PADERBORN UNIVERSITY
The University for the Information Society

Faculty for Computer Science, Electrical Engineering and Mathematics

A Model-driven Software Construction Approach for Cyber-physical Systems

Uwe Pohlmann

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Referee:
Prof. Dr. Matthias Tichy
Prof. Dr. Gregor Engels

Paderborn, April 17, 2018

ISBN 978-3-00-059657-5

Abstract

Systems and software engineers realize innovative functionality in cyber-physical systems by connecting previously independent systems. The development of these systems becomes more challenging because the systems' complexity, the amount of the safety-critical software, the heterogeneity of the underlying platforms, and the connectivity of the systems increase steadily. However, existing model-driven approaches do not cover the different engineers' concerns efficiently during the different development tasks. Many of these approaches have a focus on the formal design and the verification of safety and real-time properties. Though, the question how to retain the systems' safety during following construction tasks remains open. This thesis focuses on the construction tasks: physical system integration validation, resource allocation, and the implementation. It presents a highly automated end-to-end model-driven approach for the development of interacting cyber-physical systems that are realized on heterogeneous, distributed platforms. The approach covers the validation of the system integration via model-in-the-loop simulation, constraint-driven software to hardware allocation, and generative software construction. Consequently, the approach retains the system's safety with respect to systematic errors through different development tasks. It enables engineers to detect and avoid design and implementation flaws. Thereby, it improves the reliability of systems during their operation. It enables an effective and efficient seamless development. This thesis evaluates the approach by providing an integrated development environment and conducting several case studies from the automotive and automation domain. The evaluation shows the effective and in comparison to state of the art approaches more efficient application of: (1) the model-in-the-loop simulation in the context of virtual prototypes, (2) the allocation constraint specification and the automated allocation planning, and (3) the generation of executables for heterogeneous, distributed embedded platforms.

Zusammenfassung

Systemingenieure und Softwaretechniker realisieren innovative Funktionen von cyber-physischen Systemen durch die Vernetzung von vormals unabhängigen Systemen. Die Entwicklung dieser Systeme wird schwieriger, weil die Systemkomplexität, die Menge an sicherheitskritischer Software, die Heterogenität der verwendeten Plattformen und der Vernetzungsgrad stetig ansteigen. Jedoch bedienen existierende modellgetriebene Entwicklungsansätze die verschiedenen Bedürfnisse der beteiligten Ingenieure während der verschiedenen Entwicklungsaufgaben nicht effizient genug. Existierende Ansätze fokussieren sich typischerweise auf den formalen Entwurf und die Verifizierung von Sicherheits- und Echtzeiteigenschaften. Sie beantworten jedoch nicht die Frage wie die Systemsicherheit während der folgenden Konstruktionsschritte beibehalten werden kann. In dieser Dissertation fokussiere ich mich auf die Konstruktionsschritte Validierung der Systemintegration, Zuordnung von Systemressourcen und auf die Implementierung. Ich stelle einen durchgängig modellgetriebenen Entwicklungsansatz für interagierende cyber-physische Systeme vor, welche durch heterogene, verteilte Software- und Hardwareplattformen gekennzeichnet sind. Der Ansatz beinhaltet Verfahren für die Systemintegrationsvalidierung mittels „Model-in-the-Loop“ Simulation, die automatische Softwareverteilung, welche durch Entwurfsanforderungen stark beschränkt ist, und die generative Softwarekonstruktion. Dementsprechend stellt der Ansatz die Systemsicherheit in Bezug auf systematische Fehler während der verschiedenen Entwicklungsaufgaben sicher. Er ermöglicht Ingenieuren und Technikern Entwurfs- und Implementierungsfehler frühzeitig zu entdecken und im Folgenden systematisch zu vermeiden. Hierdurch verbessert der Ansatz die Zuverlässigkeit der Systeme während ihres Betriebs. Er ermöglicht eine effektive und effiziente, durchgängige Entwicklung. Zur Anwendung und Evaluierung des Ansatzes stelle ich eine integrierte Entwicklungsumgebung zur Verfügung und führe verschiedene Fallstudien aus der Automobil- und der Automatisierungsdomäne durch. Die Evaluierung zeigt die effektive und im Vergleich zum Stand der Technik effizientere Anwendung von (1) der „Model-in-the-Loop“ Simulation im Kontext eines virtuellen Prototyps, (2) der Spezifikation von Allokationsbeschränkungen und der automatisierten Allokationsplanung und (3) der Generierung von ausführbaren Programmen für heterogene, verteilte, vernetzte eingebettete Plattformen.

Danksagung

Ich möchte mich an dieser Stelle bei meinen Freunden, Kollegen und bei meiner Familie bedanken, ohne die diese Arbeit nicht möglich gewesen wäre. Insbesondere das angenehme Arbeits- und Forschungsklima des Heinz Nixdorf Instituts, des Fraunhofer-Instituts IEM und des s-labs, welches durch die vielen Kollegen geprägt wird, die vielen gemeinsamen Stunden und die anregenden Diskussionen, haben mir immer geholfen.

Ich möchte mich bei meinen Betreuern Wilhelm Schäfer und Matthias Tichy bedanken. Durch die vielen Anregungen, Rückmeldungen und Diskussionen konnte ich meine Forschung vorantreiben und in dieser Arbeit darstellen. Neben dem fachlichen Rat waren die persönlichen Ratschläge sehr inspirierend. Gregor Engels danke ich für die Übernahme des weiteren schriftlichen Gutachtens. Ich möchte den weiteren Mitgliedern meiner Promotionskommission, Eric Bodden, Matthias Meyer und Stefan Sauer, danken.

Meine Arbeit hat von der interdisziplinären Arbeit im Forschungsprojekt ENTIME profitiert. Hier möchte ich den bisher nicht erwähnten Projektmitarbeitern Farisoroosh Abrishamchian, Harald Annacker, Frank Bauer, Daniel Kruse, Sarah Flottmeier, Viktor Just, Thomas Schierbaum, Heinrich Teichrieb und Felix Oestersötebier danken. Dies gilt auch für die Mitarbeiter im Forschungsprojekt it's OWL – Intelligente Vernetzung, in dem ich mitarbeiten durfte. Namentlich gilt der Dank Peer Adelt, Johannes Ax, Lars Dürkop, Volker Geneiß, Thorsten Jungeblut, Jan Jatzkowski, Uwe Mönks, Konstantin Nußbaum, Henning Trsek, Jens Otto und Lukasz Wisniewski.

Weiterhin möchte ich meinen Kollegen Stefan Dziwok, Christopher Gerking, Christian Heinzemann und David Schubert aus der MechatronicUML Entwicklungsgruppe danken mit denen ich viele Konzepte detailliert entwickelt, diskutiert und verbessert habe. Dank gilt auch meinen Bürokollegen Stefan Dziwok, Christian Bremer und Sven Merschjohann und den weiteren aktuellen und ehemaligen Kollegen der Fachgruppe Softwaretechnik. Diese sind namentlich Anas Anis, Matthias Becker, Steffen Becker, Manuel Benz, Christian Brenner, Christian Brink, Eric Bodden, Andreas Peter Dann, Nicola Danielzik, Tobias Eckardt, Markus Fockel, Jens Frieben, Johannes Geismann, Faezeh Ghassemi, Joel Greenyer, Jutta Haupt, Stefan Henkler, Ben Hermann, Jörg Holtmann, Thorsten Koch, Stefan Krüger, Sebastian Lehrig, Renate Löffler, Jürgen Maniera, Ahmet Mehic, Sven Merschjohann, Matthias Meyer, Vera Meyer, Lisa Nguyen, Faruk Pasic, Goran Piskachev, Marie Christin Platenius, Claudia Priesterjahn, Jan Rieke, David Schmelter, Philipp Schubert, Johannes Späth, Lars Stockmann, Christian Stritzke, Julian Suck, Oliver Sudmann, Dietrich Travkin, Markus von Detten und Benedict Wohlers.

Die Integration von Studenten in Forschungsaktivitäten ist mir sehr wichtig. Deshalb wäre diese Arbeit in der vorliegenden Form nicht möglich gewesen ohne die Mitarbeit der Studenten Jan Bobolz, Melanie Bruns, Ingo Budde, Mike Czech, Andreas Peter Dann, Johannes Geismann, Marcus Hüwe, Arthur Krieger, Jan-Niclas Strüwer, Sebastian Thiele, Mario Rose, Rebekka Wohlrab und Boris Wolf.

Zum Schluss möchte ich mich bei meiner Familie für ihre Unterstützung bedanken. Dank gilt meinen Brüdern Martin und Andreas, wie auch meiner Mutter Mathilde und meinem Vater Dietrich. Besonderer Dank gilt meiner Freundin Anna für ihre Mithilfe und Unterstützung.

Contents

1

Introduction

Cyber-Physical Systems (CPSs) are the driving force for rapid changes in our daily life [aca11]. A main characteristic of CPSs is the close interaction with each other and with their environment [MKM+16]. Innovations in CPSs contribute to safety, mobility, efficiency, and comfort. CPSs are heterogeneous. Engineers develop them for several domains, like automotive, railway, production, and avionic [aca11]. The requirements of different domains vary to a large extent. An example of a CPS is a modern car that gets and sends information about the current traffic, senses its environment, like the current road conditions, and is controlled semi-automatically. The connected computers that execute the software within a CPS are called Electronic Control Units (ECUs) [VBK10]. ECUs are heterogeneous, e.g., they differ in their processor architecture and offer only limited hardware resources, like processing speed or memory size [VBK10]. Furthermore, different, heterogeneous, operating systems, runtime environments, and communication middlewares provide rich functionalities to realize and manage the software behavior and communication.

Today, the software components of a car are distributed over 100 connected, heterogeneous ECUs that are networked by multiple communication channels [VBK10; PG14; Mah05]. As a result, the system's complexity is high and still grows due to the increasing amount of software within CPSs. The number of functions within a system and the number of connected systems that interact with each other and form a distributed *system of systems* [BS06] increases steadily. However, software engineers can realize new advanced functionalities by *cooperative* CPSs if they are able to manage the development complexity [BKN+05; KK12]. The connected CPSs communicate via asynchronous message exchange [CDKB11]. An example for a cooperative CPS is a cooperative overtaking assistance system [TSZ09]. The involved cars and the infrastructure communicate with each other to reduce the risk of an overtaking maneuver [TSZ09]. Therefore, software engineers have to manage and handle the resulting system's complexity during the development. This requires *systematic software engineering practices* that guide engineers and automate complex, error-prone engineering tasks [MKM+16].

Additional to functional requirements, CPSs are liable to legal safety requirements in general [IEC61508] and are liable to domain-specific safety requirements, like automotive safety [ISO26262-6]. Especially, the message-based communication and its integration within the system has to be reliable. The sources of errors are manifold and diverse. Incorrect information can lead to the wrong conclusion about the own system or about other systems. Messages may contain incorrect information due to a buggy integration of the communication software and the control software. Messages can get lost due to a high network traffic in combination with a bad software allocation. The data integrity of messages can be damaged due to a wrong encoding and decoding, which is required by one of the varying network protocols.

Furthermore, CPSs are part of the physical world and interact closely with the environment. Both, CPSs and the environment are in motion. As a result, CPSs run under real-time constraints [Kop11] and have to provide a certain Quality of Service (QoS) [DPST11]. It is a

threat to the safety if a system reacts with the wrong timing, e.g., if a car applies the brakes too late. Some information is also only valid for a certain time and can result in a wrong conclusion if a receiver processes a message and the containing information too late.

Model-Driven Engineering (MDE) is a software engineering paradigm that addresses the software development complexity of CPSs [Sch06]. It combines *Domain-Specific Languages (DSLs)* and *model transformation engines*. DSLs represent domain concepts effectively by using known concepts of the domain experts [VBD+13]. Model transformation engines transform the domain knowledge to formal, analyzable models. Thereby, they hide the complexity of the formal models from the domain experts. The goal of MDE is to derive *correctness-by-construction* by using models with formal semantics and using strong tool support [Cha06].

Our MDE approach that focuses on the platform-independent design and verification of cooperative CPSs is MechatronicUML [*BDG+14; *DPP+16]. MechatronicUML provides a coordination model [*DPP+16; Dzi17] that defines communication protocols and a component model [*DPP+16; Hei15]. The component model defines the platform-independent software architecture and the software (communication) behavior. MechatronicUML adapts concepts like the component-based architecture specification and the state-based behavior specification from the Unified Modeling Language (UML) [UML] for the development of CPSs.

Previous work on MechatronicUML provides a formal semantics [Hei15] that enables timed model checking [Hir08; GDHS15; Dzi17] for formal verification of the coordination model. Furthermore, previous work on MechatronicUML provides an approach for specifying a software architecture that integrates event-discrete, state-based software components, and time-continuous, control software components [BGO06]. MechatronicUML provides examples and case studies [*TJD+12; *GST14; *PTD+14; *HSD15], which are engineered with the MechatronicUML tool suite [*DGB+14] in a model-driven way. As a result, MechatronicUML supports the specification and analysis of a platform-independent software architecture and real-time (communication) behavior for CPSs. Additionally, previous work provides concepts for reconfiguration of the software architecture [Bur06b; THHO08; Tic09; HB13; Hei15; SHG16]. In this thesis, we do not consider reconfiguration. We assume a static configuration of the software architecture.

1.1 Problem Statement

We cover the following three problems during this thesis for constructing CPSs. The problems refer to the properties, which the introduction describes, and to problems that exits when using state-of-the art MDE approaches for developing CPSs during the development phases of the simulation-based validation, the resource allocation, and the final software construction phase itself.

1.1.1 Simulation of Cyber-physical Systems

The reliability of the software of CPSs depends on the ability to operate in the expected way over time [ISO25010]. That means that each system part and the integration of the event-discrete, state-based software, the time-continuous controller software, the physical properties, and the environment does not fail and perform their intended functions. The simulation of CPSs is able to validate the overall, integrated system's behavior. Therefore, we require an integrated model of event-discrete software parts, time-continuous software parts, physical system parts, and the environment to validate the correct integration [ÅK14]. However, current model-driven analysis approaches that verify and validate the system's behavior do not consider or only limitedly the combination of discrete, event-driven interaction behavior and continuous, physical-driven behavior. UML-based standards like Modeling and Analysis of Real-Time

Embedded Systems (MARTE) [MARTE] or UML-based methods like CHESS [CSCS13] support modeling the software of CPSs. However, they do not provide formal semantics enabling to analyze real-time (communication) behavior [Hei15]. Formal approaches to analyze state-based, real-time behavior, like timed model checking on timed automata [BY04], are specialized to verify the conformance of the system's behavior to safety and liveness properties formally on all execution paths but fail to analyze hybrid (event-discrete/time-continuous) systems [AD94]. Formal approaches that use hybrid model checking approaches to analyze models that contain time-continuous differential equations and event-discrete timed automata are able to analyze only small models or have to overapproximate the behavior [Hen00; FLD+11].

Parallel work during this thesis on Model-in-the-Loop (MiL) simulation using MechatronicUML for analyzing the hybrid behavior of Heinzemann et al. [Hei15; HRS13; *HPR+12] from our joint research group depends on the commercial tool MATLAB Simulink [Sim]. It does not support the open Modelica standard [MODELICA]. In contrast to MATLAB, which provides the Stateflow language for modeling state-based behavior and message-based communication, Modelica does not provide modeling constructs for model the state-based behavior that can communicate with each other via messages. Therefore, the approach of Heinzemann et al. cannot be transferred to Modelica directly and a new solution is required. Furthermore, the approach of Heinzemann et al. requires a lot of manual effort to create the state-based behavior in combination with message-based communication if engineers like to work directly within MATLAB without using the MechatronicUML approach. An approach that allows to use the benefits of the MechatronicUML approach but also enables systems engineers to use Modelica directly without the additional hurdle to learn a new language improves the acceptability.

1.1.2 Specifying Constraints for Allocation Planning

The context of a software component, like the platform where it is allocated to, is crucial for the reliability and safety of its execution. Current model-driven engineering approaches can be split up in two fields of research cope with the allocation problem. On the one hand, *Architecture Description Languages (ADLs)*, like the Architecture Analysis and Design Language (AADL) [FG12], East-ADL [CFJ+11], or the UML profile MARTE [MARTE] provide language elements to describe the platform properties. However, these languages offer no support to define allocation constraints effectively and offer no algorithmic support for allocation planning. On the other hand, allocation planning and Design Space Exploration (DSE) approaches like DIF [MMM12], ArcheOpterix [ABGM09], or Autofocus3 [VEH14] use general algorithms, like Boolean Satisfiability Problem (SAT) solving, constraint logic programming [JM94], or offer specific meta-heuristics [BR03]. They restrict the engineers' flexibility, who are no experts in formalizing constraint satisfaction problems because they provide only predefined hard-coded allocation constraints but not a domain-specific language for software engineers [KPP+15]. The tools that provide model-driven engineering approaches are sound but they are inflexible considering new constraints directly without extending the provided frameworks, which requires knowledge of operations research or constraint programming. In contrast, general constraint solving approaches, like SAT, CLP, ILP solvers [PBG05; JL87; KLMJ10] or heuristic-based algorithms [BR03], are very flexible. However, software engineers require domain knowledge about how to encode an allocation problem formally, which is complex and error-prone.

1.1.3 Constructing Software for Distributed Cyber-Physical Systems

Even the construction of the system's software for a distributed deployment is not or only considered to a limited extent by model-driven engineering approaches. UML-based [UML] standards like MARTE [MARTE] or UML-based methods like CHESS [CSCS13] support modeling the software and platform of CPSs. However, they do not provide a systematics for the software construction in different contexts by means of the implementation that enables a distributed deployment on heterogeneous software and hardware platforms. Component-based development approaches, like SOFA HI [PWT+08; HPM+10], ProCom [BCC+08; CFMS10], or DEECo [MKM+16; BGH+13] rely on general-purpose containers or runtime environments that do not distinguish between different component types and its specific context requirements [BHK06]. Component-based deployment frameworks and standards, like Autosar [AUTOSAR14c], are designed for homogeneous systems and do not consider the functional requirements of the application software and its communication behavior.

Previous work on MechatronicUML of Burmester and Henkler for automated implementation [BGS05; Bur06b; Hen12] rely on a general purpose runtime environment that does not provide modeling and generation mechanisms for of-the-shelf communication channels, middlewares, and different components' context requirements. As a result, platform-specific models, deployment mapping models, or transformations are missing in MechatronicUML to construct the software that considers these requirements of CPSs.

1.2 Contribution

The goal of this thesis is to retain safety during the construction after the software design with respect to systematic errors. Therefore, this thesis contributes a model-driven engineering approach for distributed CPSs. In contrast to related approaches, we start with formal verified platform-independent models. As a key novelty, we develop a model-driven construction approach for CPSs that covers three aspects: (1st) simulation-based MiL testing, (2nd) allocation engineering, and (3rd) software construction.

We focus on these three aspects because in combination, our contributions retain safety with respect to systematic errors during the proceeding development after specifying and verifying a platform-independent software specification. Firstly, software and systems engineers are able to test the specified software in the loop with an integrated simulation model of the physical system and its environment. Secondly, the software engineers construct a correct software to hardware allocation automatically. Lastly, they refine the platform-independent specification to a platform-specific specification that considers the computed allocation. Software engineers can finally create software artifacts, which run on heterogeneous software and hardware platforms by using our approach. The artifacts are able to communicate with each other via various communication channels by using different off-the-shelf communication middlewares in a distributed environment.

We build our approach on the basis of our MDE approach MechatronicUML [*BDG+14; *DPP+16]. Thereby, we profit especially from the capabilities to specify and verify safety and liveness properties formally of advanced real-time coordination behavior of cooperating CPSs [Dzi17]. Furthermore, we profit from the advanced MechatronicUML component model that has its strength in specifying a software architecture for CPSs [Hei15]. As a result, MechatronicUML verifies the software components using a compositional model checking approach and a refinement check for message-based communication protocols [Hei15]. We extend the development process of MechatronicUML [HSST13; *DPP+16] to integrate our

contributions seamlessly. Furthermore, we integrate and contribute our tooling implementation as part of the MechatronicUML tool suite [*DGB+14].

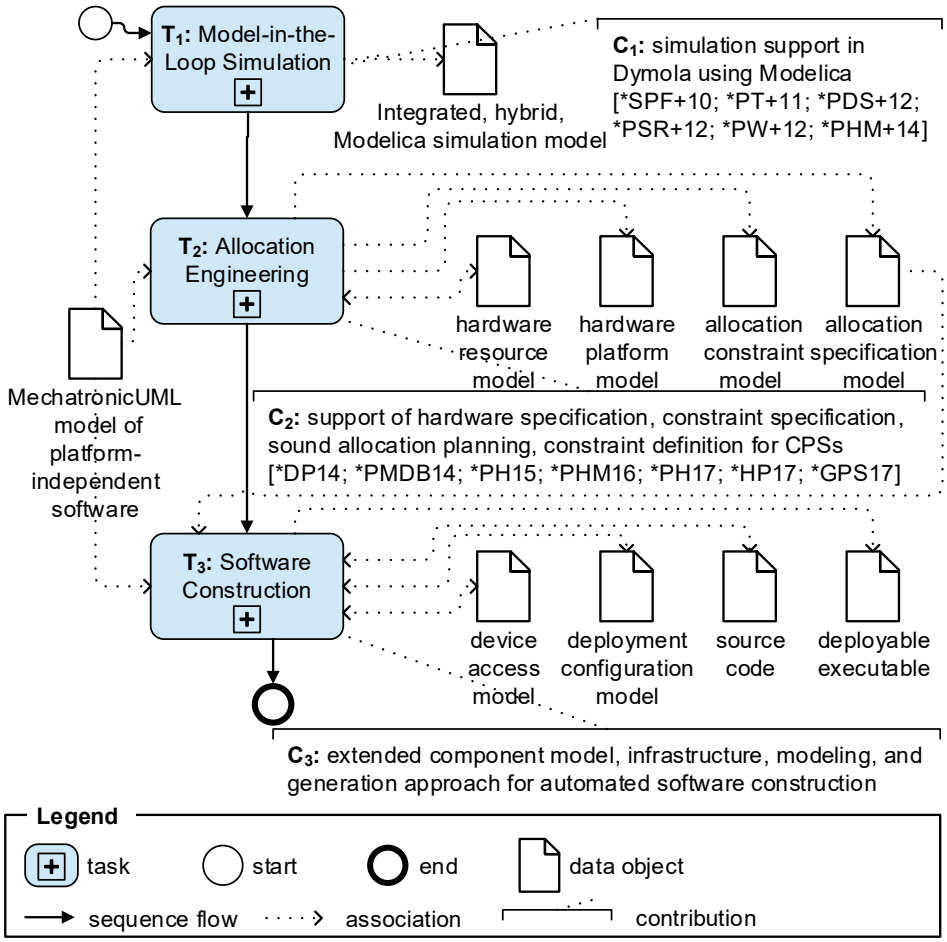


Figure 1.1: Overview of the Extended MechatronicUML Engineering Process

Figure 1.1 shows the development process that this thesis uses. The process requires as its input a verified platform-independent model of the software of a CPS specified using MechatronicUML. The development process for validating the software model in the context of its dynamic system and environment and for constructing the software for a distributed deployment on heterogeneous software and hardware platforms consists of the three software engineering tasks – **T1:** Model-in-the-Loop Simulation, **T2:** Allocation Engineering, and **T3:** Software Construction.

Early ideas of this thesis were presented at the international doctoral symposium on components and architecture [*Poh13]. Furthermore, the following theses of supervised students contribute concepts and implementation to this thesis [Wol11; Dan13; Hüw13; Gei15; Dan16].

1.2.1 Model-in-the-Loop Simulation

Firstly, we contribute (cf. **C1** in Figure 1.1) an approach for the *MiL simulation* [Plu06] to test the platform-independent, message-based, real-time communication in combination with a model of its dynamic system, and a model of its environment. Software engineers and system engineers can either use a provided Modelica library [*PDS+12] manually within their

domain tool or generate an *integrated, hybrid Modelica simulation model* from a platform-independent MechatronicUML model automatically to simulate the dynamic system's behavior [*PHM+14]. Furthermore, the concept is based on our previous findings and lessons learned by contributing different concepts for generating simulation models [*SPF+10; *PT+11; *PSR+12; *PW+12]. This thesis validates the applicability of the MiL simulation approach by conducting a case study on a automation case and on an automotive case.

1.2.2 Allocation Engineering

Secondly, we contribute (cf. $\mathbf{C_2}$ in Figure 1.1) an approach for the model-driven *allocation engineering* that fulfills safety-critical constraints to plan the intra-system allocation of software component instances to connected ECUs of a CPS automatically. Therefore, we contribute the support for defining a hardware specification, a constraint specification, a sound allocation planning, and predefined constraint definitions for CPSs. Software engineers and allocation engineers can use our approach to specify *hardware resource models*, *hardware platform models*, *allocation constraint models*, and *allocation specification models* [*DP14; *PMDB14; *PH15; *PHM16; *PH17]. The allocation specification model is calculated automatically by exploring the design space in a model-driven way [*PH15]. Thereby, engineers are able to compute feasible allocation specifications at design time that satisfy topology, software, and timing dependencies and memory, scheduling, and routing constraints. This thesis validates the applicability of the allocation engineering approach by conducting a case study on two automotive cases.

1.2.3 Software Construction

Thirdly, we contribute (cf. $\mathbf{C_3}$ in Figure 1.1) an extended component model, a component-container-based infrastructure, a platform-specific modeling, and a generation approach for the automated software construction. Software engineers can use the approach to specify *device access models* and *deployment configuration model* and to *generate source code* and *deployable executables*. This generated implementation considers software components' context requirements explicitly by using the existing functionality of software platforms and hardware resources. Thereby, engineers are able to build deployable software artifacts for distributed CPSs. This thesis validates the software construction approach by conducting a case study on an automotive case.

1.3 Running Example: Cooperative Overtaking System Using Car-2-X Communication

This section introduces the cooperative overtaking system [TSZ09] as the running example of this thesis. In the course of this thesis, we use the running example to demonstrate and evaluate our contributions.

One emerging trend in the automotive industry is autonomous driving. The motivation for autonomous driving is for example that emissions are reduced due to an optimized traffic flow by avoiding human errors, like unnecessary hard braking maneuvers. Cars, which have car-to-car and car-to-infrastructure communication capabilities (Car-2-X), can handle more advanced scenarios by cooperation than autonomous cars that cannot communicate [BKN+05; CGZC16]. For example, a leading car can send an obstacle warning message to the following car. The following car can reduce its speed smoothly. Today, networked navigation software already uses mobile communication data and movement profiles for reducing traffic jams. However, cars do not cooperate directly with each other. They only use globally available data

for guiding drivers. As a result, for example, every car gets an obstacle warning message for a certain time in a certain area, even if a car has already passed the obstacle.

Cooperative cars [BKN+05] are specified to enable a dialog between cars and to negotiate driving maneuvers. Drivers could be guided and assisted, or a driving maneuver could be performed fully autonomously. The software of a car has to read sensor values, set controller reference values, or actuator reference values for enabling this kind of advanced behavior. Furthermore, it has to perform the communication. As an example, Figure 1.2 shows an autonomous cooperating overtaking maneuver [TSZ09]. "Autonomous cars coordinate their overtaking behavior by means of an asynchronous message exchange protocol. The coordination addresses the safety-critical nature of overtaking maneuvers: it excludes a potential cause of collisions by ensuring that, whenever the rear car overtakes, the front car does not accelerate[, and no other car is approaching]" [*PHM+14].

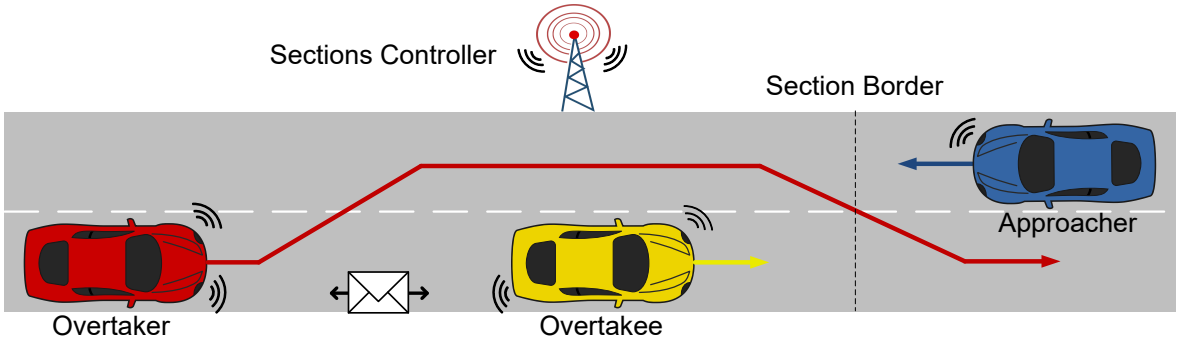


Figure 1.2: Autonomous Cooperating Overtaking Maneuver of Connected Cars

If the overtaker is close to the overtakee it asks if it agrees to be overtaken. In this case, the overtakee is not allowed to accelerate during the overtaking activity. The overtaker must read the values of its ultrasonic sensors for detecting the distance to the overtakee. It must exchange messages with the overtakee for overtaking and must set the overtaking speed value. Furthermore, the overtaker must ask the section controller if the left lane is free or if an approacher is close. Each car communicates with the section controller when entering or leaving a section. Thereby, a car can also be informed before entering a dangerous section, e.g., a scene of an accident within a section. The overall behavior of this scenario is complex. The complexity scales even more in realistic scenarios, where traffic signs, lane changes, emergency scenarios, and broken vehicles have to be considered.

1.4 Thesis Structure

Figure 1.3 shows the outline of this thesis. The next chapter describes the MechatronicUML development process and its modeling views as the foundation of this thesis. Afterward, Chapter 3 describes the foundation and the process of MiL simulation, our Real-Time Coordination Modelica library, and our approach for the automatic transformation of software architecture specified using MechatronicUML to Modelica. Following, Chapter 4 describes an extension of our running example and the process for the allocation engineering, our hardware platform modeling approach, our allocation constraint modeling approach, and our automated allocation planning approach. Then, Chapter 5 describes an extension of the MechatronicUML component model, and the process and concepts for the software construction. Chapter 3, Chapter 4, and Chapter 5 describe the contributions of this thesis. Each of these three chapters conclude with the same five sections: they explain the tooling

implementation, validate the approach by conducting a case study, describe the limitations of the approach, survey related work, and give a summary. The thesis concludes in Chapter 6 with a summary of the thesis and an outlook to possible future work.

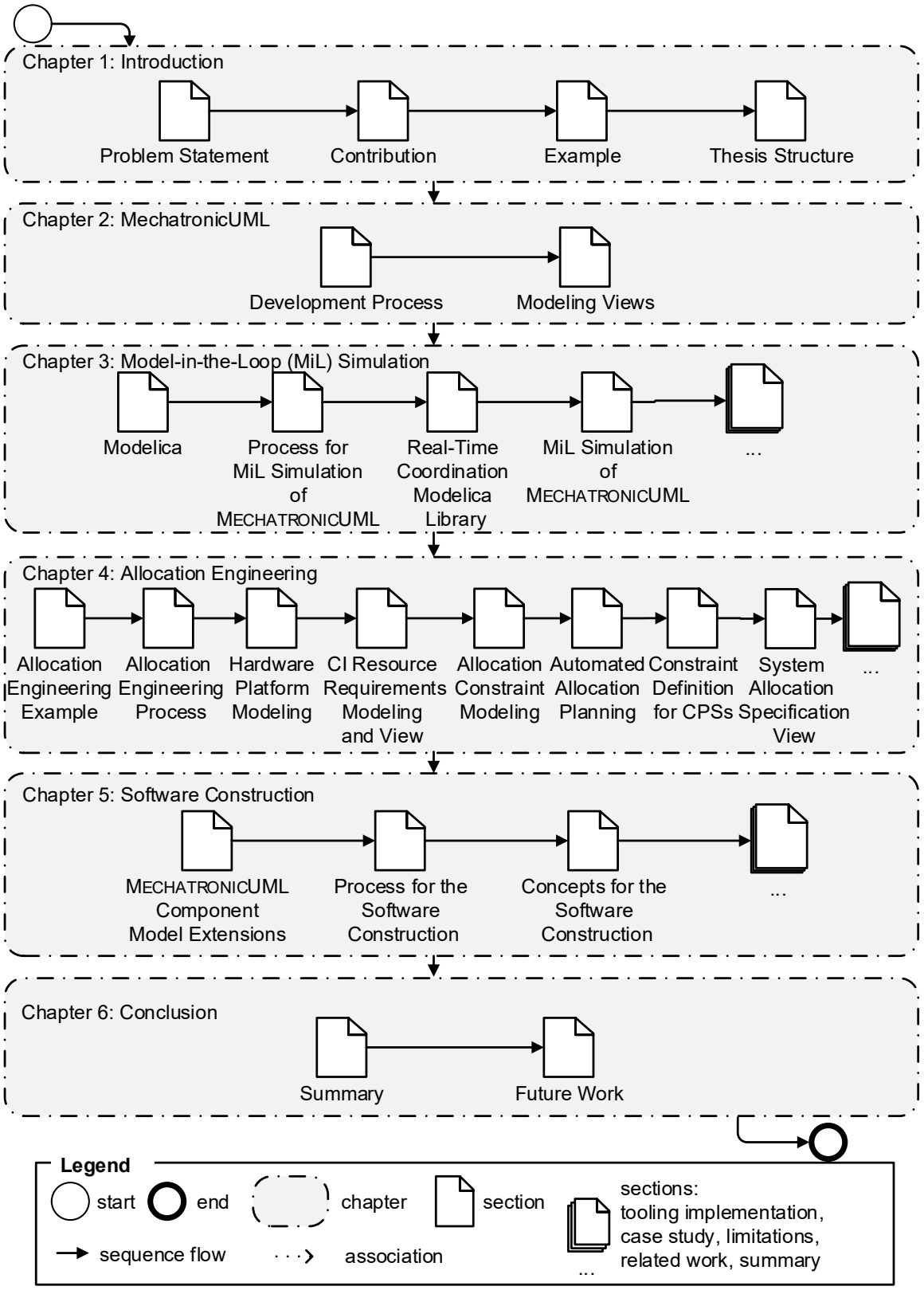


Figure 1.3: Thesis Structure

2

MechatronicUML

MechatronicUML [*BDG+14; *DPP+16] is a software engineering approach that treats the development of cooperative Cyber-Physical Systems (CPSs). Therefore, it focuses on the special characteristics of these systems. Firstly, software controls continuous physical processes, e.g., the acceleration and braking of a car. As a result, it relies on the laws of physics and real-time constraints that are coupled tightly to physical effects. Furthermore, the behavior of the software depends on measured and analyzed environment conditions and system conditions to control continuous physical processes. Secondly, cooperative CPSs involve several subsystems. The subsystems interact heavily with each other. They need asynchronous message-based communication to coordinate each other. The communication has to be integrated carefully with continuous system parts, like controllers, actuators, and sensors. Thirdly, the software runs on distributed embedded hardware with limited communication, memory, and processing resources.

2.1 Development Process

Figure 2.1 shows the MechatronicUML platform-independent developing process [HSST13] using the Business Process Model and Notation (BPMN) [BPMN]. The source of the process is a discipline-spanning conceptual design of the system under development. We usually use the domain-specific language CONSENS [DDGI14] to specify the conceptual design. In Task T_1, software engineers specify a component model based on the requirements, system structure, and information flow that is defined within the conceptual design. This component model defines all types of components, the hierarchical and horizontal composition of components, and the concrete component instance configuration. Software engineers can use the automated transformation that Rieke provides for generating from CONSENS, an initial component model that is specified using MechatronicUML [Rie14]. In the following Task T_2, software engineers specify the coordination protocol between different coordination roles. A coordination role represents a negotiating party. The coordination protocol defines a contract, including real-time behavior, for the message exchange between negotiation parties. Software engineers can choose coordination patterns, called Real-Time Coordination Pattern from a pattern catalog [DHT12; DBHT12; Dzi17] and refine them to coordination protocols. The coordination protocols must fulfill safety requirements and real-time requirements. Therefore, the coordination protocols are verified [GTB+03; GDHS15] using timed model checking via the UPPAAL model checker [UPP]. In Task T_3, software engineers specify the functional real-time behavior of the components. Firstly, they specify the internal component behavior. Secondly, they specify the coordination role that each discrete port represent. Thirdly, they integrate internal behavior and coordination behavior. Therefore, they refine the role coordination behavior to a port behavior [HH11]. The port behavior has to fulfill the coordination contract, which can be checked automatically [HBDS15]. The component model in combination with the component behavior makes up the MechatronicUML platform-independent software model.

This platform-independent model is the source for the subsequent task of analyzing the hybrid system's behavior via Model-in-the-Loop (MiL) simulation, the refinement to a platform-specific model, and finally the generation of executable platform-specific software artifacts.

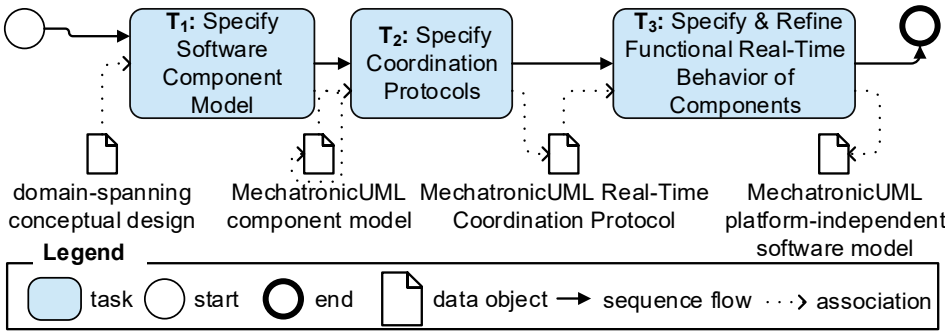


Figure 2.1: Overview of the MechatronicUML Platform-Independent Developing Process (Adapted from [HSST13])

2.2 Modeling Views

MechatronicUML provides four views [*DPP+16]. Furthermore, it provides an editor for each view [*DGB+14]. "Stakeholders need views for the modification of model elements [FPK+12]." [*PMDB14] "Viewpoints contain different view (types) for the stakeholders. 'A view type defines rules according to which views of the respective type are created. [GBB12]' A view is an instance of a view type. For simplification, we use only the term view in the following of this thesis.

Figure 2.2 shows the views Real-Time Coordination Protocol, Real-Time Statechart (RTSC), Component Diagram, and Component Instance Configuration Diagram. Furthermore, MechatronicUML offers views for defining reconfiguration [Hei15], which is beyond the scope of this thesis. This thesis focuses on static software.

2.2.1 Real-Time Coordination Protocol

In the first part, Figure 2.2 shows the Real-Time Coordination Protocol view for defining the communication contract between coordination roles. The protocol for coordinating an overtaking maneuver is called Overtaking Coordination, which is displayed within the dashed ellipse. The protocol describes the communication between the Overtaker role and the Overtakee role. Both roles can exchange messages with each other. A coordination role is strongly typed, e.g., it can only send and receive messages of types that are specified. The coordination role Overtakee has the sender message types accept and decline. Furthermore, it has the receiver message types request and finished. The receiver and sender message types of the coordination role Overtaker are vice versa. Each coordination role defines the size of the message buffer, the buffer storage strategy, and buffer replacement strategy for messages. The coordination roles Overtaker and Overtakee have a buffer size of 5 for each message type. MechatronicUML defines a static size to allow a static memory allocation. That means the buffer can store at most five messages of each receiver message type. In MechatronicUML, it is also allowed to specify different buffer sizes for different message types. Both roles store each received message type within an own First in, First Out (FIFO)-queue. Additionally, both roles discard an incoming message if the message queue of the corresponding message type is full. A connector

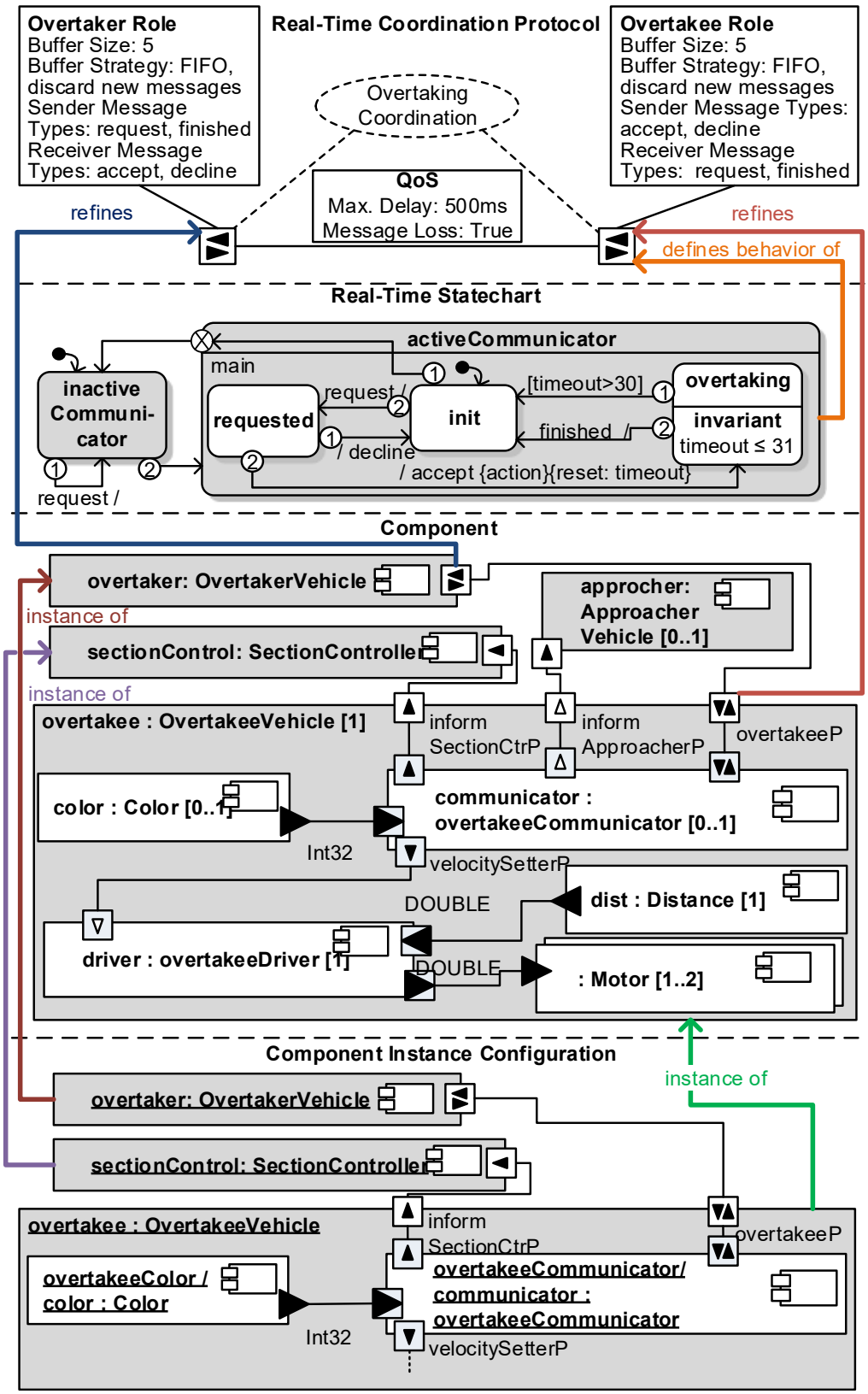


Figure 2.2: MechatronicUML: Formal & Domain-Specific Language for Hybrid Software of Cooperative Cyber-Physical Systems

links the coordination roles of a coordination protocol. The connector defines Quality of Service (QoS) assumptions that a system during runtime must fulfill to guarantee that verified safety and real-time requirements hold.

The connector between coordination roles rely to QoS assumptions. The sending message order has to be retained unchanged. The keyword Max. Delay refers to a static time expression. The delay defines the time between sending the message until it is allowed to be consumed. In this thesis, we refine the definition in comparison to the MechatronicUML specification [*DPP+16], which just requires that the message has to be stored within the buffer. According to Tindell the communication delay between sending a message and being able to access the message must be bound to guarantee that all timing requirements of all communication participants are met [TBW95] This refinement enables a static analysis of end-to-end deadlines. A deterministic real-time communication network protocol must ensure that the maximum delay is never violated. However, this requirement can be weakened by the QoS assumption that is indicated by the keyword Message Loss. The network protocol does not have to guarantee a reliable deliverable if this property is set to true. Messages that arrive too late or that are not consumed on time have to be discarded. In general, the coordination protocol must work safely even if messages get lost. If a message arrives in a wrong message order the late message is also discarded if a newer message is already consumed.

The connector of the coordination protocol Overtaking Coordination requires that a message that is sent via communication medium requires at most 500 ms until it is consumed. Furthermore, the connector allows that messages can get lost. That means the protocol is never in an unsafe state even if the communication medium is unreliable. The only property that must hold during runtime is that the message is either consumed before 500 ms are over or not at all. The verification guarantees that the protocol is safe if all QoS assumptions are fulfilled. Nevertheless, the scheduling policies have to be chosen correctly as Geismann et al. [Gei15; *GPS17] show. A detailed description of Real-Time Coordination Protocols is described in the MechatronicUML technical report [*DPP+16].

2.2.2 Real-Time Statechart

In the second part, Figure 2.2 shows the RTSC view for defining the behavior of coordination roles and components. The shown statechart is the behavior of the Overtaker coordination role. It consists of the simple state inactiveCommunicator and the composite state activeCommunicator. The state inactiveCommunicator reflects the status when the Overtakee does not change its state as a result of any received message. It has a self-transition that consumes incoming request messages. At any time, the statechart can activate the communication behavior by switching to the composite state activeCommunicator. A composite state can contain regions, which are orthogonal statecharts. The state activeCommunicator contains only the region and statechart main. This statechart consists of the three simple states requested, init, and overtaking. The initial state init reflects the status when a car drives in a normal mode and does not want to overtake any other car. At any time, it can receive the message request from the Overtaker role and switch from state init to requested. In the state requested it can either decide to respond by a decline or an accept message. Messages can contain message parameter to send or receive further information. If the Overtakee responses with a decline message it switches back to the state init. If the Overtakee responses with an accept message it switches to the state overtaking. When this transition fires a transition event is triggered and executes an action. An action is an algorithmic execution of some instructions (cf. Section 2.2.3). Furthermore, the transition event triggers that the clock timeout is reset to the value zero. In this thesis, we omit to calculate a scheduling for the execution of MechatronicUML components and its behavior. As a result, we assume that each statechart is scheduled, evaluated, and executed if a transition

is able to fire and that the firing of transitions and the execution of all actions require a negligible amount of time. Thereby, we also do not require to distinguish between urgent transitions where no time passes between enabling and firing a transition and non-urgent transitions where may time pass between both events, e.g., due to a scheduling delay. The target state overtaking reflects the status when an overtaking maneuver can take place and the Overtakee should not accelerate. The state overtaking is allowed to be active as long as the value of the clock timeout is smaller or equal to the value 31. This is specified by using a time-invariant. The state overtaking has two outgoing transitions. If both transitions could fire the transition with the highest priority is chosen (indicated by the number within the circle). The transition with the trigger message finished has a higher priority than the transition with the clock constraint [timeout $>$ 30]. If one of both transitions fire the statechart switches from the state overtaking to the initial state init. If this state is active the transition to the exit point (crossed circle) could fire. As a result, the state init and the state activeCommunicator are deactivated and the state inactiveCommunicator is activated. A state can contain entry events or exit events that could trigger actions. A detailed description of RTSCs and all its features is described in the MechatronicUML technical report [*DPP+16].

"In order to formally verify that the coordination behavior excludes deadlocks and unsafe state combinations, we use the timed model checker UPPAAL. UPPAAL performs a formal verification of safety requirements on timed automata. We refer to [CGP00] for a detailed presentation of model checking principles. As a result of the transformation to UPPAAL, we can prove the absence of violated safety requirements under the assumption that the continuous behavior in interplay with the physics preserves the timed state-based behavior. We transform MechatronicUML RTSCs to UPPAAL to verify specified safety requirements [Ger13; GDHS15]. (...) The safety requirements are specified by a dialect of TCTL. If a violation of a safety requirement is detected, UPPAAL [[UPP]] provides a corresponding counterexample." [*PHM+14]

2.2.3 Action Language

In the second part, Figure 2.2 shows an action within the RTSC at the transition from the state requested to the state overtaking. The action is specified by a simple textual imperative programming language called MechatronicUML action language [*DPP+16]. The MechatronicUML action language is inspired by the UML action language ALF [ALF] that is usable for UML-based models. MechatronicUML uses the action language to formally define instructions that are executed based on entry events, exit events, and transition events. Furthermore, transition guards that restrict the firing of transitions are specified using the action language. The action language provides blocks, local variable declarations and initializations, operation calls and implementation, assignments, type castings, loops, and if-statements. Furthermore, it provides assignment, logical, comparing, and negation operators within expressions. The MechatronicUML specification [*DPP+16] gives a detailed explanation of the MechatronicUML action language capabilities.

Listing 2.1 shows an example implementation of the named transition action using the MechatronicUML action language. Firstly, a Block expression is used to group the following expressions. Afterward, the value of the message parameter frontVelocity of the message accept is bound to the local variable velocity. Thereafter, the variable overtakingSpeed of the Real-Time Statechart is set either to the value of the local variable velocity plus 10 or to the value velocity plus 15 depending on the value of the local variable velocity.

```
1 int32 velocity := accept.frontVelocity;
  if(velocity <= 50)   {overtakingSpeed := velocity+10;}
  else  {overtakingSpeed := velocity+15;}
```

Listing 2.1: MechatronicUML Action Language Example

2.2.4 Component Model

In the third part, Figure 2.2 shows the Component Model view for defining the component model, which includes the software architecture definition. The shown component diagram shows the software architecture for the cooperative overtaking scenario. MechatronicUML defines a component as a self-contained, active, functional software artifact with clearly-defined interfaces to other platform-independent software components. The MechatronicUML design method [*DPP+16] only considers the platform-independent modeling. Therefore, the MechatronicUML component model does not state any component's context dependencies with respect to platform-specific (technical) concerns. MechatronicUML differs between atomic and structured components. Atomic components are either of the kind discrete or of the kind continuous. The behavior of discrete components is defined by using RTSCs. The behavior of continuous components is defined by domain specific approaches, like MATLAB Simulink or Modelica. "(...)[T]he MechatronicUML component model syntactically decouples the event-discrete part of the behavior specification from the time-continuous part. Discrete components may only interact with continuous components based on hybrid ports whose values are only updated in fixed, predefined time intervals. We do not apply any assumptions on how the values of a hybrid port change and we do not allow, in particular, to include time-continuous variables in RTSCs." [Hei15] A structured component is composed of other components. MechatronicUML allows hierarchical and horizontal component composition. Furthermore, each part of a component has a cardinality, which specifies how many instances can occur during runtime.

The component diagram consists of the structured component overtaker of type OvertakerVehicle, the structured component sectionControl of type sectionController, the structured component overtakee of type OvertakeeVehicle, and the structured component approacher of type ApproacherVehicle. We distinguish the different types of vehicles according to the coordination role for simplicity. In reality, a vehicle has to represent all role types. Furthermore, we do not show the component parts of the components overtaker, sectionControl, and approacher. We further define that all component types occur at most once by the cardinality [1] and that an approacher is not always required, which is indicated by the cardinality [0..1]. Message ports are displayed as rectangles that contain small triangles that represent if the ports can send or receive messages. Ports with triangles that are filled are mandatory and port with triangles that are non-filled are optional. Signal ports that represent the communication with sensors, actuators, or controllers are displayed as triangles and hybrid ports that discretize continuous signals according to a period are displayed as a rectangle with a bigger triangle. The orientation of the triangle shows if the signal is an input or output signal. The structured component overtakee consists of three continuous component parts: the continuous component part color of type Color with cardinality [0..1], the continuous component part dist of type Distance with cardinality [1], and the continuous component parts of type Motor with cardinality [1..2]. Internally, the Motor components are realized as a continuous feedback controller. Furthermore, overtakee consists of two discrete component parts: the discrete component part driver of type overtakeeDriver with cardinality [1] and the discrete component part communicator of type overtakeeCommunicator with the cardinality [0..1]. Ports that are linked with each other via a connector can communicate. The behavior of the discrete port overtakeeP of

the component communicator refines the behavior of the Overtakee role. Furthermore, the discrete port of the component overtaker refines the behavior of the Overtaker role. A detailed description of the MechatronicUML component model and all its features is described in the MechatronicUML technical report [*DPP+16].

2.2.5 Component Instance Configuration

In the fourth part, Figure 2.2 shows the Component Instance Configuration view for defining the component instance model. The component instance configuration diagram shows an excerpt of a concrete instantiation of the cooperative overtaking scenario without an approacher. The instance configuration consists of the three component instances overtaker, sectionControl, and overtakee. The type of all component instances is defined within the component model. The component instance overtakee instantiates all parts with the upper bound of cardinality. A detailed description of the MechatronicUML component instance configuration and all its features is described in the MechatronicUML technical report [*DPP+16].

3

Model-in-the-Loop Simulation

Cyber-Physical Systems (CPSs) are called hybrid systems [LZ07] because they consist of event-discrete parts from the computational world and of time-continuous parts from the physical world. We describe the software architecture of these systems with the MechatronicUML component model and a concrete manifestation with the MechatronicUML component instance model. The component model supports to specify and connect discrete components that define state-based event-discrete behavior implemented as Real-Time Statecharts (RTSCs). Furthermore, it supports to specify and connect continuous components that represent parts that interact with the physical world, like sensors, actuators, and feedback controllers.

The software of CPSs has to fulfill high-quality safety requirements [ISO26262-6; ISO25010; IEC61508]. The reliability of each system unit and the reliability of the integration of discrete, state-based software, time-continuous controller software, physical properties, and the environment contribute to the overall reliability of the software of CPSs. During runtime on a concrete Electronic Control Unit (ECU), further factors, like synchronized clocks or the availability of resources, have to be considered. In this chapter, we focus on the logical conformance of the system units and its integration to the required functionality. Therefore, the conformance of the overtaking with approacher scenario depends on these reliability properties (cf. Section 1.3). An error in the interaction between different cooperating cars can lead to a crash between the overtaking car and the approaching car. The overtaking with approacher scenario contains a so-called hybrid automaton. The continuous feedback controller of the motor components behaves differently depending on its reference values that are defined by the discrete state-based driver component. As a result, a hybrid model checking problem has to be solved to formally verify the conformance of the system's behavior to safety and liveness properties [Hen00]. Hybrid model checkers are able to check only small models. The hybrid model checker SpaceEx can check, e.g., models with up to 200 variables [RLW+14]. Furthermore, many model checkers can verify only hybrid systems that contain only linear equations, like HyTech [HHW97]. As a result of these limitations, hybrid model checkers are not usable for normal-sized realistic examples, like a car-following model [AL98]. The simulation of CPSs is able to validate the overall, integrated system's behavior. Therefore, software engineers and system engineers require an integrated model of event-discrete software parts, time-continuous software parts, physical system parts, and the environment to validate the correct integration [ÅK14]. Unfortunately, a simulation run considers only individual execution paths.

A testbed of the whole integrated system can be used to validate the reliability of the system's behavior. However, a testbed requires ECUs, sensors, actuators, and physical parts. These parts are not always available in the design phase of a project. Debugging the system's behavior if the parts are available is also complex because the root cause of the bug can be diverse, e.g., the software design, the integration of different parts, or even the electrical wiring. Moreover, in industrial automation and production systems engineers are often not able to validate the software until putting the overall system into operation at the customer's facility.

It could lead to high costs to identify and correct violated (extra-) functional requirements when already the application software design is full of flaws [SDD+04; SBB+02]. In the worst-case, a project fails.

Software engineers can build models, like an automaton of the system's state space, to verify the correctness of the discrete software. Furthermore, control engineers or system engineers can build models, like nonlinear differential equations, to analyze the dynamic behavior of a physical system [ÅM08]. An approach for testing a CPS in the context of a model of its environment is Model-in-the-Loop (MiL) simulation [Plu06]. Commercial Off-The-Shelf (COTS) tools like MATLAB Simulink [Sim] or Dymola [Dym] support MiL simulation but do not have native support for asynchronous, message-based communication with buffers [Hei15]. In addition, the available models of different systems parts, like continuous feedback controllers and timed automata, unfortunately are specified in different formalisms. "This development process hinders early (...)[validation] of system models to detect errors in the design phase as early as possible and to avoid costly error removal in later development stages." [*PSR+12] Table 3.2 in Section 3.9 summarizes related work and their capabilities.

MechatronicUML was designed from the beginning with the correctness-by-construction paradigm [Cha06] in mind and provides a component model that integrates event-discrete, state-based software components and time-continuous, control software components [BGO06]. The behavior of the discrete components is specified by using a so called Real-Time Statechart, which is a combination of UML state machines [UML] and timed automata [AD94]. Its semantics is defined formally [Hei15] and by a model transformation to timed automata [GDHS15]. In contrast to UML and timed automata it provides first level modeling elements for asynchronous, message-based communication. The component model of MechatronicUML provides interfaces between discrete components and continuous components. The behavior of continuous components has to be defined within control-engineering or system-engineering tools like MATLAB Simulink or Dymola.

We use MiL simulation to validate the correctness of the integrated behavior specified by using the hybrid MechatronicUML component model. Therefore, we need a software engineering approach that allows to use MiL simulation of a model that supports state-based behavior, asynchronous, message-based communication, and nonlinear differential equations. Furthermore, the approach has to built up on the existing MechatronicUML component-based development approach to preserve the ability to verify the communication behavior formally by using timed model checking. The simulation model should be built up on an open standard. Furthermore, control engineers and system engineers should be able to build up an integrated simulation model within their simulation tool, e.g., by using predefined modeling elements for asynchronous communication from a library. The solution should provide a combination of formal verification and simulation-based testing. We focus on Modelica 3.2 [MODELICA] and the tool Dymola [Dym] because Modelica supports hybrid Differential Algebraic Equations (DAEs) as its mathematical framework. DAEs have advantages to Ordinary Differential Equations (ODEs), which MATLAB Simulink uses, for specifying and simulating physical models [Lee10]. Furthermore, Modelica has a strong industry- and academy-driven open community that is not owned by a single vendor. We use the MechatronicUML platform-independent design models, where the correctness of real-time and safety properties of the discrete parts are formally verified [Hei15; GDHS15]. Thereby, engineers can focus within MiL simulation runs on testing the integration of continuous and discrete components. Nevertheless, the approach should also enable to use Modelica directly to model state-based, real-time coordination behavior without depending on MechatronicUML and the MechatronicUML tool suite. Thereby, we allow engineers to use the benefits of the MechatronicUML approach but also to use Modelica directly without the additional hurdle to learn a new language.

In the following, we summarize our requirements. We develop the requirements in the context of the goals of this thesis. Therefore, we do not claim that the list is complete in general. Our requirements on the basis of [Wol11; Hei15; Dzi17] are as follows. A software design approach for CPSs approach should ...

- R.3.1 be able to specify the system architecture with hierarchical nesting and multiple communication styles.
- R.3.2 be able to specify the state-based software behavior.
- R.3.3 be able to specify the clock-based real-time behavior.
- R.3.4 be able to specify the message exchange and buffering.
- R.3.5 support MiL software simulation in combination with continuous system parts.
- R.3.6 support formal verification of discrete behavior.

In the following, we explain and justify our requirements for the software design of CPSs. We require that the approach is able to model the architecture to represent the system's structure. The system's structure of a CPS is assembled of components or blocks respectively [SysML]. Additionally, the architecture should, according to Pop *et al.* [PHH+14], support hierarchical nesting and different communication styles. Without fulfilling requirement R.3.1, a system design approach cannot be used to develop efficiently. Concerning requirement R.3.2, we require the support of modeling the state-based system's behavior, i.e., the behavior that concerns the discrete states of the system itself and for cooperation, the state of coordination protocols between system parts and independently operated CPSs [*HSD15]. The discrete states are important for defining verifiable safety properties [GDHS15; Dzi17]. Clocks are the basis for specifying the real-time behavior by timed automata and to perform timed model checking [AD94; BY04]. Clocks can be reset but not altered to arbitrary values. The requirement R.3.2 necessitates that it is possible to define clocks and to specify the state-based real-time behavior dependent on the values of the clocks. Besides the internal behavior of the system itself, the coordination behavior is essential for the correctness of the system's behavior [Dzi17]. Therefore, requirement R.3.4 enforces that a design approach should support to specify the asynchronous (state-based) message exchange between CPSs and the buffering of messages [Dzi17]. Requirement R.3.5 is based on the motivation of this chapter that timed model checking is necessary but not sufficient to validate the overall system's behavior in interaction with physical boundary conditions and its environment [Lee08]. Therefore, we need an overall MiL simulation that combines and validates the system's behavior that results from the combination of the discrete and the continuous system's behavior. Furthermore, we need the support of formal verification of discrete behavior as defined by the requirement R.3.6 because MiL simulation only tests certain scenarios but cannot ensure that the discrete software specification holds the safety properties [Hei15]. Nevertheless, fulfilling the safety properties is essential to operate a CPS reliable.

Firstly, the contribution of this chapter is a Modelica library that supports and eases the modeling of message-based, real-time communication within Modelica tools, like Dymola [Dym] natively. Secondly, this chapter contributes an approach for performing MiL simulation of CPSs that are specified and verified using MechatronicUML. Therefore, this chapter contributes "(...)an automatic transformation of a formally verified software specification to Modelica involving existing Modelica libraries."[*PHM+14] The MechatronicUML model contains a definition of the software architecture as a component model and contains a concrete component manifestation as a Component Instance Configuration (CIC). The behavior of discrete components is specified using RTSCs. The model fulfills the abstract syntax and static semantics definition of MechatronicUML [*DPP+16].

Our approach solves four challenges and fulfills all stated requirements. Firstly, Modelica supports only communication in the form of data exchange via signals. Natively, events cannot be exchanged between different model elements. Therefore, we provide Modelica classes that support synchronous and asynchronous communication. The communication itself is realized via Modelica signals but the provided Modelica classes contain logic that emulate the behavior of the required synchronous and asynchronous communication. Secondly, the Modelica solution StateGraph2 [OME+09] for modeling the state-based behavior has only limited support to specify timing behavior. We provide Modelica classes that enable to specify timing behavior using clocks, like timed automata [AD94]. We evaluate that the communication elements and the timing elements, which we define, are sufficient to model coordination behavior by performing a case study. Thirdly, system engineers who perform MiL simulation runs should be able to inspect the simulation run using a single language and tooling like Dymola. Systems engineers only have to learn one language and have to operate only one tool, which eases to inspect the simulation runs. Therefore, our approach generates pure Modelica code. In contrast to other approaches, like Tripakis and Broman [TB14] and Theorin and Johnßon [TJ14], we do not need to perform co-simulations. A co-simulation involves the interaction and synchronization of different simulation tools and simulation runs. In contrast to other approaches that use the external C code interface of Modelica, like Sanz, Urquia, and Dormido [SUD08], our approach enables to inspect the behavior of message queues during a simulation run. Fourthly, simulation models should preserve properties that are verified in an earlier process task. Our transformation needs to preserve the semantics of MechatronicUML. Therefore, we provide a formal syntax and semantics definition of the Modelica library that we define for representing MechatronicUML concepts in Modelica. The semantics definition origins from the MechatronicUML semantics. Furthermore, we provide an information description of how we preserve the MechatronicUML semantics and tested it using our implementation. A formal proof that the transformation preserves the MechatronicUML semantics has to be done in the future. We evaluate that our transformation covers required MechatronicUML constructs for modeling cooperative real-time scenarios and preserves the semantics of MechatronicUML by performing two case studies.

We split this chapter into six sections. In the following, Section 3.1 introduces Modelica, shows how to perform MiL simulation of Modelica models using Dymola, and describes its basic concepts to model the system's structure and the state-based behavior. Afterward, Section 3.2 describes the overall process and required toolchain for MiL simulation of MechatronicUML. Following this, Section 3.3 introduces and evaluates the key artifact in the process, our *Real-Time Coordination* Modelica library. The library enables to model coordination behavior in the form of state-based real-time communication directly within Modelica. Subsequently, Section 3.4 shows an automatic transformation of models specified using MechatronicUML to models specified using Modelica. Engineers use the automatically created Modelica models to validate the integrated behavior via a MiL simulation. The transformation uses elements of the *Real-Time Coordination* Modelica library as target artifacts. Thereafter, Section 3.5 describes our implementation of the transformation. Subsequently, Section 3.6 describes the evaluation and Section 3.7 summarizes the current limitations of the approach. The chapter ends with a discussion of related work in Section 3.8 and a summary in Section 3.9.

The content that this chapter presents has been published in [*PT+11], [*PDS+12], [*PW+12], [*PSR+12], [*PDM+14], [*PHM+14], [*PTD+14], and [*GST14]. Concepts for the translation of MechatronicUML models to Modelica and the development of the foundations of the corresponding Real-Time Coordination Modelica library have been contributed by the Master's Thesis of Wolf [Wol11].

3.1 Modelica

Modelica is a non-proprietary textual modeling language for CPSs [MODELICA]. The Modelica Association, which is a non-profit, non-governmental organization, has been developing the language since 1996. The language focuses on the physical nature of CPSs but is not limited to it. It uses an object-oriented programming paradigm for structuring models and offers to model the system's behavior in the form of declarative equations or imperative algorithmic statements. Modelica does not offer to invoke operations on a specific class or an object. It only allows to specify functions, which are static and side-effect free.

Hybrid DAEs serve as the mathematical framework behind the language [ÅK14]. Hybrid DAEs include ODEs, algebraic equations, and conditional equations. ODEs are mathematical models of continuous dynamic systems that depend on time derivatives of state variables. Algebraic equations describe relations between variables that do not involve derivatives of variables. Conditional equations are equations that become active at certain points in time or during a specific time interval. [Fri14]

Furthermore, Modelica offers to specify annotations. These annotations are structured comments that store additional information, like the graphical representation of a class or an object. The graphical representation are used to define so-called Modelica connection diagrams that eases engineers to grasp the structure of the modeled systems. Modelica's modeling and simulation environments use annotations for documentation purposes and for graphical model editing. [Fri14]

3.1.1 Modeling System's Structures

Modelica enables to split a system into subsystems and to define interfaces by instantiating connectors for each subsystem. The subsystems are connected by so-called *constitutive equations* or *connect equations* [MEO98]. Modelica enables to define templates in the form of classes for creating subsystems, by using the object-oriented programming paradigm. A Modelica model is an object diagram, which is called connection diagram. It consists of objects and connections between object-connectors. Objects are also called components in Modelica. Engineers can set values of class parameters while instantiating objects. The Modelica Association distributes model libraries, which enable the reuse and the combination of existing models. Engineers who use libraries, accelerate their development time and are able to integrate different model parts from different engineering domains easily. "The open source Modelica standard library contains about 1000 model components and more than 500 functions from many domains." [ÅK14] Furthermore, engineers can create and distribute their own libraries.

In this paragraph, we show an example of a two-wheel robot model specified using Modelica. Each wheel of the robot has an own motor that can be controlled individually. The robot, its rolling wheels, and the three-dimensional position should be visualized during a simulation run. Figure 3.1 shows in the upper part a simple connection diagram, which consists of the two objects constLeft and constRight that provide constant values and the object Robot, which represents the physical parts of a two-wheel robot. The Modelica standard library provides the class Constant as the type of the objects constLeft and constRight within the package Modelica.Blocks.Sources. The parameter k of both constant objects is set to 4. The output connector of the object constLeft is connected to the omegaL input connector of the Robot object and the output connector of the object constRight is connected to the input connector omegaR respectively. Thus, both wheels of the robot get the reference speed 4.

Figure 3.1 shows in the lower part the physical subsystem of the Robot object. The casing and the electrical circuit boards are abstracted to a rigid Body object in Modelica. The Modelica standard library provides the class Body within the package

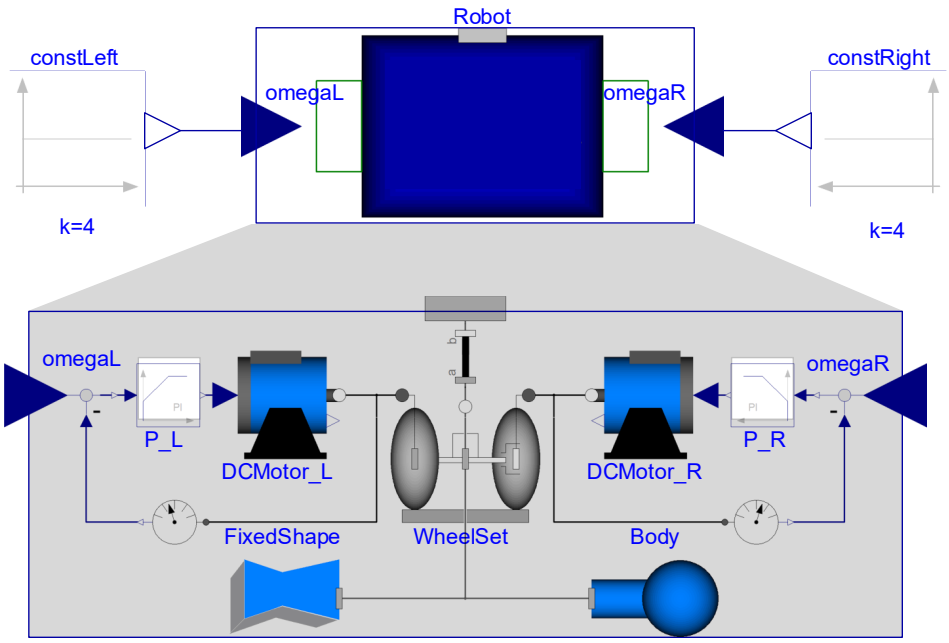


Figure 3.1: Modelica Connection Diagram of a Two-Wheel Robot

Modelica.Mechanics.MultiBody.Parts. An object of the class FixedShape from the package Modelica.Mechanics.MultiBody.Visualizers.FixedShape visualizes a Body as an animation during a simulation run. The pair of wheels of the robot is modeled by an object with the type WheelSet from the package Modelica.Mechanics.MultiBody.Parts.RollingWheelSet. The wheels share a common axle, whereby each wheel is individually controlled by a DC motor. The motors are represented by the objects DCMotor_L and DCMotor_R. The feedback controllers P_L and P_R control the motors and get their reference values from the corresponding input connectors omegaL and omegaR. The classes of the motor objects and controller objects consist of further sub-parts that represent electrical, mechanical, and control behavior.

Listing 3.1 shows the behavior of the feedback controllers that are specified by using Modelica equations. The behavior represents a PI feedback controller with a proportional gain and an integrator gain. Modelica equations have a declarative mathematical semantics, i.e., they connect the involved variables logically and are valid also for mathematical transformations of the equations. Therefore, it would be also valid to write $y = kp * u + yi;$. Also the order in which equations are stated do not influence its semantics.

In contrast to equations Modelica also offers to specify algorithmic code by using the keyword algorithm instead of equation. In this case, the semantics of the shown statements is imperative, like in the programming languages C or Java. That means, the order of statements within algorithmic code matters. Furthermore, in contrast to an *if-clause* that is always evaluated as long as the condition is true Modelica also offers a *when-clause* that is only evaluated once when the condition switches from false to true (rising edge).

3.1.2 Modeling the State-based Behavior

StateGraph2 is a Modelica library for the state-based modeling [OME+09]. It provides the basic elements for modeling statecharts in Modelica. The StateGraph2 library provides the three main classes step, transition, and parallel. Otter *et al.* stated the complete formal definition

```
/*
variable u is controller input
variable y is controller output
parameter outMin is lower limit of output
parameter outMax is upper limit of output
parameter kp is proportional gain of the controller
parameter ki is integrator gain of the controller
parameter y_start=0  is initial of the output;
*/
initial equation
   if initType == Init.SteadyState then
      der(yi) = 0;
   elseif initType == Init.InitialState or
     initType == Init.InitialOutput then
      yi = y_start;
end if;
equation
   yk = kp*u;
   y = yk +yi;
   der(yi) = if (y<outMin and u<0) or (y>outMax and u>0) then 0 else
      ki*u;
```

Listing 3.1: Behavior Equations of the PI Feedback Controllers

of a StateGraph2 model in [OME+09]. We extend the StateGraph2 library in this thesis to map RTSCs of MechatronicUML to Modelica in Section 3.3.

Instantiating the class step as an object represents a discrete system state. Instantiating the class parallel as an object represents a composite state of hierarchical statecharts that can embed other step objects, so called subgraphs. Step objects and parallel objects can be marked as an initial state, by setting their BOOLEAN initialStep parameter to true. A parallel object has an entry-connector that can be connected to an inPort of an embedded step object to activate it when the parallel object is active. Furthermore, parallel objects can be suspended, resumed, and deactivated by corresponding ports. Any kind of state (step,parallel) is called according to [OME+09] a generalized step. A generalized step is described by the 5-tuple $< I, R, O, S, \Gamma_s >$ where I is a vector of inPorts, R a vector of resume port, O a vector of outPorts, S a vector of suspend ports, and Γ_s a set of subgraphs. Triggering any inPort activates the corresponding step object and its subgraphs. Triggering all outPorts together deactivates a step and all its subgraphs. Triggering any suspend port deactivates a step and all its subgraphs and stores the last state configuration. Triggering any resume port activates a step and all its subgraphs to the last known state configuration. A known state configuration is deleted if any *inPort* is triggered. Engineers can model a transition of one discrete system state to another discrete system state by instantiating another step object and a transition object. Furthermore, the outPort of the first step object has to be connected to the inPort of a transition object and the outPort of the transition object has to be connected to the inPort of the second step object. A transition object has several condition parameters, which define when a transition is able to fire. [OME+09]

Figure 3.2 shows a simple StateGraph2 connection diagram that is part of the class SpeedMonitor, defined in Modelica. For simplicity, we call step objects in the following *states* and we call parallel objects in the following *composite states*. The diagram defines the behavior for detecting, whether a car drives slow or fast, e.g., to decide if it is allowed to be overtaken. The diagram contains the composite state main. This state is marked as the initial state. The composite state main state embeds the states slow and fast. The state slow gets active when the state main gets active. The transition t2 from the state slow to the state fast fires if the guard

condition expression *speed* >= 3 evaluates to true. As a result, the state slow is deactivated and the state fast is activated. The speed value corresponds to the input port speed. When the state fast is active, the output port fast gets the value true because the activePort of the state fast is connected to the port fast. The transition t1 from the state fast to the state slow fires if the guard condition expression *speed* < 3 evaluates to true. As a result, the state fast is deactivated, the port fast gets the value false, and the state slow is activated again. The current discrete state of the SpeedMonitor depends on the continuous system state, which is represented by the current speed of the system. Therefore, SpeedMonitor is a hybrid system.

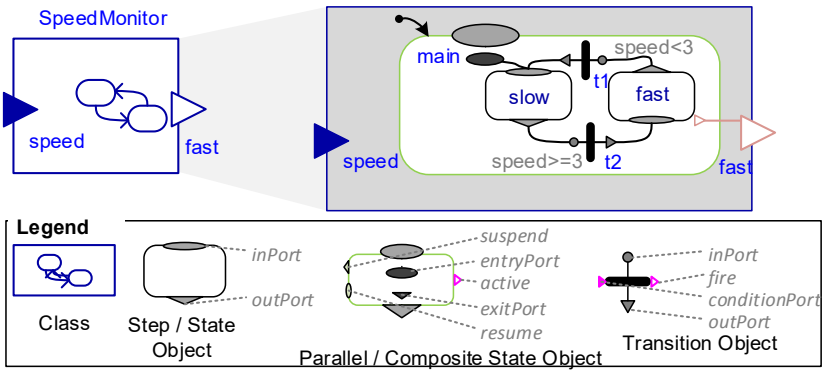


Figure 3.2: SpeedMonitor Class with the State-based Behavior

The StateGraph2 Modelica library is the first choice for modeling the state-based behavior manually when using Dymola [Dym]. An alternative is encoding the state-based behavior within a single class as Modelica algorithmic code. Modelica algorithmic code has an imperative semantics. The algorithmic code must encode the state-based behavior using if-else statements that differentiate between states and performs the state changes. Such an encoding is a very complex manual task when not using a code generator [*PT+11; *SPF+10]. However, the StateGraph2 Modelica library is not sufficient for modeling real-time coordination scenarios, like the cooperating overtaking maneuver (cf. Section 1.3). Especially, the library does not support all required concepts for translating RTSCs of MechatronicUML to library elements. The library has the following limitations: "Statecharts are used to describe the behavior of reactive systems. The reactions of such systems are based on their current internal state and the external input. Formalisms for Mealy machines, Harel's statecharts [Har87], and most common automata-based formalisms support events that can be used for a message-based communication. However, StateGraph2 does not have syntactical constructs. Different steps or transitions can only communicate via shared variables. In real systems, this is not possible when the systems are distributed and have no access to shared memory. The need of shared memory makes it difficult to reuse components as they depend on their environment and not only on their interface description. Therefore, a message-based mechanism is very important. This may be either an asynchronous or a synchronous communication." [*PDS+12]

"StateGraph2 has only a limited support to specify timing behavior. Only the execution of transitions can be delayed. The variable waitTime of a transition specifies the time a transition waits before it fires when its guard evaluates to true. If during the waiting period the guard evaluates back to false, the transition does not fire. Therefore, the construct delayedTransition of StateGraph2 can be misinterpreted because the semantics includes more than a simple delay. In contrast to StateGraph2, timed automata [AD94] use clocks to store time independently of a concrete state. Clocks can be read and reset in any state and upon firing of a transition.

