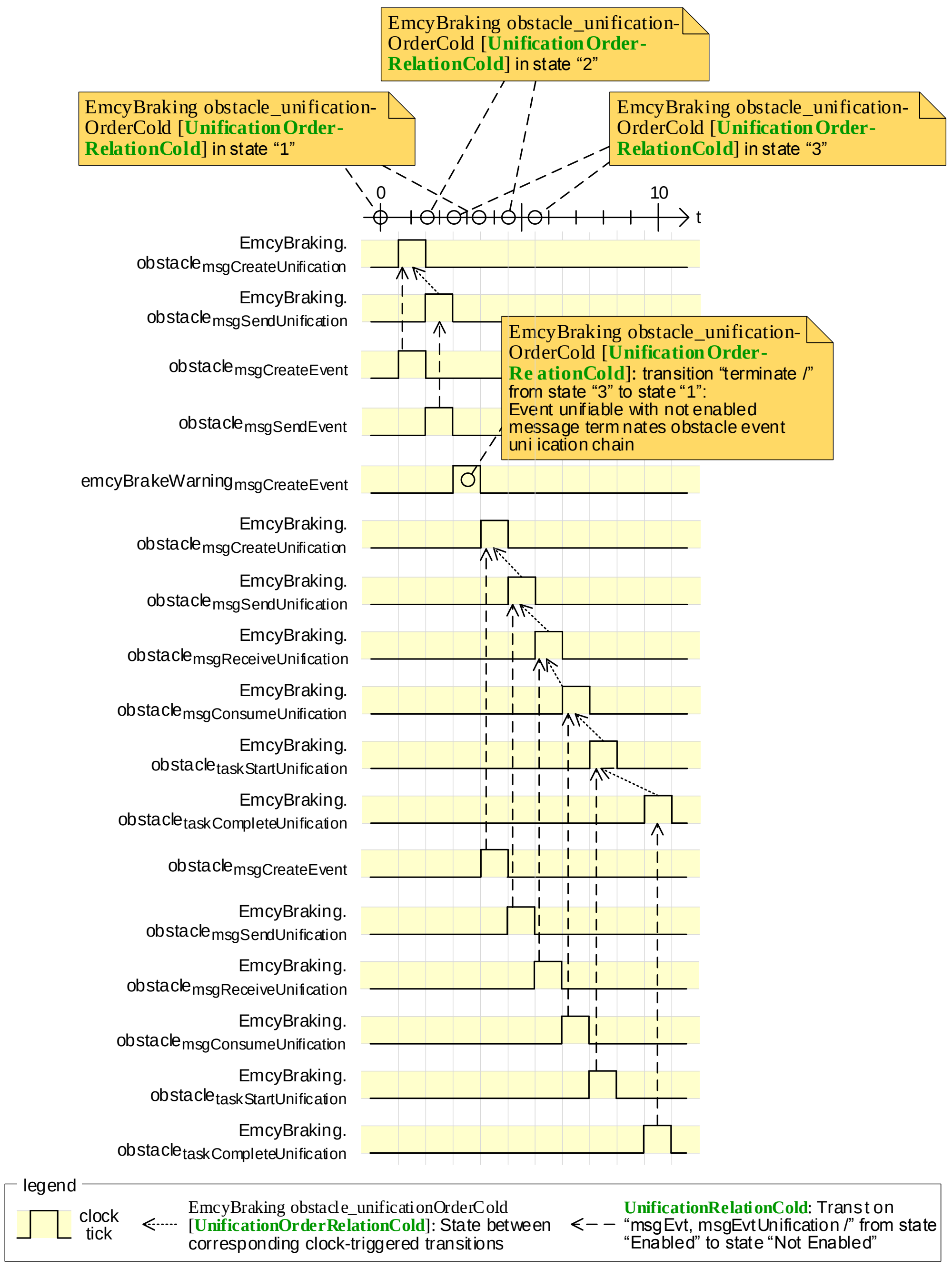

EmcyBraking obstacle_unification-OrderCold [UnificationOrder-RelationCold] in state “2”
EmcyBraking obstacle_unification-OrderCold [UnificationOrder-RelationCold] in state “1”
EmcyBraking obstacle_unification-OrderCold [UnificationOrder-RelationCold] in state “3”
0
10
t
EmcyBraking. obstacle msgCreateUnification
EmcyBraking. obstacle msgSendUnification
obstacle msgCreateEvent
obstacle msgSendEvent
EmcyBraking obstacle_unification-OrderCold [UnificationOrder-RelationCold]: transition “terminate /” from state “3” to state “1”: Event unifiable with not enabled message terminates obstacle event unification chain
emcyBrakeWarning msgCreateEvent
EmcyBraking. obstacle msgCreateUnification
EmcyBraking. obstacle msgSendUnification
EmcyBraking. obstacle msgReceiveUnification
EmcyBraking. obstacle msgConsumeUnification
EmcyBraking. obstacle taskStartUnification
EmcyBraking. obstacle taskCompleteUnification
obstacle msgCreateEvent
EmcyBraking. obstacle msgSendUnification
EmcyBraking. obstacle msgReceiveUnification
EmcyBraking. obstacle msgConsumeUnification
EmcyBraking. obstacle taskStartUnification
EmcyBraking. obstacle taskCompleteUnification
legend
clock tick
EmcyBraking obstacle_unificationOrderCold [UnificationOrderRelationCold]: State between corresponding clock-triggered transitions
UnificationRelationCold: Transition “msgEvt, msgEvtUnification /” from state “Enabled” to state “Not Enabled”

Figure B.6: Example—order of message unification occurrences and message event unification in CCSL simulation for cold messages (cf. CCSL models in the lower right of Figure B.3 and Figure B.5)

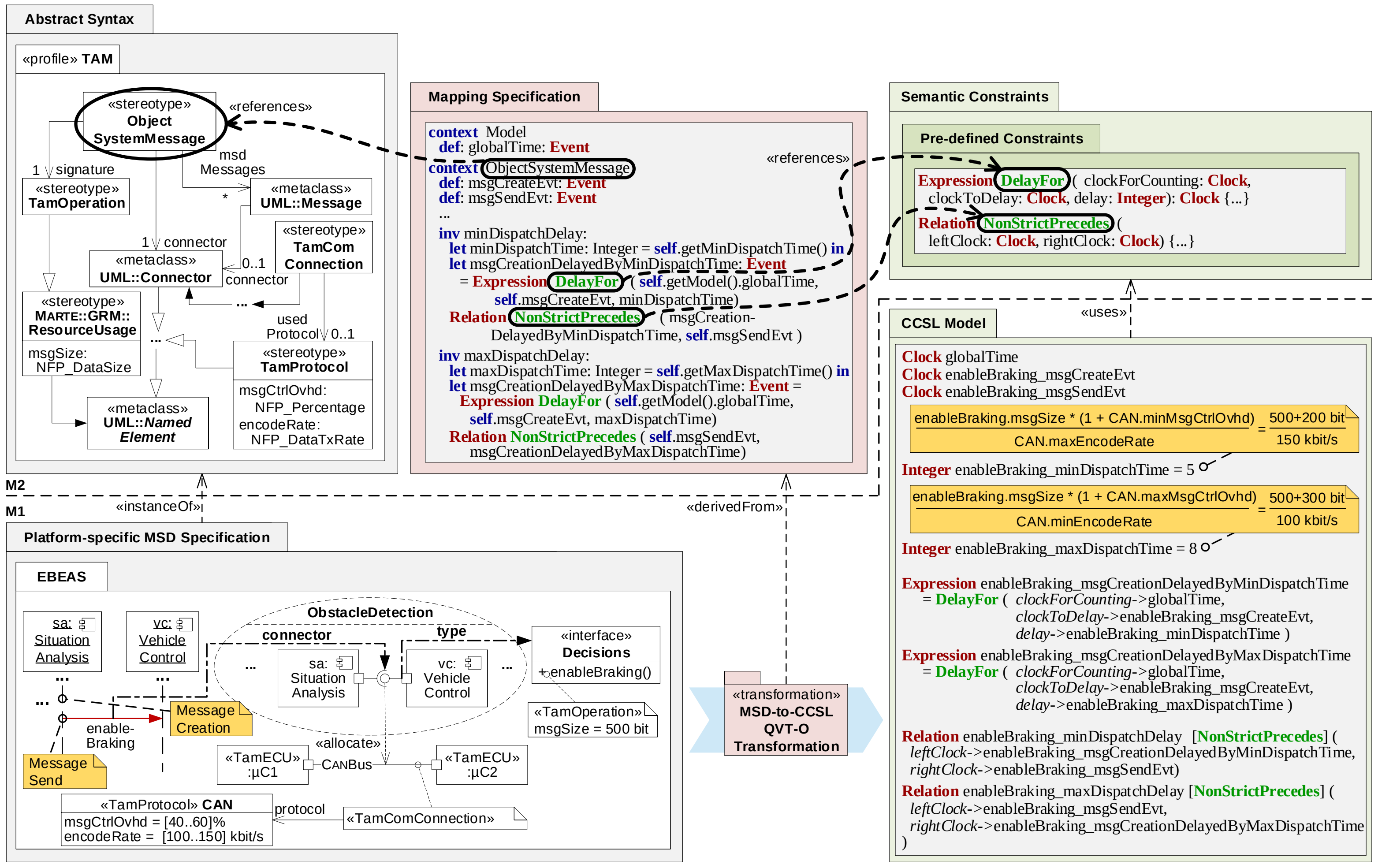

Abstract Syntax
«profile» TAM
«stereotype» Object SystemMessage
«references»
1 signature
«stereotype» TamOperation
msd Messages
* «metaclass» UML::Message
1 connector
«metaclass» UML::Connector
0..1 connector
«stereotype» TamCom Connection
«stereotype» MARTE::GRM:: ResourceUsage
msgSize: NFP_DataSize
used Protocol 0..1
«stereotype» TamProtocol
msgCtrlOvhd: NFP_Percentage
encodeRate: NFP_DataTxRate
«metaclass» UML::Named Element
Mapping Specification
context Model
def: globalTime: Event
context ObjectSystemMessage
def: msgCreateEvt: Event
def: msgSendEvt: Event
...
inv minDispatchDelay:
let minDispatchTime: Integer = self.getMinDispatchTime() in
let msgCreationDelayedByMinDispatchTime: Event
= Expression DelayFor (self.getModel().globalTime,
self.msgCreateEvt, minDispatchTime)
Relation NonStrictPrecedes (msgCreation-DelayedByMinDispatchTime, self.msgSendEvt)
inv maxDispatchDelay:
let maxDispatchTime: Integer = self.getMaxDispatchTime() in
let msgCreationDelayedByMaxDispatchTime: Event =
Expression DelayFor (self.getModel().globalTime,
self.msgCreateEvt, maxDispatchTime)
Relation NonStrictPrecedes (self.msgSendEvt,
msgCreationDelayedByMaxDispatchTime)
«references»
Semantic Constraints
Pre-defined Constraints
Expression DelayFor(clockForCounting: Clock,
clockToDelay: Clock, delay: Integer): Clock {...}
Relation NonStrictPrecedes (
leftClock: Clock, rightClock: Clock) {...}
«uses»
CCSL Model
Clock globalTime
Clock enableBraking_msgCreateEvt
Clock enableBraking_msgSendEvt
enableBraking.msgSize * (1 + CAN.minMsgCtrlOvhd) / CAN.maxEncodeRate = 500+200 bit / 150 kbit/s
Integer enableBraking_minDispatchTime = 5
enableBraking.msgSize * (1 + CAN.maxMsgCtrlOvhd) / CAN.minEncodeRate = 500+300 bit / 100 kbit/s
Integer enableBraking_maxDispatchTime = 8
Expression enableBraking_msgCreationDelayedByMinDispatchTime
= DelayFor (clockForCounting->globalTime,
clockToDelay->enableBraking_msgCreateEvt,
delay->enableBraking_minDispatchTime)
Expression enableBraking_msgCreationDelayedByMaxDispatchTime
= DelayFor (clockForCounting->globalTime,
clockToDelay->enableBraking_msgCreateEvt,
delay->enableBraking_maxDispatchTime)
Relation enableBraking_minDispatchDelay [NonStrictPrecedes] (
leftClock->enableBraking_msgCreationDelayedByMinDispatchTime,
rightClock->enableBraking_msgSendEvt)
Relation enableBraking_maxDispatchDelay [NonStrictPrecedes] (
leftClock->enableBraking_msgSendEvt,
rightClock->enableBraking_msgCreationDelayedByMaxDispatchTime
)
M2
M1
«instanceOf»
«derivedFrom»
Platform-specific MSD Specification
EBEAS
sa: Situation Analysis
vc: Vehicle Control
ObstacleDetection
connector
type
«interface» Decisions
+enableBraking()
sa: Situation Analysis
vc: Vehicle Control
Message Creation
enable-Braking
Message Send
«TamOperation» msgSize = 500 bit
«allocate»
«TamECU» :µC1
CANBus
«TamECU» :µC2
«TamProtocol» CAN
msgCtrlOvhd = [40..60]%
encodeRate = [100..150] kbit/s
protocol
«TamComConnection»
«transformation» MSD-to-CCSL QVT-O Transformation

Figure B.7: Specification excerpt of semantics for distributed message dispatch delays including example models

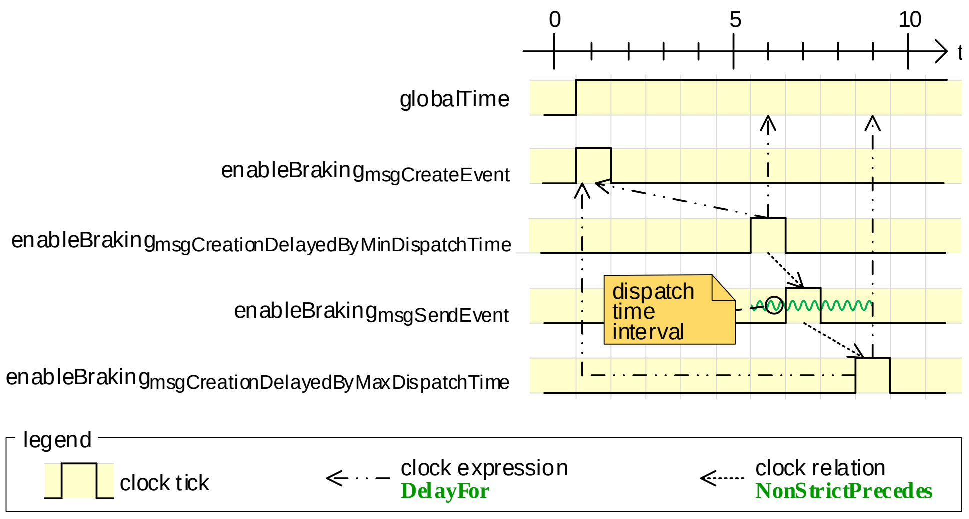

0
5
10
t
globalTime
enableBraking$_{msgCreateEvent}$
enableBraking$_{msgCreationDelayedByMinDispatchTime}$
enableBraking$_{msgSendEvent}$
enableBraking$_{msgCreationDelayedByMaxDispatchTime}$
dispatch time interval
legend
clock tick
clock expression
DelayFor
clock relation
NonStrictPrecedes

Figure B.8: Exemplary CCSL run simulating distributed message dispatch delays (cf. CCSL model in the lower right of Figure B.7)