Therefore, this concept is more flexible for specifying timing behavior. To conclude, the variable waitTime alone is too limited to describe real-time behavior." [*PDS+12]

3.1.3 MiL Simulation of Modelica Models

Engineers can test Modelica models of CPSs by a MiL simulation [Plu06]. "Simulation is useful because it allows exploration of the behavior of complex systems in a safe setting." [ÅK14] Numerical solving of DAEs enables to simulate the integrated behavior [ÅK14]. Various commercial and free simulation environments are available for Modelica. Dymola [Dym] is one of the major simulation environments for Modelica, which has an industrial acceptance [ÅK14].

Dymola performs several tasks to generate an executable simulation model from a Modelica model. Firstly, it transforms the Modelica model to a flat set of equations, constants, variables, and function definitions. Secondly, the equations are analyzed and sorted according to their data-flow dependencies. Thirdly, an optimizer tries to simplify the equation set. Fourthly, execution code, like C, is generated and compiled to an executable that is linked to a numeric solver. The solver computes during an execution, which represents a simulation run, values of the variables for specific simulation intervals, e.g., [t, t+1]. The length of a simulation interval ([t, t+1]) is called step size. Fixed-step solvers use a fixed step size during the simulation that the user chooses or the solver chooses itself at the beginning of the simulation. Variable-step solvers vary the step size during a simulation run depending on the needed accuracy, caused by the model changes. Varying the step size adds computational overhead to each step, but can reduce the overall computation time by avoiding unnecessary steps. [Fri14]

Figure 3.3 shows the result of a simulation run. In the left part, it shows a plot that displays the speed of a front and a rear robot. The speed of the front robot is constant, whereas the speed of the rear vehicle varies and it adapts to the speed of the front robot if it is too close. Furthermore, the plot displays the output variable fast of the rear robot that depends on its current speed. Engineers may correct their design of the state-based behavior of the SpeedMonitor after considering the result of the simulation run because it does not contain a stop state. In the right part, the figure shows snapshots of the 3D view during the simulation run.

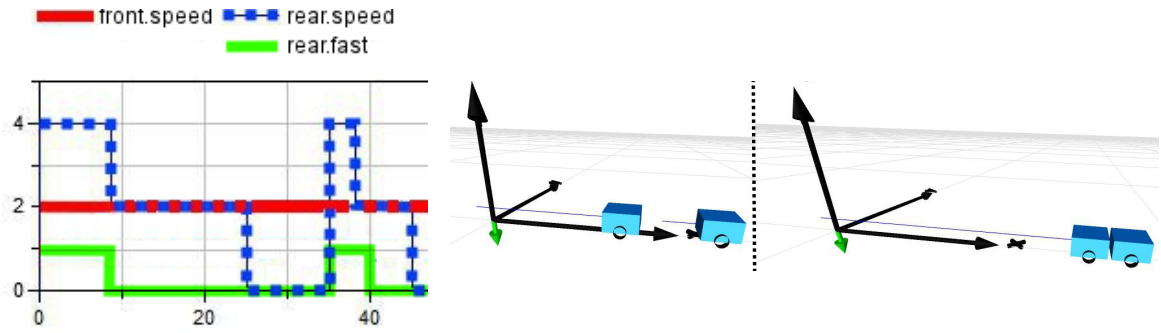


Figure 3.3: Model-in-the-loop Simulation Run with Variable Plot and 3D View

3.2 Process for Model-in-the-Loop Simulation of MechatronicUML

This section describes the process and the toolchain for the transformation of platform-independent MechatronicUML models into Modelica models. Furthermore, it describes how to simulate the transformed MechatronicUML models in Dymola [Dym]. As a result,

we ensure the fulfillment of the assumption that the control behavior together with the physics preserves the timed coordination behavior of MechatronicUML.

Figure 3.4 shows the process for deriving a MiL simulation of a MechatronicUML model using the Business Process Model and Notation (BPMN) [BPMN]. The process adapts the process of Heinzemann [Hei15] who defines the process for MATLAB Simulink and Stateflow in parallel to this work. The process involves software engineers who are familiar with MechatronicUML and system engineers who are familiar with Modelica. The source of the process is a MechatronicUML model of the platform-independent software and a Modelica model of the physical parts and the controllers. In Task T_1 software engineers transform their MechatronicUML model to a Modelica model automatically. They use the transformation that we describe in the following sections. Afterward, In Task T_2, system engineers define Modelica components for testing scenarios and Modelica components that simulate further environment properties for a MiL simulation. Next, engineers can integrate in Task T_3 a Modelica model for devices and controllers into the existing model. Therefore, either continuous components are implemented or replaced by existing Modelica components. Furthermore, continuous ports that are not yet connected to continuous components can be connected to Modelica components. As a result, we have a Modelica model that is valid and can be used for testing a CPS via simulation runs. Finally, in Task T_4 system engineers validate the integrated platform-independent model in Dymola by performing MiL simulation runs. As a result, the interaction between the coordination software and the physical system is validated. Thereby, the quality is improved in an early development phase, when no physical prototype is available. As a result, the software gets a higher-quality.

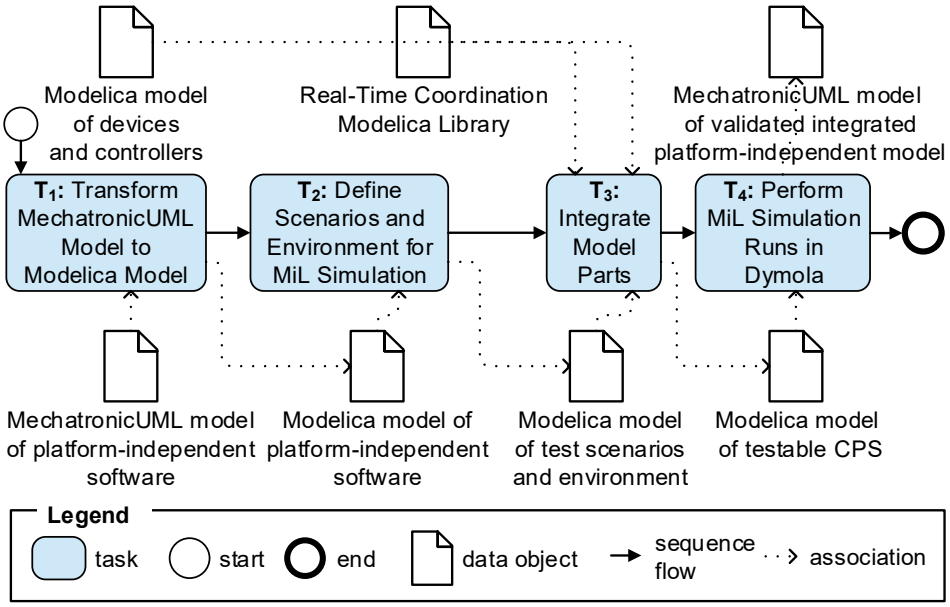


Figure 3.4: Process for Deriving a MiL Simulation of a MechatronicUML Model (Adapted from [Hei15])

Figure 3.5 shows an overview of the transformation approach and the involved toolchain for the MechatronicUML to Modelica transformation. The following Section 3.3 describes the Real-Time Coordination Modelica library, which this transformation uses and Section 3.4 describes the transformation itself.

The left part of the figure shows the different parts of a platform-independent MechatronicUML model (cf. Section 2). It serves as the source model for the transformation and is defined by the MechatronicUML meta-model [*DPP+16]. We

use the MechatronicUML Tool Suite to build up these models. The structural parts of a MechatronicUML model are the Interface Description that defines all message types, the Component Types, and the initial CIC. The behavioral parts of a MechatronicUML model are the RTSCs and the expressions defined by the MechatronicUML action language.

The right part of Figure 3.5 shows the parts of a Modelica model that are required to emulate the runtime behavior of MechatronicUML in a MiL simulation. It serves as the target model for the transformation and is defined by the textual syntax specification of Modelica [MODELICA]. In the following of this section, we use the graphical annotations of Modelica to illustrate the transformation. The structural parts are defined by components in the form of Modelica classes and Modelica connection diagrams (cf. Section 3.1.1). The state-based and communication behavioral parts are defined by elements from the StateGraph2 library (cf. Section 3.1.2) and by elements from the Real-Time Coordination library (cf. Section 3.3). The behavioral parts that are specified by the MechatronicUML action language are defined by algorithmic Modelica code (cf. Section 3.1).

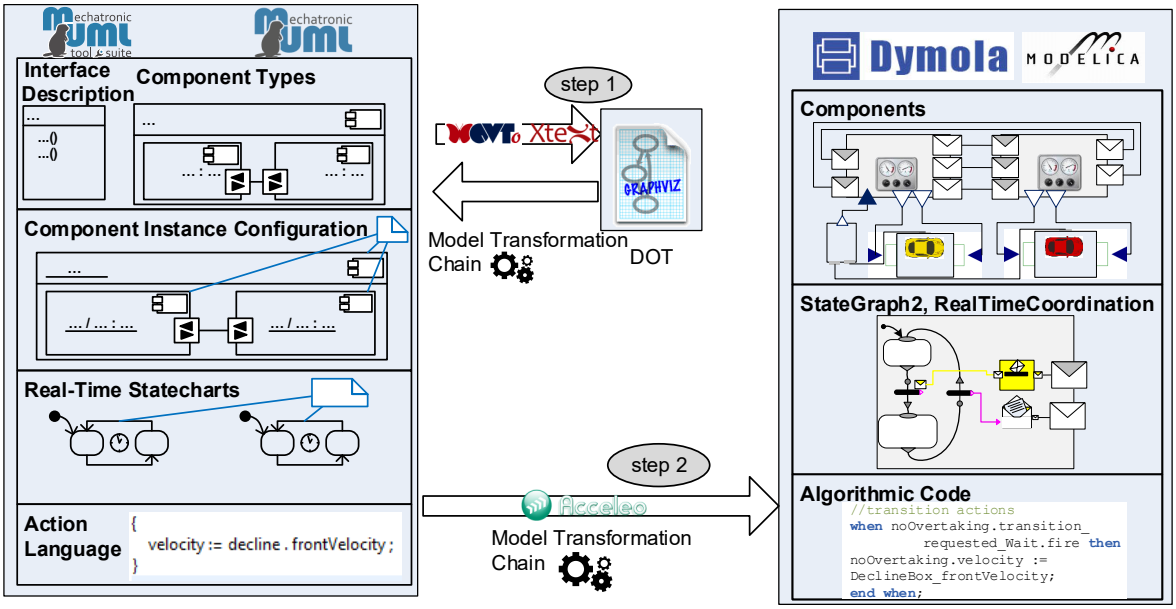


Figure 3.5: Toolchain for Transformation of MechatronicUML Models to Modelica Models with Graphical Annotations (Adapted from [*PHM+14])

The middle part of Figure 3.5 shows the transformation steps within the toolchain. In step 1, we transform the CIC and the RTSCs into the graph description language DOT [DOT]. We use DOT to compute a layout that we can use to generate graphical annotations for Modelica that are used to represent the generated Modelica elements within a connection diagram. In Modelica we have in contrast to MechatronicUML more model elements (Modelica objects), e.g., for each transition an own Modelica object. Therefore, we require a new layout and cannot adopt the layout of the existing MechatronicUML diagrams. During the transformation to DOT we create nodes for all elements that become a Modelica object and edges for all elements that become a Modelica connection. We split this transformation into two phases. In the first phase, the transformation uses the procedural Model-to-Model (M2M) transformation engine Query View Transformation Operational (QVTo) [QVT] to create a graph model. In the second phase, the transformation uses the grammar-based Model-to-Text (M2T) transformation engine of Xtext [Xte] to create a textual DOT-file. Afterward, we use the external layout tool Graphviz [GRAPHVIZ] to render the graph and store this calculated layout information as graphical annotations for the MechatronicUML

elements. Figure 3.5 sketches the annotations as blue-framed icons. "This layout information is utilized in the subsequent transformation to Modelica for generating the Modelica graphical annotations." [*PHM+14] Thereby, we are able to improve the understandability of generated Modelica models for engineers within Dymola [Dym]. In step 2, we transform the platform-independent MechatronicUML model to Modelica classes and objects. "We encompass the component instance configuration, consisting of discrete components (i.e., containing state-based coordination behavior) as well as continuous components (i.e., stubs for the control behavior to be embellished within Dymola)." [*PHM+14] The transformation uses the template-based transformation engine Acceleo [Acc] to create textual Modelica-files.

3.3 Real-Time Coordination Modelica Library

This section describes the Real-Time Coordination Modelica library. The section is based on our publication [*PDS+12] on the international Modelica conference in 2012. The goal of the library is to offer sophisticated elements for modeling coordination behavior to ease the manual modeling of coordination behavior within Dymola and to ease the transformation of MechatronicUML to Modelica.

"We developed the Real-Time Coordination Library [*PDS+12] in Modelica to provide system engineers the possibility to extend their existing physical models with coordination behavior, within their known language and tooling. The Real-Time Coordination Library represents the MechatronicUML concepts for Real-Time Coordination Protocols (RTCPs) including RTSCs. The library is freely available under the Modelica 2 license." [*PDM+14] The Real-Time Coordination Modelica library has been awarded with the second price for the best free Modelica library in 2012. It is published as official 3rd-party Modelica library and is available for download on Github [PWD].

3.3.1 Synchronization Connectors and Ports

A network of timed automata uses synchronization channels to communicate with each other. Synchronization channels enable to send an event to other timed automata and if another receiver automaton is able to handle the event both automaton can fire transitions jointly [AD94; BY04]. Especially, this kind of communication enables to perform synchronized actions within orthogonal regions of a composite state in a safe way. MechatronicUML uses synchronization channels to reconcile the communication with at least two other software components or respectively CPSs [*DPP+16].

"For the modeling of synchronous communication, we extended transitions by synchronization ports (sync ports). Sync ports sub-divide into sender sync ports and receiver sync ports. A sender sync port of one transition is connected to a receiver sync port of another transition by a synchronization connector. We represent a sender sync port as a non-filled orange circle, a receiver sync port as a filled orange circle and a synchronization connector as an orange line. (...) A transition that is connected via its sender or receiver sync ports to the receiver or sender sync ports of other transitions is allowed to fire if it is able to fire together with at least one of the connected transitions." [*PDS+12] A synchronization connector can have a selector expression of type Integer. Transitions with a synchronization connector that have a selector can only fire synchronized if the selector expressions of both synchronization connectors have the same value during the simulation.

Figure 3.6 shows an enhanced StateGraph connection diagram that is part of the class OvertakeeCommunicator. The diagram defines the states for a car that can be overtaken by another car in a cooperative manner. We call this car overtakee. By default, the state combination init, arbitrarySpeed is active. In this state combination, the overtakee is allowed

to drive with an arbitrary speed. An overtakee changes to the state combination requested, arbitrarySpeed if it gets a request to become overtaken. An overtakee can now decide if it agrees to be overtaken or if it declines the request. It changes to the initial state combination init, arbitrarySpeed if it declines the request. The overtakee commits itself not to speed up during the overtaking maneuver if it agrees to be overtaken. Therefore, it has to change from the state combination requested, arbitrarySpeed to the state combination overtaking, noSpeedUp. Such synchronous state changes require that transitions communicate with each other synchronously and agree to fire together or not at all. The transitions t3 and t5 have such a synchronization channel that is realized by the added synchronization ports and a synchronization connector. The overtakee changes back to the default state combination init,arbitrarySpeed synchronously when the overtaking maneuver is done. Therefore, the transitions t4 and t6 are connected via a synchronization channel.

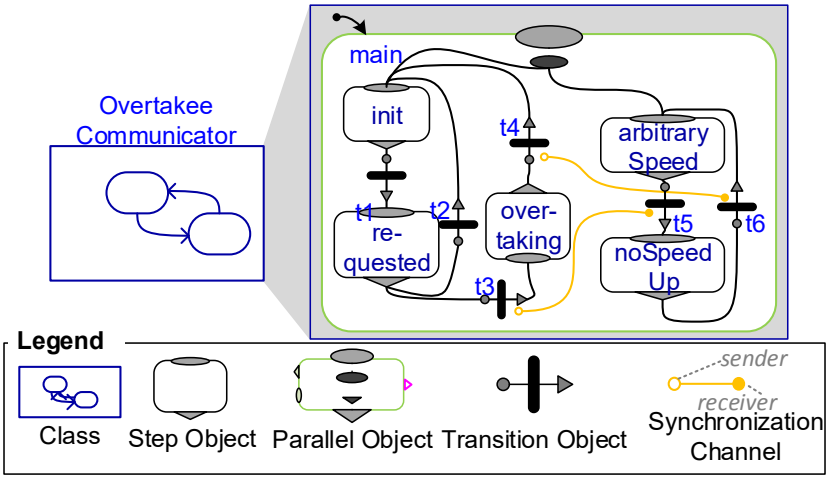


Figure 3.6: OvertakeeCommunicator Class with Synchronized Parallel Behavior

The implementation in Modelica is presented with the help of the dependency graph in Figure 3.7 for one sender and one receiver transition. Synchronization channels can also be used to sync n sender transitions with m receiver transitions. In the following, we only explain the one-to-one synchronization for simplicity. The n-to-m synchronization uses basically the same algorithm. We use for loops to iterate over all sender and receiver transitions. We add to the transition class the Boolean property preFire. Furthermore, we add to sending transition the Boolean properties fire_ready_s and fire_s and add to receiving transition the Boolean properties fire_ready_r and fire_r. We use these class properties to perform the handshake between sender and receiver transitions by checking the following conditions.

"First, the necessary conditions for firing each of the transitions (without synchronization) have to be satisfied, i.e., the preceding generalized Step has to be active, the *[guard] condition* of the transition must hold and the optional *condition port* of the transition must be set. If all of these conditions hold, the property *preFire* of each of the transitions will evaluate to true." [*PDS+12]

"Furthermore, if an *after time* is specified for the transition it must have expired. The after time construct is new and replaces the delay (wait) time from the original version of the StateGraph2 library. [We do not support the delay time semantics when using the Real-Time Coordination Library.] It differs from the delay time in that at least the after time must have expired [after the state gets active] to let the transition fire. In contrast, the semantics of the delay time is that the delay time must have expired after the transition is fireable [and

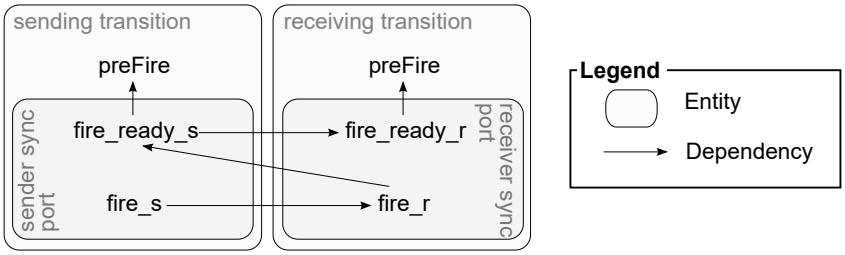


Figure 3.7: Dependency Graph of Conditions for Firing of Transitions with Synchronization (Adapted from [*PDS+12])

not only that the state is active] in order to let the transition fire. We introduced the after time semantics because it might happen that for two transitions that need to synchronize the time instants in which they are allowed to fire might not match due to their delay time. " [*PDS+12]

The final handshake is implemented as follows: "If preFire of the sending transition, i.e., the transition whose receiver sync port is connected to the synchronization connector, is true, the signal fire_ready_r of the receiver sync port is set to true. If for the sending transition, i.e., the transition whose sending sync port is connected to the synchronization connector, holds that preFire is true and it receives the signal fire_ready_r over its sender sync port then the signal fire_ready_s of its sender sync port is set to true. If the signal fire_ready_s is true in the receiving transition the signal fire_r of the receiver sync port is set to true. Finally, if fire_r is recognized to be true in the sending transition the signal fire_s of its sender sync port is set to true and both transitions are ready to fire." [*PDS+12]

3.3.2 Message-Based Communication

Systems that act independently can exchange information by using asynchronous message-based communication [CDKB11]. Thereby, the communicating parties are only loosely coupled and do not have to wait for the communication partner. The receiver also does not have to pause it current processing because incoming messages are buffered in the background and the receiver can decide when it can process the message. The logical communication is realized on demand at any time and no external clock is required for synchronization on the application layer. The coordination of CPSs can be realized by using asynchronous message-based communication [Hei15; Dzi17].

"For the modeling of asynchronous communication, we introduce two new components named message and mailbox. Each instance of the message component has two purposes. On the one hand, it defines a certain message type by specifying an array of formal parameters which might be of type Integer, Boolean or Real. As an example, one message type might be defined by the array $(Integer[2], Boolean[1], Real[1])$. The parameter array of a message type is also-called its *signature*. On the other hand, an instance of the message component is responsible for sending a message whenever a connected transition fires. A transition is able to signal to a message component instance to send a message if the firePort of the transition is connected to the conditionPort of the message component instance." [*PDS+12]

"For each message type exists exactly one instance of the mailbox component with the same signature. The message type sends its messages to the mailbox instance [for buffering]. To specify which message type belongs to which mailbox instance the message_output_port of the message type is connected to the mailbox_input_port of the mailbox instance." [*PDS+12]

"A mailbox instance defines a finite FIFO queue where the size of the queue is settable at design time. In order to let a transition receive a certain message from such a queue, its

transition_input_port is connected to the mailbox_output_port of the mailbox instance. Then, the transition is allowed to fire if the mailbox instance signals that at least one message is present. (...) Synchronous and asynchronous communication can be combined at one transition. Besides the synchronization conditions the mailbox instance additionally has to signal to the transition that at least one message is available." [*PDS+12] The simulation run fails with a proper failure message to the user if the size of a mailbox is too small during a simulation run. Currently, the mailbox does not considers Quality of Service (QoS) connector assumptions that concern the delay of messages.

Figure 3.8 shows the OvertakeeCommunicator class with enhanced asynchronous communication behavior by using messages and mailboxes. As a result, the class has the ability to send, to receive, to buffer, and to process messages to realize complex state-based coordination behavior. The mailbox instance RequestBuffer buffers all incoming request messages. The transition t1 from state init to state requested is able to fire as soon as a request message is buffered and the init state is active. The transition t1 deactivates the state init, dequeues the message from the mailbox, processes it, and activates the state requested while being fired. Accordingly, transition t4 fires if the state overtaking is active and the mailbox FinishedBuffer stores a finish message. Transition t2 deactivates the state requested, sends a decline message, and activates the state init directly, without waiting for a response, if it fires. Accordingly, transition t3 fires and sends an accept message if the state combination requested, arbitrarySpeed is active.

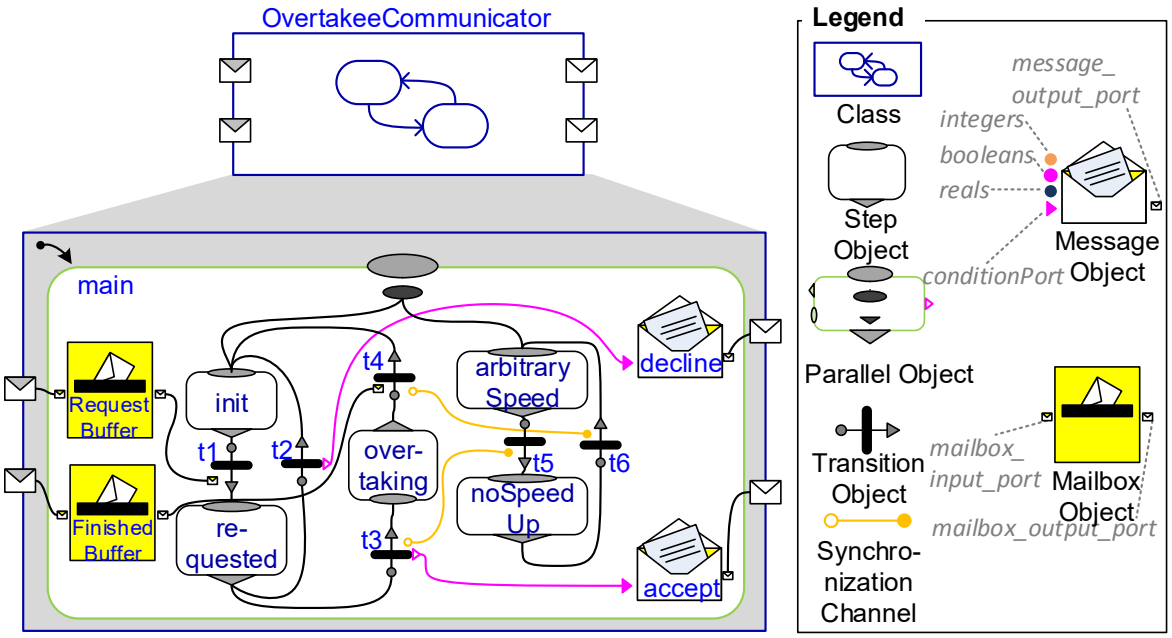


Figure 3.8: OvertakeeCommunicator Class with Asynchronous Communication Behavior

"If two extended StateGraph2 models are included in different component instances they might still communicate asynchronously across the boundaries of these component instances with the help of delegation ports. Therefore, one component defines an output delegation port and the other defines an input delegation port. Both delegation ports are connected. Then, the component instance containing the message type connects the message type to the output delegation ports and the component instance containing the mailbox instance connects the mailbox instance to the input delegation port." [*PDS+12]

Figure 3.9 shows the class CommunicatorTest that instantiates OvertakeeCommunicator and OvertakerCommunicator for testing the coordination behavior between these objects. Both objects

have delegation ports for transmitting messages. The object OvertakeeCommunicator is able to send the messages decline and accept as an answer to a received request message that can be sent by the object OvertakerCommunicator. Furthermore, the object OvertakerCommunicator can inform the object OvertakeeCommunicator that the overtaking maneuver is finished via sending a finished message.

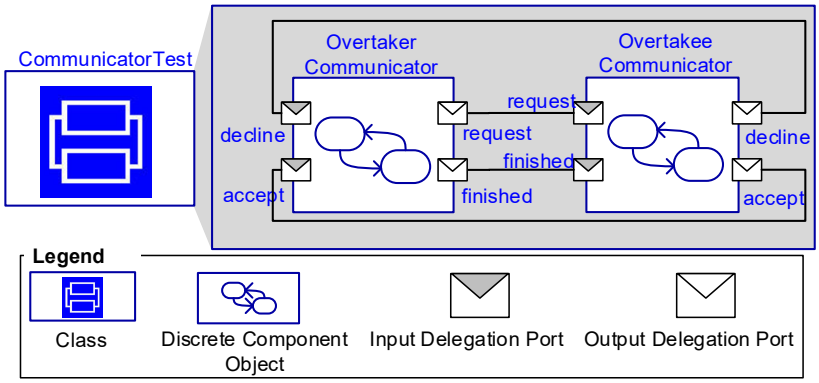


Figure 3.9: CommunicatorTest Class with Two Communicating Objects

3.3.3 Clocks, Invariants, and Clock Constraints

Timed automata [AD94] and MechatronicUML [*DPP+16] specify clock-based timing behavior. This specification enables to verify safety and real-time properties formally using timed model checking [BY04; GDHS15]. Invariants constrain states to be active only within certain clock ranges. Clock constraints restrict when transitions are allowed to fire.

"For the modeling of real-time behavior according to timed automata, we extended the StateGraph2 library by three components named Clock, Invariant, and ClockConstraint. Clocks are real-valued variables whose values increase continuously and synchronously with time. Clocks might be reset to zero upon the activation of a generalized step or firing of a transition. An invariant is an inequality that specifies an upper bound on a clock, e.g., c < 2 or c <= 2 where c is a clock. Invariants are assigned to generalized steps and are used to specify a time span in which this generalized step is allowed to be active. A clock constraint might be any kind of inequality specifying a bound on a certain clock, e.g., c > 2, c >= 5, c < 2, c <= 5 where c is a clock. Clock constraints are assigned to transitions in order to restrict the time span in which a transition is allowed to fire." [*PDS+12]

"Clocks are displayed as a rectangle containing a clock icon, invariants are displayed as rectangles containing the corresponding inequality and a transition icon. Clock constraints are displayed as [a] rectangle containing the corresponding inequality and a step icon. The clock which is used by an invariant or a clock constraint is connected via its (...)[value] port with the clockValue port of the invariants, and clock constraints." [*PDS+12]

Figure 3.10 shows the OvertakeeCommunicator class with enhanced real-time communication behavior by a clock, a clock constraint, and an invariant. We omit the states arbitarySpeed and noSpeedUp for better readability. The diagram defines the behavior for an overtakee that commits itself to cooperate only for a temporally bounded time interval. The overtakee guarantees the overtaker that it does not speed up for 30 seconds. The timeout_clock measures the time since transition t3 has fired. The clock value is reset to zero upon the firing of transition t3, as its reset port is connected to the fire port of transition t3. Additionally, transition t3 deactivates the state requested, sends the accept message with the temporal upper bound of 30 seconds as

parameter, and activates the state overtaking. The transition t7 fires when the clock constraint activates its firePort and the state overtaking is active. The clock constraint activates its firePort when the value of the clock that is connected to the value port is greater or equal to the value of the bound port. The bound value is set to 30 by the bound object. Furthermore, a state invariant asserts that the clockValue is always smaller or equal to the bound value k=31 seconds as long as state the overtaking is active. Transition t7 deactivates the state overtaking and activates the state init.

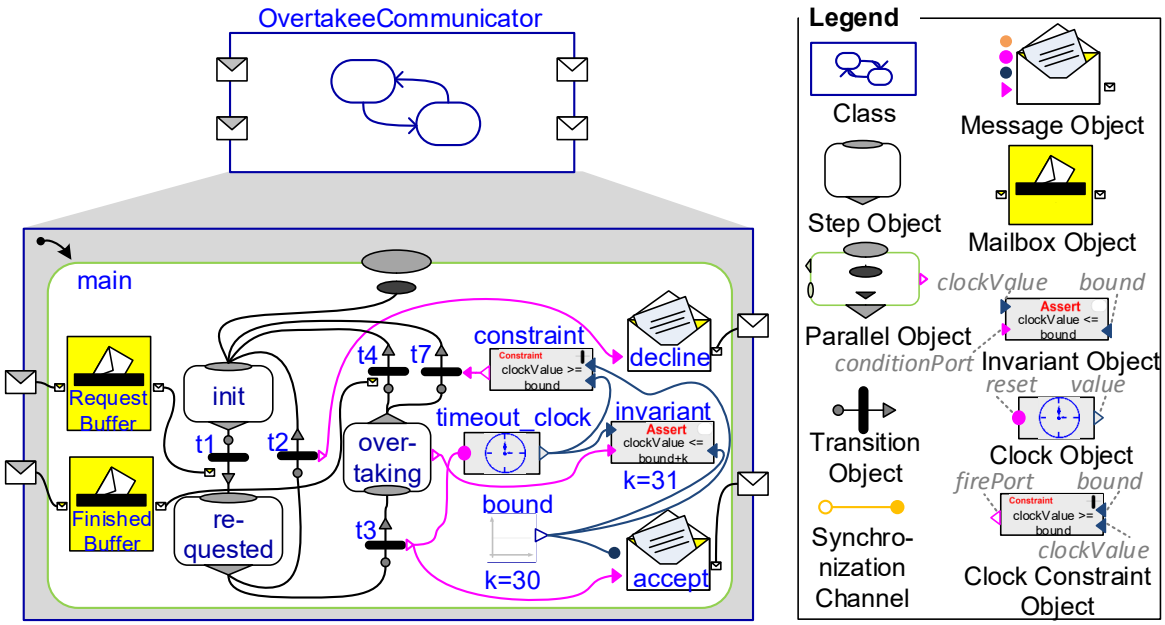


Figure 3.10: OvertakeeCommunicator Class with Real-Time Communication Behavior

3.3.4 Formal Syntax and Semantics Definition of the Real-Time Coordination Library

"This section covers the formal definition of an extended StateGraph2 model. The Real-Time Coordination library extends the structure of the model given in [OME+09] by synchronization connectors, mailboxes, clocks, invariants and clock constraints whereas the former two are required for synchronous and asynchronous communication resp. and the latter three are used for the specification of real-time behavior analogously to timed automata [AD94]. Due to the possibility of synchronization of two transitions, we altered the delay time of a transition to an after time, which has slightly different semantics." [*PDS+12]

"The extension is represented by the following tuple

$$Ext := (Sync, MBox, C, INV, CC)$$

where $Sync$ denotes the set of synchronization connectors required for synchronous communication. Let Msg be the set of messages used for asynchronous communication. Then $MBox : Msg \to \mathbb{N}$ determines for each message how often it is available in its corresponding mailbox. The real-time extension is covered by the set C of clocks, the set Inv of invariants and the set CC of clock-constraints." [*PDS+12]

"(...) [T]he set of messages results from all possible combinations of message parameters. We abstracted from message parameters here, simply saying that there exists a set of distinct

messages. [In MechatronicUML message parameters do not influence the semantics of the message processing. For example, it is not possible to consume a message only if its parameters have certain values. Nevertheless, for each message instance whose message type has parameters the corresponding parameter values are sent.] Furthermore, in the implementation of our extension there exists the MailBox component for the realization of asynchronous communication. Since the number of messages included in a certain mailbox suffices to be able to determine whether a transition that requires such a message is able to fire, we abstracted from the mailboxes here in the form of the $MBox$ function. [The message parameters of each message instance are stored within the MailBox and hand over to the consuming transition but are not relevant for the message processing semantics.] The following definition consists of elements that were already defined in [OME+09]. For the sake of completeness, we describe and list them." [*PDS+12]

"With the help of our extension Ext, we define an *extended StateGraph2 model (ESGM)* Γ as follows:

$$\Gamma := (V_c, G, T, G_I, G_E, Ext)$$

where

- V_c is a set of Boolean expression as defined in [OME+09].
- G is a set of generalized steps $G = \{g_1, g_2, \ldots\}$
  A generalized step g_i is defined as a 7-Tuple

$$g_i = (\Gamma_s, I, O, S, R, Inv_{g_i}, RESET_{g_i})$$

  where
  - Γ_s is a possibly empty set of sub-graphs $\Gamma_s = \{\gamma_1, \gamma_2, \ldots\}$. A sub-graph $\gamma_i \in \Gamma_s$ is again an ESGM. Note that this recursive definition allows an arbitrary deep nesting of ESGMs.
  - I is a vector of in (entry) ports $I = [i_1, i_2, \ldots]$. An in-port is a connection point incoming transitions of g_i are connected to.
  - O is a vector of out (exit) ports $O = [o_1, o_2, \ldots]$. An out-port is a connection point outgoing transitions of g_i are connected to.
  - S is a possibly empty vector of suspend ports $S = [s_1, s_2, \ldots]$. A suspend port is a connection point. Outgoing transitions of g_i are connected to it. The difference to out-ports is that the active steps of all sub-graphs of g_i are stored for later restore.
  - R is a possibly empty vector of resume ports $R = [r_1, r_2, \ldots]$. A resume port is a connection point. Ingoing transitions of g_i are connected to it. The difference to in-ports is that the active generalized steps of sub-graphs of g_i that were active when g_i was left by a suspend port are restored.
  - $Inv_{g_i} \subseteq Inv$ is a set of invariants. An invariant describes that a clock must never exceed a certain bound when the generalized step g_i is active. It is denoted as an inequality of the form $c \leq n$, where $c \in C$ is a clock and $n \in \mathbb{N}$ is a natural number (including zero).
  - $RESET_{g_i} \in C$ is a set of clocks that member clocks are to be reset to zero when the generalized step g_i is activated.

  A generalized step that has in and out-ports but no other ports and no sub-graphs, i.e., $I \neq \emptyset$, $O \neq \emptyset$ and $R = S = \Gamma_s = \emptyset$ is called *step*. A generalized step that has resume ports, suspend ports or sub-graphs, i.e., $R \neq \emptyset$, $S \neq \emptyset$ or $\Gamma_s \neq \emptyset$ holds, is called *parallel step*.
- T is a set of transitions $T = \{t_1, t_2, \ldots\}$. A transition $t_i \in T$ is defined by the 10-tuple

$$t_i = (p_{t_i}^{IR}, p_{t_i}^{OS}, C_{t_i}, A_{t_i}, CC_{t_i}, R_{t_i}, S_{t_i}^R, S_{t_i}^S, M_{t_i}^R, M_{t_i}^S)$$

  where
  - $p_{t_i}^{IR}$ is a connected port of an in or resume vector of a succeeding generalized step $g_i \in G$.

  - $p_{t_i}^{OS}$ is a connected port of an out or suspend vector of a preceding generalized step $g_i \in G$.
  - $C_{t_i} \in V_c$ is the fire condition associated with t_i.
  - $A_{t_i} \in \mathbb{R}$ is the after time associated with t_i. Note that we consciously chose the name after time instead of delay time as in the original definition in [OME+09] since the semantics of the after time will be different from the one of the delay time.
  - $CC_{t_i} \in CC$ are the clock constraints associated with t_i.
  - $R_{t_i} \in C$ are the clocks to be reset when t_i fires.
  - $M_{t_i}^R \subseteq Msg$ is the message that must be received when t_i fires.
  - $M_{t_i}^S \subseteq Msg$ is the message that is sent when t_i fires.
  - $S_{t_i}^R \subseteq Sync$ is the synchronization connector that has to be set by another transition when t_i fires.
  - $S_{t_i}^S \subseteq Sync$ is the synchronization connector that is set if t_i is firable.

  We further define that a transition might have at most one message that is to be received and at most one message that is to be sent, i.e., $|M_{t_i}^R| \leq 1$ and $M_{t_i}^S \leq 1$ resp., and at most one synchronization connector over which a signal is sent or received, i.e., $|S_{t_i}^R| + |S_{t_i}^S| \leq 1$.
- $G_I \subseteq G$ contains the initial generalized step of Γ.
- $G_E \subseteq G$ contains the exit generalized step of Γ."[*PDS+12]

"As a well-formedness constraint, we assume that every ESGM has exactly one initial state and at most one exit state, i.e., $|G_I| = 1$ and $|G_E| \leq 1$. Furthermore, we assume that the *uppermost ESGM* $\Gamma = (V_c, G, T, G_I, G_E, Ext)$, i.e., that ESGM that is not embedded by any other ESGM, does not have an exit generalized step, i.e., $G_E = \emptyset$." [*PDS+12]

"For the definition of the semantics, we give an interpretation algorithm that is analogous to the one given in [OME+09]. Additionally, consider the added elements, i.e., when a generalized step is active the corresponding invariant must not be violated. Further, when a transition fires, its clock constraint must be satisfied, it must be able to synchronize, and to receive the required messages."[*PDS+12]

We published the following interpretation algorithm in [*PDS+12], which describes how Modelica uses the library elements during a simulation run:

1. Activate the initial generalized step $g \in G_I$. If g has sub-graphs, then recursively activate the initial generalized steps of all of its embedded sub-graphs.
2. Determine the set $T_{fireable}$ of all transitions t_i that satisfy:
   - its condition C_{t_i} is true,
   - the required after time A_{t_i} has passed,
   - its in or resume port $p_{t_i}^{IR}$ is set to true,
   - if its preceding generalized step has sub-graphs, the exit generalized steps of all of these sub-graphs are recursively activated
   - if $M_{t_i}^R \neq \emptyset$ and $m \in M_{t_i}^R$ is the message to be received by t_i, the mailbox of m contains at least one message, i.e., $MBox(m) > 0$.
   - there exists no other transition $t_j \in T_{fireable}$ that has the same preceding generalized basic step and has higher priority than t_i where the priority results from the index of the transition in the port vector (see [OME+09]).
3. For all $t_i \in T_{fireable}$ do:
   i. if $S_{t_i}^S \neq \emptyset$ and $s \in S_{t_i}^S$ is the synchronization connector of t_i for sending a signal, set s to true
4. Determine the set $T_{syncable}$ of all transitions $t_i \in T_{firable}$ that satisfy:
   - either $S_{t_i}^R = \emptyset$ or
   - if $S_{t_i}^R \neq \emptyset$ and $s \in S_{t_i}^R$ is the synchronization connector of t_i, t_i is set to true
5. For all $t_i \in T_{syncable}$ fire t_i as follows:
   i. Deactivate the preceding generalized step g of t_i. If g_i includes sub-graphs deactivate these sub-graphs recursively.

ii. Activate the succeeding generalized step g' of t_i. If g' includes sub-graphs activate these sub-graphs recursively as follows:
   - if t_i is connected to g' by a resume port, the generalized steps of g' and of all sub-graphs of g' that were active the last time g' was active are recursively activated
   - else, activate all initial generalized steps of g' and its sub-graphs recursively.

iii. if $M^R_{t_i} \neq \emptyset$ and $m \in M^S_{t_i}$ is the message to be received by t_i, then take the first message out of the mailbox of m, i.e., $MBox := (MBox \setminus \{(m, d)\}) \cup \{(m, d-1)\}$ where $d \in \mathbb{N}$ is the amount of messages in the mailbox before t_i fires.

iv. if $M^S_{t_i} \neq \emptyset$ and $m \in M^S_{t_i}$ is the message to be sent by t_i, then put one message into the mailbox of m, i.e., $MBox := (MBox \setminus \{(m, d)\}) \cup \{(m, d+1)\}$ where $d \in \mathbb{N}$ is the amount of messages in the mailbox before t_i fires.

6. Goto 2.

3.3.5 Case Study

We evaluate our Real-Time Coordination Modelica library based on coordination requirements of CPSs. Therefore, we conduct a case study on the basis of the guidelines defined by Kitchenham, Pickard, and Pfleeger [KPP95] and Runeson and Höst [RH08].

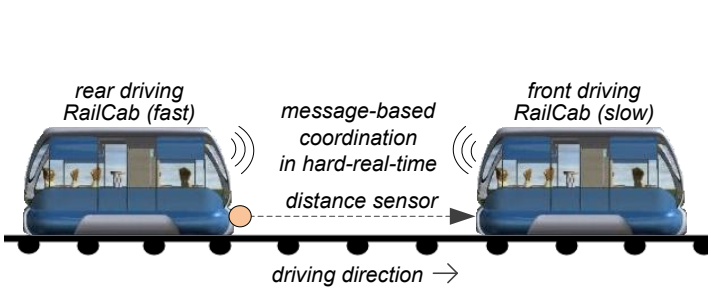
3.3.5.1 Context and Case

The objective of our case study is to evaluate, whether the Real-Time Coordination Modelica library is sufficient to model CPSs, whose structure and behavior is specified using MechatronicUML. Therefore, we evaluate the evaluation question: Is it possible to model real-time coordination patterns as defined by Dziwok *et al.* [DHT12; DBHT12] using the Real-Time Coordination Modelica library? The real-time coordination patterns represent proven coordination protocols for the coordination between CPSs as reusable entities. We show by answering this question that the library is able to model general coordination use cases and is not limited to a specific coordination application. Furthermore, we realize the coordination behavior of a platooning scenario from the railway domain. Within a platoon or respectively convoy, so called RailCabs, drive closely to use the slipstream to reduce the air resistance and thereby the required energy for driving [*HSD15]. RailCabs are autonomous railroad cars. If the coordination of RailCabs fails due to a logical fault within the coordination behavior an accident may lead to serious harm.

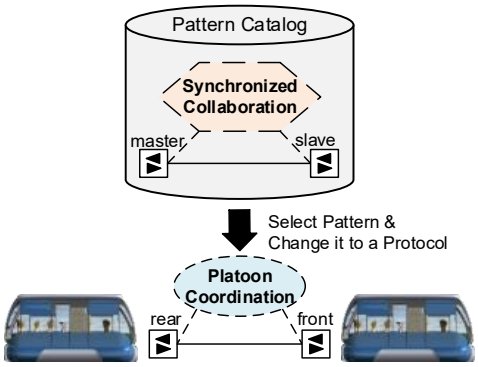
Figure 3.11a shows the platooning scenario that we model and simulate. In particular, we use the synchronized-collaboration pattern that we implement using the Real-Time Coordination Modelica library to realize the platoon coordination protocol as shown in Figure 3.11b. The behavior of this system could be adapted to an automotive platooning scenario. During performing this case study, a Modelica Coordination Pattern library, which formalizes the eight real-time coordination patterns as easily reusable Modelica classes, was created and published [*PDM+14]. The resulting Modelica Coordination Pattern library is available for download on Github [PT] as official 3rd party Modelica library under the open source Modelica license. We refer to our paper [*PDM+14] for further library details and a process for developing using the library.

3.3.5.2 Hypothesis

We define the *evaluation hypothesis* H_1 that the elements of the Real-Time Coordination Modelica library are sufficient to model syntactically and semantically correct CPSs as defined by MechatronicUML.



(a) Scenario



(b) Using a Pattern for Realization

Figure 3.11: Railway Platooning (Adapted from [*PDM+14])

3.3.5.3 Analysis Procedure

For evaluating our hypothesis, we implement the eight real-time coordination patterns [DBHT12] and a concrete coordination scenario manually using the tool *Dymola 2014 FD01* [Dym] and our library [PWD]. We check if these commonly coordination design problems can be modeled and can by simulated. Furthermore, we check if we can model and simulate the concrete application scenario, which has additional domain-specific information, like concrete time values and message parameters. We consider our evaluation as successful if the eight coordination patterns and the two coordination scenarios from different domains are formalized as Modelica classes and are tested by simulation runs that show the intended behavior. We examine the simulation runs by plotting variables in diagrams and activate the debug option of Dymola to log all events during the simulation and the initialization. Especially, discrete changes that take no simulation time and their order can be observed within this log in detail.

3.3.5.4 Preparation of the Data Collection

In preparation for the case study, we consider general coordination use cases by obtaining the real-time coordination pattern catalog defined by Dziwok *et al.* [DBHT12] and study the development process for developing coordination protocols using these design patterns [DHT12]. Furthermore, we define a platooning behavior that is inspired by [HP14] but focuses only on building up and breaking up a platoon of two RailCabs and neglect structural reconfigurations.

3.3.5.5 Data Collection Procedure

We start by implementing for each of the eight patterns a Modelica package. Each package contains Modelica classes for the two involved roles of the coordination, e.g., producer and consumer. The corresponding Modelica package contains only one Modelica class if both roles are equal, e.g., the role peer for the master-slave-assignment pattern. Furthermore, we implement artificial test cases for the patterns that instantiate and connect the roles.

Afterward, we use the realized synchronized collaboration pattern to implement the real-time coordination protocol of the platoon coordination scenario. Figure 3.12 shows the realization in Modelica. We instantiate, refine, and connect the Modelica classes Protocol_Master_Role and Protocol_Slave_Role of the involved communicating roles for the realization. We refine the idle state of the platoon coordination protocol and add message and mailbox parameters for

transmitting status information. Next, we integrate two instances of the class Robot shown in Figure 3.1 and connect each to its corresponding protocol role. The robots serve as a physical testbed for a RailCab where the cruising speed can be controlled. Furthermore, we implement a Modelica Distance class that measures the distance between both robots. Furthermore, we add Modelica timetables that set and change the cruising speed of the robots during a simulation run and decide if a platoon leader is ready to build up a platoon.

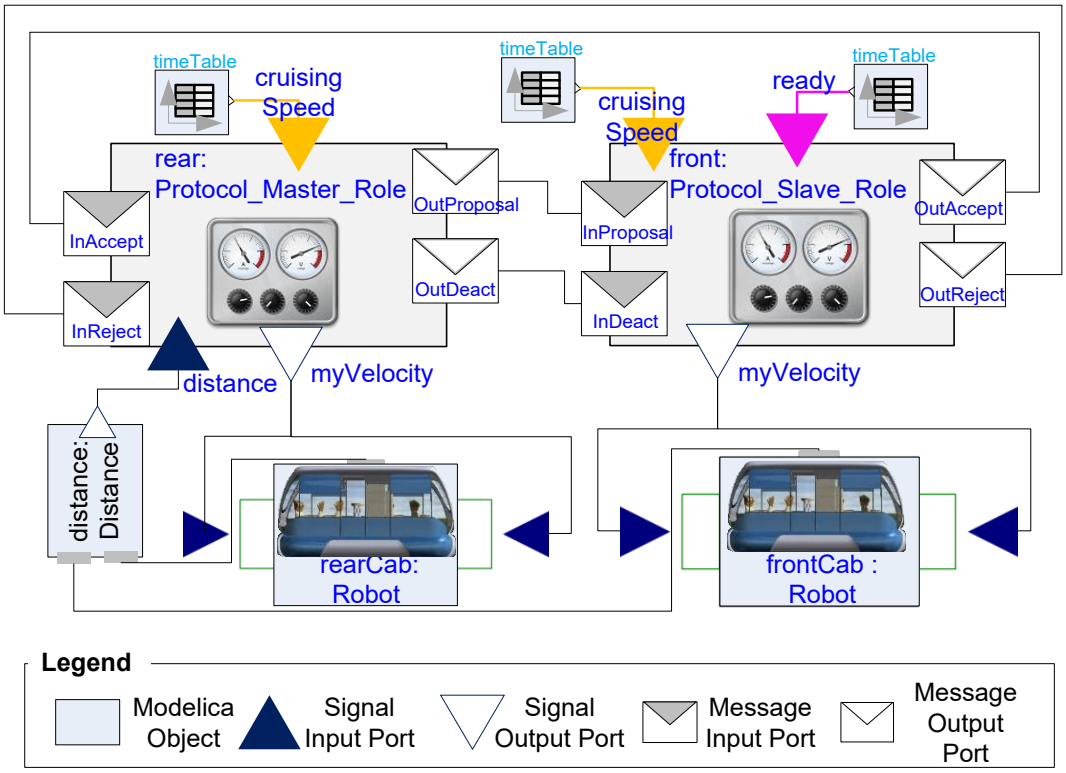


Figure 3.12: Railway Platooning Realized in Modelica (Adapted from [*PDM+14])

Finally, we check if the artificial test cases and the platoon coordination scenario validate in Dymola without errors. If this is so, we translate the Modelica model to simulation code and compile it with the Microsoft Visual Studio 2010 C++ compiler. Thereafter, we simulate all models using the variable-step DASSL integration method [Pet82; CK06] of Dymola [Dym]. A manual audit of the simulation results shows that all test cases fulfill the intended state-based behavior and exchange the specified message sequences. Furthermore, the simulation results of different simulation runs of the platooning scenario show that the rear RailCab sends platoon proposals to the front RailCab if the distance between both RailCabs is getting smaller. Then the rear RailCab gets either an accept message from the front RailCab or a reject message. In both cases the rear RailCab gets the current cruising speed of the front RailCab. The rear RailCab builds up a platoon and adapts its speed to the same speed of the front RailCab in the case of receiving an accept message. The rear RailCab slows down to drive significantly slower than the front RailCab for a period of time in the case of receiving a reject message. An active platooning is broken up after a period of time by sending a deactivation message from the rear RailCab to the front RailCab. Concluding, the audit of the simulation behavior shows that the observed behavior is compliant to the description within the pattern catalog [DBHT12] and the intended platooning behavior.

3.3.5.6 Interpreting the Results

The results of our case study show that our Real-Time Coordination Modelica Library is sufficient for modeling and simulating general coordination use cases as defined by the real-time coordination pattern catalog [DBHT12]. The audit and compilation of the models and simulation runs show that the implemented models are syntactically and semantically correct by means of the definition of MechatronicUML. Thus, we conclude that our hypothesis H_1 is fulfilled.

3.3.5.7 Threats to Validity

Empirical studies have limitations and the results rely to a large extent on the research design. Each empirical study has to face threats to validity [RH08]. We use the systematics of Runeson and Höst for analyzing and reporting these threats by splitting the threats into the construct validity, reliability, internal validity, and external validity.

The construct "validity reflect to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions." [RH08] Our case study does not consider synchronization channels yet. They become important if one software component should implement several protocols that depend on each other. We have not considered them in this case study. Nevertheless, we provide reliable examples. Furthermore, we only tested the preservation of the semantics of MechatronicUML with all patterns, one application example, and artificial test cases. We think that our test result are valid, hence we are experts of the MechatronicUML semantics and the patterns and we perform a detailed examination. Despite the fact that we rate the examples as realistic coordination use cases, other examples from other domains could be remarkably different. We only tested our models with the DASSL variable-step solver provided by Dymola [Dym]. Other solvers might provide different results. The reliability "is concerned with to what extent the data and the analysis are dependent on the specific researchers." [RH08] We only checked manually that the modeled examples that use our Real-Time Coordination Modelica library fulfill the same behavior as MechatronicUML models. Although we carefully audit the simulation runs, we might have missed any deviations. The reliability of the study may vary if other modelers use different Modelica elements to realize the coordination patterns. Nevertheless, our implemented pattern are available publicly [PWD]. We do not examine any causal relations and do not generalize our findings because they are MechatronicUML specific. Therefore, we do not consider internal and external validity, which take care of these concerns [RH08]. Nevertheless, we think that coordination pattern and therefore also our Real-Time Coordination Library are usable for other cases. For example, Winkler uses the coordination pattern for designing the coordination for an inductive energy transmission system [Win14]. Furthermore, Dziwok evaluated the coordination pattern and its design [Dzi17].

3.4 Model-in-the-Loop Simulation of MechatronicUML

This section describes the transformation of the structural MechatronicUML model parts to Modelica and the transformation of MechatronicUML behavior to Modelica by using the StateGraph2 library (cf. Section 3.1.2), our Real-Time Coordination Library (cf. Section 3.3), and algorithmic code (cf. Section 3.1).

3.4.1 Transforming Component Instance Configurations (CiCs) to Modelica Connection Diagrams

This section describes the transformation of a MechatronicUML CIC into a Modelica connection diagram (cf. Section 3.1.1). A CIC, which is the source of the transformation, can contain multiple connected (hierarchical) component instances. The output, which is a Modelica connection diagram, contains multiple Modelica objects and their corresponding classes. The Modelica object hierarchy represents the structure of the CIC.

The next section defines how to transform atomic component instances. Afterward, Section 3.4.1.2 specifies the transformation of structured component instances.

3.4.1.1 Atomic Component Instance

We create a Modelica package and a contained Modelica class for each atomic component instance of a CIC.

Figure 3.13 shows the generation template for creating all required Modelica connectors for the newly created class. We generate a Modelica connector for each continuous port instance and for each hybrid port instance. In Modelica, connectors have the same name as the port instances in MechatronicUML. In-ports are transformed to Input connectors and out-ports are transformed to Output connectors. We generate different types of Input and Output connectors depending on the data type of the port. We generate Real connectors for ports of the type DOUBLE, Integer connectors for ports of the types (U)INT8, (U)INT16, (U)INT32, (U)INT64, and Boolean connectors for ports of the type BOOLEAN. Furthermore, we transform the port cardinality to the connector dimension for ports that represent arrays by appending the dimension within square brackets after the connector name. Discrete port instances are split up to several Message Delegation Ports as a single message delegation port can only handle a specific message type. Therefore, we generate for each receiver message type for each discrete port instance a separate message Input Delegation Port. Currently, we can only handle primitive message parameters.

Listing 3.2 shows the declaration template for Input Delegation Ports. We have to set the number of Integer, Boolean, and Real parameters via redeclaration because thereby Modelica allows to change the concrete type of the parameter arrays to the required size. As a result, the port can delegate messages with parameters. Accordingly, we generate for each raise message type for each discrete port instance a separate message Output Delegation Port.

```
1  RealTimeCoordinationLibrary.RealTimeCoordination.MessageInterface.
     InputDelegationPort [portName/](redeclare Integer
        integers['[' + (messageType.getIntegerParameters()->size()) + ']'/],
     redeclare Boolean
5       booleans['[' + messageType.getBooleanParameters()->size() + ']'/],
     redeclare Real
        reals['[' + messageType.getRealParameters()->size() + ']'/])
```

Listing 3.2: Creates a Declaration for an Input Delegation Port

Figure 3.14 shows an example of the transformation of a discrete atomic component instance.

In the left part, Figure 3.14 shows the MechatronicUML atomic component instance overtakeeCommunicator. The component instance has the hybrid port instance :color:INT32 that can receive a signal that encodes a color value. Furthermore, it has the discrete ports :overtakeeP, :informSectionCtrP, velocitySetterP, and informApproacherP. Each discrete port instance has a sender message interface and a receiver message interface description. The interface defines, which messages can be sent via the corresponding port or can be received. The discrete port instance

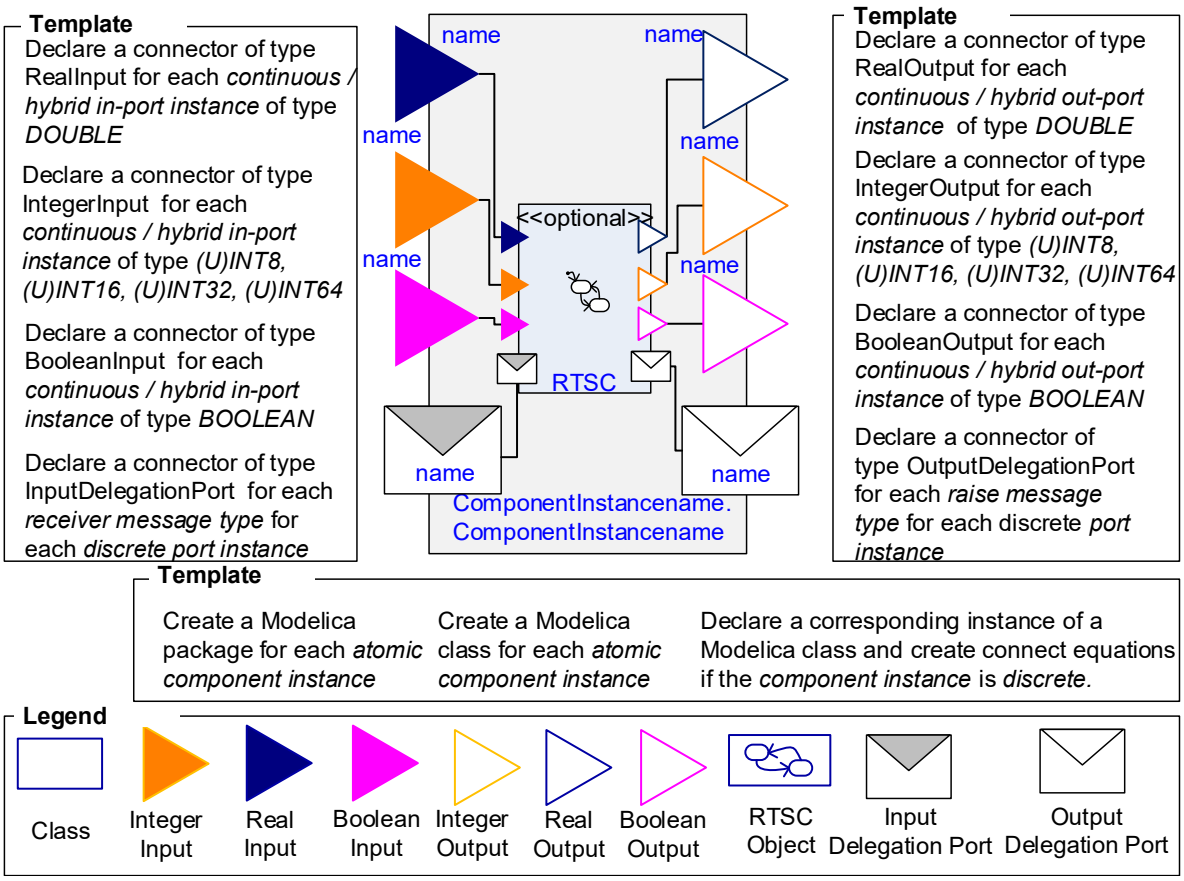


Figure 3.13: Generation Template for Creating a Modelica Class for an Atomic Component Instance

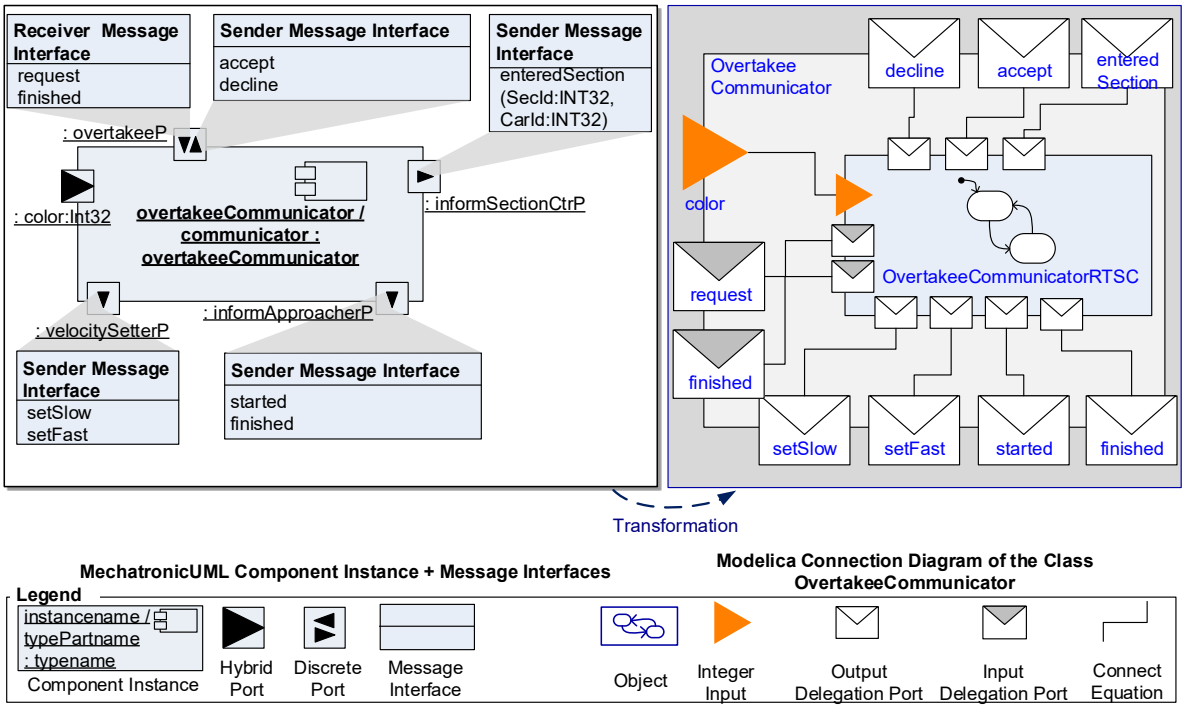


Figure 3.14: Transformation of a MechatronicUML Discrete Atomic Component Instance to a Modelica Class (Adapted from [*PHM+14])

:overtakeeP can receive the messages request and finished and can send the messages accept and decline. Furthermore, the discrete port instance :informSectionCtrP can send the message enteredSection(SecId:INT32,CarId:INT32), the discrete port instance :informApproacherP can send the messages started and finished, and the discrete port instance :velocitySetterP can send the messages setSlow and setFast.

In the right part, Figure 3.14 shows the generated Modelica class OvertakeeCommunicator. In this example, we focus on the generation of the class connectors and behavior instantiation. The hybrid port :color:INT32 is transformed to an Integer input connector in Modelica. The messages of the sender interfaces are transformed to corresponding OutputDelegationPort Modelica connectors and the receiver interfaces are transformed to corresponding InputDelegationPort Modelica connectors. A Modelica object is declared for the behavior of a discrete component instance. The connectors of the Modelica class are connected with the corresponding connectors of the object via connect equations.

3.4.1.2 Structured Component Instance

A structured component instance is composed of other components and defines the architecture for these components. We create a Modelica package and a contained Modelica class for each structured component instance that is part of a CIC. The template for creating the external interface is the same as for an atomic component instance. That means, we apply the template that Figure 3.13 shows also for structured component instances. Nevertheless, the internal structure and the delegation of signals and messages is different.

Figure 3.15 shows the generation template for creating the internal structure of a structured component instance. We declare a variable for each component instance that is part of the structured component instance. Therefore, we take the embedded CIC of the structured component instance and get all contained component instances. The instance variable gets the name of the component instance and is of the type ComponentInstanceName.ComponentInstanceName. Furthermore, we create for each contained assembly connector instance of the structured component instance a Modelica connect equation between the connectors of the containing Modelica objects. Afterward, we create for contained delegations connect equations between the connectors of the structured component instance class and the connectors of the contained Modelica objects.

Figure 3.16 shows an example of the generation of a Modelica class for a MechatronicUML structured component instance that consists of two discrete components instances and four continuous component instances. The structure represents the software architecture of the overtakee car.

In the upper part, Figure 3.16 shows the MechatronicUML structured component instance overtakee that contains the discrete component instances overtakeeCommunicator and overtakeeDriver. The component instance overtakeeCommunicator can send messages to the component instance overtakeeDriver and can communicate with other systems via its delegation ports. Furthermore, the structured component instance overtakee contains the continuous component instances overtakeeColor, overtakeeDistance, overtakeeMotorL, and overtakeeMotorR. The continuous component instance overtakeeColor is assembled with the discrete component instance overtakeeCommunicator and can send a color signal value. The continuous component instance overtakeeDistance is assembled with the discrete component instance overtakeeDriver and provides a distance signal. Lastly, the discrete component instance overtakeeDriver is assembled with a left and a right motor component instance and provides the reference velocity signal to these motors.

In the lower part, Figure 3.16 shows the generated Modelica class overtakee. In this example, we focus on the generation of the delegation connectors and instantiation of the contained components. For each sender message interface and receiver message interface of all delegation

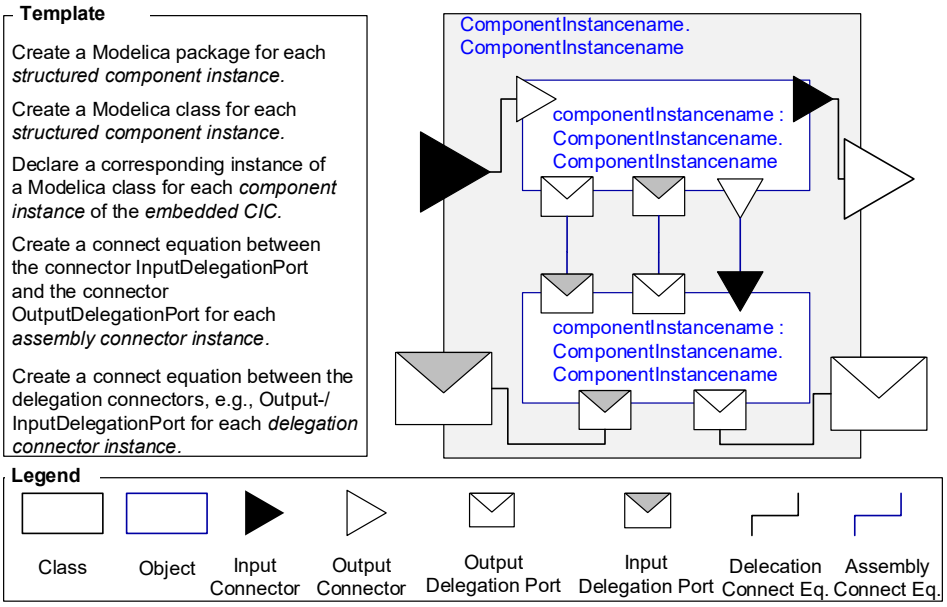


Figure 3.15: Generation Template for Creating the Internal Structure of a Modelica Class for a Structured Component Instance

ports, we generate a corresponding delegation connector. The message interfaces are the same as for the atomic component overtakeeCommunicator as only this component instance has delegation ports. Furthermore, we instantiate for each component instance a corresponding Modelica object and connect their connectors of the assembled component instances. Lastly, we connect the delegation connectors.

3.4.2 Transforming RTSCs to Modelica Models

This section describes the transformation of a component RTSC into a Modelica connection diagram that is based on StateGraph2 (cf. Section 3.1.2) and our Real-Time Coordination Modelica library (cf. Section 3.3). A root RTSC that is the source of the transformation can contain a composite state with multiple orthogonal regions that refer to further RTSCs. The output, a Modelica connection diagram, contains multiple Modelica objects and their corresponding classes. The Modelica object hierarchy represents the structure and behavior of the RTSCs.

The next section defines how to transform the basic parts of each RTSC. Afterward, Section 3.4.2.2 specifies the transformation of hierarchical RTSCs with entry points and exit points. Following, Section 3.4.2.3 explains how RTSCs with synchronized orthogonal behavior have to be transformed. Then, Section 3.4.2.4 enhance the RTSC transformation concept by defining how to transform message-based communication. Subsequently, Section 3.4.2.5 specifies the transformation of real-time elements. Finally, Section 3.4.2.6 sets the definition of how the transformation handles actions that an RTSC can invoke.

3.4.2.1 Basic RTSCs

We create a Modelica package and a contained Modelica class for the RTSC that is referenced as the behavior of a discrete atomic component. This class inherits the class parallel from StateGraph2. "The states of RTSCs can be hierarchical and can contain several parallel subordinate state machines, which refer to a new state machine. Therefore, at the Modelica

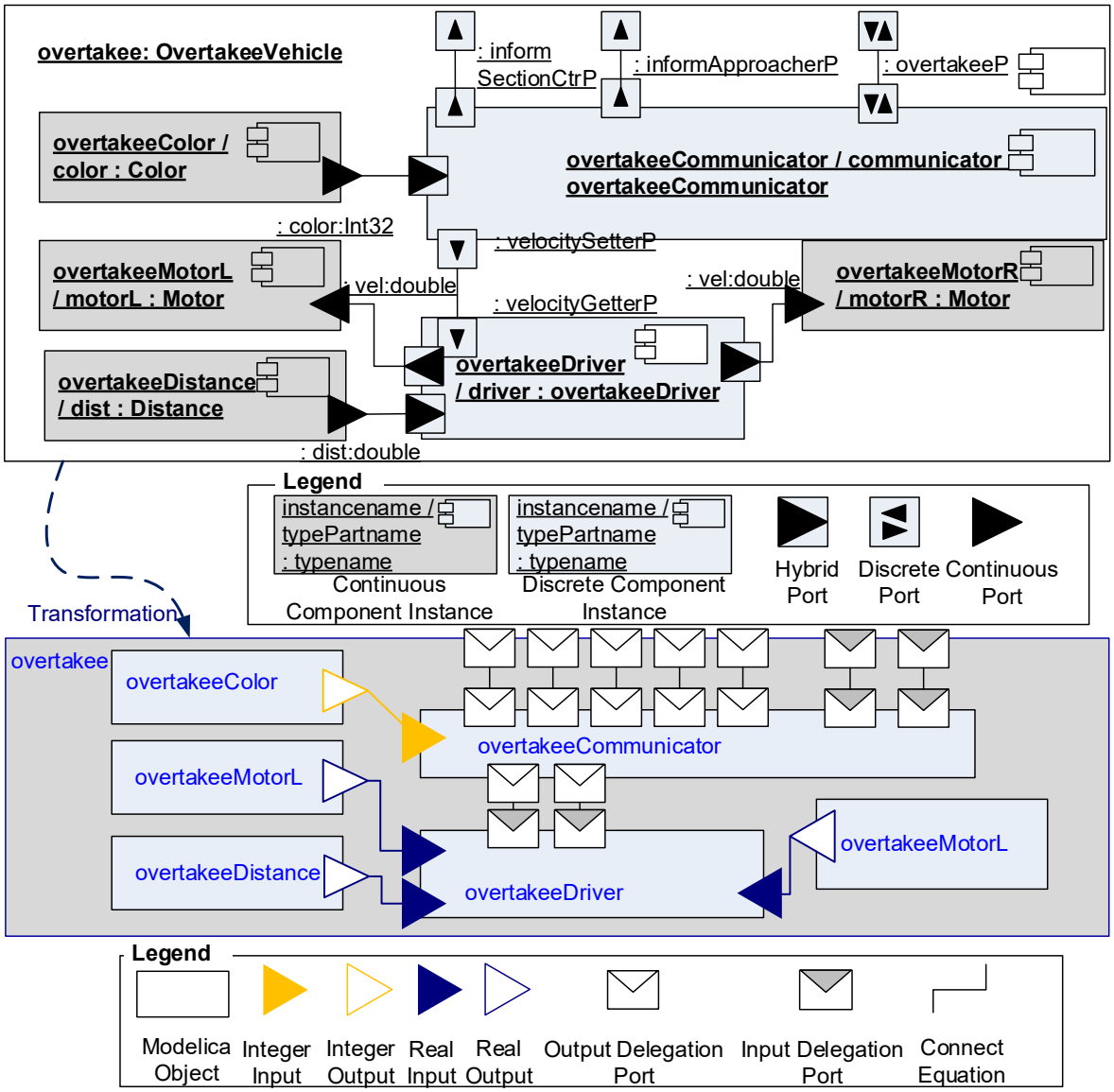


Figure 3.16: Transformation of a MechatronicUML Structured Component Instance into a Modelica Connection Diagram

level, we obtain the following recursive tree structure of Modelica [packages and] classes: 'StateMachine–States–StateMachine'." [*PHM+14] The classes for the subordinate state machines inherit the class partialParallel from StateGraph2. The only difference between partialParallel and parallel is that partialParallel has to be instantiated and cannot run by itself.

Figure 3.17 shows the basic generation template for creating a representation of RTSCs in Modelica. For simplification, we do not show the generation of synchronizations, invariants, clock constraints, messages, and mailboxes here. They are described in the following sections. All the elements that are created in Modelica use elements from the Modelica standard library, the StateGraph2 library, and our Real-Time Coordination library (cf. Section 3.3). We declare for each hybrid in-port instance an input connector and for each hybrid out-port instance an output connector. Thereby, it is possible to read the value of the hybrid in-port and to write new values to the out-port. Thereafter, we declare variables and create objects for all embedded elements. We declare partialParallel objects for composite states and step objects for

simple states. The initial state of a state machine is marked by setting the parameter initalStep to true.

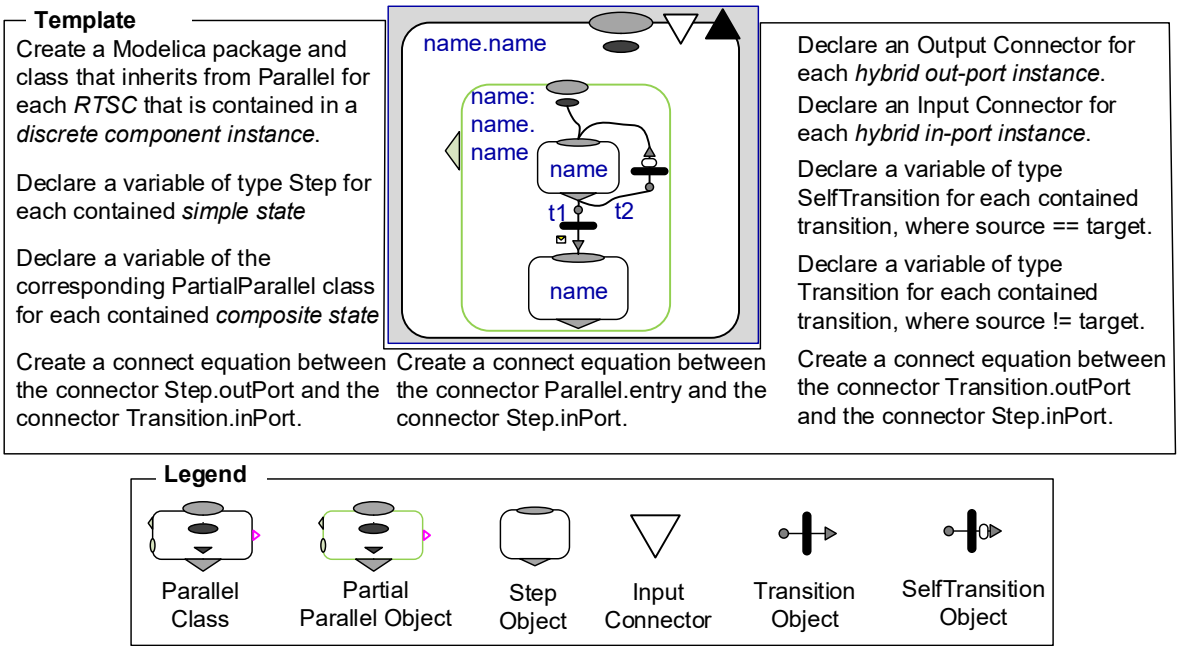


Figure 3.17: Basic Generation Template for Creating a Modelica Class for a Real-Time Statechart

Next, we create transition objects for transitions where the target is different from the source, and we create selfTransition objects for transitions where the target and source are equal. We use the basic transition generation template that Listing 3.3 shows. Each transition gets a name that relates to its source state, its priority, and its target name. Furthermore, we delay each transition by using the shortest possible Modelica time span because in MechatronicUML it is not allowed that more than one transition in the same region fires at the same time instance. We set the parameter use_after to true and set the afterTime to the value $10^{(-6)}$ seconds. Thereby, we also prevent endless loops in Modelica. Transition guards are transformed into a transition guard condition directly. The getGuardsExpression call encapsulates all guard variables with the Modelica statement *pre*, which prevents algebraic loops and conforms to the MechatronicUML semantics. The basic template in Listing 3.3 omits the properties that are required for entry points, exit points, synchronizations, message-based communication, and real-time behavior. These transition properties are described in the following sections.

```
1  RealTimeCoordinationLibrary.RealTimeCoordination.Transition
     transition_[source.name/]+'_' + transition.priority+'_'[target.name/](
     use_after=true,
      afterTime=10^(-6),
5     [if not transition.guard.oclIsUndefined()]
        condition = [transition.guard.getGuardExpression()/]
     [/if]
```

Listing 3.3: Basic Declaration Template for a Transition

Subsequently, we connect the states with *simple transitions* and *high-level transitions*. We call transitions *simple* if the source and the target state have no hierarchy and *high-level* if the source or the target is a *composite state* that has further hierarchical levels. The

MechatronicUML semantics defines that high-level transitions that sources are on a higher level have a higher priority than a transition that source is on a lower level. Note that alternative statechart semantics, like the run-to-completion semantics of UML state machines [UML; HK04] behave differently and that our transformation specification of MechatronicUML real-time statecharts to Modelica cannot be used without any changes for other statechart approaches.