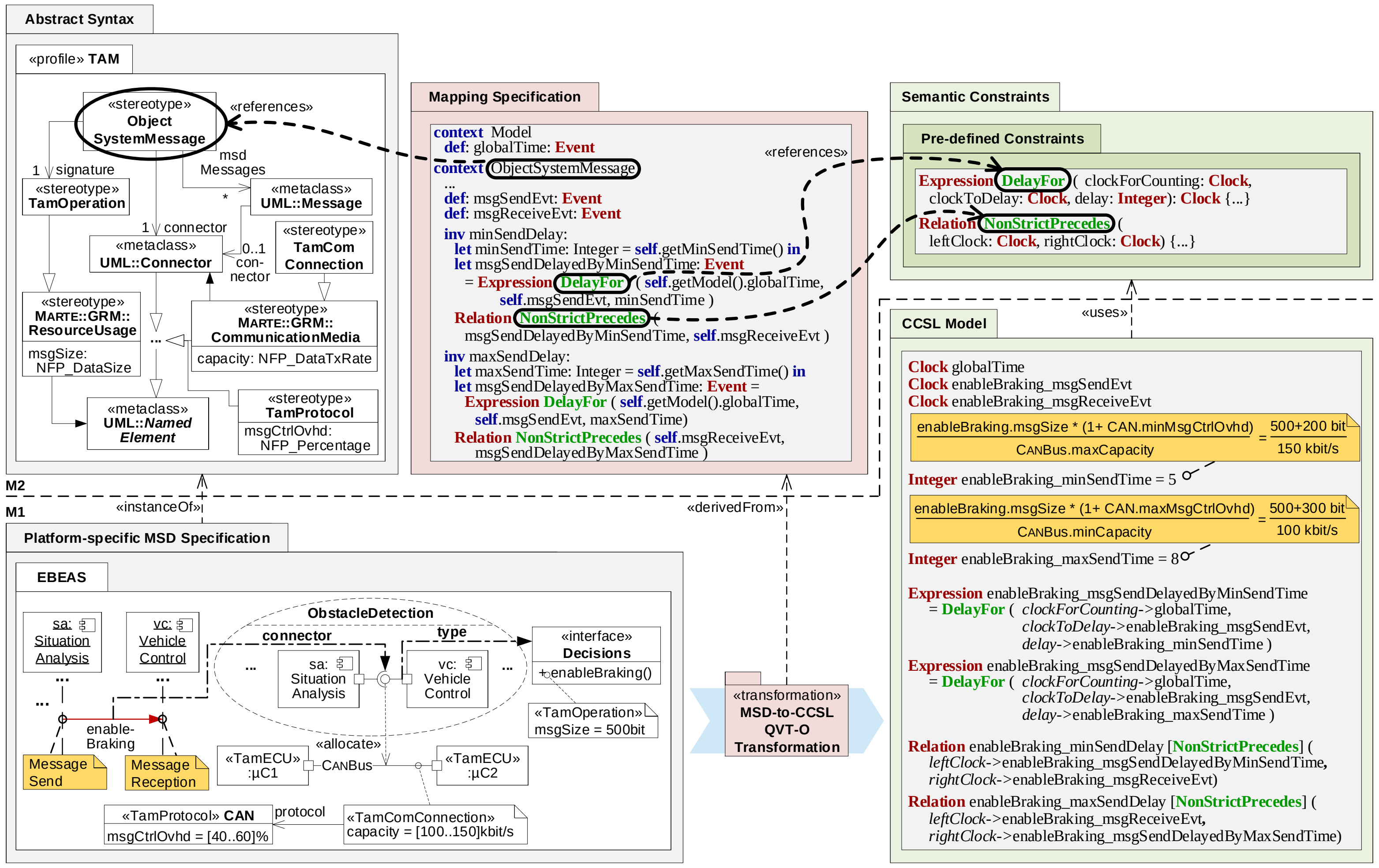

Abstract Syntax
«profile» TAM
«stereotype» Object SystemMessage
«references»
1 signature
msd Messages
«stereotype» TamOperation
«metaclass» UML::Message
1 connector
«metaclass» UML::Connector
0..1 con-nector
«stereotype» TamCom Connection
«stereotype» MARTE::GRM:: ResourceUsage
msgSize: NFP_DataSize
«stereotype» MARTE::GRM:: CommunicationMedia
capacity: NFP_DataTxRate
«metaclass» UML::Named Element
«stereotype» TamProtocol
msgCtrlOvhd: NFP_Percentage
Mapping Specification
context Model
def: globalTime: Event
context ObjectSystemMessage
def: msgSendEvt: Event
def: msgReceiveEvt: Event
inv minSendDelay:
let minSendTime: Integer = self.getMinSendTime() in
let msgSendDelayedByMinSendTime: Event
= Expression DelayFor (self.getModel().globalTime, self.msgSendEvt, minSendTime)
Relation NonStrictPrecedes (msgSendDelayedByMinSendTime, self.msgReceiveEvt)
inv maxSendDelay:
let maxSendTime: Integer = self.getMaxSendTime() in
let msgSendDelayedByMaxSendTime: Event =
Expression DelayFor (self.getModel().globalTime, self.msgSendEvt, maxSendTime)
Relation NonStrictPrecedes (self.msgReceiveEvt, msgSendDelayedByMaxSendTime)
«references»
Semantic Constraints
Pre-defined Constraints
Expression DelayFor(clockForCounting: Clock, clockToDelay: Clock, delay: Integer): Clock {...}
Relation NonStrictPrecedes (leftClock: Clock, rightClock: Clock) {...}
«uses»
CCSL Model
Clock globalTime
Clock enableBraking_msgSendEvt
Clock enableBraking_msgReceiveEvt
enableBraking.msgSize * (1+ CAN.minMsgCtrlOvhd) / CANBus.maxCapacity = 500+200 bit / 150 kbit/s
Integer enableBraking_minSendTime = 5
enableBraking.msgSize * (1+ CAN.maxMsgCtrlOvhd) / CANBus.minCapacity = 500+300 bit / 100 kbit/s
Integer enableBraking_maxSendTime = 8
Expression enableBraking_msgSendDelayedByMinSendTime = DelayFor (clockForCounting->globalTime, clockToDelay->enableBraking_msgSendEvt, delay->enableBraking_minSendTime)
Expression enableBraking_msgSendDelayedByMaxSendTime = DelayFor (clockForCounting->globalTime, clockToDelay->enableBraking_msgSendEvt, delay->enableBraking_maxSendTime)
Relation enableBraking_minSendDelay [NonStrictPrecedes] (leftClock->enableBraking_msgSendDelayedByMinSendTime, rightClock->enableBraking_msgReceiveEvt)
Relation enableBraking_maxSendDelay [NonStrictPrecedes] (leftClock->enableBraking_msgReceiveEvt, rightClock->enableBraking_msgSendDelayedByMaxSendTime)
M2
M1
«instanceOf»
«derivedFrom»
Platform-specific MSD Specification
EBEAS
sa: Situation Analysis
vc: Vehicle Control
enable-Braking
Message Send
Message Reception
ObstacleDetection
connector
type
sa: Situation Analysis
vc: Vehicle Control
«interface» Decisions
+enableBraking()
«TamOperation» msgSize = 500bit
«allocate»
«TamECU» :µC1
CANBus
«TamECU» :µC2
«TamProtocol» CAN
msgCtrlOvhd = [40..60]%
protocol
«TamComConnection» capacity = [100..150]kbit/s
«transformation» MSD-to-CCSL QVT-O Transformation

Figure B.9: Specification excerpt of semantics for distributed message send delays including example models

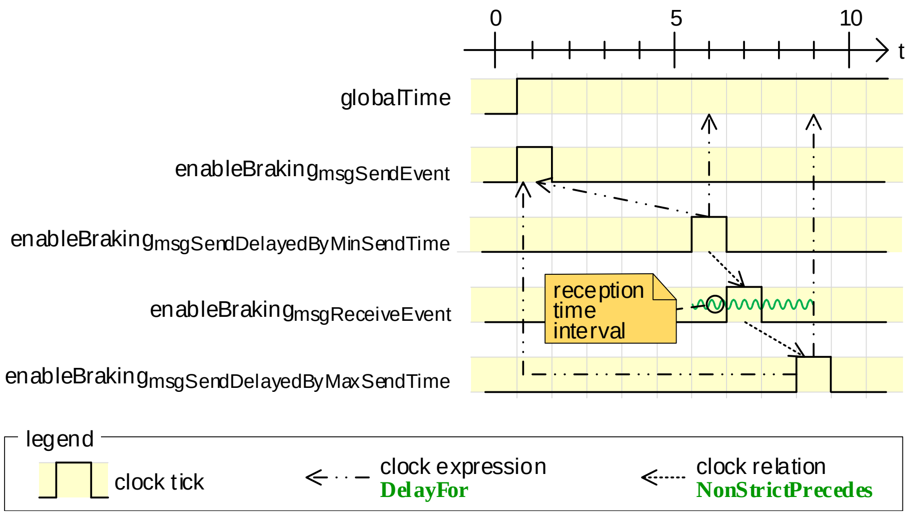
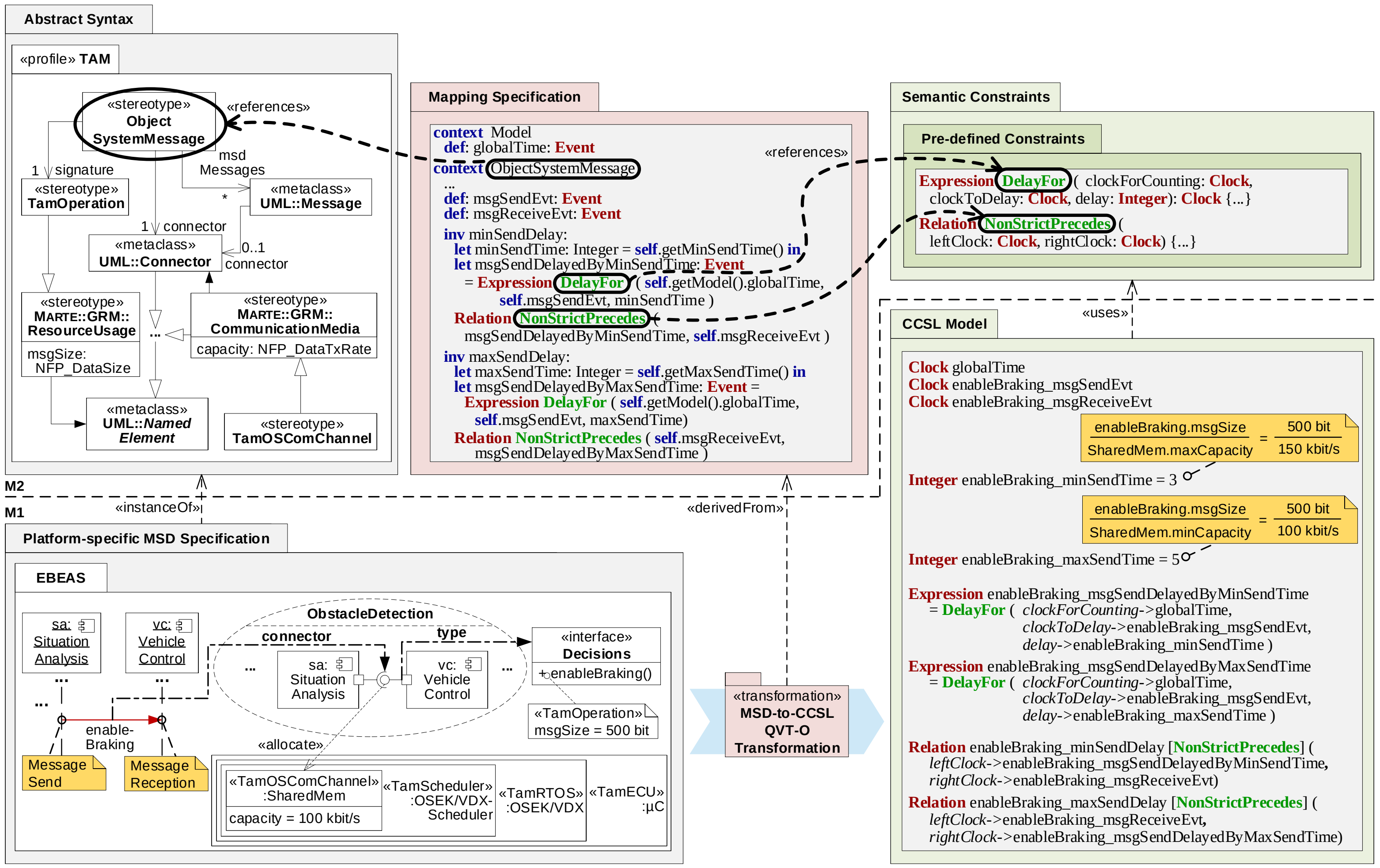

0
5
10
t
globalTime
enableBraking$_{msgSendEvent}$
enableBraking$_{msgSendDelayedByMinSendTime}$
enableBraking$_{msgReceiveEvent}$
enableBraking$_{msgSendDelayedByMaxSendTime}$
reception time interval
legend
clock tick
clock expression
DelayFor
clock relation
NonStrictPrecedes

Figure B.10: Exemplary CCSL run simulating distributed message send delays (cf. CCSL model in the lower right of Figure B.9)



Mapping Specification

```
context  Model
  def: globalTime: Event
context ObjectSystemMessage
  ...
  def: msgSendEvt: Event
  def: msgReceiveEvt: Event
  inv minSendDelay:
    let minSendTime: Integer = self.getMinSendTime() in
    let msgSendDelayedByMinSendTime: Event
      = Expression DelayFor ( self.getModel().globalTime,
            self.msgSendEvt, minSendTime )
    Relation NonStrictPrecedes (
      msgSendDelayedByMinSendTime, self.msgReceiveEvt )
  inv maxSendDelay:
    let maxSendTime: Integer = self.getMaxSendTime() in
    let msgSendDelayedByMaxSendTime: Event =
      Expression DelayFor ( self.getModel().globalTime,
        self.msgSendEvt, maxSendTime)
    Relation NonStrictPrecedes ( self.msgReceiveEvt,
        msgSendDelayedByMaxSendTime )
```

Semantic Constraints — Pre-defined Constraints

```
Expression DelayFor( clockForCounting: Clock,
  clockToDelay: Clock, delay: Integer): Clock {...}
Relation NonStrictPrecedes (
  leftClock: Clock, rightClock: Clock) {...}
```

CCSL Model

```
Clock globalTime
Clock enableBraking_msgSendEvt
Clock enableBraking_msgReceiveEvt
Integer enableBraking_minSendTime = 3
Integer enableBraking_maxSendTime = 5

Expression enableBraking_msgSendDelayedByMinSendTime
    = DelayFor (  clockForCounting->globalTime,
                  clockToDelay->enableBraking_msgSendEvt,
                  delay->enableBraking_minSendTime )
Expression enableBraking_msgSendDelayedByMaxSendTime
    = DelayFor (  clockForCounting->globalTime,
                  clockToDelay->enableBraking_msgSendEvt,
                  delay->enableBraking_maxSendTime )

Relation enableBraking_minSendDelay [NonStrictPrecedes] (
    leftClock->enableBraking_msgSendDelayedByMinSendTime,
    rightClock->enableBraking_msgReceiveEvt)
Relation enableBraking_maxSendDelay [NonStrictPrecedes] (
    leftClock->enableBraking_msgReceiveEvt,
    rightClock->enableBraking_msgSendDelayedByMaxSendTime)
```

$\frac{\text{enableBraking.msgSize}}{\text{SharedMem.maxCapacity}} = \frac{500\ \text{bit}}{150\ \text{kbit/s}}$

$\frac{\text{enableBraking.msgSize}}{\text{SharedMem.minCapacity}} = \frac{500\ \text{bit}}{100\ \text{kbit/s}}$

Figure B.11: Specification excerpt of semantics for internal message send delays including example models

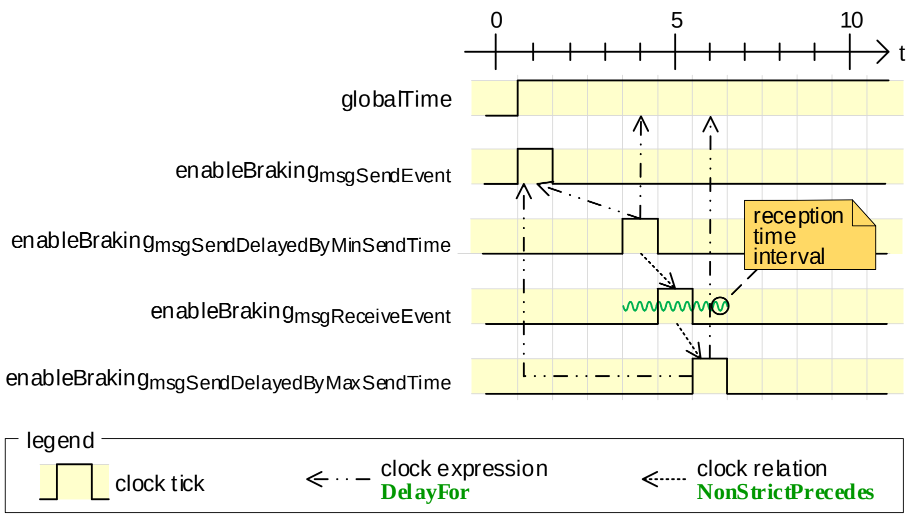
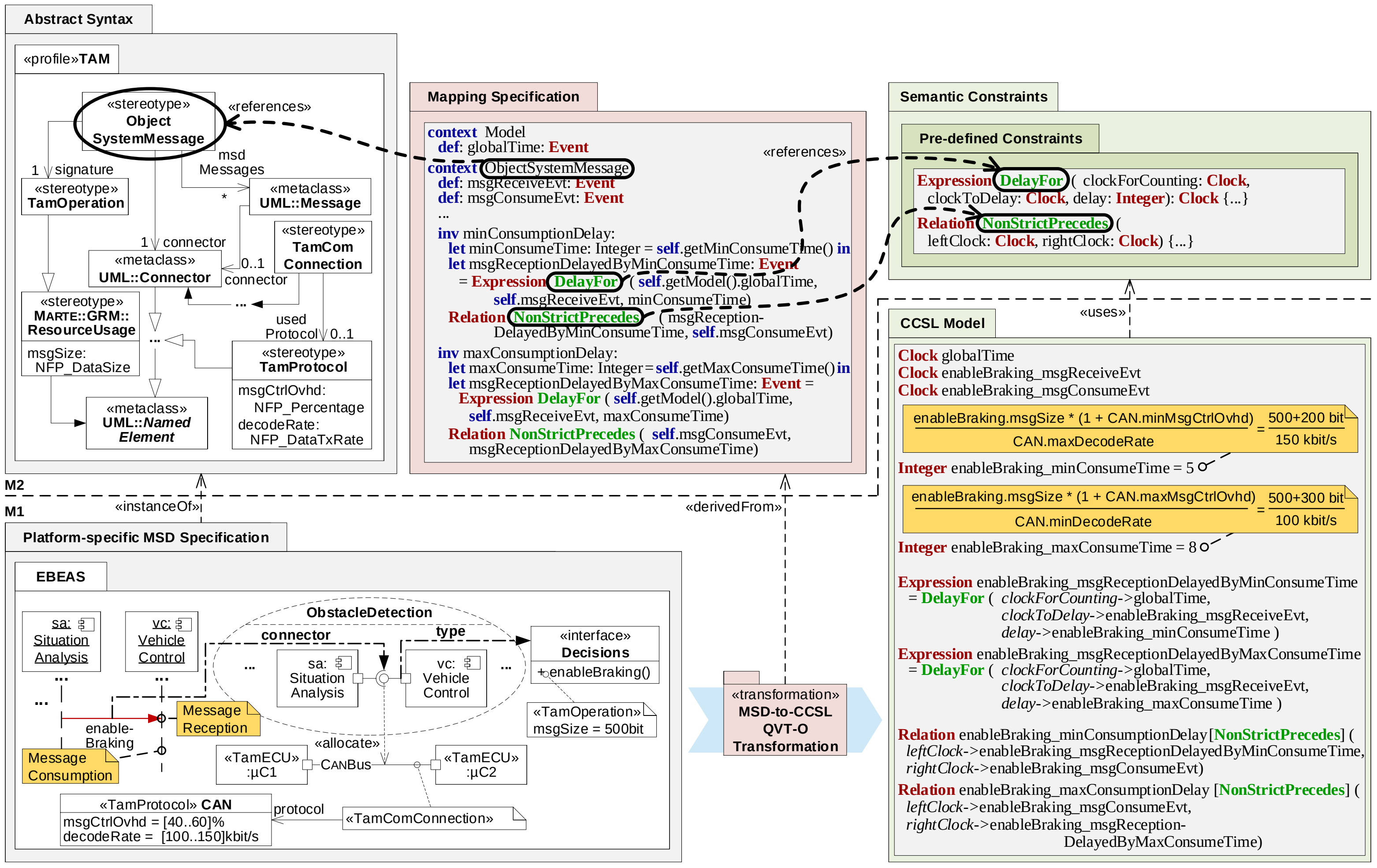

0
5
10
t
globalTime
enableBraking$_{msgSendEvent}$
enableBraking$_{msgSendDelayedByMinSendTime}$
enableBraking$_{msgReceiveEvent}$
enableBraking$_{msgSendDelayedByMaxSendTime}$
reception time interval
legend
clock tick
clock expression
DelayFor
clock relation
NonStrictPrecedes