A *simple transition* is connected by generating connect equations that connect the outPort of a step object to the inPort of a transition object and the outPort of a transition object to the inPort of a step object. Furthermore, MechatronicUML defines priorities for multiple outgoing transitions of a state. Only the transition with the highest priority fires, when multiple transitions of a state can fire. MechatronicUML defines that a higher value reflects a higher priority. Modelica defines the priority of outgoing transitions via the position of a connect equation within the outPort[1..n] array of a step object, where n represents the number of outgoing transitions. Modelica defines that a lower value reflects a higher priority. Therefore, we transform the priority (priority(i)) of the transition i to the priority outPort[n-priority(i)].

A *high-level transition* that has a composite state as target is connected by generating the same connect equations as for simple transitions. Furthermore, the hierarchical levels have to be connected. MechatronicUML defines priorities for multiple regions of a state. The priority defines a deterministic execution order for orthogonal regions; a higher value reflects a higher priority. In Modelica, regions are represented by (partial)parallel objects. The execution order of orthogonal (partial)parallel objects is defined via the position of a connect equation of a inPort within the entry[1..n] array of the parent (partial)parallel object, where n represents the number of regions. Modelica defines that a lower value reflects a higher priority. Therefore, we generate connect equations that connect the entry port of the (partial)parallel object with the inPorts of the embedded initial states. The priority of region i (priority(i)) is transformed to entry[n-priority(i)].

A *high-level transition* that has a composite state as source is connected by generating connect equations that connect the suspend port of a (partial)parallel object to the inPort of a transition object and the outPort of a transition object to the inPort of a step object. The connection with the suspend port reflects the MechatronicUML semantics that high-level transitions have a higher priority because the StateGraph2 library privileges transitions that are connected to suspend ports toward transitions that are connected to simple states (cf. Section 3.3.4).

Figure 3.18 shows an example of the generation of a Modelica class for an RTSC. The behavior represents an alternative version of the OvertakeeCommunicator behavior.

In the left part, Figure 3.18 shows an RTSC that consists of the two states inactiveCommunicator and activeCommunicator in the upper level. The statechart can switch between both states via *high-level transitions*. The state activeCommunicator is a composite state as it consists of the two regions main and monitor. The region main consists of three simple states and five transitions and the region monitor consists of two simple states and two transitions.

In the right part, Figure 3.18 shows the generated Modelica class. In this example, we focus on the generation of (hierarchical) states and transitions. The simple states inactiveCommunicator, init, requested, overtaking, arbitrarySpeed, and noSpeedUp are transformed into Modelica step objects. The composite state activeCommunicator and the regions main and monitor into (partial)parallel objects. Furthermore, Figure 3.18 shows the nine generated transition objects. We renumber the transition names consecutively for better readability.

States and transitions are connected via Modelica connect equations. For example, for the simple transition t1 between the states init and requested, the connect equations `connect(inactiveCommunicatoroutPort[1],t1.inPort[1])`, `connect(t1.outPort[1],requested.inPort[1])` are generated. For multiple outgoing

transitions the transition priority has to be reflected by the position in the outPort array. For example, the transitions t3 has a higher priority than the transition t2. We generate the connect equations `connect(requested.outPort[2],t2.inPort[1])`, `connect(requested.outPort[1],t3.inPort[1])` to get this semantics.

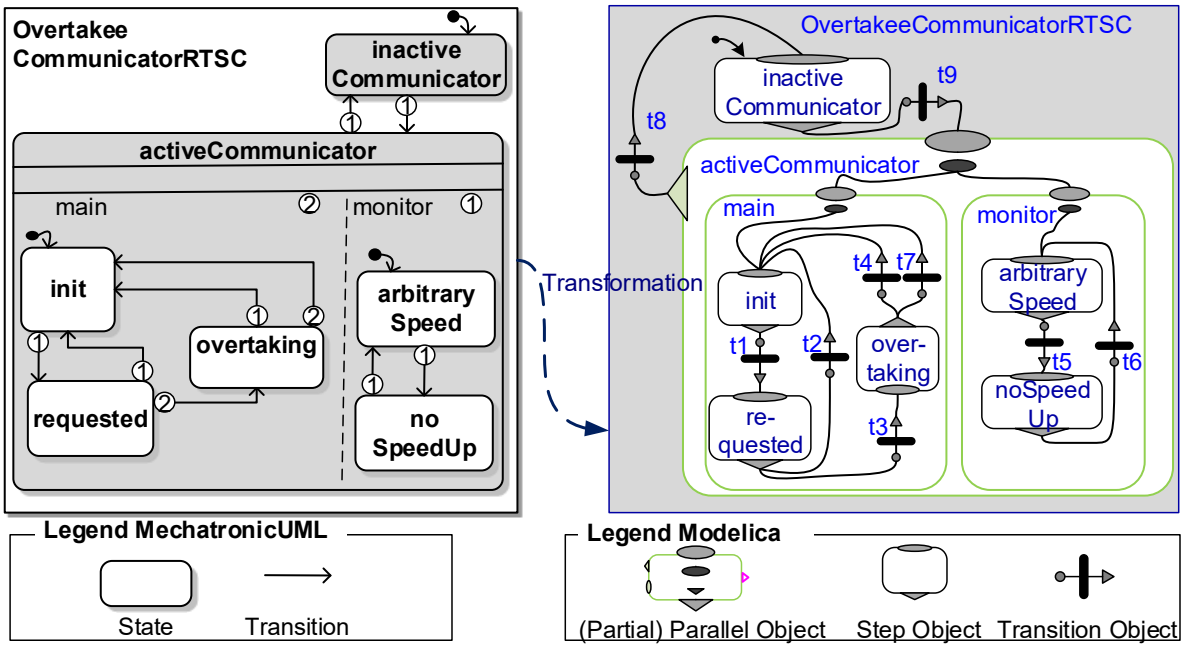


Figure 3.18: Transformation of a MechatronicUML Real-Time Statechart to a Modelica Class

The transition t9 is an example of connect equations for a *high-level transition* with a composite state as *target*. Similar to simple transitions, the connect equations `connect(inactiveCommunicator.outPort[1], t9.inPort[1])`, `connect(t9.outPort[1], activeCommunicator.inPort[1])` are generated for the high-level transition with the target composite state activeCommunicator. Furthermore, for this high-level transition the connect equations `connect(activeCommunicator.entry[1], main.inPort[1])`, `connect(activeCommunicator.entry[2], monitor.inPort[1])` and `connect(main.entry[1], init.inPort[1])`, `connect(monitor.entry[1], ArbitrarySpeed.inPort[1])` are generated to activate both regions and their initial states in the correct order.

The transition t8 is an example of connect equations for a *high-level transition* with a composite state as *source*. The connect equations `connect(activeCommunicator.suspend[1], t8.inPort[1])`, `connect(t8.outPort, inactiveCommunicator.inPort[1])` are generated for the high-level transition with the source composite state activeCommunicator. Instead of the outPort it uses the suspend port for connecting the source state to a transition. As a result, the transition t8 has a higher priority than the transitions t1-t7.

3.4.2.2 Hierarchical RTSCs with Entry Points and Exit Points

We create a Modelica package and a contained Modelica class for RTSCs with entry points or exit points by using the basic generation template (cf. Figure 3.17). An entry point can be used to activate a composite state and explicit states that are contained in the orthogonal regions. An entry point is an alternative for a high-level transition, where the contained initial states of the orthogonal regions are activated. An exit point can be used to deactivate a

composite state, if an explicit state combination is active and certain conditions are fulfilled. The Modelica parallel class does not support special connectors that reflect the entry points or exit points semantics. Therefore, we have to transform the RTSC to get the entry point and exit point semantics.

Figure 3.19 shows the generation template for creating a representation of RTSCs with entry points in Modelica. We create transition objects for transitions where the target is an entry point. Additionally, we connect it the same way as a high-level transition that has a composite state as target. Furthermore, we create a new inner entry point variable of the type Boolean for each entry point. The variable gets the name that is concatenated of the state name of an entry point and the name of the entry point. We set this variable to true by a Modelica *when* statement in the case when the transition with the entry point as target fires. We set this variable to false within a Modelica *when-clause* in the case when the composite state of the entry point is deactivated. Next, we create a step object named 'generatedInit' for the statechart of each region. This step object becomes the new initial state by being connected to the entry connector of the composite state. We declare a variable of the type transition for each transition, where the source is an entry point. The entry point variable is set as guard condition of this transition object. As a result, the transition only fires if the state is entered via the entry point. Lastly, we declare a new transition object for each initial state of a region and connect it to the default step object with the lowest priority. This transition always fires and activates the initial state when the parent state is activated by a high-level transition.

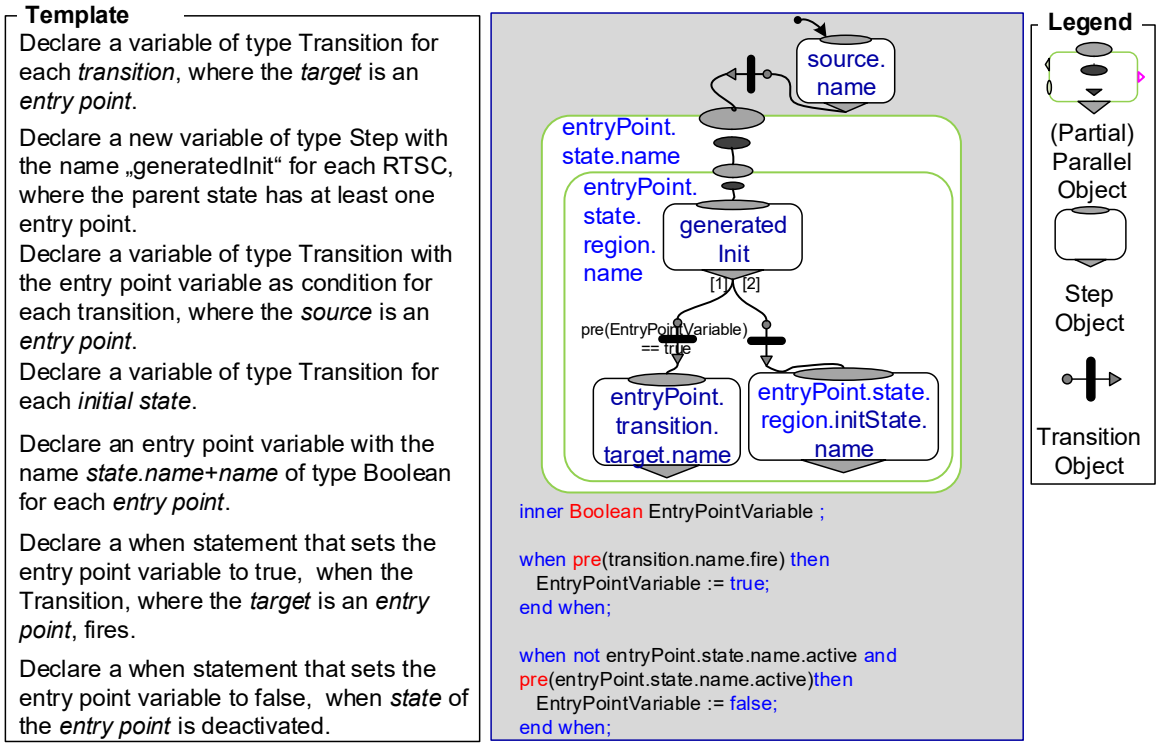


Figure 3.19: Generation Template for Extending a Modelica Class for a Real-Time Statechart with Entry Points

Figure 3.20 shows an example of the generation of a Modelica class for an RTSC with an entry point. The behavior represents an alternative version of the OvertakeeCommunicator behavior (cf. Figure 3.18).

In the left part, Figure 3.20 shows an RTSC that consists of the two states inactiveCommunicator and activeCommunicator in the upper level. The state machine can switch from the state

inactiveCommunicator to the state activeCommunicator via the entry point that is named EntryPoint and can switch from the state activeCommunicator to the state inactiveCommunicator via a high-level transition. The state activeCommunicator is a composite state as it consists of the two regions main and monitor. The region main shows the initial state init and the transition from the entry point to the state init. The region monitor shows the initial state arbitrarySpeed and the transition from the entry point to the state arbitrarySpeed.

In the right part, Figure 3.20 shows the generated Modelica class. In this example, we focus on the generation of relevant states and transitions for entry points. The simple states inactiveCommunicator, init, and arbitrarySpeed are transformed into Modelica step objects and the composite state activeCommunicator and the regions main and monitor into (partial)parallel objects by the basic generation template (cf. Figure 3.17). The high-level transition from the state activeCommunicator to the state inactiveCommunicator is also transformed by the basic generation template to the transition object t8.

Furthermore, Figure 3.20 shows both step objects generatedInit that are added for both regions by the entry point generation template (cf. Figure 3.19). It shows also the generated *entry transition objects* t10 and t12 with the guard condition pre(activeCommunicatorEntryPoint) and the *initial transition objects* t11 and t13. The *entry transition objects* activate init in the case when the transition t9 fires. The *initial transition objects* activate init if activeCommunicator is activated by any other equation.

The lower site of Figure 3.20 shows generated Modelica (algorithmic) code. Modelica defines only textual syntax for (algorithmic) code and no graphical annotations. The code declares the Boolean entry point variable activeCommunicatorEntryPoint. The variable gets the attribute inner, which enables elements that are defined on a lower level to read this variable. The *when-clause* with the condition pre(t9.fire) sets the entry point variable to true when the transition t9 fires. As a result, the guard condition of transitions t10 and t12 is true. The *when-clause* with the condition not activeCommunicator.active and pre(activeCommunicator.active) sets the entry point variable to false when the composite state activeCommunicator is deactivated, e.g., if the transition t8 fires.

Figure 3.21 shows the generation template for creating a representation of RTSCs with exit points in Modelica. We create transition objects for transitions, where the source is an exit point and connect them the same way as a high-level transition that has a composite state as its source. Furthermore, we create a new inner exit point variable of the type Integer for each exit point. The variable gets the name that is concatenated of the state name of an exit point and the name of the exit point. We set this variable to zero within a Modelica *when-clause* in the case when the composite state of the exit point is activated. Next, we create an *exit point step* object by the name of the exit point for the statechart of each region. We declare two variables of the type transition for each transition of each region, where the target is an exit point. The transitions connect the source state of a transition, where the target is an exit point and the exit point step object. The exit point variable is incremented within a Modelica *when-clause* in the case when an exit point step object is activated. The exit point variable is decremented within a Modelica *when-clause* in the case when an exit point step object is deactivated. The exit point variable is set as guard condition of the corresponding transition object, where the source is an exit point. As a result, the transition only fires if the exit point variable is equal to the size of the incoming transitions of the exit point. Therefore, each incoming transition of the exit point has fired.

Figure 3.22 shows an example of the generation of a Modelica class for an RTSC with an exit point. The behavior represents an alternative version of the OvertakeeCommunicator behavior (cf. Figure 3.18).

In the left part, Figure 3.22 shows an RTSC that consists of the two states inactiveCommunicator and activeCommunicator in the upper level. The state machine can switch from the state

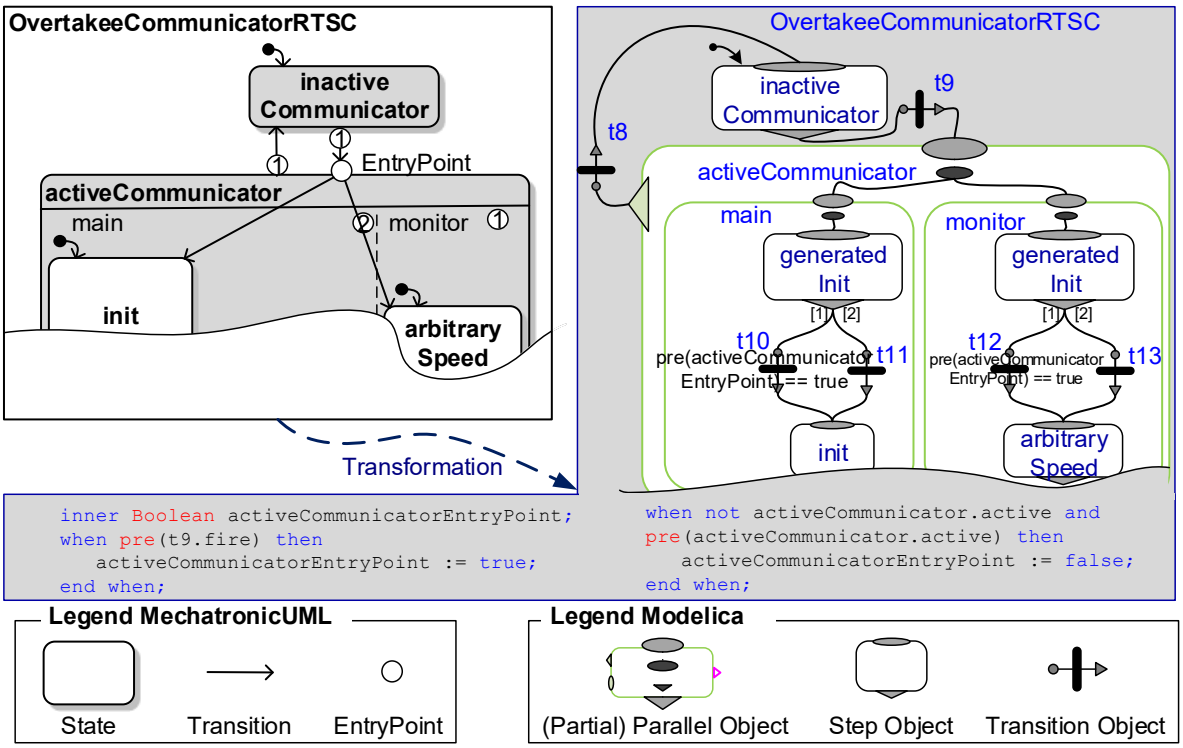


Figure 3.20: Transformation of a MechatronicUML Real-Time Statechart with an Entry Point to a Modelica Class

inactiveCommunicator to the state activeCommunicator via a high-level transition and can switch from the state activeCommunicator to the state inactiveCommunicator via the exit point by the name of the ExitPoint. The state activeCommunicator is a composite state as it consists of the two regions main and monitor. The region main shows the initial state init and the transition from the state init to the exit point by the name of the ExitPoint. The region monitor shows the initial state arbitrarySpeed and the transition from the state arbitrarySpeed to the exit point by the name of the ExitPoint.

In the right part, Figure 3.22 shows the generated Modelica class. In this example, we focus on the generation of relevant states and transitions for exit points. The simple states inactiveCommunicator, init, and arbitrarySpeed are transformed into Modelica step objects and the composite state activeCommunicator and the regions main and monitor into (partial)parallel objects by the basic generation template (cf. Figure 3.17). The high-level transition from the state inactiveCommunicator to the state activeCommunicator is also transformed by the basic generation template to the transition object t9.

Furthermore, Figure 3.22 shows both ExitPoint step objects that are added for both regions by the exit point generation template (cf. Figure 3.21). It shows also the generated transition objects t10 and t12 that activate the ExitPoint step objects. The transition objects t11 and t13 are added to prevent a deadlock for the case that not all exit point steps are activated. The transition object t8 is generated for the transition, where the source is an exit point. The guard condition is set to pre(activeCommunicatorExitPoint==2). This means, it fires only when the variable activeCommunicatorExitPoint is equal to two, which should be true when both ExitPoint step objects are active.

The lower site of Figure 3.22 shows the generated Modelica (algorithmic) code. The code declares the Integer exit point variable activeCommunicatorExitPoint. The variable gets the attribute inner, which enables elements that are defined on a lower level to read this variable.

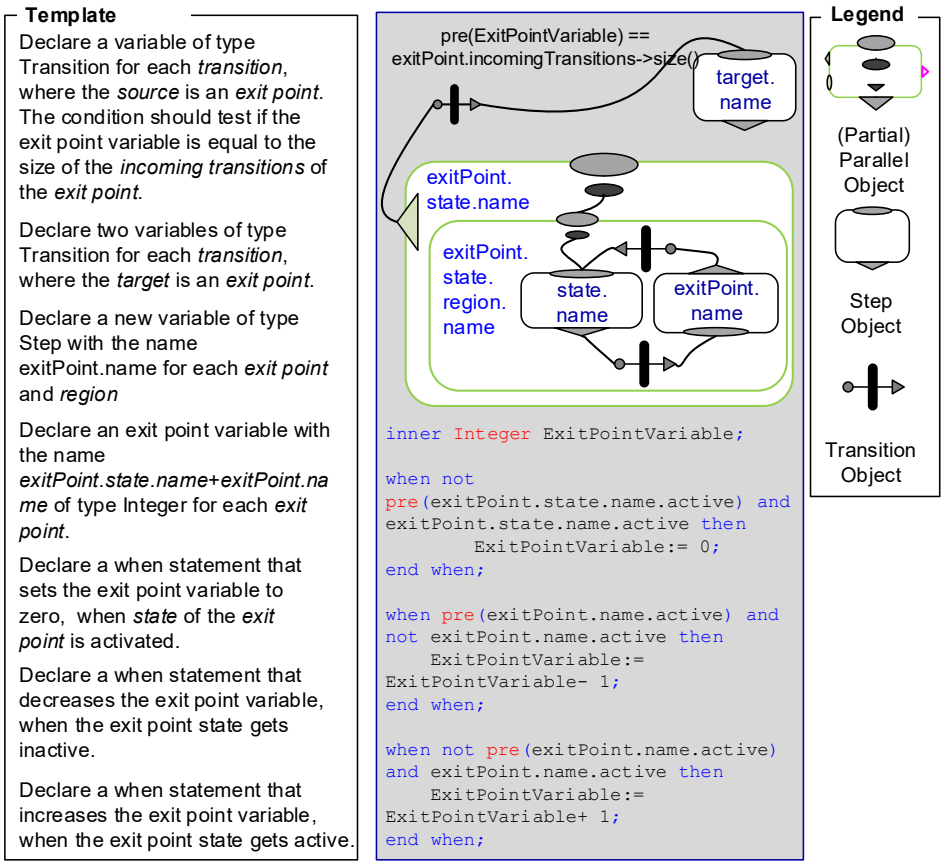


Figure 3.21: Generation Template for Extending a Modelica Class for a Real-Time Statechart with Exit Points

The *when-clause* with the condition not pre(activeCommunicator.active) and activeCommunicator.active sets the exit point variable to zero when the state activeCommunicator is activated. The *when-clause* with the condition not pre(activeCommunicator.[main | monitor].ExitPoint.active) and activeCommunicator.[main | monitor].ExitPoint.active) increments the exit point variable when an ExitPoint step object is activated. The *when-clause* with the condition pre(activeCommunicator.[main | monitor].ExitPoint.active) and not activeCommunicator.[main | monitor].ExitPoint.active) decrements the exit point variable when an ExitPoint step object is deactivated. As a result, the transition t8 fires only when both ExitPoint step objects are active at the same time.

3.4.2.3 Synchronized RTSCs with Orthogonal Behavior

We create a Modelica package and a contained Modelica class for RTSCs with synchronization channels by using the basic generation template (cf. Figure 3.17). A synchronization channel belongs to an RTSC and can be used to synchronize the firing of transitions that belong to orthogonal regions. A transition that has a sender synchronization needs another transition from an orthogonal region that has a receiver synchronization of the same synchronization channel type to be able to fire. Both transitions fire jointly in an atomic way if all conditions of both transitions are fulfilled. Synchronizations prevent that a transition fires without a specific synchronization partner. A synchronization channel can have a selector expression of the type Integer. Transitions with a synchronization that have a selector expression can only fire synchronized if the selector expression has the same value during runtime. In contrast to MechatronicUML, it is not possible in Modelica to define a synchronization channel within

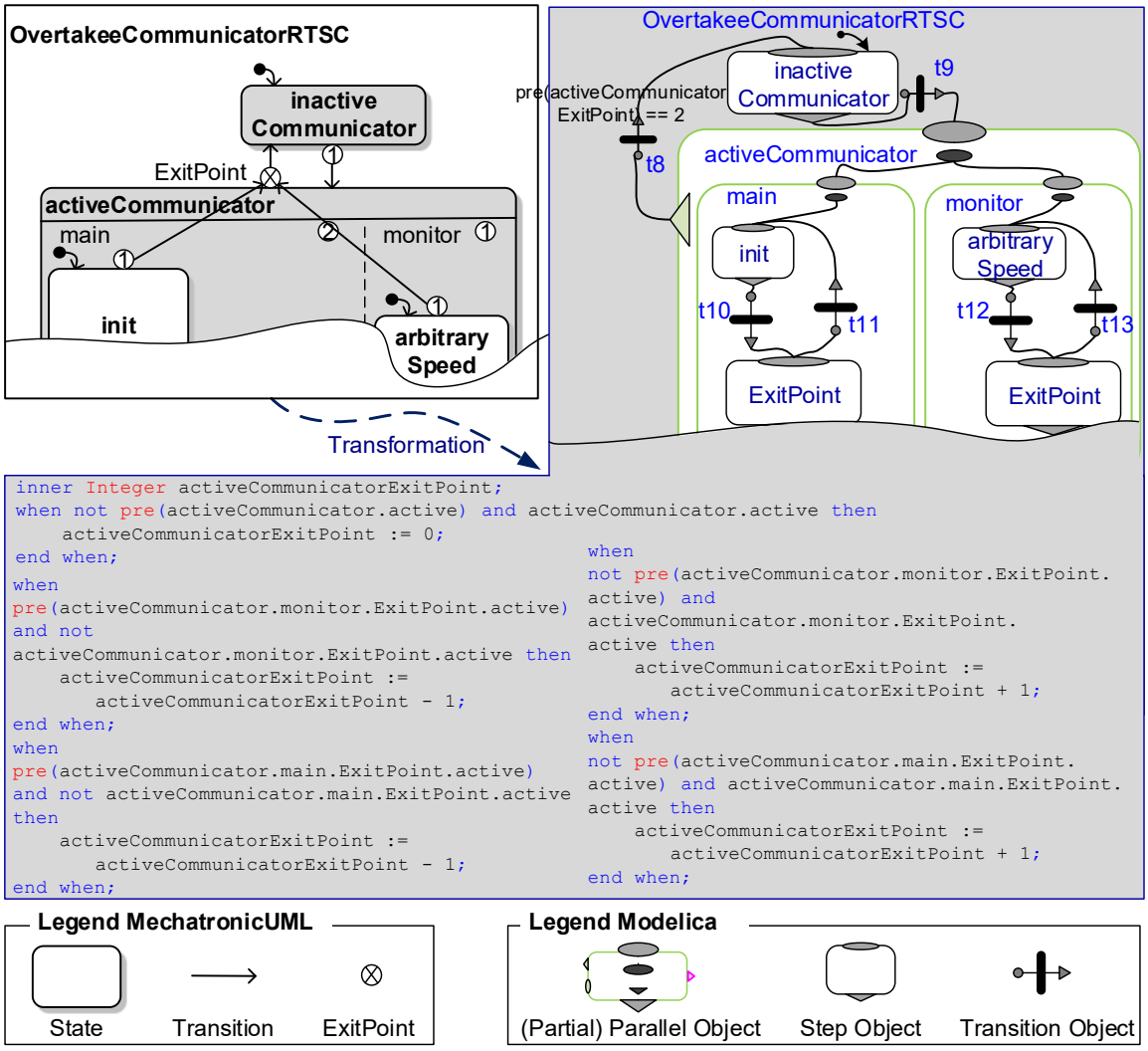


Figure 3.22: Transformation of a MechatronicUML Real-Time Statechart with an Exit Point to a Modelica Class

a parallel class. Instead, the transition class handles synchronizations directly. The transition class of the Real-Time Coordination Modelica library supports to synchronize with other connected transitions. Therefore, we have to transform synchronization channels and their usage to connected transitions.

Listing 3.4 shows the generation template for creating a representation of a transition that uses a synchronization channel with a selector expression in Modelica. We create a transition object that has the connector array *RealTimeCoordination.Internal.Interfaces.Synchron.sender* if it has a synchronization of kind SEND by setting the parameter use_syncSend to true. Furthermore, we set the size of this connector array to the number of transitions that have a corresponding receiver synchronization. We also transfer the name of the synchronization channel to the parameter synchChannelName. In the opposite, we create a transition object that has the connector array *RealTimeCoordination.Internal.Interfaces.Synchron.receiver* if it has a synchronization of kind RECEIVER by setting the parameter use_syncReceive to true. Furthermore, we set the size of this connector array to the number of transitions that have a corresponding sender synchronization. We also transfer the name of the synchronization channel to the transition parameter synchChannelName. Lastly, we set the Integer parameter

selectorExpression for transitions that use a synchronization channel with a selector. We transform the MechatronicUML selector expression of the synchronization to a corresponding Modelica equation.

```
RealTimeCoordinationLibrary.RealTimeCoordination.Transition
transition_[source.name/]+'_' + transition.priority+'_'[target.name/](
[if transition.synchronization.kind = SynchronizationKind::SEND]
   use_syncSend=true,
   numberOfSyncSend=[transition.synchronization.syncChannel.
      getSyncReceivingTransitions(transition)->size()/],
   syncChannelName=''[transition.synchronization.syncChannel.getName()/]''
[else]
   use_syncReceive=true,
   numberOfSyncReceive=[transition.synchronization.syncChannel.
      getSyncSendingTransitions(transition)->size()/],
   syncChannelName=''[transition.synchronization.syncChannel.getName()/]''
[/if]
[if not transition.synchronization.selectorExpression.oclIsUndefined()]
   selectorExpression=[generateExpression(transition.synchronization.
      selectorExpression)/],
[/if]
```

Listing 3.4: Declaration Template for a Transition with a Synchronization Channel

Listing 3.5 shows the generation template for connecting transitions that use a synchronization channel. In a first step, we search all transitions that use a specific synchronization channel as the sender synchronization and store these transitions in an ordered set named sendingTransitionSet. In the next step, we search for each transition within this set that uses the same channel as the receiver synchronization and store them in an ordered set named receivingTransitionSet. Finally, we create a connect equation for each of these transitions. The connect equation connects the sender connector of the sender transition object with the receiver connector of the receiver transition object. Each connect equation uses a unique position in the sender and the receiver connector array by using the OCL operation indexOf(object:OrderedSet) that returns the position of the *object* in a sequence.

```
[for (ch : SynchronizationChannel | state.channels)]
   [let sendingTransitionSet : OrderedSet(Transition) =
   ch.getSyncSendingTransitions()]
   [for (trans_send : Transition | sendingTransitionSet)]
      [let receivingTransitionSet : OrderedSet(Transition) =
      ch.getSyncReceivingTransitions(trans_send)]
      [for (trans_recv : Transition | receivingTransitionSet)]
         connect([trans_send.getFullyQualifiedName()/].sender[
         '[' +    receivingTransitionSet->indexOf(trans_recv) + ']'/],
         [trans_recv.getFullyQualifiedName()/].receiver[
         '[' + ch.getSyncSendingTransitions(trans_recv)->
         indexOf(trans_send) + ']'/]);
      [/for][/let]
   [/for][/let]
[/for]
```

Listing 3.5: Connect Template for a Transition with Synchronization Channel

Figure 3.23 shows an example of the generation of a Modelica class for an RTSC with two synchronization channels. The behavior represents an alternative version of the OvertakeeCommunicator behavior (cf. Figure 3.18).

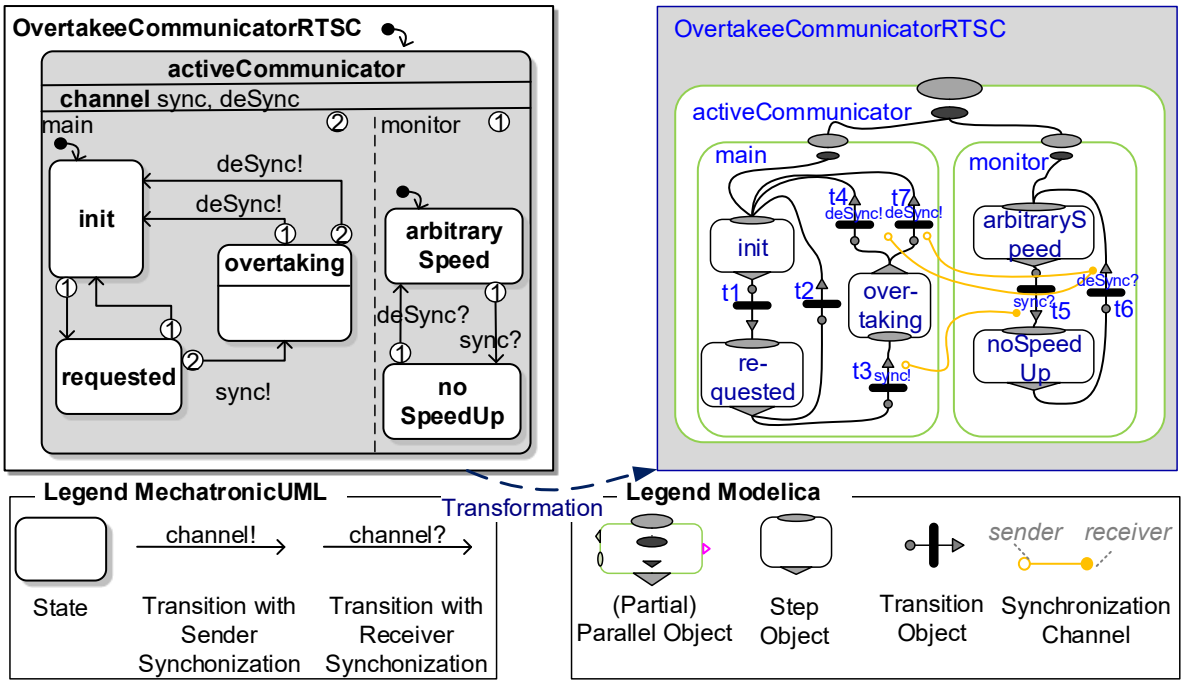


Figure 3.23: Transformation of a MechatronicUML Real-Time Statechart with Synchronization Channel

In the left part, Figure 3.23 shows an RTSC that has the state activeCommunicator in the upper level. The state activeCommunicator is a composite state as it consists of the two regions main and monitor. Furthermore, the state activeCommunicator defines the synchronization channels sync and deSync. The transition from the state requested to the state overtaking uses the synchronization channel sync as sender. The transition from the state arbitrarySpeed to the state noSpeedUp can synchronize with the former transition at runtime because it uses the same synchronization channel as receiver. Both transitions from the state overtaking to the state init use the synchronization channel deSync as sender. The transition from the state noSpeedUp to the state arbitrarySpeed can synchronize at runtime with one of the former transition because it uses the same synchronization channel as receiver.

In the right part, Figure 3.23 shows the generated Modelica class. In this example, we focus on the generation of the synchronization channels. We set the transition parameters by the declaration template for a transition with a synchronization channel (cf. Listing 3.4). The parameter use_syncSend of the transitions t3, t4, and t7 is set to true. Additionally, the parameter use_syncReceive of the transitions t5 and t6 is set to true. The parameter syncChannelName of the transitions t3 and t5 is set to the channel name 'sync'. In the same way, the parameter syncChannelName of the transitions t4, t7, and t6 is set to the channel name 'deSync'. The transition t3 is synchronized with the transition t5 by the connect equation `connect(t3.sender[1],t5.receiver[1])`. The transition t6 is synchronized with the transition t4 or t7 by the connect equations `connect(t4.sender[1],t6.receiver[1])`, `connect(t7.sender[1],t6.receiver[2])`.

3.4.2.4 RTSCs with Message-Based Communication

We create a Modelica package and a contained Modelica class for RTSCs with trigger and raise message events by using the basic generation template (cf. Figure 3.17). A raise message event belongs to a transition and can be used to communicate with another transition of

another software component. It has a type, which defines how many parameters it contains. A raise message event is generated when the transition that defines the raise message event fires. The parameter binding binds parameters to a concrete value when the transition fires. The message event is then transferred to the receiver component, where it is enqueued in a message queue. The message queue has a fixed size. A transition with a trigger message event consumes the message event when it fires. It can only fire, if the message is in the queue and all other conditions are fulfilled. A message event can only be consumed once. In Modelica the message queue is represented by instances of the mailbox class. Message types are represented by instances of the message class. The transition class of the Real-Time Coordination Modelica library supports to generate raise message events and to consume trigger message events. Therefore, we have to transform messages and their usage to connected transitions. The connection is done via message objects, delegation ports, and mailbox objects.

Figure 3.24 shows the generation template for creating a representation of RTSCs with raise messages, message queues, and trigger messages in Modelica. We create a mailbox object for each message type that is used as a trigger message at a transition.

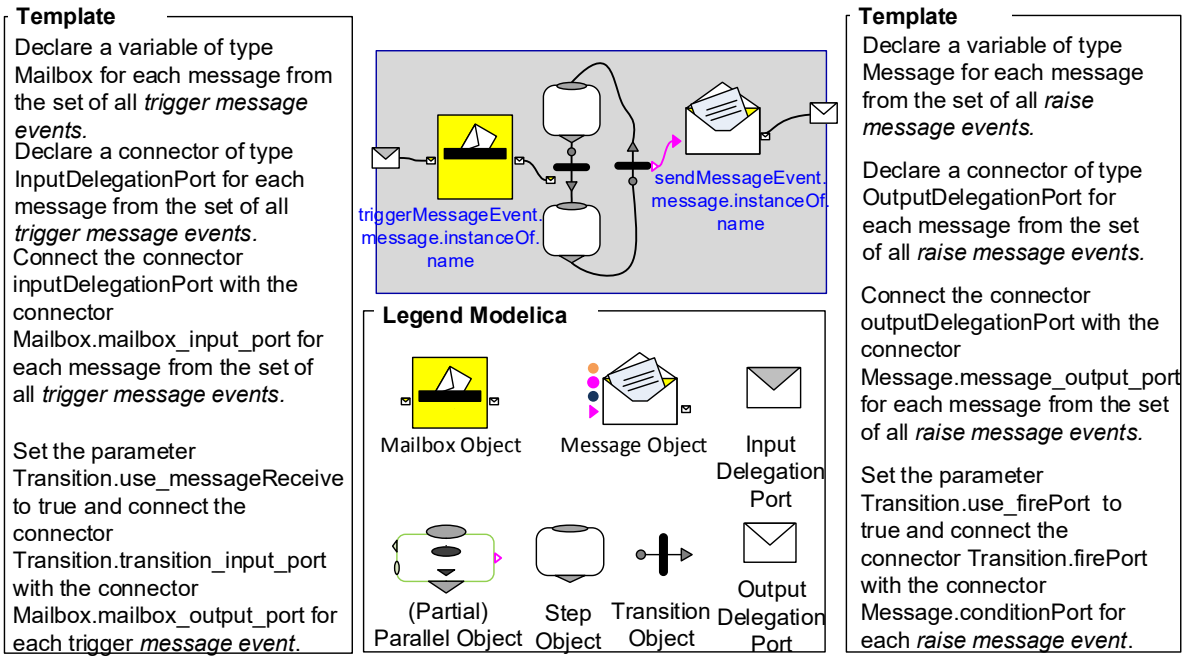


Figure 3.24: Generation Template for Extending a Modelica Class for a Real-Time Statechart with Trigger and Raise Message Events

Listing 3.6 shows the declaration template for a mailbox. It is declared by the name of the message type. The size of the mailbox_input_port (nIn) is set to one, as it is only connected to one delegation port. The size of the mailbox_output_port (nOut) is set to the number of transitions that use the message as a trigger message event. The size of the mailbox queue (queueSize) is set to the size of the receiver message buffer that belongs to a discrete port in MechatronicUML. Furthermore, we have to set the number of Integer, Boolean, and Real parameters (numberOfMessageIntegers, numberOfMessageReals, numberOfMessageBooleans). They are determined by the message type. Additionally, we declare a connector of the type InputDelegationPort for each message from the set of all trigger message events. We use a similar declaration template as for the atomic components (cf. Listing 3.2). Next, we connect the delegation port with the mailbox input port. Message events that are queued in the mailbox can be consumed by transitions that have a trigger message event.

```
[for (ev : AsynchronousMessageEvent | events->getAsyncMessageEventRepr())]
    RealTimeCoordinationLibrary.RealTimeCoordination.Mailbox
     [ev.message.instanceOf.name](
    nIn=1,
    nOut=[ev.getAsyncMessageEventEquivalenceClass(events)->size()/],
    queueSize=[rtsc.behavioralElement.oclAsType(DiscretePort).
    receiverMessageBuffer->select(buf:MessageBuffer|
    buf.messageType=ev.message.instanceOf)->at(1).bufferSize.value],
    numberOfMessageIntegers=[ev.getIntegerParameters()->size()/],
    numberOfMessageReals=[ev.getRealParameters()->size()/],
    numberOfMessageBooleans=[ev.getBooleanParameters()->size()/]);
[/for]
```

Listing 3.6: Declaration Template for a Mailbox and Parameter Variables

Listing 3.7 shows the generation template for creating a representation of a transition with a trigger message event in Modelica. We create a transition object that has the connector array *RealTimeCoordination.Internal.Interfaces.Asynchron.transition_input_port* by setting the parameter use_messageReceive to true. We set the size of this connector array (numberOfMessageReceive) to one as MechatronicUML transitions can only be triggered by a single message. Furthermore, we have to set the number of Integer, Boolean, and Real parameters to the size of the corresponding parameters of the MechatronicUML message type. Lastly, we create for each message parameter a corresponding variable. This variable stores the value of the parameter when the message is consumed.

```
RealTimeCoordinationLibrary.RealTimeCoordination.Transition
transition_[source.name/]+'_' + transition.priority+'_'[target.name/](
[if not transition.triggerMessageEvent.oclIsUndefined()]
    use_messageReceive=true,
    numberOfMessageReceive=1,
    numberOfMessageIntegers=
        [transition.triggerMessageEvent.getIntegerParameters()->size()/],
    numberOfMessageReals=
        [transition.triggerMessageEvent.getRealParameters()->size()/],
    numberOfMessageBooleans=
        [transition.triggerMessageEvent.getBooleanParameters()->size()/]
    [for (param : Parameter |
        transition.triggerMessageEvent.message.instanceOf.parameters)]
        [param.getType()/]    [param.
        getTriggerMessageEventParameterVariableName(
        ev.message.instanceOf)/];
    [/for]
[/if];
```

Listing 3.7: Declaration Template for a Transition with a Trigger Message Event

Listing 3.8 shows the generation template for assigning the trigger message parameter to the corresponding parameter variables. We create a *when-clause* that is executed when the transition with the trigger message event fires. The *when-clause* assigns the value of the message parameter to the corresponding variable. The template differs between Integer, Boolean, and Real parameters because they are stored in different arrays. The assignment of trigger message parameters finishes the consumption of a trigger message. Next, we show the generation for raise message events.

Listing 3.9 shows the declaration template for a message. It is declared by the name of the message type. The size of the input port (nIn) is set to the number of transitions that define the

```
[for (ev : AsynchronousMessageEvent | events)]
  when [ev.getTransition().getFullyQualifiedName()/].fire then
  [let integerParameters : OrderedSet(Parameter) = ev.getIntegerParameters()]
  [for (param : Parameter | integerParameters)]
    [param.getTriggerMessageEventParameterVariableFullyQualifiedName(ev)/]
      :=
    [ev.getTransition().getFullyQualifiedName()/].transition_input_port['[1]
      '/].
    integers['[' + integerParameters->indexOf(param) + ']'/];
  [/for][/let]
  [let realParameters : OrderedSet(Parameter) = ev.getRealParameters()]
  [for (param : Parameter | realParameters)]
    [param.getTriggerMessageEventParameterVariableFullyQualifiedName(ev)/]
      :=
    [ev.getTransition().getFullyQualifiedName()/].transition_input_port['[1]
      '/].
    reals['[' + realParameters->indexOf(param) + ']'/];
  [/for][/let]
  [let booleanParameters : OrderedSet(Parameter) = ev.getBooleanParameters()]
  [for (param : Parameter | booleanParameters)]
    [param.getTriggerMessageEventParameterVariableFullyQualifiedName(ev)/]
      :=
    [ev.getTransition().getFullyQualifiedName()/].transition_input_port['[1]
      '/].
    booleans['[' + booleanParameters->indexOf(param) + ']'/];
  [/for][/let]
  end when;
[/for]
```

Listing 3.8: Template for Assigning Trigger Message Parameter

message type as raise message event. The variables for Integer, Boolean, and Real parameters is set to the size of the corresponding message parameters. Lastly, we create for each parameter a variable that stores the value of the parameter binding when a transition with a raise message event fires. In another step, we declare a connector of the type OutputDelegationPort for each message from the set of all raise message events and connect the message object with the delegation port (cf. Figure 3.24). We use a similar declaration template as for atomic components (cf. Listing 3.2).

```
[for (ev : AsynchronousMessageEvent | events->getAsyncMessageEventRepr())]
  RealTimeCoordinationLibrary.RealTimeCoordination.Message
    [ev.getMessageName()/](
  nIn=[ev.getAsyncMessageEventEquivalenceClass(events)->size() /],
  numberOfMessageIntegers=[ev.getIntegerParameterBindings()->size()/],
  numberOfMessageReals=[ev.getRealParameterBindings()->size()/],
  numberOfMessageBooleans=[ev.getBooleanParameterBindings()->size()/]);
  [for (pb : ParameterBinding | ev.message.parameterBinding)]
    [pb.getType()/] [pb.getRaiseMessageEventPBVariableName(ev)/];
  [/for]
[/for]
```

Listing 3.9: Declaration Template for a Raise Message Event

Listing 3.10 shows the template for the parameter binding. We declare a *when-clause* that is executed when the transition with the raise message event fires. The statement binds the value of the parameter binding to the parameter variable.

```
[for (ev : AsynchronousMessageEvent | events)]
  when [ev.getTransition().getFullyQualifiedName()/].fire then
  [for (pb : ParameterBinding | ev.message.parameterBinding)]
    [pb.getRaiseMessageEventPBVariableFullyQualifiedName(ev)/] :=
      [pb.value.generateExpression(false, ev.getTransition().
      statechart.getFullyQualifiedName())/];
  [/for]
  end when;
[/for]
```

Listing 3.10: Declaration Template for a Raise Message Event Parameter Binding

In a last task, we have to connect the transition object with the corresponding message object for each raise event. Therefore, we set the parameter Transition.use_firePort to true and connect the connector Transition.firePort with the connector Message.conditionPort for each raise message event.

Figure 3.25 shows an example of the generation of a Modelica class for an RTSC with two raise message events and two trigger message events. The message events have no parameter and parameter bindings. The behavior represents an alternative version of the OvertakeeCommunicator behavior (cf. Figure 3.18).

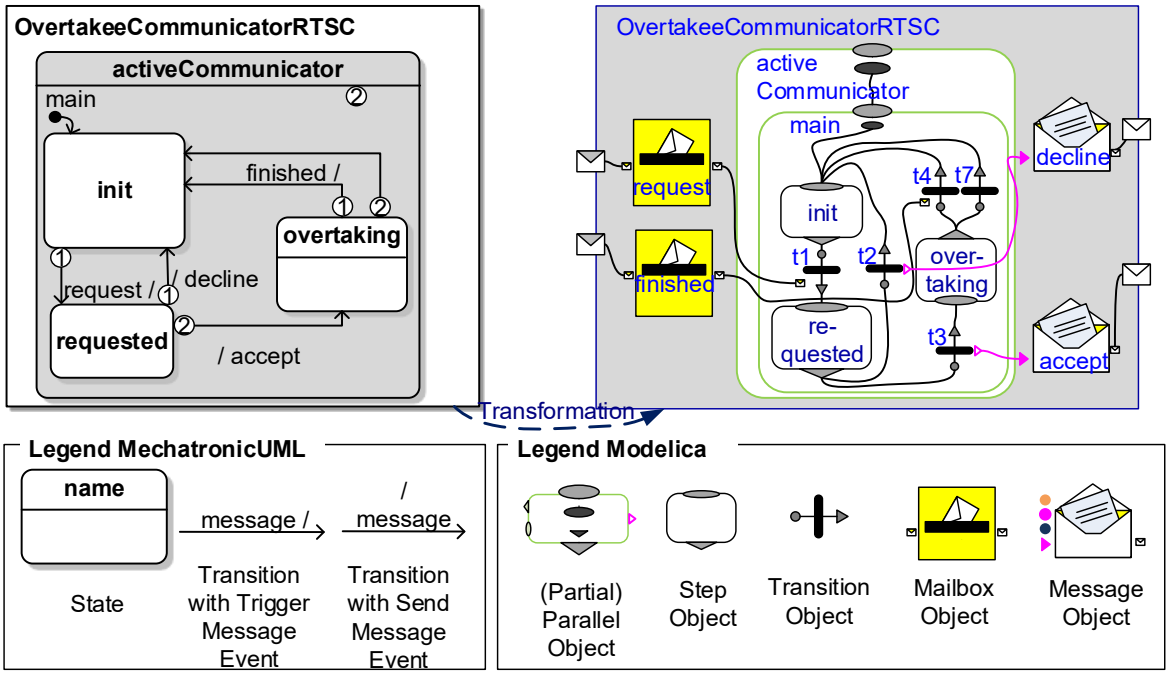


Figure 3.25: Transformation of a MechatronicUML Real-Time Statechart with a Trigger and a Raise Message Events

In the left part, Figure 3.25 shows an RTSC that has the state activeCommunicator in the upper level. The state activeCommunicator is a composite state as it consists of the region main. The transition from the state init to the state requested has the trigger message event request. Furthermore, the transition from the state overtaking to the state init has the trigger message event finished. Moreover, the transition from the state requested to the state init has the raise message event decline and the transition from the state requested to the state overtaking has the raise message event accept.

In the right part, Figure 3.25 shows the generated Modelica class. In this example, we focus on the generation of the message events. The mailbox objects request and finished and corresponding input delegation ports are created for the raise message events. The message objects decline and accept and corresponding output delegation ports are created for the trigger message events. As companion, the connect equations between the objects request and t1, finished and t4, decline and t2, and accept and t3 are created.

3.4.2.5 RTSCs with Real-Time Elements

We create a Modelica package and a contained Modelica class for RTSCs with clocks, clock resets, clock constraints, and time invariants by using the basic generation template (cf. Figure 3.17). A clock belongs to a statechart and can be used to measure the time by its time value. The time value of a clock can be reset to zero by an entry event, exit event, and transition event. Furthermore, a time value has a time unit. Clock constraints can refer to a clock and compare it to a certain bound. The accuracy of the clock constraint depends on the value and step size of the clock value. A transition has clock constraints as its guard to restrict when it is allowed to fire. A state has a clock constraint as its time invariant. The time invariant represent a safety property that has to be always fulfilled during runtime. In Modelica clocks are represented by instances of the clock class that refers to the Modelica built-in variable *time*. The accuracy of the time value during the simulation depends on the chosen solver and the simulation step size [LFF10]. If required the accuracy can be very high with a step size of about a nanosecond [LFF10]. This Modelica build-in variable time has the time unit *seconds* by default [MODELICA]. Therefore, other time units have to be normalized by being multiplied with the corresponding factor, e.g., the factor 10^{-3} for the normalization of milliseconds to seconds. This clock class defines a Boolean connector (reset) that resets the value of the clock whenever the connector reset receives a true signal. Clock constraints of transitions are represented by instances of the ClockConstraint class. They have to be connected to the condition port of the corresponding transition. Time invariants of states are represented by instances of the Invariant class. They have to be connected to the corresponding state to be active. The simulation stops with a meaningful error message if a time invariant gets violated during a simulation run.

Figure 3.26 shows the generation template for creating a representation of RTSCs with clocks, clock resets, clock constraints, and invariants in Modelica. We create a mailbox object for each message type that is used as a trigger message at a transition. We instantiate for each clock of the RTSC a clock object, for each invariant of a state an invariant object, and for each clock constraint of a transition a clock constraint object.

We differ between three kinds of clock resets: (1) clock resets as entry event of a state are transformed into a connect equation between the activePort connector of the corresponding step object and the reset connector of the clock object; (2) Clock resets as exit event of a state are transformed into a logical not object that is connected to the activePort connector of the step object and a connect equation between the inverted output of the not object and the reset connector of the clock object; (3) clock resets as transition event are transformed to a connect equation between the transition firePort connector and the clock reset connector.

We create two connect equations for time invariants. The first, between the clock output connector and the invariant clockValue connector to delegate the clock value. The second, between the step object activePort connector and the invariant conditionPort connector. As a result, the invariant has to be fulfilled only when the connected state is active. Furthermore, we set the bound parameter of the invariant to the corresponding value from MechatronicUML.

Lastly, we create two connect equations for clock constraints of transitions. The first, between the clock output connector and the clock constraint clockValue connector to delegate the

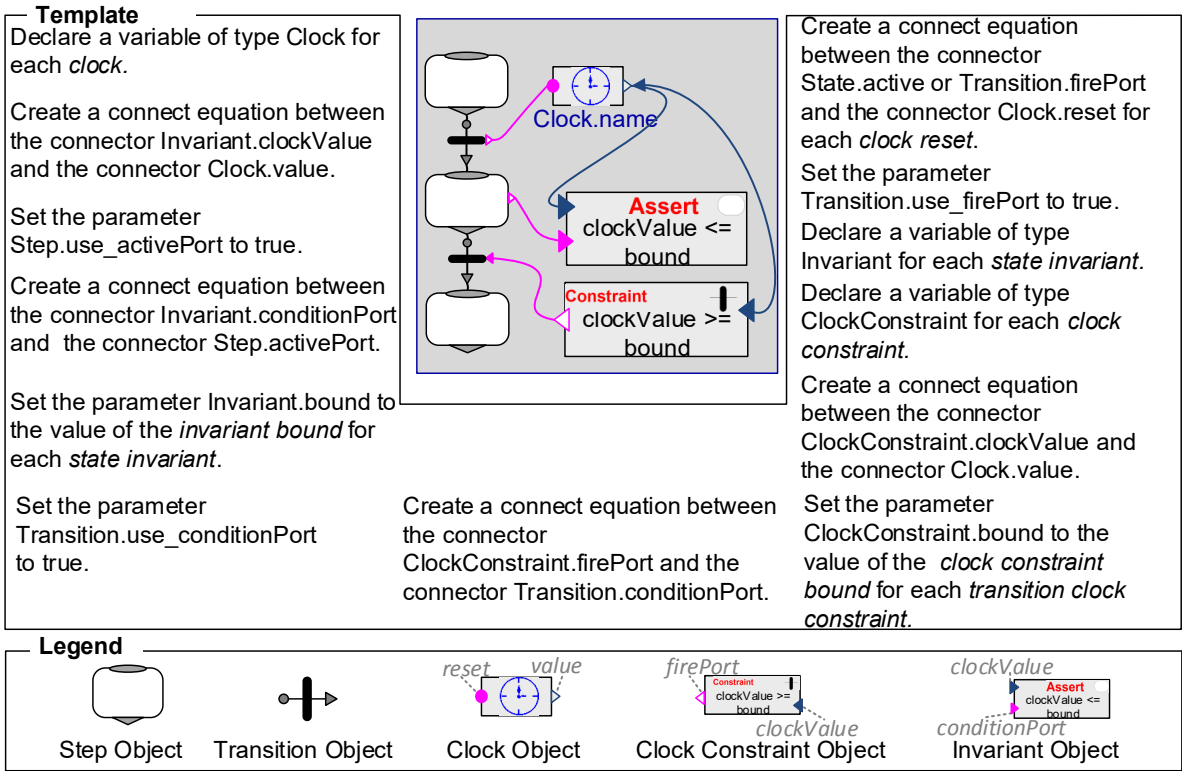


Figure 3.26: Generation Template for Extending a Modelica Class for a Real-Time Statechart with Clocks, Clock Resets, Clock Constraints, and Invariants

clock value. The second, between the transition conditionPort connector and the clock constraint firePort connector. As a result, the transition object can only fire when the value of the firePort connector is true. Furthermore, we set the bound parameter of the clock constraint to the corresponding value from MechatronicUML.

Figure 3.27 shows an example of the generation of a Modelica class for an RTSC with clocks, clock resets, clock constraints, and time invariants. The behavior represents an alternative version of the OvertakeeCommunicator behavior (cf. Figure 3.18).

In the left part, Figure 3.27 shows an RTSC that has the state activeCommunicator in the upper level. The state activeCommunicator is a composite state as it consists of the region main. The statechart of the region main declares the clock timeout. The transition from the state requested to the state overtaking resets the clock timeout. The state overtaking has a time invariant. The time invariant restricts that the value of the clock timeout is greater than 31. Moreover, the transition from the state overtaking to the state init with priority 2 has the clock constraint timeout>30. This transition can only fire when timeout becomes grater than 30 seconds.

In the right part, Figure 3.27 shows the generated Modelica class. In this example, we focus on the generation of the real-time elements.

The clock object timeout is created for the clock. The clock constraint object and the invariant object are created for the corresponding MechatronicUML elements. As companion, the connect equations between the transition object t7 and the clock constraint object and the step object overtaking and the invariant object are created. Additionally, the connect equations between the clock object and the clock constraint object and invariant object are created. Lastly, the bound parameters are set.

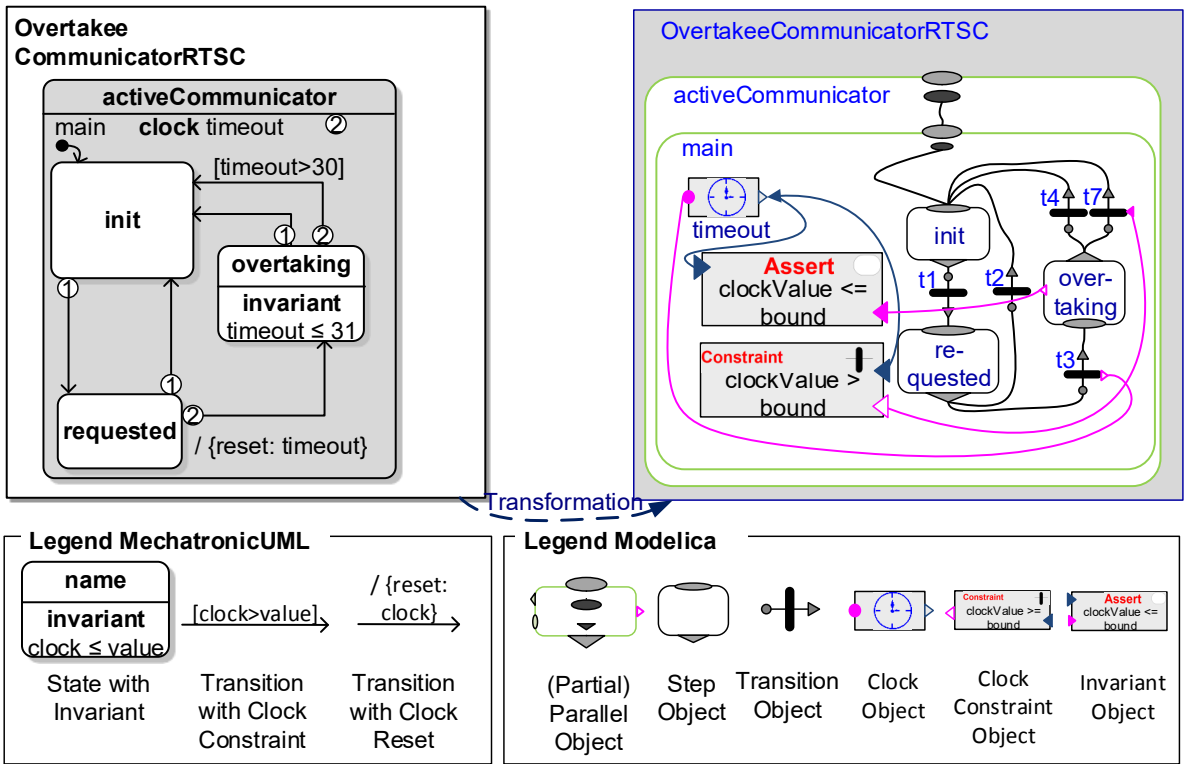


Figure 3.27: Transformation of a MechatronicUML Real-Time Statechart with Clocks, Clock Resets, Invariants, and Clock Constraints

3.4.2.6 RTSCs with Actions

We create a Modelica package and a contained Modelica class for RTSCs with variables, operations, entry events, exit events, and transition events by using the basic generation template (cf. Figure 3.17). Variables belong to an RTSC and store values. Operations belong to an RTSC and can be used to implement reusable statements. An operation has input parameters and one return value of a concrete type. An operation is implemented using the MechatronicUML action language [*DPP+16]. Entry events and exit events belong to a state. Transition events belong to a transition. An action belongs to each event. An entry action is executed when a state is activated, an exit action when a state is deactivated, and a transition action when a transition fires. An action is implemented using the MechatronicUML action language [*DPP+16]. In Modelica variables can be directly added to the class of the RTSC. Operations are represented by Modelica functions. A function is site-effect free. It has several inputs and outputs. Within an algorithm section the output variables have to be set. Entry events and exit events are represented by the value of the active variable of a step object and transition events by the value of the fire variable of a transition object.

Listing 3.11 shows the generation template for creating statechart variables in Modelica. Variables of RTSCs are translated to inner data variables in the class of the statechart that defines the variable and as outer data variable in all contained statecharts. We use the same name as in the MechatronicUML model. The MechatronicUML primitive data type DOUBLE is transformed to the Modelica Real primitive type. The MechatronicUML primitive data types INT8, INT16, INT32, and INT64 are transformed to the Modelica Integer primitive type. The MechatronicUML primitive data type BOOLEAN is transformed to the Modelica Boolean primitive type. Additionally, we declare the size of the array, if the

variable is an array. Constant variables get the Modelica keyword constant. As a result, they cannot be changed during a simulation run.

```
[for (variable : Variable | rtsc.allAvailableVariables)]
  [if rtsc.variables->includes(variable)]
  inner [if variable.constant]constant[/if] [variable.dataType.getType() /]
  [if dataType.oclIsKindOf(ArrayDataType)]
  '['/][for (interType : ArrayDataType | types) separator(', ')]
  [interType.cardinality.value/]
  [/for][']' [/if]
  variable.getName()/](start=[generateExpression(
     variable.initializeExpression)/]);
  [else]
  outer [if variable.constant]constant[/if]
   [variable.dataType.getType() /]
   [if dataType.oclIsKindOf(ArrayDataType)]
   '['/][for (interType : ArrayDataType | types)
    separator(', ')][interType.cardinality.value/]
    [/for][']' [variable.getName()/]; [/if]
  [/if]
[/for]
```

Listing 3.11: Declaration Template for Variables

```
function [operation.getName()/]
[for (parameter : Parameter | operation.parameters)]
   input [parameter.dataType.getType() /]
   [parameter.getName()/];
[/for]
   output [operation.returnType.getType() /]
   [operation.getOperationOutputVariableName() /];
algorithm
   [generateExpression(operation.implementations->at(1))/]
end [operation.getName()/];
```

Listing 3.12: Declaration Template for Operations

Listing 3.12 shows the generation template for creating an operation as a function in Modelica. The function is declared with the name of the operation. Afterward, we create for each parameter of an operation a corresponding input variable. Then, we create an output variable for the return value of a corresponding type. Lastly, the implementation of the operation is transformed to corresponding Modelica statements within an algorithm section. We omit the detailed explanation of the generateExpression template as it is strait forward. It is mainly a transformation of the action language syntax to Modelica syntax. For example, the following while loop **while**(a){a:=**false**;} is transformed to **while**(a) **loop** a := **false**; **end while**;.

Listing 3.13 shows the generation template for creating actions of entry events, exit events, and transition events in Modelica. The generated code belongs to the algorithm section of the root statechart. Thereby, it is possible to write all variables at a single point without violating the Modelica single-assignment rule.

The single-assignment rule enforces that only one equation or statement is allowed to assign a variable. The system of equations cannot be solved if the number of unknown variables is different than the total number of equations. “Multiple assignments to a variable within the same algorithm section is counted as a single assignment.” [Fri14] In this case, the last assignment sets the new value of the variable.

Entry events are similar to *when-clauses* that react when the active variable of a step object switches from false to true. Likewise, exit events are similar to Modelica *when-clauses* that react when the active variable of a step object switches from true to false. Transition events occur when the fire variable of a transition object becomes true. We generate for the action of each event corresponding Modelica statements within the when section.

```
// state entry actions
[for (state : realtimestatechart::State | stateSet)]
   when not pre([state.getFullyQualifiedName() /].active)
   and [state.getFullyQualifiedName() /].active then
   [generateExpression(state.entryEvent.action.expressions->at(1)) /]
   end when;
[/for]
// state exit actions
[for (state : realtimestatechart::State | stateSet)]
   when not [state.getFullyQualifiedName() /].active
   and pre([state.getFullyQualifiedName() /].active) then
   [generateExpression(state.exitEvent.action.expressions->at(1))/]
   end when;
[/for]
// transition event actions
[for (transition : Transition | transitionSet)]
   when pre([transition.getFullyQualifiedName() /].fire) then
   [generateExpression(transition.action.expressions->at(1)) /]
   end when;
[/for]
```

Listing 3.13: Declaration Template for Entry Actions, Exit Actions, and Transition Actions

Figure 3.28 shows an example of the generation of a Modelica class for an RTSC with variables, operations, and actions. The behavior represents an alternative version of the OvertakeeCommunicator behavior (cf. Figure 3.18).

In the left part, Figure 3.28 shows an RTSC that has the state activeCommunicator in the upper level. This state defines the variables refSpeed, error, and blockSpeed. The variable refSpeed represents the intended speed by the driver. The variable blockSpeed represents the current speed of the overtakee at the moment when the overtaking maneuver starts. The variable error shall become true if the overtakee wants to drive faster than the blockSpeed during the overtaking maneuver. Furthermore, the state activeCommunicator defines the operation verifySpeed(double ref, double blockSp) that gets the double parameters ref and blockSp and returns the boolean parameter out.

The state activeCommunicator is a composite state as it consists of the region monitor. The statechart of the region monitor has the states arbitrarySpeedUp and noSpeedUp. The state arbitrarySpeedUp sets the variable error to false as entry action. The state noSpeedUp sets the variable error to the return value of the operation call of verifySpeed as entry action. The variable refSpeed is bound to the operation parameter ref and the variable blockSpeed is bound to the parameter blockSp. The transition action of the transition from the state arbitrarySpeedUp to the state noSpeedUp sets the value of the variable blockSpeed to the current value of the variable refSpeed. The state noSpeedUp has a self-transition that could be used to trigger the entry action periodically. Additionally, a transition from the state noSpeedUp to the state arbitrarySpeedUp exists without having any effects.

In the right part, Figure 3.28 shows the generated Modelica class. In this example, we focus on the generation of the variables, operations, and actions.

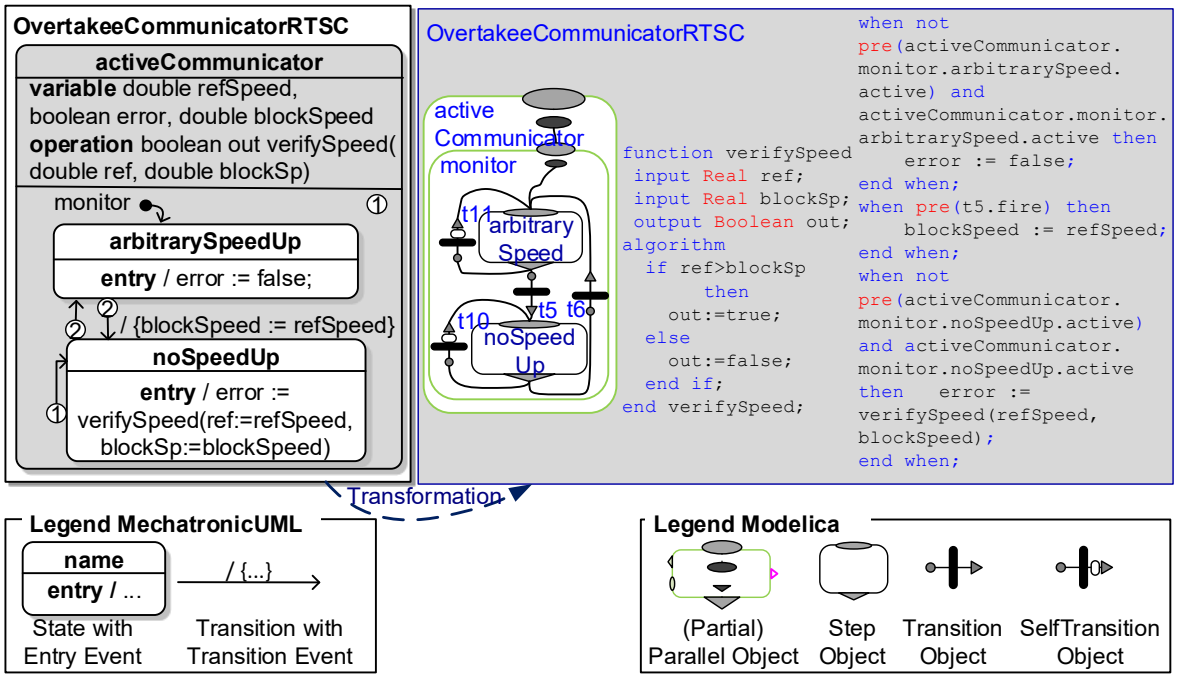


Figure 3.28: Transformation of a MechatronicUML Real-Time Statechart with Variables, Operations, and Actions

The function verifySpeed is created for the corresponding MechatronicUML operation. Both operation parameters are transformed to the input Real variables ref and blockSp. The return value out is transformed to the output Boolean variable out. The implementation of the operation is transformed to corresponding Modelica statements within its algorithm section. The entry action of state arbitrarySpeedUp is transformed to the when section that has the variable arbitrarySpeed.active as a trigger. Next, the transition action is transformed to the when section that has the variable t5.fire as a trigger. Lastly, the entry action of state noSpeedUp is transformed to the when section that has the variable noSpeedUp.active as a trigger. The operation call of verifySpeed within this action is transformed into a corresponding Modelica function call.

3.5 Tooling Implementation

We implement the transformation from MechatronicUML to Modelica prototypically and integrate the transformation within the Eclipse-based MechatronicUML Tool Suite. For quality assurance, we implement the plugin `example.test`. This plugin provides a unit test that executes the Modelica transformation for all MechatronicUML `example` plugins on which it depends. As a minimum set, we provide six simple MechatronicUML examples that use message-based communication between components, simple and hierarchical RTSCs using entry and exit points, clock-based behavior, and operations. Thereby, we cover the main concepts that this chapter describes. The set of standard test cases can be extended at each time by adding a corresponding plugin dependency. The unit test runs a Modelica compiler on each generated code and fails if the Modelica transformation or the Modelica compiler fails. The Modelica compiler fails also if the resulting files do not exist or if they are empty. Otherwise, the test finishes successfully. Furthermore, we use a ticket system for structuring the development and manual testing tasks. Different persons perform the development and

testing tasks independently. We perform simulation test runs on the generated examples manually using the "all states" coverage criteria [UPL12] for the RTSC of each discrete software component. Furthermore, weekly meetings are used to discuss the next tasks and results. We use a Jenkins continuous integration server [Ber12] to build all plugins automatically and to run the unit tests automatically if any registered MechatronicUML example plugin or any Modelica adapter plugin change.

Figure 3.29 shows the main plugins and their dependencies. The plugin `modelica.adapter.ui` implements the integration into the user interface. It enables to start a transformation on a `.muml` file from the context menu. The user interface plugin uses the plugin `modelica.adapter` to perform the transformation. This plugin implements the transformation as an M2T transformation using the tool Acceleo [Acc]. It uses the meta-model plugin `pim` to load MechatronicUML models, which are the source of each transformation. Furthermore, the transformation plugin `modelica.adapter` uses the meta-model plugin `modelica.transform`, which is a helper model that represents provisional results during the transformation. The meta-model plugins use the Eclipse modeling framework [SBMP08]. Moreover, the plugin `modelica.adapter` uses the plugin `graphviz.blackbox` to transform MechatronicUML models to a `graphviz` model using QVTo [QVT]. Created graphviz models are layouted by the tool graphviz [GRAPHVIZ]. The calculated layout is annotated at the corresponding MechatronicUML elements. The transformation plugin `modelica.adapter` uses the layout to create layouted Modelica models that contain elements of the `Real-Time Coordination Library`.

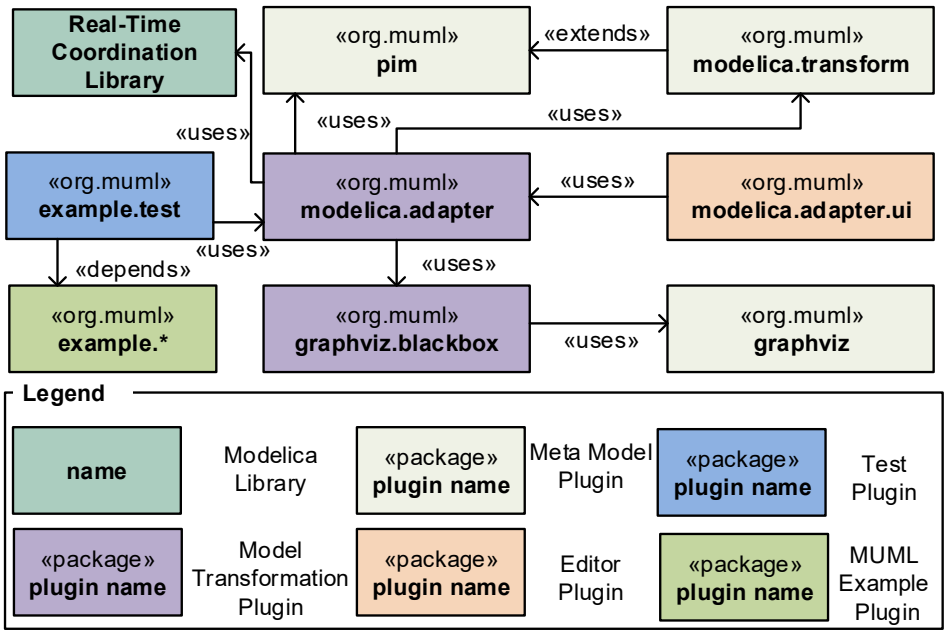


Figure 3.29: Plugins Implementing the Translation of MechatronicUML Models to Modelica Models

3.6 Case Study

In this section, we evaluate our approach for MiL simulation of CPSs in Dymola. We perform the evaluation as a case study on the basis of the guidelines defined by Kitchenham, Pickard, and Pfleeger [KPP95] and Runeson and Höst [RH08]. In particular, we evaluate the transformation

of MechatronicUML models to Modelica. The design of the case study is inspired by the case study of Heinzemann [Hei15], who evaluates the transformation of MechatronicUML models to MATLAB Stateflow/Simulink in parallel to this work. We created and discussed the cases together in joint work. We perform our evaluation for the two realistic examples: (1) cooperating delta robots [*PTD+14] and (2) coordinated overtaking of two cars [*PHM+14]. We reproduce the evaluation of Heinzemann using Modelica and our transformation instead of MATLAB Simulink and the corresponding transformation of Heinzemann to avoid new construction threats to validity [Hei15]. Using our approach, we reach the same result as Heinzemann.

3.6.1 Context and Cases

The aim of our case study is to evaluate if the transformation of MechatronicUML models to Modelica provides syntactically and semantically correct models. Therefore, we evaluate the following evaluation questions:

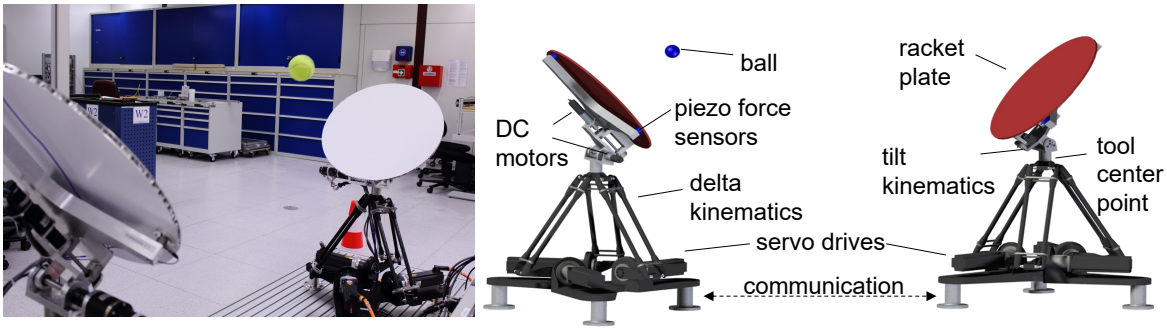
1. Is the generated Modelica code syntactically correct and contains all modeling elements defined by the MechatronicUML model?
2. Is the generated Modelica code semantically correct, i.e., does the behavior of the generated Modelica model in a simulation run conforms to the behavior defined by the MechatronicUML specification [*DPP+16]?

These models should be simulated in combination with environmental and physical parts of CPSs in Dymola. We acknowledge a generated Modelica model to be syntactically correct if it fulfills the Modelica concrete syntax specification [MODELICA]. Furthermore, we acknowledge a generated Modelica model to be semantically correct if it shows the same behavior as the MechatronicUML model.

We perform the case study on cooperating CPSs from the production domain and from the automotive domain. In the following paragraphs, we describe these systems and their characteristics.

Cooperating Delta Robots Figure 3.30 shows a prototype and a sketch of two equal cooperating delta robots that we consider in our case study. The prototype that Figure 3.30a shows is the result of the interdisciplinary driven joint project ENTIME [*GST14]. Each robot is constructed to a large extent using on the market available solution elements. Figure 3.30b shows a sketch of the solution elements that are used to build the robots. The delta robots use delta kinematics, which are often used for pick-and-place application in production systems [*GST14].

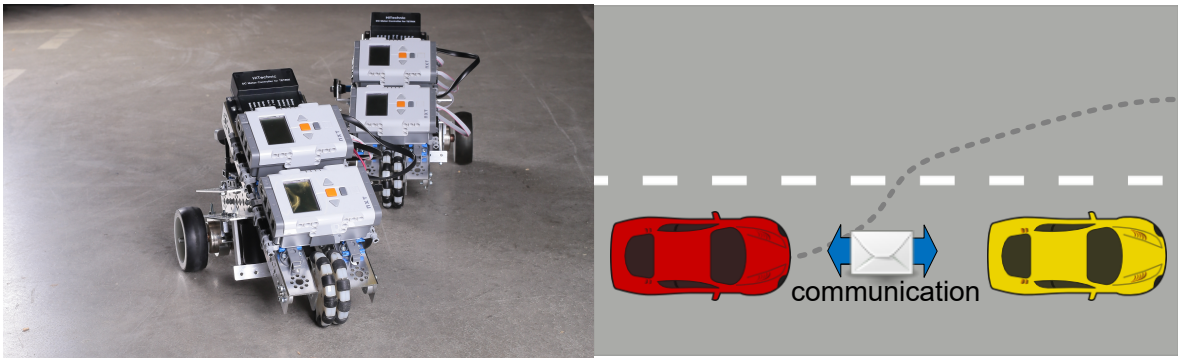
These robots juggle a ball between each other in a cooperative manner without using visual information that is provided by a camera. A robot cannot sense the ball until the moment when it strikes the ball. As a result, each robot relies to plan its strike on the information of the flying ball trajectory that is provided by the other robot via communication. During a strike, a robot senses the position of the ball on its racket plate by piezo force sensors and computes a predicted ball trajectory based on the sensed position. Afterward, the robot sends a message that contains the predicted ball trajectory to the other robot. This robot receives the message and calculates a strike trajectory for the racket plate to strike the ball at the predicted position. During the strike, it senses the ball again and starts the process vice versa. We model the communication using MechatronicUML. Nevertheless, the state-based discrete communication protocol is quite simple, it shows the tight integration between software-based coordination and physical processes considering real-time requirements.



(a) Physical Prototype

(b) Sketch (Adapted from [*PTD+14])

Figure 3.30: Cooperating Delta Robots (Adapted from [*GST14])



(a) Physical Prototype

(b) Sketch (Adapted from [*PHM+14])

Figure 3.31: Cooperative Overtaking of Two cars (Adapted from [*PHM+14])

Cooperative Overtaking Cars Figure 3.31 shows a robot-based prototype and a sketch of two cars performing a cooperative overtaking maneuver that we consider in our case study. The prototype that Figure 3.31a shows is the result of the supervised master project group Cybertron (`https://www.hni.uni-paderborn.de/swt/lehre/projektgruppenarchiv/projektgruppe-cybertron/`). The prototype is constructed using on the market available Lego elements. The Lego robot is a testbed for a real car but easier to handle. Figure 3.31b shows a sketch of the overtaking maneuver.

The behavior of the coordinated overtaking case study reflects a part of the autonomous cooperating overtaking maneuver (cf. Section 1.3). It considers the communication between the overtaker and overtakee but omits the communication between the cars and the section control. The rear car overtakes the front car after assuring itself whether it is safe to overtake or not by asking the front car. Furthermore, the front car agrees not to accelerate during the overtaking maneuver. As a result, the overtaking maneuver is safer than relying only on local sensor data. We model the communication using MechatronicUML. The model consists of multiple communicating components and hierarchical RTSCs. The scenario has to fulfill hard real-time requirements.

3.6.2 Hypothesis

Heinzemann [Hei15] verifies safety properties of the protocol behavior of the MechatronicUML example models formally using UPPAAL [UPP]. Furthermore, he verifies the correct refinement of the protocol behavior to component behavior formally using

a refinement check [Hei15]. As a result, we consider the models to be correct with respect to their specification, as we rely on the same models like Heinzemann. Furthermore, we define similar two evaluation hypotheses like Heinzemann.

- H1 The generated Modelica models are syntactically correct with respect to the Modelica specification [MODELICA].
- H2 The behavior of the generated Modelica models in a simulation run conforms to the behavior defined by the MechatronicUML specification [*DPP+16].

3.6.3 Analysis Procedure

We evaluate our first hypothesis H1 by loading the generated model in Dymola *2014 FD01* because Dymola checks if the models are syntactically correct when it loads the models. Afterward, we can start checking our hypothesis H2. As a prerequisite, we have to replace continuous components by components from an existing Modelica library or we have to implement them manually because we generate only stubs for them without meaningful behavior. In a next step, we have to check if the generated model in combination with the added or implemented continuous components is well-posed. That means the model must have the same number of variables and equations. Modelica checks this by its single-assignment-rule, which forbids to assign a value to a variable from two locations, which would result in two equations for one variable. Dymola cannot compile Modelica models if they are not well-defined. Finally, we perform simulation runs and perform conformance testing [LY96; Gar05] by comparing the behavior of the discrete state-based communication with the intended behavior of the MechatronicUML specification [*DPP+16] manually. We examine the simulation runs by plotting variables in diagrams and activate the debug option to log all events during simulation and initialization using the coverage criteria "all states" [UPL12], i.e., each state becomes active at least once.

3.6.4 Preparation of the Data Collection

We prepare the case study by loading the existing MechatronicUML models of Heinzemann of the cooperating delta robots and the cooperating cars. Afterward, we update them to the latest version of the MechatronicUML Tool Suite [*DGB+14]. Each model contains coordination protocols, a software architecture in the form of a component diagram, and a behavior specification of each discrete component in the form of an RTSC. We create a CIC that serves as the input for our transformation to Modelica.

Figure 3.32 shows the CIC of the cooperating delta robots r1 and r2. Each robot consists of the hierarchical hybrid software component instances of the types RobotSW and InformationProcessing. The InformationProcessing component instance consists of the discrete component instances of the types CoordinationModule, SensorProcessing, and StrikeProcessing. These components have incoming and outgoing sampling ports, which have to be connected to environment components and physical continuous components. The unidirectional protocol Prediction is implemented between the component instances SensorProcessing and CoordinationModule. Additionally, the unidirectional protocol Strike is implemented between the component instances CoordinationModule and StrikeProcessing. The bidirectional protocol TurnTransmission is implemented by the component instance of CoordinationModule to communicate and cooperate with the robot r2.

Figure 3.33 shows the component instance configuration of the two cooperating cars. It consists of the discrete component instances of the types Rear_Sw and Front_Sw and the two continuous component instances of the type Car, the continuous component instance of the type Distance. The sampling ports of the discrete component instances are connected

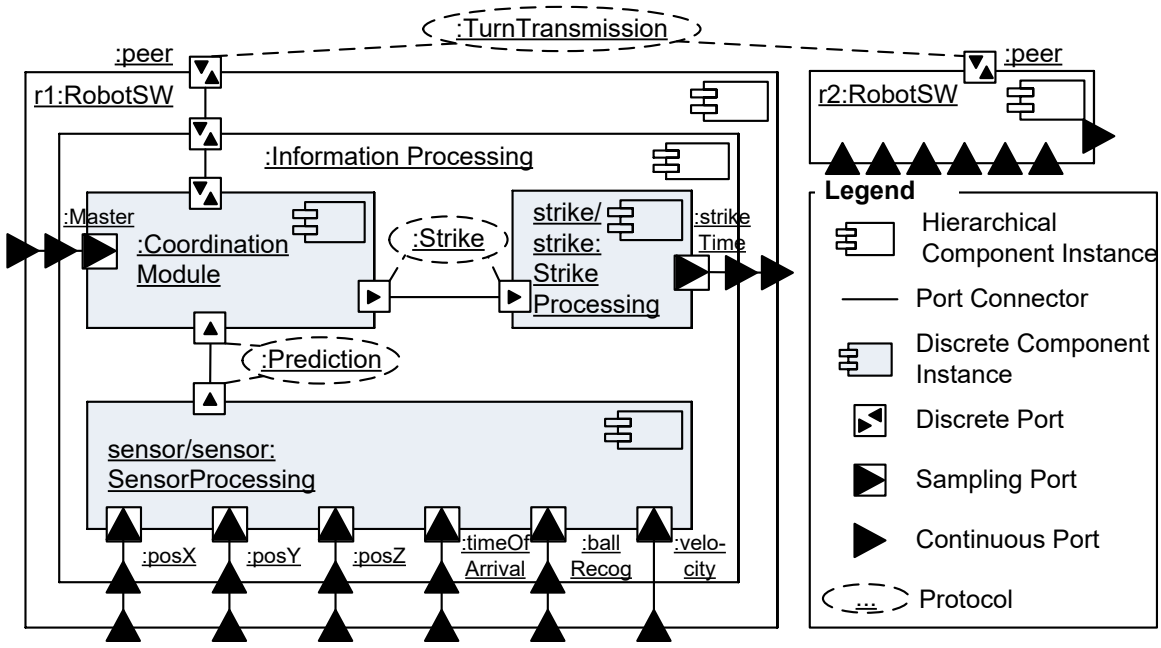


Figure 3.32: CIC of two Cooperating Delta Robot (Adapted from [*GST14])

to the continuous ports of the continuous component instances. The bidirectional protocol OvertakingCoordination is implemented between the discrete component instances of the types Rear_Sw and Front_Sw.

3.6.5 Data Collection Procedure

We start by loading the MechatronicUML model of the two delta robots within the MechatronicUML Tool Suite. Afterward, we validate the model within the MechatronicUML Tool Suit and check if the CIC and the RTSCs of the components fulfill the static semantics that is defined in [*DPP+16] using OCL meta-model constraints. The models pass this check. In the next step, we transform the CIC and its RTSCs to Modelica. Then, we load the model with Dymola successfully and instantiate two environment objects in the root class. We connect the connectors of the environment objects with the connectors of both robots. Thereafter, we validate the whole Modelica model and switch from the modeling view in Dymola to the simulation view. We choose the DASSL solver [Pet82; CK06], translate the Modelica model to a simulation model, and start a simulation run that lasts 30 seconds. This simulation time is sufficient for examining the coordination behavior. We examine that both robots start a cooperation and exchange the information about the ball trajectory successfully. The behavior that we observe during the simulation runs conforms to the behavior that is specified by the MechatronicUML specification [*DPP+16]. This means, that all states are de(activated) correctly, transitions only fire if all conditions are true and no other transition withing the same statechart fires, all messages are sent, buffered, and processed correctly, and timing behavior is correctly.

Next, we load the MechatronicUML model of the cooperating cars and repeat successfully the same steps as for the delta robots. In contrast to the delta robots where we add environment objects, we change the type of the continuous components to the corresponding classes that represent the physical behavior of the cars and the distance component. During the simulation run, we examine the negotiation via message-based communication of two cars that perform an overtaking maneuver. Furthermore, we examine if the cars behave safely in the case when

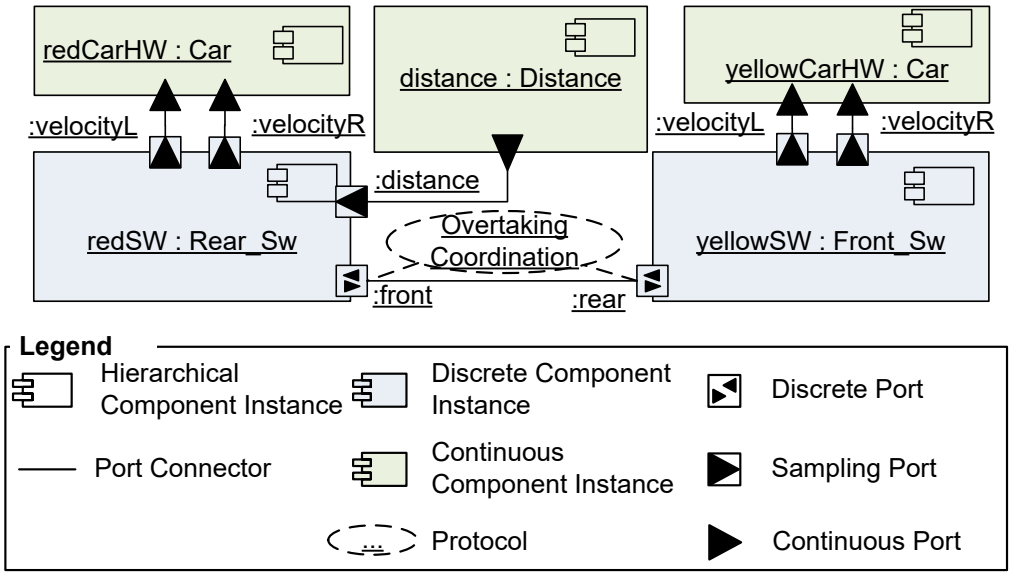


Figure 3.33: CIC of two Cooperating Cars (Adapted from [*PHM+14])

the front car declines an overtaking maneuver, i.e., the rear car reduces its speed and does not overtake. Lastly, we examine the scenario when the front car accepts to be overtaken. The observed behavior corresponds to the intended MechatronicUML behavior because the real-time, state-based behavior conforms to the RTSC specification and the message exchange between the component instances conforms to the expected message sequence within the simulation runs.

Additionally to our evaluation hypotheses, we evaluate the performance briefly to give an impression how much time the transformation of the MechatronicUML models to Modelica models need. We measure the execution times on a Lenovo Thinkpad with an Intel Core i7-3540M CPU, which runs at a speed of 2.9 GHz. The computer has 8GB DDR 3 RAM with a speed of 1600MHz and a Samsung 840 EVO hard disk with a capacity of 500GB. We use Windows 7 (64 Bit) as the operating system.

Table 3.1 shows the statistics locations of the results of the performance test while transforming both cases from the MechatronicUML model to the Modelica model. According to Devore, the shown statistics *locations* "serve to characterize the data set and convey some of its salient features"[Dev15]. The first row shows the aspect of each column and the first column shows the corresponding location that each row presents. The statistics locations are based on the performance measurement results that Table F.1 in Appendix F.1 shows. The first transformation run is always an outlier. Acceleo makes some technical optimizations and compilations during the first run, which takes significant time. Acceleo also caches some results, which improves the performance after the first run.

Table 3.1: Modelica Transformation: Execution Time Measurement Descriptive Statistics

Location	**Cooperative Overtaking Cars**	**Cooperating Delta Robots**
Average	5158 ms	8237 ms
Min	3844 ms	6700 ms
1st Quartil	4195 ms	6842 ms
Median	4342 ms	7510 ms
3rd Quartil	5054 ms	8072 ms
Max	10668 ms	15346 ms

3.6.6 Interpreting the Results

The results of our case study show that we transform both MechatronicUML models to syntactically correct Modelica models that contain all model elements, which are defined by the MechatronicUML model. Therefore, we see our first hypothesis H1 and research question as fulfilled. Furthermore, the transformation is executed automatically. No manual intervention is required. Additionally, conformance testing [LY96; Gar05] by observing the simulation runs show that the created models behave as intended. We use the coverage criteria "all states" during the test runs [UPL12]. Therefore, we conclude that the transformation that this section presents preserves the semantics of MechatronicUML. As a result, we see our second hypothesis H2 and research question also as fulfilled.

3.6.7 Threats to Validity

Empirical studies have limitations and the results rely to a large extent on the research design. Each empirical study has to face threats to validity [RH08]. We use the systematics of Runeson and Höst for analyzing and reporting these threats by splitting the threats into the construct validity, internal validity, external validity, and reliability. Furthermore, we report the corrective actions that were performed to reduce the impact of the threats.

3.6.7.1 Construct validity

"This aspect of validity reflect to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions." [RH08] Currently, not all MechatronicUML elements and the combination of MechatronicUML elements are supported, as described by Section 3.7. We test the preservation of the semantics based on the presented case study and a set of simple MechatronicUML examples, which we create by ourselves. External users or industrial users may have different expectations or requirements, which we miss within our self-created examples. Nevertheless, we observed the execution carefully but we may miss some corner cases. We only test our models with the DASSL variable-step solver provided by Dymola [Dym]. Other solvers may provide different results. Furthermore, the coverage criteria "all transitions" or "all transition-pairs" for testing the state-based behavior provide a more reliable result than "all states" but result in a higher effort [BLW04].

3.6.7.2 Internal validity

"This aspect of validity is of concern when causal relations are examined." [RH08] Currently, we do not judge about a causal relation.

3.6.7.3 External validity

"This aspect of validity is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case." [RH08] The conducted empirical evaluation that uses a case study that considers two cases using a new approach cannot ensure external validity. Therefore, it is a big threat to our case study. Nevertheless, by applying the approach on one case from the production domain and one case from the automotive domain we assume that it can also be transferred to further cases and further domains, like avionics, where the behavior of software and communication is integrated with the dynamics of the physical processes [SWYS11]. Furthermore, the case study design may help another researcher to define a systematic guideline how to evaluate MiL simulation approaches of CPSs.

3.6.7.4 Reliability

"This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers." [RH08] We perform the comparison of the intended MechatronicUML semantics with the observed semantics of simulation runs manually. The interpretation of the results could vary as it is a subjective interpretation of the researcher. We discuss the interpretation of the results with several researchers and master students from our software engineering research group.

3.7 Limitations

Our approach for the translation of MechatronicUML models to Modelica models underlies the following limitations:

1. We do not support that different message types are being enqueued within a single queue. It is currently not possible to implement this directly within Modelica due to the limitation of Modelica to support only static type definitions. It is possible to enhance the transformation of MechatronicUML to Modelica in the future and to unify all message types to a message with one double parameter, like Heinzemann [Hei15]. Messages with more than one parameter have to be split up to several messages. The tracing of MechatronicUML messages during a simulation run in Modelica becomes more complicated after such a transformation.
2. Our transformation does not support user-defined structured/complex data types. This feature could be added for static types in the future.
3. Explicit type casts of primitive types are not considered within a transformation. This feature could be added in the future.
4. It is not allowed that the type of a message parameter is an array. The implementation of the mailbox would become much more complicated due to the limitation that all types of Modelica must be static. A transformation that splits arrays to several messages could be provided, but would add a lot of overhead.
5. Operations are not allowed to have side-effects. Modelica operations only support side-effect free operations. In the future, we have to define a transformation that analyzes an operation and passes all required statechart variables to a Modelica operation as parameters and as a return values.
6. We do not support the generation of Modelica models for reconfigurable software components [HB13; Hei15; SHG16]. Different Modelica-based tools offer different possibilities to consider reconfiguration behavior [ZLB07; EMK15]. Future work should investigate how to combine the Modelica capabilities for considering reconfiguration in combination with the reconfiguration approach of Heinzemann [Hei15] and the real-time coordination Modelica library that this chapter contributes.
7. Maximum message transfer delay of MechatronicUML assemblies are not considered. In the future, we have to add a network layer in Modelica to support QoS assumptions of network connectors.
8. Urgent transitions and urgent states are not considered. In the future, we have to consider that in urgent states no time can be spent and that urgent transitions have a higher priority than non-urgent transitions. Within the simulation it cannot happen that a transition becomes enabled but does not fire because we do not represent platform-specific scheduling in the simulation and therefore a statechart is always in a virtual running state.

3.8 Related Work

This section describes related work that considers the MiL simulation of CPSs. Especially, we describe the model of the discrete behavior of the different approaches. Furthermore, we describe how the transformation to a simulation environment is done if the modeling is not performed directly within Modelica. We discuss the supported communication styles of the different approaches and how they model real-time behavior. We also describe if the related approaches consider formal verification because our approach MechatronicUML verifies the discrete parts formally via timed model checking.

The next section includes approaches that use Modelica as modeling language and simulation environment. Afterward, Section 3.8.2 covers approaches that use other modeling approaches and transform their models to Modelica for MiL simulation. Next, Section 3.8.3 describes approaches that use other modeling languages and simulation environments. Finally, Section 3.8.4 describes approaches that support the Functional Mockup Interface (FMI) standard for co-simulating [FMI4CS] or exchanging [FMI4ME] model components.

3.8.1 MiL Simulation Using Modelica as Modeling Language and Simulation Environment

Modelica offers to build Domain-Specific Languages (DSLs) in the form of libraries (cf. Section 3.1.1). The Modelica community offers libraries for different modeling approaches. This section discusses three approaches that provide Modelica libraries for their approach. Furthermore, the section discusses the Modelica DSL for defining state machines, which is available since Modelica version 3.3 and is not based on standard Modelica elements. All approaches use only Modelica for modeling and simulating systems.

Otter *et al.* [OME+09] model the state-based behavior directly with the StateGraph2 library within Modelica. Therefore, the approach has the mentioned limitations concerning modeling of communication and real-time behavior (cf. Section 3.1.2). "Otter *et al.* propose a transformation of StateGraph2 to NuSMV [OME+09] for formal verification. In contrast to our approach that uses UPPAAL for timed model checking, NuSMV is only able to verify untimed behavior. This invalidates the approach for the use in the context of real-time CPS." [*PHM+14]

Zeigler, Kim, and Praehofer provide the formalism Discrete EVent System specification (DEVS) [ZKP00] for modeling and analysis of discrete event systems. "DEVS is the formalism that allows a modeler to specify a hierarchically decomposable system as a discrete event model that can be later simulated by a simulation engine." [CZ94] Sanz, Urquia, and Dormido provide a Modelica library for modeling and simulating using DEVS [SUD09]. They provide the usage of messages and mailboxes in Modelica via external C-functions [SUD08]. In contrast, our approach uses only Modelica elements. Thereby, we can use the Modelica compiler to analyze if the model is well-defined and it is possible to track the mailbox behavior within a Modelica simulation. External C-functions have a black-box behavior, which internal behavior could not be analyzed during a simulation run. Furthermore, engineers have to specify time-advance functions for modeling real-time behavior [UMMW12]. A system changes to a specific state when a time-advance function expires. In contrast, our approach differs between clocks, clock constraints, and invariants. DEVS networks can be verified formally [Hwa12], e.g., by using UPPAAL timed automata [SW10]. In contrast to our approach, they do not support the transformation of the same model to a model checker and a simulation environment. Engineers have to do this manually, which is tedious and error-prone. Furthermore, the verification approaches of DEVS consider the behavior of the whole network instead of a compositional

verification approach that MechatronicUML uses and which tackles the state explosion problem [Hei15].

"Reo [Arb04] is a channel-based coordination model for component-based systems or applications. Using Reo, designers can compose complex connectors out of simple ones. In contrast to our [approach] (...), Reo's main focus is the connector design and not the behavior design of each role. Moreover, Reo does not provide concepts for defining the component behavior [(communication, real-time)] based on the designed coordination. Brandt et al. simulate Reo circuits using Modelica [BSKA12]." [*PHM+14] In contrast to our approach, they offer no formal verification of their models using model checking.

"Elmqvist *et al.* [EGMD12] introduce a new concept in Modelica 3.3 for state machines. In contrast to StateGraph2 [and the Real-Time Coordination Library in Modelica 3.2] that use Modelica objects, this is a new sublanguage for state machines in Modelica. Up to now, there is no support of message-based communication between state machines. [The current definition provides no extension or alternation points. As a result, it is not clear if and how the definition is extensible.] Therefore, we do not use this new formalism as the implementation form for the state-based behavior." [*PHM+14] Modelica 3.3 adds support of synchronized discrete clocks that tick depending on a period or event. Transitions can use clocks as guards. Modelica 3.3 state machines do not support state invariant as first class elements. [EGMD12] "Currently, there is no transformation to a timed model checker available to verify safety requirements." [*PHM+14]

3.8.2 MiL Simulation Using Modelica as Simulation Environment

Modelica uses hybrid DAEs as its mathematical framework. Therefore, it has advantages for specifying and simulating physical models [Lee10]. However, Modelica 3.2 is limited to connection diagrams and offers no other (visual) formalism for specifying and verifying models. Other approaches use their own DSL, like MechatronicUML, to overcome these limitations and transform a specified and eventually verified model to Modelica to perform MiL simulation runs. This section describes these approaches that use Modelica as simulation back end.

"Donath *et al.* [DHBN08] provide a domain-specific language for state machines in SimulationX, which is based on UML. They use a model-to-text transformation to transform these models into Modelica algorithmic code. In contrast to our approach, they do not cover the needs of software engineers in the domain of safety-critical real-time systems, which communicate with each other. Among other things, component-based development, asynchronous message-based communication, timed behavior, and formal verification belong to these needs." [*PHM+14]

Schamai [Sch09] defines a UML profile called ModelicaML to support modeling Modelica classes and connection diagrams within UML. Developers define UML state machines to specify state-based behavior. We define in previous work, a transformation to Modelica algorithmic code [*Poh10; *SPF+10] on the basis of Schamai's work. Trigger events represent asynchronous communication within a state machine. Buffered messages between components can be represented by UML operations that are called as an effect of transitions[*PT+11]. Clocks, clock constraints, and invariants are not supported as first class entities. Like in Modelica, it is possible to model them manually. "In contrast to the ModelicaML approach that generates plain Modelica algorithmic code, we combine in this (...)[thesis] the use of existing Modelica libraries with model transformation techniques." [*PHM+14] Knapp, Merz, and Rauh [KMR02] show that UML state machines can be verified by timed model checking. As UML has semantic variation points and issues [SFP13] the semantics of the ModelicaML state machines and the semantics of Knapp, Merz, and Rauh [KMR02] have to be unified to get comparable results.

The OMG working group on SysML and Modelica integration [SysML] defines a SysML4Modelica UML profile and a transformation specification of SysML to Modelica [SysML4Modelica; PBB+10]. According to the SysML specification [SysML], the approach uses UML state machines [UML]. The transformation specification from SysML to Modelica does not cover the transformation of state machines [SysML4Modelica].

Grafchart is a modeling language that is designed for the behavior description of sequential Programmable Logic Controller (PLC) applications. Its syntax is based on the graphical syntax of Sequential Function Chart (SFC), which is one of the five languages that the IEC 61131-3 standard [IEC61131] defines. The semantics of Grafchart is based on Petri nets [JÅ99]. "Dressler [Dre04] describes a toolchain from JGrafchart to Modelica. JGrafchart is a tool to model state machines. The state machine elements are transferred into Modelica classes and objects without the usage of libraries. The state-based behavior is transferred into Modelica algorithmic code. The layout of the state machines is transferred from the JGrafchart diagrams into Modelica annotations. In contrast to our approach, no external tool is used for calculating a new layout." [*PHM+14] A limitation of the transformation from JGrafchart to Modelica is that it does not consider the communication abilities of JGrafchart [Dre04]. "A variable representing time may be introduced to create time-delayed actions and time-limited actions." [JÅ99] Toolchains for (timed) model checking are also available for JGrafchart [BEH+04]. In contrast to our approach, JGrafchart does not support component-based software development with message-based communication." [*PHM+14]

"Dempsey offers a toolchain from MATLAB Simulink models into Modelica using Simelica and the AdvancedBlocks library [Dem03]. Unfortunately, Stateflow blocks are not supported, which are required to model state-based software behavior." [*PHM+14] Therefore, this approach neither supports specifying state-based communication and real-time behavior nor verifying safety-requirements.

3.8.3 MiL Simulation Using Other Modeling Languages and Simulation Environments

MiL simulation has a long tradition [ÅK14]. Modelica has several competitors. In industry, MATLAB Simulink is the dominant product [ÅK14]. We show in parallel work to this thesis in [*HPR+12] that MechatronicUML models can be translated into MATLAB Simulink models. Heinzemann, Rieke, and Schäfer [HRS13; Hei15] show that even reconfiguration can be simulated. This section describes more approaches that use their own simulation environment to simulate state-based behavior in combination with physical systems.

Ptolemy II [Lee10] is a heterogeneous modeling approach that supports hybrid simulation of different discipline models. The Ptolemy II approach is a related approach to Modelica. It supports the modeling of discrete behavior in the form of finite state machines and continuous-time behavior in the form of ODEs [Lee10]. In contrast, our approach uses Modelica because it "has significant advantages, particularly for specifying physical models based on differential-algebraic equations." [Lee10] The state machines of Ptolemy II are reactive to events. They do not use concepts like mailboxes. In contrast to our approach, the execution of a state machine takes no time. As a result, timing behavior is only supported by delaying the execution of a state machine as a result of an event or by sending output events depending on a time condition [Lee09]. Ptolemy II models are verified by a transformation to a hybrid model checker [RLW+14], which restricts them to check only small models with up to 200 variables [FLD+11], which is not sufficient for realistic examples.

Burmester *et al.* [BGH+07] provide an approach for simulating CPSs that are specified by using an early version of MechatronicUML in combination with CAMeL-View [CAM]. CAMeL-View uses, like Ptolemy II and Simulink, ODEs for describing continuous parts. In

contrast to our approach, they require generating C++-Code of the discrete behavior model and of the continuous model that needs to be integrated manually. "The simulation is performed based on code rather than on models as in our approach. In contrast to our approach, this significantly hardens the inspection of the model while performing MIL simulations (...)" [Hei15] The communication, real-time, and verification capabilities of our approach are similar because our approach is an evolution of their approach.

Palachi, Cohen, and Takashi integrate continuous components, like Simulink blocks, into SysML models [PCT13]. As a result, they mix discrete and continuous behavior. They generate C++-code for the discrete behavior via Rhapsody and compile it into a MATLAB S-Function to simulate the behavior within Simulink. Furthermore, a Simulink model is generated that references the former integrated Simulink blocks and links them to the MATLAB S-Functions. In contrast to our approach, the discrete behavior of a state machine is a black box during simulation and only variables that are defined within the interfaces can be inspected. Furthermore, they do not consider communication, real-time, and verification at all.

3.8.4 MiL Simulation Using Functional Mockup Units

"A recent development in the industrial simulation community is the introduction of the FMI designed to facilitate tool interoperability at the level of compiled dynamic models. FMI specifies an XML schema for model metadata, such as names and units, and a C API for evaluation of the model equations. The first version of FMI was introduced in 2010 and since then a large number of tools have adopted the standard." [ÅK14] The Modelica association defines an FMI standard for co-simulation [FMI4CS] and for model exchange [FMI4ME]. An executable that implements an FMI is called Functional Mockup Unit (FMU). Each co-simulated FMU includes its own solver. FMUs that implement the FMI model exchange standard use the solver of the hosting tool. We provide an approach and prototype to embed generated and compiled C Code of a CIC that only contains discrete component within an FMI-wrapper [*PW+12; *PSR+12]. The resulting FMU can be combined with continuous components via continuous ports within a simulation environment, like Dymola, via model exchange. The approach considers the MechatronicUML semantics including the real-time elements of RTSC. The approach cannot handle asynchronous communication between several FMUs directly because of the limitations of the FMI standard [FGP14]. These issues have to be addressed in the future.

Grafchart (cf. Section 3.8.2), the modeling language for the behavior description of PLC applications, discusses the co-simulation of state machines with continuous systems without providing a prototype and an evaluation [TJ14]. The approach enables to use the simulation time within the state machines. Nevertheless, Theorin and Johnßon [TJ14] do not discuss how to support communication capabilities.

Tripakis and Broman [TB14] describe a formal model for bridging the semantic gap between heterogeneous modeling formalisms and FMI using co-simulation. They discuss how to map networks of timed automata, Ptolemy II state machines, and Rhapsody UML state machines to FMUs. The work set the formal foundation for implementing state machines as FMUs. In contrast to our approach, they do not provide an implementation and evaluation.

Feldman, Greenberg, and Palachi [FGP14] describe an approach to export a SysML/UML state machine from Rhapsody as an FMU that implements the FMI model exchange standard. The FMU can be used inside every FMI-compliant tool. In contrast to our MechatronicUML approach, their approach is limited to the standard real-time and communication capabilities of UML [UML]. We combine within RTSCs the UML state machine capabilities with features of timed automata [*DPP+16]. Moreover, the approach [FGP14] does not support communication between FMUs.

3.9 Summary

This section summarizes to what degree the related work and MechatronicUML in combination with the results of this thesis fulfill the requirements R.3.1-R.3.6, which the introduction of this chapter defines for a model-driven software design approach for CPSs. We claim that the requirements are necessary but not sufficient because we focus on the safety of the platform-independent functional behavior but not, e.g., on the execution of concrete platforms. Table 3.2 shows a summary of the fulfillment of the requirements.

The requirement R.3.1 considers the architecture as the basis for a design approach. This includes hierarchical nesting and multiple communication styles. All model-driven MiL simulation approaches that are based on UML/SysML except [DHBN08] fulfill this requirement because they use UML's composite structure or SysML's internal block diagram to define the structure. They support hierarchical nesting as well as multiple communication styles. Furthermore, we rate that Ptolemy II fulfills this requirement, nevertheless they use a different concept for defining the architecture, where Ptolemy II differs between so called domains [BLL+08]. In contrast, we rate that simulation approaches that use simple block diagram or respectively connection diagram, like Modelica-based, FMI-based, or Simulink-based approaches, do only fulfill this requirement partially because they only support the communication via signals. Although other communication styles can be simulated by using signals [*HPR+12; *PSR+12; *PDS+12]. Grafchart does not state how and if they can specify a system's architecture [Dre04; TJ14] or if it is embedded into another approach. Therefore, we set the corresponding cell to "?" (unknown). MechatronicUML provides a hierarchical component model and supports different communication styles for message-based and signal-based communication (cf. Section 2).

The requirement R.3.2 requires state-based behavior, which all approaches that are based on finite state machines, like Modelica StateGraph2, UML/SysML state machine, or MATLAB Stateflow, fulfill. DEVS is process oriented and does not fulfill our requirement [ZKP00; SUD09]. REO [Arb04; BSKA12] is a coordination model that focuses on the different kinds of communication channels and does not fulfill our requirement. SysML4Modelica excludes UML state machines [SysML4Modelica; PBB+10]. Simulink without Stateflow also does not support state-based behavior [Dem03]. MechatronicUML fulfills the requirement by providing RTSCs that are based on timed automata, which are an extension of finite state machines [AD94].

Clocks, which the requirement R.3.3 necessitates, are used by timed automata [AD94] but are also used in synchronous languages, like Esterel or SIGNAL [Gam10]. Therefore, all approaches that are based on timed automata and synchronous state machines fulfill this requirement. Grafchart also supports clocks [Dre04; TJ14]. UML-based approaches support time in form of time events, which are not restricted like clocks and cannot be verified by using timed model checking. Therefore, we state that UML-based approaches do not fulfill the requirement R.3.3. RTSC of MechatronicUML supports clocks as first class entities because it is based on timed automata.

Besides the correct timing-behavior also the asynchronous message-based communication and buffing, which the requirement R.3.4 demands, is essential for cooperating CPSs and the intra-system's behavior. This requirement is only fulfilled by the approach Ptolemy II [Lee10]. Furthermore, our former work [*PT+11] and the standalone Modelica Real-Time Coordination library [*PDS+12] that Section 3.3 presents fulfill this requirement. [TB14] and MATLAB Stateflow/Simulink fulfill this requirement partly because they support asynchronous communication but do not provide the required buffers and buffering strategies. All other approaches do not support message exchange and buffering as first class entities. MechatronicUML supports this requirement in Modelica [*PDS+12; *PHM+14], in

Table 3.2: Comparison of the Requirements Fulfillment for Software Engineering Approach for CPSs (Legend: ✓: fulfilled; ◑: fulfilled partially; ∅: not fulfilled; ?: unknown)

Approach	R.3.1	R.3.2	R.3.3	R.3.4	R.3.5	R.3.6
Modelica StateGraph2 Library [OME+09]	◑	✓	∅	∅	✓Modelica	✓
Modelica DEVS Library [ZKP00; SUD09]	◑	∅	✓	✓	✓Modelica + External C	∅
Modelica REO Library [Arb04; BSKA12]	◑	∅	∅	∅	✓Modelica	∅
Modelica Synchronous State Machine [EGMD12]	◑	✓	✓	∅	✓Modelica	∅
Tailored UML State Machine [DHBN08]	◑	✓	∅	∅	✓Modelica	∅
UML State Machine [Sch09; *SPF+10]	✓	✓	∅	∅	✓Modelica	∅
UML State Machine [*PT+11]	✓	✓	∅	✓	✓Modelica	∅
[SysML4Modelica; PBB+10]	✓	∅	∅	∅	Modelica	∅
Grafchart State Machine [Dre04; TJ14]	?	✓	✓	∅	✓Modelica / FMI	✓
Simulink [Dem03]	◑	∅	∅	∅	✓Modelica	∅
Ptolemy II (Finite State Machines) [Lee10]	✓	✓	∅	✓	✓Ptolemy II	✓(Hybrid)
UML State Machine [PCT13]	✓	✓	∅	∅	✓MATLAB S-Function	∅
UML State Machine / Timed Automata Variants [TB14]	◑	✓	✓	◑	◑FMI	∅
SysML/UML State Machine [FGP14]	✓	✓	∅	∅	✓FMI	∅
MATLAB Stateflow [HR07; MWC10]	◑	✓	∅	◑	✓Simulink	✓
Modelica Real-Time Coordination Library [*PDS+12]	◑	✓	✓	✓	✓Modelica	∅
MechatronicUML including this thesis	✓	✓	✓	✓	✓Modelica / MATLAB / FMI / CAMel-View	✓(Timed)

MATLAB [*HPR+12], and in CAMeL-View [BGH+07] but does not fulfill this requirement within its FMI representation [*PSR+12].

The support of MiL software simulation in combination with continuous system parts is fulfilled by all approaches using different approaches as necessitated by the requirement R.3.5. The simulation approaches differ within their capabilities because the different approaches differ within their modeling capabilities. Tripakis and Broman [TB14] fulfill the requirement only partially because they only present theoretical work and do not provide an implementation and a corresponding evaluation. MechatronicUML enables to use Modelica, MATLAB, FMI, and CAMel-View.

Lastly, only three related approaches that provide MiL simulation fulfill the requirement R.3.6 and support formal verification by means of model checking. Ptolemy II [Lee10] supports hybrid model checking, which has scaling problems [Hen00; FLD+11]. MechatronicUML is the only approach that combines MiL simulation with timed model checking.

4

Allocation Engineering

A Cyber-Physical System (CPS) forms a distributed system of systems through the networking of embedded systems [aca11]. Software engineers who develop a single CPS have to define the onboard allocation of the component instances to Electronic Control Units (ECUs) within the system under development [ŠCV13b]. CPSs have to fulfill many safety requirements. The allocation of component instances to hardware resources is essential to fulfill these requirements. A software component instance cannot run on any ECU because it has, e.g., special resource requirements or depends on another component instance. Moreover, redundant component instances have to be allocated to different, independent ECUs to fulfill safety requirements [ISO26262-1]. Furthermore, advanced driver assistance systems, like Anti-lock Braking System (ABS) and Brake-by-wire, are running on different ECUs and have to communicate if they depend on each other [ZS14]. Therefore, also the limited resources of the network have to be considered during the allocation.

Planning an allocation that fulfills all constraints is NP-complete [Luk10; Hüw13]. For this reason, it is not an applicable solution for nontrivial systems to specify a specific allocation specification manually and to test afterward if all constraints are fulfilled. Meta-heuristic approaches compute new, varying allocation specifications automatically and test if the computed solution is feasible [Luk10]. Therefore, they do not perform well for planning allocation specifications if the solution space that fulfills all constraints is rather small compared to the search space [Luk10]. In contrast, solvers for constraint satisfaction problems, like Boolean Satisfiability Problem (SAT) [PBG05], Constraint Logic Programming (CLP) [JL87], or Integer Linear Programming (ILP) [KLMJ10] solvers, search a solution systematically, e.g., by using branch-and-bound algorithms [Dak65]. They provide efficient solvers for NP-complete problems [CJP83] but need exponential-time if $P \neq NP$ [Sip06] in the worst-case. However, encoding an allocation problem manually as a constraint satisfaction problem is a complex and error-prone task [MMM12; ZP13; Luk10]. But indeed, using the manually encoded constraint satisfaction problem enables to compute a feasible solution if the encoding is sound. Software engineers or respectively allocation engineers have to create a feasible allocation specification that fulfills all constraints. They need a model of the problem to compute a feasible allocation specification. In detail, they require as an input for computing an allocation: (1) a concrete software model, (2) a concrete hardware platform model, (3) and an allocation constraint model that refers to component instances and hardware resources.

Currently, some component-based Model-Driven Engineering (MDE) approaches like DIF [MMM12], Autofocus3 [VEH14], or ArcheOpterix [ABGM09] provide allocation engineers with user interfaces to select constraints from a predefined set of allocation constraints. They do not offer the possibility to specify new allocation constraints without changing or extending the framework or tooling. The approaches provide for each constraint a model transformation into an encoding as a formal constraint satisfaction problem. Hence, engineers have to become experts in how to specify allocation satisfaction problems if they like to implement a new constraint that is not handled within the approach. The advantage of the tools is that they

compute feasible allocation specification for the given constraints and hide the formal encoding from the user. As a drawback, the approaches are inflexible regarding the specification of new constraints and do not cover all required constraints (e.g., coherence, location, timing, memory resource, execution/network schedulability) for planning the allocation specification of a CPS. Furthermore, the existing approaches for design-space exploration like SAOL [KPP+15], SCALL [ŠC15], or OpenDSE [Luk10] only consider flat hardware models or flat software models and do not support hierarchical models, which are provided, e.g., by a UML composite structure model [UML], the UML MARTE hardware model [MARTE], or our MechatronicUML approach [*BDG+14; *DPP+16]. Table 4.8 in Section 4.13 summarizes the related work and their capabilities in more detail.

MechatronicUML was developed for the platform-independent software design that can be verified formally. It provides the so-called component instance configuration to define the system's software architecture. The allocation of the software to distributed platforms was not considered in detail for the requirements of CPSs.

We like to compute allocation specifications automatically on the basis of a software architecture, a hardware platform, and constraints. The software engineering approach should allow software engineers and allocation engineers to specify the system's allocation constraints on the basis of the component instance configuration and the platform requirements of the contained software components. Furthermore, the approach should enable to describe the hardware platform by means of ECUs, sensors, and actuators properties as well as the network architecture and properties. Especially, in contrast to existing work, the engineers should be able to specify new constraints without requiring the knowledge how to encode the allocation problem as a constraint satisfaction problem. The encoding of the allocation problem should be sound and traceable concerning the references the software model and hardware model.

In the following, we summarize our requirements. We develop the requirements in the context of the goals of this thesis. Therefore, we do not claim that the list is complete in general. Our requirements on the basis of [Dan13; Hüw13; BCD+14] are as follows. A model-driven allocation engineering approach for CPSs approach should ...

- R.4.1 be able to specify and refer to a hierarchical software architecture.
- R.4.2 be able to specify and refer to the relevant resource consumption requirements of component instances or respectively of tasks.
- R.4.3 be able to specify and refer to a hierarchical hardware architecture of CPSs.
- R.4.4 be able to specify and refer to the relevant hardware resource properties (memory,computing,network).
- R.4.5 be able to specify constraints for a certain subset of component instances.
- R.4.6 be able to use predefined or specify new allocation constraints of component instances for coherence/dependency, incompatibility, target platform properties, and resource usage.
- R.4.7 provide a method to add new custom allocation constraints, e.g., a constraint DSL for software engineers, who are no experts in constraint satisfaction problems.
- R.4.8 perform a sound and efficient allocation planning.

In the following, we explain and justify our requirements. The requirement R.4.1 necessitates that an allocation engineering approach should refer to existing hierarchical software architecture models, e.g., a software component model. This model contains the information about the different parts of the software and how they are assembled. This information is essential for the allocation planning to fulfill the allocation constraints of component instances. Correspondingly, the resource consumptions of component instances are required by an allocation approach (requirement R.4.2). The allocation engineer must be able

to model and refer to them. Otherwise, the allocation planning cannot decide if an (embedded) platform with limited resources can provide them during runtime. Therefore, always the worst-case resource demands must be specified. A lack of resources during runtime can lead to risky software failures. Likewise to the software architecture, the hardware platform and its provided resources have to be specified. Without this information an allocation approach cannot plan safely. The requirements R.4.3 and R.4.4 cover these aspects. Furthermore, not all component instances have to fulfill the same allocation constraints. An examples for a general constraint is that each component instance must be allocated to exactly one ECU. In contrast, each component instance has individual requirements. For example a component instance depends on another instance or it requires specific platform properties. Furthermore, not all component instances have deadlines that require a real-time scheduling and a schedulability analysis. Therefore, the requirement R.4.5 necessitates that for each specified constraint it should be possible to narrow the set of component instances that have to fulfill the constraint. Component instances have to fulfill different allocation constraints. For example, they have to consider coherence and dependency constraints, incompatibility constraints, target platform property constraints, and resource usage constraints [MBAG11; ŠCV13b; Luk10; ZP13; MMM12]. As a result, the requirement R.4.6 requires that an allocation approach must support to use or to specify these allocation constraint types. In addition to the requirement R.4.6, the requirement R.4.7 considers that allocation engineers who are software engineering experts but no experts in constraint satisfaction problems are able to specify constraints by themselves. Specifying constraint satisfaction problems is a specific domain knowledge and not every software engineer is deeply trained in the specification of constraint satisfaction problems [ABDM01]. Lastly, the requirement R.4.8 necessitates that an allocation planning algorithm should always find a feasible solution efficiently in a sound way on the basis of the specified allocation constraints, component instances of a concrete software architecture, and ECUs of a concrete hardware platform. The result of the allocation planning should be an allocation specification that allocates each component instance to a specific ECU such that all allocation constraints are fulfilled.

The contribution of this chapter is a model-driven allocation engineering approach. For this purpose, we identify viewpoints and views for the stakeholder's concerns in model-driven allocation engineering for a safe deployment. We need a domain-specific language with different views for the different stakeholders and their viewpoints. Domain-specific languages describe what concepts exist and what properties they have, which is also named *ontological meta-modeling* [AK03]. We define an approach for the multi-view modeling of hardware platforms. The approach contains a systematic process and a formal hardware platform description language. It uses concepts known from view-based and component-based software engineering [GBB12; Bur13]. Moreover, we provide a domain-specific language to engineers for specifying allocation constraints in an easy and expressive way. The language integrates the Object Constraint Language (OCL) to specify model queries that select, for instance, a subset of component instances that have to fulfill a certain constraint. We generate a formal constraint satisfaction problem, encoded as an ILP, from the query results automatically on the basis of the software design model and hardware platform design model. This formal specification is solved by an ILP solver and the solution is translated back to a feasible allocation specification.

Figure 4.1 shows an overview of our approach. In the upper part, it shows our system under development as the so-called context model. The context model consists of a component model COMP, e.g., the MechatronicUML component model and a hardware platform model ECU, e.g., the MechatronicUML hardware platform model that Section 4.3 describes. Engineers specify constraints using our Domain-Specific Language (DSL) via *model queries*. Section 4.5 introduces our DSL and explains how to specify constraints and its model queries. The example uses the following three queries for describing constraints: (1) Query all combinations

of component instances, whose names start with the string 'gui'; the resulting tuple elements should be collocated to the same ECU. (2) Query all combinations of component instances, whose names start with the string 'secure' and ECUs, whose names start with the string 'tpm'; the resulting tuple elements describe possible allocation candidates of the matched component instances and ECUs. (3) Query how much memory each component instance requires on each ECU and how much memory each ECU has available. The resulting tuple elements are used to constrain the memory usage during the allocation planning. Within our DSL we use OCL to specify these model queries. In the middle part, Figure 4.1 shows an excerpt of the results for each query using OCL sets and tuples. We use OCL within our DSL also to calculate values in an even more complex manner, e.g., to calculate the response time of a component instance that is required for schedulability analysis. In the lower part, Figure 4.1 shows an excerpt of a generated ILP. Our approach transforms each constraint into specific ILP constraints, which dependent on the constraint type and the query result, as shown in the left part. Furthermore, it transforms the context model into ILP variable definitions and general constraints using the context model, as shown in the right part. Finally, the ILP is solved and the solution is transformed into an allocation specification model that refers to the modeling elements of the context model. An advantage of this approach is its flexibility regarding the specification of new constraints by allocation engineers because no knowledge of the encoding and solving as a formal allocation problem is required. That means that we automatize the tedious and error-prone encoding of the allocation problem as a *system of equations* by providing automated model transformations.

Our approach solves four challenges and fulfills all stated requirements. Firstly, existing approaches for modeling CPSs, e.g., ProCom [CFMS10] or SOFA HI [PWT+08] mostly consider hardware platform specifications in a simplified manner and do not cover the required ontological concepts and properties that engineers need for allocation planning . "Further, existing approaches that have a more sophisticated support for hardware specification modeling, e.g., MARTE [MARTE] lack a systematic process for specifying hardware platforms and they do not separate the different concerns of the different involved stakeholders." [*PMDB14] Furthermore, MARTE [MARTE] offers many modeling elements that are not relevant for allocation planning. These approaches are not appropriate for defining allocation constraints and calculating feasible allocations automatically. Therefore, we provide a systematic hardware platform modeling process that separates the different concerns of the different involved stakeholders. Furthermore, we provide a formal hardware platform description language that targets the needs of allocation for CPSs by providing mechanisms and modeling elements to describe a hierarchical, connected hardware platform and its memory, computing, and network resources as well as its sensors and actuators. Secondly, existing approaches for allocation engineering, e.g., ArcheOpterix [ABGM09] or the Deployment Improvement Framework [MMM12] only offer to select and solve predefined allocation constraints, which are not sufficient for the requirements of CPSs because they do not consider resource constraints, like the maximum message response time between two software component instances. Additionally, the approaches miss the ability to add new constraints without requiring the knowledge how to encode them as a mathematical constraint satisfaction problem. Therefore, we provide a DSL by defining its syntax and semantics to describe constraints using the model query abilities of OCL [*PH15]. Thirdly, a model-driven allocation engineering approach should support to generate a feasible allocation specification automatically. Therefore, we specify a transformation of our constraint DSL to an ILP. We use existing solvers to calculate a feasible solution. Using this solution, we create an allocation specification model automatically using the result of the ILP [*PH15]. Fourthly, during the transformation, the semantics of allocation constraints should be preserved. Therefore, we provide a formal proof to ensure that our transformation to an ILP is sound by means of semantics-preservation [*HP17]. We evaluate: (1) if our model-driven allocation

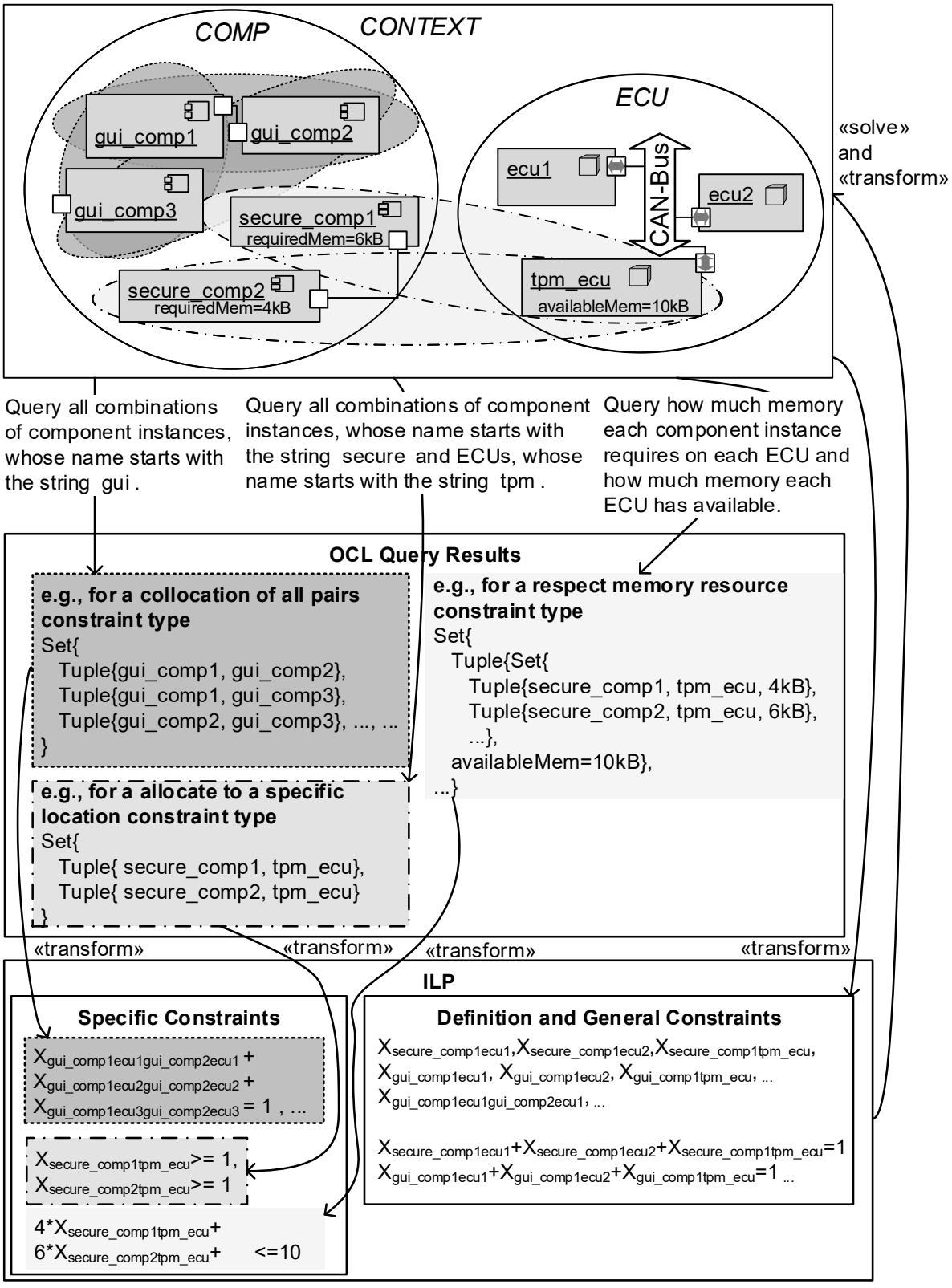


Figure 4.1: Overview: Model-Driven Allocation Engineering – From MechatronicUML to ILP and Back Again

engineering approach returns correct results, (2) if it is possible to specify common allocation constraints with less effort, and (3) the scalability of our approach. For this, we conduct a case study based on our running overtaking example and the Brake-by-wire case study from Aleti [Ale15].

The remainder of the chapter is structured as follows: The next section refines our running example system (cf. Section 1.3). Especially, it describes the properties and structure of the hardware platform and resource demands of the component instances. Furthermore, it describes allocation constraints for the example that a feasible allocation specification must fulfill. Afterward, Section 4.2 describes the engineering process and involved stakeholders for our model-driven allocation engineering approach. Subsequently, Section 4.3 describes our language to model hardware resources and hardware platforms and Section 4.4 describes our MechatronicUML extensions to model the resource requirements of component instances. Following, Section 4.5 formalizes our language to model allocation constraints of CPSs in a model-driven way. Section 4.6 describes how to automate the allocation planning within our model-driven allocation engineering approach. Based on the described model-driven allocation engineering approach, Section 4.7 demonstrates how to specify advanced allocation constraints of CPSs. The result is an allocation specification model for MechatronicUML that stores the computed feasible solution. This feasible allocation specification model can be visualized within the allocation specification view that Section 4.8 introduces. The whole allocation planning task is transparent for allocation engineers. They obtain a feasible solution without having to know how to encode and solve the allocation problem formally. Section 4.9 describes how we implement the approach using our MechatronicUML Tool Suite. Furthermore, Section 4.10 provides an evaluation of the approach via a case-study and Section 4.11 explains the limitations of our approach. The chapter ends with describing related work in Section 4.12 and giving a summary in Section 4.13.

The content of this chapter is based on the papers [*Poh13; *PMDB14; *PH15; *PHM16; *HP17; *PH17]. The Bachelor's thesis of Dann [Dan13] contributes concepts for the hardware description language. The Bachelor's thesis of Hüwe [Hüw13] and the seminar theses [*SDP15] and the final document [BCD+14] of the project group Cybertron contribute concepts for the allocation engineering approach.

4.1 Allocation Engineering Example

We create a software design, a hardware platform design, and define allocation constraints for introducing our model-driven allocation engineering approach. The designs represent the autonomous cooperating overtaking maneuver (cf. Section 1.3). For simplicity, we use a robot as a testbed because it is less complex than a complete car. The hardware platform of the robot uses Arduino-based ECUs [Ard]. The electronic circuit design of Arduino-based ECUs are publicly available as open-source, which eases to reproduce our results.

The system Robot consists of the optional platform parts Telematics and Assistance and the mandatory platform parts Chassis and HMI. Figure 4.2 shows the schema of our hardware platform prototype. The platform part Telematics has one Arduino Nano ECU that is connected to a color sensor via an I2C bus and has a wifi network connector. The platform part HMI has one Arduino Nano ECU that is connected to a display via an I2C bus. The platform part Assistance has two Arduino Nano ECUs that are connected with each other via a CAN bus. The platform part Chassis has two Arduino Gemma ECUs, which are connected to a distance sensor and two DC motors via a CAN bus. From each platform part, at least one ECU is connected to a CAN bus.

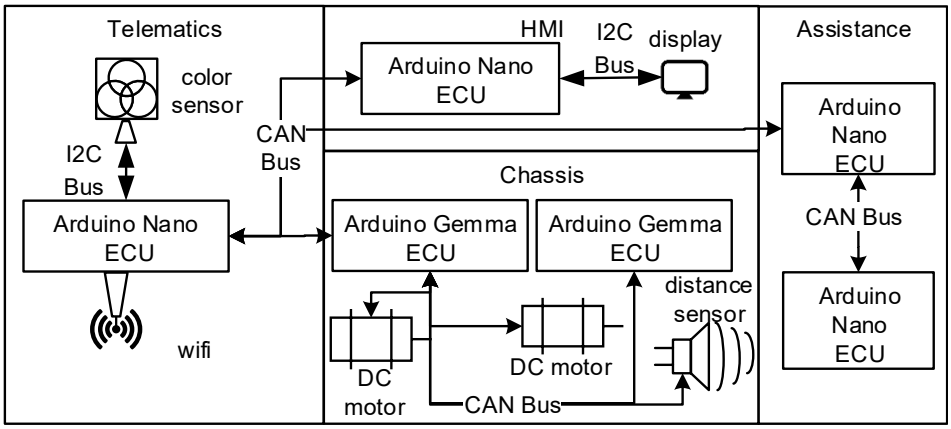


Figure 4.2: Hardware Platform Schema of the Robot System

We use MechatronicUML as a modeling language for describing the software component model. Furthermore, we define allocation requirements that have to be considered when planning the allocation specification of component instances to ECUs.

Figure 4.3 shows the software component instance diagram of the robot that plays the overtakee role. It is an extended version of the component diagram that we use in Chapter 3 for demonstrating the Modelica transformation of a structured component instance (cf. Figure 3.16). The structured component instance overtakee contains the discrete component instances overtakeeCommunicator, overtakeeDriver, and overtakeeHMI. The component instance overtakeeCommunicator can send messages to the component instance overtakeeDriver and to the component instance overtakeeHMI. Furthermore, the component instance overtakeeCommunicator can communicate with other systems via its delegation ports. Moreover, the structured component instance overtakee contains the continuous component instances overtakeeColor, overtakeeDisplay, overtakeeDistance, overtakeeMotorL, and overtakeeMotorR. The continuous component instance overtakeeColor is assembled with the discrete component instance overtakeeCommunicator and can send a color signal value. The continuous component instance overtakeeDisplay is assembled with the discrete component instance overtakeeHMI and receives a code signal. The continuous component instance overtakeeDistance is assembled with the discrete component instance overtakeeDriver and provides a distance signal. Lastly, the discrete component instance overtakeeDriver is assembled with two motor component instances and provides the reference velocity signals (velL,velR) to these motors.

An allocation specification has to satisfy several constraints, which we present in the following. We label the constraints to refer to them in the evaluation in Section 4.10.

A.1 For security reasons, it is not allowed that the component instance overtakeeCommunicator and the component instance overtakeeDriver are allocated to the same ECU. That means, it should not be possible to get access to the overtakeeDriver component instance, e.g., by a buffer overflow of the overtakeeCommunicator component instance, if an attacker manipulates the overtakeeCommunicator component instance via its external communication interface. The component instance overtakeeCommunicator encapsulates the external communication.

A.2 Furthermore, the component instance overtakeeHMI and the component instance overtakeeDisplay have to be allocated to the same ECU as they have a high communication rate.

A.3 The component instance overtakeeDriver has to be allocated to an ECU of the Chassis platform part due to a legacy engineering requirement.

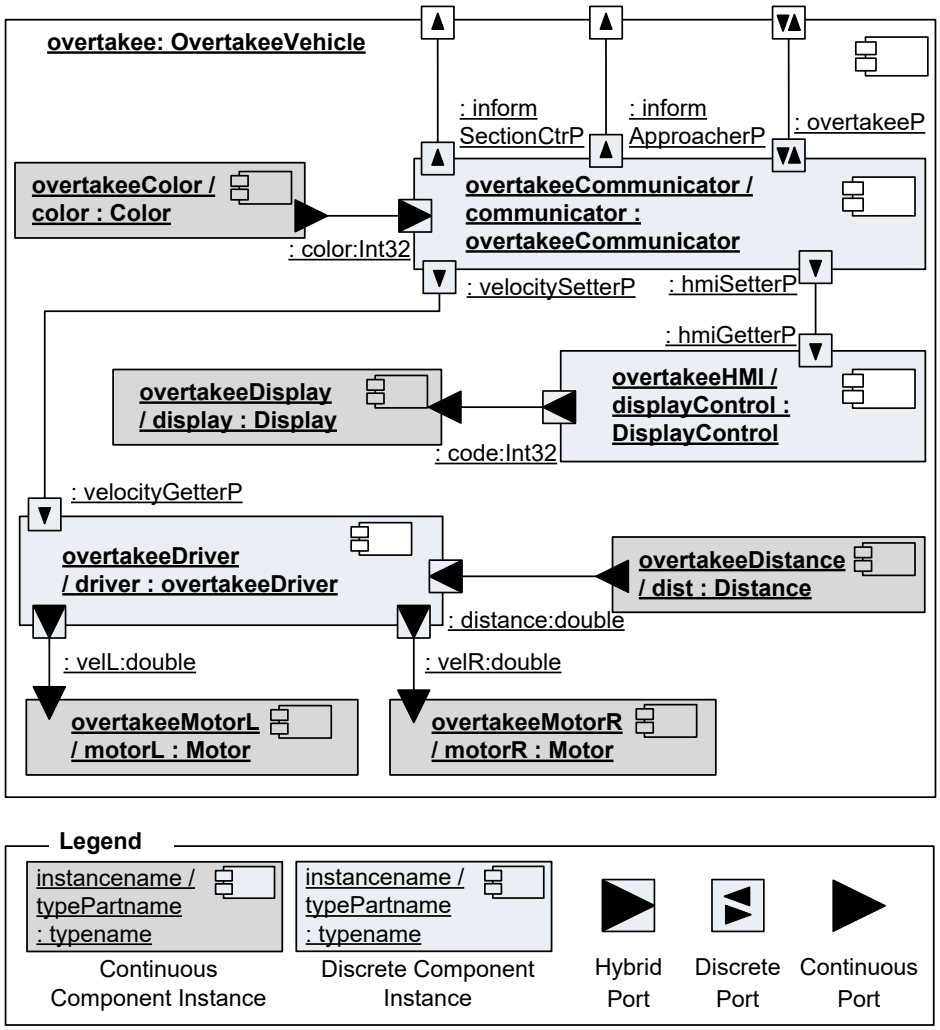


Figure 4.3: Software Component Instance Configuration Model for a Concrete Robot Variant

A.4 Additionally, a feasible allocation should always guarantee that component instances, which communicate with each other, are allocated to the same ECU or to ECUs that are connected via a communication bus or via a point-to-point network.

A.5 Moreover, component instances that read values from a sensor or write values to an actuator should have access to the corresponding device.

A.6 Component instances require resources for being executed. A typical resource constraint is the required memory that is needed to store data during execution [Ale15]. This constraint requires that the available memory of an ECU is always greater than or equal to the consumed memory of all component instances that are allocated to the corresponding ECU.

Another typical constraint is schedulability, i.e., the required processing time to fulfill a hard real-time scheduling. The execution of software by an ECU takes processing time. Operating systems that execute software on ECUs represent software units as *tasks*, instead of component instances. In a Real-Time Operating System (RTOS), a task has task properties, like a fixed or dynamic priority, a period, a deadline, and a Worst-Case Execution Time (WCET). The *scheduler* of an operating system defines during runtime, which task is executed at which time. It uses the task properties and a scheduling strategy to make that decision.

Schedulers that interrupt an executed task to give priority to another task are called preemptive schedulers [Kop11]. A feasible allocation specification should always guarantee the schedulability of tasks. We use two different task-schedulability analysis strategies to show how to handle scheduling-constraints during allocation engineering. Thereby, we refer to the Earliest Deadline First (EDF) schedulability test for dynamic prioritized, preemptive, periodic tasks and the worst-case Response-Time Analysis (RTA) schedulability test for static prioritized, preemptive, periodic tasks [ZP13]. Tasks require a deadline. EDF is a sufficient schedulability condition under the assumption that the deadline <= period. RTA is a necessary and sufficient schedulability condition with arbitrary deadlines [ZP13].

A.7 The processor utilization is always below or equal to 100% using an EDF scheduling for preemptive, dynamically prioritized component instances or tasks, which are allocated to the corresponding ECU [ZP13; LL73].

Moreover, the transmission of signals and messages by a network or communication bus takes time because we consider a distributed software allocation. Therefore, network/bus bandwidth is a required resource for message transmission. A feasible allocation should always guarantee the schedulability of message transmissions. Signals and messages are transmitted within a data frame on a communication media. The access to a communication media is limited as only one data frame can be transmitted at the same time. In real-time networks, a task has message properties, like a priority, a period, a deadline, and a transmission time. The schedulability of data transmission depends on the media access control algorithm and the message properties. In real-time systems, the media access is controlled by a media access control scheduler. A *scheduler* defines, during runtime, which message is transmitted at which time. For example, the Control Area Network (CAN) protocol uses a priority-based media access strategy. A CAN data frame has an identifier (ID), which represents the priority of the message. We use the transmission schedulability analysis strategy of Davis *et al.* [DBBL07] to show how to handle network scheduling-constraints during allocation engineering. This analysis calculates the *worst-case response times* of all CAN messages. The analysis proves timing-correctness for the CAN-based system and guarantees that all messages and signals that share a CAN bus meet their deadlines. It is a necessary and sufficient schedulability analysis. Deadlines are defined based on its safety requirements as a result of an end-to-end deadline (time for a critical action/response to an event within a distributed system) [SH96].

A.8 Communicating component instances are allocated such that all messages meet their deadline [DBBL07].

4.2 Allocation Engineering Process

In this chapter, we create a model-driven allocation engineering method using a new DSL. Good practice for designing, implementing, and using domain-specific languages is to define a process besides defining a DSL and a tooling that eases the usage of the DSL [VBD+13]. Our new DSL should be used to specify constraints to calculate a feasible allocation specification of component instances to ECUs automatically for the sketched Arduino robot.

"The design of the language should follow the principles of abstraction, partitioning, and temporal segmentation of modeler tasks as defined by Kopetz [Kop11, p.35]. The principle of abstraction makes it possible to handle the complexity, partitioning separates the different concerns, and temporal segmentation allows tasks that can be processed sequentially and reduces the amount of information that must be processed in parallel. The language should support the specification of the different parts and abstraction levels (...). The language should

abstract from unnecessary details. On one hand, it should separate the different concerns and on the other hand, it should link the specified information. Established concepts from view-based [GBB12] software engineering already provide such abilities. View-based modeling distinguishes viewpoints and views (cf. ISO/IEC 42010 [ISO42010]). A viewpoint on a system specifies a set of architectural concepts and structuring rules to focus on a particular concern of a stakeholder [GBB12; FPK+12]. 'A view defines the presentation of model elements to a stakeholder and the way(s) how they can be modified.' [FPK+12]." [*PMDB14]

This section describes the stakeholders and their concerns in model-driven allocation engineering for the safe deployment with respect to our modeling process with four viewpoints and five process' tasks. Figure 4.4 shows this process using the Business Process Model and Notation (BPMN) [BPMN] with stakeholders and their viewpoints shown as BPMN lanes. The process involves hardware specialists that are familiar with hardware data sheets and hardware platform architects that are familiar with the structural composition of hardware platforms. Furthermore, the process involves software engineers who are familiar with the resource consumption and runtime properties of the software on different hardware resources. Additionally, the process involves allocation engineers that are familiar with the required allocation constraints for the developed application in the context of the hardware platform. Therefore, they need special knowledge about the final software architecture and hardware platform and the dependencies between them. In the following, we explain the process tasks, stakeholders, and viewpoints in more detail.

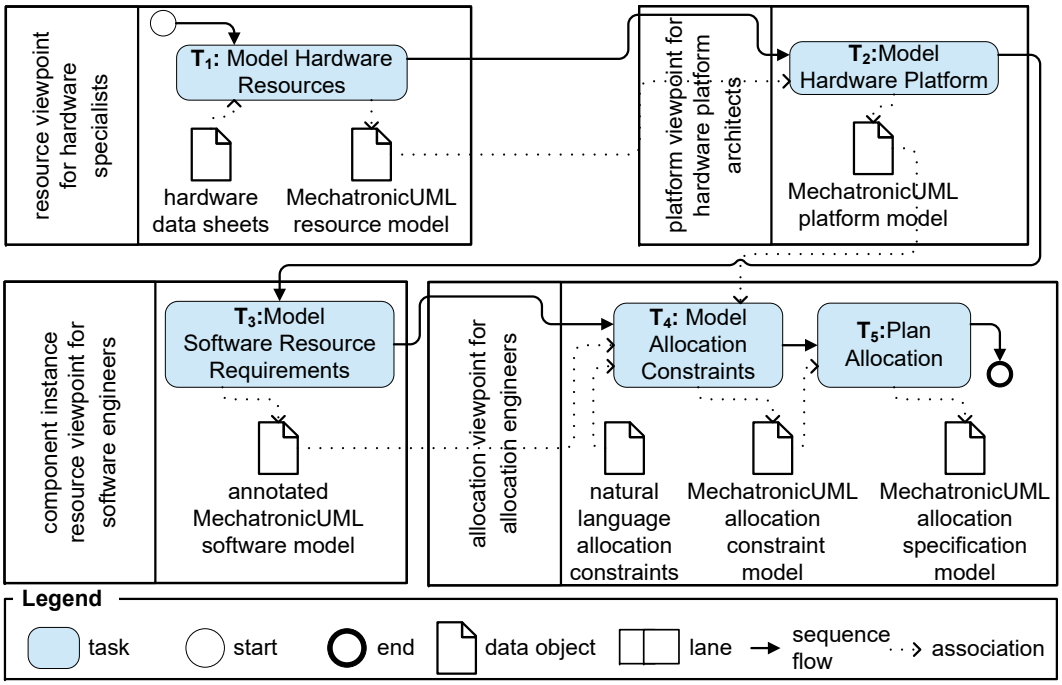


Figure 4.4: Allocation Engineering Process

"We call the viewpoint for the hardware specialists that describe the resources the resource viewpoint." [*PMDB14] The sources of the process are hardware data sheets, which hardware specialists have to transform into formal machine-readable models that contain detailed resource information. "This information about computing resources, like CPU clock rate, is needed for worst-case execution time (WCET) analyses. Furthermore, WCETs are needed for allocation planning to decide, how many components can be executed fast enough on one resource. In addition, allocation planning needs the information about memory resources in order to

decide if memory requirements of components can be fulfilled. Hardware specialists describe in (...) [Task T_1] the different types of hardware resources." [*PMDB14] They model an ECU type, e.g., an Arduino-based ECU and store the result in a MechatronicUML hardware resource model. Furthermore, "hardware specialists specify different instances of a resource type by parameterizing certain predefined parameter values, like the speed of a processor or size of a memory." They model a resource instance as instances of the resource type, e.g., the Arduino Nano and Arduino Gemma as instances of the Arduino ECU type. Hardware specialists save their work by storing it within a MechatronicUML resource model. Section 4.3.2 describes the corresponding view. Hardware platform architects can open and use this model in the next task.

"We call the viewpoint for the hardware platform architects the platform viewpoint. In (...) [Task T_2], hardware platform architects compose hardware platforms for a system, like a car. They choose the required hardware platform parts. A part could be a resource from the resource viewpoint or another platform part. For each part, they define a cardinality that specifies how often this part can be included in the final platform. The cardinality represents a variation point of a platform type for a product line. For example, a car could have an optional Telematics part or multiple ECUs for the Assistance system. (...) [Furthermore], hardware platform architects describe the concrete hardware platform instance. This represents a concretely implemented hardware platform of a real system (...)." [*PMDB14] We call this model, which contains the final hardware platform instance description, the MechatronicUML platform model. Section 4.3.3 describes the corresponding view.

We call the viewpoint for the software engineers that define the resource requirements of the component instances the component instance resource viewpoint. In Task T_3, software engineers can annotate required memory and real-time properties like deadlines, priorities, periods, and WCETs for different possible allocation targets. Thereby, the allocation engineering approach can calculate memory, computing, and network bandwidth consumptions for specific ECUs. We call this model that contains the extended MechatronicUML component model the annotated MechatronicUML software model. Section 4.4 introduces the corresponding view.

We call the viewpoint for the allocation engineers, which define allocation constraints and plan the allocation of component instances to hardware resources, i.e., mainly ECUs, the allocation viewpoint. Allocation "engineers have to cope with topology-, software-, and timing-dependencies and memory-, scheduling-, and routing-constraints at design time." [*PH15] In Task T_4, allocation engineers define these constraints in a formal MechatronicUML constraint model. The sources of the process Task T_4 are a MechatronicUML platform model and an annotated MechatronicUML software model and allocation constraints in natural language that are collected during the earlier requirements engineering phase [PR11; HBM+16]. Section 4.5.1 describes the corresponding view. In Task T_5, allocation engineers plan the allocation. The source of the process is the MechatronicUML allocation constraint model that contains the allocation constraints and references the annotated MechatronicUML software model and to the hardware platform model. In our approach, this process task is executed automatically without manual intervention of the allocation engineer. The final result is a MechatronicUML allocation specification model. Section 4.8 describes the corresponding view.

4.3 Hardware Platform Modeling

Planning the allocation of component instances to hardware resources requires detailed architectural knowledge of the hardware platform. This means that allocation engineers have to know how the hardware resources are distributed and connected and they have to have detailed domain knowledge of hardware resources, i.e., which capabilities, like memory, computing

power, and network latency the hardware resources provide. Therefore, Section 4.3.2 provides a language and tooling for Task 1 (Model Hardware Resources) of the allocation engineering process (cf. Figure 4.4) for CPSs. Furthermore, Section 4.3.3 provides a language and tooling for Task 2 (Model Hardware Platform) of the allocation engineering process (cf. Figure 4.4) for CPSs.