Figure B.12: Exemplary CCSL run simulating internal message send delays (cf. CCSL model in the lower right of Figure B.11)



```
context  Model
  def: globalTime: Event
context ObjectSystemMessage
  def: msgReceiveEvt: Event
  def: msgConsumeEvt: Event
  ...
  inv minConsumptionDelay:
    let minConsumeTime: Integer = self.getMinConsumeTime() in
    let msgReceptionDelayedByMinConsumeTime: Event
      = Expression DelayFor ( self.getModel().globalTime,
            self.msgReceiveEvt, minConsumeTime)
    Relation NonStrictPrecedes  ( msgReception-
            DelayedByMinConsumeTime, self.msgConsumeEvt)
  inv maxConsumptionDelay:
    let maxConsumeTime: Integer = self.getMaxConsumeTime() in
    let msgReceptionDelayedByMaxConsumeTime: Event =
      Expression DelayFor ( self.getModel().globalTime,
        self.msgReceiveEvt, maxConsumeTime)
    Relation NonStrictPrecedes (  self.msgConsumeEvt,
        msgReceptionDelayedByMaxConsumeTime)
```

```
Expression DelayFor ( clockForCounting: Clock,
  clockToDelay: Clock, delay: Integer): Clock {...}
Relation NonStrictPrecedes (
  leftClock: Clock, rightClock: Clock) {...}
```

```
Clock globalTime
Clock enableBraking_msgReceiveEvt
Clock enableBraking_msgConsumeEvt
```

$$\frac{\text{enableBraking.msgSize * (1 + CAN.minMsgCtrlOvhd)}}{\text{CAN.maxDecodeRate}} = \frac{\text{500+200 bit}}{\text{150 kbit/s}}$$

```
Integer enableBraking_minConsumeTime = 5
```

$$\frac{\text{enableBraking.msgSize * (1 + CAN.maxMsgCtrlOvhd)}}{\text{CAN.minDecodeRate}} = \frac{\text{500+300 bit}}{\text{100 kbit/s}}$$

```
Integer enableBraking_maxConsumeTime = 8

Expression enableBraking_msgReceptionDelayedByMinConsumeTime
  = DelayFor (  clockForCounting->globalTime,
                clockToDelay->enableBraking_msgReceiveEvt,
                delay->enableBraking_minConsumeTime )
Expression enableBraking_msgReceptionDelayedByMaxConsumeTime
  = DelayFor (  clockForCounting->globalTime,
                clockToDelay->enableBraking_msgReceiveEvt,
                delay->enableBraking_maxConsumeTime )

Relation enableBraking_minConsumptionDelay[NonStrictPrecedes] (
  leftClock->enableBraking_msgReceptionDelayedByMinConsumeTime,
  rightClock->enableBraking_msgConsumeEvt)
Relation enableBraking_maxConsumptionDelay [NonStrictPrecedes] (
  leftClock->enableBraking_msgConsumeEvt,
  rightClock->enableBraking_msgReception-
                      DelayedByMaxConsumeTime)
```

Figure B.13: Specification excerpt of semantics for distributed message consumption delays including example models

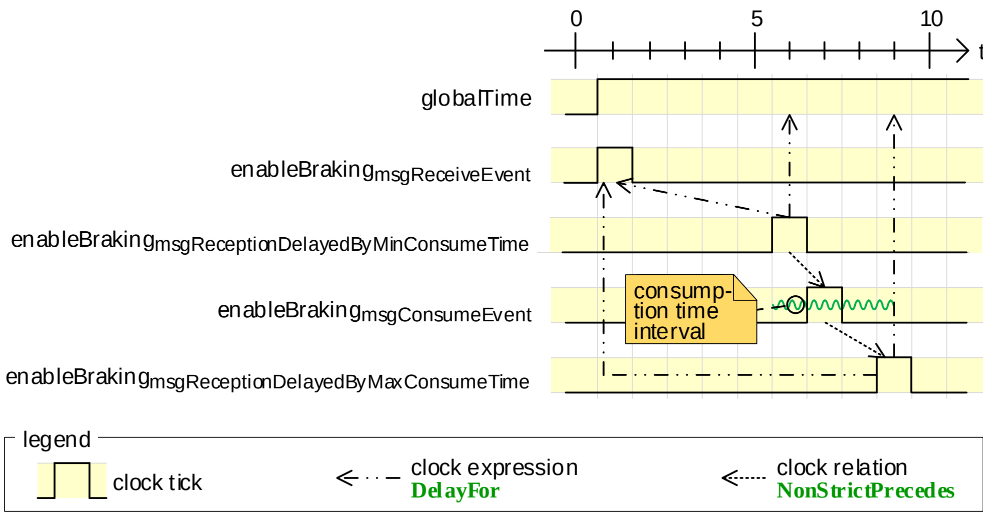
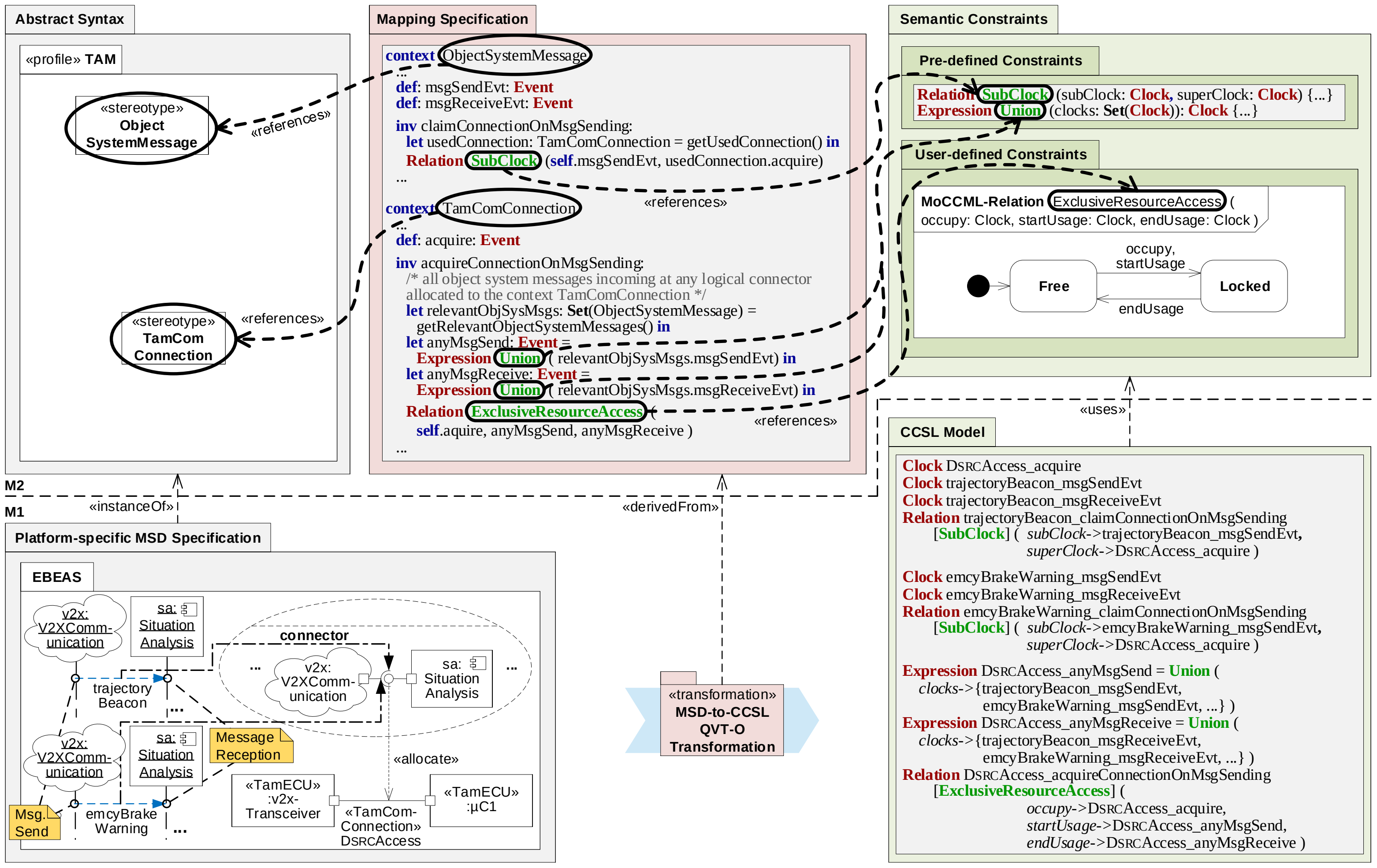

0
5
10
t
globalTime
enableBraking$_{msgReceiveEvent}$
enableBraking$_{msgReceptionDelayedByMinConsumeTime}$
enableBraking$_{msgConsumeEvent}$
enableBraking$_{msgReceptionDelayedByMaxConsumeTime}$
consump-tion time interval
legend
clock tick
clock expression
DelayFor
clock relation
NonStrictPrecedes

Figure B.14: Exemplary CCSL run simulating distributed message consumption delays (cf. CCSL model in the lower right of Figure B.13)



```
context ObjectSystemMessage
  ...
  def: msgSendEvt: Event
  def: msgReceiveEvt: Event

  inv claimConnectionOnMsgSending:
    let usedConnection: TamComConnection = getUsedConnection() in
    Relation SubClock (self.msgSendEvt, usedConnection.acquire)
  ...

context TamComConnection
  ...
  def: acquire: Event

  inv acquireConnectionOnMsgSending:
    /* all object system messages incoming at any logical connector
    allocated to the context TamComConnection */
    let relevantObjSysMsgs: Set(ObjectSystemMessage) =
      getRelevantObjectSystemMessages() in
    let anyMsgSend: Event =
      Expression Union ( relevantObjSysMsgs.msgSendEvt) in
    let anyMsgReceive: Event =
      Expression Union ( relevantObjSysMsgs.msgReceiveEvt) in
    Relation ExclusiveResourceAccess (
      self.aquire, anyMsgSend, anyMsgReceive )
  ...
```

```
Clock DSRCAccess_acquire
Clock trajectoryBeacon_msgSendEvt
Clock trajectoryBeacon_msgReceiveEvt
Relation trajectoryBeacon_claimConnectionOnMsgSending
      [SubClock] (  subClock->trajectoryBeacon_msgSendEvt,
                    superClock->DSRCAccess_acquire )

Clock emcyBrakeWarning_msgSendEvt
Clock emcyBrakeWarning_msgReceiveEvt
Relation emcyBrakeWarning_claimConnectionOnMsgSending
      [SubClock] (  subClock->emcyBrakeWarning_msgSendEvt,
                    superClock->DSRCAccess_acquire )

Expression DSRCAccess_anyMsgSend = Union (
   clocks->{trajectoryBeacon_msgSendEvt,
            emcyBrakeWarning_msgSendEvt, ...} )
Expression DSRCAccess_anyMsgReceive = Union (
   clocks->{trajectoryBeacon_msgReceiveEvt,
            emcyBrakeWarning_msgReceiveEvt, ...} )
Relation DSRCAccess_acquireConnectionOnMsgSending
      [ExclusiveResourceAccess] (
                    occupy->DSRCAccess_acquire,
                    startUsage->DSRCAccess_anyMsgSend,
                    endUsage->DSRCAccess_anyMsgReceive )
```

Figure B.15: Specification excerpt of semantics for access to communication channels including example models

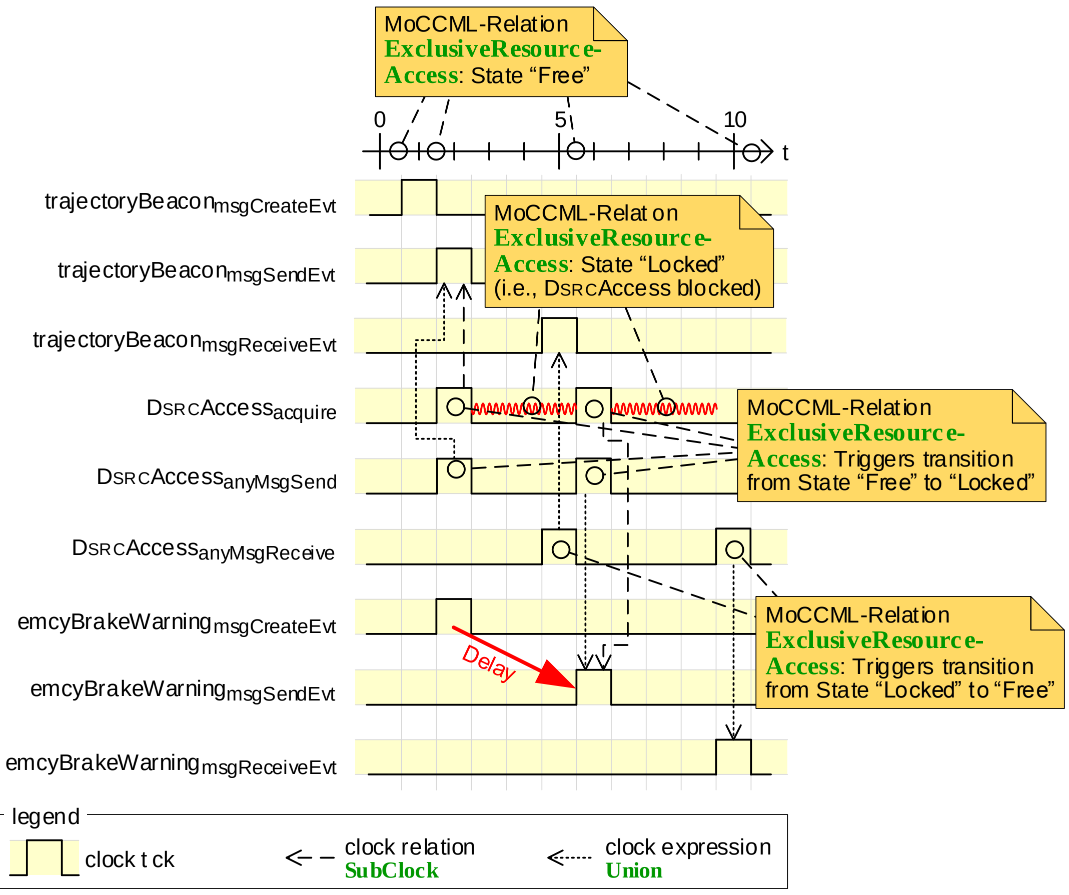
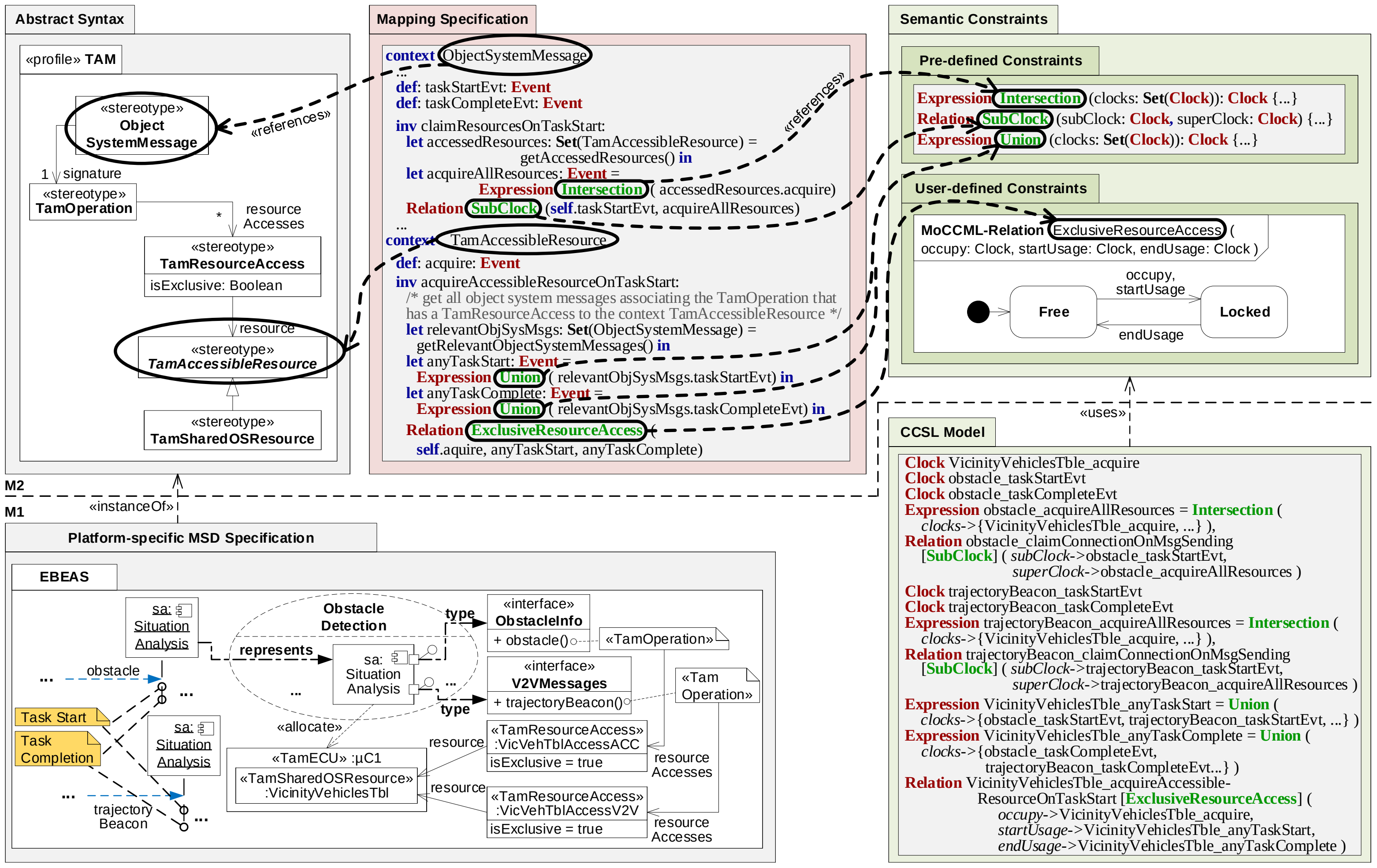