"In contrast to [platforms in] classical software engineering, the hardware platforms for CPSs are very heterogeneous and much more limited. There are resources for special tasks, like digital signal processing or image processing. The interconnection between hardware resources is very heterogeneous as well. The resources have special interfaces to connect devices like sensors or actuators. Further, the topology is logically structured dependent on the fulfilled task. Figure 4.5 shows a typical in-vehicle network topology. The figure illustrates that for the powertrain, telematics, body/comfort, and chassis, different networks are used that are connected via a gateway. Each sub-module, like the infotainment system, could be split up again. The many different variants in which CPS will be developed, add another dimension of complexity. (...) As a result, the development gets more complicated and this leads to an error-prone development." [*PMDB14]

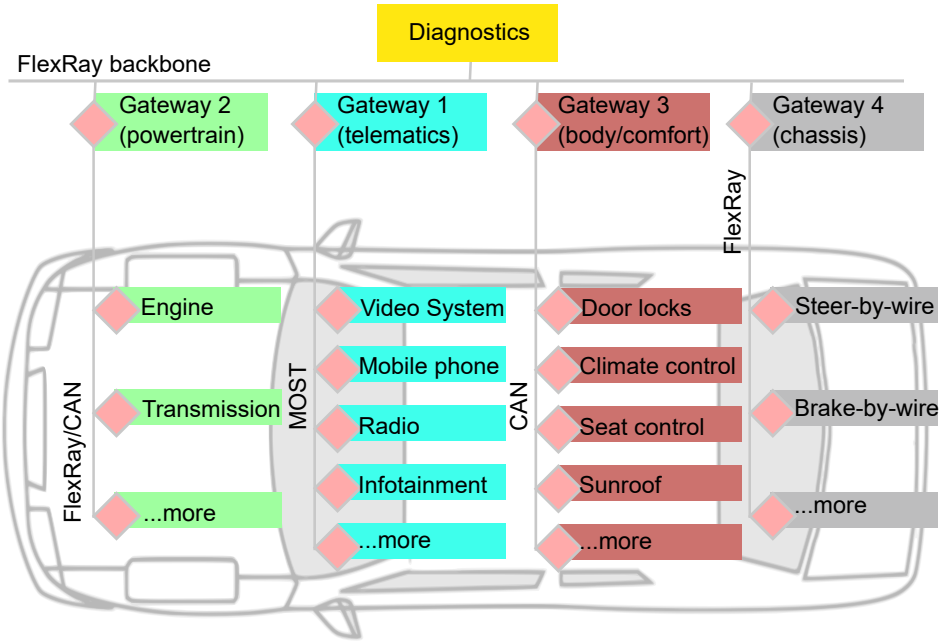


Figure 4.5: In-Vehicle Network Topology (Adapted from [*PMDB14; All09])

4.3.1 Overview

We create the MechatronicUML Hardware Platform Description Language (HPDL) for hardware specialists and hardware platform architects to describe the hardware capabilities, resources, and structure [*PMDB14]. The HPDL provides the ontological concepts and properties for allocation planning for CPSs. We use *ontological instance of* and *ontological typed via* relations to reuse and refine modeling elements [AK03]. The HPDL is a *linguistic instance of* the Eclipse EMF meta-model. For our HPDL, we provide a syntax and semantics specification as a technical report [*DP14]. Further, we provide the abstract syntax specification as an Eclipse-based meta-model. We provide for each view a graphical editor and integrate all editors into the MechatronicUML Tool Suite [*DGB+14]. Our approach is inspired by

Modeling and Analysis of Real-Time Embedded Systems (MARTE) [MARTE]. We do not use MARTE directly because MARTE offers many modeling elements that are not relevant for allocation planning. Moreover, MARTE does not differ between the hardware resources for the execution of software and a hardware platform that defines the hardware architecture. The hardware architecture description is useful to group the hardware into several independent logical parts. We need this for defining allocation constraints on the basis of specific platform parts. Furthermore, MARTE is designed as a UML profile, which is not compatible with the MechatronicUML meta-model.

Figure 4.6 shows a summary of the main classes and their relation to each other of the abstract syntax of the HPDL as a meta-model. Furthermore, it shows which class belongs to which viewpoint. Figures A.1–A.5 in Appendix A.1 show the complete meta-model of the HPDL.

The classes Resource and ResourceInstance belong to the Resource Viewpoint. Hardware specialists use the class Resource to define kinds of resources, like an ontological type of an ECU, a sensor, or an actuator. A ResourceInstance defines a concrete manifestation of a resource. Therefore, hardware specialists set the parameters for the speed of a processor or the concrete amount of available memory. The classes Platform, Part, ResourcePart, PlatformPart, PlatformInstanceConfiguration, and PlatformInstance define the Platform Viewpoint. A hardware Platform type is assembled of several Parts, where each Part has a multiplicity. The multiplicity defines the lower and upper bound for the number of Parts that a concrete PlatformInstanceConfiguration can instantiate. The classes Platform (component), ResourcePart (leaf), and PlatformPart (composite) use the composite pattern [GHJV95]. A Part is either a ResourcePart, which is typed via a ResourceInstance from the Resource Viewpoint or via a PlatformPart, which is typed via another Platform from the Platform Viewpoint. We use the composite pattern [GHJV95] to implement the hierarchical composition of hardware platforms. A concrete hardware platform of a product is defined as a PlatformInstanceConfiguration. A PlatformInstanceConfiguration is assembled of a fixed number of PlatformInstances. A PlatformInstance is typed by a Platform.

We describe the classes, their relations, and the concrete syntax within each view on the basis of models that we build for the Arduino-based robot (cf. Figure 4.2) in more detail in the following sections.

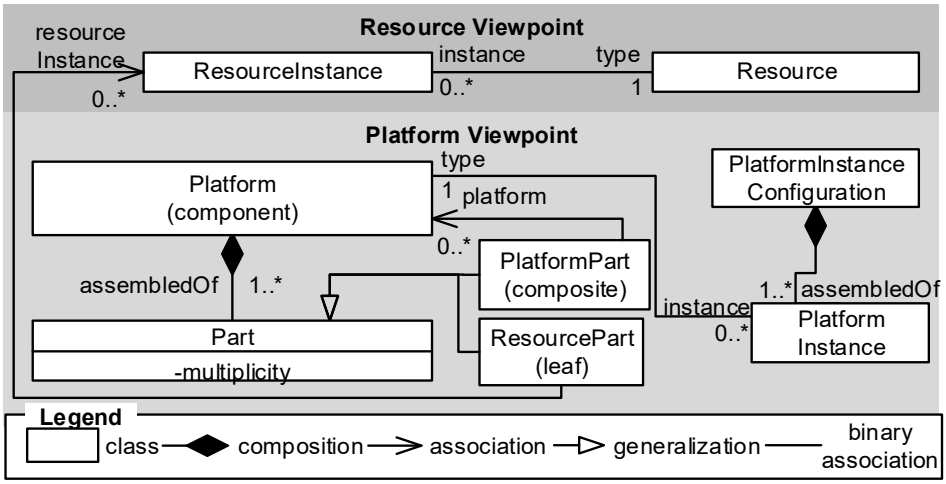


Figure 4.6: Summary of the Hardware Platform Description Language Meta-Model (Adapted from [*PMDB14])

4.3.2 Resource Views

"The resource type view is the first view for the resource viewpoint. In the resource type view, hardware specialists describe types of resources. A specified resource type can be reused in different contexts and in different variants. Referring to MARTE [MARTE, p. 92], we take a hardware resource type as a logical unit that offers capacities or services for executing software. To offer the possibility to model heterogeneous hardware resources like microcontrollers or control units, we distinguish between atomic hardware resources and structured hardware resources." [*PMDB14]

The differentiation between atomic hardware resources and structured hardware resources allows us to consider the modular structure of microcontrollers and control units. A structured resource is assembled of several atomic resources that offer basic functionality. We distinguish between the following types of atomic resources: computing resources, memory resources, and communication resources [*PMDB14].

A computing resource is a hardware resource that executes program code [BU10]. Allocation engineers require information about the computing resource for performing task-schedulability analysis [ZP13]. A computing resource is either a general-purpose processor, e.g., Intel Pentium processor or a programmable logic device, e.g., a Xilinx Spartan Field Programmable Gate Array (FPGA). A processor is characterized by its family, e.g., x86, ARM, or Atmel AVR. Furthermore, we distinguish, like the Unified Modeling Language (UML) profile MARTE [MARTE], the architectures of the instruction set that defines the operations that a processor executes. Most WCET tools, like [Lil00], use this information to analyze executables, which differ depending on the instruction set of the processor [HHL+11]. Compilers consider the processor family and instruction set to compile a program into sequences of instructions [Muc97]. Therefore, WCET analyses compute different values for different processor families and instruction sets [WEE+08]. WCET analyses, like [WEE+08], also consider the number of processor pipelines and the number of processor cores, and the cache [WEE+08]. Therefore, we offer the possibility to specify these properties for each processor.

A memory resource is a hardware resource that stores data within its memory capacity [MARTE]. Allocation engineers require information about the memory resource for ensuring that the target hardware fulfills the memory requirements of the allocated software [MMM12]. Memory resources differ within their memory kind, volatility, and access kind. Engineers can model for example that a memory resource is a RAM chip that is volatile, and has the access kind READ_WRITE. A special kind of memory is the processor cache, which is a fast buffer memory. We offer the ability to specify the associativity and the number of sets as Integer values as some WCET analyses consider these cache properties [LMW95]. Furthermore, hardware specialists can choose the replacement policies and the offered write policies because WCET analyses consider these cache properties [LMW95].

A communication resource is a hardware resource that is an interface for data exchange and to transfer data between distributed hardware resources. Allocation engineers require information about the communication resource for performing message-schedulability analysis like [DBBL07]. A communication resource can use multiple hardware pins for transmitting an electrical signal. We only need the communication properties and not the actual hardware wiring for allocation planning because only the communication properties influence the networking of resources for sending and receiving information. Therefore, we abstract in our language from the digital pins of a hardware resource. When instantiating structured resources within a hardware platform instance, a communication resource becomes a hardware port instance. Hardware specialists have to define a multiplicity for each communication resource. The multiplicity determines the number of allowed hardware port instances within a platform instance. For example an Arduino may have [0..2] CAN hardware ports.

Additionally, it is mandatory for hardware specialists to define the communication protocol that a hardware resource uses. Allocation engineers require information about the communication protocol for performing message-schedulability analysis because different communication protocols require different analysis. The approach by Davis *et al.* [DBBL07] provides a schedulability analysis for the event-triggered CAN protocol and the approach of Lukasiewycz [Luk10] provides a schedulability analysis for the time-triggered FlexRay [ISO17458] protocol. A communication protocol is a data transmission protocol within a point-to-point network, like Transmission Control Protocol (TCP) [TCP] or User Datagram Protocol (UDP) [UDP], or a data transmission protocol within a communication bus, like CAN [ISO11898]. It defines a contract for encoding and transmitting data. We offer hardware specialists to choose between several link protocol kinds, e.g., BLUETOOTH and ETHERNET. Additionally, we offer several bus protocol kinds, e.g., CAN and I2C. Hardware specialists can define if the network protocol is event-triggered or time-triggered, which is important to perform a schedulability analysis. Schedulability that guarantees a worst-case response time of network messages within a distributed safety-critical real-time system is a necessary condition for planning an allocation. Further properties that influence the schedulability of a network message are, if the data transmission is serial, the network bandwidth, i.e., the data transmission rate, and the data frame size. Communication protocols can be grouped and stored in protocol repositories, e.g., to distinguish between Ethernet-based protocols and Fieldbus protocols. Currently, hardware specialists must take care by themselves that the chosen data transmission protocol, link/bus protocol kinds, and trigger-kind are compatible to each other because we do not specify meaningful modeling constraints yet.

A special kind of an atomic resource that cannot be embedded within a structured resource is a device. A device is a hardware resource, which is responsible for interacting with its environment. A device is either a SENSOR or an ACTUATOR. A device can also have communication resources to exchange data with structured resources.

Figure 4.7 shows the resource type view for our Arduino robot example (cf. Figure 4.2), which hardware specialists should describe within our language. An Arduino board can be modeled as the structured resource type Arduino [Ard]. As computing resource, it has a processor from the Atmel AVR family with a RISC architecture and one processing core. It has a volatile memory resource of kind RAM and a non-volatile memory resource of kind Flash as memory resources. Furthermore, it has three different kinds of optional communication resources: (1) the communication resource canNI with a multiplicity of [0..2] that uses the CAN bus protocol; (2) the communication resource wifiNI with a multiplicity of [0..1] that uses the TCP network protocol; (3) the communication resource i2cNI with a multiplicity of [0..1] that uses the I2C bus protocol. The bus protocols I2C and CAN with a data rate of *1 Mbit/s* are grouped into the network protocol repository Fieldbus. The network protocols TCP and UDP with a data rate of *10 Mbit/s* are grouped into the network protocol repository Ethernet. The interaction with the environment is realized in our example with four devices: (1) a distance sensor; (2) a color sensor; (3) a motor actuator; (4) a display actuator. All devices have a single bus port that uses either the I2C protocol or the CAN protocol.

"The resource instance view is the second view for the resource viewpoint. In the resource instance view, a resource type can be concretized to a resource instance. In contrast to the resource type view, hardware specialists specify concrete capacities for the different atomic resources within a structured resource. For example, they can set the concrete RAM size or the CPU speed. Thus, hardware specialists can reuse one specified resource type in several contexts by parameterization of the resource instances, which differ in the specified capacities. The resulting resource instances act as the type specification for the resource parts in the following platform views." [*PMDB14] Communication resources are transformed to hardware ports automatically.

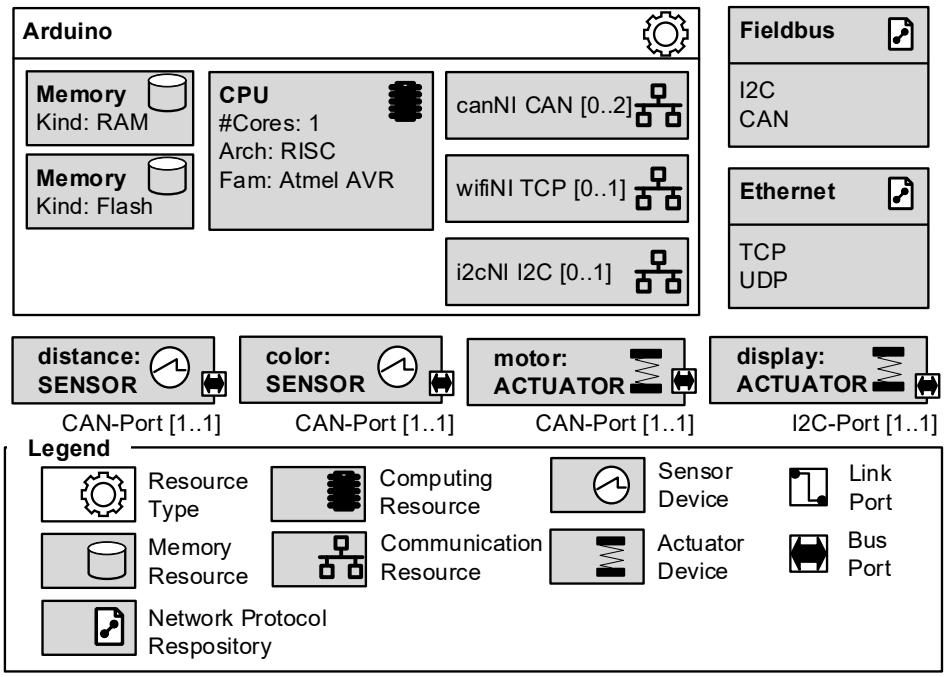


Figure 4.7: Resource Type View

In particular, a processor instance gets a Float value that defines the average Cycles Per Instruction (CPI), i.e., the number of cycles that it needs to handle a single instruction in average and a Float value that defines the Million Instructions Per Second (MIPS) for defining the processor power in more detail [AR04]. Both values can be used to calculate the WCET of a program, e.g., via the CPI value [LM95; IT97] or via the MIPS value [Lil00]. A memory resource instance has the additional Float properties memory size (mandatory) and throughput (optional). The memory size can be specified using the data size units Bit, Byte, KByte, MByte, and GByte. The throughput can be specified using the data rate units bits per second, kilobits per second, and megabits per second. We use the memory size to determine if the required memory of an allocation does not exceed the available memory [*PH15]. The throughput as an optional parameter can be used to allocate component instances that read and write much data only to an ECU that can handle the throughput requirement [*PMDB14].

Figure 4.8 shows the resource instance view for the resource type Arduino (cf. Figure 4.7). Firstly, hardware specialists instantiate the Gemma resource instance to model the Arduino Gemma board and to specify concrete capacities for the embedded processor resource and the memory resources. For the computing resource ATtiny85 hardware specialists specify that the number of CPI is equal to 2 (most instructions need only 1 cycle, but in the worst-case they need 2 cycles), the number of MIPS is equal to 8 (the used ATtiny85 provides 1 MIPS per MHz), and the speed (clock rate) is equal to 8 MHz. For the embedded memory instances RAM and Flash, hardware suppliers specify the size. The embedded memory resource RAM has a capacity of 512 Bytes and the memory resource Flash has a capacity of 8 KBytes. Secondly, hardware specialists instantiate the Nano resource instance to model the Arduino nano board. The computing resource ATmega168 requires in the worst-case 2 CPI and calculates 16 MIPS with a processor speed of 16 Mhz. The memory resources provide 1 KByte of RAM and 16 KByte of Flash.

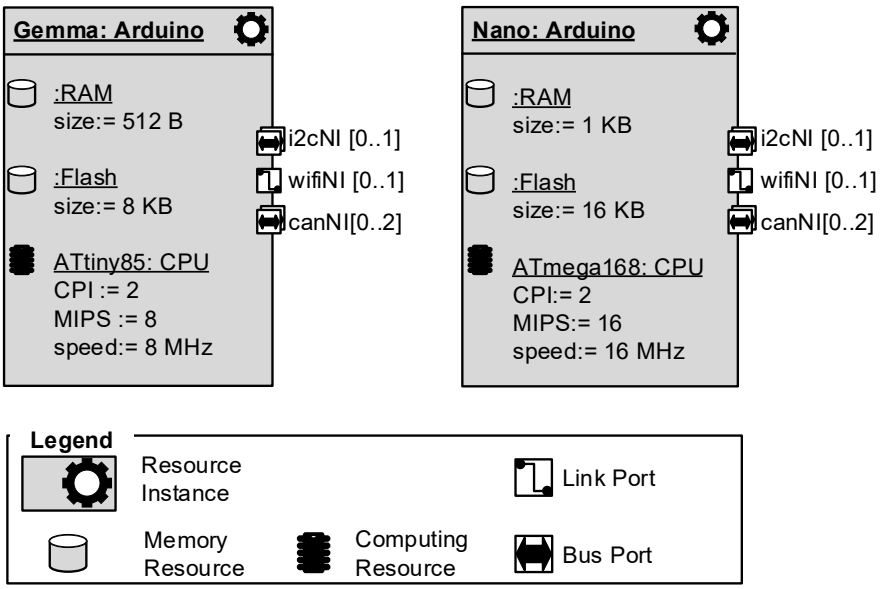


Figure 4.8: Resource Instance View

4.3.3 Platform Views

"For the platform viewpoint, we define two views. Firstly, hardware architects use the platform type view for specifying the horizontal and hierarchical composition of a hardware platform, including platform variability points. Secondly, hardware architects use the platform instance view for specifying the hardware platform of one concrete product." [*PMDB14]

A hardware platform contains different parts and allows the communication between these parts via hardware ports. New platforms can be created by composing hardware platform parts and resource instance parts. A hardware platform part is typed over another platform type and a resource instance part is typed over a resource instance. "As a result, hardware platform architects can model a hardware platform that is grouped into different subsystems and networks, by dividing the specification of a huge hardware platform into several smaller and manageable platforms. Each resource and platform part has a multiplicity for specifying variation points and constraints of a hardware product line. The multiplicity defines the lower and upper bound for instantiating. Platform architects connect the hardware ports of the resource and platform parts via communication media. A communication media is an instance of a communication resource and is either a link, e.g., Ethernet wire, or a bus, e.g., CAN. Link-ports can be directly connected with each other via a link communication media. Bus-ports are connected to a common shared media, e.g., a CAN bus. All ports that are connected to a bus can communicate with each other. Moreover, it is possible that a bridge, e.g., a gateway or a router, connects different communication media." [*PMDB14]

Figure 4.9 shows the platform type view for all platform parts of the Arduino robot example. The Arduino robot provides the five platform types (1) Telematics, (2) HMI, (3) Assistance, (4) Chassis, and (5) Robot in relation to the Arduino robot schema (cf. Figure 4.2), Thus, platform architects model the four platform types (1-4) independently and embed them as platform parts in the hierarchical platform type Robot. The multiplicities of the resource parts are used to reflect the different requirements of resources for different robot variants that are possible to build using the flexibility of the Arduino platform. The multiplicity [0..1] of the platform Robot reflects an optional platform part and the multiplicity [1..1] of the platform type Robot reflects a mandatory platform part.

The platform type Telematics consists of two embedded resource instance parts. The first resource instance part wifi_board is typed over the resource instance Nano and has a multiplicity of [1..1]. The second resource instance part is typed over the sensor color and has a multiplicity of [1..1]. The wifi_board has three hardware ports (CAN bus port, I2C bus port, and TCP link port) that inherit its multiplicity from the resource definition. Additionally, the Telematics platform type has the bus I2C. The platform type Telematics delegates the data of the link port and the CAN bus port of the embedded wiFi_board to its outside.

The platform type HMI consists of two embedded resource instance parts. The first resource instance part hmi_board is typed over the resource instance Nano with a multiplicity of [1..2]. The second resource instance part is typed over the actuator display and has a multiplicity of [1..1]. Additionally, the HMI platform type has the bus CAN and the bus I2C. The CAN network interfaces (canNI) of the Nano boards can be either connected to the local CAN bus or can be delegated to the outside. The network interfaces (i2cNI) of the Nano boards are connected via the I2C bus to the I2C-port of the display actuator.

The platform type Assistance consists of one embedded resource instance part. The assis_board is typed over the Gemma with a multiplicity of [1..4]. This means that the final robot can have from one to four instances of this board. Additionally, the Assistance platform type has the bus CAN. The CAN network interfaces (canNI) of the Nano boards can be either connected to the local CAN bus or can be delegated to the outside.

The platform type Chassis consists of three embedded resource instance parts. The first resource instance part motor_board is typed over the resource instance Gemma with a multiplicity of [1..3]. The second resource instance part is typed over the sensor distance and has a multiplicity of [1..1]. The third resource instance part is typed over the actuator motor and has a multiplicity of [2..2], as our example robot is a two-wheeled robot with two motors that can be controlled independently. Additionally, the Chassis platform type has the bus CAN. The CAN network interfaces (canNI) of the Gemma boards can be either connected to the local CAN bus or can be delegated to the outside. The CAN-ports of the distance sensor and motor actuators are connected to the local CAN bus.

Finally, platform architects compose the platform Robot of the platform parts Telematics, HMI, Assistance, and Chassis. The Telematics part and the Assistance part are optional as they have the multiplicity [0..1]. All parts are connected with each other via their CAN network interfaces and the CAN bus that belongs to the platform Robot. The wifi connection is delegated to the outside and can be used to communicate with other robots.

"For describing the concrete hardware platform of one product, we define the platform instance view for the platform viewpoint. In contrast to the platform type view, hardware platform architects define a fixed number of resource and platform parts and ports." [*PMDB14] The hardware platform instance is created semi-automatically via a Model-to-Model (M2M) transformation after specifying the number of parts. Links have to be set manually if no unique creation rule can be applied.

Figure 4.10 shows a concrete instance of the platform type Robot. It consists of a Telematics, an HMI, an Assistance, and a Chassis instance, which are connected with each other via the CAN_R bus. The Telematics instance consists of one wifi_board instance and a color sensor instance. Both are connected via an I2C bus. The wifi network interface of the wifi_board is delegated to the outside of the robot platform. Further, the HMI instance consists of an hmi_board instance and a display actuator instance that are connected via the I2C_H bus. The Assistance instance consists of two assis_board instances that are connected via the CAN_A bus. Finally, the Chassis instance consists of two motor_board instances, a distance sensor instance, and two motor actuator instances. The board and the devices are connected with each other via the CAN_C bus. We use the hardware platform instance configuration that Figure 4.10 shows

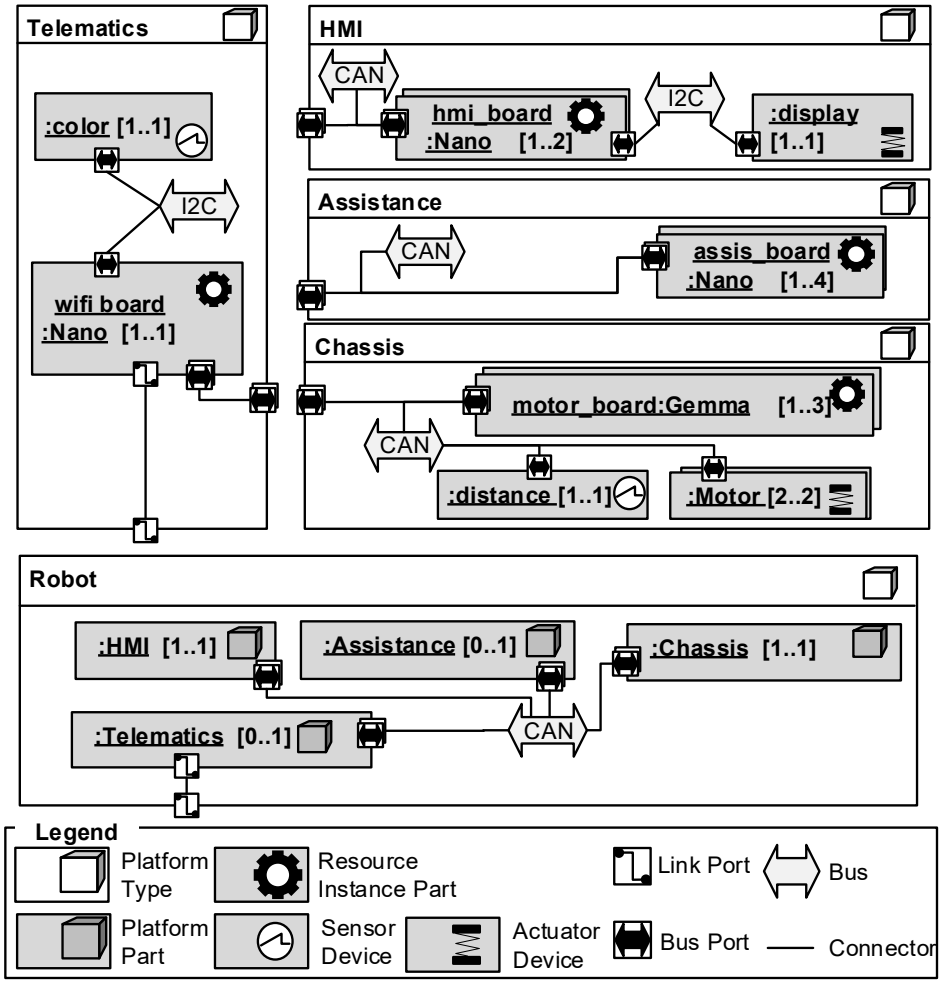


Figure 4.9: Platform Type View (Adapted from [*PMDB14])

in the case study that Section 4.10 describes to evaluate the overall model-driven allocation engineering approach.

4.4 Component Instance Resource Requirements Modeling and View

Software component instances require different resources on different ECUs. Therefore, we must be able to specify platform-specific resource requirements of component instances as the MechatronicUML component model represents only the platform-independent software properties.

Figure 4.11 shows the platform-specific modeling extensions as a class diagram that we define for defining memory, task-scheduling, and network-scheduling extensions. MechatronicUML enables to extend modeling elements via so called *extension classes*. An extension class can be compared to a UML stereotype. The class RequiredMemory extends a ComponentInstance and associates a DataSite to model its demand of required memory on a concrete ResourceInstance. Additionally, the class WCET extends a ComponentInstance and associates a TimeValue to model its demand of processing time (cf. Table 4.1) on a concrete ResourceInstance. Furthermore, the

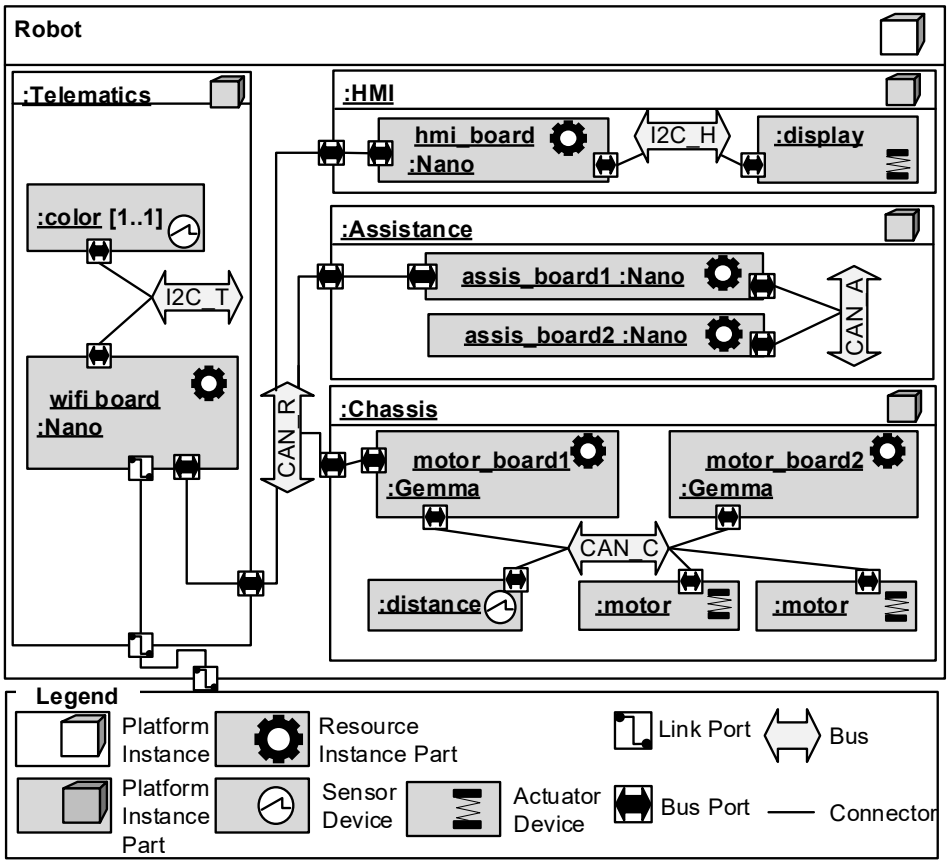


Figure 4.10: Platform Instance Configuration View (Adapted from [*PMDB14])

class Scheduling extends a ComponentInstance and associates a NaturalNumber optionally to model its priority, a TimeValue to model its period, and a TimeValue to model its deadline. Lastly, the class CANMessageFrame extends a (sending) PortInstance. It has a DataSize to model the size of a message and has a NaturalNumber to model the priority of a message.

Table 4.1 shows an example for modeling required memory and timing properties using our running example. It shows in the first column the software component instance name and in the second column the maximum required amount of memory in bytes (B) of the corresponding component instance at runtime. All component instances need together 3008 bytes if they are allocated to the same ECU. Note, we do not provide an approach for determining the worst-case memory usage of a component instance. The required memory depends on several factors, like the implementation, platform, and compiler. Chapter 5 of this thesis provides an automatic implementation of MechatronicUML models. This implementation can be used to create a software artifact for each component instance. This artifact could be started and run as a task. The memory usage of each task can be analyzed by using tools like pmap [PMAP] or Valgrind massif [SNW08]. The resulting worst-case memory consumption values must be annotated as the required memory of a component instance. Furthermore, the third column to the sixth column of Table 4.1 show from left to right for each component instance its priority, period, deadline, and WCET. Referring to [ZP13], each task has a unique priority and higher values have a higher rank than smaller values. All time values are given in milliseconds (ms). We define that each software component instance is mapped to a single task. It is possible to generate tasks and their properties from a MechatronicUML

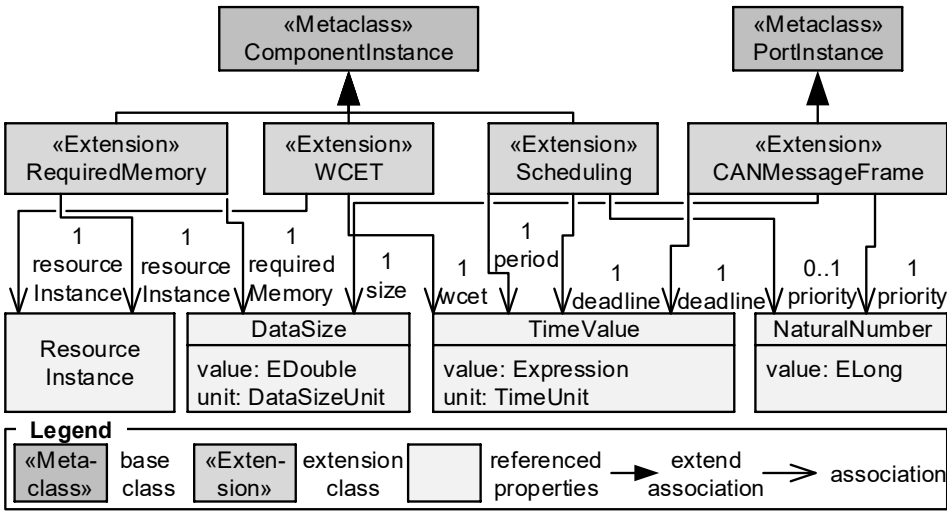


Figure 4.11: Class Diagram of Platform-Specific Model Extensions for Resource Demands

component instance configuration, which uses a defined subset of Real-Time Statechart (RTSC) semantics, automatically [*GPS17; Gei15]. Periods and deadlines could also result from safety requirements. For simplicity, we only consider the largest WCET of a component instance on any of our used ECUs. Engineers can choose between several research prototypes and commercially available tools for calculating WCETs [WEE+08]. The generated implementation from a MechatronicUML model, which Chapter 5 describes, could be a basis. Defining the values of our extensions in a systematic way is not in the scope of this thesis. Sections 4.7.5, 4.7.6, and 4.7.7 use the values from Table 4.1 for specifying resource constraints.

Table 4.1: Task Properties of the Software Component Instances

Software Component Instance / Task	**Required Memory**	**Priority**	**Period**	**Relative Deadline**	**WCET**
overtakeeColor	64 B	1	10 ms	10 ms	1 ms
overtakeeDisplay	256 B	2	10 ms	10 ms	5 ms
overtakeeDistance	128 B	3	10 ms	10 ms	1 ms
overtakeeMotorL	128 B	4	10 ms	10 ms	2 ms
overtakeeMotorR	128 B	5	10 ms	10 ms	2 ms
overakteeCommunicator	1024 B	6	40 ms	40 ms	20 ms
overakteeDriver	512 B	7	10 ms	10 ms	5 ms
overakteeHMI	768 B	0	30 ms	30 ms	10 ms

Table 4.2 shows an example for modeling message properties using our running example. It shows a row for each message that is sent or received by a device that is connected to the software component instance overtakeeDriver. For simplicity, we focus only on these messages. In the left column, Table 4.2 shows the message name. Furthermore, the other columns show from left to right for each message its priority, period, deadline, and transmission time. We assume that each MechatronicUML message and continuous signal access is mapped to a single data frame for simplicity. This assumption may not hold for complex MechatronicUML messages as a data field of a CAN data frame is limited to 8 bytes [DBBL07]. In this case, the message has to be split, sent, and reassembled, e.g., by using the proposed CAN bus protocol stack of van Glabbeek and Höfner [vGH17]. Each messages in Table 4.1 has a unique priority

and smaller values have a higher rank than larger values referring to [DBBL07]. All time values are given in milliseconds (ms). Determining the values of the described message properties is not in the scope of this thesis. We only provide the modeling elements for the described message properties of Davis *et al.* [DBBL07]. Section 4.7.8 uses the values from Table 4.2 for specifying a CAN bus resource constraint.

Table 4.2: Message Frame Properties of the Communication between overtakeeDriver and its Connected Devices

Message	Priority	Period	Relative Deadline	Transmission Time
velL	1	2.5 ms	2.5 ms	1 ms
velR	2	3.5 ms	3.25 ms	1 ms
distance	3	3.5 ms	3.25 ms	1 ms

4.5 Allocation Constraint Modeling

"'One major influence on the architecture of software systems used in the industry are constraints that need to be satisfied in order for the system to be accepted.' [ABG+13] In particular, engineers have to cope with topology-, software-, and timing-dependencies and memory-, scheduling-, and routing-constraints at design time. '(...) [C]onstraint satisfaction is a crucial aspect (...) in the design of embedded systems. However, constraints add more complexity to the [allocation] problem.' [ABG+13] In realistic automotive systems, the constraint set is very large and could reach about 126 thousand equations [ZP13]. Currently, engineers use approaches like linear programming [Luk10] or SAT-based techniques [ZP13] [or CLP [JL87; JM94]] to encode the allocation problem and to generate a feasible allocation specification automatically that fulfills all constraints. Encoding the allocation problem as a linear program [or as plain logic constraints that link variables [WW98]] is a complex and error-prone task. An example of a constraint is that each software component is allocated to exactly one ECU. Therefore, a binary decision variable x_{c_i,e_j} is required for each possible allocation to specify such a constraint. $x_{c_1,e_2} = 1$ means that the software component c_1 is allocated to the ECU e_2. Analogously, $x_{c_1,e_1} = 0$ means that the software component c_1 is not allocated to the ECU e_1. For all software components c_i the constraint $\sum_{j=1}^{100} x_{c_i,e_j} = 1$ must hold to be allocated exactly once." [*PH15]

The following Section 4.5.1 describes a new view for the allocation constraint modeling that software engineers and allocation engineers use to specify allocation constraints in a model-driven way. Thereafter, Section 4.5.2 describes our OCL-based allocation specification library that eases constraint modellers the specification of allocation constraints in the context of MechatronicUML.

4.5.1 Allocation Constraint View

The allocation constraint view is the first view for the allocation viewpoint. The view is a textual view. In the allocation constraint view, allocation engineers describe all constraints that a feasible allocation specification has to fulfill. Calculating an allocation of component instances to hardware resources automatically requires a formal model of allocation constraints. The foundations of the formal constraint model are (1) the software component instance model (cf. Figure 4.3), (2) its resource requirements model (cf. Table 4.1, Table 4.2), and (3) the hardware platform instance model (cf. Figure 4.10).

We introduced in [*PH15] the textual language named Allocation Specification Language (ASL) and tooling for modeling allocation constraints. We use the language for Task T_3 (Model Allocation Constraints) of the allocation engineering process (cf. Figure 4.4). "The primary design goal of the ASL is to provide a means to the user for specifying allocation constraints in an easy and expressive way. In order to achieve this, the language embeds OCL [OCL], which is widely used in the model-driven world. We use OCL to select allocation candidates that should fulfill the corresponding constraint (cf. Figure 4.1 on Page 87). Which allocation candidates are selected for the final feasible solution is calculated during the allocation planning by a constraint solving algorithm (cf. Section 4.6). Furthermore, in the case of resource constraints we calculate using OCL the amount that each corresponding allocation candidate requires of the corresponding resource if it is selected by the constraint solver during the allocation planning. We assume that the potential users of the ASL are already familiar with OCL." [*PH15]

Embedding OCL into our language enables to use all OCL standard operations and this allows to include self-written OCL libraries and its operations stored in a library by using the OCL include statement. We provide the self-written OCL library as a standard allocation constraint library for MechatronicUML (cf. Appendix B.4) as an Eclipse plugin (cf. Section 4.9). The OCL library simplifies the allocation constraint specification based on the MechatronicUML component instance model and the MechatronicUML hardware platform instance model because it provides a rich set of OCL operations. For example, these operations help allocation engineers to get all component instances, the children of a structured component instance, or provide for all ECUs the memory value.

The ASL enables to refer to the existing models and gives allocation engineers the freedom to model the required allocation constraints. Its provides a fixed set of features that are defined formally and allow an automatic allocation planning. Figure 4.12 shows the textual structure of an allocation constraint specification within the ASL. The underlying meta-model of the ASL is described in Appendix B.2. A constraint always begins with the ASL keyword **constraint**, followed by one of the following keywords that define the type of the constraint. We support the four different allocation constraint types: **collocation**, **separateLocation**, **requiredLocation**, and **requiredResource**. We used this literature [MBAG11; ŠCV13b; Luk10; ZP13; MMM12] to identify the required allocation constraint types for CPSs. The allocation constraint type defines the semantics of an allocation constraint. Further, allocation engineers have to define a name for each allocation constraint. Following, allocation engineers have to define the named parts of the OCL expression and their types. The definition starts with the keyword **descriptors**. The result of the OCL expression must bind corresponding objects or values of the evaluated model to these parts. The different allocation types require different named parts. The allocation type **requiredResource** requires two additional special named parts defined by using the keywords **weight** and **bound**. We describe the possible named parts in more detail in the corresponding Sections 4.5.1.3-4.5.1.6. Lastly, allocation engineers specify which parts of the specification model should be affected by the allocation constraint. Therefore, they have to specify an OCL expression that binds objects or values to the corresponding parts. The result is a set of several OCL tuples, where each tuple binds the required named parts.

Figure 4.13 shows an example how we specify an allocation constraint and evaluate it in the context of a concrete model. In the upper left part, Figure 4.13 shows the constraint named feasibleTargets of the type requiredLocation. The constraint defines for a fixed set of model elements of the type ComponentInstance to which targets of the type StructuredResource they can be allocated. The **descriptors** definition specifies the named parts namedPart1 and namedPart2 are typed over the corresponding types. The following OCL expression defines a binding for both named parts. We define the OCL example query in a pretty static way to explain the general concept. OCL provides a rich set of operations that enables to specify constraint smarter than the shown example. The example shows a constraint that defines that

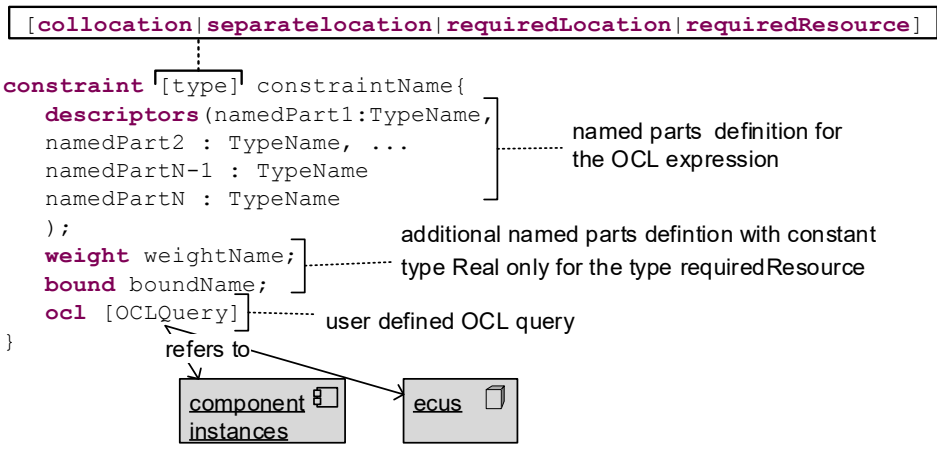


Figure 4.12: Structure of an Allocation Constraint within the ASL

the instance component1 must be allocated either to the resource ecu1 or ecu2. The example query creates a Set that consists of two Tuples. Each Tuple binds the named parts. The first Tuple searches for a component instance with the name component1 and defines the binding to the named part namedPart1 and searches for an ECU with the name ecu1 and defines the binding to the named part namedPart2. The second Tuple also searches for a component instance with the name component1 and defines the binding to the named part namedPart1 and searches for an ECU with the name ecu2 and defines the binding to the named part namedPart2.

Allocation engineers can evaluate all allocation constraints in the context of a concrete model. In the upper right part, Figure 4.13 shows an example context model consisting of the component instance component1 and the ECUs ecu1, ecu2, and ecu3. The OCL query evaluates to a concrete OCL set. In the lower part, Figure 4.13 shows the resulting OCL set that contains the corresponding tuples that bind component1 to namedPart1 and ecu1 and ecu2 to namedPart2. We use the OCL result in combination with our ASL semantics to define a transformation to an ILP for the automated allocation planning in Section 4.6.

The remainder of this section is structured as follows. Firstly, Section 4.5.1.1 defines a context model for allocation constraints for MechatronicUML because currently MechatronicUML misses a model that relates a component instance configuration to a concrete hardware platform instance configuration. Such a model is required for allocation planning. Thereafter, Sections 4.5.1.3–4.5.1.7 define the semantics of the different allocation constraint types formally and the semantics of the combination of multiple constraints.

4.5.1.1 OCL Context for MechatronicUML

An OCL expression requires a context that defines the limited situation in which the expression is valid and can be evaluated. Therefore, the ASL keyword **oclContext** refers to a concrete context class. We define a meta-model that defines this context for MechatronicUML. "We assume that all OCL expressions are specified in the syntactical context of the class OCLContext. Figure 4.14 shows that the class OCLContext has references to the classes ComponentInstanceConfiguration and HWPlatformInstanceConfiguration. Hence, it is possible to specify OCL expressions that refer to MechatronicUML language elements, like StructuredComponentInstance and HWPlatformInstance." [*PH15] The specification of the constraints within the ASL is independent of a concrete model instance. The specification could be valid for a group of model instances. If allocation engineers like to plan the allocation specification for a concrete model instance they have to choose a concrete ComponentInstanceConfiguration,

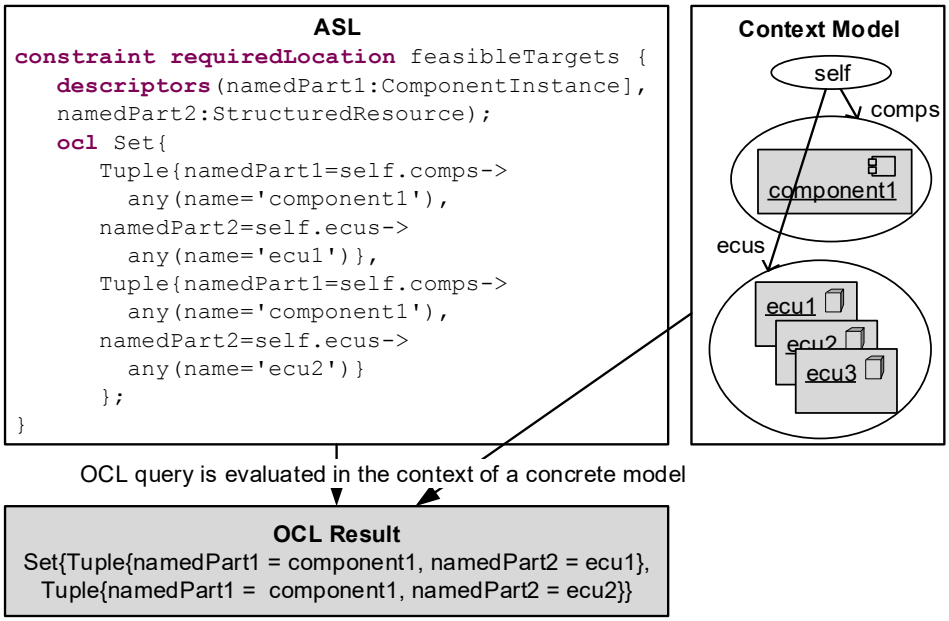


Figure 4.13: Example: Specifying and Evaluating an ASL Constraint in the Context of a Concrete Model

HWPlatformInstanceConfiguration, and the allocation specification manually using a special wizard within our tooling before evaluating the constraints in the context of the concrete model instance.

4.5.1.2 Formal Preliminaries

In the following sections we provide a formal definition of the ASL semantics. We need this definition to specify a mapping to an ILP and to prove that the mapping is correct, i.e., keeps the semantics of the constraint type. Therefore, we provide in this section the formal preliminaries that are needed to define the ASL semantics.

"Let M be the object model [OCL] that is represented by the class diagram in Figure 4.14. Moreover, let $\sigma = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})$ be a system state for the object model M. Basically, such a system state corresponds to an object diagram (σ_{CLASS} = objects, σ_{ATT} = attribute values, σ_{ASSOC} = links between objects) that is typed over the class diagram in Figure 4.14. Let OCL be the set of all syntactically correct (in the context of the class OCLContext) OCL expressions. To evaluate an OCL expression $\Psi \in OCL$, an evaluation environment is required. We assume that $eval_{\sigma(\Psi)}$ denotes the evaluation result of the OCL expression $\Psi \in OCL$ and that the evaluation environment's system state is σ. We usually omit the σ and simply write $eval(\Psi)$." [*PH15]

"Let $COMP = \sigma_{CLASS}(ComponentInstance)$ be a set of MechatronicUML component instances, e.g., (...)[the component instances overtakeeDriver, overtakeeCommunicator, ... that are shown in Figure 4.3]. Analogously, let $ECU = \sigma_{CLASS}(StructuredResourceInstance)$ be a set of MechatronicUML ECUs, e.g.,(...)[the hardware resource instances wifi_board, hmi_board, ... that are shown in Figure 4.10]. A mapping $f \in ECU^{COMP} := \{f : COMP \rightarrow ECU\}$ is called an allocation. Let $c \in COMP$ and $e \in ECU$. Then, $f(c) = e$ means that the component instance c is allocated to the ECU e." [*PH15]

"In OCL, a tuple consists of several named parts, which are used to access its elements [OCL]. Listing 4.1 demonstrates how to create and access a tuple in OCL. The first two lines define

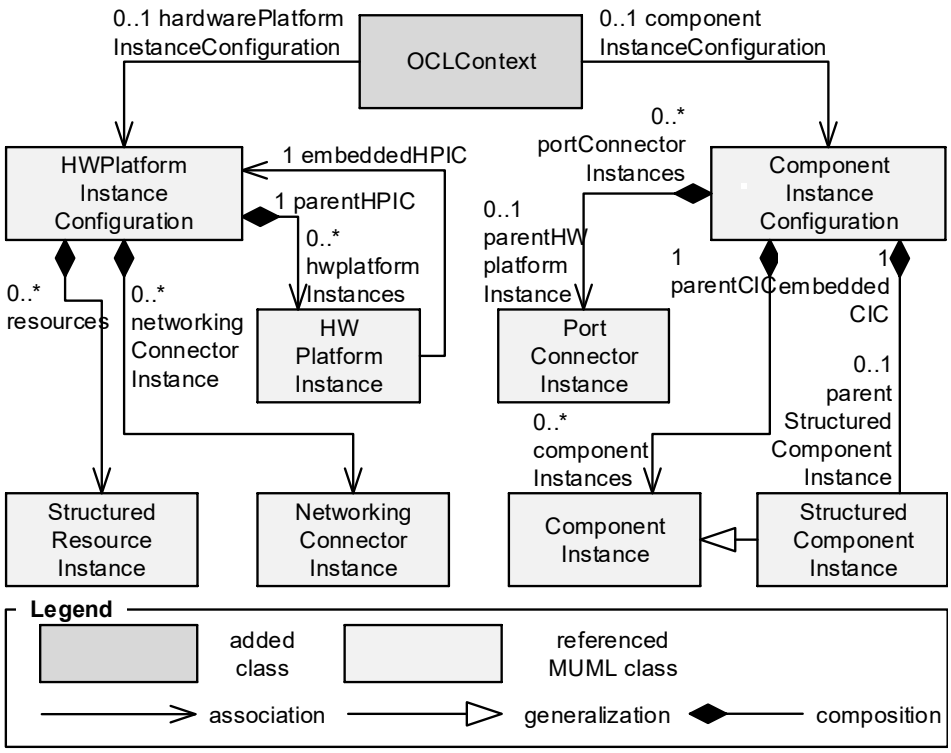


Figure 4.14: OCLContext Meta-Model (Adapted from [*PH15])

via the keyword let the temporary variable t as a tuple composed of the named parts *comp* and *ecu*. This allows us to use the tuple t within the subsequent expression. The third line constructs the new tuple object and binds the object e to the named part *ecu* and the object c to the named part *comp*. We assume that e has the type StructuredResourceInstance and c has the type ComponentInstance and both variables are available in the expression's scope. The fourth line is used to access the element that was bound to the named part *comp*. Thus, the whole OCL expression in Listing 4.1 evaluates to the object c." [*PH15]

"We interpret an OCL tuple as a mathematical tuple by using the order of the specified tuple descriptors. For convenience, we omit tuple descriptors in the following and assume the implicit order of the named parts from the tuple's type definition. Hence, the OCL tuple t in Listing 4.1 corresponds to the mathematical tuple (c, e). More generally, an OCL tuple of the type $Tuple(n_1 : T_1, ..., n_k : T_k)$, where T_i is a type, is interpreted as the mathematical tuple $(n_1, ..., n_k)$. Furthermore, an OCL set s of the type $Set(T)$, where T is a type, is considered as a mathematical set. For instance, to select an arbitrary element e from the set s the notation $e \in s$ is used." [*PH15]

```
1   let t : Tuple(comp : ComponentInstance,
    ecu : StructuredResourceInstance)
    = Tuple{ecu = e, comp = c}
    in t.comp
```

Listing 4.1: Creating and Accessing a Tuple in OCL (Adapted from [*PH15])

Formally, a constraint Φ is represented by 3-tuple $(\Phi_{type}, \Phi_{OCL_T}, \Phi_{OCL_E})$. The first element $\Phi_{type} \in \{\text{collocation}, \text{separateLocation}, \text{requiredLocation}, \text{requiredResource}\}$ represents the type of the constraint. The second element Φ_{OCL_T} represents an OCL type. The third element $\Phi_{OCL_E} \in OCL$ represents an OCL expression, whose OCL type conforms to Φ_{OCL_T}. [*PH15]

4.5.1.3 Collocation Constraint Type

"The *collocation* constraint is used to specify that two component instances have to be allocated to the same ECU." [*PH15] Thereby, it is possible to specify constraints concerning software component dependencies, e.g., to enforce that strongly coupled component instances, which have to be executed in sequence, are allocated to the same ECU. This avoids parallel execution and that safety-critical communication is performed via an unreliable bus. Furthermore, it reduces communication latency as shared memory is faster than accessing a network.

The *collocation* constraint Φ is defined as [*PH15]:

- $\Phi_{type} = \text{collocation}$
- $\Phi_{OCL_T} = Set(Tuple(c_1 : ComponentInstance, c_2 : ComponentInstance))$
- $\Phi_{OCL_E} \in OCL$

The set of feasible allocations $F_\Phi \subseteq ECU^{COMP}$ is defined as:

$$\hat{f} \in F_\Phi \overset{def.}{\iff} \forall (c_1, c_2) \in eval(\Phi_{OCL_E}) \colon \hat{f}(c_1) = \hat{f}(c_2) \tag{4.1}$$

That means for all pairs of component instances that are part of the evaluation result of the OCL expression only allocation specifications fulfill the constraint where both instances are allocated to the same result of the function $\hat{f}$, i.e., the same ECU.

4.5.1.4 Separate Location Constraint Type

"The *separateLocation* constraint is used to specify that two components have to be allocated to different ECUs." [*PH15] Thereby, it is possible to specify constraints concerning component instances, which are mutually incompatible on the same ECU. Furthermore, redundant component instances have to be allocated to different, independent ECUs to fulfill safety requirements [ISO26262-1].

The *separateLocation* constraint Φ is defined as [*PH15]:

- $\Phi_{type} = \text{separateLocation}$
- $\Phi_{OCL_T} = Set(Tuple(c_1 : ComponentInstance, c_2 : ComponentInstance))$
- $\Phi_{OCL_E} \in OCL$

The set of feasible allocations $F_\Phi \subseteq ECU^{COMP}$ is defined as:

$$\hat{f} \in F_\Phi \overset{def.}{\iff} \forall (c_1, c_2) \in eval(\Phi_{OCL_E}) \colon \hat{f}(c_1) \neq \hat{f}(c_2) \tag{4.2}$$

That means for all pairs of component instances that are part of the evaluation result of the OCL expression only allocation specifications fulfill the constraint where both instances are allocated not to the same result of the function $\hat{f}$, i.e., to different ECUs.

4.5.1.5 Required Location Constraint Type

The *requiredLocation* constraint is used to specify that a component instance has to be allocated to specific ECUs, ECUs that are part of specific platforms, or ECUs that have specific properties. [*PH15] Thereby, it is possible to specify constraints concerning topology, software-dependencies, and routing. For example, to avoid that safety-critical components are allocated to "non-secure" ECUs that are designed for entertainment purposes. These may not have a trusted platform with appropriate authentication mechanisms for access control [AEH+10]. [*PH15]

The *requiredLocation* constraint Φ is defined as [*PH15]:

- Φ_{type} = requiredLocation
- $\Phi_{OCL_T} = Set(Tuple(c_1 : ComponentInstance, e_1 : StructuredResourceInstance, \ldots, c_n : ComponentInstance, e_n : StructuredResourceInstance))$
- $\Phi_{OCL_E} \in OCL$

We calculate so-called equivalence classes of component instances in the case that a required location constraint depends on more than one component instance, e.g., $(Set(Tuple(c_1 : ComponentInstance, c_2 : ComponentInstance))$. For example, we calculate the equivalence classes within a constraint that requires that communicating components are only allocated to different ECUs if these ECUs are connected by a network link or a network bus (cf. Section 4.7.4). In this case, each pair of communicating component build an own equivalence class, i.e., an own allocation candidate. One of the potential allocation candidates for each component instance of an equivalence class must be chosen. For example, we consider our initial example from Figure 4.1 that we show in the introduction of this chapter. The component instance gui_comp1 communicates with the component instance gui_comp2. Therefore, both component instances should be allocated to the same ECU or to connected ECUs. Both component instances build together an equivalence class within the required location constraint. Feasible allocation options of this equivalence class are ((gui_comp1,ecu1),(gui_comp2,ecu1)), ((gui_comp1,ecu1),(gui_comp2,ecu2)), or ((gui_comp1,ecu1), (gui_comp2,tpm_ecu)), and further combination where gui_comp1 is allocated to one of the other ECUs. If the required location constraint refers only to a single component instance $(Set(Tuple(c_1 : ComponentInstance))$, each component instance forms its own equivalence class.

Formally equivalence classes are defined as: Let $\sim$ be a binary equivalence relation on the OCL evaluation result set $eval(\Phi_{OCL_E})$ such that:

$$(c_1, e_1, c_2, e_2, \ldots, c_n, e_n) \sim (c'_1, e'_1, c'_2, e'_2, \ldots c'_n, e'_n) \tag{4.3}$$

$$\overset{def.}{\iff} \{c_1, c_2, \ldots, c_n\} = \{c'_1, c'_2, \ldots, c'_n\} \tag{4.4}$$

The set $Q := eval(\Phi_{OCL_E})/_\sim$ denotes the quotient set, which is the set of equivalence classes for the evaluation result $eval(\Phi_{OCL_E})$. According to the definition of the Equations 4.3 and 4.4, two different tuple from an evaluation result are in the same equivalence class if both tuple contain the same component instances.

The set of feasible allocations $F_\Phi \subseteq ECU^{COMP}$ is defined as:

$$\hat{f} \in F_\Phi \overset{def.}{\iff} \forall g \in Q: \tag{4.5}$$

$$\bigvee_{(c_1, e_1, \ldots, c_n, e_n) \in g} \hat{f}(c_1) = e_1 \wedge \hat{f}(c_2) = e_2 \wedge \ldots \wedge \hat{f}(c_n) = e_n \tag{4.6}$$

As a result, at least one of the feasible allocation options for each equivalence class must be chosen. In the case that a component instance has to be allocated to exactly one ECU one of the possibilities has to be chosen. However, we do not require to choose an ECU by the required location constraint but add during our transformation to an ILP an extra constraint that requires to choose an ECU (cf. Section 4.6.3.1).

4.5.1.6 Required Resource Constraint Type

"The *requiredResource* constraint is used to specify that an allocation has to respect certain resource restrictions." [*PH15] Thereby, it is possible to specify constraints concerning memory, task-scheduling, and network-scheduling.

The *requiredResource* constraint Φ is defined as [*PH15]:

- Φ_{type} = requiredResource
- $\Phi_{OCL_T} = Set(Tuple(bound : Real, weightSet : Set(Tuple($
  $c_1 : ComponentInstance,$
  $e_1 : StructuredResourceInstance, \ldots,$
  $c_n : ComponentInstance,$
  $e_n : StructuredResourceInstance, weight : Real))))$
- $\Phi_{OCL_E} \in OCL$

The set of feasible allocations $F_\Phi \subseteq ECU^{COMP}$ is defined as:

$$\hat{f} \in F_\Phi \overset{def.}{\iff} \forall (bound, weightSet) \in eval(\Phi_{OCL_E}): \tag{4.7}$$

$$\sum_{(c_1,e_1,\ldots,c_n,e_n,weight)\in weightSet} weight \cdot h(\hat{f}, c_1, e_1, \ldots, c_n, e_n) \leq bound \tag{4.8}$$

, where

$$h(\hat{f}, c_1, e_1, \ldots, c_n, e_n) := \begin{cases} 1 & \text{if } \hat{f}(c_1) = e_1 \wedge \ldots \wedge \hat{f}(c_n) = e_n \\ 0 & \text{else} \end{cases} \tag{4.9}$$

That means that we add the weight to the sum if the allocation denoted in the tuple weightSet is part of the allocation specification ($\hat{f}$). Otherwise, we do not count the weight. For example, we consider the two communicating component instances $c1$ and $c2$ and the ECUs $e1$ and $e2$ that are connected to each other via a common communication bus. We have to consider the bus load of the communication traffic between both component instances only if component instance $c1$ is allocated to ECU $e1$ and component instance $c2$ is allocated to ECU $e2$ or vice versa. We do not have to consider the bus load of the communication traffic between both component instances if both instances are allocated to the same ECU.

4.5.1.7 Combination of Multiple Constraints

"So far, we defined for each constraint a set of feasible allocations. Since an allocation has to usually respect multiple constraints, we have to define the set of feasible allocations for a set of constraints. Let C be a set of constraints. The set of feasible allocations $F_C \subseteq ECU^{COMP}$ is defined as $F_C := \bigcap_{\Phi \in C} F_\phi$." [*PH15] That means that the set of feasible allocations consists of the intersection of all constraints. The set of feasible allocations may become empty if allocation engineers specify conflicting allocation constraints.

4.5.2 OCL-based Allocation Specification Library

"We designed the ASL to be modular so that it could be reused for different languages and is not restricted to MechatronicUML. The binding to MechatronicUML is done via a context model (cf. Figure 4.14) and the OCL-based Allocation Specification Library. The purpose of the library is to provide frequently used OCL expressions so that the users do not have to specify them manually." [*PH15]

"A library encapsulates operations that are frequently used when specifying constraints. Instead of writing operations from scratch repeatedly, it is possible to include a library and have access to all of its operations. The library *OCLContext.ocl* is a standard library for MechatronicUML that provides various operations, which ease the constraint specification." [*PH15] Appendix B.4 shows the whole library. It is possible that allocation engineers create their own libraries and import them via the include statement to ease the constraint specification.

Listing 4.2 shows four example OCL operations. We use these operations in Section 4.7.1 for defining a *collocation* constraint. The OCL keyword context defines for which class the following operations can be called. For MechatronicUML, the context is defined in general as the class oclcontext::OCLContext. "The operation allocateToSameECU gets two Strings as input and returns a Set that contains a 2-tuple in which the two components that should be collocated are referenced. For deriving these components the operation getSWInstance(instanceName:String) is called. This operation searches in all available components an instance that has the specified name. It uses the operation getAllEmbeddedInstances() via the operation getAllSWInstances(), which returns all available components in the component instance configuration, which is accessed via the OCLContext's componentInstanceConfiguration reference. The whole derivation process of the components is transparent to the user who uses the allocateToSameECU operation." [*PH15]

```
1  context oclcontext::OCLContext
   def: allocateToSameECU(instance1 : String, instance2 : String)
   : Set(Tuple(firstComponent : instance::ComponentInstance,
   secondComponent : instance::ComponentInstance)) =
5  Set{Tuple{firstComponent = self.getSWInstance(instance1),
   secondComponent = self.getSWInstance(instance2)}}

   def: getSWInstance(instanceName : String) :
   instance::ComponentInstance =
10 self.getAllSWInstances()->any(name = instanceName)

   def: getAllSWInstances() : Set(instance::ComponentInstance) =
   self.componentInstanceConfiguration.getAllEmbeddedInstances()

15 context instance::ComponentInstanceConfiguration
   def: getAllEmbeddedInstances() :
   Set(instance::ComponentInstance) =
   self.componentInstances->closure(c |
   if c.oclIsKindOf(instance::StructuredComponentInstance) then
20    c->asSet()->union(
      c.oclAsType(instance::StructuredComponentInstance).
      embeddedCIC.componentInstances)
   else  c->asSet() endif )
```

Listing 4.2: OCL Operations from the Allocation Specification Library for the Collocation Constraint (Adapted from [*PH15])

4.6 Automated Allocation Planning

Our model-driven allocation engineering approach solves the allocation problem on the basis of the specified allocation constraints automatically. The complexity of allocation planning is a NP-complete problem and the specification as a system of equations is error-prone. Allocation engineers need a method to generate the formal representation as a system of equations automatically to solve the allocation problem efficiently. We need a formal specification, like an ILP encoding, that can be solved automatically by existing solvers. Therefore, this section provides a formal ILP representation of the ASL semantics and an implemented transformation that computes feasible allocation specifications for Task 4 (Plan Allocation) of the allocation engineering process (cf. Figure 4.4). As a result, allocation engineers get the allocation of

component instances to hardware resources of CPSs automatically. The input of the process task is a MechatronicUML allocation constraint model.

In more detail, we split Task T_5 (Plan Allocation) into three subtasks. Figure 4.15 shows the refined process using the BPMN [BPMN]. The input of the process is an allocation constraint model generated from the ASL. Based on this, the OCL expressions can be evaluated to obtain the OCL tuples. In Task $T_{5.1}$ (Generate Linear Program), we provide a transformation that transforms the allocation constraint model into a constraint satisfaction problem, which Section 4.6.1 defines. For this, we use a linear program meta-model that we contribute in Section 4.6.2 as the abstract syntax to define the concrete ILP instance. Section 4.6.3 defines how we use this meta-model to encode the different ASL constraint types as a linear program model. We use a model of linear programs to decouple the allocation problem description from different possible concrete solving mechanisms. The transformation creates an ILP, which uses only binary decision variables [*PH15]. In Task $T_{5.2}$ (Solve Linear Program), a transformation converts the abstract syntax of a linear program model into a concrete input for different solvers. Appendix B.6 describes how the abstract syntax is represented as a concrete input for a solver. A solver calculates a feasible solution of the linear program if a feasible allocation specification exists, which is the result of Task $T_{5.2}$. In Task $T_{5.3}$ (Generate Allocation Specification Model), a transformation generates an allocation specification model from the solved linear program model. The output is a feasible MechatronicUML allocation specification model if the former task provides a solution. Otherwise, the allocation specification model is empty. Section 4.6.4 describes the Task $T_{5.2}$ and Task $T_{5.3}$.

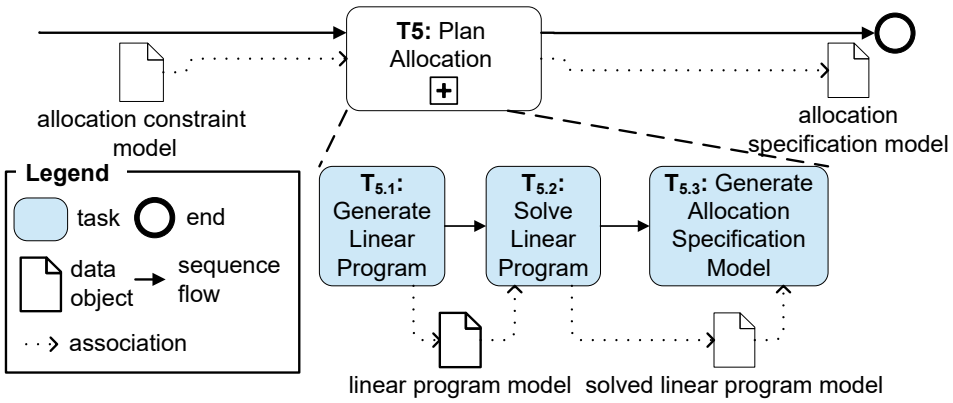


Figure 4.15: Allocation Planning Process (Adapted from [*PH15])

Figure 4.16 shows an example of how we transform an allocation constraint in the context of a concrete model to an ILP. In the upper part, Figure 4.16 shows the same ASL constraint and the same example model as in Figure 4.13 on Page 107. The OCL query evaluates to the same concrete OCL result that Figure 4.16 shows in the middle part. In the following transformation step, the OCL set is transformed to an ILP inequality. The ILP inequality depends on the OCL expression result and the constraint type because the different constraint types require a different representation within an ILP. Sections 4.5.1.5–4.5.1.6 describe the semantics of our different allocation types. On the basis of this semantics definition, Section 4.6.3 specifies the transformation of the different allocation constraint types to an ILP. Especially, Section 4.6.3.5 shows the transformation for the shown constraint type requiredLocation. In the lower left part, Figure 4.16 shows the corresponding part of the ILP. Furthermore, the transformation creates ILP allocation variables for each possible allocation and an equation, which defines that each component instance has to be allocated to exactly one ECU. Section 4.6.3.1 defines

this transformation. In the lower right part, Figure 4.16 shows the corresponding part of the ILP.

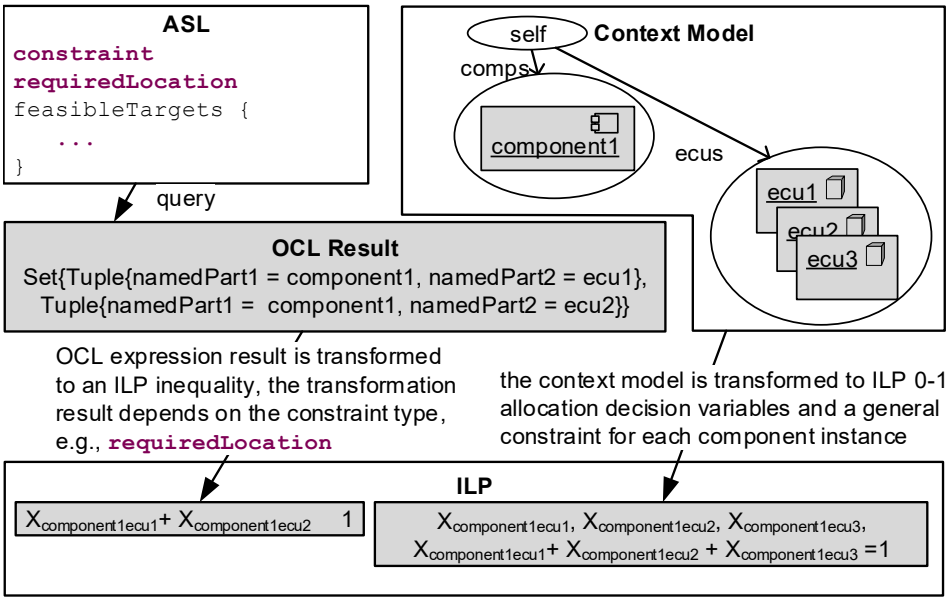


Figure 4.16: Example: From an ASL Constraint Specification to an ILP Model

4.6.1 Constraint Satisfaction Problems

In the following, we describe the ILP and Knapsack problems because we encode our allocation planning problem as these problems. Furthermore, we give a summary of solving algorithms. The definition is analogous to the original definition of Schrijver [Sch86] and the Bachelor's Thesis of Hüwe [Hüw13].

4.6.1.1 Integer Linear Programming Problem

Within this thesis, we use ILPs that represent a constraint satisfaction problem to describe the allocation planning problem of component instances to distributed hardware resources. Definition 1 defines the ILP problem according to Schrijver [Sch86].

Definition 1 (Integer Linear Programming (ILP) Problem) Given $b, c \in \mathbb{Q}^n$ and $A \in Mat(m, n; \mathbb{Q})$. Determine $x \in \mathbb{Z}^n$, under the constraints $A \cdot x \leq b$ and

$$c^T \cdot x = max\{c^T \cdot x' | A \cdot x' \leq b \text{ and } x' \in \mathbb{Z}^n\}$$ □

$Mat(m, n; \mathbb{Q})$ is the set of matrices, with m rows and n columns with values in $\mathbb{Q}$. $c \in \mathbb{Z}^n$ is the column vector and c^T the corresponding row vector.

4.6.1.2 Knapsack Problem

In more detail, we use the Knapsack problem [Sch86] as a special form of the ILP problem to describe the allocation planning problem. The Knapsack problem restricts x to be true, i.e., 1 or to be false, i.e., 0. The problem is also known as 0-1-ILP or Pseudo-Boolean problem [ARMS02]. The 0-1 Integer Programming problem and the Knapsack problem belong

to one of the 21 NP-complete problems of Karp [Kar72]. The allocation planning problem corresponds to this problem, as a component instance can either be allocated to a resource and the corresponding variable x is 1 or it is not allocated to a resource and the corresponding variable is 0. Definition 2 defines the Knapsack problem according to Schrijver [Sch86].

Definition 2 (Knapsack Problem) Given $b, c \in \mathbb{Q}^n$ and $A \in Mat(m, n; \mathbb{Q})$. Determine $x \in \{0, 1\}^n$, under the constraints $A \cdot x \leq b$ and

$$c^T \cdot x = max\{c^T \cdot x' | A \cdot x' \leq b \text{ and } x' \in \{0, 1\}^n\}$$ □

4.6.1.3 Solving NP-Complete Problem

Due to the Cook–Levin theorem [Sip06] we know that if a polynomial time solvable algorithm exists for any NP-complete problem than for all NP-complete problem a polynomial time solvable algorithm exists because NP-complete problems can be reduced to another NP-complete problem. That means as long as $P \neq NP$ we cannot solve the problem in polynomial time: $ILP \in P \Leftrightarrow P = NP$.

Kumar *et al.* [KLMJ10] give an overview of ILP solvers. A simple and one of the first known branch-and-bound algorithms for solving (mixed) integer linear program was presented by Dakin [Dak65]. The satisfiability of Boolean formula where each clause is limited to at most three literals (3-SAT) is another famous NP-complete problem and belongs to one of the 21 NP-complete problems of Karp [Kar72]. Therefore, also SAT solvers [PBG05] are used to solve Knapsack-based problems, like the allocation of components to ECUs [ZP13].

4.6.2 Linear Program Modeling

"We provide a meta-model for general linear programs so that we can use model-to-model (M2M) transformation techniques. The linear program model decouples the problem description from concrete solver inputs, which result in a better understandability and maintainability of the transformation." [*PH15] "Usually, a LinearProgram consists of unknown variables $x_1, ..., x_n \in \mathbb{R}$, m constraints, and a linear objectiveFunction $\varphi : \mathbb{R}^n \to \mathbb{R}$. A ConstraintExpression $1 \leq i \leq m$ has the form $\sum_{k=1}^{n} a_{i,k} \cdot x_k \theta b_i$ where $a_{i,k}, b_i \in \mathbb{R}$ and $\theta \in \{<, \leq, =, \geq, >\}$." The goal is to find an assignment for the unknowns $x_1, ..., x_n$ such that all constraints hold and $\varphi(x_1, ..., x_n) = goal\{\varphi(x'_1, ..., x'_n) | x'_1, ..., x'_n \in \mathbb{R}$ and all constraints hold for $x'_1, ..., x'_n\}$, where $goal \in \{min, max\}$ [Sch86]. If the x_i are restricted to the LPDataType BINARY $\{0, 1\}$ they are also called binary decision variables and the linear program is called a 0-1-ILP." [*PH15] Figure B.7 in Appendix B.5 shows the corresponding class diagram of the meta-model for linear programs.

The 0-1-ILP model, which results from this M2M transformation, is transformed via a model to text transformation to the LP format [LPS] that corresponds to the concrete textual syntax as defined in Appendix B.6. Other solvers, like SCIP (Solving Constraint Integer Programs) [SCI; MFG+17] or GUROBI [GUR] are also able to process this input. Furthermore, we provide a transformation of our 0-1-ILP model to the solver OPT4J [LGRT11] via a Java program that uses the OPT4J API [OPT4J]. "OPT4J combines Boolean Satisfiability Problem (SAT)-solving with evolutionary optimization in contrast to LPSolve that solves linear programs classically with branch-and-bound [Sch86]. A solution of the ILP, if it exists, corresponds to a feasible allocation." [*PH15]

4.6.3 0-1-ILP Representation of ASL Constraints

In the upper part, Figure 4.17 shows the function $f \in F_{ECU^{COMP}}$ that defines a feasible allocation specification. In the lower part, Figure 4.17 shows the corresponding ILP variables and inequalities. The starting point of the transformation is an allocation specification. The idea of the transformation from the ASL specification to an ILP specification is that each allocation can be mapped to a corresponding equation or inequality, respectively [*PH15]. We prove in [*HP17] for each transformation of a constraint Φ that the equation/inequality is true if and only if the corresponding function f is a feasible allocation specification with respect to Φ ($f \in F_\Phi$). Thereby, we prove: $f \in F_\Phi \Leftrightarrow equation/inequality = true$.

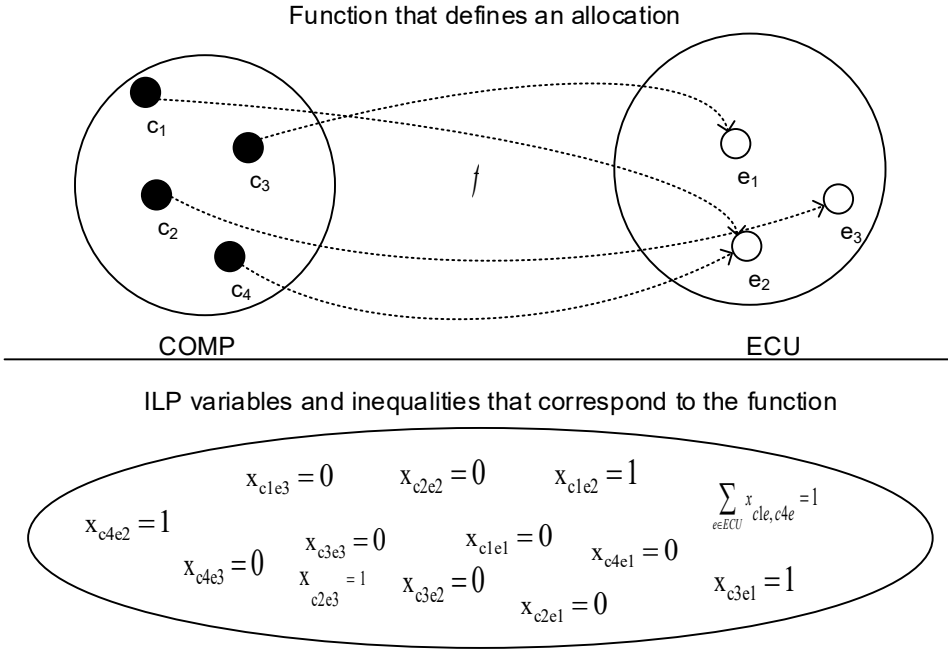


Figure 4.17: ASL Semantics Definition and 0-1-ILP Representation

4.6.3.1 Transformation of General Constraints to a 0-1-ILP

For all component instances $c \in$ COMP and ECUs $e \in$ ECU, we introduce binary decision variables $x_{c,e} \in \{0, 1\}$. These binary decision variables represent an allocation. The binary decision variable $x_{c,e} = 1$ defines that component instance c is allocated to ECU e.

In order to make sure that each component instance is allocated to exactly one ECU, we add the following equation to the ILP:

$$\sum_{e \in ECU} x_{c,e} = 1 \tag{4.10}$$

for all $c \in COMP$. Let $f \subseteq$ COMP $\times$ ECU be a binary relation such that $(c, e) \in f \overset{def.}{\Longleftrightarrow} x_{c,e} = 1$ for $c \in$ COMP and $e \in$ ECU. Since the binary decision variables should represent an allocation, we have to show that the relation f corresponds to a mapping. We formally prove in [*HP17] that f is left-total (for all $c \in$ COMP there exists an $e \in$ ECU such that $(c, e) \in f$) and right-unique(for all $c \in$ COMP and for all $e, e' \in$ ECU such that $\forall (c, e), (c, e') \in f$ it holds that e and e' are the same ECU $e = e'$) [*PH15]. As a result, if these properties of the mapping

f hold and the equation is true it applies that each component instance is always allocated to exactly one ECU.

4.6.3.2 Advanced ILP Variable Dependencies

"In order to transform language elements like the *requiredLocation* constraint [(cf. Section 4.5.1.5)], we have to express more advanced dependencies such as component c_1 is allocated to ECU e_1 and component c_2 is allocated to ECU e_2. For this, we introduce new binary decision variables and inequalities, which are based on concepts that are presented in [MMM12]. More precisely, for $c_1, ..., c_l \in$ COMP, $e_1, ..., e_l \in$ ECU ($l \geq 2$), we introduce a new binary decision variable $x_{c_1,e_1,...,c_l,e_l} \in \{0, 1\}$. The idea is that this variable takes the value 1 if and only if c_1 is allocated to $e_1, ...,$ and c_l is allocated to e_l. Therefore, we have to couple it with the previously introduced binary decision variables using appropriate inequalities." [*PH15]

We add the following inequalities and intermediate binary decision variables

$$\forall i = 1, \ldots, l \colon x_{c_1,e_1,...,c_l,e_l} \leq x_{c_i,e_i} \tag{4.11}$$

$$\sum_{i=1}^{l} x_{c_i,e_i} \leq l - 1 + x_{c_1,e_1,...,c_l,e_l} \tag{4.12}$$

Equation 4.11 implies if the variable $x_{c_1,e_1,...,c_l,e_l} = 1$ than $x_{c_i,e_i} = 1$ for $i = 1, .., l$. Furthermore, we know by the definition $(c, e) \in f \overset{def.}{\iff} x_{c,e} = 1$ of f that a tuple (c_i, e_i) is part of f and thereby that the variable $x_{c_i,e_i} = 1$. Equation 4.12 implies than $1 \leq x_{c_1,e_1,...,c_l,e_l}$ and consequently, $x_{c_1,e_1,...,c_l,e_l} = 1$. This means that the variable $x_{c_1,e_1,...,c_l,e_l}$ takes the value 1 if and only if c_1 is allocated to $e_1, ...,$ and c_l is allocated to e_l. Our technical report [*HP17] shows the proof formally. In the following, we formally define the transformation of the different allocation types to an ILP.

4.6.3.3 Transformation of Collocation Constraints to a 0-1-ILP

Let Φ be a collocation constraint. Φ is transformed to the following ILP equation:

$$\forall (c_1, c_2) \in eval(\Phi_{OCL_E}) \colon \sum_{e \in ECU} x_{c_1,e,c_2,e} = 1 \tag{4.13}$$

"That means that for each tuple (c_1, c_2) from the evaluation result set there exists exactly one ECU e such that the binary decision variable $x_{c_1,e,c_2,e}$ takes the value 1, which implies that c_1 and c_2 are both allocated to e." [*PH15] The proof that the ILP equation preserves the semantics of the collocation constraint type is shown in [*HP17].

4.6.3.4 Transformation of SeparateLocation Constraints to a 0-1-ILP

Let Φ be a separateLocation constraint. Φ is transformed to the following ILP equation:

$$\forall (c_1, c_2) \in eval(\Phi_{OCL_E}) \colon \sum_{e \in ECU} x_{c_1,e,c_2,e} = 0 \tag{4.14}$$

"That means that for each tuple (c_1, c_2) from the evaluation result set there exists no ECU e such that the binary decision variable $x_{c_1,e,c_2,e}$ takes the value 1, which implies that c_1 and c_2 are both allocated different ECUs." [*PH15] The proof that the ILP equation preserves the semantics of the separateLocation constraint type is shown in [*HP17].

4.6.3.5 Transformation of RequiredLocation Constraints to a 0-1-ILP

Let Φ be a requiredLocation constraint. Φ is transformed to the following ILP equation:

$$\forall g \in Q\colon \sum_{(c_1,e_1,\ldots,c_n,e_n)\in g} x_{c_1,e_1,\ldots,c_n,e_n} \geq 1 \tag{4.15}$$

"That means, we require that there exists at least one tuple $(c_1, e_1, ..., c_n, e_n)$ in the equivalence class g such that the binary decision variable $x_{c_1,e_1,...,c_n,e_n}$ takes the value 1, which implies that c_i is allocated to e_i for $i = 1, ..., n$." [*PH15] The proof that the ILP equation preserves the semantics of the *requiredLocation* constraint type is shown in [*HP17].

4.6.3.6 Transformation of RequiredResource Constraints to a 0-1-ILP

Let Φ be a requiredResource constraint. Φ is transformed to the following ILP equation:

$$\forall (bound, weightSet) \in eval(\Phi_{OCL_E})\colon \tag{4.16}$$

$$\sum_{(c_1,e_1,\ldots,c_n,e_n,weight)\in weightSet} weight \cdot x_{c_1,e_1,\ldots,c_n,e_n} \leq bound \tag{4.17}$$

"This transformation directly corresponds to the definition of the *requiredResource* constraint. Basically, we just replaced the h function, which is defined by Equation (4.9), with the corresponding binary decision variable in Inequality (4.17)." [*PH15] The proof that the ILP inequalities preserve the semantics of the *requiredResource* constraint type is shown in [*HP17].

4.6.4 Back-Transformation to the System Allocation Specification Model

"Once we transformed the allocation [constraint] specification to an ILP, we leverage LPSolve [LPS], OPT4J [OPT4J], or SCIP [SCI] to solve the ILP. It returns the first feasible solution by assigning concrete values to the variables. In order to obtain a system allocation, we parse this output. Each variable $x_{c,e}$ with $x_{c,e} = 1$ is transformed into an allocation object, which references the corresponding component instance and ECU. For this transformation, we make use of QVT-o's *invresolve* mechanism [QVT]. The created allocation object is added as a child to a MechatronicUML system allocation specification model. Figure 4.18 shows the MechatronicUML system allocation specification meta-model. A SystemAllocation references a MechatronicUML ComponentInstanceConfiguration and HWPlatformInstanceConfiguration. An Allocation maps a ComponentInstance to a StructuredResourceInstance." [*PH15] A feasible system allocation must contain an allocation for each component instance that is contained in a component instance configuration. Furthermore, a feasible allocation specification must fulfill all constraints. The proof of the correctness of the allocation specification creation is shown in [*HP17].

4.7 Constraint Definition for Cyber-physical Systems

The allocation of component instances to resources is essential for CPSs to fulfill safety requirements. Therefore, this section shows how we can specify eight different constraints from existing literature using our model-driven allocation engineering approach.

The remainder of this section is structured as follows. Section 4.7.1 shows an example for a software dependency constraint. The constraint requires that a set of component instances is

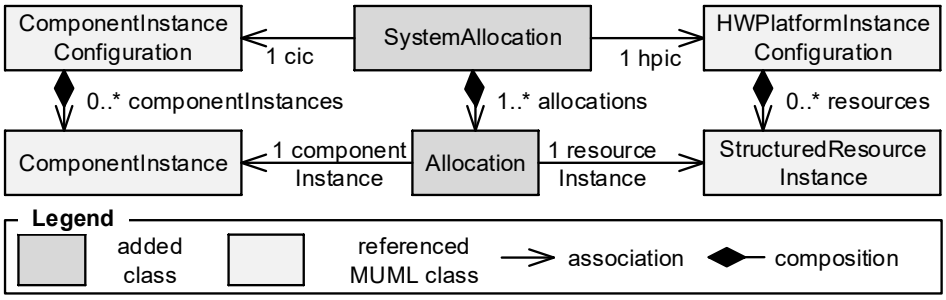


Figure 4.18: System Allocation Meta-Model (Adapted from [*PH15])

allocated (collocated) to the same ECU. Thereafter, Section 4.7.2 shows a hardware topology constraint of component instances. Following, Section 4.7.3 defines a software-incompatibility constraint that is required for security reasons. Furthermore, Section 4.7.4 shows an allocation constraint, which is necessary to enable communication between component instances in the case of a distributed allocation. Sections 4.7.5–4.7.8 define resource consumption constraints of component instances in terms of memory usage (cf. Section 4.7.5), process schedulability for rate monotonic scheduling (cf. Section 4.7.6) and deadline monotonic scheduling (cf Section 4.7.7), and network schedulability for CAN buses (cf. Section 4.7.8).

4.7.1 Software-Dependency Collocation Constraint

We implement a constraint that ensures that the component instance overtakeeHMI and the component instance overtakeeDisplay have to be allocated to the same ECU (cf. Section 4.1) because both instances depend on each other and have a high communication rate. Thus, we specify an OCL expression that evaluates to:
Set{Tuple{firstComponent = overtakeeHMI, secondComponent = overtakeeDisplay}}.

Listing 4.3 shows this constraint of the type **collocation** with the name collocateHMIAndDisplay. The constraint evaluates to a set that consists of a 2-tuple whose concrete type is defined by the descriptors. The elements of the tuple can be accessed via the names firstComponent and secondComponent. We use the OCL operation allocateToSameECU(instance1:String, instance2:String) to define the allocation constraint that the components with the names overtakeeHMI and overtakeeDisplay have to be collocated to. Later, the evaluation result of this operation call is transformed to corresponding ILP constraints. The OCL operation allocateToSameECU(...) is stored in the MechatronicUML-specific allocation operation OCL library (cf. Appendix B.4).

```
1  constraint collocation collocateHMIAndDisplay {
     descriptors (firstComponent:instance::ComponentInstance,
      secondComponent:instance::ComponentInstance);
     ocl   self.allocateToSameECU('overtakeeHMI','overtakeeDisplay'); }
```

Listing 4.3: A Constraint of the Type collocation

4.7.2 Topology-Dependency Required Location Constraint

We implement a constraint that ensures that the component instance overtakeeDriver is allocated to the platform instance Chassis (cf. Section 4.1). Therefore, we specify an

OCL expression that evaluates to: Set{Tuple{component = overtakeeDriver, allowedResource = motor_board1},Tuple{component = overtakeeDriver, allowedResource = motor_board2}}.

The constraint evaluates to a set that consists of several 2-tuples. The elements of a 2-tuple can be accessed via the names component and allowedResource. We use the OCL operation allocateComponentToPlatform(component : String, platform : String) to specify that the component with the name overtakeeDriver has to be allocated to one of the ECUs of the Chassis platform. The OCL operation allocateComponentToPlatform(...) is stored in the MechatronicUML specific allocation operation OCL library (cf. Appendix B.4). Furthermore, engineers can add additional possible target ECUs by either combining the set that is computed by the OCL operation allocateComponentToPlatform(...) with another computed set using the OCL union operation.

```
1 constraint requiredLocation requiredChassisPlatform {
    descriptors (component:instance::ComponentInstance,
      allowedResource:hwresourceinstance::ResourceInstance);
    ocl  self.allocateComponentToPlatform('overtakeeDriver', 'Chassis'); }
```

Listing 4.4: A Constraint of the Type requiredLocation

4.7.3 Software-Incompatibility Separate Location Constraint

For security reasons, we implement a constraint that ensures that the component instance overtakeeCommunicator and the component instance overtakeeDriver have to be allocated to different ECUs (cf. Section 4.1). Thereby, we avoid that if an attacker compromises the component instance overtakeeCommunicator the critical component instance overtakeeDriver that has access to the motor gets affected directly. Thus, we specify an OCL expression that evaluates to: Set{Tuple{firstComponent = overtakeeCommunicator, secondComponent = overtakeeDriver}}.

Listing 4.5 shows this constraint of the type **separateLocation** with the name separateCommunicatorFromDriver. The constraint evaluates to a set that consists of a 2-tuple. The elements of the tuple can be accessed via the names firstComponent and secondComponent. We use the OCL operation allocateToDifferentECUs(instance1 : String, instance2 : String) to specify that the components that are named overtakeeCommunicator and overtakeeDriver cannot be collocated. The allocateToDifferentECUs operation is an alias operation, which we introduce for usability reasons, for the allocateToSameECU operation. It is possible to use the OCL operation allocateToSameECU directly as it returns the required 2-tuple. The OCL operation allocateToDifferentECUs(...) is stored in the MechatronicUML specific allocation operation OCL library (cf. Appendix B.4).

```
1 constraint separateLocation separateCommunicatorFromDriver {
    descriptors (firstComponent:instance::ComponentInstance,
      secondComponent:instance::ComponentInstance);
    ocl  self.allocateToDifferentECUs('overtakeeCommunicator','overtakeeDriver'); }
```

Listing 4.5: A Constraint of the Type separateLocation

4.7.4 Software-Communication Location Constraint

Furthermore, a feasible allocation specification should always guarantee that component instances, which communicate with each other, are allocated to the same ECU or to ECUs that are connected via a communication bus or a direct network link. For example, the

component instance overtakeeHMI communicates with the component instances overtakeeDisplay and overtakeeCommunicator. We obtain, for example, a feasible allocation, with respect to the component communication and ECU interconnection, if we allocate overtakeeHMI and overtakeeDisplay to ECU hmi_board and overtakeeCommunicator to ECU wifi_board. In this case, overtakeeHMI and overtakeeDisplay can communicate via shared memory and overtakeeDisplay and overtakeeCommunicator can communicate via the CAN bus of the platform Robot. It would be an infeasible allocation if overtakeeCommunicator would be allocated to the ECU assis_board2 because it shares no bus with the ECU hmi_board. Therefore, we specify an OCL expression that evaluates to:
Set{Tuple{c1 = overtakeeHMI, e1 = hmi_board, c2 = overtakeeDisplay, e2 = hmi_board},..., Tuple{c1 = overtakeeHMI, e1 = hmi_board, c2 = overtakeeCommunicator, e2 = wifi_board} }.

Listing 4.6 shows the communicatingComponents constraint, which ensures that communicating components are allocated to ECUs that can communicate with each other. The allocateCommunicatingComponentsToConnectedECUs() operation computes the Cartesian product of the set consisting of pairs of communicating components and the set consisting of pairs of connected ECUs. The set that represents the Cartesian product gets very large depending on the amount of connected component instances and connected ECUs. Tuples of the Cartesian product should be rejected if it is already known that they are infeasible due to other constraints, e.g., by building the cut set to other location constraints. Thereby, the size of the set can be reduced, which improves the efficiency of the translation to an ILP. The communicatingComponents constraint evaluates to a set that consists of several 4-tuples. The elements of a 4-tuple can be accessed via the names c1, e1, c2, and e2. The semantics of a tuple from this set is that the component that is referred by the named part c1 can be allocated to the ECU referred by e1, while the component referred by c2 is allocated to the ECU referred by e2. Since a tuple in OCL is unordered, the tuple *descriptors* (c1:instance::ComponentInstance, e1:hwresourceinstance::ResourceInstance), (c2:instance::ComponentInstance, e2:hwresourceinstance::ResourceInstance) are used to group the corresponding pairs of a component instance and an ECU. The semantics of the tuple Tuple{c1 = overtakeeHMI, e1 = hmi_board, c2 = overtakeeCommunicator, e2 = wifi_board} is that the component overtakeeHMI can be allocated to the hmi_board if the component overtakeeCommunicator is allocated to the wifi_board and vice versa. We use the OCL operation allocateCommunicatingComponentsToConnectedECUs() to calculate all tuples of communicating component instances and connected ECUs. In practice it would be useful to filter tuples from the resulting product set if possible because the OCL operation does not scale well. The size of the product set is $|CommunicatingComponentInstances| \cdot |ConnectedECUs|$. The OCL operation allocateCommunicatingComponentsToConnectedECUs(...) is stored in the MechatronicUML specific allocation operation OCL library (cf. Appendix B.4).

```
1  constraint requiredLocation communicatingComponents {
     descriptors
      (c1:instance::ComponentInstance, e1:hwresourceinstance::ResourceInstance),
      (c2:instance::ComponentInstance, e2:hwresourceinstance::ResourceInstance);
5    ocl self.allocateCommunicatingComponentsToConnectedECUs(); }
```

Listing 4.6: A Constraint of the Type requiredLocation

Likewise, software component instances that read sensor data or control actuators need access to these devices. Therefore, the ECU where the software component instance is allocated requires a physical connection to the required device. For example, in our system, only ECUs which are connected to the same bus as a device, can access the device values. As a result,

the component instance overtakeeMotorL has to be allocated to one of the ECUs of the Chassis platform that can write the value of the motor, via the I2C bus. Therefore, we specify an OCL expression that evaluates to:
Set{Tuple{component = overtakeeMotorL, resource = motor_board1},Tuple{component = ...}}. We define for our example a static allocation constraint that guarantees that all continuous component instances have access to the required devices.

Listing 4.7 shows this constraint of the type **requiredLocation** with the name allowedECUsForDeviceAccess . The constraint evaluates to a set that consists of several 2-tuples. The elements of a 2-tuple can be accessed via the names component and resource. We use the OCL operation allocateToECU(component : String, resource : String) to specify that the component instance with the name overtakeeMotorL has to be allocated to the ECU motor_board1 or motor_board2. The resource parameter for the OCL operation allocateToECU(component : String, resource : String) is computed via the OCL helper operation allowedDeviceAllocations(componentName : String) that returns for each component instance that accesses a device the static set of allowed ECUs.

```
constraint requiredLocation allowedECUsForDeviceAccess {
  descriptors (component:instance::ComponentInstance,
    resource:hwresourceinstance::ResourceInstance);
  ocl  self.getAllSWInstances()->collect(ci |
    self.allowedDeviceAllocations(ci.name)->collect(ecu |
    self.allocateToECU(ci, ecu)))->asSet()  ; }
context oclcontext::OCLContext
  def: allowedDeviceAllocations(componentName : String) : Set(String) =
  if componentName    = 'overtakeeColor' then Set{'wifi_board'}
  else if componentName = 'overtakeeDisplay' then Set{'hmi_board'}
  else if componentName = 'overtakeeDistance' then Set{'motor_board1','motor_board2'}
  else if componentName = 'overtakeeMotorL' then Set{'motor_board1','motor_board2'}
  else if componentName = 'overtakeeMotorR' then Set{'motor_board1','motor_board2'}
  else Set{} endif ...
```

Listing 4.7: A Constraint of the Type requiredLocation

4.7.5 Memory Usage Resource Constraint

In this section, we show how required memory constraints can be specified using the ASL. For example, to avoid that more main memory of an ECU is used by a software component instance than an ECU provides. The component instance overtakeeHMI needs 768 bytes, the component instance overtakeeDisplay needs 256 bytes, and the component overtakeeCommunicator needs 1024 bytes according to Table 4.1 on Page 103. Moreover, the ECU hmi_board's amount of available memory is 1024 bytes. Hence, it is possible that overtakeeHMI and overtakeeDisplay are both allocated to hmi_board but, for example, overtakeeHMI and overtakeeCommunicator cannot be both allocated to hmi_board.

Listing 4.8 shows a constraint of the type **requiredResource** with the name memoryConsumption. It is used to guarantee that the component instances, which are allocated to the same ECU, do not exceed the ECU's available memory. The keywords **weight**, **bound**, and **descriptors** define the return type of the OCL query. The constraint type **requiredResource** always requires all three keywords. Thus, we specify an OCL expression that binds for each ECU the available memory to the named part availableMemory and that binds a set to the named part requiredMemory that describes the memory consumption of each component, if it is allocated to that ECU. Listing 4.9 shows the concrete type of the OCL expression. The memory extensions

of the component instances are accessible via the OCL operation getMemoryExtensions(). This operation selects all self.extensions of a component instance that are of the kind RequiredMemory (cf. Figure 4.11 in Section 4.4). The memory value of a RequiredMemory extension is accessible via the attribute requiredMemory and the OCL operation convertToByte() that normalizes the requiredMemory value to the memory unit Byte.

```
constraint requiredResource memoryConsumption {
weight requiredMemory;
bound availableMemory;
descriptors (componentInstance:instance::ComponentInstance,
  resourceInstance:hwresourceinstance::ResourceInstance);
ocl self.maxMemoryConsumption(); }
```

Listing 4.8: A Constraint of the Type requiredResource (Adapted from [*PH15])

```
Set(Tuple(requiredMemory:Set(
    Tuple(componentInstance:instance::ComponentInstance,
    resourceInstance:hwresourceinstance::ResourceInstance,
    requiredMemory:Real)),
    availableMemory:Real))
```

Listing 4.9: Return Type of the OCL Expression Defined in Listing 4.8

An excerpt of the OCL result of the operation maxMemoryConsumption() for the ECU hmi_board and the component instances overtakeeHMI, overtakeeDisplay, and overtakeeCommunicator is: Set{Tuple{requiredMemory = Set{ Tuple{componentInstance = overtakeeHMI, resourceInstance = hmi_board, requiredMemory = 768}, Tuple{componentInstance = overtakeeDisplay, resourceInstance = hmi_board, requiredMemory = 256}, Tuple{componentInstance = overtakeeCommunicator, resourceInstance = hmi_board, requiredMemory = 1024} },availableMemory = 1024}}. In general, for each component instance, which can be potentially allocated to the hmi_board, the set that is referred by the outer tuple's requiredMemory named part should include a corresponding tuple.

The resource constraint (cf. Section 4.5.1.6) evaluates to a set that consists of several 2-tuples (one 2-tuple for each ECU). "Such a 2-tuple has the [outer] named parts availableMemory and requiredMemory. The [outer] named part availableMemory represents an ECU's available memory. The [outer] named part requiredMemory refers to a set that consists of [an inner] 3-tuples. Such a 3-tuple has the [inner] named parts componentInstance, resourceInstance, and requiredMemory. [Note, we use the name requiredMemory as an outer named part and as an inner named part because than users only have to define a single name. OCL allows to define and use hierarchical nested data structures.] The requiredMemory [inner] named part represents the required memory of the component that is referred by the componentInstance [inner] named part if it is allocated to the ECU that is referred by the resourceInstance [inner] named part. Each 2-tuple represents a [resource] constraint whose left-hand side (marked by the keyword **weight**) has to be smaller than or equal to the right-hand side (marked by the keyword *bound*). The left-hand side is the sum over the referred [inner] 3-tuples. An [inner] 3-tuple contributes its requiredMemory value to sum, if its referred componentInstance is allocated to its referred resourceInstance, and 0 otherwise. The right-hand side is given by the 2-tuple's availableMemory [outer] named part." [*PH15]

We use the OCL operation maxMemoryConsumption() to get the required memory, which the component instance needs, and the available Random-Access Memory (RAM), which the ECU provides, of all combinations of component instances and ECUs. The OCL operation

maxMemoryConsumption() is stored in the MechatronicUML specific allocation operation OCL library (cf. Appendix B.4).

Referring to the excerpt of the previous OCL result, we see that only certain combinations of the component instances componentInstance, resourceInstance, and requiredMemory can be allocated to the ECU hmi_board because, e.g., $768 + 256 + 1024 \leq 1024$ does not hold if all instances are allocated to the same ECU. Nevertheless, $768 + 256 + 0 \leq 1024$ holds, if overtakeeHMI and overtakeeDisplay are allocated to hmi_board and overtakeeCommunicator is allocated to a different ECU. Hence, it is possible that overtakeeHMI and overtakeeDisplay are both allocated to hmi_board, but, for example, overtakeeHMI and overtakeeCommunicator cannot be both allocated to hmi_board. Summing up the required memory and ensuring that the calculated allocation specification is feasible is done during the allocation planning by the constraint solver.

4.7.6 Processor Usage Resource Constraint – Response-Time Analysis for Task Scheduling

In this section, we show how sufficient constraints for task schedulability based on the RTA presented by Zeller and Prehofer [ZP13] can be specified using the ASL. Thereby, we avoid that the utilization of a processor gets too high by allocating too many preemptive, static prioritized tasks to the same ECU. As a result, a task scheduling would not be able to meet all deadlines of tasks with real-time requirements during runtime.

The component instance overtakeeHMI has the lowest priority of $prio_{HMI} = 0$, a period of $T_{HMI} = 30ms$, a deadline of $D_{HMI} = 30ms$, and a maximum execution time of $WCET_{HMI} = 10ms$ according to Table 4.1 on Page 103. The component instance overtakeeDisplay has a higher priority of $prio_{Display} = 2$, a period of $T_{Display} = 10ms$, a deadline of $D_{Display} = 10ms$, and a maximum execution time of $WCET_{Display} = 5ms$. According to Listing 4.3, both components must be allocated to the same ECU. Therefore, we ensure the schedulability if both instances are allocated to the same ECU.

Listing 4.10 shows a constraint of the type **requiredResource** with the name rta. It is used to guarantee that the component instances, which are collocated, can be scheduled. The listing implements the sufficient and necessary schedulability constraint for preemptive tasks with fixed priorities. The keywords **weight**, **bound**, and **descriptors** define the return type of the OCL query. The constraint type **requiredResource** always requires all three keywords. Thus, we specify an OCL expression that binds for each component instance its deadline to the variable upperBound and that binds a set to the variable responseTime that describes the response time of a component instance (c1), if it is allocated with another component instance (c2) to the same ECU (e1==e2). Listing 4.11 shows the concrete type of the OCL expression. The real-time scheduling execution extensions of the component instances are accessible via the OCL operation getSchedulingExtensions(). This operation selects all self.extensions of a component instance that are of the kind Scheduling (cf. Figure 4.11 in Section 4.4). The period and deadline values of a Scheduling extension are accessible via the attributes period and deadline and the OCL operation getLiteralVal() that normalizes the period value to the time unit milliseconds. The priority value is accessible via the attribute priority. The real-time WCET execution extensions of the component instances are accessible via the OCL operation getWCETExtensions(). This operation selects all self.extensions of a component instance that are of the kind WCET. The value of a WCET extension is normalized to the time unit milliseconds. The attribute is accessible via wcet.getLiteralVal().

Equation (4.18) defines the RTA task schedulability conditions for all component instances c and for all ECUs e. The constraint inequality was defined by [ZP13]. The left-hand side of the inequality represents the upper bound for the response time of component instance c on ECU e if it is allocated there $x_{c,e} = 1$ and if the component c' is also allocated on

```
constraint requiredResource rta {
weight responseTime;
bound upperBound;
descriptors
 (c1:instance::ComponentInstance, e1:hwresourceinstance::ResourceInstance),
 (c2:instance::ComponentInstance, e2:hwresourceinstance::ResourceInstance);
ocl   self.RTA(); }
```

Listing 4.10: A RTA Task Scheduling Constraint of the Type requiredResource

```
Set(Tuple(responseTime:Set(
  Tuple(c1:instance::ComponentInstance,e1:hwresourceinstance::
      ResourceInstance,
  Tuple(c2:instance::ComponentInstance,e2:hwresourceinstance::
      ResourceInstance,
  responseTime:Real)),
  upperBound:Real))
```

Listing 4.11: Return Type of the OCL Expression Defined in Listing 4.10

ECU e ($x_{c',e} = 1$). The right-hand side of the inequality represents the deadline D_c of the component instance c. $WCET_{c,e}$ is the Worst-Case Execution Time of the task that represents the software component instance c on the ECU e. T_c is the period, $prio_c$ is the priority, and D_c is the deadline of the corresponding component instance c. The variable $x_{c',e,c,e}$ is a helper variable that takes the value 1 if the instances c' and c are both allocated to the ECU e.

$$\forall c \in \text{COMP}, \forall e \in \text{ECU}:$$
$$WCET_{c,e}x_{c,e} + \sum_{\substack{c' \in \text{COMP},\\ prio_{c'} > prio_c}} (WCET_{c',e}(1 - \frac{WCET_{c',e}}{T_{c'}}) + \frac{WCET_{c',e}}{T_{c'}}D_c)x_{c',e,c,e} \leq D_c \quad (4.18)$$

Thus, we specify an OCL expression that returns for each pair of the Cartesian product of COMPxECU its response time according to Equation (4.18). The OCL result for the ECU hmi_board and the component instances overtakeeHMI and overtakeeDisplay is:

Set{Tuple{responseTime = Set{Tuple{c1 = overtakeeHMI, c2 = overtakeeDisplay, e1 = hmi_board, e2 = hmi_board, responseTime = 17.5}, Tuple{c1 = overtakeeHMI, c2 = overtakeeHMI, e1 = hmi_board, e2 = hmi_board, responseTime = 10}}, upperBound = 30}} In general, for each component instance which can be potentially allocated to the hmi_board, the set that is referred by the outer tuple's responseTime named part should include a corresponding tuple.

The constraint evaluates to a set that consists of several 2-tuples (one 2-tuple for each pair of the Cartesian component instance product and each ECU). Such a 2-tuple has the named parts upperBound and responseTime. The named part upperBound represents the deadline of a task that a feasible scheduling must always hold. The named part responseTime refers to a set that consists of 5-tuples. Such a 5-tuple has the named parts c1, e1, c2, e2, and responseTime. The inner responseTime represents the response time of the component instance that is referred by the c1 named part if it is allocated to the ECU that is referred by the e1 named part and if the component instance that is referred by the name c2 is allocated to the ECU by the e2 named part. Furthermore, the priority of c1 must be lower than or equal to the priority of c2. Each 2-tuple represents a constraint whose left-hand side (weight) has to be smaller than or equal to the right-hand side (bound). The left-hand side is the sum over the referred 5-tuples.

A 5-tuple contributes its responseTime value to the sum, if (1) its referred c1 is allocated to its referred e1, and (2) if its referred c2 is allocated to its referred e2, and (3) if its referred $e1 == e2$. Otherwise, it contributes the value 0. The right-hand side is given by the 2-tuple's upperBound and is according to Equation (4.18) the deadline of the task that is referred by c1.

We use the OCL operation RTA() to get the response time that the component instance needs and its deadline of all combinations of component instances and ECUs. The OCL operation RTA() is stored in the MechatronicUML specific allocation operation OCL library (cf. Appendix B.4).

Referring to the excerpt of the previous OCL result, we see that the RTA scheduling constraint holds with the given task properties (cf. Table 4.1) because:

$$\begin{aligned} WCET_{HMI} = 10ms, WCET_{Display} = 5ms, T_{HMI} = 30ms, \\ T_{Display} = 10ms, D_{HMI} = 30ms, D_{Display} = 10ms, \\ prio_{HMI} < prio_{Display} : \\ 10ms + 5ms(1 - \frac{5ms}{10ms}) + \frac{5ms}{10ms}30ms /^* \text{ referring to Equation (4.18)} ^*/ \\ = 10ms + 17.5ms = 27.5ms \leq 30ms. \end{aligned} \tag{4.19}$$

The utilization of the *hmi_board* is:

$$U = \sum_{i=1}^{n} \frac{WCET_{i,j}}{T_i} = \frac{WCET_{HMI}}{T_{HMI}} + \frac{WCET_{Display}}{T_{Display}} = \frac{10}{30} + \frac{10}{5} = 0.8\overline{3} \tag{4.20}$$

if overtakeeHMI and overtakeeDisplay are allocated to the same ECU. The component instances overtakeeHMI and overtakeeDisplay would not fulfill the RTA scheduling constraint if we change the priority of the component instance overtakeeHMI to $prio_{HMI} = 8$. As a result, the response time of the component instance overtakeeDisplay, would be greater than its deadline if both instances are allocated to the same ECU:

$$5ms + 10ms(1 - \frac{10ms}{30ms} + \frac{10ms}{30ms}10ms) = 5ms + 10ms = 15ms \nleq 10ms. \tag{4.21}$$

The constraint solver totalizes the response times to the final sum and ensures that all allocation constraints of a calculated allocation specification are fulfilled. The utilization of the ECU does not change by changing the task priorities.

4.7.7 Processor Usage Resource Constraint – EDF Schedulability Task Analysis

In this section, we show how a sufficient constraint for task schedulability based on the EDF analysis [ZP13; LL73] can be specified using the ASL. For example, this constraint avoids that the utilization of a processor gets too high due to the allocation of too many preemptive, dynamically prioritized tasks to the same ECU. As a result, a task schedule may not be able to meet all deadlines of tasks with real-time requirements during runtime. In the last section, we show an example where choosing the right priority order is critical for the schedulability. Therefore, a dynamically prioritized scheduling mechanism avoids possible failures.

An EDF scheduler chooses the task priorities dynamically based on their deadlines. The task with the earliest deadline gets the highest priority. The component instance overtakeeDisplay has a higher priority than the component instance overtakeeHMI because it has an earlier deadline ($D_{Display} = 10ms < D_{HMI} = 30ms$) according to Table 4.1.

Listing 4.12 shows a constraint of the type **requiredResource** with the name edf. It is used to guarantee that the component instances, which are collocated, can be scheduled. The

listing implements a sufficient condition for schedulability for preemptive tasks under the assumption that the task *deadline* $\leq$ *period* and all tasks are activated periodically. The keywords **weight**, **bound**, and **descriptors** define the return type of the OCL query. The constraint type **requiredResource** always requires all three keywords. Thus, we specify an OCL expression that binds for each ECU the maximum processor utilization to the variable upperBound and that binds a set to the variable computationDemand. The set consists of tuples where each tuple represents the processor utilization if the componentInstance if it is allocated to the ECU. Listing 4.13 shows the concrete type of the OCL expression. We use the OCL operation EDF() to get for all ECUs the processor utilization of the allocation candidates. The OCL operation EDF() is stored in the MechatronicUML specific allocation operation OCL library (cf. Appendix B.4) and corresponds to the Equation 4.22 that we describe in the following paragraphs. The real-time scheduling execution and wcet extensions of the component instances are accessible the same way as for the RTA constraint (cf. Section 4.7.6).

```
1 constraint requiredResource edf {
    weight computationDemand;
    bound upperBound;
    descriptors (componentInstance:instance::ComponentInstance,
5    resourceInstance:hwresourceinstance::ResourceInstance);
    ocl   self.EDF(); }
```

Listing 4.12: An EDF Task Scheduling Constraint of the Type requiredResource

```
1 Set(Tuple(computationDemand:Set(
    Tuple(componentInstance:instance::ComponentInstance,
    resourceInstance:hwresourceinstance::ResourceInstance,
    computationDemand:Real)),
5    upperBound:Real))
```

Listing 4.13: Return Type of the OCL Expression Defined in Listing 4.12

Inequality (4.22) defines the EDF task schedulability constraint. The schedulability inequality was defined and proved by [LL73] and extended by the variable $x_{c,e}$ that represents the allocation. The demand of computation time of the component instance c is only considered when the instance is allocated to the ECU e. Therefore, each summand is multiplied by $x_{c,e}$. Thereby, the utilization is only added if the component instance is allocated to the corresponding ECU. The sum on the left-hand side represents the utilization of the processor of the ECU e. The right-hand side represents the upper bound of the ECUs processor utilization, which is 100% and represented by 1. $WCET_{c,e}$ is the worst-case execution time of the task that represents the software component instance c on the ECU e. T_c is the period of the corresponding task c.

$$\forall e \in \text{ECU} : \sum_{c \in \text{COMP}} \frac{WCET_{c,e}}{T_c} x_{c,e} \leq 1 \tag{4.22}$$

Thus, we specify the OCL operation EDF() that returns for each ECU its utilization according to Equation (4.22). The OCL result for the ECU hmi_board and the component instances overtakeeHMI and overtakeeDisplay is:

Set{Tuple{computationDemand
= Set{Tuple{componentInstance = overtakeeHMI, computationDemand = 0.333, resourceInstance =

hmi_board}, Tuple{componentInstance = overtakeeDisplay, computationDemand = 0.5, resourceInstance = hmi_board}}, upperBound=1}}

In general, for each component instance, which can be potentially allocated to the hmi_board, the set that is referred by the outer tuple's computationDemand named part should include a corresponding tuple.

The constraint evaluates to a set that consists of several 2-tuples (one 2-tuple for each ECU). Such a 2-tuple has the outer named parts computationDemand and upperBound. The named part upperBound represents the utilization of an ECU that a feasible scheduling must always hold. The outer named part computationDemand refers to a set that consists of 3-tuples. Such a 3-tuple has the inner named parts componentInstance, resourceInstance, and computationDemand. The inner computationDemand represents the computation time demand of the component instance that is referred by the componentInstance of the processor that belongs to the resource instance that is referred by the resourceInstance. Each 2-tuple represents a constraint whose left-hand side (weight) has to be smaller than or equal to the right-hand side (bound). The left-hand side is the sum over the referred 3-tuples. A 3-tuple contributes its computationDemand value to the sum if its referred componentInstance is allocated to its referred resourceInstance. Otherwise, it contributes the value 0. The right-hand side is given by the 2-tuple's upperBound named part and is according to Zeller and Prehofer [ZP13] equal to 1.

We see that the EDF scheduling constraint holds with the given task properties (cf. Table 4.1) because $0,\overline{3} + 0.5 = 0.8\overline{3} \leq 1$ if overtakeeHMI and overtakeeDisplay are allocated to the same ECU. Therefore, we refer to the excerpt of the previous OCL result and the utilization result that Equation (4.20) shows on Page 126. Summing up the response times and ensuring that the calculated allocation specification is feasible is done during the allocation planning by the constraint solver.

4.7.8 Network Usage Resource Constraint – CAN Message Scheduling Response-Time Analysis

In the following, we show how sufficient constraints for message schedulability based on the CAN RTA [DBBL07] can be specified using the ASL. For example, to avoid that the utilization of a CAN message bus gets too high by allocating too many tasks to ECUs that are connected to the message bus and communicate with each other. As a result, a safety-critical message may not meet its deadline during runtime.

According to Table 4.2, the message distance has the lowest priority of $prio_{distance} = 3$, a period of $T_{distance} = 3.5ms$, a deadline of $D_{distance} = 3.25ms$, and a maximum transmission time of $C_{distance} = 1ms$. The message velR has a higher priority of $prio_{velR} = 2$, a period of $T_{velR} = 3.5ms$, a deadline of $D_{velR} = 3.25ms$, and a maximum transmission time of $C_{velR} = 1ms$. Furthermore, the message velL has the highest priority of $prio_{velL} = 1$, a period of $T_{velL} = 2.5ms$, a deadline of $D_{velL} = 2.5ms$, and a maximum transmission time of $C_{velL} = 1ms$.

According to the platform schema that Figure 4.2 shows, the distance sensor and both motors are connected to the same CAN bus. Therefore, we ensure that the messages do not delay each other so that they do not meet their deadlines. Furthermore, the utilization of the CAN bus must be smaller than 100%.

Listing 4.14 shows a constraint of the type **requiredResource** with the name canRTA. It is used to guarantee that all messages that are transmitted via the CAN_C can be transmitted without breaking their deadlines and guarantees that the utilization of the CAN bus is not exceeded. The listing implements a sufficient and necessary schedulability constraint for non-preemptive message transmission with fixed priorities on a CAN bus. The keywords **weight**, **bound**, and **descriptors** define the return type of the OCL query. The constraint type

requiredResource always requires all three keywords. Thus, we specify an OCL expression that binds for each message the message deadline to the variable messageDeadline and that binds a set to the variable messageResponseTime that describes the message response time of each message within a concrete set of messages that share a bus. Therefore, we determine the sets of messages that can share the same bus by examining the Cartesian product of component instances and ECUs. Listing 4.15 shows the concrete type of the OCL expression. The real-time message extensions of the component instances are accessible via the OCL operation getMessageExtensions(). This operation selects all self.extensions of a component instance that are of the kind CANMessageFrame (cf. Figure 4.11 in Section 4.4). The size value of a CANMessageFrame extension is accessible via the attribute size and the OCL operation convertToByte() that normalizes the size value to the unit Byte. The deadline value in milliseconds of a CANMessageFrame extension is accessible via the attribute deadline and the OCL operation getLiteralVal(). The priority value is accessible via the attribute priority. Furthermore, the OCL helper operations getUtilization, getTransmissionTime, getBlockingTime, getBitTransmissionTime, calculateBusyPeriod, are provided to calculate the corresponding values for a concrete BusInstance using the extension values and the formulas defined by Davis *et al.* [DBBL07].

```
1  constraint requiredResource canRTA {
     weight messageResponseTime;
     bound messageDeadline;
     descriptors
5     (c1:instance::ComponentInstance, e1:hwresourceinstance::ResourceInstance),
      (c2:instance::ComponentInstance, e2:hwresourceinstance::ResourceInstance);
     ocl   self.canRTA('CAN_C');}
```

Listing 4.14: A CAN RTA Message Scheduling Constraint of the Type requiredResource

```
1  Set(Tuple(messageResponseTime:Set(
     Tuple(c1:instance::ComponentInstance,e1:hwresourceinstance::ResourceInstance),
     Tuple(c2:instance::ComponentInstance,e2:hwresourceinstance::ResourceInstance)
     messageResponseTime:Real)),
5    messageDeadline:Real))
```

Listing 4.15: Return Type of the OCL Expression Defined in Listing 4.14

We assume that all component instances that send the corresponding messages can send them via the same CAN bus. The used CAN response-time analysis of Davis *et al.* [DBBL07] requires that we provide a worst-case assumption which messages share the same bus at the maximum. In the end, the ILP solver decides on which ECU component instances are allocated and if they use the bus to communicate with each other.

We calculate the worst-case messageResponseTime in OCL for a concrete possible allocation. All OCL operations for determining the message schedulability constraints are implementations of the inequalities of Davis *et al.* [DBBL07].

We specify an OCL expression to determine the worst-case response time for each message within a concrete worst-case set of messages that share a bus. Therefore, we determine the sets of messages that can share the same bus by examining the Cartesian product of component instances and ECUs and calculating the response times according to Davis *et al.* [DBBL07]. The worst-case set of messages is an over-approximation. The actual set of messages is determined by the allocation planning.

The OCL result for the messages velL, velR, and distance that share the CAN bus CAN_C is: Set{Tuple{messageResponseTime = Set{Tuple{c1 = overtakeeDriver, c2 = overtakeeDistance, e1 = motor_board1, e2 = motor_board2, messageDeadline = 2.5}, Tuple{messageResponseTime = Set{Tuple{c1 = overtakeeDriver, c2 = overtakeeDistance, e1 = motor_board1, e2 = motor_board2, messageDeadline = 3.25}, Tuple{messageResponseTime = Set{Tuple{c1 = overtakeeDriver, c2 = overtakeeDistance, e1 = motor_board1, e2 = motor_board2, messageDeadline = 3.25}} In general, for each component instance, which can be potentially allocated to the motor_board1 or motor_board2 and sends messages, the set that is referred by the outer tuple's messageResponseTime named part should include a corresponding tuple.

The constraint evaluates to a set that consists of several 2-tuples (one 2-tuple for each message). Such a 2-tuple has the named parts messageDeadline and messageResponseTime. The named part messageDeadline represents the deadline of a message that a feasible scheduling must always hold. The named part messageResponseTime refers to a set that consists of 5-tuples. Such a 5-tuple has the named parts c1, e1, c2, e2, and messageResponseTime. The inner messageResponseTime named part represents the worst-case response time of a message that is sent by the component instance that is referred by the c1 named part if it is allocated to the ECU that is referred by the e1 named part. Furthermore, to cause bus load, the receiver component instance that is referred by the name c2 of the message is allocated to the ECU by the e2 named part. Furthermore, the bus utilization that the component instances c1 and c2 produce must be lower than 100%. Each 2-tuple represents a constraint whose left-hand side (weight) has to be smaller than or equal to the right-hand side (bound). The left-hand side is the sum over the referred 5-tuples. A 5-tuple contributes its messageResponseTime value to the sum, if (1) its referred c1 is allocated to its referred e1, and (2) if its referred c2 is allocated to its referred e2. Otherwise, it contributes the value 0. The right-hand side is given by the 2-tuple's messageDeadline named part and is according to Davis *et al.* [DBBL07] the deadline of a message that is sent by c1.

We use the OCL operation canRTA() to get the deadline of each message and to get the response time of each message for all combinations of component instances and ECUs. The operation considers the worst-case set of messages. The OCL operation canRTA() is stored in the MechatronicUML specific allocation operation OCL library, which we introduce in Section 4.5.2.

We refer to [DBBL07, Section 3.3] for an example of the calculation of the corresponding worst-case response time and bus utilization. The bus utilization U_{BusC} is about 97% if the three messages velL, velR, and distance should be transmitted via the CAN bus CAN_C. The worst-case response time $R_{distance}$ of the distance message is 3.5, which means that it does not hold its deadline and we do not find a valid schedule under the given constraints. A solution to overcome this issue could be to change the protocol to the newer CAN FD protocol [Har13], which allows higher data rates and thereby lowers the transmission time. Nevertheless, it must be proven if the shown CAN scheduling analysis also holds for the CAN FD protocol. Alternatively, the protocol could be switched to Flexray [ISO17458], which also allows higher data rates. Lukasiewycz presents linear scheduling constraints for Flexray [Luk10].

4.8 System Allocation Specification View

In the system allocation specification view, allocation engineers can view the result of the automatic allocation planning, adapt a calculated system allocation specification, or specify a system allocation manually. We use a visual view that is similar to UML deployment diagrams [UML] and a more compact textual view based on a cross-table consisting of component instances as rows and resource instances as columns.

Figure 4.19 shows an example of the visual allocation specification view. It shows the result of the automatic allocation planning of the example that Section 4.1 describes. We omit the transmission schedulability analysis (cf. Section 4.5.1.6) as this constraint cannot be fulfilled in the example without changing the hardware platform model due to the technical limitation 2 that Section 4.11 describes in more detail.

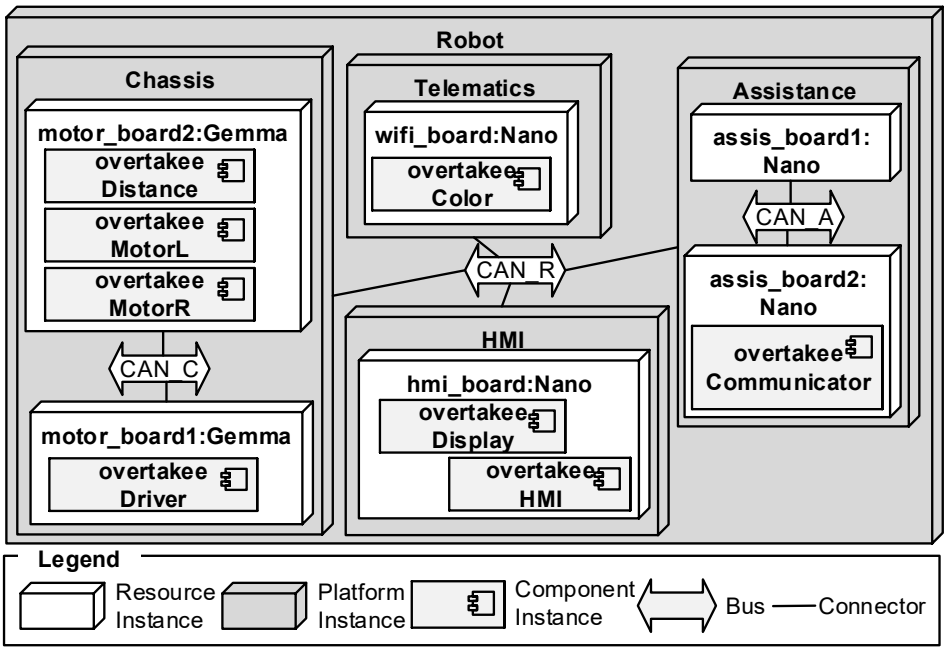


Figure 4.19: Allocation Specification – Graphical View

The automatic planning allocates the component instance overtakeeColor to the resource instance wifi_board of the platform instance Telematics. It allocates the component instances overtakeeDisplay and overtakeeHMI to the resource instance hmi_board of the platform instance HMI. Furthermore, it allocates the component instance overtakeDriver to the resource instance motor_board1 and the component instances overtakeeDistance, overtakeeMotorL, and overtakeeMotorR to the resource instance motor_board2 of the platform instance Chassis. Finally, it allocates the component instance overtakeeCommunicator to the resource instance assis_board2 of the platform instance Assistance. No component instance is allocated to the resource instance assis_board1 so that it could be omitted for this component instance configuration.

Table 4.3 shows the same allocation as displayed in Figure 4.19 as a cross-table. The rows show each component instance and the columns each resource instance. A component instance is allocated to a resource instance if the value of the cell of the intersection of a row and a column is equal to 1. A component instance is not allocated to a resource instance if the value is 0. Furthermore, in the last three rows the table shows the available memory of each resource instance, the sum of the consumed memory on a certain resource instance for the given allocation specification, and the overall processor utilization.

4.9 Tooling Implementation

We implement the hardware platform modeling, allocation constraint modeling, and the automatic allocation planning prototypically. We integrate the modeling languages and transformations within the Eclipse-based MechatronicUML Tool Suite [*DGB+14].

Table 4.3: Allocation Specification – Table View

ECU→ SC↓	**motor_ board1**	**motor_ board2**	**wifi_ board**	**hmi_ board**	**assis_ board1**	**assis_ board2**
overtakee Color	0	0	1	0	0	0
overtakee Display	0	0	0	1	0	0
overtakee Distance	0	1	0	0	0	0
overtakee MotorL	0	1	0	0	0	0
overtakee MotorR	0	1	0	0	0	0
overtakee Communicator	0	0	0	0	0	1
overtakee Driver	1	0	0	0	0	0
overtakeeHMI	0	0	0	1	0	0
available memory	512 B	512 B	1024 B	1024 B	1024 B	1024 B
consumed memory	512 B	384 B	64 B	1024 B	0 B	1024 B
processor utilization	50%	50%	10%	83%	0	50%

For quality assurance, we implement the plugins `allocation.algorithm.ilp.test` and `allocation.algorithm.opt4j.test`. This plugins provide unit tests that execute the ASL to ILP and the ILP to OPT4J transformation for specific *test cases*. A test case is "a pair of input and output in the case of deterministic transformative systems." [UPL12] The input for the ILP test cases consists of a known MechatronicUML model where component instances should be allocated to ECUs considering ASL constraint specifications. The output for the ILP test cases are expected ILP models. After executing the ASL to ILP transformation in the context of the test cases, we compare the generated output with the expected output. We provide test cases that cover general constraints, collocation constraints, seperateLocation constraints, requiredLocation constraints, and requiredResource constraints. Thereby, we cover the concepts that this chapter describes. Furthermore, the OPT4J test cases validate that the generated OPT4J input, which is created from the ILP models, are equal to the expected OPT4J input. Furthermore, weekly meetings are used to discuss and review the current implementation status and the next tasks. We use a Jenkins continuous integration server [Ber12] to build all plugins automatically and to run the unit tests automatically if the implementation of any ASL plugin changes.

Figure 4.20 shows the main plugins and their dependencies. The plugins `hardware.resourceinstance.diagram`, `hardware.resource.diagram`, `hardware.platform.diagram`, and `hardware.platforminstance.diagram` implement four visual diagram editors for the hardware platform modeling (cf. Section 4.3). We implement the diagram editors using the Graphical Modeling Framework (GMF) [Gro09]. The editors use the meta-model plugin `hardware`. This plugin implements the meta-model of the HPDL (cf. Figures A.1-A.5). The `hardware` plugin extends the `core` plugin as it inherits from the MechatronicUML base classes `NamedElement` and `CommentableElement`. We develop the meta-models by using the Eclipse Modeling Framework (EMF) [SBMP08] and encode the static semantics completely by using OCL [OCL].

Furthermore, the plugin `allocation.language.xtext` implements the textual editor for modeling constraints using the ASL (cf. Section 4.5) and provides our OCL operation library for allocation engineering using MechatronicUML. We implement the textual editor using Xtext [EB10]. The plugin `allocation.language.xtext` uses the meta-model plugin `allocation.language`. This plugin implements the meta-model of the ASL (cf. Figure B.6) and the OCL context meta-model (cf. Figure 4.14) that defines the scope of the ASL. The meta-model of the ASL extends the `ocl.xtext` plugin, which defines the OCL meta-model and

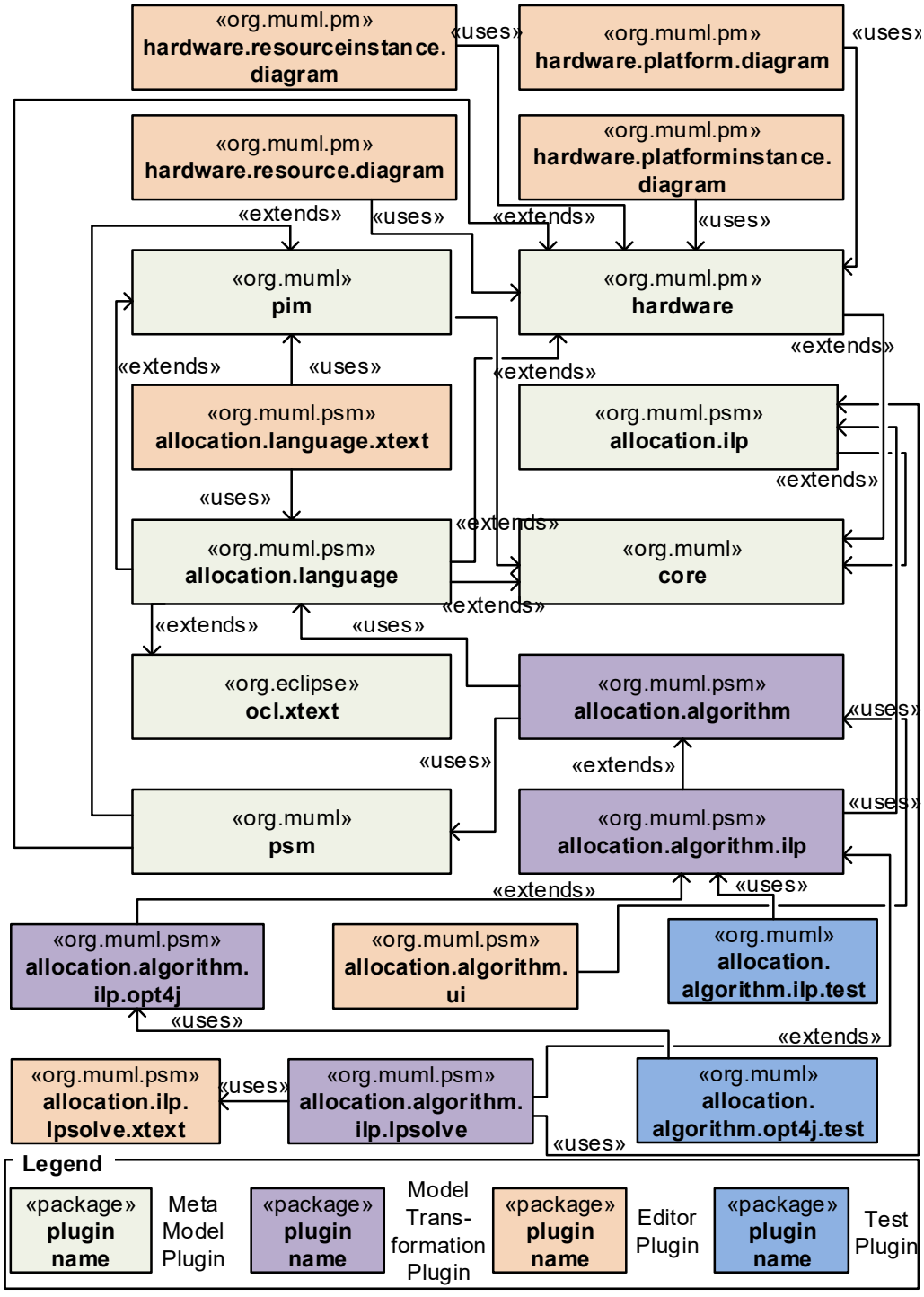


Figure 4.20: Plugins Implementing MechatronicUML Model-Driven Allocation Engineering Approach

textual OCL grammar [Wil11]. The OCL context meta-model extends the MechatronicUML meta-model plugins `core`, `pim`, and `hardware`. The context meta-model defines the scope of the ASL.

Additionally, the editor plugin `allocation.algorithm.ui` implements the integration into the user interface. It enables to start the allocation planning on a `.muml` file from the context menu. The user interface plugin uses the plugin `allocation.algorithm` to perform the automatic allocation planning. The plugin uses the meta-model plugins `psm`. The meta-model plugin `psm` implements the meta-model for storing a system allocation specification (cf. Figure 4.18) for MechatronicUML. The plugin `allocation.algorithm` implements an extension point, where transformation plugins for an automatic allocation planning can register. The plugin `allocation.algorithm.ilp` registers at this extension point. It provides a transformation from the ASL to an ILP. We implement the transformation as M2M-transformation using QVTo [QVT]. It uses the meta-model plugin `allocation.ilp` that implements the meta-model for linear programs (cf. Figure B.7). The plugin `allocation.algorithm.ilp.lpsolve` implements the Model-to-Text (M2T) transformation from a linear program model to the solver LpSolve [LPS] and SCIP [SCI]. Therefore, it uses the plugin `allocation.ilp.lpsolve.xtext` that defines a textual grammar for LpSolve using Xtext [EB10]. The plugin `allocation.algorithm.ilp.op4j` implements the transformation from a linear program model to the alternative solver OPT4J [OPT4J].

4.10 Case Study

This section describes the evaluation of our model-driven allocation engineering approach. We perform the evaluation as a case study on the basis of the guidelines defined by Kitchenham, Pickard, and Pfleeger [KPP95] and Runeson and Höst [RH08]. We perform our evaluation for the following two realistic examples. (1) The autonomous cooperating overtaking example (cf. Figure 1.2) as introduced in Section 1.3 and described in more detail in Section 4.1. (2) The Brake-by-wire case study from Aleti [Ale15]. Brake-by-wire technology replaces traditional mechanical, hydraulic, or pneumatic coupling between a brake pedal, which triggers a braking maneuver, and a brake that inhibits a motion with electronic sensors and actuators that are controlled via software. We published a first version of the Brake-by-wire case study in the context of our allocation engineering approach in [*PH15]. In the following case study, we add in comparison to the first version a task scheduling analysis and add a scalability analysis.

4.10.1 Context and Cases

The aim of our case study is to evaluate if the model-driven allocation engineering approach enables to plan allocation specification automatically for realistic examples and to compare the modeling effort in comparison with the ILP-based approach. Furthermore, we evaluate if the actual implementation fulfills the formal specification. Although we provide a proof that the formal ILP encoding preserves the semantics of the allocation constraint types, we evaluate that our implementation of the ILP encoding is correct and preserves the semantics. Therefore, we evaluate the following evaluation questions:

- Q1 Is our modeling approach for hardware platforms (cf. Section 4.3) and software component's resource requirements sufficient to compute the required information for planning allocations automatically?
- Q2 How much effort do allocation engineers need to specify allocation constraints in a formal way using our modeling approach in comparison to the manual ILP encoding?

Q3 Is the resulting allocation specification of the automatic allocation planning feasible and fulfills all stated allocation constraints?

Q4 How does the number of component instances and ECUs affect the scalability of the creation of the ILP model and the solving of the corresponding ILP model.

We perform the case study on the overtakee of the cooperative overtaking example that represents a cooperative CPS from the automotive domain and on the Brake-by-wire system that represents a *classical* CPS. The following paragraphs describe these systems.

Cooperating Overtaking–Overtakee: We refer to Section 4.1 for a detailed description of the overtakee of the cooperative overtaking system. We consider the Constraints **A.1**-**A.7**. Especially, we use the EDF scheduling strategy for the schedulability Constraint **A.7**.

Brake-by-wire System: Figure 4.21 shows the component instance configuration of the Brake-by-wire system. The system consists of the three structured component instances (Caliper, BrakePedal, Wheel), the discrete atomic component instance sc6/CentralBrakeControl, and the continuous component instance sc9/ParkingBrakeSensor. The structured component instance Caliper consists of the atomic discrete component instance sc4/CaliperControl and the atomic continuous component instances sc1/Temperature, sc2/CaliperPosition, sc3/ParkingBrake, and sc5/CaliperClamp. The structured component part BrakePedal consists of the atomic discrete component instance sc10/BrakePedalControl and the atomic continuous component instances sc11/PedalPosition, sc12/BrakeForce, and sc13/BrakeFeedback. [*PH15]

Table 4.4 shows the task properties of the Brake-by-wire system. We use one task per software component instance. Furthermore, we define that structured component instances require no memory or processing time because they are only logical constructs. We adopt the WCET from [Ale15]. We define that each sensor and actuator should be controlled with a frequency of 20 Hz. As a result, we require a period and deadline of 50 ms. In the worst-case, we get an end-to-end deadline from component instance sc12/BrakeForce:Sensor, which triggers a braking maneuver, to sc5/CaliperClamp:Actuator, which executes a braking maneuver, of 250 ms. A car that travels with 200 km/h drives a distance of additional 13.89 m during this delay.

Table 4.4: Task Properties of the Brake-by-Wire System

Task	Required Memory	Period	Relative Deadline	WCET
caliper	0 B	-	-	-
sc1	64 B	50 ms	50 ms	12 ms
sc2	128 B	50 ms	50 ms	10 ms
sc3	64 B	50 ms	50 ms	2 ms
sc4	512 B	50 ms	50 ms	20 ms
sc5	256 B	50 ms	50 ms	5 ms
sc6	1024 B	50 ms	50 ms	10 ms
wheel	0 B	-	-	-
sc7	64 B	50 ms	50 ms	12 ms
sc8	128 B	50 ms	50 ms	8 ms
sc9	128 B	50 ms	50 ms	4 ms
brakePedal	0 B	-	-	-
sc10	512 B	50 ms	50 ms	6 ms
sc11	64 B	50 ms	50 ms	10 ms
sc12	128 B	50 ms	50 ms	10 ms
sc13	64 B	50 ms	50 ms	4 ms

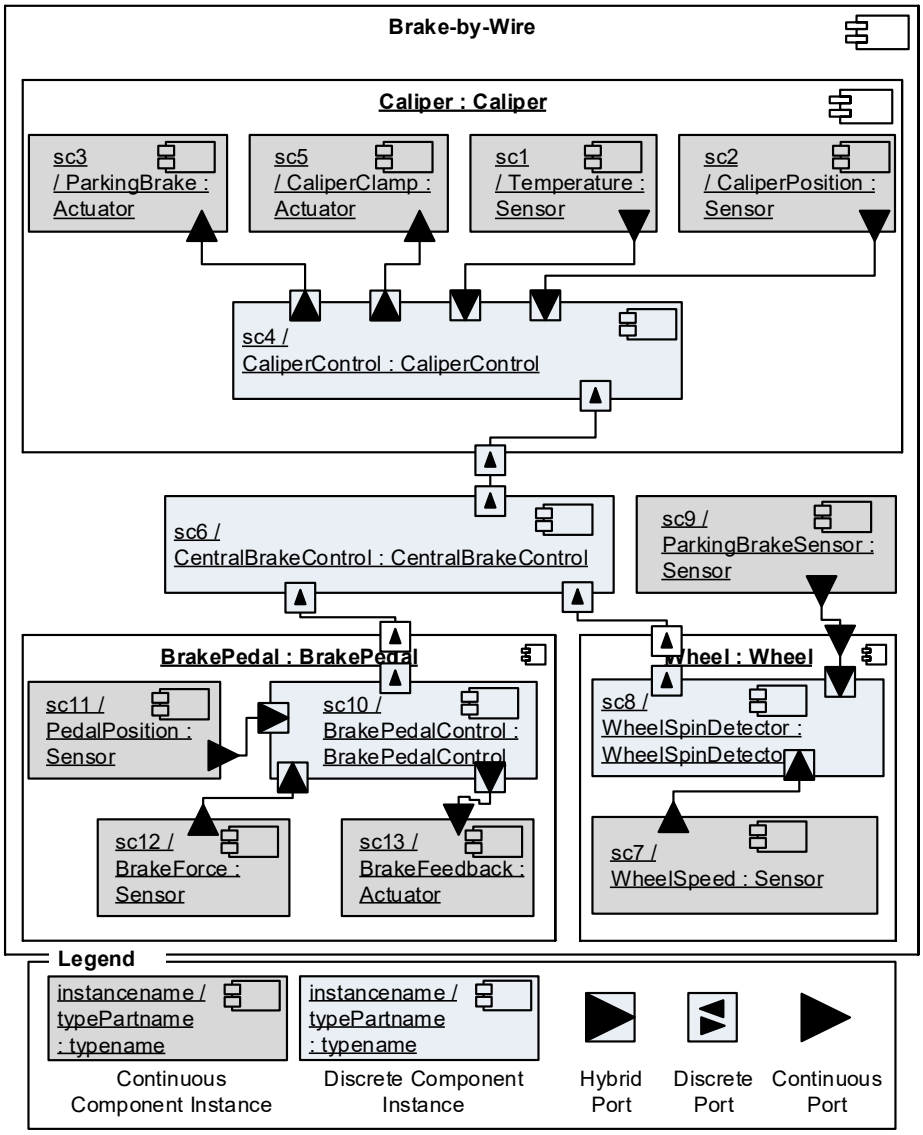


Figure 4.21: Brake-By-Wire Component Instance Configuration (Adapted from [*PH15])

Figure 4.22 shows the hardware platform instance configuration for the Brake-by-wire system. It consists of the platform instance parts Assistance, Brake, and HumanInterface. Assistance consists of the structured resource instances hw1:ECU, hw2:ECU, and hw3:ECU. All these instances are connected by bus1:CAN. The hardware platform part Brake consists of the structured resource instances hw4:ECU, hw5:ECU, hw6:ECU, and hw7:ECU. All these instances are connected by bus2:CAN. These ECUs have access to the temperature sensor named 1, the wheel speed sensor named 7, the caliper position sensor named 2, the parking brake actuator named 3, and the caliper clamp actuator named 5. The hardware platform part HumanInterface consists of the structured resources hw8:ECU and hw9:ECU. These ECUs have access to the pedal position sensor named 11, the brake force sensor named 12, the parking brake sensor named 9, and the brake feedback actuator named 13. The structured resource instances hw2:ECU, hw4:ECU, and hw8:ECU are additionally connected by bus4:CAN. [*PH15]

The allocation constraints are according to Aleti [Ale15]:

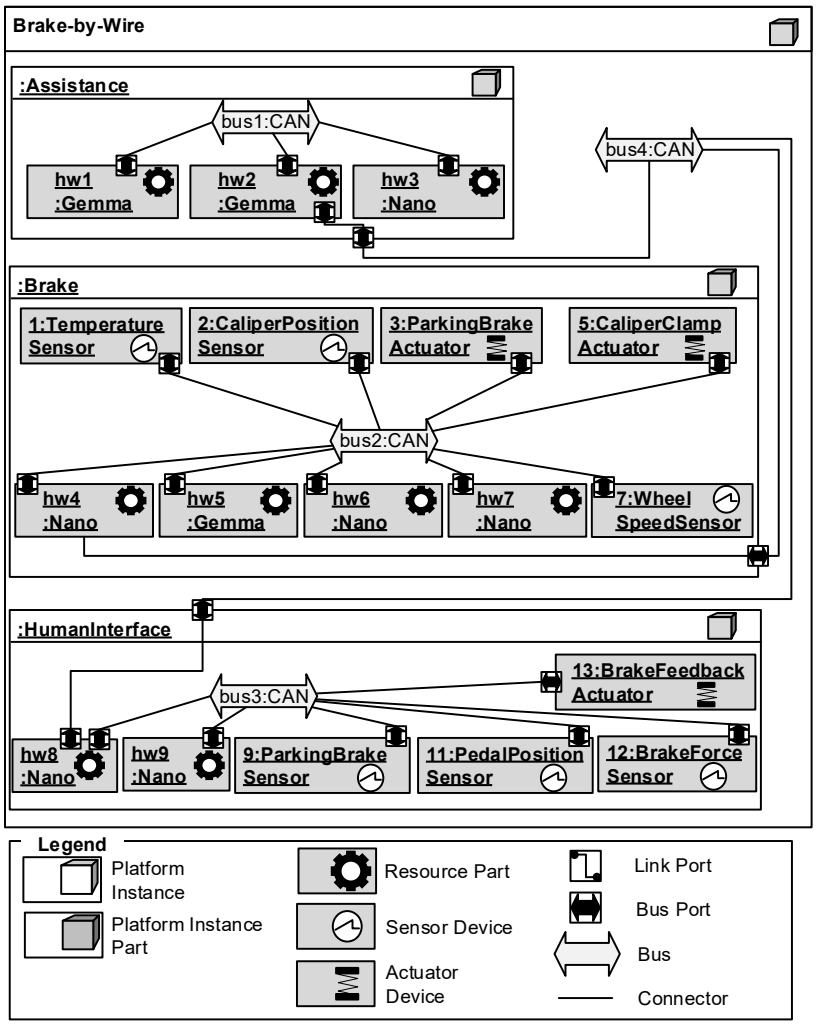


Figure 4.22: Brake-By-Wire Hardware Platform Instance Configuration (Adapted from [*PH15])

B.1 The component instances sc11/PedalPosition:Sensor and sc12/BrakeForce:Sensor must be collocated.

B.2 The component instances sc8/WheelSpinDetector and the software component instance sc7/WheelSpeed must be collocated.

Furthermore, we added the following constraints:

B.3 The component instances sc1/Temperature, sc2/CaliperPosition, sc3/ParkingBrake, and sc5/CaliperClamp have to be allocated to the resource instances hw4-hw7.

B.4 Communicating components have to be allocated to connected resource instances that share the same bus.

B.5 Software component instances, which access devices, have to be connected to the same bus.

B.6 The consumed memory has to be smaller than or equal to the available memory.

B.7 Task scheduling with respect to the EDF scheduling strategy should be possible.

4.10.2 Hypotheses

We set the following evaluation hypotheses for this case study. Hypotheses H1 and H2 refer to our evaluation question Q1. Hypothesis H3 refers to our evaluation question Q2. Furthermore, Hypotheses H4 and H5 refer to our evaluation question Q3 and hypothesis H6 refers to our evaluation question Q4.

- H1 Using the platform-specific modeling extensions (cf. Section 4.11) enables allocation engineers to formalize the platform-specific software resource demands, which are required for computing an allocation automatically.
- H2 Using the HPDL (cf. Section 4.3) enables hardware specialists and hardware platform architects to formalize the hardware capabilities, which allocation engineers require for computing an allocation automatically.
- H3 The effort of allocation engineers formalizing allocation constraints using the ASL reduces in comparison to formalizing the constraints as an ILP manually.
- H4 The allocation specification that is automatically computed is a feasible allocation specification. A feasible allocation specification fulfills (1) all allocation constraints and (2) each software component instance of the input component instance configuration is allocated to exactly one resource instance of the input hardware platform instance configuration.
- H5 The allocation specification that is computed is empty if no feasible allocation specification exists.
- H6 The time for creating the ILP model and for solving the corresponding ILP model increases in relation to the number of component instances and ECUs.

4.10.3 Analysis Procedure

We rate the first two hypotheses H1 and H2 as fulfilled if we are able to model the hardware platforms and corresponding platform-specific resource demands of the component instances. Furthermore, the allocation planning must be able to gather all required resource information to calculate a feasible allocation specification. Especially, the required resource constraints for EDF scheduling and memory consumption require detailed software and hardware properties. We rate the hypothesis H3 as fulfilled if allocation engineers have to write fewer lines of code using the ASL in comparison to a corresponding ILP. Subsequently, we rate the hypothesis H4 as fulfilled if the computed allocation specification is feasible, which we check manually. Furthermore, we rate the hypothesis H5 as fulfilled if modifying the allocation constraints in such a way that no feasible solution exists results in an empty allocation specification. Additionally, we use the unit tests that we describe in Section 4.9 to check that our ASL to ILP transformation works as expected. Lastly, we rate hypothesis H6 as fulfilled if the required time increases in relation to a higher number of allocated model elements.

4.10.4 Preparation of the Data Collection

In preparation for the case study, we model the component instance configurations of both systems manually. Figure 4.3 on Page 90 and Figure 4.21 on Page 136 show these component instance configurations. We use the platform-specific modeling extensions RequiredMemory to specify the memory consumption, WCET to specify the required processing time per execution, and Scheduling to specify the period and deadline of software tasks. Furthermore, we model the hardware platform instance configurations that Figure 4.10 on Page 102 and Figure 4.22 on Page 137 show manually. Additionally, we prepare for the scalability analysis different scaling configurations of the software and hardware model of the Brake-by-wire case. The

Table 4.5: Allocation Correctness Evaluation for Overtaking Example

Constraint	Type	Fulfilled	Explanation
A.1	separate location	yes	component instances overtakeeCommunicator and overtakeeDriver are allocated to different ECUs
A.2	collocation	yes	component instances overtakeeHMI and overtakeeDisplay are allocated to the same ECU
A.3	required location	yes	overtakeeDriver is allocated to an ECU (motor_board1) that is part of the Chassis platform
A.4	required location	yes	communicating component instances share always a common bus
A.5	required location	yes	component instances that access a device share always the same bus with the device
A.6	required resource	yes	the available memory is always greater or equal than the consumed memory
A.7	required resource	yes	processor utilization is always smaller than 100%
A.8	required resource	not considered	constraint formalized using the ASL, but not considered during solving due to a technical limitation

configurations differ in the number of component instance configurations and hardware platform instance configurations. Lastly, we model the allocation constraints of both systems using the ASL. Section 4.1 describes the constraints for the running overtaking example and the beginning of this section describes the constraints for the Brake-by-wire system. We use the MechatronicUML Tool Suite [*DGB+14] and the plugins that Section 4.9 describes for specifying all models.

4.10.5 Data Collection Procedure

We perform the evaluation by starting the "Plan System Allocation" wizard within the MechatronicUML Tool Suite. We select the root component instance configuration, the root hardware platform instance configuration, and the allocation specification that we define by using the ASL of both systems. Afterward, we start the transformation process. The process creates and solves a linear program model by performing the transformation that Section 4.6.3 describes. Furthermore, it creates model of the system allocation specification by performing the back-transformation that Section 4.6.4 describes.

Table 4.3 on Page 132 shows the computed allocation specification for the overtakee. We get the same allocation result during our case study, which Section 4.8 describes . We check manually if the constraints **A.1**–**A.7** of the overtaking example are fulfilled using Table 4.3. Table 4.5 shows the results. The considered constraints **A.1**–**A.7** of the overtaking example are fulfilled.