MoCCML-Relation
ExclusiveResource-Access: State "Free"
0
5
10
t
trajectoryBeacon$_{msgCreateEvt}$
trajectoryBeacon$_{msgSendEvt}$
trajectoryBeacon$_{msgReceiveEvt}$
DSRCAccess$_{acquire}$
DSRCAccess$_{anyMsgSend}$
DSRCAccess$_{anyMsgReceive}$
emcyBrakeWarning$_{msgCreateEvt}$
emcyBrakeWarning$_{msgSendEvt}$
emcyBrakeWarning$_{msgReceiveEvt}$
MoCCML-Relation
ExclusiveResource-Access: State "Locked" (i.e., DSRCAccess blocked)
MoCCML-Relation
ExclusiveResource-Access: Triggers transition from State "Free" to "Locked"
MoCCML-Relation
ExclusiveResource-Access: Triggers transition from State "Locked" to "Free"
Delay
legend
clock tick
clock relation SubClock
clock expression Union

Figure B.16: Exemplary CCSL run simulating exclusive communication channel access



```
context ObjectSystemMessage
 ...
 def: taskStartEvt: Event
 def: taskCompleteEvt: Event
 inv claimResourcesOnTaskStart:
   let accessedResources: Set(TamAccessibleResource) =
                  getAccessedResources() in
   let acquireAllResources: Event =
               Expression Intersection( accessedResources.acquire)
   Relation SubClock (self.taskStartEvt, acquireAllResources)
 ...
context TamAccessibleResource
 def: acquire: Event
 inv acquireAccessibleResourceOnTaskStart:
   /* get all object system messages associating the TamOperation that
   has a TamResourceAccess to the context TamAccessibleResource */
   let relevantObjSysMsgs: Set(ObjectSystemMessage) =
    getRelevantObjectSystemMessages() in
   let anyTaskStart: Event =
    Expression Union( relevantObjSysMsgs.taskStartEvt) in
   let anyTaskComplete: Event =
    Expression Union( relevantObjSysMsgs.taskCompleteEvt) in
   Relation ExclusiveResourceAccess(
    self.aquire, anyTaskStart, anyTaskComplete)
```

```
Expression Intersection (clocks: Set(Clock)): Clock {...}
Relation SubClock (subClock: Clock, superClock: Clock) {...}
Expression Union (clocks: Set(Clock)): Clock {...}
```

```
MoCCML-Relation ExclusiveResourceAccess (
occupy: Clock, startUsage: Clock, endUsage: Clock )
```

```
Clock VicinityVehiclesTble_acquire
Clock obstacle_taskStartEvt
Clock obstacle_taskCompleteEvt
Expression obstacle_acquireAllResources = Intersection (
    clocks->{VicinityVehiclesTble_acquire, ...} ),
Relation obstacle_claimConnectionOnMsgSending
    [SubClock] ( subClock->obstacle_taskStartEvt,
                 superClock->obstacle_acquireAllResources )
Clock trajectoryBeacon_taskStartEvt
Clock trajectoryBeacon_taskCompleteEvt
Expression trajectoryBeacon_acquireAllResources = Intersection (
    clocks->{VicinityVehiclesTble_acquire, ...} ),
Relation trajectoryBeacon_claimConnectionOnMsgSending
    [SubClock] ( subClock->trajectoryBeacon_taskStartEvt,
                 superClock->trajectoryBeacon_acquireAllResources )
Expression VicinityVehiclesTble_anyTaskStart = Union (
    clocks->{obstacle_taskStartEvt, trajectoryBeacon_taskStartEvt, ...} )
Expression VicinityVehiclesTble_anyTaskComplete = Union (
    clocks->{obstacle_taskCompleteEvt,
             trajectoryBeacon_taskCompleteEvt...} )
Relation VicinityVehiclesTble_acquireAccessible-
             ResourceOnTaskStart [ExclusiveResourceAccess] (
                 occupy->VicinityVehiclesTble_acquire,
                 startUsage->VicinityVehiclesTble_anyTaskStart,
                 endUsage->VicinityVehiclesTble_anyTaskComplete )
```

Figure B.17: Specification excerpt of semantics for access to shared OS resources including example models

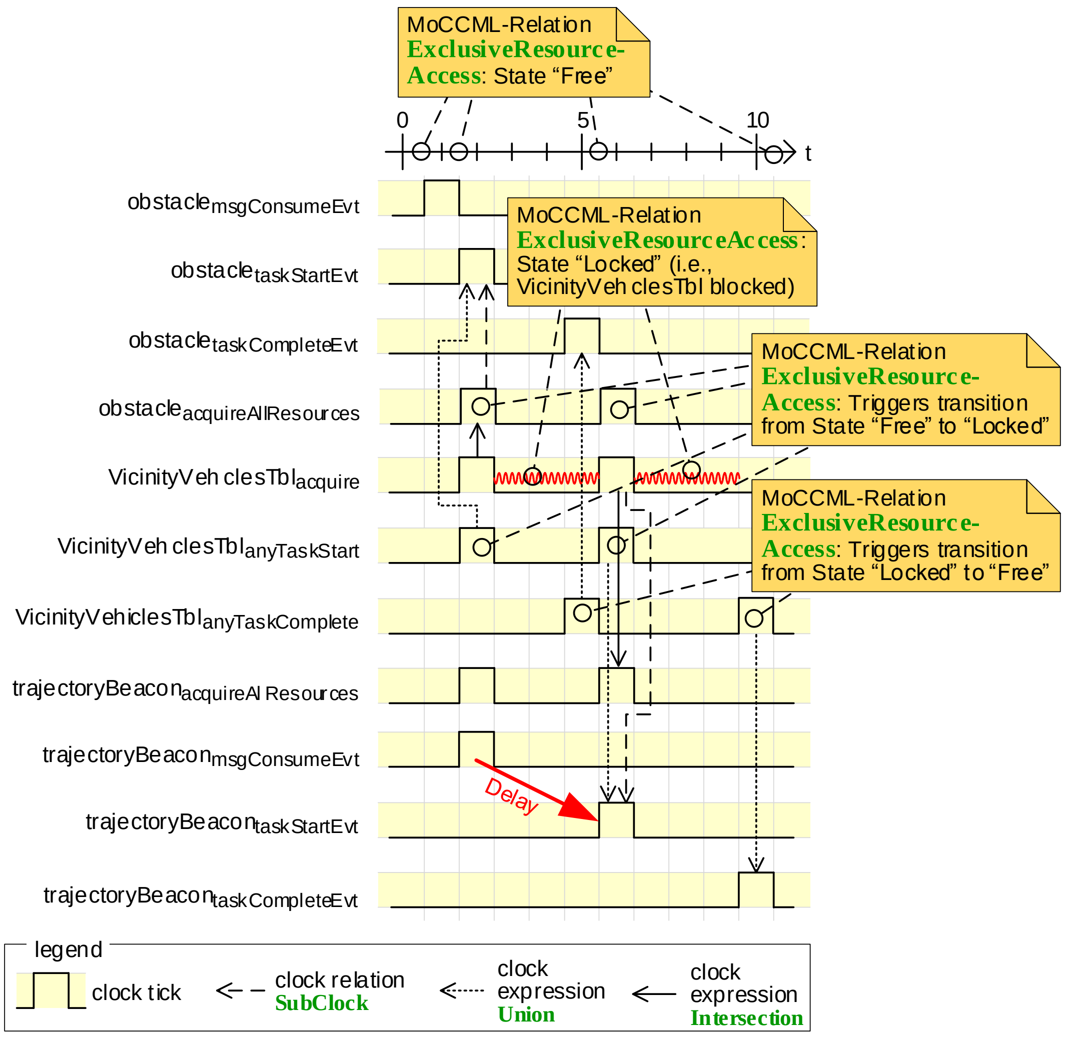
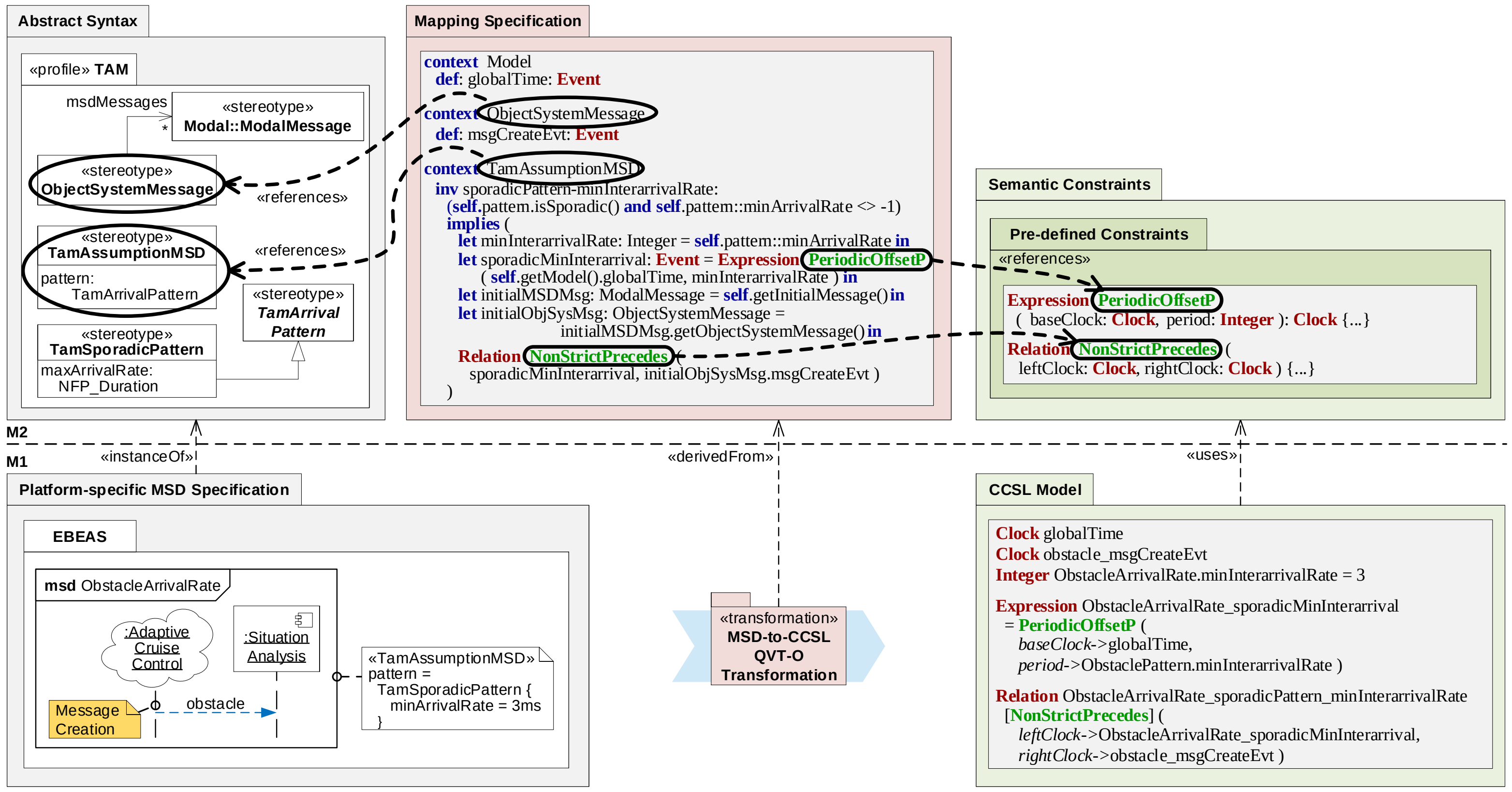

MoCCML-Relation ExclusiveResource-Access: State "Free"
0
5
10
t
obstacle msgConsumeEvt
obstacle taskStartEvt
obstacle taskCompleteEvt
obstacle acquireAllResources
VicinityVehiclesTbl acquire
VicinityVehiclesTbl anyTaskStart
VicinityVehiclesTbl anyTaskComplete
trajectoryBeacon acquireAllResources
trajectoryBeacon msgConsumeEvt
trajectoryBeacon taskStartEvt
trajectoryBeacon taskCompleteEvt
MoCCML-Relation ExclusiveResourceAccess: State "Locked" (i.e., VicinityVehiclesTbl blocked)
MoCCML-Relation ExclusiveResource-Access: Triggers transition from State "Free" to "Locked"
MoCCML-Relation ExclusiveResource-Access: Triggers transition from State "Locked" to "Free"
Delay
legend
clock tick
clock relation SubClock
clock expression Union
clock expression Intersection

Figure B.18: Exemplary CCSL run simulating exclusive shared OS resource access



```
context  Model
  def: globalTime: Event

context ObjectSystemMessage
  def: msgCreateEvt: Event

context TamAssumptionMSD
  inv sporadicPattern-minInterarrivalRate:
    (self.pattern.isSporadic() and self.pattern::minArrivalRate <> -1)
    implies (
      let minInterarrivalRate: Integer = self.pattern::minArrivalRate in
      let sporadicMinInterarrival: Event = Expression PeriodicOffsetP
          ( self.getModel().globalTime, minInterarrivalRate ) in
      let initialMSDMsg: ModalMessage = self.getInitialMessage() in
      let initialObjSysMsg: ObjectSystemMessage =
                       initialMSDMsg.getObjectSystemMessage() in
      Relation NonStrictPrecedes (
        sporadicMinInterarrival, initialObjSysMsg.msgCreateEvt )
    )
```

```
Expression PeriodicOffsetP
 (  baseClock: Clock,  period: Integer ): Clock {...}
Relation NonStrictPrecedes (
  leftClock: Clock, rightClock: Clock ) {...}
```

```
Clock globalTime
Clock obstacle_msgCreateEvt
Integer ObstacleArrivalRate.minInterarrivalRate = 3

Expression ObstacleArrivalRate_sporadicMinInterarrival
  = PeriodicOffsetP (
    baseClock->globalTime,
    period->ObstaclePattern.minInterarrivalRate )

Relation ObstacleArrivalRate_sporadicPattern_minInterarrivalRate
  [NonStrictPrecedes] (
    leftClock->ObstacleArrivalRate_sporadicMinInterarrival,
    rightClock->obstacle_msgCreateEvt )
```

Figure B.19: Specification excerpt of semantics for sporadic arrival patterns with minimum arrival rate including example models

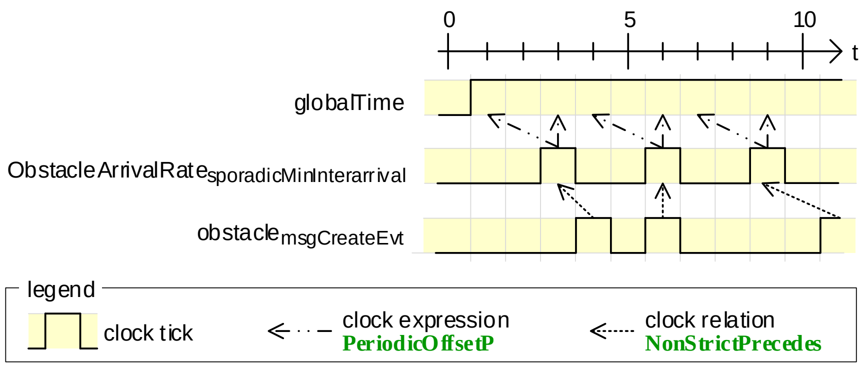
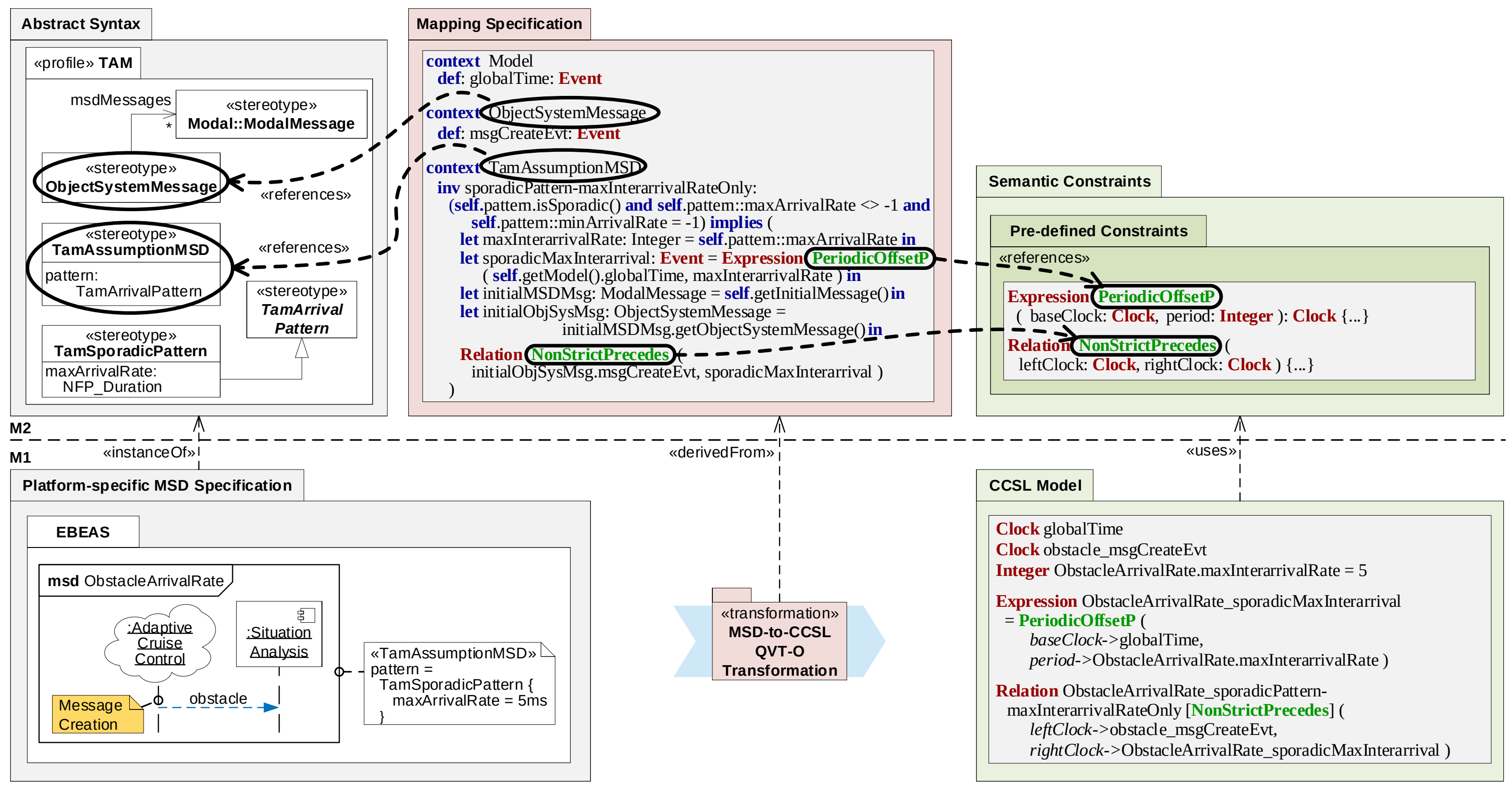


Figure B.20: Example—sporadic arrival pattern with minimum arrival rate in CCSL


Abstract Syntax
«profile» TAM
msdMessages
*
«stereotype» Modal::ModalMessage
«stereotype» ObjectSystemMessage
«references»
«stereotype» TamAssumptionMSD
pattern: TamArrivalPattern
«references»
«stereotype» TamArrival Pattern
«stereotype» TamSporadicPattern
maxArrivalRate: NFP_Duration
Mapping Specification
Semantic Constraints
Pre-defined Constraints
«references»
M2
M1
«instanceOf»
«derivedFrom»
«uses»
Platform-specific MSD Specification
EBEAS
msd ObstacleArrivalRate
:Adaptive Cruise Control
:Situation Analysis
Message Creation
obstacle
«TamAssumptionMSD» pattern = TamSporadicPattern { maxArrivalRate = 5ms }
«transformation» MSD-to-CCSL QVT-O Transformation
CCSL Model

```
context Model
  def: globalTime: Event

context ObjectSystemMessage
  def: msgCreateEvt: Event

context TamAssumptionMSD
  inv sporadicPattern-maxInterarrivalRateOnly:
    (self.pattern.isSporadic() and self.pattern::maxArrivalRate <> -1 and
       self.pattern::minArrivalRate = -1) implies (
      let maxInterarrivalRate: Integer = self.pattern::maxArrivalRate in
      let sporadicMaxInterarrival: Event = Expression PeriodicOffsetP
          ( self.getModel().globalTime, maxInterarrivalRate ) in
      let initialMSDMsg: ModalMessage = self.getInitialMessage() in
      let initialObjSysMsg: ObjectSystemMessage =
                   initialMSDMsg.getObjectSystemMessage() in
      Relation NonStrictPrecedes (
        initialObjSysMsg.msgCreateEvt, sporadicMaxInterarrival )
    )
```

```
Expression PeriodicOffsetP
  ( baseClock: Clock, period: Integer ): Clock {...}
Relation NonStrictPrecedes (
  leftClock: Clock, rightClock: Clock ) {...}
```

```
Clock globalTime
Clock obstacle_msgCreateEvt
Integer ObstacleArrivalRate.maxInterarrivalRate = 5

Expression ObstacleArrivalRate_sporadicMaxInterarrival
  = PeriodicOffsetP (
      baseClock->globalTime,
      period->ObstacleArrivalRate.maxInterarrivalRate )

Relation ObstacleArrivalRate_sporadicPattern-
  maxInterarrivalRateOnly [NonStrictPrecedes] (
      leftClock->obstacle_msgCreateEvt,
      rightClock->ObstacleArrivalRate_sporadicMaxInterarrival )
```

Figure B.21: Specification excerpt of semantics for sporadic arrival patterns with maximum arrival rate including example models

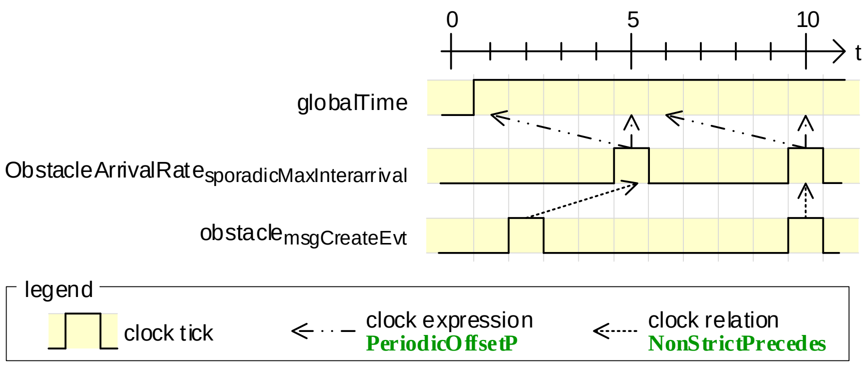
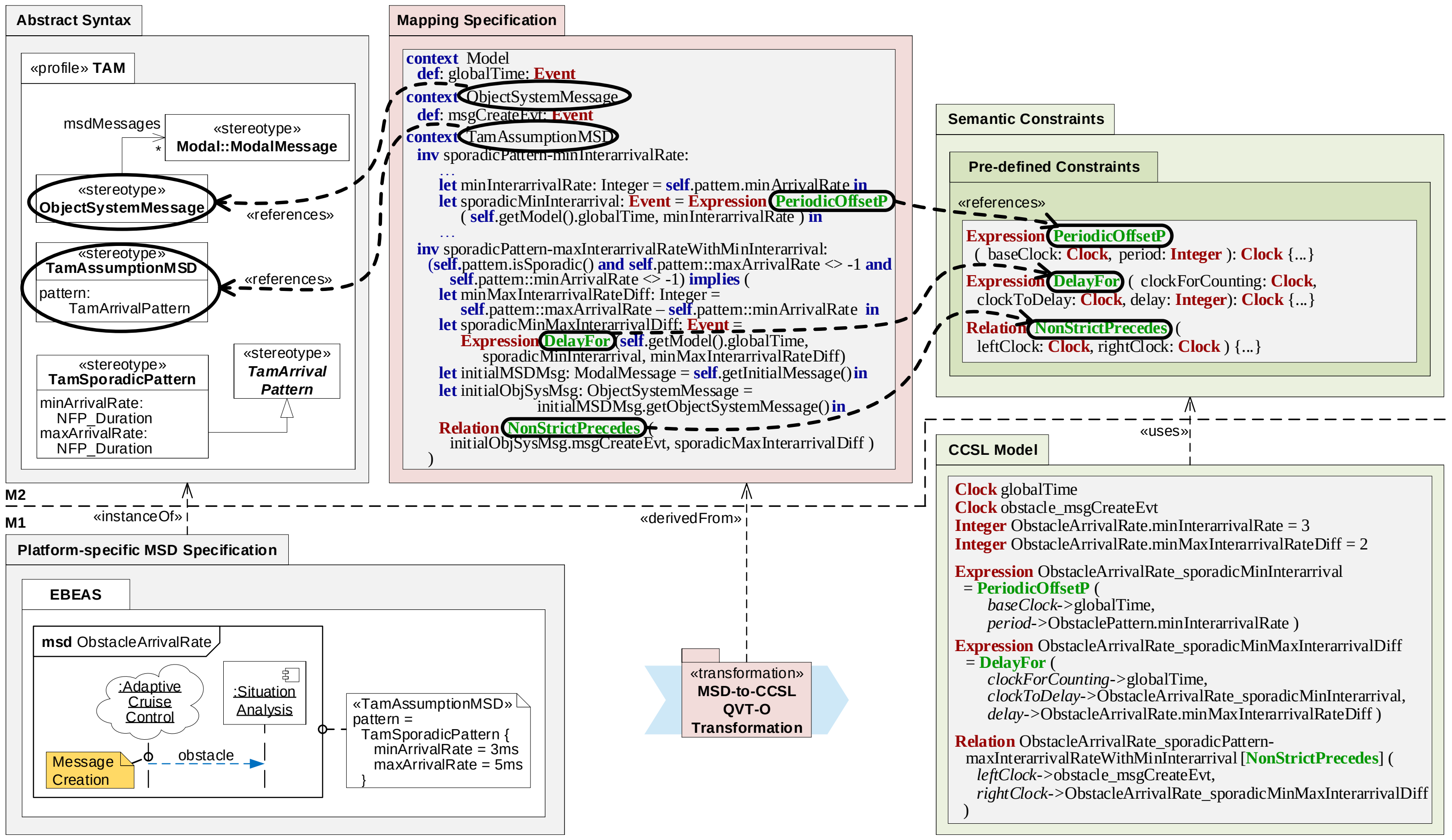


Figure B.22: Example—sporadic arrival pattern with maximum arrival rate in CCSL


Abstract Syntax
«profile» TAM
msdMessages
«stereotype» Modal::ModalMessage
*
«stereotype» ObjectSystemMessage
«references»
«stereotype» TamAssumptionMSD
pattern: TamArrivalPattern
«references»
«stereotype» TamSporadicPattern
minArrivalRate: NFP_Duration
maxArrivalRate: NFP_Duration
«stereotype» TamArrival Pattern
Mapping Specification
Semantic Constraints
Pre-defined Constraints
«references»
«uses»
CCSL Model
M2
M1
«instanceOf»
«derivedFrom»
Platform-specific MSD Specification
EBEAS
msd ObstacleArrivalRate
:Adaptive Cruise Control
:Situation Analysis
Message Creation
obstacle
«transformation» MSD-to-CCSL QVT-O Transformation

```
context  Model
  def: globalTime: Event
context ObjectSystemMessage
  def: msgCreateEvt: Event
context TamAssumptionMSD
  inv sporadicPattern-minInterarrivalRate:
      ...
      let minInterarrivalRate: Integer = self.pattern.minArrivalRate in
      let sporadicMinInterarrival: Event = Expression PeriodicOffsetP
          ( self.getModel().globalTime, minInterarrivalRate ) in
      ...
  inv sporadicPattern-maxInterarrivalRateWithMinInterarrival:
    (self.pattern.isSporadic() and self.pattern::maxArrivalRate <> -1 and
       self.pattern::minArrivalRate <> -1) implies (
      let minMaxInterarrivalRateDiff: Integer =
          self.pattern::maxArrivalRate – self.pattern::minArrivalRate  in
      let sporadicMinMaxInterarrivalDiff: Event =
          Expression DelayFor (self.getModel().globalTime,
              sporadicMinInterarrival, minMaxInterarrivalRateDiff)
      let initialMSDMsg: ModalMessage = self.getInitialMessage() in
      let initialObjSysMsg: ObjectSystemMessage =
                      initialMSDMsg.getObjectSystemMessage() in
      Relation NonStrictPrecedes (
        initialObjSysMsg.msgCreateEvt, sporadicMaxInterarrivalDiff )
    )
```

```
Expression PeriodicOffsetP
  ( baseClock: Clock, period: Integer ): Clock {...}
Expression DelayFor ( clockForCounting: Clock,
  clockToDelay: Clock, delay: Integer): Clock {...}
Relation NonStrictPrecedes (
  leftClock: Clock, rightClock: Clock ) {...}
```

```
«TamAssumptionMSD»
pattern =
 TamSporadicPattern {
   minArrivalRate = 3ms
   maxArrivalRate = 5ms
 }
```

```
Clock globalTime
Clock obstacle_msgCreateEvt
Integer ObstacleArrivalRate.minInterarrivalRate = 3
Integer ObstacleArrivalRate.minMaxInterarrivalRateDiff = 2

Expression ObstacleArrivalRate_sporadicMinInterarrival
  = PeriodicOffsetP (
      baseClock->globalTime,
      period->ObstaclePattern.minInterarrivalRate )

Expression ObstacleArrivalRate_sporadicMinMaxInterarrivalDiff
  = DelayFor (
      clockForCounting->globalTime,
      clockToDelay->ObstacleArrivalRate_sporadicMinInterarrival,
      delay->ObstacleArrivalRate.minMaxInterarrivalRateDiff )

Relation ObstacleArrivalRate_sporadicPattern-
  maxInterarrivalRateWithMinInterarrival [NonStrictPrecedes] (
    leftClock->obstacle_msgCreateEvt,
    rightClock->ObstacleArrivalRate_sporadicMinMaxInterarrivalDiff
  )
```

Figure B.23: Specification excerpt of semantics for sporadic arrival patterns with both minimum and maximum arrival rate including example models

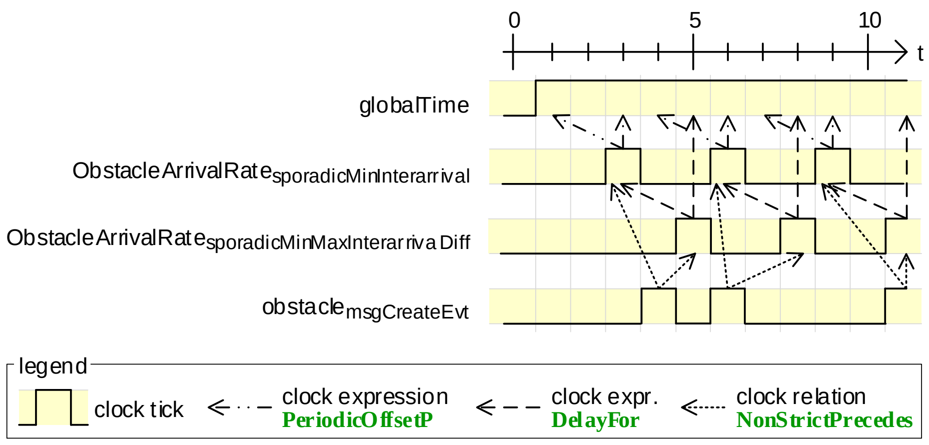

0
5
10
t
globalTime
ObstacleArrivalRate$_{\text{sporadicMinInterarrival}}$
ObstacleArrivalRate$_{\text{sporadicMinMaxInterarrivalDiff}}$
obstacle$_{\text{msgCreateEvt}}$
legend
clock tick
clock expression
PeriodicOffsetP
clock expr.
DelayFor
clock relation
NonStrictPrecedes

Figure B.24: Example—sporadic arrival pattern with both minimum and maximum arrival rate in CCSL

B.2 Complete MSD Semantics for Timing Analyses: ECL Mapping Specification and User-defined MoCCML Relations

Listing B.1: ECL pseudocode for the overall MSD specification

```
context MSDSpecification
 /* The reference clock. Ticks always since no constraints are applied to it. */
 def: globalTime:Event
```

Listing B.2: ECL pseudocode for object system messages

```
context ObjectSystemMessage
 def: msgCreateEvt:Event
 def: msgSendEvt:Event
 def: msgReceiveEvt:Event
 def: msgConsumeEvt:Event
 def: taskStartEvt:Event
 def: taskCompleteEvt:Event
 def: anyEvt:Event
 def: taskSyncFlag:Event

 /* Enforces msgCreateEvt to tick only on ticks of the globalTime DSE clock of
     the MSD specification (cf. Listing B.1) */
 inv createEvtOnlyWithGlobalClock:
  Relation SubClock(
    subClock->self.msgCreateEvt,
    superClock->self.getModel().globalTime
  )

 /* Enforces msgSendEvt to tick only on ticks of the globalTime DSE clock of the
     MSD specification (cf. Listing B.1) */
 inv sendEvtOnlyWithGlobalClock:
  Relation SubClock(
    subClock->self.msgSendEvt,
    superClock->self.getModel().globalTime
  )

 /* Enforces msgReceiveEvt to tick only on ticks of the globalTime DSE clock of
     the MSD specification (cf. Listing B.1) */
 inv receiveEvtOnlyWithGlobalClock:
  Relation SubClock(
    subClock->self.msgReceiveEvt,
    superClock->self.getModel().globalTime
  )

 /* Enforces msgConsumeEvt to tick only on ticks of the globalTime DSE clock of
     the MSD specification (cf. Listing B.1) */
 inv consumeEvtOnlyWithGlobalClock:
  Relation SubClock(
    subClock->self.msgConsumeEvt,
    superClock->self.getModel().globalTime
  )

 /* Enforces taskStartEvt to tick only on ticks of the globalTime DSE clock of
     the MSD specification (cf. Listing B.1) */
 inv taskStartEvtOnlyWithGlobalClock:
  Relation SubClock(
    subClock->self.taskStartEvt,
    superClock->self.getModel().globalTime
  )