Furthermore, Table 4.6 shows the computed allocation specification for the Brake-by-wire system. The rows show each component instance and the columns show each resource instance. A component instance is allocated to a resource instance if the value of the row/column intersection cell is equal to 1. A component instance is not allocated to a resource instance if the value is 0. Furthermore, in the last three rows the table shows the available memory of

Table 4.6: Computed Allocation (Adapted from [*PH15])

ECU→ SC↓	**hw1**	**hw2**	**hw3**	**hw4**	**hw5**	**hw6**	**hw7**	**hw8**	**hw9**
caliper	0	0	0	1	0	0	0	0	0
sc1	0	0	0	1	0	0	0	0	0
sc2	0	0	0	1	0	0	0	0	0
sc3	0	0	0	1	0	0	0	0	0
sc4	0	0	0	0	1	0	0	0	0
sc5	0	0	0	1	0	0	0	0	0
sc6	0	0	1	0	0	0	0	0	0
wheel	0	0	0	1	0	0	0	0	0
sc7	0	0	0	1	0	0	0	0	0
sc8	0	0	0	1	0	0	0	0	0
sc9	0	0	0	0	0	0	0	1	0
brake Pedal	1	0	0	0	0	0	0	0	0
sc10	0	1	0	0	0	0	0	0	0
sc11	0	0	0	0	0	0	0	1	0
sc12	0	0	0	0	0	0	0	1	0
sc13	0	0	0	0	0	0	0	1	0
available memory	512 B	512 B	1024 B	1024 B	512 B	1024 B	1024 B	1024 B	1024 B
consumed memory	0 B	512 B	1024 B	704 B	512 B	0 B	0 B	384 B	0 B
processor utilization	0%	12%	20%	83%	10%	0%	0%	56%	0%

each resource instance, the sum of the consumed memory on a certain resource instance for the given allocation specification, and the overall processor utilization.

We check manually if the constraints **B.1**–**B.7** of the Brake-by-wire system are fulfilled using Table 4.6. Table 4.7 shows the result. All constraints of the Brake-by-wire system example are fulfilled.

Additionally, we check that the allocation constraint specification with the additional constraint that contradicts the collocation constraint computes an empty allocation specification. We get an empty allocation specification, which corresponds to the expected result.

We count the Lines of Code (LOC) of the allocation specification using the ASL and the ILP of both cases for evaluating the specification effort. Our ASL Overtakee allocation specification, without the ASL library code, has 90 LOC. It results in an ILP with 2252 LOC. The ASL Brake-by-wire allocation specification has 103 LOC. It results in an ILP with 7046 LOC. The predefined ASL library has currently about 900 LOC. Furthermore, our transformation implementation has 885 LOC. In contrast to an ILP, the library and transformation code has to be written just once.

We evaluate the scalability using the Brake-by-wire case only. We measure the execution times on a Lenovo Thinkpad with an Intel Core i7-3540M CPU, which runs at a speed of 2.9 GHz. The computer has 8GB DDR 3 RAM with a speed of 1600 MHz and a Samsung 840 EVO hard disk with a capacity of 500 GB. We use Windows 7 (64 Bit) as the software platform and SCIP [SCI; MFG+17] as the ILP solver. The input of the SCIP is able to compute the

Table 4.7: Allocation Correctness Evaluation for Brake-by-wire System

Constraint	Type	Fulfilled	Explanation
B.1	collocation	yes	component instances sc11 and sc12 are allocated to the same ECU (hw8)
B.2	collocation	yes	component instances sc7 and sc8 are allocated to the same ECU (hw4)
B.3	required location	yes	component instances sc1, sc2, sc3, and sc5 are allocated to the ECU hw5
B.4	required location	yes	communicating component instances share always a common bus
B.5	required location	yes	component instances that access a device share always the same bus with the device
B.6	required resource	yes	the available memory is always greater or equal than the consumed memory
B.7	required resource	yes	processor utilization is always smaller than 100%

input in the LPSolve format. Figure 4.23 shows the scalability analysis results. Each diagram shows the following six graphs:

1. Time for serializing LPSolve text from an ILP specification model.
2. Time for creating the general ILP constraints (cf. Section 4.6.3.1).
3. Time for creating the collocation constraints (cf. Section 4.6.3.3).
4. Time for creating the required location constraints (cf. Section 4.6.3.5).
5. Time for creating the required resource constraints (cf. Section 4.6.3.6).
6. Time for solving the ILP specification (cf. Section 4.6.1).

We analyze three different scaling situations using the Brake-by-wire case. Firstly, in the upper diagram Figure 4.23 shows the scenario that scales the number of Brake-by-wire software component instance configurations (cf. Figure 4.21). Table F.6 in Appendix F.2 shows the corresponding raw data. Secondly, in the middle diagram Figure 4.23 shows the scenario that scales the number of Brake-by-wire hardware platform instance configurations (cf. Figure 4.22). Table F.7 in Appendix F.2 shows the corresponding raw data. Thirdly, in the lower diagram Figure 4.23 shows the scenario that scales the number of Brake-by-wire software component instance configurations and hardware platform instance configurations equally. Table F.8 in Appendix F.2 shows the corresponding raw data. Additionally, we evaluate both cases in its standard configuration in more detail to ensure that our performance evaluation produces reliable results. Appendix F.2 describes these results.

4.10.6 Interpreting the Results

The result of our case study shows that we are able to formalize resource demands of component instances for memory usage and processor usage that is usable during allocation planning. Therefore, we rate our first evaluation hypothesis H1 as fulfilled. Additionally, the MechatronicUML extension mechanism, which we use for specifying platform-specific software resource demands, enables allocation engineers to provide light-weight meta-model extensions if further resource demands should be modeled. Likewise, we rate the second hypothesis H2 as fulfilled because we are able to formalize the hardware capabilities that are required for allocation planning that considers topology, routing, and resource constraints. In

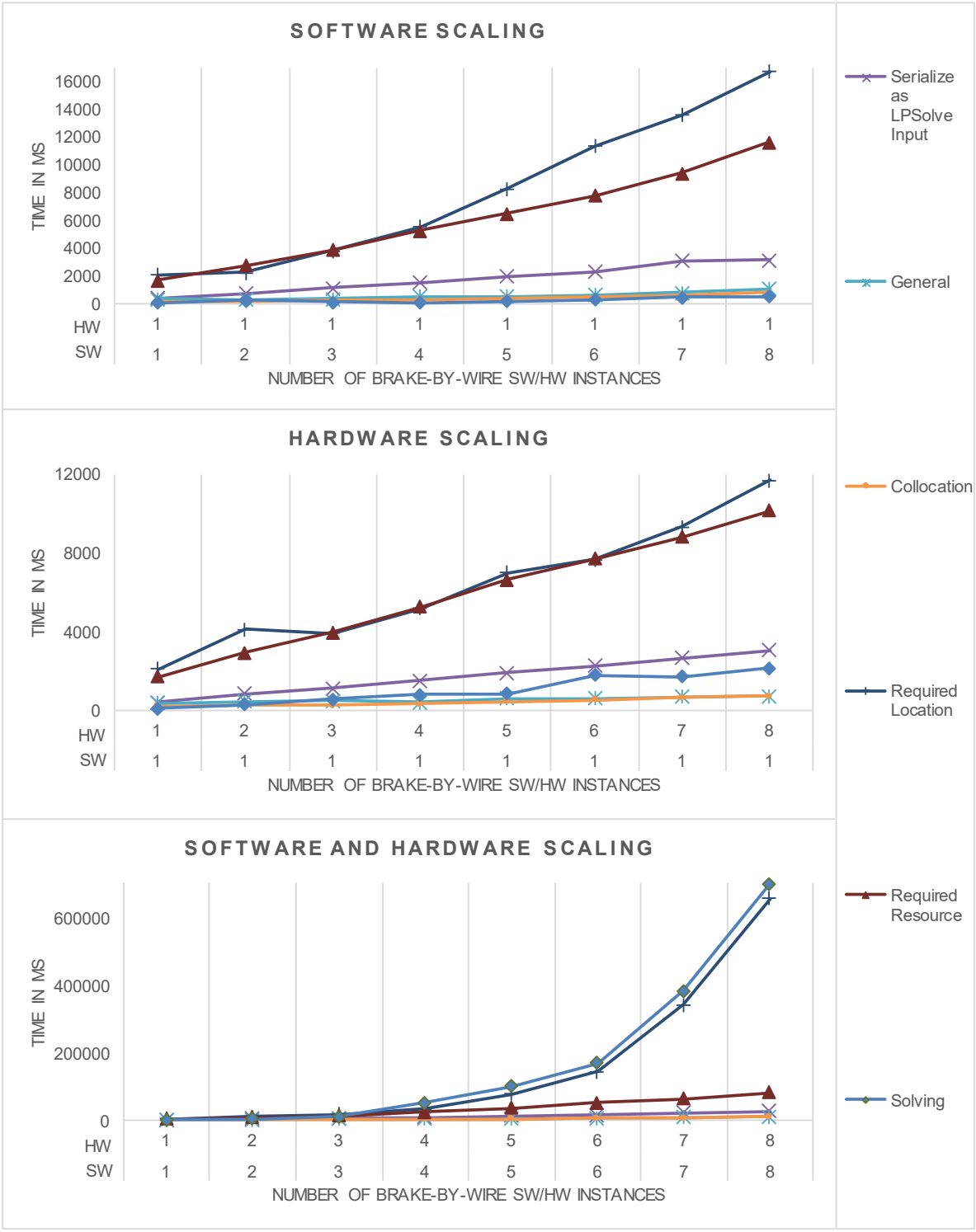


Figure 4.23: Scalability Analysis using the Brake-by-wire Case

our opinion, we can answer the research question Q1 with *yes* because our modeling approach provides the required information for allocation planning. For both cases, we show that we provide enough information to solve complex allocation problems within the automotive domain. Nevertheless, due to our threats to validates, which the next section describes, models from other domains or special resource requirements, like energy consumption, may require additional information.

Modeling the $8 + 7 = 15$ allocation constraints of both examples using the ASL results in an ILP with $537 + 2001 = 2538$ constraints. We need $90\ LOC + 103\ LOC = 193\ LOC$ using the ASL. Encoding the same constraints for our both examples results in an ILP with $2252\ LOC + 7046\ LOC = 9298\ LOC$. Furthermore, in contrast to our ASL, an ILP cannot decouple the pure constraint specification from the software model and hardware model. As a result, these models are much more complex and require a lot of effort to be created and to be maintained manually. Therefore, we rate the evaluation hypothesis H3 as fulfilled. We can answer research question Q2 based on the results of the two cases. We get a relative change of $\frac{(193\ LOC - 9298\ LOC)}{9298\ LOC} \approx -0.9792$ from an ILP specification to an ASL specification when we use LOC as the metric for measuring the effort.

The result of the automatic allocation planning shows that we calculate two feasible allocation specifications and one empty allocation specification for an allocation specification without a feasible solution. Furthermore, all unit tests, which check that the ASL to ILP transformation produces expected results, pass. Therefore, we rate the evaluation hypotheses H4 and H5 as fulfilled. Concerning research question Q3, we claim that we always get a feasible or an empty allocation specification. Especially, because we provide a formal proof that our transformation preserves the defined semantics and our evaluation shows that our implementation provides the expected results.

Concerning the scalability, Figure 4.23 shows with one exception that the time increases when the number of elements increases. The exception takes effect when the ILP model that should be solved becomes insolvable. In this case, the required time for getting a solution first drops from 230 ms to 120 ms and from 120 ms to 60 ms when increasing the number of Brake-by-wire component instance configurations from 2 to 3 and from 3 to 4. When increasing the model size even more the solving time increases again. The time that ILP solvers require depends on different factors. Besides the number of variables and equations/inequalities also the size of the search space, feasible solution space, and structure of the problem affect the runtime [Luk10]. Furthermore, different solvers use different search strategies. Even, the solver performance of the same solver for the same problem has a high variability on different platforms [KAA+11]. Therefore, we cannot judge why the time drops but we guess that it becomes partly easier to rate the problem instance as insolvable for the solver. In general, we conclude that our hypothesis H6 is true that the transformation and solving time increases when the problem size increases. Figure 4.23 shows in relation to research question Q4 that the transformation time and the model size have a linear relationship and that some have a non-linear relation. During our development, we did not measure and optimize the performance of the constraint implementation and transformation implementation. It is left for future work to evaluate how the performance can be optimized and what are good and bad practices for implementing queries and the transformation with regard to scalability and performance. The largest model for which our approach is usable on our evaluation computer has 104 component instances and 72 ECUs. In comparison to actual automotive systems, the considered number of component instances is about a factor of 10 smaller but the number of ECUs is only about a factor of 1.5 smaller. We conclude that our approach is promising to be used in practice if a system can be separated in independent subsystems for which the allocation specification can be planned independently.

4.10.7 Threats to Validity

Empirical studies have limitations and the results rely to a large extent on the research design. Each empirical study has to face threats to validity [RH08]. We use the systematics of Runeson and Höst for analyzing and reporting these threats by splitting the threats into the construct validity, internal validity, external validity, and reliability. Furthermore, we report the corrective actions that were performed to reduce the impact of the threats.

4.10.7.1 Construct validity

The first case was designed by the same researcher that developed also the model-driven allocation approach. Therefore, a threat is that the construction of the case by the same researcher has a bias based on the already developed approach. Nevertheless, as a corrective action the second case study was designed on the basis of the case study of Aleti [Ale15]. Furthermore, we compare the LOC of the ASL with the LOC that we generated by ourselves. The constraints might be encoded within an ILP in a more compact way. However, our ILP encoding is based on an encoding described by [MMM12] and our early manual encoding show comparable results. Our evaluation shows also that the numbers of LOC differ significantly.

4.10.7.2 Internal validity

Our input model consists of different modeling aspects and each aspect may affect the transformation time, the solving time, the size of the ILP, and the overall scalability in a different way. Furthermore, structural aspects of the input model may affect these time and size properties.

4.10.7.3 External validity

External validity is a big threat to our case study because we cannot guarantee that our results can be generalized to other cases or domains. We only use two cases from the automotive domain and we only consider our specific MechatronicUML meta-model as a context model. We may have missed a type of allocation constraint, where the ASL, the HPDL, and the platform-specific modeling extensions are not sufficient. We only specify 15 constraints for two examples. We specify 3 collocation constraints, 1 separate location constraint, 6 required location constraints, and 5 required resource constraints. Although we see these constraints for the two examples as realistic, other constraints could be different.

Furthermore, we compare the effort to specify constraints using the ASL only with the effort to specify constraints using an ILP. Our comparison results cannot be transformed to other constraint languages like CLP [JL87; JM94] or Satisfiability Modulo Theory (SMT) [DB11] directly because these languages support richer logic predicates and quantifiers [WW98].

We consider only allocation constraints from a part of the automotive domain. Allocation constraints from other domains or parts could be different. Nevertheless, in our opinion, the results should be transferable because other domains, like avionics, railway, manufacturing, use distributed platforms and the allocation of software is restricted [ABG+13].

We assume that we could measure equally strong results if we consider further cases designed by MechatronicUML. Additionally, we assume that we could measure equally strong results if other EMF-based meta-models are used where model elements of a specific type should be allocated to one other target model element of a specific type. For example, we rate production systems from the manufacturing domain as a very promising domain. The IEC 61499 [IEC61499] specifies a reference model for distributed automation [YRBK15], where our results are applicable. [*PH15]

4.10.7.4 Reliability

The check of the constraints' correctness, which Table 4.5 and Table 4.7 show, was performed manually by a researcher. The researcher may have made a mistake during checking the constraints correctness. Furthermore, the researcher was an expert at using our ASL and of the language OCL. He may have a biased impression of the effort that is needed for specifying allocation constraints. It is our personal opinion that it is manageable to specify valid allocation constraints, which may not be true for novices, engineers, and researchers from different domains or for researchers with a different personal background.

4.11 Limitations

Our approach for the model-driven allocation engineering underlies the following limitations:

1. We only support binary allocation decisions, i.e., a component instance is either completely allocated to a resource instance or not at all. We do not support to split a software component and to allocate parts to different resource instances or to alter the component instances configuration or hardware platform instance configuration automatically. It is possible to execute an atomic component instance on a single ECU in parallel if the ECU has a processor with more than one core [*GPS17; Gei15].
2. We do not support to allocate continuous component instances to sensors or actuators. We consider continuous component instances as black-boxes and assume that they require processing time to be executed. Nevertheless, it may be useful to consider continuous components just as stubs that could be allocated to hardware devices. As a result, it would be possible to consider messages between discrete component instances and sensor devices via a bus. Currently, both instances would be allocated to the same processing resource. Therefore, no message traffic on a bus would occur.
3. Our approach is limited to compute allocations to find a feasible solution. It does not support changing replications of component instances or changing ECU parameters, like the size of memory. Allocation is the only *solution category* [ABG+13] that we support.
4. In this chapter, we do not provide use cases, allocation constraint specifications, or OCL operations to allocate connector instances of a component instance configuration to network connectors or network buses. Such an allocation could be useful if alternative routes between two component instances exist.
5. In this chapter, we only consider defining allocation constraints and finding feasible solutions and not optimal solutions. The project group Cybertron [BCD+14] defines an extension of the ASL to measure the quality of allocation specification and defining optimization goals. It is part of future work, to use this extension to define meaningful optimization goals for cooperative CPSs.
6. We do not support constraint logic programming [JL87], satisfiability modulo theories [DB11], or mixed integer linear programming models [NTAM16] for solving the allocation problem. Currently, we only implement 0-1-ILP-based solving via LPSolve [LPS] and SCIP [SCI]. Furthermore, we support meta-heuristic [BR03] solving of the generated ILPs via Opt4J [OPT4J]. Opt4J can solve ILPs via meta-heuristic search methods if the coefficients are not of type float.
7. We do not implement a location constraint that relies on Safety Integrity Levels (SILs) [ISO26262-6] that constrains the allocation of higher-level component instances to lower-level structured resources. We have to provide meta-model extensions that define SILs for component instances and structured resources to support such a *safety-constraint* [VEH14]. Furthermore, we have to provide a corresponding ASL constraint.

8. Scaling is naturally a big problem for solving an NP-complete problem. Currently, our approach performs no optimization to improve the scalability. The generated linear program model may be analyzed and solved more efficiently by approaches that combine ILP or SAT solvers with meta-heuristics [OPT4J] or with CLP approaches [MP12; DB11]. Furthermore, mixed-integer linear programming models might be more efficient [NTAM16].

4.12 Related Work

Our model-driven allocation engineering approach relates mainly to two research areas. Firstly, Architecture Description Languages (ADLs) [Cle96], which provide capabilities to model hardware platforms and hardware resources for CPSs. Secondly, automatic allocation planning [Dea07] or respectively design space exploration approaches [ABG+13; LGHT08].

4.12.1 Architecture Description Languages for Modeling Hardware

Clements [Cle96] defines system-oriented (i.e.,hardware) features and process-oriented features that ADLs provide. In the following, we present different approaches, which consider hardware architecture properties. We distinguish between approaches (1) that are based on UML profiles and approaches (2) that provide their domain-specific meta-model. In the second group, we examine automotive approaches separately. In general, we focus on approaches that consider distributed embedded systems. These approaches offer methods to specify hardware properties, e.g., hardware resource description, like memory capacity and the interconnection of ECUs, like communication buses. Note, several of the related approaches that we describe in the following sections do not have a strong focus on the hardware model although it is an important part for model-driven allocation engineering.

4.12.1.1 UML Profile-Based ADLs

"The UML profile MARTE [MARTE] provides a hardware model for real-time embedded systems. The hardware model provides different views on different levels of abstraction. MARTE also differs between types and instances. In contrast to our model, MARTE provides a prescriptive description of the platform for hardware design, which is superfluous for our purpose. That means they consider much more details for other purposes. Moreover, MARTE does not differ between the resources for execution of software and a platform, which is useful if a hardware platform should be grouped into several independent logical parts. We need this for defining allocation constraints on the basis of specific platform parts. Further, MARTE does not support the definition of multiplicities." [*PMDB14]

"The CHESS modeling language [CSCS13] provides a UML profile, a method, and a toolchain for the component-based development of real-time embedded systems. The CHESS modeling language is based on certain languages such as MARTE [MARTE], SysML and EAST-ADL [EAST-ADL]. The CHESS specification of a platform model is similar to the specification of a software component model, but in contrast to our model, it neither supports multiplicity nor does it differ between resources and platforms." [*PMDB14]

"The D&C Specification [OMG06] is a UML profile for component-based distributed systems, which provides a model for distributed hardware platforms. The SAE Architecture Analysis & Design Language AADL [FGH06] provides an architectural model to describe a hardware platform. In contrast to our model, the D&C and AADL specifications do not differ between

types and instances. Furthermore, the platform cannot be hierarchically composed and multiplicities cannot be specified either." [*PMDB14]

4.12.1.2 Domain-Specific ADLs with Independent Meta-Model

"ProCom [CFMS10], SOFA HI [PWT+08], and RoboCop [Maa05] are component models for embedded systems. They provide runtime environments or virtual nodes for [deploying] their software components. The runtime environments or virtual nodes handle the resource management and abstract from the concrete underlying hardware platform. ProCom supports in a second task the mapping of virtual nodes to physical nodes of a hardware platform. The hardware platform description [CFMS10] is very simple. It consists of physical nodes with different types of hardware ports and different types of connections." [*PMDB14]

"Malek et al. [MMM12] provide a framework for automatic allocation planning. Their language provides nodes for hardware and links between them to represent a network. For describing the properties of a platform it is possible to choose between a few predefined properties for hardware nodes like the size of the memory or for network links reliability and bandwidth. They provide no differentiation between computation resources and devices, like actuators and sensors. Further, they provide no hierarchical composition for a hardware platform." [*PMDB14]

"Sapienza et al. [SSC13] provide a meta-model for component-based systems, which differs between software and platform components. Further, platform components can be of type hardware or software. In contrast to our meta-model, they provide no support for multi-view modeling and hierarchical platform composition." [*PMDB14]

4.12.1.3 Automotive ADLs

"AUTOSAR provides a hardware model [AUTOSAR14d] for electronic control units for the automotive industry. The model contains the necessary information to assist the system partitioning / software allocation. AUTOSAR does not differ between resources and the platform. For describing the hardware platform and hardware resources, they only provide the element HWElement, which has as a description the HWCategory. The Category defines what kind of element the HWElement is, e.g., «AUTOSAR/ProcessingUnit». 'Currently no special attributes are defined for the processing unit.'[AUTOSAR14d, p.32]. As a result, it cannot be guaranteed on the basis of the AUTOSAR meta-model that a WCET for a software component can be calculated. It needs further modeling rules to enable an automatically interpretation." [*PMDB14]

"East-ADL [EAST-ADL] relies on AUTOSAR and extends it by adding abstraction levels. For example, it enables the specification of types and instances. In contrast to our model, AUTOSAR and East-ADL provide no concept of multiplicities of hardware elements and define no viewpoints, views, and concrete syntax. Further, we abstract from hardware details, like concrete connection pins and add the ability to calculate a valid allocation of software components to hardware nodes-based on our defined language." [*PMDB14]

4.12.2 Design Space Exploration for Allocation Planning

"Aleti et al. [ABG+13] provide a systematic literature review about software architecture optimization methods. In this field of research, 26 percent [of the considered publications] do not consider constraints at all. Only 7 percent give importance to physical constraints and 5 percent to memory constraints, which are critical for the design of CPSs. We present in the following [paragraphs] different [related] approaches, which consider safety-critical constraints, e.g., memory and are related to model-driven and component-based software engineering

methods." [*PH15] We focus on approaches in the field of distributed embedded systems and CPSs, but are not limited to them.

Kugele *et al.* [KPP+15] present a DSL for describing allocation constraints called System Architecture Optimization Language (SAOL) and a transformation to different constraint solver techniques. They support separate location respectively called dislocality, location, and resource constraints. Location constraints are split up into three types: (1) fix-constraints, i.e., a component instance requires a specific ECU; (2) matching-constraint, i.e., a component instance requires an ECU with a specific property; (3) compatibility-constraints, i.e., component instances with specific properties require an ECU with a specific property. In contrast to our language where we always describe the requirements of component instances Kugele *et al.* also allow to specify constraints for particular ECUs, which are called property constraints. The expressiveness is the same. Only the point of view for the constraint specification changes. In contrast to our approach, which uses OCL as its query language, Kugele *et al.* define new language elements, like forall, where, min, count, sum, logical operators, and arithmetic operators by themselves. SAOL supports evolutionary algorithms, SMT-based, and ILP-based solving strategies. In contrast to our approach, they do not provide a proof that the encoding within the different solving strategies preserves the constraint semantics. Like our approach, SAOL uses EMF and Xtext for the specification of the language and the relation to component instances and ECUs. In contrast to our approach, it cannot be used for any EMF-based allocation problem because the supported EMF types are restricted to the specific source and target meta-model.

"Aleti et al. [ABGM09] present a framework for modeling, evaluating, and optimizing embedded systems, called ArcheOpterix. ArcheOpterix can extract architecture descriptions by parsing different specification languages, like AADL or EAST-ADL. Furthermore, ArcheOpterix has an architecture analysis module that provides the *Architecture Constraint Validation Interface* to specify and evaluate constraints. As standard features, localization, collocation, and memory constraints can be specified. The constraint specification is done by matrices." [*PH15] ArcheOpterix uses evolutionary algorithms [Fog95], which is a meta-heuristic algorithm [BR03], to solve optimization problems. "In contrast to our approach, ArcheOpterix provides no support for deriving the constraint specification for complex models. However, ArcheOpterix could be a future back-end for solving and optimizing models that are specified by our presented ASL. For this, the MechatronicUML software components, hardware platforms, and computed allocation specification tuples have to be transformed into an ArcheOpterix input file. Additionally, constraint evaluators have to be developed in ArcheOpterix for *requiredResource*." [*PH15]

"Malek et al. [MMM12] present an extensible framework for improving a distributed software system's deployment architecture, called deployment improvement framework (DIF). They provide also a constraint model for collocation, location, and resource constraints. Cross tables are used to define location and collocation constraints. Therefore, constraints can only be defined by considering flat software architectures and hardware platforms. Information about the hierarchical architecture cannot be used for defining constraints. For a large set of components and ECUs, defining constraints is tedious and error-prone. In contrast to DIF, we use the power of OCL and a domain-specific language to ease the specification of constraints. Additionally, DIF provides transformations to different solving mechanisms for the allocation problem, like MINLP, MIP, Greedy, and genetic algorithms. Our transformation to ILP is inspired by these transformations." [*PH15]

Zeller *et al.* [ZPW+11] present mathematical foundations for encoding the allocation problem for automotive systems as a system of SAT-based linear equations. Zeller *et al.* describe how to encode required location constraints, like a specific task to ECU assignments and end-to-end deadlines. Furthermore, they describe how to encode required resource constraints,

like memory, task scheduling, and network bandwidth usage. Zeller *et al.* use and compare Sat4J [LP10] and a simulated annealing algorithm [TBW92], which is a meta-heuristic-based approach, for solving the encoded allocation problem. In their evaluation, they show that SAT-based solving scales better than simulated annealing algorithm. "In our approach, we abstract from formal and difficult aspects by introducing the ASL and an automatic transformation to an ILP model. As a result, engineers can specify allocation constraints efficiently for an existing component-based software and hardware model and generate an allocation specification automatically. Our ILP-based formalization of allocation constraints is inspired by Zeller et al. and [is] therefore highly related." [*PH15] Furthermore, we integrate our approach in a model-driven specification approach that apart from allocation engineers supports software engineers and hardware platform architects.

Metzner and Herde [MH06] present mathematical foundations for encoding the allocation problem for distributed embedded real-time systems as a system of linear and non-linear integer equations, called RTSAT. They only consider required resource constraints. Especially, they focus on task schedulability and message schedulability that have to meet (end-to-end) deadlines. Furthermore, they describe how to encode memory constraints and to enforce, like our approach, that each task has to be allocated to exactly one ECU. Allocation engineers have to encode the allocation problem manually using the proposed equations. Metzner and Herde use the SAT solver GOBLIN [FH03] that is able to solve integer-arithmetic equations. Worst-case response times of tasks are calculated in a customized algorithm because it can be solved for partial allocations uniquely by using a fixed point equation [MH06]. As a result, they claim that their approach scales better than purely SAT-based solving. In our approach, we also calculate the response time of tasks and messages using OCL independently of the allocation problem. In contrast to RTSAT, allocation engineers do not have to encode the allocation problem manually as a formal system of equations in our approach. Nevertheless, our mathematical formalization of an allocation is comparable to the formalization as a SAT-based encoding, although we use a 0-1 ILP-based encoding.

"Koziolek and Reussner [KR11] present a quality optimization framework, called PerOpteryx, for component-based business information systems. They define a degree of freedom model that restricts the design space. Furthermore, by OCL and an input model of a component-based system, their software tool 'can automatically detect instances of the degrees, which define the design space, and instantiate the optimization problem.' [KR11] Possible allocation candidates are selected by using meta-heuristics [BR03]. These candidates may also represent an invalid allocation. Allocations can be constrained by OCL, but the constraints are checked after selecting candidates. Therefore, the meta-heuristic approach may select many invalid candidates before finding a valid candidate. PerOpteryx searches allocation candidates in a design space, which is restricted by a degree of freedom model and not by a constraint model. In contrast to PerOpteryx, our approach tackles CPSs, which have to fulfill many constraints. Therefore, we focuses on constraint solving. If optimization should also be considered then, according to Lukasiewycz [Luk10], firstly solving allocation constraints via constraint solvers and secondly, finding a optimal solution via meta-heuristics, is more efficient than the other way around, if the search space (searching without considering constraints) of a problem is huge compared to the valid design space (searching, while considering constraints) [Luk10]. Nevertheless, we only consider finding feasible solutions and not optimal solutions." [*PH15]

"Lukasiewycz [Luk10] presents an open design space exploration framework (OpenDSE) [Ope] for embedded systems. The framework provides an own meta-model, implemented in Java. It enables to model applications that consist of flat data-dependent tasks. A task can be allocated to a resource that is structured in a flat hardware architecture. Engineers can create the application model by instantiating corresponding objects in Java. Constraints can also be programmed directly in Java by implementing the SpecificationConstraint interface. For

optimization, OpenDSE uses Opt4J [LGRT11], which combines constraint solving with meta-heuristic optimization [BR03]. In contrast to OpenDSE, our allocation approach is combined with a model-driven component-based software engineering approach for CPSs. Nevertheless, we think the combination of constraint solving and meta-heuristic-based optimization is promising for CPSs. Therefore, we provide (...) [a] transformation to Opt4J for solving and optimizing allocation specifications." [*PH15]

Schätz, Hölzl, and Lundkvist [SHL10] present a tool for the design-space exploration via a model transformation. They do not focus on a specific domain. They demonstrate their approach by implementing allocation planning for a very simple custom software component and hardware meta-model using EMF. Furthermore, they provide custom textual DSLs to model software, hardware, and constraints in a rule-based manner. The kinds of constraints that can be formalized are not limited to a specific subset. The formalized model is transformed to Prolog [CM87], solved, and transformed back into an EMF model. In contrast to the approach by Schätz, Hölzl, and Lundkvist, our approach supports hierarchical software and hardware models and is based on the standardized constraint language OCL. Furthermore, our transformation to ILP takes advantage of the allocation type. Therefore, allocation engineers do not have to cope with the encoding of different allocation types and a transformation to a system of equations in our approach.

Voss et al. [VEH14; BV15] present a tool, called AutoFocus3, for the design-space exploration for the allocation planning of automotive systems. The software and hardware modeling is based on a custom DSL that enables software engineers and hardware platform architects to specify flat system models. AutoFocus3 supports to model required location constraints by defining a fixed mapping and defining if SIL properties should be considered. Furthermore, resource constraints in the form of processor, message scheduling, memory, and energy consumption can be considered. Allocation engineers can select whether the constraints should be considered during the allocation planning or not. The supported constraints are hard-coded within the tool AutoFocus3. AutoFocus3 [AVT+15] uses a transformation [VS12] to the SMT [DB11] solver Z3 [DB08] for solving allocation problems. SMT combines the strength of SAT-based approaches with constraint logic programming [JL87]. In contrast to Autofocus3, our approach supports collocation constraints and separate location constraints. Furthermore, our approach is not limited to a fixed set of resource constraints and uses the power of OCL to query, e.g., complex location constraints. In our approach, allocation engineers are able to add new allocation constraints. Our transformation to an ILP and the semantics definition of the ASL take advantage of the allocation type and are therefore not limited to a fixed set of constraints. Additionally, our hardware model provides hierarchical composable hardware architectures with more resource details. Nevertheless, SMT is a promising back-end for solving the allocation problem and could replace our formalization as an ILP.

Feljan, Carlson, and Seceleanu [FC14; FCS12] present a toolbox for MATLAB Simulink [Sim] to allocate tasks to ECU cores for multicore embedded systems. The software architecture and the ECU cores are specified using MATLAB Simulink. Constraints are encoded by MATLAB programming. They support collocation, separate location, and required location constraints. Resource constraints are currently not considered. Nevertheless, they focus on optimizing the end-to-end response times for tasks. Feljan, Carlson, and Seceleanu solve the allocation problem by a local search strategy. The transformation is hard-coded by using MATLAB programming. Constraints are checked programmatically. Furthermore, they rate the quality of allocations by simulations performed with MATLAB Simulink. In contrast to [FC14; FCS12], we focus on specifying allocation constraints and provide an automatic transformation to a solver which is transparent to the allocation engineer. Furthermore, we focus on allocating component instances to distributed connected ECUs and not on specific

components. Geismann, Pohlmann, and Schmelter [*GPS17; Gei15] show how to allocate tasks to ECUs by using our ASL. Afterward, they map the tasks to specific cores for each ECU.

Švogor et al. [ŠCV13b; ŠCV13a; ŠC15] present a formal model and first tool prototype for multi-criteria software component allocation on a heterogeneous platform. Software and hardware models are defined with set theory formally. Required resource constraints and optimization goals are specified using a system of equations. Furthermore, they state that collocation constraints, separate location constraints, and required location constraints could also be defined by additional equations. Additionally, they define how to specify matrices for specifying resource consumption. Švogor et al. use a genetic algorithm, which they create manually for their specific model instance to solve and optimize the allocation problem. They focus on the optimization of solutions and not the constraint solving parts. Švogor and Carlson [ŠC15] describe a first prototype, called SCALL, which implements their approach. In contrast to [ŠCV13b; ŠCV13a; ŠC15], allocation engineers who are using our approach do not have to encode the allocation problem manually as a constraint satisfaction problem. Furthermore, we provide tooling for our approach and provide case studies. Nevertheless, our mathematical formalization of an allocation is similar to the formalization of Švogor et al.

Wang, Merrick, and Shin [WMS04] present an allocation planning algorithm for embedded systems within the automotive domain. They evaluate their approach on the basis of generated artificial models. Furthermore, they directly encode a fixed set of allocation resource constraints within their algorithm. They only consider resource allocation constraints. Wang, Merrick, and Shin implement a custom allocation algorithm that uses an informed branch-and-bound strategy and forward checking based on a domain-specific allocation heuristics. In contrast to [WMS04], our approach enables allocation engineers to define custom allocation constraints for their software and hardware model instances. Furthermore, with ILP, we use a general-purpose solver that is not domain specific. Therefore, we benefit from any improvement of ILP solvers. In the future, the performance of allocation planning could profit from a hybrid approach that combines ILP solving with a guidance that is based on specific domain knowledge.

Kuchcinski [Kuc01] presents a formal model for distributed embedded systems and domain constraints. Constraints are formalized as CLP. Especially, they formalize so-called *global constraints* to express complex relations between domain variables. A global constraint is somehow comparable to a constraint type, which we use in our ASL. Kuchcinski discusses in his paper mainly resource constraints. He uses two meta-heuristic approaches together with a branch-and-bound algorithm to solve the CLP. Kuchcinski searches not only for a feasible allocation that is schedulable; he directly searches for a valid schedule. Therefore, he could consider more complex task dependencies during design-space exploration. As a result, the exploration does not only have to find feasible values for decision variables that define an allocation but also has to find values assigned to system parameters, e.g., when a task should be scheduled. In contrast to [Kuc01], we focus only on allocation constraints. The approach of Kuchcinski requires that allocation engineers have to encode the system model and all constraints directly using CLP. As a result, they need to be experts in this field because they have to know how to achieve the formal constraints and not only which constraints they want to have. In contrast to our approach, the approach of Kuchcinski is not embedded in a model-driven design approach, like MechatronicUML.

Klobedanz [Klo14] presents a formal model for distributed embedded systems that is comparable to [Kuc01] but also considers redundancy. In contrast to [Kuc01], who uses CLP, he uses a greedy-based approach with polynomial runtime to solve the allocation problem. The greedy heuristic considers scheduling properties. It is not clear how the greedy scheduling heuristics performs if several constraints or constraint types have to be considered. The greedy algorithm does not guarantee that no feasible solution exists if it terminates without finding a

feasible solution. In contrast, our approach guarantees that if no feasible solution is found that no feasible solution exists. Furthermore, our approach enables allocation engineers to define their own allocation constraints without altering the implementation of the allocation planning algorithm manually.

Frieben and Tichy [FT11] provide a model for the allocation of IEC 61499 function blocks onto distributed devices and of logical connections to networks segments or network links. They consider location, resource, and routing constraints. Furthermore, they consider the optimization of reaction times of execution chains in the context of predefined end-to-end deadlines and to minimize the network bandwidth on specific segments. The constraints and optimization rules are formalized by using the domain-specific *optimization constraint language* that is provided by the IBM's commercial ILOG CPLEX tool [CPLEX]. This language allows to define conditional constraints for subsets, likewise to OCL. Furthermore, in contrast to standard ILPs it provides logical expressions, like '*or*' or '*implies*' and mathematical functions, like *sum()* or *max()*. IBM's ILOG CPLEX uses constraint programming solvers and mixed integer solver [CPLEX]. In contrast to our approach, the approach of Frieben and Tichy transforms only the software architecture and hardware platform to the constraint model. The predefined constraints are specified directly within the constraint modeling environment that is provided by CPLEX. The solution allows only to define general constraints that must hold for the allocation of all function blocks or connections, or that depend on predefine properties of them. Constraints that depend on application specific properties of the concrete software architecture or hardware platform, e.g., a constraint for a component with a specific name have to be added within the generated constraint model manually. In contrast, our approach enables to specify such constraints by using OCL that can refer to specific names. Furthermore, our ASL provides four predefined allocation types, where as engineers who use IBM's *optimization constraint language* are able to define any mathematical constraint. Due to the limitation 4 that Section 4.11 describes, it is up to further research if our ASL supports the routing constraints of Frieben and Tichy in the context of allocating logical connections to networks segments.

4.13 Summary

This section summarizes to what degree the related work and MechatronicUML in combination with the results of this thesis fulfill the requirements R.4.1-R.4.8, which the introduction of this chapter defines for a model-driven allocation engineering approach for CPSs. We claim that the requirements are necessary but not sufficient because we focus on safety constraints but not, e.g., on optimization. Table 4.8 shows a summary of our opinion about the fulfillment of the requirements.

The requirement R.4.1 necessitates that a model-driven allocation engineering approach should be able to specify and refer to a hierarchical software architecture. Several software development approaches support hierarchical nesting (cf. Table 5.3 in Section 5.8). Nevertheless, only PerOpteryx, AutoFocus3, and IEC 61499/[FT11] support hierarchical nesting within their component-based software architecture and allocation approach. SAOL, DIF, [SHL10], and SCALL support only flat component-based software architectures within their specification and allocation approach. ArcheOpterix does not provide a component model by itself but uses the AADL or East-ADL but cannot refer to the hierarchical structures [ABGM09]. OpenDSE supports a network of flat tasks instead of a component-based software architecture. The other approaches only consider how to formulate allocation constraints on the basis of mathematical sets and do not provide an engineering approach. Therefore, we set the corresponding cell to "n.a.". MechatronicUML supports to specify and refer to component instances from

Table 4.8: Comparison of the Requirements Fulfillment for Allocation Engineering Approaches for CPSs (Legend: ✓: fulfilled; ◑: fulfilled partially; ∅: not fulfilled; n.a.: not applicable; ?: unknown)

Approach	R.4.1	R.4.2	R.4.3	R.4.4	R.4.5	R.4.6	R.4.7	R.4.8
SAOL [KPP+15]	◑	✓	◑	✓	✓	◑	✓	◑
ArcheOpterix [ABGM09]	◑	◑	◑	◑	◑	◑	∅	◑
DIF [MMM12]	◑	✓	◑	✓	◑	◑	∅	✓
Zeller *et al.* [ZPW+11]	n.a	n.a.	n.a.	n.a.	n.a.	◑	∅	✓
Metzner and Herde [MH06]	n.a.	n.a.	n.a.	n.a.	n.a.	◑	∅	✓
PerOpteryx [KR11]	✓	✓	◑	✓	✓	n.a.	n.a.	◑
OpenDSE [Luk10]	◑	✓	◑	✓	✓	◑	∅	✓
Schätz et al.[SHL10]	◑	∅	◑	∅	∅	◑	n.a.	◑
AutoFocus3 [AVT+15]	✓	?	◑	?	◑	◑	∅	✓
Feljan et al. [FC14; FCS12]	n.a.	n.a.	n.a.	n.a.	n.a.	◑	∅	?
SCALL [ŠCV13a; ŠCV13b; ŠC15]	◑	✓	◑	✓	∅	∅	n.a.	◑
Wang et al. [WMS04]	n.a	n.a	n.a	n.a	n.a	◑	∅	✓
Kuchcinski [Kuc01]	n.a.	n.a.	n.a.	n.a.	n.a.	◑	∅	✓
Klobedanz [Klo14]	n.a.	n.a.	n.a.	n.a.	n.a.	∅	∅	◑
IEC 61499 + IBM ILOG CPLEX [FT11]	✓	✓	◑	✓	◑	◑	∅	✓
MechatronicUML including this thesis	✓	✓	✓	✓	✓	✓	✓	✓

hierarchical software architectures. The ASL itself can also be used for other hierarchical software architectures, like a UML composite structure.

Concerning the requirement R.4.2, the approaches SAOL, PerOpteryx, OpenDSE, DIF, SCALL, and IEC 61499 [FT11] fulfill this requirement on the basis of the available resource requirements of the components/tasks. ArcheOpterix fulfills it only partially because it provides only a resource evaluator for memory resources although the AADL provides more resource requirements. Additional evaluators could be added. The other approaches either consider resource requirements not at all or do not provide an engineering approach. [SHL10] provides one attribute per component that is named load. We decide that this attribute does not fulfill the requirement because it is to simple to specify the resource demands of a software component. Considering the paper for AutoFocus3 [AVT+15] we could not decide if and how software resource consumptions are specified. MechatronicUML provides component instance extensions for specifying computing, memory, and network bandwidth consumption (cf. Section 4.4). Furthermore, the execution properties priority, deadline, and WCET can be annotated. These extensions can be added to existing component instance configurations. They can also vary for different ECUs. We refer to these properties by accessing the corresponding extensions by using OCL.

The hardware architecture is the second important part for defining allocation constraints and for planning a feasible allocation. Therefore, the requirement R.4.3 considers the specification of the hardware specification and the usage of the hardware architecture during the allocation planning. Besides MechatronicUML and ArcheOpterix, which enables to use East-ADL [EAST-ADL], no other approaches support the modeling of hierarchical hardware architectures as they are also offered by MARTE [MARTE]. SAOL, DIF, PerOpteryx, OpenDSE, [SHL10], AutoFocus3, SCALL, and IEC 61499 [FT11] support only flat hardware architecture modeling and usage for allocation planning. ArcheOpterix allows referring to the hardware leaf nodes. The other approaches do not provide an engineering method for using hardware architectures.

The requirement R.4.4 requires to access the provided resources of the hardware architecture. SAOL, PerOpteryx, DIF, OpenDSE, SCALL, and IEC 61499 [FT11] fulfill the requirement because they provide several resource descriptions. ArcheOpterix fulfills the requirement partially because it only supports to refer to memory resources, although AADL and East-ADL offer more resource properties. Additional evaluators could be added. [SHL10] provides one attribute per hardware node that is named load. We decide that this attribute does not fulfill the requirement to specify the provided resources of a hardware node because it does not cover the needs of a CPS. AutoFocus3 provides no information concerning its resource description and usage for allocation planning [AVT+15]. The other approaches provide no engineering approach for describing hardware resources. MechatronicUML provides a rich set of hardware properties that are selected based on the requirements for allocation planning that considers limited computing power, memory, and network bandwidth (cf. Section 4.3.2).

After setting the basis for allocation planning by describing the software architecture and the hardware platform, the third important part is the definition of allocation constraints. The requirements R.4.5, R.4.6, and R.4.7 define the different aspects of defining allocation constraints.

The requirement R.4.5 requires that allocation engineers must be able to specify which component instances must fulfill certain constraints because different component instances have different needs. SAOL, OpenDSE, and PerOpteryx fulfill this requirement. ArcheOpterix, DIF, and AutoFocus3 fulfill it partially because at least for some constraints the set of component instances is restricted by using matrices/selections. The approach of Frieben and Tichy [FT11] fulfills this requirement partially because the domain-specific optimization programming language, which is used as an allocation constraint specification, enables conditional/filtered constraints. Nevertheless, the approach does not enable engineers to define their own conditions or filter rules respectively. The other approaches do not provide any constraint restriction mechanism or do not provide an engineering method. Within the ASL of MechatronicUML, OCL is used to determine the subset of component instances that have to fulfill the constraint. Thereby, it is possible to compute the set of component instances on the basis of different properties.

The requirement R.4.6 necessitates that a model-driven allocation approach has to support to use predefined allocation constraints that specify component instance coherence/dependency, incompatibility, target platform properties, and resource usage. The most approaches provide mechanisms to define only a subset of the allocation constraint types. Section 4.12 describes the capabilities of each approach in detail. PerOpteryx is set to "n.a." because its allocation planning approach does not provide a constraint-driven solving strategy but enables to reject allocation candidates that are computed by meta-heuristics. SCALL only focuses on optimization and does not provide allocation constraints. The ASL of MechatronicUML provides the allocation types collocation, separateLocation, requiredLocation, and requiredResource to meet the requirements. Section 4.5 and Section 4.7 show how these constraint types can be used to define sophisticated allocation constraints for CPSs.

Besides the use of constraints, software engineers have to add certain domain specific constraints because the existing approaches do not provide all required allocation constraints. The requirement R.4.7 necessitates that software engineering experts are able to add new domain specific constraints, like EDF schedulability, without having to know how constraint satisfaction problems can be formalized mathematically. The only related work that focuses on this topic is SAOL. Nevertheless, SAOL does not support collocation constraints. The other approaches require that engineers have to define the encoding into a constraint satisfaction problem by themselves, which requires this specific knowledge. PerOpteryx fulfills the requirement R.4.7 indirectly because engineers use only OCL and do not cope with constraint satisfaction problems.

Nevertheless, PerOpteryx does not support a correctness-by-construction approach and only checks if an allocation specification candidate is feasible. In the approach of [SHL10], engineers must know how to encode new constraints in Prolog. SCALL does not take care of constraints at all. Therefore, we set these approaches to "n.a.". The ASL of MechatronicUML provides four allocation constraint types, which are necessary to define complex constraints for CPSs. Allocation engineers can add new constraints without having to know how to encode the constraints as a constraint satisfaction problem because we define for each allocation constraint type the transformation to an ILP. Furthermore, we prove that this encoding preserves the semantics of the allocation constraint types, which we define formally.

Lastly, the requirement R.4.8 necessitates that the allocation planning must be sound and efficient. Sound means, if the allocation planning terminates, it either finds a feasible solution or no solution exists. Furthermore, we require that the encoding as a constraint satisfaction problem must be defined clearly. Efficient means, the search for the solution should be done in a systematic way, which narrows down the search space steadily. DIF, Zeller et al., Metzner and Herde, OpenDSE, AutoFocus3, Wang et al., and Kuchcinski encode the allocation problem as constraint satisfaction problem. Thereby, they get a feasible solution by using efficient solvers. None of the approaches shows the correctness of the encoding. Nevertheless, we did not disprove the correctness of the described encoding and state that these approaches fulfill the requirement R.4.8. SAOL also encodes the allocation problem as a constraint satisfaction problem but does not describe how the allocation problem is encoded [KPP+15]. In our opinion, they fulfill the requirement for this reason only partially. ArcheOpterix, PerOpteryx, and SCALL use meta-heuristic approaches. The computed results are feasible. Meta-heuristic approaches can be more efficient than constraint solving if the search space and solution space is huge. Lukasiewycz [Luk10] and Zeller and Prehofer [ZP13] show that constraint solvers are faster and meta-heuristics become inefficient if the solution space is compared to the search space quite small. Furthermore, these approaches do not guarantee that if they terminate with no solution that no solution exists. Therefore, we state that ArcheOpterix, PerOpteryx, and SCALL only fulfill the requirement partially. The approach of Klobedanz [Klo14] uses a greedy approach, which may terminate with no feasible solution even if a solution exists. The approach does not reduce the search space systematically. Therefore, it only fulfills the requirement partially. Schätz, Hölzl, and Lundkvist [SHL10] use Prolog and apply their evaluation only on small size problems with eight component instances and three ECUs. We cannot decide if the resulting Prolog program can be solved efficiently. Therefore, we decide that [SHL10] fulfills the requirement partially. We cannot decide if Feljan et al. fulfill the requirement on the basis of the given information in [FC14; FCS12]. The ASL of MechatronicUML encodes the allocation problem as an ILP. In [*HP17], we prove that the encoding preserves the semantics of the allocation constraint types. Therefore, we claim that an ILP solver calculates a sound allocation specification. Furthermore, our evaluation in Section 4.10 shows that we are able to calculate a solution for a midsized allocation problem (104 component instances, 72 ECUs) on a standard laptop under half an hour using the solver SCIP. Therefore, in our opinion, the model-driven allocation engineering approach that we added to MechatronicUML performs an efficient allocation planning.

5

Software Construction

In Cyber-Physical Systems (CPSs) the reliability of the deployed software is essential during runtime. The reliability depends among other things on the correct implementation of the functional concerns of a component itself, like the state-based behavior and on the correct implementation of the technical concerns, like establishing a network connection to another component. Especially, handling the communication between components and its lifecycle on distributed platforms is critical for the overall system's behavior [SW07]. Furthermore, handling the lifecycle of components is critical. Component instances have to be created and configured correctly to run safely on an Electronic Control Unit (ECU).

Each component type has different context requirements caused by different communication partners, software platforms, hardware platforms, sensors, and actuators. Even the building of the software implementation can lead to errors because the build process of an executable depends on the allocation of the software component and its external dependencies.

The goal of this chapter is to automate the complex process of generating an implementation to a high degree. Thereby, we increase the implementation speed and increase the quality and reliability of the implementation using a repeatable process [Sel03]. Therefore, we require a precise definition of a component model, its semantics, and its context requirements. Furthermore, we require on the basis of the allocation specification a platform-specific configuration model that defines how a component accesses sensors and actuators. Additionally, we require for the software construction a code generation infrastructure that defines how the functional and technical concerns of a component are realized on a concrete platform, how they depend on each other, and how they depend on a middleware or operating system [VSK05]. Especially, the implementation of the functional parts of a component should be reusable for different domains like automotive or production and on different platforms without changing its implementation.

A general-purpose Model-Driven Engineering (MDE) process for constructing software from models is the Model-Driven Architecture (MDA) [OMG03]. Figure 5.1 shows an MDA-based transformation. MDA provides a terminology definition for MDE. It separates different concerns in several layers of the software model. The *Platform-Independent Model (PIM)* contains the design of the application software. The *Platform Model (PM)* contains a description of hardware and software platform parts. The platform-specific *mapping models* reference a PIM to a PM. Platform-specific marks, like tagged values of UML stereotypes, that can be associated to elements from the *PIM* may contain additional platform-specific information, like a library path. Transformation engines create a Platform-Specific Model (PSM) and code by using mapping models and marks.

In literature, architecture-centric model-driven development describes the basic practice how to use code generation where the software architecture plays the central role [SVC06]. The application implementation is split up into schematic repetitive code, generic code, and individual code. Model-to-Model (M2M) and Model-to-Text (M2T) transformations create the schematic repetitive code for each application model individually. The generic code contains

code that is the same for each application and for each component respectively, e.g., a library. Individual code is implemented manually for each component and platform, e.g., an adapter of an operation to another operation.

Furthermore, a layered architecture on the basis of containers for components is proposed by Völter, Salzmann, and Kircher [VSK05]. Figure 5.2 shows this architecture. The functional concerns of an application are embedded within components. The technical concerns are handled by containers. Between components and containers specific interfaces are defined. Furthermore, the literature defines patterns for the software construction of distributed systems [VSW02; BHS07]. However, the MDA process, the architecture-centric model-driven development, and the software construction pattern for distributed systems are independent approaches and are not adjusted for the functionality required by the distributed CPSs. Nevertheless, these approaches can contribute together to a solution how to automate the construction of the software.

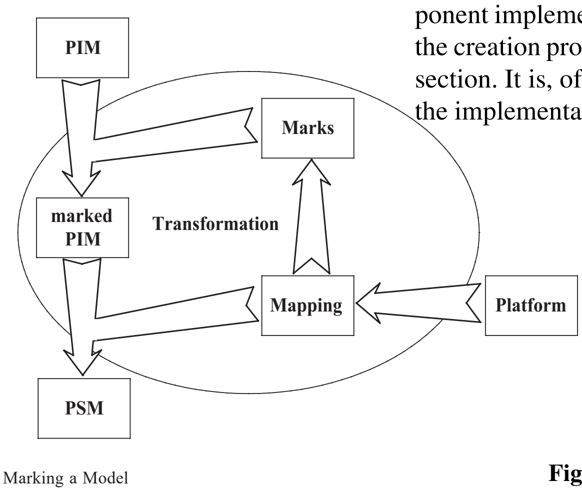


Figure 5.1: MDA Transformation Approach (Adapted from [OMG03])

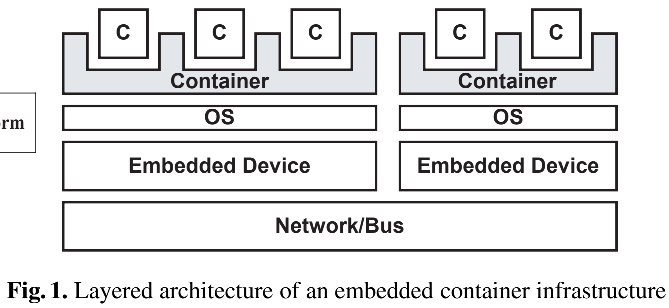


Figure 5.2: Layered Architecture of a Container Infrastructure (Adapted from [VSK05])

Component/Container infrastructures, like Common Object Request Broker Architecture (CORBA) [CORBA] and Enterprise JavaBeans (EJB) [EJB06], exist for the component-based development. These approaches isolate functionality from technical concerns by using components for functional concerns and container for technical concerns. They provide different implementations for different operating systems. However, their container infrastructures are customized for the corresponding component model. Furthermore, their container implementation is not adapted for the context requirements of specific components on specific platforms. The approaches provide general purpose containers which implementation is large and consumes many resources during runtime. Even embedded variants like CORBA for embedded [CORBAe] still have a too large memory footprint in many scenarios if they rely on the OMG specification. [VSK05]

Component-based development approaches, like SOFA HI [PWT+08; HPM+10], ProCom [BCC+08; CFMS10], or DEECo [MKM+16; BGH+13] rely on general-purpose containers or runtime environments that do not distinguish between different component types and their specific context requirements [BHK06]. Component-based deployment frameworks and standards, like Autosar [AUTOSAR14c], are designed for homogeneous systems, e.g., Autosar-based, and do not consider the analysis of the functional requirements of the application software and its communication behavior. Approaches that rely on formal approaches, like UPPAAL [UPP] that uses timed automata [BY04], provide code generation [AFM+04] but do

not support the deployment to distributed platforms. Table 5.3 in Section 5.8 summarizes related work and their capabilities.

MechatronicUML has a special focus on the platform-independent architecture and safety-critical real-time behavior on the functional application layer. Previous work on MechatronicUML of Burmester for automated implementation [BGS05; Bur06b] does not consider the communication aspect during the implementation and that each software component requires a different component context, like software libraries, middlewares, operating systems, hardware configurations, on different platforms [BHK06]. Additionally, previous work on MechatronicUML of Henkler for integrating legacy software components focuses on the functional integration on the basis of message exchange but does not consider the integration of technical concerns and the integration of the component context. Both existing concepts rely on a general purpose runtime environment that does not support distributed communication via exiting communication middlewares and different components context requirements. As a result, no platform-specific models, mapping models, or M2M/M2T transformations exists that consider the following requirements of CPSs.

Dann [Dan16] identifies and classifies a set of 50 requirements by conducting a systematic literature review. In the following, we describe our summarized top-level requirements. We develop the requirements in the context of the goals of this thesis. Therefore, we do not claim that the list is complete in general. A model-driven software construction approach for CPSs should...

- R.5.1 be able to specify a software architecture with hierarchical nesting and multiple communication styles.
- R.5.2 be able to specify the intra-system and the inter-system's behavior.
- R.5.3 be able to specify the hardware platform and the allocation.
- R.5.4 be able to verify and validate the system correctness.
- R.5.5 be able to specify the platform-specific coupling.

Furthermore, to construct the software correctly (effectively) and efficiently the code generation of a model-driven construction approach should ...

- R.5.6 be able to generate functional code for the software architecture and the behavior.
- R.5.7 be able to generate infrastructure code for the intra-system's and the inter-system's behavior that fulfills component specific Quality of Service (QoS) policies, like buffering and maximum delay
- R.5.8 be able to generate the customized technical code for the deployment on heterogeneous hardware platforms and software platforms.

In the following, we explain and justify our requirements for the software construction of CPSs. The requirements R.5.1, R.5.2, and R.5.4 are a summary of the requirements R.3.1, R.3.2, R.3.3, R.3.4, R.3.5, and R.3.6 from Chapter 3 because the software construction approach should be based on a well-defined design approach. Concerning requirement R.5.1, we require that the approach is able to model the software architecture to represent the system's structure. The software structure of a CPS is assembled of software components. According to Pop *et al.* [PHH+14], the software architecture should support horizontal and hierarchical composed models and different communication styles. Furthermore, it should enable to reuse software components and the integration of third-party algorithms. Without fulfilling the requirement R.5.1, a software construction approach cannot be used to develop efficiently. Concerning requirement R.5.2, we require the support of modeling the system's behavior. The system's behavior consists of: (1) the internal system's behavior; (2) the communication

behavior between system parts; and (3) the communication behavior between independently operated CPSs [*HSD15]. In Chapter 4, we show how important it is to model the hardware platform to construct a feasible allocation automatically. The underlying platform is also important during the compilation and the technical realization of the system's functionality. Therefore, we consider requirement R.5.3 as mandatory. Due to the safety-critical environment of CPSs, we require that the specification of the system's correctness is verified and validated (requirement R.5.4). Otherwise, if already the specification is incorrect a software construction approach cannot produce safe software artifacts. Especially, verified safety properties can ensure that a system acts fault-tolerant in the case of failures. Thereby, human harm is avoided in the case of failures and unexpected situations. Furthermore, the execution of the continuous components and its interaction with sensors, actuators, or controllers have to be supported.

Due to the safety-critical environment of CPSs, the software must have a high integrity. Therefore, the goal of a software construction approach must be to support correctness-by-construction [Cha06]. We target correctness in this thesis by using MechatronicUML as the modeling and verification approach [*DPP+16] and extending it with software construction capabilities by means of automatic implementation or respectively code generation. Such a construction must fulfill certain requirements to result in software artifacts that can be executed in distributed, connected, heterogeneous environments. Therefore, we deduce additional requirements for the automated software construction from models.

The requirement R.5.6 is necessary to represent the different functional aspects of the software component-based architecture and behavior. Furthermore, a CPS forms a system of systems [BS06]. Accordingly, the requirement R.5.7 enforces that infrastructure code that handles the intra-system and inter-system communication must be able to support the whole lifecycle of a communication link, i.e., discover, (re)connect, stop, and destroy [LCP+05] and must support the required functionality for the asynchronous message exchange in heterogeneous, distributed networks. This includes the encoding, transmission, and message buffering. Thereby, it is important to build up the communication on the basis of standardized network protocols and communication middlewares to reach a widespread acceptance and to meet legal restraints [PG14; Mah05]. Especially, the QoS policies that are used as assumptions during verification have to be supported [Dzi17]. Additionally, the infrastructure code has to handle failures, like communication loss. A software construction approach has to take care that the generated infrastructure is dedicated to the components requirements. As a result, the components can be deployed on different heterogeneous hardware and software platforms and are able to be executed in different execution contexts. Therefore, the requirement R.5.8 states that a component- and platform-specific infrastructure has to be created to realize customized technical concerns to execute software components on a specific target platform, e.g., for sending messages via different network protocols and communication middlewares.

The contribution of this chapter is an approach for the automated software construction on the basis of the ideas of the *MDA* [OMG03] approach, the *architecture-centric model-driven software development* approach [VSK05; SVC06], and the *model-driven middleware* [GBK+08] approach. Figure 5.3 shows an overview of the process that this approach supports. The process consists of the tasks **T_1:** Platform Independent Implementation, **T_2:** Platform-Specific Modeling, **T_3:** Platform-Specific Implementation, and **T_4:** Program Building. Section 5.2 describes this process in detail.

This chapter contributes an extension of our MechatronicUML component model [*DPP+16; Hei15] and an architecture-centric, container-based deployment architecture for CPSs that allows us to encapsulate platform-independent functional concerns and platform-specific technical concerns. The approach generates the implementation for applications automatically based on established implementation patterns using MechatronicUML models as input. Using our approach enables software engineers to handle the lifecycle of distributed,

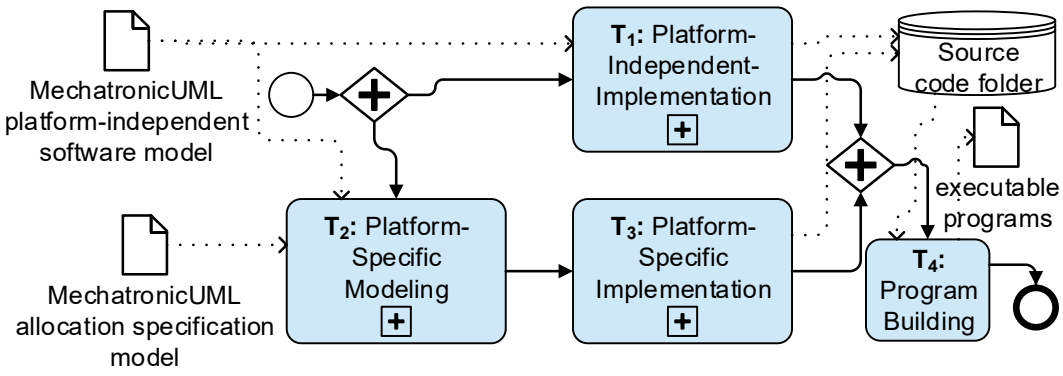


Figure 5.3: Overview of the Automated Model-Driven Software Construction Process

deployed components during runtime. Additionally, it enables them to handle and use different communication middlewares, protocols, and media. Software engineers can reuse and configure software components on different platforms and can integrate 3rd-party software artifacts. Furthermore, the approach automates building executables from the generated implementation by providing a makefile.

Our approach solves four challenges and fulfills all stated requirements. Firstly, in order to deploy a component-based application onto a distributed CPS, an implementation of functional and technical concerns is required [SVC06]. Therefore, we provide an automated implementation based on the MechatronicUML meta-model. Secondly, the implementation of the functional concerns of the software components must be reusable on different platforms and in different contexts [LW07]. We provide an implementation infrastructure that separates function concerns from technical concerns and enables that the functional implementation is reusable on different platforms and in different configurations. Furthermore, on the basis of the second challenge the third challenge is that the platform-independent implementation and the platform-specific implementation have to be separated to avoid that the functional code has to be re-validated/re-certified if the underlying platform changes. As a result, we split up the implementation approach into a platform-independent automated implementation of functional concerns and a platform-specific implementation of technical concerns Finally, the fourth challenge is that the intrasystem communication as well as the intersystem communication that fulfill a certain QoS must be supported [DBO+05; DPST11]. Therefore, we provide a generation of a communication middleware configuration that fulfills the communication QoS requirements. We evaluate the applicability and effectiveness of our approach by performing a case study. The case study uses a variant of the autonomous cooperating overtaking example (cf. Figure 1.2 in Section 1.3). Therefore, we deploy the generated code onto distributed ECUs on the basis of Raspberry Pis [Rasa]. The communication is realized using the OMG communication middleware standard Data Distribution Service (DDS) [DDS].

The remainder of the chapter is structured as follows: The next section extends our MechatronicUML component model [Hei15; *DPP+16]. Thereafter, Section 5.2 describes the development process for the automated software construction for MechatronicUML models that can be deployed to heterogeneous, distributed CPSs. Following, Section 5.3 introduces our concept for the automated software construction. We split up the section into five subsections that describe the aspects that are required to generate deployable executables from platform-independent models. In the following, Section 5.4 describes the implementation of the editors and the transformations for using the concept that this chapter describes. This tooling is integrated within the MechatronicUML Tool Suite [*DGB+14]. Next, Section 5.5

describes the evaluation of the automated software construction approach on the basis of a case study that realizes a variant of the cooperative overtaking scenario using the developed tooling. Section 5.6 concludes the conceptual and technical limitations of the approach. The chapter ends with a description of related work in Section 5.7 and a summary in Section 5.8.

The content of this chapter uses and extends approaches of supervised student work. In detail, concepts for the component model extensions and platform-specific modeling have been contributed by the Bachelor's Thesis of Rose [Ros14] and the final document [BCD+14] of the project group Cybertron. The Master's theses of Dann [Dan16] and Geismann [Gei15] and the final document [BCD+14] of the project group Cybertron contribute concepts for the automated software construction.

5.1 MechatronicUML Component Model Extensions

The development approach of MechatronicUML [*DPP+16; Hei15] is based on a component-based development methodology [Szy02] and uses *active components* [CSVC11] that run in a separate task. The *component model* has to define what is a component and how engineers can construct, compose/assemble, and deploy a component [LW07]. It should be possible to reason about the semantics of these operations [LW07]. Furthermore, components should be *reusable* [LW07].

The MechatronicUML component model has a precise definition [*DPP+16; Hei15]. Nevertheless, we identified three issues that are left open by the definition. Firstly, the existing software libraries, respectively *passive components* [CSVC11], cannot be used within components' behavior. Secondly, components cannot be reused within different platform contexts because they do not support different configurations. Thirdly, the semantics of communication between hybrid ports and continuous ports on distributed platforms is only defined informally via a description text [*DPP+16]. A formal representation, e.g., as a finite-state automaton is missing.

This section extends the MechatronicUML component model. Therefore, it defines in Section 5.1.1 how to integrate existing software libraries. Afterward, Section 5.1.2 and Section 5.1.3 define a parametrization extension for MechatronicUML. Finally, Section 5.1.4 defines the semantics of the communication between hybrid ports and continuous ports. The semantics definition enables a distributed deployment of assembled discrete and continuous components, can be analyzed via model checking, and eases the code generation.

5.1.1 Integration of Software Libraries

Differential and trigonometric functions are required to control a CPS [ÅK14]. They cannot be modeled efficiently using only statecharts. The same holds for algorithms like sorting or functions for storing data within a database. Modeling all these functionalities from scratch, e.g., by using our action language, would cause a huge effort without generating benefits within the formal analysis or simulation. Thus, *software reuse* [Kru92] is a key aspect of creating high-quality software. Using software libraries is the common approach to build software. Libraries bundle passive software components that are only executed if they are called. They enable to reuse existing legacy code efficiently. Therefore, we need an interface between active software components specified using MechatronicUML and existing software libraries.

We introduce *operations* for components to cope with this problem. A library is represented as an operation repository and may include several operations. An operation is a passive component that is only active if it is called by an action in a Real-Time Statechart (RTSC). We extend our component meta-model to support operations and operation repositories.

Figure 5.4 shows the resulting meta-model. An OperationRepository contains Operations and is used to organize them into logical units. An Operation represents the signature of an encapsulated function. It has references to Parameters that define its input parameters and a reference to a DataType that defines its return type. A Component has a reference to OperationRepositories that define the external dependency of a component. An Operation can be called within an entry action, exit action, or transition action of an RTSC by using an OperationCall. An OperationCall is a subclass of the MechatronicUML class Expression. This enables its integration within the MechatronicUML action language. Furthermore, an OperationCall has references to the called Operation and to the corresponding ParameterBindings. RTSCs are allowed to call only operations that are contained within an OperationRepository that the parent Component of the RTSC references.

The return value of an operation call conforms to a specific data type and can be assigned to a statechart variable. Such a variable may influence the behavior of a statechart, e.g., by being used within a transition guard. Thereby, a verified behavior may be changed. According to Heinzemann et al. a port statechart must always be a valid refinement of a role statechart for safety-critical systems [HBDS15]. Therefore, if a port statechart refines a role statechart it is only allowed to assign the return value of operation calls to variables that either do not influence the statechart behavior or to variables that where model checked using the MechatronicUML non-deterministic choice expression. Such expressions allow the specification of a set of possible values for variables of the role statechart during the model checking. The operation call must return a value within this set or the result must be transformed to a valid value using the MechatronicUML action language.

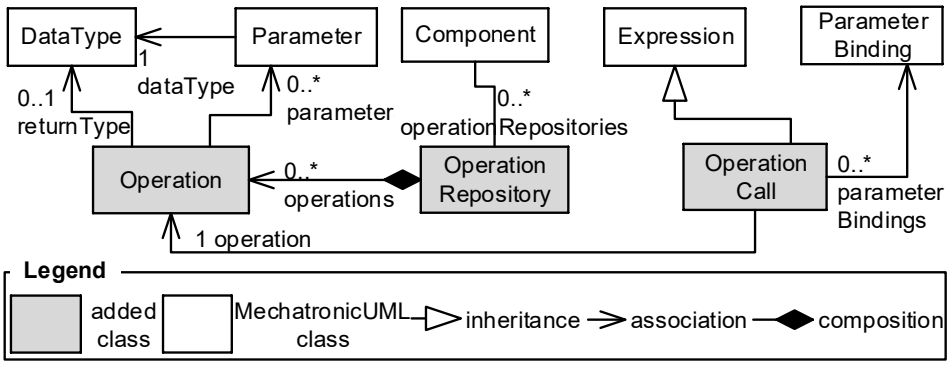


Figure 5.4: Meta-Model of Components with Operations

Figure 5.5 shows an example of using and defining operations on the basis of the section controller software component of our running example (cf. Section 1.3). This component registers for each street section which car is currently situated within the section. A software engineer provides a software library that is able to store this information within a database. The engineer provides two operations within the operation repository SectionControlDatabase that the left part of Figure 5.5 shows; BOOLEAN createDatabase() and updateCarSection(secId:INT32,carId:INT32). The behavior of the SectionControl component uses these operations to manage the registration of cars for a section control as Figure 5.5 shows in the RTSC. The statechart consists of the states init, createDatabase, and idle. The transition from the state idle to the state createDatabase calls the operation createDatabase and assigns the return value to the statechart variable status. The statechart activates the state idle if the creation was successful and activates the state init if the creation fails. A car can register itself by sending the message enteredSection(secId:INT32,carId:INT32) to the component sectionController. The statechart consumes the message enteredSection within the state idle by its self-transition

and calls the operation updateCarSection. Furthermore, it passes the message parameters secId and carId as arguments to the operation updateCarSection.

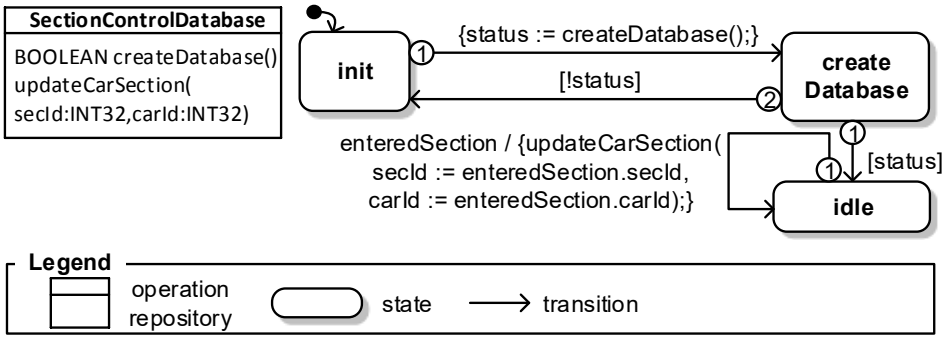


Figure 5.5: Defining and Using Operations within the Component SectionController

5.1.2 Reuse and Configuration of Components via Parametrization

Component instances require specific information additional to the general information that their component type(s) provide [Tic09]. An analogy in object-orient programming is the relation of a class to a concrete object. A class has a constructor that enables to instantiate objects using different values for certain member variables. The MechatronicUML component model [*DPP+16; Hei15] misses them currently although Tichy [Tic09] proposes to use attributes for components within a previous MechatronicUML branch for self-optimized systems. In contrast to Tichy, we require not only attributes or respectively parameters that contain numerical values but also parameters for timing values and operation calls. Different instances of a continuous component access a device via different API calls and consider different timing constraints.

Figure 5.6 shows the resulting meta-model. A Component has ComponentParameters. The class ComponentParameter is an abstract class. We provide the subclasses ValueParameter, TimeParameter, and CallParameter. We introduce different kinds of parameters to prevent misuse within an RTSC and to enable a better static analysis. The class ValueParameter can be used within statecharts to assign a value to a parameter or within a comparison expression. The class TimeParameter can be used within comparison expressions of clock constrains and time invariants of statecharts. The class CallParameter can be used, like ValueParameter, to assign a value to a parameter or within a comparison expression. In contrast to a ValueParameter a CallParameter can be referenced by a CallExpression of the MechatronicUML action language.

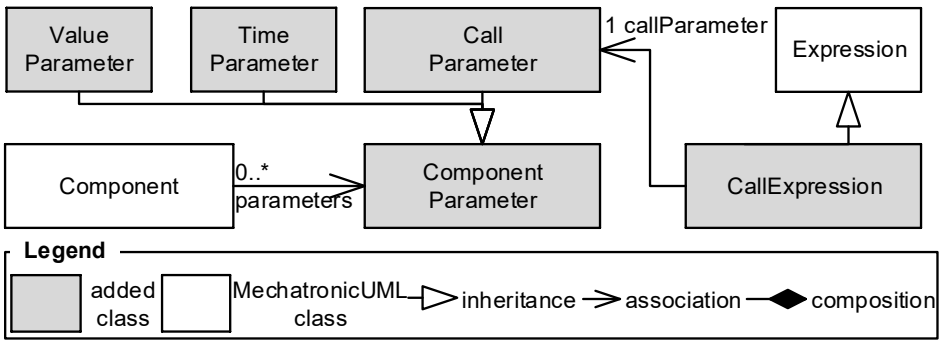


Figure 5.6: Meta-Model of Components with Parameters

Figure 5.7 shows the component ParameterizedComponent and its RTSC as an example. The component has the valueParameter DOUBLE initValue, the timeParameter timeConstraint, and the callParameter accessFunction(). Furthermore, the RTSC has the variable DOUBLE apiValue and the clock localClock. The statechart consists of the two states Init and ExcecuteAccessFunction. When the statechart switches from state Init to state ExcecuteAccessFunction the exit action apiValue:=initValue is executed. Thereby, the variable apiValue gets the value of the valueParameter. A component instance has to bind a concrete value to the valueParameter. Afterward, when the state ExcecuteAccessFunction is activated, the entry action apiValue:=accessFunction() is executed. This action overwrites the value of the variable apiValue with the return value of the callParameter. A component instance has to bind an APICallExpression (cf. Section 5.3.3.1) to the callParameter. Section 5.3.3 describes APICallExpression in more detail because this requires platform-specific information. We provide another example of using component parameters in Section 5.1.4 when we define the semantics of continuous and hybrid ports.

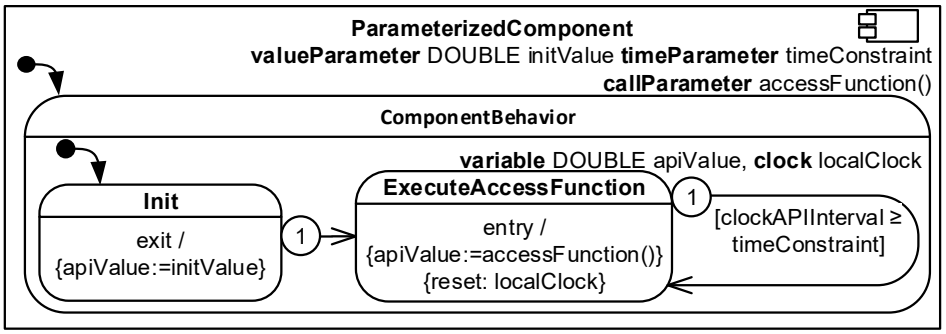


Figure 5.7: Example of

5.1.3 Parameter Binding for Component Instances

Concrete bindings for parameters have to be created when a component is initialized in order to adapt component types to concrete component instances. A software component's implementation can access these parameter bindings during its execution and adapts its behavior correspondingly. A component instance has to bind concrete functions or time values to its parameters. Therefore, a component instance needs a property that contains these bindings. We introduce component instance parameter bindings to cope with this problem.