```

```
/* Enforces taskCompleteEvt to tick only on ticks of the globalTime DSE clock of
    the MSD specification (cf. Listing B.1) */
inv taskCompleteEvtOnlyWithGlobalClock:
 Relation SubClock(
   subClock→self.taskCompleteEvt,
   superClock→self.getModel().globalTime
 )

inv anyEventTriggering:
 /* define a new clock that ticks on the tick of any of the message event DSE
    clocks */
 let allEvents:Event = Expression Union(
   clocks→{self.msgCreateEvt, self.msgSendEvt, self.msgReceiveEvt, self.
      msgConsumeEvt, self.taskStartEvt, self.taskCompleteEvt}
 ) in
 /* let the anyEvt DSE clock tick together with the allEvents clock */
 Relation Coincides(
   clock1→self.anyEvt,
   clock2→allEvents
 )

inv claimCoreOnTaskStart:
 /* get the TamScheduler of the TamRTOS of the TamECU that the software
    component is allocated to */
 let relevantScheduler: TamScheduler = getRelevantScheduler() in
 /* let the taskStartEvt DSE clock only tick if the dispatch DSE clock of the
    corresponding TamScheduler can tick (cf. Listing B.4) */
 Relation SubClock(
   subClock→self.taskStartEvt,
   superClock→relevantScheduler.dispatch
 )

inv nonPreemptiveFixedPriorityPolicy_noInterferenceWithHigherPrioTasks:
 /* get all object system messages corresponding to TamOperations of all
    software components deployed to the same ECU, where these TamOperations
    have a higher priority than the TamOperation of the context object system
    message */
 let higherPrioTaskObjectSystemMessages:Set(ObjectSystemMessage) =
    getRelevantHigherPrioTaskObjectSystemMessages() in
 /* define a new clock that ticks only when there is no higher prio task that is
     ready to execute. That is, if any higher prio task is in the state “Ready
    to Execute” of the MoCCML relation “NonPreemptiveFixedPriorityTaskReadiness
    ” (cf. Figure B.25 and invariant
    nonPreemptiveFixedPriorityPolicy_syncWithLowerPrioTasks), this clock does
    not tick. */
 let noHigherPrioTaskReady:Event = Expression Intersection(
   clocks→higherPrioTaskObjectSystemMessages.taskSyncFlag) in
 /* allow the taskStartEvt of the context object system message to tick only on
    ticks of noHigherPrioTaskReady (i.e., the task corresponding to the
    context system messages is allowed only to start when there is no higher
    priority task ready to execute). The TamScheduler (cf. Listing B.4) further
    decides about whether taskStartEvt is allowed to tick (i.e., it determines
     whether the corresponding processing unit is free and hence the task can
    be dispatched). */
 Relation SubClock(
   subClock→self.taskStartEvt,
   superClock→noHigherPrioTaskReady
 )

inv nonPreemptiveFixedPriorityPolicy_syncWithLowerPrioTasks:
 /* allows the taskSyncFlag of the context object system message to tick only in
     the state “Idle or Running” (cf. MoCCML relation in Figure B.25) */
 Relation NonPreemptiveFixedPriorityTaskReadiness (
   lowerPrioTaskAllowedToBeDispatched→self.taskSyncFlag,
   msgConsume→self.msgConsumeEvt,
   taskStart→self.taskStartEvt
```

```
  )

inv claimConnectionOnMsgSending:
  let allocatedConnection:TamComConnection = getAllocatedConnection() in
  /* let the msgSendEvt DSE clock only tick if the acquire DSE clock of the
      corresponding TamComConnection can tick (cf. Listing B.5) */
  Relation SubClock(
    subClock->self.msgSendEvt,
    superClock->allocatedConnection.acquire
  )

inv claimResourcesOnTaskStart:
  /* get all TamAccessibleResources that the associated TamOperation has a
      TamResourceAccess to */
  let accessedResources:Set(TamComConnection) = getAccessedResources() in
  /* let the taskStartEvt DSE clock only tick if all acquire DSE clocks of the
      corresponding TamAccessibleResources can tick (cf. Listing B.6) */
  let acquireAllResources:Event = Expression Intersection(clocks->accessedResources
      .acquire) in
  Relation SubClock(
    subClock->self.taskStartEvt,
    superClock->acquireAllResources
  )

inv minDispatchDelay:
  /* cf. Equation (4.5) on Page 169 and Equation (4.6) on Page 170 */
  let minDispatchTime:Integer = self.getMinDispatchTime() in
  let msgCreationDelayedByMinDispatchTime:Event = Expression DelayFor(
    clockForCounting->self.getModel().globalTime,
    clockToDelay->self.msgCreateEvt,
    delay->minDispatchTime)) in
  Relation NonStrictPrecedes(
    leftClock->msgCreationDelayedByMinDispatchTime,
    rightClock->self.msgSendEvt
  )

inv maxDispatchDelay:
  /* cf. Equation (4.1) on Page 137 */
  let maxDispatchTime:Integer = self.getMaxDispatchTime() in
  let msgCreationDelayedByMaxDispatchTime:Event = Expression DelayFor(
    clockForCounting->self.getModel().globalTime,
    clockToDelay->self.msgCreateEvt,
    delay->maxDispatchTime)) in
  Relation NonStrictPrecedes(
    leftClock->self.msgSendEvt,
    rightClock->msgCreationDelayedByMaxDispatchTime
  )

inv minSendDelay:
  /* cf. Equation (4.8) on Page 171 and Equation (4.9) on Page 171 */
  let minTransmissionTime:Integer = self.getMinTransmissionTime() in
  let msgSendingDelayedByMinTransmissionTime:Event = Expression DelayFor(
    clockForCounting->self.getModel().globalTime,
    clockToDelay->self.msgSendEvt,
    delay->minTransmissionTime)) in
  Relation NonStrictPrecedes(
    leftClock->msgSendingDelayedByMinTransmissionTime,
    rightClock->self.msgReceiveEvt
  )

inv maxSendDelay:
  /* cf. Equation (4.2) on Page 138 */
  let maxTransmissionTime: Integer = self.getMaxTransmissionTime() in
  let msgSendingDelayedByMaxTransmissionTime: Event = Expression DelayFor(
    clockForCounting->self.getModel().globalTime,
    clockToDelay->self.msgSendEvt,
      delay->maxTransmissionTime)) in
    Relation NonStrictPrecedes(
      leftClock->self.msgReceiveEvt,
      rightClock->msgSendingDelayedByMaxTransmissionTime
    )

  inv minConsumptionDelay:
    /* cf. Equation (4.11) on Page 172 and Equation (4.12) on Page 172 */
    let minConsumptionTime:Integer = self.getMinConsumptionTime() in
    let msgReceptionDelayedByMinConsumptionTime:Event = Expression DelayFor(
      clockForCounting->self.getModel().globalTime,
      clockToDelay->self.msgReceiveEvt,
      delay->minConsumptionTime)) in
    Relation NonStrictPrecedes(
      leftClock->msgReceptionDelayedByMinConsumptionTime,
      rightClock->self.msgConsumeEvt
    )

  inv maxConsumptionDelay
    /* cf. Equation (4.3) on Page 138 */
    let maxConsumptionTime:Integer = self.getMaxConsumptionTime() in
    let msgReceptionDelayedByMaxConsumptionTime:Event = Expression DelayFor(
      clockForCounting->self.getModel().globalTime,
      clockToDelay->self.msgReceiveEvt,
      delay->maxConsumptionTime)) in
    Relation NonStrictPrecedes(
      leftClock->self.msgConsumeEvt,
      rightClock->smsgReceptionDelayedByMaxConsumptionTime
    )

  inv minExecutionDelay:
    /* cf. Equation (4.13) on Page 173 */
    let minExecutionTime:Integer = self.getMinExecTime() in
    let taskStartDelayedByMinExecTime:Event = Expression DelayFor(
      clockForCounting->self.getModel().globalTime,
      clockToDelay->self.taskStartEvt,
      delay->minExecutionTime)) in
    Relation NonStrictPrecedes(
      leftClock->taskStartDelayedByMinExecTime,
      rightClock->self.taskCompleteEvt
    )

  inv maxExecutionDelay:
    /* cf. Equation (4.4) on Page 138 */
    let maxExecutionTime:Integer = self.getMaxExecTime() in
    let taskStartDelayedByMaxExecTime:Event = Expression DelayFor(
      clockForCounting->self.getModel().globalTime,
      clockToDelay->self.taskStartEvt,
      delay->maxExecutionTime)) in
    Relation NonStrictPrecedes(
      leftClock->self.taskCompleteEvt,
      rightClock->taskStartDelayedByMaxExecTime
    )

    /* state space reduction: allow object system message clocks only to tick when
        corresponding unification occurrence clock can tick */
    inv msgCreateEvtOnlyOnMsgCreateUnification:
      let msgCreateUnificationOccurrence:Event = Expression Union(
        clocks->self.msdMessages.msgCreateUnification
      ) in
      Relation SubClock(
        subClock->self.msgCreateEvt,
        superClock->msgCreateUnificationOccurrence
      )

    /* state space reduction: allow object system message clocks only to tick when
        corresponding unification occurrence clock can tick */
    inv msgSendEvtOnlyOnMsgSendUnification:
     let msgSendUnificationOccurrence:Event = Expression Union(
       clocks→self.msdMessages.msgSendUnification
     ) in
     Relation SubClock(
       subClock→self.msgSendEvt,
       superClock→msgSendUnificationOccurrence
     )

    /* state space reduction: allow object system message clocks only to tick when
        corresponding unification occurrence clock can tick */
    inv msgReceiveEvtOnlyOnMsgReceiveUnification:
     let msgReceiveUnificationOccurrence:Event = Expression Union(
       clocks→self.msdMessages.msgReceiveUnification
     ) in
     Relation SubClock(
       subClock→self.msgReceiveEvt,
       superClock→msgSendReceiveOccurrence
     )

    /* state space reduction: allow object system message clocks only to tick when
        corresponding unification occurrence clock can tick */
    inv msgConsumeEvtOnlyOnMsgConsumeUnification:
     let msgConsumeUnificationOccurrence:Event = Expression Union(
       clocks→self.msdMessages.msgConsumeUnification
     ) in
     Relation SubClock(
       subClock→self.msgConsumeEvt,
       superClock→msgConsumeOccurrence
     )

    /* state space reduction: allow object system message clocks only to tick when
        corresponding unification occurrence clock can tick */
    inv taskStartEvtOnlyOnTaskStartUnification:
     let taskStartUnificationOccurrence:Event = Expression Union(
       clocks→self.msdMessages.taskStartUnification
     ) in
     Relation SubClock(
       subClock→self.taskStartEvt,
       superClock→taskStartOccurrence
     )

    /* state space reduction: allow object system message clocks only to tick when
        corresponding unification occurrence clock can tick */
    inv taskCompleteEvtOnlyOnTaskCompleteUnification:
     let taskCompleteUnificationOccurrence:Event = Expression Union(
       clocks→self.msdMessages.taskCompleteUnification
     ) in
     Relation SubClock(
       subClock→self.taskCompleteEvt,
       superClock→taskCompleteOccurrence
     )
```

Listing B.3: ECL pseudocode for MSD messages

```
context ModalMessage
 def: msgCreateUnification:Event
 def: msgSendUnification:Event
 def: msgReceiveUnification:Event
 def: msgConsumeUnification:Event
 def: taskStartUnification:Event
 def: taskCompleteUnification:Event

```

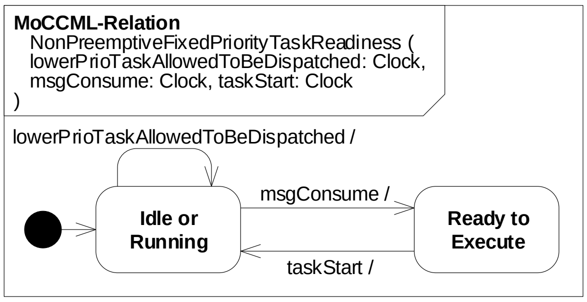
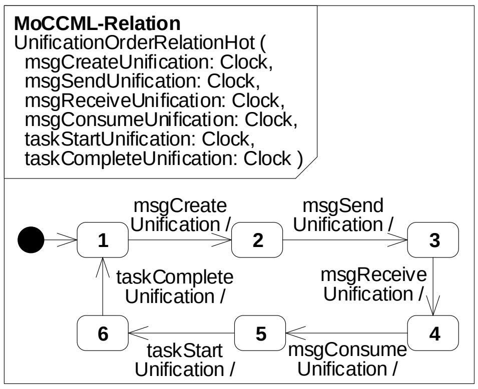

MoCCML-Relation
NonPreemptiveFixedPriorityTaskReadiness (
lowerPrioTaskAllowedToBeDispatched: Clock,
msgConsume: Clock, taskStart: Clock
)
lowerPrioTaskAllowedToBeDispatched /
msgConsume /
Idle or Running
Ready to Execute
taskStart /

Figure B.25: MoCCML relation NonPreemptiveFixedPriorityTaskReadiness

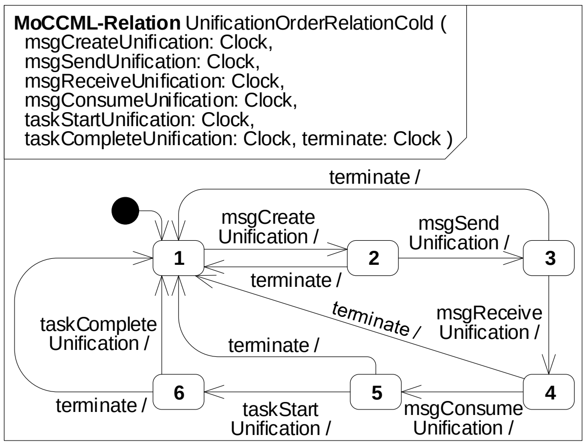
```
inv unificationOrderHot:
  self.isHot() implies
   /* cf. MoCCML relation in Figure B.26(a) */
   Relation UnificationOrderRelationHot(
     msgCreateUnification->self.msgCreateUnification,
     msgSendUnification->self.msgSendUnification,
     msgReceiveUnification->self.msgReceiveUnification,
     msgConsumeUnification->self.msgConsumeUnification,
     taskStartUnification->self.taskStartUnification,
     taskCompleteUnification->self.taskCompleteUnification,
   )
  )

inv unificationOrderCold:
  self.isCold() implies (
   /* all remaining MSD messages in the parent MSD of the context MSD message */
   let notEnabledMessages: Set(Message) = self.getMSD().getNotEnabledMessages()
       in
   /* all object system messages for the not enabled MSD messages */
   let notEnabledObjectSystemMessages: Set(ObjectSystemMessage) =
       notEnabledMessages.getObjectSystemMessages() in
   /* define a new clock ticking together with the anyEvt clock for the not
       enabled object system messages */
   let terminate: Event = Expression Union(
    clocks->notEnabledObjectSystemMessages.anyEvt
   ) in
   /* cf. MoCCML relation in Figure B.26(b) */
   Relation UnificationOrderRelationCold(
     msgCreateUnification->self.msgCreateUnification,
     msgSendUnification->self.msgSendUnification,
     msgReceiveUnification->self.msgReceiveUnification,
     msgConsumeUnification->self.msgConsumeUnification,
     taskStartUnification->self.taskStartUnification,
     taskCompleteUnification->self.taskCompleteUnification,
     terminate->terminate
   )
  )

/* Chains task completion unification occurrence and message creation
    unification occurrence of two consecutive MSD messages in one MSD */
inv messagePrecedence:
  not self.isInitialMessage() implies (
   let precedingMessage:Message = self.getPrecedingMessage() in
   Relation Precedes(
     leftClock->precedingMessage.taskCompleteUnification,
     rightClock->self.msgCreateUnification
   )
  )

inv createUnificationInitialCold:
  (self.isCold() and self.isInitialMessage()) implies (
    let objectSystemMessage = self.getObjectSystemMessage() in
    Relation Coincides(
      clock1->objectSystemMessage.msgCreateEvt,
      clock2->self.msgCreateUnification
    )
  )

inv createUnificationNotinitialHot:
  (self.isHot() and not self.isInitialMessage()) implies (
   let objectSystemMessage = self.getObjectSystemMessage() in
   let precedingMessage:ModalMessage = self.getPrecedingMessage() in
   /* cf. MoCCML relation in Figure B.27(a) */
   Relation UnificationRelationHot(
     msgEvt->objectSystemMessage.msgCreateEvt,
     msgEvtUnification->self.msgCreateUnification,
     enableMsgEvtUnification->precedingMessage.taskCompleteUnification
   )
  )

inv createUnificationNotinitialCold:
  (self.isCold() and not self.isInitialMessage()) implies (
   let objectSystemMessage = self.getObjectSystemMessage() in
   let precedingMessage:ModalMessage = self.getPrecedingMessage() in
   /* all remaining MSD messages in the parent MSD of the context MSD message */
   let notEnabledMessages: Set(Message) = self.getMSD().getNotEnabledMessages()
       in
   /* all object system messages for the not enabled MSD messages */
   let notEnabledObjectSystemMessages: Set(ObjectSystemMessage) =
       notEnabledMessages.getObjectSystemMessages() in
   /* define a new clock ticking together with the anyEvt clock for the not
       enabled object system messages */
   let terminate:Event = Expression Union(
     clocks->notEnabledObjectSystemMessages.anyEvt
   ) in
   /* cf. MoCCML relation in Figure B.27(b) */
   Relation UnificationRelationCold(
     msgEvt->objectSystemMessage.msgCreateEvt,
     msgEvtUnification->self.msgCreateUnification,
     enableMsgEvtUnification->precedingMessage.taskCompleteUnification,
     terminate->terminate
   )
  )

inv sendUnificationHot:
  self.isHot() implies (
   let objectSystemMessage = self.getObjectSystemMessage() in
   /* cf. MoCCML relation in Figure B.27(a) */
   Relation UnificationRelationHot(
     msgEvt->objectSystemMessage.msgSendEvt,
     msgEvtUnification->self.msgSendUnification,
     enableMsgEvtUnification->self.msgCreateUnification
   )
  )

inv sendUnificationCold:
  self.isCold() implies (
   /* all remaining MSD messages in the parent MSD of the context MSD message */
   let notEnabledMessages: Set(Message) = self.getMSD().getNotEnabledMessages()
       in
   /* all object system messages for the not enabled MSD messages */
   let notEnabledObjectSystemMessages: Set(ObjectSystemMessage) =
       notEnabledMessages.getObjectSystemMessages() in
   /* define a new clock ticking together with the anyEvt clock for the not
       enabled object system messages */
   let terminate:Event = Expression Union(
        clocks→notEnabledObjectSystemMessages.anyEvt
      ) in
      /* cf. MoCCML relation in Figure B.27(b) */
      Relation UnificationRelationCold(
        msgEvt→objectSystemMessage.msgSendEvt,
        msgEvtUnification→self.msgSendUnification,
        enableMsgEvtUnification→self.msgCreateUnification,
        terminate→terminate
      )
    )

  inv receiveUnificationHot:
    self.isHot() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      /* cf. MoCCML relation in Figure B.27(a) */
      Relation UnificationRelationHot(
        msgEvt→objectSystemMessage.msgReceiveEvt,
        msgEvtUnification→self.msgReceiveUnification,
        enableMsgEvtUnification→self.msgSendUnification
      )
    )

  inv receiveUnificationCold:
    self.isCold() implies (
      let notEnabledMessages: Set(Message) = self.getMSD().getNotEnabledMessages()
          in
      let notEnabledObjectSystemMessages: Set(ObjectSystemMessage) =
          notEnabledMessages.getObjectSystemMessages() in
      let terminate:Event = Expression Union(
        clocks→notEnabledObjectSystemMessages.anyEvt
      ) in
      /* cf. MoCCML relation in Figure B.27(b) */
      Relation UnificationRelationCold(
        msgEvt→objectSystemMessage.msgReceiveEvt,
        msgEvtUnification→self.msgReceiveUnification,
        enableMsgEvtUnification→self.msgSendUnification,
        terminate→terminate
      )
    )

  inv consumeUnificationHot:
    self.isHot() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      /* cf. MoCCML relation in Figure B.27(a) */
      Relation UnificationRelationHot(
        msgEvt→objectSystemMessage.msgConsumeEvt,
        msgEvtUnification→self.msgConsumeUnification,
        enableMsgEvtUnification→self.msgReceiveUnification
      )
    )

  inv consumeUnificationCold:
    self.isCold() implies (
      let notEnabledMessages: Set(Message) = self.getMSD().getNotEnabledMessages()
          in
      let notEnabledObjectSystemMessages: Set(ObjectSystemMessage) =
          notEnabledMessages.getObjectSystemMessages() in
      let terminate:Event = Expression Union(
        clocks→notEnabledObjectSystemMessages.anyEvt
      ) in
      /* cf. MoCCML relation in Figure B.27(b) */
      Relation UnificationRelationCold(
        msgEvt→objectSystemMessage.msgConsumeEvt,
        msgEvtUnification→self.msgConsumeUnification,
        enableMsgEvtUnification→self.msgReceiveUnification,
        terminate→terminate
      )
    )

  inv taskStartUnificationHot:
    self.isHot() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      /* cf. MoCCML relation in Figure B.27(a) */
      Relation UnificationRelationHot(
        msgEvt→objectSystemMessage.taskStartEvt,
        msgEvtUnification→self.taskStartUnification,
        enableMsgEvtUnification→self.msgConsumeUnification
      )
    )

  inv taskStartUnificationCold:
    self.isCold() implies (
      let notEnabledMessages: Set(Message) = self.getMSD().getNotEnabledMessages()
          in
      let notEnabledObjectSystemMessages: Set(ObjectSystemMessage) =
          notEnabledMessages.getObjectSystemMessages() in
      let terminate:Event = Expression Union(
        clocks→notEnabledObjectSystemMessages.anyEvt
      ) in
      /* cf. MoCCML relation in Figure B.27(b) */
      Relation UnificationRelationCold(
        msgEvt→objectSystemMessage.taskStartEvt,
        msgEvtUnification→self.taskStartUnification,
        enableMsgEvtUnification→self.msgConsumeUnification,
        terminate→terminate
      )
    )

  inv taskCompleteUnificationHot:
    self.isHot() implies (
      let objectSystemMessage = self.getObjectSystemMessage() in
      /* cf. MoCCML relation in Figure B.27(a) */
      Relation UnificationRelationHot(
        msgEvt→objectSystemMessage.taskCompleteEvt,
        msgEvtUnification→self.taskCompleteUnification,
        enableMsgEvtUnification→self.taskStartUnification
      )
    )

  inv taskCompleteUnificationCold:
    self.isCold() implies (
      let notEnabledMessages: Set(Message) = self.getMSD().getNotEnabledMessages()
          in
      let notEnabledObjectSystemMessages:Set(ObjectSystemMessage) =
          notEnabledMessages.getObjectSystemMessages() in
      let terminate:Event = Expression Union(
        clocks→notEnabledObjectSystemMessages.anyEvt
      ) in
      /* cf. MoCCML relation in Figure B.27(b) */
      Relation UnificationRelationCold(
        msgEvt→objectSystemMessage.taskCompleteEvt,
        msgEvtUnification→self.taskCompleteUnification,
        enableMsgEvtUnification→self.taskStartUnification,
        terminate→terminate
      )
    )
```

Listing B.4: ECL pseudocode for TamSchedulers

```
context TamScheduler
  def: dispatch:Event
```

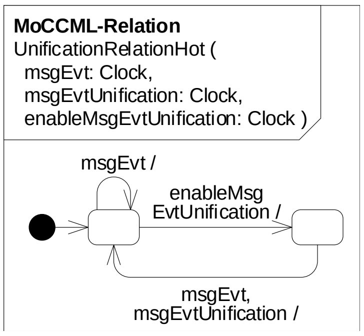

MoCCML-Relation
UnificationOrderRelationHot (
msgCreateUnification: Clock,
msgSendUnification: Clock,
msgReceiveUnification: Clock,
msgConsumeUnification: Clock,
taskStartUnification: Clock,
taskCompleteUnification: Clock)
msgCreate Unification /
msgSend Unification /
msgReceive Unification /
msgConsume Unification /
taskStart Unification /
taskComplete Unification /
1
2
3
4
5
6

(a) MoCCML relation UnificationOrderRelationHot

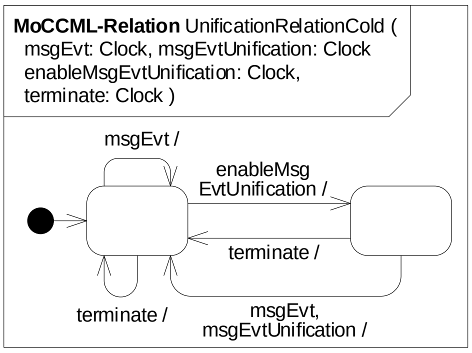

MoCCML-Relation UnificationOrderRelationCold (
msgCreateUnification: Clock,
msgSendUnification: Clock,
msgReceiveUnification: Clock,
msgConsumeUnification: Clock,
taskStartUnification: Clock,
taskCompleteUnification: Clock, terminate: Clock)
terminate /
msgCreate Unification /
msgSend Unification /
terminate /
terminate /
msgReceive Unification /
taskComplete Unification /
terminate /
terminate /
taskStart Unification /
msgConsume Unification /
1
2
3
4
5
6

(b) MoCCML relation UnificationOrderRelationCold

Figure B.26: MoCCML relations for unification occurrence orders


MoCCML-Relation
UnificationRelationHot (
msgEvt: Clock,
msgEvtUnification: Clock,
enableMsgEvtUnification: Clock)
msgEvt /
enableMsg EvtUnification /
msgEvt, msgEvtUnification /

(a) MoCCML relation UnificationRelationHot


MoCCML-Relation UnificationRelationCold (
msgEvt: Clock, msgEvtUnification: Clock
enableMsgEvtUnification: Clock,
terminate: Clock)
msgEvt /
enableMsg EvtUnification /
terminate /
terminate /
msgEvt, msgEvtUnification /

(b) MoCCML relation UnificationRelationCold

Figure B.27: MoCCML relations for unification

```

 inv occupyCoreOnTaskStart:
  let numCores:Integer = getProcessingUnit()::numCores in
  /* all object system messages incoming at any software component allocated to
      the TamECU containing the context TamScheduler */
  let relevantObjectSystemMessages: Set(ObjectSystemMessage) =
      getRelevantObjectSystemMessages() in
  let anyTaskStart:Event = Expression Union(
    clocks→relevantObjectSystemMessages.taskStartEvt
  ) in
  let anyTaskComplete:Event = Expression Union(clocks→relevantObjectSystemMessages
      .taskCompleteEvt) in
  /* cf. MoCCML relation in Figure B.28 */
  Relation NonPreemptiveTaskExecution(
    occupy→self.dispatch,
    newTask→anyTaskStart,
    taskFinish→anyTaskComplete,
    numCores→numCores
  )

 /* complement to last invariant: enforce that only one task starts in the case
     of multiple tasks that are simultaneously ready to start */
 inv onlyOneTaskStart:
  /* all object system messages incoming at any software component allocated to
      the TamECU containing the context TamScheduler */
  let relevantObjectSystemMessages: Set(ObjectSystemMessage) =
      getRelevantObjectSystemMessages() in
  /* enforce that only one taskStartEvt DSE of relevantObjectSystemMessages ticks
       in the case of multiple fireable ones */
  Relation Exclusion(
    clocks→relevantObjectSystemMessages.taskStartEvt
  )
```

MoCCML-Relation NonPreemptiveTaskExecution (
  occupy: Clock, newTask: Clock, taskFinish: Clock,
  numCores: Integer)

local variable runningTasks: Integer = 0

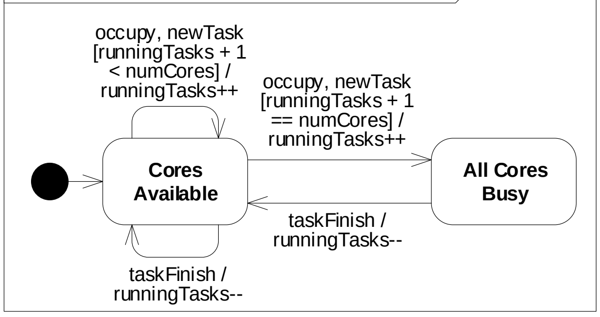

occupy, newTask [runningTasks + 1 < numCores] / runningTasks++
occupy, newTask [runningTasks + 1 == numCores] / runningTasks++
Cores Available
All Cores Busy
taskFinish / runningTasks--
taskFinish / runningTasks--

Figure B.28: MoCCML relation NonPreemptiveTaskExecution

Listing B.5: ECL pseudocode for TamComConnections

```
context TamComConnection
 def: acquire:Event

 inv acquireConnectionOnMsgSending:
  /* get all object system messages incoming at any logical connector allocated
      to the context TamConnection */
  let relevantObjectSystemMessages:Set(ObjectSystemMessage) =
      getRelevantObjectSystemMessages() in
```

```
    let anyMsgSend:Event = Expression Union(
      clocks→relevantObjectSystemMessages.msgSendEvt
    ) in
    let anyMsgReceive:Event = Expression Union(
      clocks→relevantObjectSystemMessages.msgReceiveEvt
    ) in
    /* cf. MoCCML relation in Figure B.29 */
    Relation ExclusiveResourceAccess(
      occupy→self.aquire,
      startUseage→anyMsgSend,
      endUsage→anyMsgReceive
    )

  /* complement to last invariant: enforce that only one message is sent in the
       case of multiple messages that are simultaneously ready to be sent */
  inv onlyOneMessageSend:
    /* get all object system messages incoming at any logical connector allocated
        to the context TamComConnection */
    let relevantObjectSystemMessages: Set(ObjectSystemMessage) =
        getRelevantObjectSystemMessages() in
    /* enforce that only one msgSendEvt DSE of relevantObjectSystemMessages ticks
        in the case of multiple fireable ones */
    Relation Exclusion(
      clocks→relevantObjectSystemMessages.msgSendEvt
    )
```

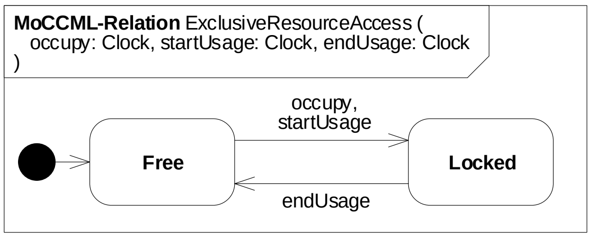

MoCCML-Relation ExclusiveResourceAccess (
occupy: Clock, startUsage: Clock, endUsage: Clock
)
occupy,
startUsage
Free
Locked
endUsage

Figure B.29: MoCCML relation ExclusiveResourceAccess

Listing B.6: ECL pseudocode for TamAccessibleResources

```
context TamAccessibleResource
  def: acquire:Event

  inv acquireAccessibleResourceOnTaskStart:
    /* get all object system messages associating the TamOperation that has a
        TamResourceAccess to the context TamAccessibleResource */
    let relevantObjectSystemMessages:Set(ObjectSystemMessage) =
        getRelevantObjectSystemMessages() in
    let anyTaskStart:Event = Expression Union(
      clocks→relevantObjectSystemMessages.taskStartEvt
    ) in
    let anyTaskComplete:Event = Expression Union(
      clocks→relevantObjectSystemMessages.taskCompleteEvt
    ) in
    /* cf. MoCCML relation in Figure B.29 */
    Relation ExclusiveResourceAccess(
      occupy→self.aquire,
      startUseage→anyTaskStart,
      endUsage→anyTaskComplete
    )

  /* complement to last invariant: enforce that only one task starts in the case
       of multiple tasks that are simultaneously ready to start */
  inv onlyOneTaskStart:
   /* all object system messages incoming at any software component allocated to
       the TamECU containing the context TamScheduler */
   let relevantObjectSystemMessages: Set(ObjectSystemMessage) =
       getRelevantObjectSystemMessages() in
   /* enforce that only one taskStartEvt DSE of relevantObjectSystemMessages ticks
        in the case of multiple fireable ones */
   Relation Exclusion(
     clocks→relevantObjectSystemMessages.taskStartEvt
   )
```

Listing B.7: ECL pseudocode for ClockResets

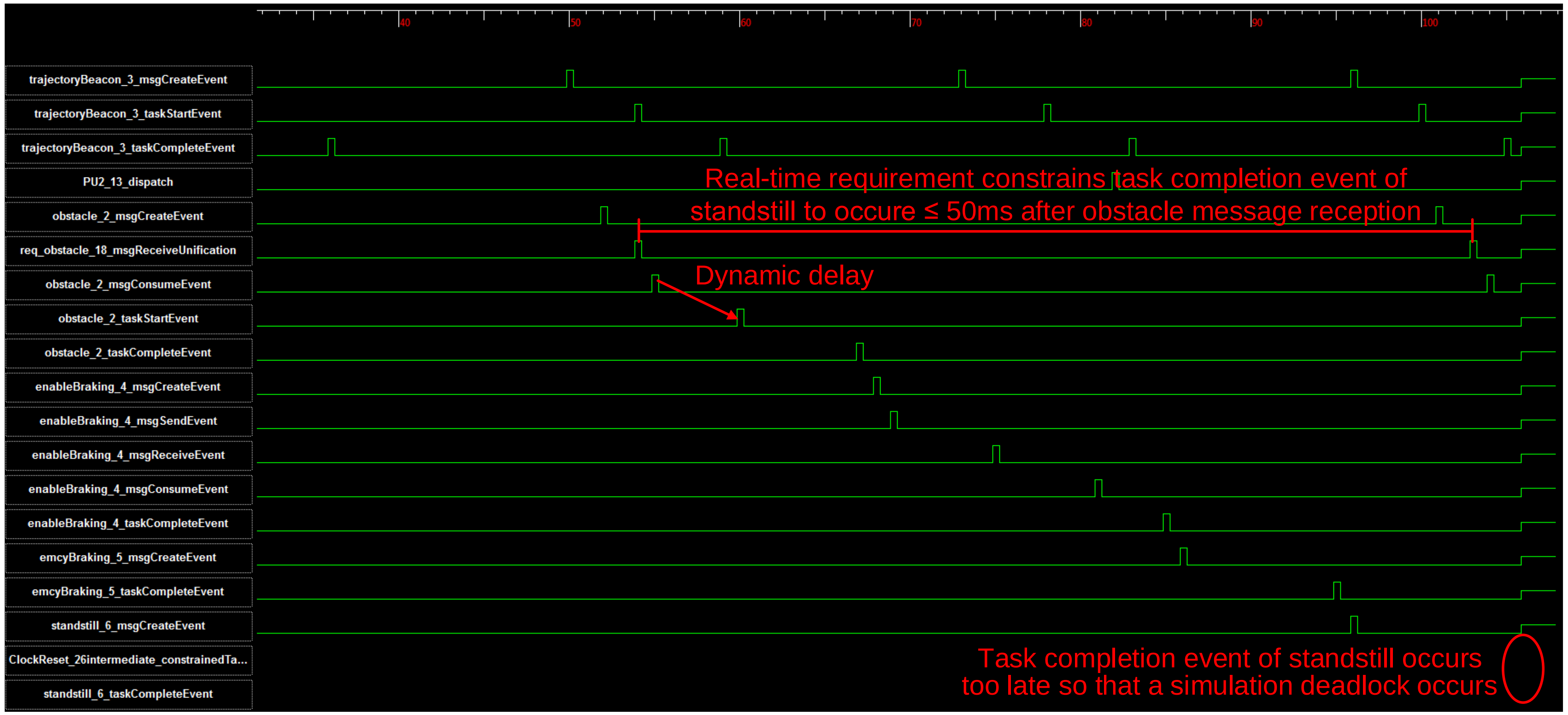
```
context ClockReset

 /* Hot time condition has the form c < value (maximal delay) */
 inv rtReqStrictUpperBound:
  let timeCondition:String = self.getAssociatedTimeCondition() in
  (timeCondition.isHot() and timeCondition.getOperator() = "<") implies (
    /* get messages before context clock reset */
    let precedingMessages:Set(ModalMessage) = self.getPrecedingMessages() in
    /* get last message reception unification occurrence before context clock
        reset */
    let precedingMsgReceiveUnification:Event = Expression Sup(
      clocks→precedingMessages.msgReceiveUnification
    ) in
    /* get messages in between context clock reset and time condition */
    let constrainedMessages:Set(ModalMessage) = self.getMessagesUntil(
        timeCondition) in
    /* get last task completion unification occurrence before time condition */
    let constrainedTaskCompleteUnification:Event = Expression Sup(
      clocks→constrainedMessages.taskCompleteUnification
    ) in
    /* define a new event that ticks timeCondition.getValue() time units after
        the tick of precedingMsgReceiveUnification */
    let upperBoundEvent:Event = Expression DelayFor(
      clockForCounting→self.getModel().globalTime,
      clockToDelay→precedingMsgReceiveUnification,
      delay→timeCondition.getValue()) in
    /* enforce that the last task complete unification occurrence of the
        constrained MSD messages ticks before upperBoundEvent. If the simulation
         cannot solve this, this represents a real-time requirement violation.
        */
    Relation Precedes(
      leftClock→constrainedTaskCompleteUnification,
      rightClock→upperBoundEvent
    )
  )

 /* Hot time condition has the form c ≤ value (maximal delay) */
 inv rtReqNonStrictUpperBound:
  let timeCondition:String = self.getAssociatedTimeCondition() in
  (timeCondition.isHot() and timeCondition.getOperator() = "≤") implies (
    let precedingMessages:Set(ModalMessage) = self.getPrecedingMessages() in
    let precedingMsgReceiveUnification:Event = Expression Sup(
      clocks→precedingMessages.msgReceiveUnification
    ) in
    let constrainedMessages:Set(ModalMessage) = self.getMessagesUntil(
        timeCondition) in
    let constrainedTaskCompleteUnification:Event = Expression Sup(
      clocks→constrainedMessages.taskCompleteUnification
    ) in
    let upperBoundEvent:Event = Expression DelayFor(
      clockForCounting→self.getModel().globalTime,
      clockToDelay→precedingMsgReceiveUnification,
       delay->timeCondition.getValue()) in
     /* enforce that the last task complete unification occurrence of the constrained MSD messages ticks before or at the same instant than upperBoundEvent. If the simulation cannot solve this, this represents a real-time requirement violation. */
     Relation NonStrictPrecedes(
       leftClock->constrainedTaskCompleteUnification,
       rightClock->upperBoundEvent
     )
   )

  /* Hot time condition has the form c > value (minimal delay) */
  inv rtReqStrictLowerBound:
   let timeCondition:String = self.getAssociatedTimeCondition() in
   (timeCondition.isHot() and timeCondition.getOperator() = “>”) implies (
     let precedingMessages:Set(ModalMessage) = self.getPrecedingMessages() in
     /* get last message reception unification occurrence before context clock reset */
     let precedingMsgReceiveUnification:Event = Expression Sup(
       clocks->precedingMessages.msgReceiveUnification
     ) in
     /* get messages after time condition */
     let messagesToDelay:Set(ModalMessage) = self.getMessagesAfter(timeCondition) in
     /* get first message creation unification occurrence after time condition */
     let delayedMsgCreateUnification:Event = Expression Inf(
       clocks->messagesToDelay.msgCreateUnification
     ) in
     /* define a new event that ticks timeCondition.getValue() time units after the tick of precedingMsgReceiveUnification */
     let lowerBoundEvent:Event = Expression DelayFor(
       clockForCounting->self.getModel().globalTime,
       clockToDelay->precedingMsgReceiveUnification,
       delay->timeCondition.getValue()) in
     /* enforce that the first message creation unification occurrence of the MSD messages to delay ticks after lowerBoundEvent */
     Relation Precedes(
       leftClock->lowerBoundEvent,
       rightClock->delayedMsgCreateUnification
     )
   )

  /* Hot time condition has the form c ≥ value (minimal delay) */
  inv rtReqNonStrictLowerBound:
   let timeCondition:String = self.getAssociatedTimeCondition() in
   (timeCondition.isHot() and timeCondition.getOperator() = “≥”) implies (
     let precedingMessages:Set(ModalMessage) = self.getPrecedingMessages() in
     let precedingMsgReceiveUnification:Event = Expression Sup(
       clocks->precedingMessages.msgReceiveUnification
     ) in
     let messagesToDelay:Set(ModalMessage) = self.getMessagesAfter(timeCondition) in
     let delayedMsgCreateUnification:Event = Expression Inf(
       clocks->messagesToDelay.msgCreateUnification
     ) in
     let lowerBoundEvent:Event = Expression DelayFor(
       clockForCounting->self.getModel().globalTime,
       clockToDelay->precedingMsgReceiveUnification,
       delay->timeCondition.getValue()) in
     /* enforce that the first message creation unification occurrence of the MSD messages to delay ticks after or at the same instant than lowerBoundEvent */
     Relation NonStrictPrecedes(
       leftClock->lowerBoundEvent,
       rightClock->delayedMsgCreateUnification
```

```
      )
   )
```

Listing B.8: ECL pseudocode for analysis contexts

```
context TamAssumptionMSD
 inv periodicPattern:
  self.pattern.isPeriodic() implies (
    let period:Integer = self.pattern::period in
    let periodicActivation:Event = Expression PeriodicOffsetP(
      baseClock->self.getModel().globalTime,
      period->period) in
    let initialMSDMsg:ModalMessage = self.getInitialMessage() in
    let initialObjSysMsg:ObjectSystemMessage = initialMSDMsg.
        getObjectSystemMessage() in
    Relation Coincides(
      clock1->initialObjSysMsg.msgCreateEvt,
      clock2->periodicActivation
    )
  )

 inv sporadicPattern minInterarrivalRate:
  (self.pattern.isSporadic() and self.pattern::ArrivalRate <> 1) implies (
   let minInterarrivalRate:Integer = self.pattern::minArrivalRate in
   let sporadicMinInterarrival:Event = Expression PeriodicOffsetP(
     baseClock->self.getModel().globalTime,
     period->minInterarrivalRate) in
    let initialMSDMsg:ModalMessage = self.getInitialMessage() in
    let initialObjSysMsg:ObjectSystemMessage = initialMSDMsg.
        getObjectSystemMessage() in
   Relation NonStrictPrecedes(
     leftClock->sporadicMinInterarrival,
     rightClock->initialObjSysMsg.msgCreateEvt
   )
  )

 /* In case no min arrival rate is specified, only the max arrival rate value is
     enforced */
 inv sporadicPattern maxInterarrivalRateOnly:
  (self.pattern.isSporadic() and self.pattern::maxArrivalRate <> 1 and self.
      pattern::minArrivalRate = 1) implies (
   let maxInterarrivalRate: Integer = self.pattern::maxArrivalRate in
   let sporadicMaxInterarrival:Event = Expression PeriodicOffsetP(
     baseClock->self.getModel().globalTime,
     period->maxInterarrivalRate) in
   let initialMSDMsg:ModalMessage = self.getInitialMessage() in
   let initialObjSysMsg:ObjectSystemMessage = initialMSDMsg.
       getObjectSystemMessage() in
   Relation NonStrictPrecedes(
     leftClock->initialObjSysMsg.msgCreateEvt,
     rightClock->sporadicMaxInterarrival
   )
  )

 /* In case a min arrival rate is specified, the difference to the min arrival
     rate is enforced */
 inv sporadicPattern maxInterarrivalRateWithMinInterarrival:
  (self.pattern.isSporadic() and self.pattern::maxArrivalRate <> 1 and self.
      pattern::minArrivalRate <> 1) implies (
   let minMaxInterarrivalRateDifference:Integer = self.pattern::maxArrivalRate
       self.pattern::minArrivalRate in
   let sporadicMaxInterarrivalDifference:Event = Expression DelayFor(
      clockForCounting->self.getModel().globalTime,
      clockToDelay->sporadicMinInterarrival,
      delay->minMaxInterarrivalRateDifference) in
    let initialMSDMsg:ModalMessage = self.getInitialMessage() in
    let initialObjSysMsg:ObjectSystemMessage = initialMSDMsg.
        getObjectSystemMessage() in
    Relation NonStrictPrecedes(
      leftClock→initialObjSysMsg.msgCreateEvt,
      rightClock→sporadicMaxInterarrivalDifference
    )
  )
```

B.3 Exemplary Timing Analysis: TIMESQUARE Screenshot


40
50
60
70
80
90
100
trajectoryBeacon_3_msgCreateEvent
trajectoryBeacon_3_taskStartEvent
trajectoryBeacon_3_taskCompleteEvent
PU2_13_dispatch
obstacle_2_msgCreateEvent
req_obstacle_18_msgReceiveUnification
obstacle_2_msgConsumeEvent
obstacle_2_taskStartEvent
obstacle_2_taskCompleteEvent
enableBraking_4_msgCreateEvent
enableBraking_4_msgSendEvent
enableBraking_4_msgReceiveEvent
enableBraking_4_msgConsumeEvent
enableBraking_4_taskCompleteEvent
emcyBraking_5_msgCreateEvent
emcyBraking_5_taskCompleteEvent
standstill_6_msgCreateEvent
ClockReset_26intermediate_constrainedTa...
standstill_6_taskCompleteEvent

Real-time requirement constrains task completion event of standstill to occure ≤ 50ms after obstacle message reception

Dynamic delay

Task completion event of standstill occurs too late so that a simulation deadlock occurs

Figure B.30: Exemplary Timing Analysis—TIMESQUARE Screenshot (cf. Section 4.5)

B.4 Case Study Details: Hypotheses H2 and H3 for the Timing Analysis based on MSDs

Table B.1: Detailed model element amounts of the platform-specific MSD specifications for hypothesis H2—platform-independent model elements

View Type	Model Element (-> Attribute)	# MSD-spec-1	# MSD-spec-2	# MSD-spec-3	# MSD-spec-4
Classifier	Component Type	6	6	8	8
	Component Type -> name	6	6	8	8
	Port	12	12	18	18
	Port -> name	12	12	18	18
	Port -> type	12	12	18	18
	Port -> isConjugated	6	6	9	9
	Interface	6	6	8	8
	Interface -> name	6	6	8	8
	Operation	6	6	32	32
	Operation -> name	6	6	32	32
	Parameter	0	0	0	0
	Parameter -> name	0	0	0	0
	Parameter -> type	0	0	0	0
Architecture	Collaboration	1	1	1	1
	Collaboration -> name	1	1	1	1
	Component Role	6	6	8	8
	Component Role -> name	6	6	8	8
	Component Role -> type	6	6	8	8
	Component Role -> partKind	6	6	8	8
	Connector	6	6	10	10
	Connector -> name	6	6	10	10
Interaction	MSD	4	4	24	24
	MSD -> name	4	4	24	24
	Lifeline	12	12	81	81
	Lifeline -> represents	12	12	81	81
	Message	8	8	86	86
	Message -> signature	8	8	86	86
	Message -> connector	8	8	86	86
	Message -> argument	0	0	0	0
	ModalMessage -> ExecutionKind	8	8	86	86
	ModalMessage -> Temperature	8	8	86	86
	ModalCondition	0	0	0	0
	ModalCondition -> Expression	0	0	0	0
	ModalCondition -> Temperature	0	0	0	0
	ClockReset	1	1	6	6
	ClockReset -> Expression	1	1	6	6
	TimeCondition	1	1	6	6
	TimeCondition -> Expression	1	1	6	6
	TimeCondition -> Temperature	1	1	6	6
	Sum	**193**	**193**	**883**	**883**

Table B.2: Detailed model element amounts of the platform-specific MSD specifications for hypothesis H2—platform-specific model elements

View Type	**Model Element (-> Attribute)**	**# MSD-spec-1**	**# MSD-spec-2**	**# MSD-spec-3**	**# MSD-spec-4**
Platform	TamResourcePlatform	1	1	1	1
	TamResourcePlatform -> name	1	1	1	1
	TamComConnection	1	5	1	7
	TamComConnection -> name	1	5	1	7
	TamComConnection -> capacity	1	5	1	7
	TamComConnection -> protocol	1	5	1	7
	TamComConnection -> usedComServices	1	5	1	7
	TamProtocol	1	3	1	3
	TamProtocol -> name	1	3	1	3
	TamProtocol -> transmOvhd	1	3	1	3
	TamProtocol -> msgCtrlOvhd	1	3	1	3
	TamProtocol -> encodeRate	1	3	1	3
	TamProtocol -> decodeRate	1	3	1	3
	TamComService	1	1	1	1
	TamComService -> name	1	1	1	1
	TamComService -> transmOvhd	1	1	1	1
	TamECU	2	5	2	7
	TamECU -> name	2	5	2	7
	TamProcessingUnit	2	5	2	5
	TamProcessingUnit -> name	2	5	2	5
	TamProcessingUnit -> numCores	2	5	2	5
	TamProcessingUnit -> speedFactor	2	5	2	5
	TamRTOS	2	2	2	2
	TamRTOS -> name	2	2	2	2
	TamRTOS -> scheduler	2	2	2	2
	TamRTOS -> comChannels	1	1	1	1
	TamRTOS -> sharedRes	1	1	1	1
	TamRTOS -> osServices	1	1	1	1
	TamOSComChannel	1	1	1	1
	TamOSComChannel -> name	1	1	1	1
	TamOSComChannel -> capacity	1	1	1	1
	TamSharedOSResource	1	1	1	1
	TamSharedOSResource -> name	1	1	1	1
	TamSharedOSResource -> accessDelay	1	1	1	1
	TamOSService	1	1	1	1
	TamOSService -> name	1	1	1	1
	TamOSService -> backgroundUtilization	1	1	1	1
	TamScheduler	2	2	2	2
	TamScheduler -> name	2	2	2	2
	TamScheduler -> isPreemptible	2	2	2	2
	TamScheduler -> schedPolicy	2	2	2	2
	TamComInterface	2	9	2	11
	TamComInterface -> name	2	9	2	11
	TamComInterface -> commTxOhvh	2	9	2	11
	TamComInterface -> commRcvOhvh	2	9	2	11
	TamResourceAccess	1	1	1	1
	TamResourceAccess -> numAccess	1	1	1	1
	TamResourceAccess -> resource	1	1	1	1
	TamOperation	6	6	6	6
	TamOperation -> execTime	6	6	6	6
	TamOperation -> msgSize	6	6	6	6
	Allocation	5	11	5	18
	Sum	**88**	**172**	**88**	**201**

Table B.3: Individual transformation execution time measurements for deriving CCSL models from the platform-specific MSD specifications for hypothesis H3

	MSD-spec-1	**MSD-spec-2**	**MSD-spec-3**	**MSD-spec-4**
Preprocessing run 1	442	379	4,998	4,646
Preprocessing run 2	424	350	4,901	4,973
Preprocessing run 3	432	417	4,970	4,717
Preprocessing run 4	407	344	4,989	4,709
Preprocessing run 5	471	360	4,938	4,726
Preprocessing run 6	427	365	4,894	4,842
Ø Preprocessing runs	**434**	**369**	**4,948**	**4,769**
MSD-to-CCSL transformation run 1	10,042	10,644	39,708	48,218
MSD-to-CCSL transformation run 2	9,075	10,683	39,302	49,099
MSD-to-CCSL transformation run 3	9,006	10,158	38,503	48,451
MSD-to-CCSL transformation run 4	9,583	10,285	36,599	48,324
Ø MSD-to-CCSL transformation runs	**9,427**	**10,443**	**38,528**	**48,523**
Ø Overall transformation execution time	**9,860**	**10,812**	**43,476**	**53,292**

C

Own Publication Contributions

[*BGH$^+$14] This paper introduces Real-time Play-out. Based on initial concepts developed in [*BBG$^+$13], I contributed to the main section describing an exemplary timed state graph.

[*DFHT13] This technical report describes a requirements engineering approach integrating two existing approaches (one based on controlled natural language and one based on semi-formal models) and the application of the integrated approach to a case study. I am one of the main authors of this report, and contributed to the basic controlled natural language approach as well as further sections.

[*FH14; *FH15; *FHM14] Markus Fockel and I were the main authors of these papers and jointly developed the underlying concepts of controlled natural language requirements and their synchronization with model-based engineering.

[*FHKS18; *FHKS17] These publications present requirement patterns for MSDs. I contributed to the real-time requirement patterns as well as the abstract (for [*FHKS18]), each the introduction, and each the conclusion.

[*FHH$^+$12] This book chapter reports results of the the automotive application project in the context of the German research project “SPES 2020” [SPES2020]. I was the deputy coordinator of the automotive application project and contributed to the concepts, texts, and figures presented in Section 12.3.2, particularly to the parts on the requirement patterns and the transition to MBSE.

[*FHM12] This paper extends the automotive development process presented in [*HMM11] with the semi-automatic establishment of valid traceability between systems requirements engineering and MBSE using model transformations as well as OCL constraint checks. Markus Fockel and I were the main authors of this paper, and we jointly developed the underlying concepts as well as contributed the major parts to all sections. Furthermore, I presented the paper.

[*GTH16] This paper extends SYSML4CONSENS as part of a commercial MBSE tool with the possibility to apply project management activities for non-technical stakeholders. Based on initial concepts developed in [*Gre15], I reviewed the overall paper.

[*HFK$^+$16] This technical report introduces the EBEAS and consolidates the MSD syntax and semantics that this thesis works with. I coordinated the creation of the technical report and contributed to all chapters, particularly to the process description and the EBEAS example. In addition, I reviewed other sections of the report.

[*HBM$^+$15; *HBM$^+$16] These publications build the basis for Chapter 3. Based on initial concepts developed in [*Ber15], I coordinated the creation of all publications, contributed to each all sections, and presented the conference paper [*HBM$^+$15].

[*HBM$^+$17] Based on the results published in [*HBM$^+$15; *HBM$^+$16], I wrote this consolidating extended abstract and presented it.

[*HFKS16] This publication introduces maturity levels for requirements engineering based on the author's industrial experiences. I coordinated the overall publication and contributed to all sections in joint work with the other authors.

[*HM13] This paper builds the basis for Section 3.2. Based on initial concepts developed in [*BBG$^+$13], I coordinated the creation of the publication, contributed the major part to all sections, and presented the paper.

[*HMD11] This paper describes an approach for the identification and correction of requirement defects in controlled natural language requirements. I coordinated the creation of the publication, contributed the major part to all sections, and presented the paper.

[*HMM11] This paper describes the constructive part of a model-based automotive development process. I coordinated the creation of the publication and contributed to all sections, particularly to the parts on the requirement patterns and the transition to MBSE. Furthermore, I presented the paper.

[*HMSN10] This paper introduces MECHATRONICUML real-time statecharts to SysML. I contributed to the concepts presented in the paper.

[*Hol10] This publication introduces our initial tool support for controlled natural language requirements specification in the automotive sector. I wrote the publication.

[*HS14] This paper introduces the consideration of delays induced by connector latencies and software execution times in Real-time Play-out and thereby presents preliminary work for Chapter 4. Based on initial concepts developed in [*Shi14], I coordinated the creation of the publication, contributed to all sections, and presented the paper.

[*HT08] This paper presents the technical realization of a new MECHATRONICUML component metamodel and component story diagrams. Based on initial concepts developed in [*Hol08], I contributed to all sections and presented the paper.

[*KDHM13] This paper builds the basis for Section 3.9.1.1 and introduces SYSML4CONSENS, a further language extension to specify system models more rigorously, and static modeling rules that the system models are verified against automatically. I particularly contributed to the conceptual and technical sections (i.e., the SYSML4CONSENS profile section, the example system model, the verification concept, and the implementation / evaluation section).

[*KHD14] This paper introduces model transformations from MSD specifications to CCSL models and thereby presents preliminary work for Chapter 4. Based on initial concepts developed in [*Koc13], I coordinated the creation of the publication, contributed to all sections, and presented the paper.

[*KHL18; *KHL17; *KHSL16] These publications present results from a research project on variant modeling and on the data exchange by means of STEP models for mechatronic production systems. I wrote each the abstract, introduction, and conclusion as well as reviewed each (parts of) the publications.

[*MFH15] This publication describes the application of SysML for the purpose of functional safety in the automotive sector. I contributed to the parts on MBSE and CONSENS as well as reviewed the overall publication.

[*MH11] This publication describes a semi-automatic transition from MBSE with SysML to the software design with AUTOSAR in the automotive sector. I contributed to all parts and reviewed the overall paper.

[*MHKM15] This publication describes an automatic derivation of initial AUTOSAR models from UML software design specifications in the automotive sector. I contributed to all parts and reviewed the overall paper.

[*MHM11] This publication describes an approach for automatically verifying automotive operating system properties inducing timed event chains w.r.t. real-time requirements specified in controlled natural language using a commercial timing analysis tool. Thereby, it presents preliminary work for Chapter 4. I contributed to all parts, particularly to the controlled natural language real-time requirements.

[*MHNM10] This project-internal publication describes an example of the automotive system "Body Control Module". I contributed to all parts.

[*PHM14] This publication presents an automatic derivation of MECHATRONICUML software design models from SYSML4CONSENS system models. I contributed the major part to all sections.

[*PHMG14] This paper introduces the distinction between coordination and control behavior in terms of terminology and presents an automatic derivation of Modelica models from MECHATRONICUML software design specifications. Particularly, I contributed to the introduction, the conclusion, and the transformation part.

[*PHM16] This publication presents a concept for the automatic computation of allocations of MECHATRONICUML software component models to MECHATRONICUML platform models. I contributed to the abstract, introduction, and conclusion as well as reviewed the publication.

[*SGH17] This paper introduces an approach for learning environment assumptions for under-specified MSD specifications by means of genetic algorithms. I wrote the abstract, the introduction, and the conclusion as well as presented the paper.

[*SNH$^+$10] This paper introduces an automatic self-healing approach for automotive systems. I particularly contributed to the running example of the driver assistance system "Adaptive Cruise Control".

[*THHO08] This paper introduces a new component metamodel as well as component story diagrams to the MECHATRONICUML design language. Based on initial concepts developed in [*Hol08], I contributed technical details.