Figure 5.8 shows the resulting meta-model. A ComponentInstance has a reference with cardinality 0..* to the abstract class ComponentInstanceParameterBinding that references a ComponentParameter. A ComponentInstanceParameterBinding is either a ValueParameterBinding that binds the ValueParameter to an Expression, a TimeParameterBinding that binds the TimeParameter to a TimeValue, or a CallParameterBinding that binds a CallParameter to an APICallExpression.

An example binding for the component ParameterizedComponent is the binding of the valueParameter to the ValueExpression "0", the binding of the timeParameter to the TimeValue "30 ms", and the CallParameter to the APICallExpression ecrobot_get_sonar_sensor(port_id:=3);. Section 5.3.3 describes APICallExpression in more detail because this requires platform-specific information. We provide another example of binding component parameters in Section 5.3.4.3 when we define the semantics of continuous and hybrid ports.

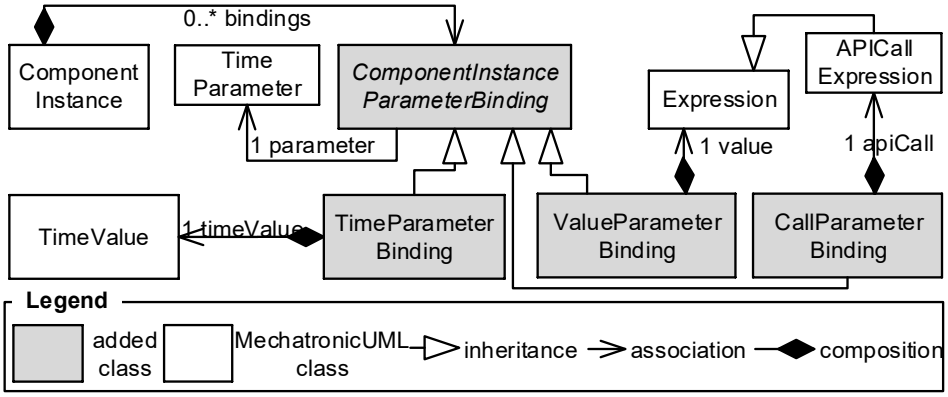


Figure 5.8: Meta-Model of Discrete Port Instance with Parameter Binding

5.1.4 Semantics of the Communication between Hybrid Port and Continuous Ports

Signals of continuous port instances that origin from device accesses may affect the behavior of discrete component instances that receive them via hybrid port instances. We need to define the semantics for accessing devices and for the transmission from continuous port instances to hybrid port instances to enable a reachability analysis or formal verification [SHG16] and to enable a distributed deployment. Additionally, we have to define the semantics in the case of a device failure or a network failure. Furthermore, the allocation of continuous component instances and discrete component instances should be independent. That means that if a discrete component instance accesses a signal from a continuous component instance it should be possible that both components are allocated to different ECUs, as long as a network connection between both ECUs exists.

We define the semantics of continuous ports, hybrid ports, and its connector by defining RTSCs. The RTSC of the continuous port accesses a device via a device Application Programming Interface (API) and communicates with an RTSC of the hybrid port via messages. The semantics of the communication between a continuous port and a hybrid port implements the Periodic Transmission Real-Time Coordination pattern [DBHT12]. We use this definition during the platform-specific implementation (cf. Section 5.3.4) for generating corresponding platform-specific code. In the following, we show the semantics of an out-port and of an in-port using the distance sensor and the motor from our running example (cf. Section 1.3).

Figure 5.9 shows the RTSC that represents the semantics of a continuous component that has a continuous out-port, like the continuous component Distance. The RTSC of the parameterized component consists of two regions. The region PeriodicTransmission_send sends the variable apiValue periodically via the message message(DOUBLE). Therefore, the state PeriodicSending has a self-transition that only fires if the value of the clock clockSending has at least the value of the period of the connected hybrid port of the continuous port. Furthermore, if the value of the variable apiValue has the value of a specific fail value, e.g., -1, the statechart switches to the state Failure and sends the message failure to the connected hybrid port. If the variable apiValue is back within an allowed state it sends again the normal message containing the value of the variable apiValue. The region API_Call periodically assigns the return value of the accessFunction to the RTSC variable apiValue. Therefore, the state ExecuteAccessFunction has a self-transition that only fires if the value of clockAPIInterval has at least the value of the variable timeInterval and the entry action that executes the corresponding behavior. The variable timeInterval may have the value of a timing constraint that restricts how often a specific sensor can be called without returning faulty values.

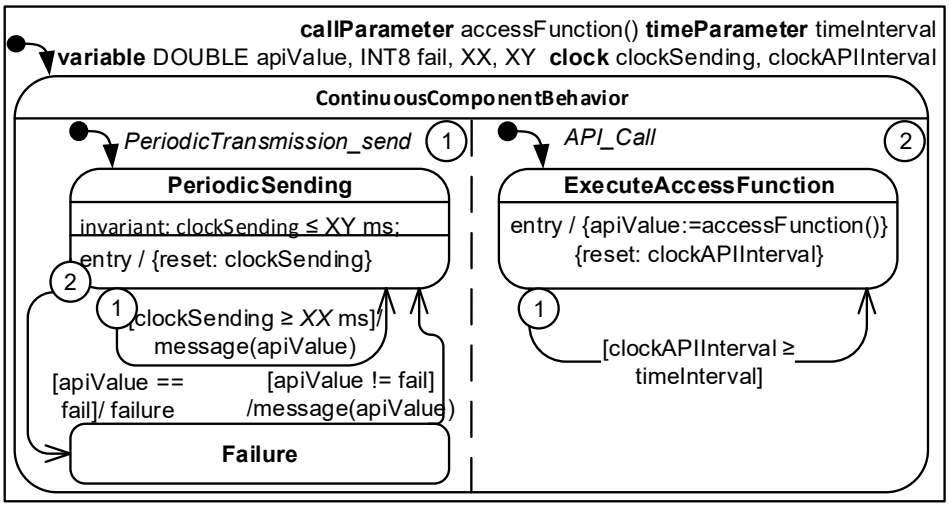


Figure 5.9: Semantics Definition for the Continuous Component Behavior Using a Continuous Out-Port

Figure 5.10 shows the parameterized RTSC that represents the semantics of a continuous component that has a continuous in-port, like the continuous component Motor. The parameterized RTSC of the continuous component consists of two regions. The region PeriodicTransmission_receive receives the variable apiValue from the message message(DOUBLE) and assigns the value to the local variable apiValue. Both regions execute their behavior independently of each other and are not synchronized via a synchronization channel because it should be possible that the access function is called independently from the communication protocol that requires a certain timing interval. Thereby, software engineers can realize an oversampling or undersampling for accessing the device. Both techniques are used in data analysis [Cha10]. Therefore, the state PeriodicReceiving has a self-transition that fires if a message is available within the incoming buffer and executes its assigned action. The region API_Call periodically calls the accessFunction(apiValue) and binds the local variable apiValue to the input parameter of this function by executing the entry action of the state ExecuteAccessFunction. Currently, we specify only a lower time bound for accessing devices, which prevents device errors that could occur if a device is accessed to often. In the future, it may be useful to add a time invariant that guarantees an upper time bound for accessing a device, which may prevent that a device switches to a sleep mode if it is not accessed for a certain period. Currently, we omit this function because we had no use case within our examined example cases. The self-transitions of this state activates the state periodically. Furthermore, the entry action binds the return value of the accessFunction to the variable named ret. The left region switches to the state Failure if the value of the variable named ret is equal to the value of the variable named fail. Furthermore, this failure state is activated if for a certain amount of time no message is received. If the state named Failure is activated the incoming transitions also send a failure message to the communication partner. The state named Failure is left if a new message is received.

The semantics of hybrid in-ports and hybrid out-ports that are the counterparts of the continuous ports are shown and described in Appendix D.5.

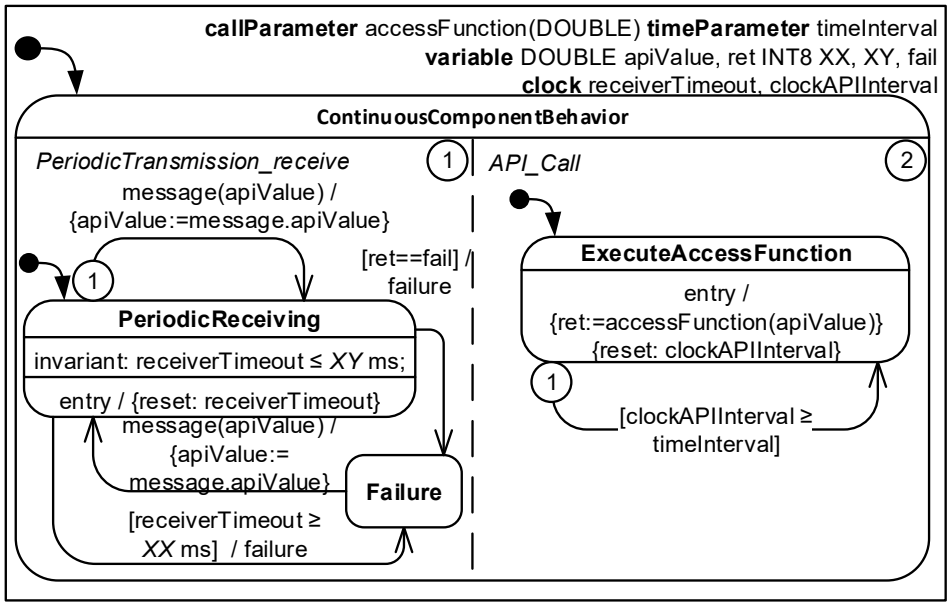


Figure 5.10: Semantics Definition for the Continuous Component Behavior Using a Continuous In-Port

5.2 Process for the Software Construction

The construction of software for distributed, cooperative CPSs is very complex and requires many tasks that software engineers need to handle. Therefore, we introduce a systematic process for the automated software construction on the basis of the MDA [OMG03] process and the architecture-centric, model-driven development process of Stahl, Völter, and Czarnecki [SVC06]. The process describes the required tasks from a PIM model to executables that can be deployed to distributed ECUs. This section introduces the process, which is a detailed version of the process shown in Figure 5.3.

Figure 5.11 shows the process using the Business Process Model and Notation (BPMN) [BPMN]. Transformations refine a PIM into a platform-independent implementation. A PSM has to be created partly by software engineers and party automatically on the basis of a MechatronicUML allocation model. The PSM is refined to a platform-specific implementation. The final implementation consists of: (1) generated schematic and repetitive code; (2) a generic platform, middleware, and library code; (3) individual user code [SVC06]. The code of all implementation parts are built to a platform-specific executable. A software engineer who is familiar with the application and with the underlying embedded software platform performs the process. We automate several tasks of the process to improve the efficiency of the software engineer and to improve the software quality. Our goal is to separate platform-independent code from platform-specific code to improve the portability. Furthermore, our goal is to support the reusability of high-quality software artifacts by integrating and using existing platforms, libraries, and individual user code. Section 5.3 describes the concept and software architecture in more detail.

The source of the process is a MechatronicUML platform-independent software model. The PIM defines the *system's functionality*. We use a MechatronicUML model of software component types for describing the software interfaces of the system's functionality. RTSCs define the platform-independent behavior of the system's functionality of each software component type. In Task $T_{1.1}$, a M2T transformation (C code generator) generates a platform-independent implementation automatically. The implementation consists of header files and

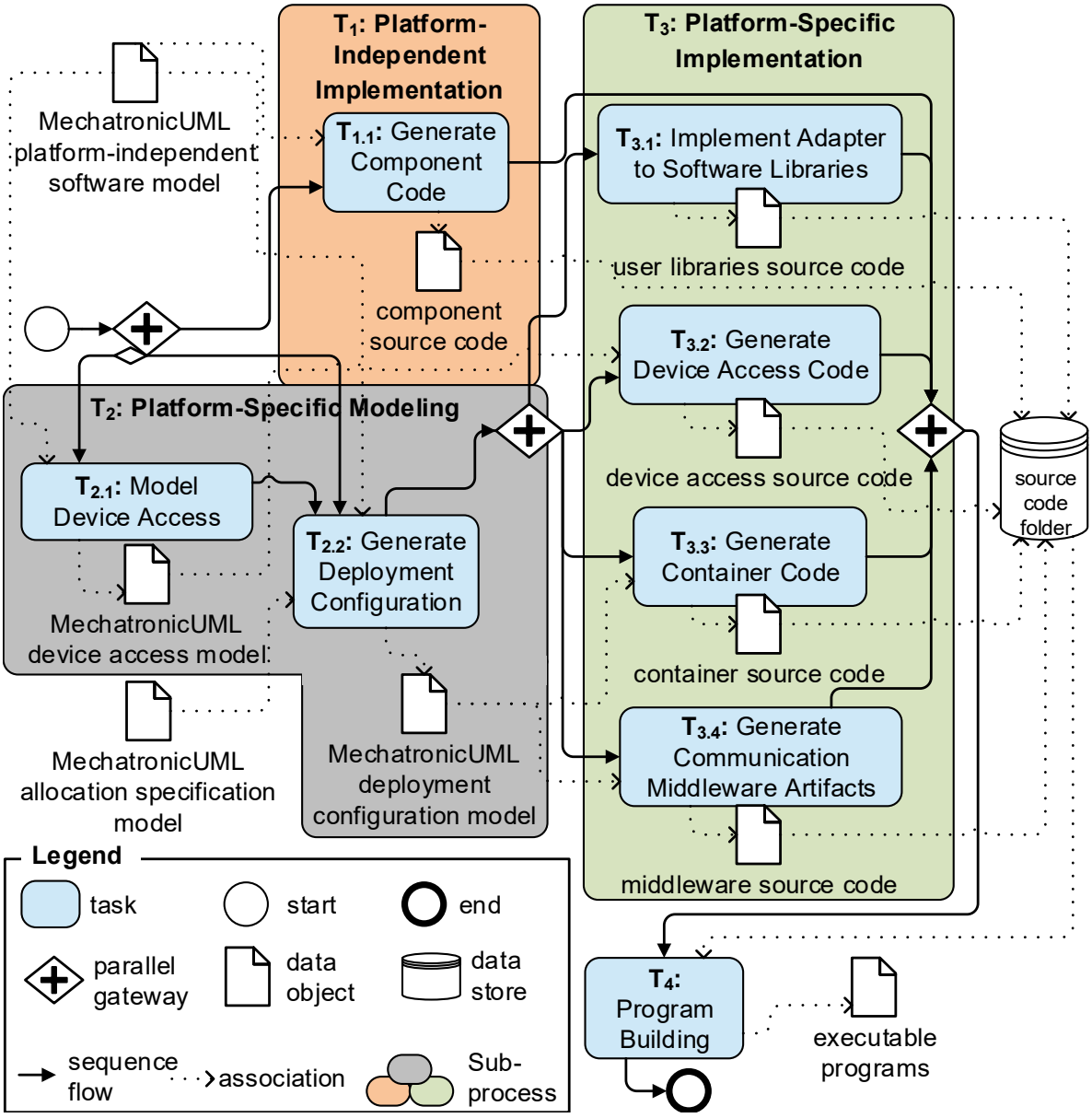


Figure 5.11: Automated Model-Driven Software Construction Process

source code files for each software component (cf. Section 5.3.2). The resulting source code can be built and compiled to object files but cannot be executed standalone. At least, it has to be linked to a platform-specific implementation to access the system time. The system time is required to calculate correct values for clocks that influence the real-time behavior. Furthermore, if a component type has interfaces then it has to be linked to further distributed software components or (3rd-party) software artifacts. Software engineers model platform-specific software artifacts in the optional Task $T_{2.1}$ (cf. Section 5.3.3). First of all, they can specify a model of the device API of the underlying software platform (cf. Section 5.3.3.1). Such a model contains information how to access devices programmatically. We define a small domain-specific language that is integrated into MechatronicUML. Additionally, software engineers specify platform-specific mappings of continuous port instances to platform-specific device API functions for accessing physical devices or feedback controllers using the API model. We use the resulting device access model to generate the platform-specific device access code (cf. Section 5.3.4.2). We generate stubs only for platform-specific device access code if engineers do not perform the optional Task $T_{2.1}$. In Task $T_{2.2}$ software engineers generate a deployment configuration (cf. Section 5.3.3.2). The source of Task $T_{2.2}$ is the platform-specific MechatronicUML device access model and a MechatronicUML allocation specification model. The output is the MechatronicUML deployment configuration model. We use this model to instrument the code generator of the container code and the middleware artifacts of a platform-specific implementation (cf. Section 5.3.4). These platform-specific software artifacts are implemented and generated within the parallel Tasks $T_{3.1}$–Task $T_{3.4}$. In Task $T_{3.1}$ software engineers implement platform-specific adapters to existing libraries manually (cf. Section 5.3.4.1). In Task $T_{3.2}$ the code generator generates a platform-specific implementation for accessing devices from the MechatronicUML device access model (cf. Section 5.3.4.2). In Task $T_{3.3}$ the code generator generates the platform-specific container code for each component type using the MechatronicUML deployment configuration model (cf. Section 5.3.4.3). In contrast to the functional parts, which are implemented within the platform-independent component code, the platform-specific container code realizes the technical parts. The container code enables to reuse the functional component code for different platforms effectively [VSW02]. The container code realizes an adapter between the component code and the underlying software and hardware platform. In Task $T_{3.4}$ a code generator generates the application- and platform-specific source code for the required specific communication middlewares using the MechatronicUML deployment configuration model (cf. Section 5.3.4.4). Finally, software engineers build the executable programs from the different source files in Task T_{4} by compiling the source files and linking the created objects files and interfaces descriptions (header files) (cf. Section 5.3.5).

5.3 Concepts for the Software Constructions

Executables are required to be deployed onto and run by ECUs. Executables are platform-specific and built from code. We generate the code to a large extent automatically in our approach. This section describes how to generate and build platform-specific code for distributed CPS starting with a verified and validated PIM following the automated model-driven software construction process of the last section.

Section 5.3.1 introduces the architecture-centric, component-container-based generation infrastructure for generating software from models. Next, Section 5.3.2 describes how to generate a platform-independent implementation of models specified using MechatronicUML. Following, Section 5.3.3 describes how to model platform-specific properties and Section 5.3.4 describes how to generate a platform-specific implementation of functional and technical

concerns. Finally, Section 5.3.5 shows how to automate the build process by generating build instructions.

5.3.1 Architecture-Centric, Component-Container-based Generation Infrastructure

The architecture-centric, component-container-based generation infrastructure separates the functional parts, like implementing a coordination protocol of a component, from the technical parts, like opening a network socket and encoding the messages as a bit stream. The functional parts are implemented within components and the technical parts are realized within containers. The generated software artifacts are ready for the distributed deployment after being built to executables.

Figure 5.12 shows the infrastructure of a generated executable from a model that is deployed on an ECU with an operating system. The executable consists of different software artifacts. Some of the artifacts are mandatory and some of the artifacts are optional.

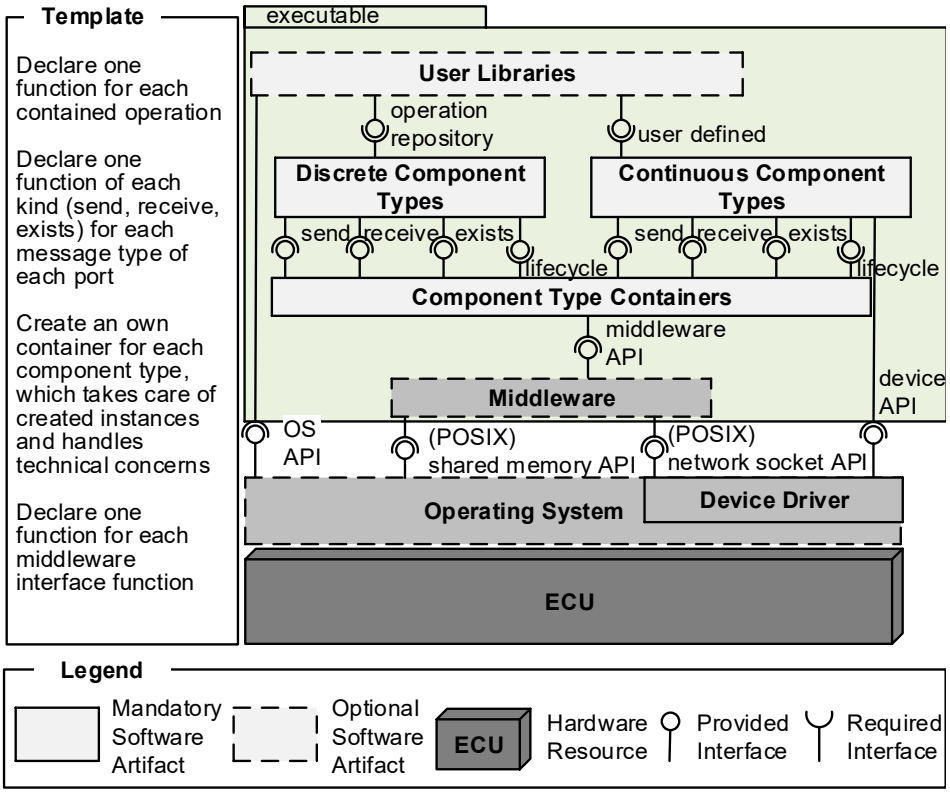


Figure 5.12: Infrastructure for Component-Container-based Executables in the Context of an ECU

The main generated artifacts are the implementation of the component types. They implement only functional concerns of a cooperative CPS. Components may depend on user-defined libraries.

Libraries bundle passive software components that are only executed if they are called. We generate one function for each contained operation of each operation repository that a component depends upon. In contrast to the mostly platform-specific implementation of an operation that may access the underlying operating system via the OS API, the signature of an operation is platform-independent. Continuous components whose implementation may

be unknown may also depend on libraries. These dependencies are user defined and software engineers have to handle them manually.

Functional concerns of components must be completed with technical concerns to fulfill their assigned task [VSW02]. Therefore, we provide for each component type that is allocated to an ECU an own component container. The component container takes care of the lifecycle of all instances of a component type and provides a stable interface for fulfilling technical concerns [VSK05; BHS07].

Unfortunately, it is not possible to host all components in a generic container without getting the overhead of the memory footprint required for using the generic container because each component has a unique interface [VSW02]. In addition, existing component infrastructures, like CORBA [CORBA] or Enterprise JavaBeans [EJB06], are only usable for a certain technology-specific component model. Therefore, they could not be used for MechatronicUML components directly. Thus, we generate for each component type a specific container that is suitable for MechatronicUML.

In our case, the stable interface between a component and its container consists of two parts. The first part provides functions for sending and receiving messages. Therefore, for each message type of a port's message interface corresponding send, receive, and exist functions are generated. Continuous components/ports and hybrid ports are transformed to discrete components/ports before code is generated according to the definitions that Section 5.1.4 introduces. Therefore, these components/ports are not handled separately during the code generation. The second part of the interface between a component and its container provides lifecycle functions for initializing, destroying, activating, and deactivating component instances. Appendix D.4 describes the used implementation pattern in more detail. Continuous components have an additional interface for accessing devices. This interface is connected to the device API of the underlying operating system. It is connected to the device API of the device drivers/libraries directly if no operating system is provided or if the provided operating system does not include a corresponding API.

A container can handle all technical concerns by itself. However, this requires to provide manually a lot of technical functionalities, like message-transfer, by itself. Many operating systems only provide basic communication protocols, like Transmission Control Protocol (TCP) [TCP] or User Datagram Protocol (UDP) [UDP]. Nevertheless, on different platforms and in different domains specific runtime environments exist, which provide a lot of functionality. They provide a kind of glue code between applications and operating systems and are typically called *middleware* [Emm00].

Common middlewares for distributed systems are CORBA-based [CORBA] middlewares and DDS-based middlewares [DDS]. Furthermore, for automotive system AUTOSAR [AUTOSAR14c] defines a middleware layer. OPC UA [MLD09] defines a middleware layer for production systems. In robotics Robot Operating System (ROS) [QCG+09] is a common middleware layer. [Dan15] identifies three classes of middlewares in the surveyed publications: (1) data-centric [DDS]; (2) object-oriented [CORBA; CORBAe]; (3) message-oriented [Mah05]. All classes of middleware have advantages and disadvantages for being used as an underlying platform. The selection of "the right" middleware depends on a lot of factors, like the domain, maturity of the underlying implementation, license restrictions, and QoS requirements. As a result, it is not possible to define one specific, best-fitting middleware for implementing the technical concerns of MechatronicUML.

Furthermore, it is not possible to specify one common universally valid container-middleware interface. Therefore, we generate specific *connectors* [BP02] that abstract from the nature of the interaction between components. At design-time, a communication between components is modeled as an assembly link, while at runtime implementing the communication comprises the execution of several functions on different layers. We generate the functions for accessing

the middleware API from the container, customized for the underlying middleware to realize the connectors. These interfaces are quite complex and provide a lot of different functions. Therefore, we generate customized glue code for mapping the one-to-one communication concept of MechatronicUML to the middleware communication paradigm and interfaces on the basis of our MechatronicUML deployment configuration model, e.g., publish-subscribe of DDS [DDS].

Middlewares use the functionality of an underlying operating system to fulfill technical concerns. They may use the shared memory API of Portable Operating System Interface (POSIX) [ISO9945] for implementing a local communication on a single ECU or the network socket API [ISO9945] of POSIX for implementing remote communication via a bus or point-to-point network.

The operating system controls the scheduling of the executables. It may control several tasks if an executable is split up into several executables or uses multithreading. It uses a real-time scheduling strategy, like Earliest Deadline First (EDF) [Kop11], if it is a Real-Time Operating System (RTOS). The executable controls its own execution if it directly runs on the firmware of an embedded device without using an operating system.

5.3.2 Platform-Independent Implementation

We need an implementation of the MechatronicUML component types, including its behavior, to realize the functional concerns of CPSs. This section describes how to generate platform-independent code from models specified using MechatronicUML. We choose ANSI C99 [ISO9899; Sta01] as the target language. The most embedded platforms provide build tools for the ANSI C99 language. The generated component code is decoupled from any target platform and is based on the component type definition of MechatronicUML. It is reusable on any target platform that provides corresponding build tools and provides the required container infrastructure (cf. Section 5.3.1).

During the generation, we do not distinguish between discrete and continuous components because we generate for continuous components the implementation as if they contain the RTSCs that are described in Section 5.1.4. We also add the behavior of hybrid ports as if they contain the RTSC that is described in Appendix D.5. Manually implemented controller behavior or code provided by tools like MATLAB Simulink or Modelica can be integrated via operations (cf. Section 5.1.1 and Section 5.3.4.1) or bindings to call parameter of components (cf. Section 5.1.3 and Section 5.3.4.3).

We use the following patterns for our implementation: the *Component Context Pattern* [BHS07], the *Handle Pattern* [BHS07], the *Builder Pattern* [BHS07], and the *Lifecycle Callback Pattern* [VSW02] for generating component code independently of a target platform. A detailed description of the patterns in the context of our platform-independent component code generation is described in Appendix D.

This section is split into three subsections. Section 5.3.2.1 discusses and defines the term platform-independent and platform-specific for C code because we require this differentiation for this chapter. Afterward, Sections 5.3.2.2–5.3.2.4 describe in more detail how the code for the platform-independent component implementation is structured and describe the dependencies of the implementation. Following, Section 5.3.2.5 describes how the behavior of a platform-independent component type is implemented.

5.3.2.1 C Code – Platform-Independent vs. Platform-Specific

The definition which C code is *platform-independent* and which is *platform-specific* is fuzzy because the C standard [ISO9899] does not define the mechanism by which C programs are

transformed for the use by specific platforms. When a C program handles data, different platforms store them in a different platform-specific memory layouts [NGC08]. Therefore, we use the term platform-independent code and the term platform-specific code relative to each other. The platform-specific code depends more on a specific platform than platform-independent code. In this thesis, we use the term platform-independent code as long as the code does not need libraries (header files) that have to access functions that depend on the underlying operating system or hardware devices. To be more precise, we use the term platform-specific for functions that access the file system, the user input, the user output, the system time, a network device, a sensor, or an actuator. These functions are essential for distributed CPSs because they realize the connection to the technical system, the real world, and to each other.

5.3.2.2 Component Definition

The main artifact that we generate for each component type is an own header file. A header file may contain definitions of custom types, variables, constants, macros, and function prototypes. It is used to separate a program into submodules and can be shared by several source files.

First of all, it defines the component type in the form of a C-*struct*. In the programming language ANSI C99 a struct is a declaration of a complex data type [ISO9899; Sta01]. Listing 5.1 shows the generated struct for the component type Overtakee. The component struct defines the ID variable as an identifier for each component instance. Furthermore, it defines a pointer to a concrete struct that represents the statechart and variables for all ports.

```
1 struct OvertakeeComponent {
  uint8_T ID; // unique identifier
  /**< The OvertakeeOvertakeeBehavStateChart of the Component Overtakee **/
  OvertakeeOvertakeeBehavStateChart* stateChart;
5 Port toSectionCtrlPort; // A  Component's Port: toSectionCtrl
  };
```

Listing 5.1: Basic Struct for Overtakee Component Type

Additionally, the header file defines the struct that defines the RTSC. Listing 5.2 shows the definition of the overtakee RTSC (cf. Figure 2.2 in Section 2). The Lines 1–6 define the enumeration type OvertakeeStates that encodes all states of the component RTSC. The struct of the statechart has a pointer to the data structure that defines the component type (Line 11). Furthermore, it defines for each statechart and substatechart a variable that stores that current active state. Each variable is of the enumeration type OvertakeeStates. The struct defines also variables for all clocks that the statechart defines, e.g., timeout (Line 17) and all variables that the statechart defines, e.g., refSpeed, error, and blockSpeed (Lines 19–21).

5.3.2.3 Component Interface for the Interaction with Its Container

The component struct provides a variable (pointer) for each CallParameter and TimeParameter if the component type has parameters. The variable pointers have to be bound to concrete variables and the function pointers have to be bound to concrete functions by the container when a component is instantiated (cf. Section 5.3.4.3). Listing 5.3 shows an example of the continuous distance component and its distance port. The distance continuous component requires for its port the time parameter timeInterval and the call parameters accessFunction, initFunction, and termFunction. We declare the time parameter as a variable pointer and the call parameters as function pointers.

```
typedef enum {
INACTIVECOMMUNICATOR,
ACTIVECOMMUNICATOR,
ACTIVECOMMUNICATOR_INIT,
ACTIVECOMMUNICATOR_REQUESTED,
ACTIVECOMMUNICATOR_OVERTAKING
}OvertakeeStates;

struct OvertakeeOvertakeeBehavStateChart {
OvertakeeComponent * parentComponent;
/**Active States**/
OvertakeeStates currentStateOfOvertakeeStatechart;
OvertakeeStates currentStateOfOvertakeeActiveCommunicator;
/** Helper Variables**/
bool_T OvertakeeStatechart_isExecutable;
bool_T OvertakeeActiveCommunicator_isExecutable;
/** Clock Variables**/
Clock timeout;
/** Statechart Variables**/
double_T refSpeed;
bool_T error;
double_T blockSpeed;
}
```

Listing 5.2: Forward Declaration of a Real-Time Statechart

```
struct DistanceComponent { ...
/** Time Parameter Variable Pointers**/
double * timeInterval;
/** Call Parameter Function Pointers**/
double (*accessFunction)(void);
void (*initFunction)(void);
void (*termFunction)(void);
}
```

Listing 5.3: Extended Declaration of the Distance Component

Furthermore, the component's header file defines forward declarations of the component container communication functions, which the corresponding container implements. These are the functions for sending and receiving messages. Listing 5.4 shows the declaration of the send function for sending the accept message (cf. Figure 2.2 in Section 2) from the overtakee to the overtaker. The function has the input parameters Port* p, and Accept_Message* msg that point to a port instance and a message instance during runtime. The pointer to the port instance is used to indicate via which port a message should be sent or be received. Additionally, it shows the declaration of the receive and the exist function for checking and dispatching the request message from the overtaker.

```
void Overtakee_toOvertaker_send_Accept_Message(Port* p,
    Accept_Message* msg);
bool_T Overtakee_fromOvertaker_recv_Request_Message(Port* p,
    Request_Message* msg);
bool_T Overtakee_fromOvertaker_exists_Request_Message(Port* p);
```

Listing 5.4: Forward Declarations of Communication Functions

Additionally, the component's header file defines forward declarations of lifecycle callback functions, which the corresponding container calls for handling the lifecycle of a component instance. Listing 5.5 shows the declaration of the four lifecycle functions: initialize, destroy, activation, and deactivation. They have a pointer to an instance of the component as their input parameter.

```
1 void Overtakee_initialize(OvertakeeComponent* component);
  void Overtakee_destroy(OvertakeeComponent* component);
  void Overtakee_activation(OvertakeeComponent* component);
  void Overtakee_deactivation(OvertakeeComponent* component);
```

Listing 5.5: Forward Declaration of Overtakee Lifecycle Callback Functions

At last, the component's header file defines the stateless functions for creating, initializing, and destructing an RTSC and for processing the logic of an RTSC. Listing 5.6 shows these declarations. The Lines 2–3 define the create function that gets as an input parameter a reference to the parent component and returns a pointer to the created statechart structure. The Lines 4–5 define the destroy function that gets a statechart structure as input and frees the allocated memory. The Lines 7–8 define the initialization function that sets the initial values of the clocks, variables, and states for the given statechart. The Lines 9–10 define the processing function that examines the current state of the given statechart, receives messages, activates transitions, fires active transitions, sets the new state configuration, executes state and transitions actions, and sends messages. The Lines 12–16 define helper functions for processing the statechart.

```
1  /**Lifecycle Functions**/
   OvertakeeOvertakeeBehavStateChart* OvertakeeOvertakeeBehavStateChart
       _create(OvertakeeComponent* parentComponent);
   void OvertakeeOvertakeeBehavStateChart
5      _destroy(OvertakeeOvertakeeBehavStateChart* rtsc);
   /**Initialization and Processing Functions**/
   void OvertakeeOvertakeeBehavStateChart
       _initialize(OvertakeeOvertakeeBehavStateChart* rtsc);
   void OvertakeeComponent
10     _processStep(OvertakeeOvertakeeBehavStateChart* rtsc)
   /**Helper Functions**/
   bool_T OvertakeeOvertakeeBehavStateChar
       t_isInState(OvertakeeOvertakeeBehavStateChart* rtsc,
        OvertakeeStates state);
15 void OvertakeeToSectionCtrlOvertakeePStateChart
       _exit(OvertakeeOvertakeeBehavStateChart* rtsc);
   }
```

Listing 5.6: Forward Declaration of a Real-Time Statechart Lifecycle and Processing Functions

5.3.2.4 Component Dependencies

The component's header file depends on further generated software artifacts or MechatronicUML library software artifacts. Figure 5.13 shows the dependency graph of the generated C-header files for the Overtakee component type. Each component's header file depends on three standard files, which are the same for each project and are copied during

the generation process once. Furthermore, it depends on two project-specific custom files that are generated automatically.

The file lib/StandardTypes.h defines the standard data types for MechatronicUML. These types are mapped to the standard C data types that the platform provides. Additionally, the file lib/port.h defines a port as an own type (cf. Listing 5.7). It has a port status that represents its current connection status (cf. Figure D.9 in Appendix D.1) and a port handle (cf. Figure D.10 in Appendix D.2).

```
1 typedef enum {
    INACTIVE, CONNECTED, CONNECTED, CONNECTIONLOST
  } PortStatus;
  typedef struct PortHandle* PortHandlePtr;
5 typedef struct Port {
    PortStatus status;
    PortHandlePtr portHandle;
  } Port;
```

Listing 5.7: Declaration of a port as a type.

The manually implemented platform-specific file lib/clock.h have to define a global clock, which accesses the system time and provides the reference value for updating all clock values during the runtime. The file has to provide the functions `Clock_getTime(Clock* aclock)`, `Clock_reset(Clock* aclock)` to the component type implementation for accessing a clock value and for resetting a clock.

The generated file types/CustomTypes.h defines the *structs* for the MechatronicUML's project specific custom data types, like a custom array. The generated file types/MessageTypes.h defines the structs for each message type and its parameters.

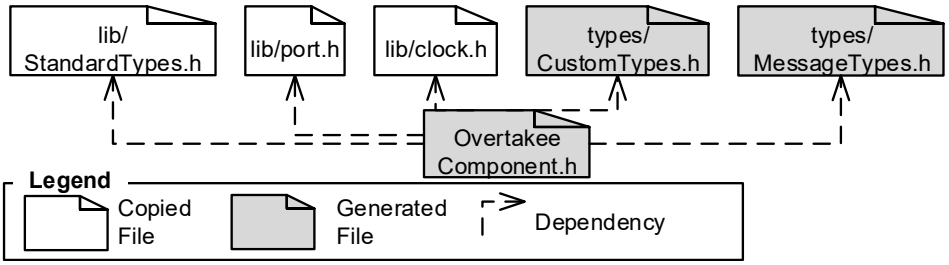


Figure 5.13: Dependency Graph of Generated Interface Files for the Overtakee Component

5.3.2.5 Component's Behavior Implementation

Implementing the component's behavior of MechatronicUML requires to implement the behavior of RTSCs and to integrate this behavior with the interfaces of the underlying component containers that provide the functions for realizing technical concerns (cf. Figure 5.12 in Section 5.3.1). Implementing statecharts in C/C++ is a well-studied field of research. A practical guide for implementing UML in C/C++ is written by Samek [Sam09]. Niaz and Tanaka already published a solution to generate C code automatic from UML statecharts in 2003 [NT03]. Several patterns can be used to implement hierarchical statecharts [BFS13]. We use the *Nested Switch* pattern [BFS13; Sam09] to implement the behavior as it provides a straightforward, comprehensible method for encoding statecharts in a language that does not support object-oriented programming.

Listing 5.8 shows a code snippet of the implementation of the processStep function of the Overtakee component RTSC (cf. Figure 2.2 in Section 2). The function implements the behavior of the component type and can be called to execute the behavior of the corresponding component instance, which it gets as an input parameter. The switch statement in Line 3 checks which state of the component instance is currently active. The processStep function executes the Lines 5–21 if the init state is active (cf. Line 4). The state init has one outgoing transition that has the request message as an enabling condition. Therefore, Line 5 checks if the message exists in its message buffer using the function Overtakee_fromOvertaker_exists_Request_Message. This function is provided by the platform-specific implementation of the container (cf. Figure 5.12 in Section 5.3.1 and Figure D.9 in Appendix D.1). The function has the corresponding port instance as an input parameter. When the container function Overtakee_fromOvertaker_recv_Request_Message is called the message is popped from the buffer and stored in the local variable msg_Request_Message. This function is provided by the platform-specific implementation of the container (cf. Figure 5.12 in Section 5.3.1 and Figure D.9 in Appendix D.1) and realizes the technical concerns for consuming the message from the buffer. Additionally, the variable msg_Request_Message is used to access the parameters of a message. The Lines 13–15 state the code comments where we would generate corresponding RTSC actions. The Lines 16–17 change the value of the state variable by setting the target state of the transition as the new value. Line 18 completes the state transition by executing the entry action of the target state. The Line 19 is empty as the state init has only one outgoing transition. The break statement in Line 20 prevents that the Line 21 and the following lines are executed directly. They are executed at the earliest when the switch statement is evaluated the next time. The generated platform-independent implementation of a MechatronicUML model can be built by a compiler, like the GNU Compiler Collection (GCC) [GCC], for different target platforms.

```
1  void OvertakeeComponent_processStep(
   OvertakeeOvertakeeBehavStateChart* stateChart) {
      switch (stateChart->currentStateOfOvertakeeActiveCommunicator) {
         case ACTIVECOMMUNICATOR_INIT:
5           if (Overtakee_fromOvertaker_exists_Request_Message(
            OvertakeeComponent_getToOvertaker(stateChart->parentComponent)){
               Request_Message msg_Request_Message;
               Overtakee_fromOvertaker_recv_Request_Message(
               OvertakeeComponent_getToOvertaker(
10             stateChart->parentComponent), &msg_Request_Message);
               // execute exit/transition actions and clock reset if required
               // change the state
               stateChart->currentStateOfOvertakeeActiveCommunicator =
               ACTIVECOMMUNICATOR_REQUESTED;
15             // execute entry actions if required
            } else {}
            break;
         case ACTIVECOMMUNICATOR_REQUESTED ...
      }
20 }
```

Listing 5.8: Code Snippet of the Implementation of the Behavior of the Overtakee Real-Time Statechart

To be executed on a specific-target platform, the generated component implementation needs to be linked against the platform-specific containers and the device API. Therefore, we

have to model this platform-specific information before generating the corresponding code for accessing the devices.

5.3.3 Platform-Specific Modeling

Engineers have to define configurations of the component instances for specific platforms. The code for the components cannot access concrete devices without the knowledge how to access the device. Software platforms of CPSs are very heterogeneous and provide specific APIs for accessing devices. State of the art approaches for CPSs are not sufficient because they often support only the generation of stubs/skeletons [CSVC11]. This requires a manual implementation of the platform-specific technical concerns, like in [PWT+08; CSCS13]. Other approaches, like MARTE [MARTE], provide no systematic process and modeling rules to be able to generate code for accessing devices and to deploy software to heterogeneous distributed platforms. Alternatively, approaches like MontiArcAutomaton [RRRW15] require to specify platform-specific code generators. Therefore, we need a mapping of component instances to concrete function calls of these device APIs for initializing, accessing, and terminating devices.

In MechatronicUML a continuous component can represent a device. A continuous port represents a device property or controller property. An example of a device property is the distance value that the ultrasonic sensor provides (cf. Figure 4.3 in Section 4.1). A continuous component and consequently its port can be instantiated several times in a component instance configuration. Each continuous port of each component instances can use a different API function.

The goals of the platform-specific modeling are: Firstly, to model the device API for embedded platforms and operating systems. Secondly, to model the mapping of continuous port instances to a concrete interaction behavior with a platform-specific device API. Thirdly, to use the mapping to generate platform-specific code. Fourthly, to model deployment configurations to generate platform-specific code for the communication.

Consequently, Section 5.3.3.1 introduces the *device access view type*. The view type provides two views. The *device API view* for specifying the device API for concrete software platforms and the *device API access view* for specifying how a concrete port instance interacts with a concrete device on a concrete platform. Furthermore, the automated software construction for specific distributed platforms requires a platform-specific deployment configuration model. Therefore, Section 5.3.3.2 introduces the *deployment configuration view type*. It defines for each ECU which component containers are required to handle component instances and how the communication for each port instance has to be handled on the different platforms using specific communication protocols and communication media. We adapt the approach of Bureš, Malohlava, and Hnětynka for using Domain-Specific Languages (DSLs) for the automatic generation of software connectors for business information systems [Bur06a; BMH08] to the domain of distributed CPSs.

5.3.3.1 Device Access Modeling

Software platforms of CPSs are very heterogeneous. For example, several different software platforms, i.e., operating systems exist for the Arduino [Ard; BBP+16] microcontroller, the single-board computer Raspberry PI [Rasa], or the LEGO Mindstorms NXT [CB13] robotics kit. Each platform provides different API functions for accessing devices. For example, for setting the value of a General-Purpose Input/Output (GPIO) pin on a Raspberry PI [Rasa] to high or low, Windows 10 IoT Core [Mic] provides the function `void GpioPin.Write(GpioPinValue value)` and Raspbian [Rasb], a Linux distribution, provides the function `void digitalWrite (int pin, int value)`. Even the parameters of the same function on the same hardware and the same

operating system vary if the device is connected to a different hardware port or GPIO pin. Furthermore, some functions of a device API have timing restrictions for its usage. An example of a timing restriction for an ultrasonic distance sensor of a LEGO Mindstorms is that it has to wait for at least 30 ms before it starts a new distance measurement [MH11]. This avoids interference in the ultrasonic sensor.

As a result, continuous component instances that have the same type and which are deployed on different hardware, have to use different function calls because each hardware may run different operating systems or may provide different device APIs. Portability [ISO25010] – the usability of the same software on different platforms – requires that the platform-independent code and the platform-specific code for accessing devices are separated because then the software can be ported to another platform easily.

Device API View

Our platform-specific modeling approach considers the modeling of the device API of software platforms in form of function signatures and time constraints for restricting the access. We provide the API Modeling Language (ApiMl), a simplified version of the *Software Resource Modeling Package* of the MARTE meta-model [MARTE]. Our language is tailored to the requirements of CPSs. The ApiMl allows the signature description of API functions within a model.

Figure 5.14 shows the meta-model of the ApiMl. An OperatingSystem provides several API repositories, where each APIRepository describes the API functions (APICommand) for exactly one sensor or actuator. An APICommand specifies its signature consisting of an ID, a return DataType, and Parameters. It is also possible to specify APICommands for subfunctions of a sensor/actuator and use them in the device API access view (cf. Section 5.3.3.1) if a complex control behavior for a device is required. Optionally, an APICommand can be annotated with a time constraint. A TimeConstraint specifies the minimum time interval between repeated calls of an API function. These time constraints have to be specified and respected in the generated code to ensure correct behavior. Listing C.2 in Appendix C.2 shows the EBNF grammar for the textual syntax of the ApiMl.

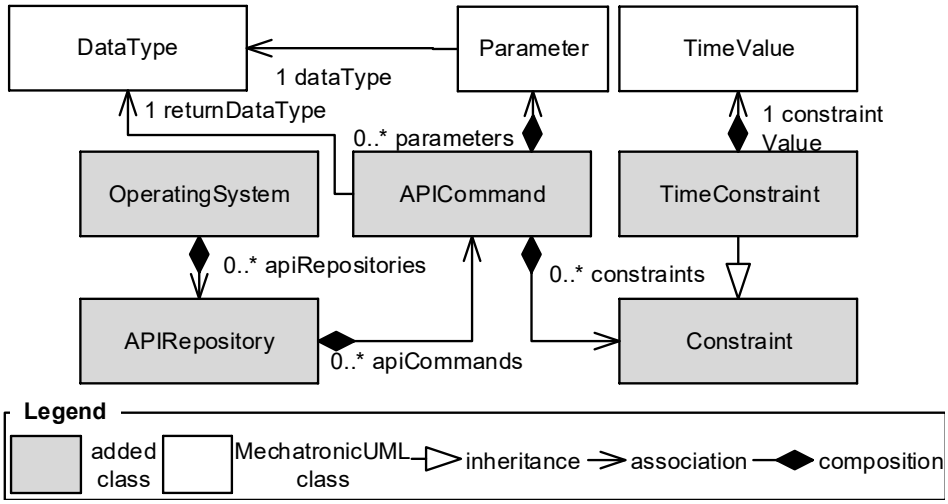


Figure 5.14: ApiMl Meta-Model

As an example, Listing 5.9 shows the API functions for accessing an ultrasonic sensor that is connected to a Mindstorms NXT microcontroller instead of an Arduino Gemma microcontroller (cf. Figure 4.2 in Section 4.1) of the operating system nxtOSEK [Chi]. The operating system nxtOSEK provides one API function for initializing and one API function

for accessing the ultrasonic sensor. The first API function ecrobot_init_sonar_sensor switches the power of the hardware port on and thus activates the ultrasonic sensor. The API function ecrobot_get_sonar_sensor returns the current distance value as an Integer. Furthermore, the ultrasonic sensor can be read with the shortest interval of 30 ms. Otherwise, corrupt distance values are returned.

```
1 OperatingSystem:nxtOSEK{ Device_API_Commands: UltrasonicSensor {
  VOID ecrobot_init_sonar_sensor(SHORT port_id);
  INT ecrobot_get_sonar_sensor(SHORT port_id)[30ms];}}
```

Listing 5.9: Example: API Commands for the Ultrasonic Sensor of the Operating System nxtOSEK

Device API Access View

Our platform-specific modeling approach considers the modeling of an allocation specific mapping model. The model maps each continuous port instance specified using the MechatronicUML to an API function specified using the ApiMl. Therefore, we introduce the textual API Mapping Modeling Language (APIMappingMl) to specify such a mapping. We embed the APIMappingMl into the action language of MechatronicUML (cf. Section 2.2.3). Thereby, we can use features like if-statements and loops. The APIMappingMl fulfills the following tasks: (1) define a mapping of port instances to an API function that should be executed regularly; (2) define a mapping of port instances to an initialization API function; (3) specify the concrete API function parameters; (4) provide an adapter between values used in a PIM and "actual" values for calling an API function.

Figure 5.15 shows the meta-model of the APIMappingMl. A MappingRepository consists of several PortAPIMappings. A PortAPIMapping references a port instance for which this mapping is specified. A PortAPIMapping consists of a required accessCommand (initialization), an optional initCommand, and an optional termCommand (termination). The accessCommand, the initCommand, and the termCommand are realized as an Expression of the MechatronicUML action language (cf. Section 2.2.3). Thereby, it is possible to alter the values before calling the APICommand. For example the value can be casted to another type or changed using conditional statements and loops. An APICallExpression is an Expression that references an APICommand, which has been specified using the ApiMl, and ParameterBindings for binding a concrete value to each parameter of the APICommand. Listing C.3 in Appendix C.3 shows the EBNF grammar for the textual syntax of the APIMappingMl.

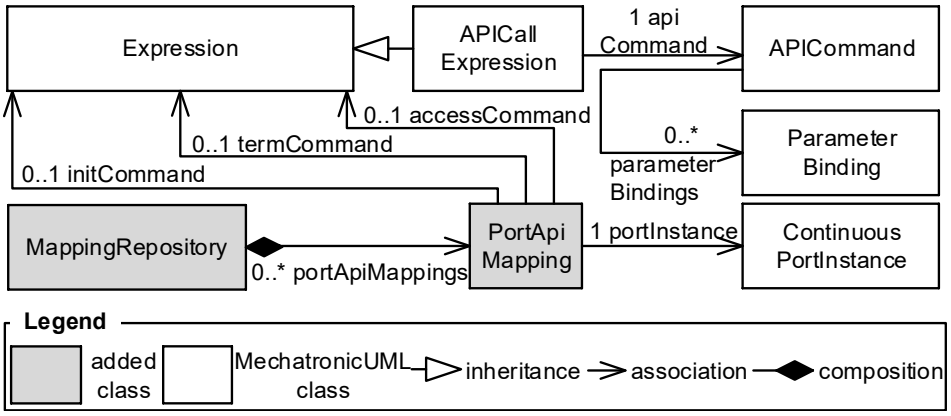


Figure 5.15: APIMappingMl Meta-Model

The API mapping for the continuous distance port instance for our robot is shown in Listing 5.10. The software of the robot has the continuous component instance overtakeeDistance, which has the port instance distance (cf. Figure 4.3 in Section 4.1). The ultrasonic sensor is connected to the Lego Mindstorms ECU, which runs the nxtOSEK operating system. Therefore, we use the API function ecrobot_get_sonar_sensor(SHORT port_id) specified in the ApiMl. The ultrasonic sensor is connected to the port "3" of a Lego Mindstorms NXT ECU. Therefore, the API function is called in the APIMappingMl with the concrete parameter binding ecrobot_get_sonar_sensor(port_id:=3). The return value is bound via an assignment to the variable distance. The API function ecrobot_get_sonar_sensor(port_id:=3) returns the INT32 value -1 if the sensor fails and otherwise INT32 values between 0 and 255. The RTSC expects the DOUBLE value -1.0 if the sensor fails and otherwise DOUBLE values between 0.0 and 1.0. Firstly, the return value is cast to the type DOUBLE. Secondly, the distance variable is divided by 255.0 if it is equal or greater than 0. The ultrasonic sensor must be initialized before it can be used and terminated if it is not used anymore. Thus, the initCommand is specified as ecrobot_init_sonar_sensor(port_id:=3) and the termCommand is specified as ecrobot_term_sonar_sensor(port_id:=3)

```
1  MappingRepository: nxtOSEKMapping{
     PortInstance:
      overtakeeDistance.distance{
        accessCommand:{
5         DOUBLE distance:= (DOUBLE) ecrobot_get_sonar_sensor(port_id:=3);
          if (distance>=0){
          distance:= distance / 255.0;
        }
        return distance; }
10      initCommand:{
         ecrobot_init_sonar_sensor(port_id:=3) ;}
        termCommand:{
        ecrobot_term_sonar_sensor(port_id:=3);}
      }
15  }
```

Listing 5.10: API Mapping for the Port Instance Distance

5.3.3.2 Deployment Configuration Modeling

The model-driven generation of code for models specified using MechatronicUML to the architecture-centric, container-based architecture requires a platform-specific deployment configuration model. This model defines the required component container, the allocation of component instances to container, and the realization of the port instances' communication by the container and respectively by the underlying middleware or operating system. Therefore, we introduce the *deployment configuration* meta-model.

Figure 5.16 shows the meta-model for specifying a deployment configuration. The generation model represents a deployment configuration for a concrete system allocation The deployment configuration specifies for each ECU of a hardware platform instance an ECU specific configuration. Each ECUConfiguration contains the configuration for each required container for each allocated component type. Each ComponentContainerConfiguration contains configurations for each component instance that are managed by the container. Each ContainerComponentInstanceConfiguration contains for each port instance a configuration. Each PortInstanceConfiguration specifies how a component container implements the communication of a port instance on a concrete ECU.

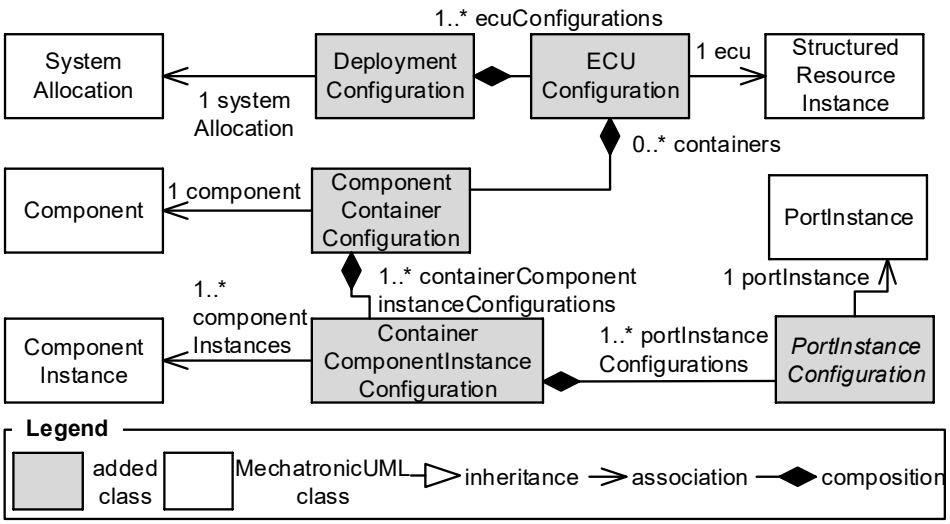


Figure 5.16: Deployment Configuration Meta-Model

Communication for CPSs can be implemented by many different communication patterns [HEJ15], e.g., shared memory, publish-subscribe, or request-reply, network protocols, e.g., TCP [TCP], UDP [UDP], or Control Area Network (CAN), and communication middlewares [Mah05], e.g., OPC-UA [MLD09], DDS [DDS], or AUTOSAR [AUTOSAR14c]. Software engineers have to decide how a container should realize the communication of each port instance. They have to provide a concrete manifestation of the port instance configuration. This manifestation defines how the platform-specific code generator realizes a connector between two port instances to establish a communication channel.

Figure 5.17 shows a meta-model that defines two manifestations of a port instance configuration. The class SharedMemoryPortInstanceConfiguration uses the communication pattern *shared memory* and the class DDSPortInstanceConfiguration uses the middleware DDS [DDS] and its communication pattern *publish-subscribe* to realize the communication. We provide for these two port instance configurations a platform-specific container code generation.

The SharedMemoryPortInstanceConfiguration can only be used if the two component instances that have to communicate are allocated to the same ECU. The class SharedMemoryPortInstanceConfiguration has two properties of type ELong to store the IDs of the communication partner.

The DDSPortInstanceConfiguration can only be used if the two component instances that have to communicate are located within the same subnetwork of a physical IP network or are allocated to the same ECU. The class DDSPortInstanceConfiguration references several DDS-specific classes to configure the communication. Creating and configuring a communication middleware, like DDS, is a complex task. We automated this task based on the information stored in the DDSPortInstanceConfiguration. Section 5.3.4.4 describes the definition of the mapping of a model specified using MechatronicUML to a DDS configuration in detail.

5.3.4 Platform-Specific Implementation

The platform-independent implementation of component types handles the functional concerns. A platform-specific implementation that handles the technical concerns is missing to realize a scenario using distributed CPSs. This section describes how to generate such a platform-specific implementation for MechatronicUML.

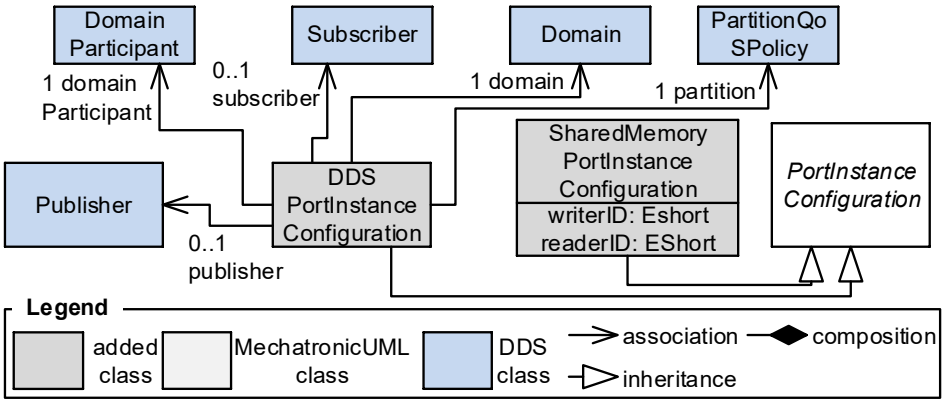


Figure 5.17: Shared Memory and DDS Port Instance Configurations

The behavior of the components requires information from accessed operations and devices to be executed correctly. Therefore, we need software adapters to access platform-specific libraries and device APIs. Section 5.3.4.1 describes how to implement the adapters to concrete software libraries manually. Next, Section 5.3.4.2 describes how to generate platform-specific code automatically to access devices. Furthermore, a component type has to be instantiated for executing its behavior and to get a state during the initialization. Consequently, the whole lifecycle of a component instance has to be handled. Therefore, we require for each component type for each ECU a mediator between the platform-independent implementation and the underlying software platform that realizes the technical concerns. Section 5.3.4.3 describes how we generate platform-specific component-containers that fulfill the task of a mediator. Moreover, even the dedicated communication middleware that realizes the communication between ECUs have to be configured and created. Section 5.3.4.4 describes how to generate communication middleware artifacts automatically using DDS [DDS].

5.3.4.1 Implement Operations as an Adapter to Software Libraries

Section 5.1.1 introduces operation repositories with operations that can be called within actions. These operations require an implementation. We generate for each operation repository a header file that contains definitions for all operations and a source file that contains stubs for all operation implementations. Software engineers can include existing software libraries within the file and provide an implementation of a platform-specific adapter manually to the existing functions of each software library. Note, the problem of reusing existing operations is comparable to the usage of device APIs. In contrast to accessing device APIs, operations do not have to consider timing restrictions. Furthermore, we do not consider the initialization, accessing, and termination explicitly, like we do it for using devices.

Listing 5.11 shows the mixture of the manual and generated implementation for the operation repository SectionControlDatabase (cf. Section 5.1.1). Line 1 includes the corresponding generated header file. The developer has to define the include statement for the software repository within the header file manually, e.g., `#include ''unqlite.h''` for including the NoSQL database implementation UnQLite [Sym]. Line 3 defines the stub for the createDatabase operation with the return type BOOLEAN. Line 4 defines a comment that is a marker for the code generator to start a protected code area that is kept on regeneration. Line 5 defines a database handle. Line 6 creates or opens an already existing database by calling the function unqlite_open. The following Lines 7-9 check if the database is correctly created and initialized. The createDatabase function returns true if the creation was successful and false otherwise. Line 10 defines a comment that is an end marker of a protected code for the code generator. The Lines 12-15

contain the generated stubs for the other functions that are part of the operation repository. These also have to be implemented manually.

```
#include "SectionControlDatabase.h"

bool_T createDatabase(){
/** Start of user code **/
unqlite *pDb; /* Database handle */
rc = unqlite_open(&pDb, "sectionControl.db", UNQLITE_OPEN_CREATE);
if (rc != UNQLITE_OK)
   return false;
else return true;
/**End of user code**/
}
void updateCarSection(int32_T secId, int32_T carId){...}
int32_T getCarSection(int32_T carId){...}
bool_T isSectionEmpty(int32_T secId){...}
```

Listing 5.11: Generated Stubs for the Operation Repository SectionControlDatabase

5.3.4.2 Generate Device Access Code

We model in Section 5.3.3.1 how to access a device API function of a continuous port instance. Software engineers model the commands accessCommand, initCommand, and termCommand as an expression using the MechatronicUML action language. In this section we define how to generate code for the modeled commands. The source code of these platform-specific commands does not differ from the implementation of the platform-independent actions (cf. Section 5.3.2.5). Therefore, we can reuse the platform-independent code generation for the MechatronicUML action language. Furthermore, we have to generate a corresponding variable in the component container for time constraints of API commands (cf. Section 5.3.3.1).

Listing 5.12 shows the generated code for the API mapping for the continuous distance port instance (cf. Listing 5.10 in Section 5.3.3.1). The Lines 1–5 show the accessCommand including the call of the function ecrobot_get_sonar_sensor(port_id:=3) and the assignment of the return value to the local variable distance. The function accessCommand returns the value of the local variable distance divided by the value 255 to the caller. The accessCommand is called via the CallParameter accessFunction of the corresponding statechart of the continuous distance port (cf. Section 5.1.4). Section 5.3.4.3 shows the missing binding of the accessFunction pointer to this accessCommand implementation as part of the instantiation of a component instance by the component lifecycle management.

The Lines 6–9 show the implementation of the initCommand and the Lines 10-13 show the implementation of the termCommand. These functions are called during the initialization and the termination of a component instance (cf. Section 5.3.4.3). Furthermore, Line 15 shows the concrete time variable for the time constraint of the device API function (cf. Listing 5.9 in Section 5.3.3.1). The variable is bound to the timeInterval variable of the corresponding component and restricts the call to the API function (cf. Listing 5.3 in Section 5.3.2.3).

5.3.4.3 Generate Container Implementation

Component containers are the mediators between the platform-independent implementation of components and technologies that realize the technical concerns. Therefore, a component container provides functions to components and uses functions of the underlying technology. The provided functions handle the lifecycle of component instances and realize the

```
double accessCommand(void)
{
	double distance = (double) ecrobot_get_sonar_sensor(3);
	return distance / 255.0;
}
void initCommand(void)
{
	ecrobot_init_sonar_sensor(3);
}
void termCommand(void)
{
	ecrobot_term_sonar_sensor(3);
}

int32_T concreteTimeInterval = 30;
```

Listing 5.12: Example: Generated C-function for the API Mapping for the Overtakee Distance Port Instance

communication (cf. Section 5.3.1). Especially, the realization of the communication depends on the requirements of the application domain. Automotive systems use communication middlewares on the basis of AUTomotive Open System ARchitecture (AUTOSAR) [AUTOSAR14c], whereas control systems use communication middlewares on the basis of DDS [DDS] or OPC UA [MLD09]. Furthermore, a concrete container implementation depends on the operating system and the network interfaces that an ECU provides. A component container serves as an abstraction layer of the underlying technologies and as an adapter between components and existing functions that are provided by the underlying technologies. This section defines which functionality containers have to provide as an abstraction layer for an implementation of models specified using MechatronicUML.

Lifecycle Management of Component Instances: A component container manages the lifecycle of all component instances that have the same type and are allocated to the same ECU. That means it creates a component instance at the beginning of its lifecycle and destroys it at the end of its lifecycle. Furthermore, it may activate and may deactivate a component instance if the software is self-adaptive. In this thesis, we consider only static software configurations. Nevertheless, the lifecycle callback pattern (cf. Appendix D.4) can also be used for reconfigurable components because it provides also function for the activation and the deactivation of component instances. The component container uses not only the lifecycle callback pattern for the lifecycle management of a component instance but also the other patterns that are described in Appendix D.

Figure 5.18 shows the instantiation of a component instance by the container as a UML sequence diagram. Note that we use an object-oriented illustration to describe the concepts for better understandability. In contrast, our actual implementation uses C and all objects are represented as structs. We allocate the memory of the required variables statically. As a result, the required resources are known during the compilation.

The Main object calls the create function of the Container object of the component type that should be instantiated. The input parameter of the create function is a unique identifier of each component instance. The Container object represents the ComponentDirector of the builder pattern. The Container creates a new ComponentInstanceBuilder object named builder. The reference is returned to the Container object. Afterward, the Container calls the buildPart function of the builder object. The builder creates a new object typed over the Component. The reference is stored in the variable instance. In the following first loop, the value of each component parameter (cf. Section 5.1.2) is bound to a concrete binding expression corresponding to

the platform-specific parameter binding (cf. Section 5.1.3). Furthermore, the second loop initializes each port instance. For each port instance, the port status is initialized with the value UNCONNECTED and a port handle is created corresponding to the specification of the platform-specific PortInstanceConfiguration (cf. Section 5.3.3.2).

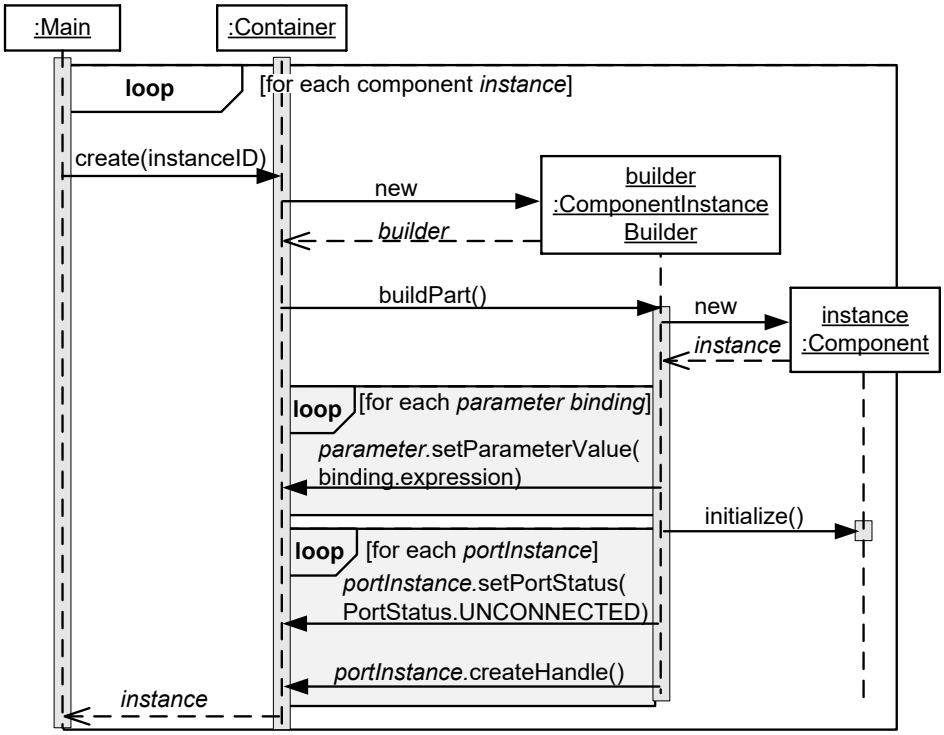


Figure 5.18: UML Sequence Diagram of the Initialization of Component Instances by a Component Container

Using the parameter binding meta-model (cf. Section 5.8) enables to specify a platform-specific configuration for a component instance. It is possible to generate the component instance parameter binding for continuous port instances based on the information provided by the APIMappingMl (cf. Section 5.3.3.1). Developers have to model bindings for other parameters manually or have to add them manually within the code.

The component's time parameters and call parameters are bound to concrete values and the device access functions are bound to concrete functions according to the parameter binding of the component instances (cf. Section 5.1.3).

Listing 5.13 shows an example of the binding that is executed during the creation of a component instance (cf. Section 5.3.4.3) via using the builder pattern (cf. Appendix D). The component instance overtakeeDistance (cf. Figure 4.3 in Section 4.1) requires bindings for its port. The TimeParameter timeInterval that is shown in Figure 5.9 in Section 5.1.4 is bound to the TimeValue of 30ms because otherwise corrupt distance values are returned. Furthermore, the CallParameter accessFunction that is shown in Figure 5.9 in Section 5.1.4 has to be bound to the access command, which is shown in Listing 5.12 in Section 5.3.4.2, that we generate from the mapping, which is shown in Listing 5.10 in Section 5.3.3.1.

In comparison, the destruction of a component instance is quite easy. The Container calls the destroy function of the component with the reference to the instance as an input parameter. The destroy function sets the value of the reference pointer that directs to the instance to the new value NULL.

```
/**Binding of Time Parameter**/
timeInterval = &concreteTimeInterval;
/**Binding of Call Parameter**/
accessFunction = &accessCommand ;
initFunction = &initCommand ;
termFunction = &initCommand ;
```

Listing 5.13: Example of the Definition of Component Parameter and Its Binding

Management of System Resources: Embedded systems provide only a limited amount of resources [GBK+08]. Containers have to take care of the resource usage. Some ECUs resources are shared between containers, like a network socket or the usage of middleware functionality. Therefore, component containers share *resource pools* to manage resources efficiently [BHS07]. Resource pools allow the sharing of resources across container boundaries and can be used to optimize the usage of resources. The deployment configuration defines statically the usage of resources. Therefore, all required resources are known during the generation of containers.

Transparent Communication: Component instances can be allocated to different ECUs. Communicating component instances exchange messages by using connectors [BP02] via the send, receive and exist functions that we define within our code framework (cf. Figure 5.12 in Section 5.3.1). The implementation of the connector is completely transparent for component developers. That means, they only have to know the corresponding interface functions for all messages that are sent and received by an RTSC. The developers of the application do not need to know how these functions are implemented. Each component container has to implement these functions for each message type that can be sent or received. The functions use the handle that the handle pattern (cf. Appendix D.2) defines to choose the right network interface.

Listing 5.14 shows an example of an implemented send function that uses the local port instance configuration. It gets the reference p to a Port instance and the reference msg to an Accept_Message message as its input parameters. In Line 3 it defines a pointer to a local handle. It is possible to provide more than one handle and to change the handle that is actually used during runtime. The switch statement in Line 5 reads which handle is set for the port and chooses the right case to realize the send function. The variable localHandle gets in Line 7 the actual handle that is stored within the port instance if the handle of the port p is of type PORT_HANDLE_TYPE_LOCAL. Furthermore, in Line 8 the function writeMessage is called. The function writeMessage is a helper function that is provided via a library for managing communication via local shared memory. The function gets the memory address where the message should be stored, the type of the message, and the message itself that should be sent as its input parameter.

Quality of Communication Service and Message Buffering: In MechatronicUML the behavior of ports has to refine the behavior of a Real-Time Coordination Protocol (cf. Section 2.2.1). The connector of a Real-Time Coordination Protocol defines QoS assumptions for the message transfer and defines message buffering properties.

The first QoS property of a Real-Time Coordination Protocol is message loss. It defines if the verification assumes that messages can get lost or that messages always arrive. That means that the verification guarantees a correct behavior that does not violate safety or real-time requirements if messages get lost if this property is *true*. Corrupt or lost messages do not have to be retransmitted. It is not allowed that messages get lost if the message loss property is false. Therefore, only reliable network protocols that guarantee deterministic communication behavior are allowed. Such a location constraint can be checked during the allocation planning

```
1  void Overtakee_toOvertaker_send_Accept_Message(Port* p,
   Accept_Message* msg){
   LocalHandle* localHandle;
   ...
5  switch(p->handle->type) {
      case PORT_HANDLE_TYPE_LOCAL:
         localHandle = (LocalHandle*) p->handle->concreteHandle;
         writeMessage(localHandle->writerID, MESSAGE_ACCEPT, msg);
      break;
10 ... }
```

Listing 5.14: Example of a Send Function

(cf. Section 4.5.1.5). The second QoS property of a Real-Time Coordination Protocol is the *maximum delay* for the message transmission and consumption. The reliable network protocol and application has to guarantee this real-time property. Otherwise, safety-relevant end-to-end deadlines and a correct coordination behavior cannot be guaranteed. That means that a reliable network must ensure that the deadline is fulfilled. Messages that arrive too late or are not consumed within the delay time have to be dropped if message loss is allowed. These *transmission assumptions* must be fulfilled by the communication service that a component container provides. Otherwise, it cannot be guaranteed that verified safety and real-time requirements hold.

A software component instance may receive messages that it cannot consume immediately. As software component instances are executed independently the sender does not block until the receiver component instance is able to consume the message. Therefore, the receiver has to buffer messages. MechatronicUML specifies that message buffer have a First in, First Out (FIFO) semantics [*DPP+16]. Unreliable network protocols, like UDP [UDP], do not guarantee that messages are received in the same order as they are sent. As a result, each message requires a timestamp of the sender so that the messages can be sorted according to the sending time. A message has to be dropped if a message with a newer timestamp has already been consumed. This behavior does not violate the semantics of MechatronicUML for connectors between port instances that are marked as unreliable because the verification guarantees correct behavior also with message loss. A more reliable network protocol that guarantees chronological order, like TCP [TCP], must be chosen if a connector does not allow message loss and if message orders have to be preserved. TCP [TCP] cannot be used to fulfill real-time properties. Therefore, reliable domain-specific protocols, like Ethernet-based real-time and industrial communication protocols [Dec05] or inter-vehicle communication protocols [WTM09], have to be used. Both component instances have to be allocated to the same ECU if no reliable protocol is available. A port instance of MechatronicUML defines for each message type a buffer size and a message displacement strategy that defines if it is allowed to displace the message with the oldest timestamp. These *buffer assumptions* must be fulfilled by the communication service that a component container provides. Otherwise, it cannot be guaranteed that verified safety and real-time requirements hold. Middlewares, like DDS [DDS] provide buffering and QoS policies out of the box [SPF04; SCH08]. We generate for DDS a middleware configuration automatically (cf. Section 5.3.4.4).

Message Marshaling and Message Unmarshaling: Message types that are specified using MechatronicUML use data types defined by MechatronicUML. The representation within C is defined within the header files StandardTypes.h, CurstomTypes.h, and MessageTypes.h (cf. Figure 5.13 in Section 5.3.2.4). Middlewares and network protocols define their own representation. A container has to transform a message into a representation that conforms to the representation of the underlying technology to be sent over the network. This is

called marshaling [BNOW93] and is implemented within the send function. The container that receives the message has to transform the message from the representation of the underlying platform back to a MechatronicUML conform representation. This is called unmarshaling [BNOW93] and is implemented within the receive function. Middlewares, like DDS [DDS] and CORBA [CORBA], provide interface description languages that can be used to generate type support. Network protocols like TCP [TCP] or CAN [ISO11898] define a data frame that can be used to transmit data. We do not define a general implementation for marshaling and unmarshaling as it depends on the underlying technology but define that a valid implementation of a container has to provide the functionality.

Lifecycle and QoS Management of Communication Links: CPSs operate in dynamic scenarios and in different environments. Normally, they rely on a dynamic network topology. However, if they rely only on a static network topology, also not all systems must be available the entire time. For example, cars move on a street and do not know where they get a network connection with other cars or with the next section control station. Therefore, a container has to manage when peers join, leave, or lose the connection to a network. The application software cannot control the coordination with the other peers if it has no information about the current status of the connections and its QoS. The lifecycle management of communication links and the management of QoS properties is a complex task in dynamic network topologies [XMDS08] because they have to manage the reliability and the QoS of the connection [LCP+05] to fulfill the QoS assumptions of the implemented Real-Time Coordination Protocols (cf. Section 2.2.1). Providing an implementation for the lifecycle management completely by oneself is a complex task. Nevertheless, existing network overlay protocols [LCP+05] and network virtualization [CB10] are common techniques to realize discovery of communication peers, (re)connecting to communication peers, clearing of connections, and monitor the communication status. A communication middleware that provides these functionalities within IP-based networks is, e.g., DDS [SPF04; SCH08; DDS]. Therefore, a container can provide an adapter to the functionality of an existing communication middleware to manage the lifecycle and the QoS of communication links.

Feedback Indicating the Connecting Status to Software Component Instances: A software component instance realizes the functional concerns of a CPSs. It needs information from other component instances and has to provide information to realize some concerns. For example, the Overtaker requires the acceptance of the Overtakee to perform a safe and cooperative overtaking maneuver. Therefore, each component instance executes an RTSC for each port instance. The execution of a port statechart requires that the port instance is connected to another port instance of another component instance that is allocated to a different ECU. A component instance can access the status of the ports connection via the function getPortStatus(Port) (cf. Appendix D.1). Furthermore, the container has to implement the function setPortStatus(Port,Status) and has to set the status depending on the connection status of the underlying network. DDS [DDS] provides the callback function on_liveliness_changed. It informs a receiver if the sender is truly lost, if a connection is recovered, and if a new sender is available. The port instance status can be updated according to such a callback. It may be useful to execute RTSCs only if a port connection is available.

5.3.4.4 Generate Communication Middleware Artifacts

Communication middlewares provide an abstraction of low-level technical realizations that are related to communication between distributed application parts [PG14]. They ease the implementation of the technical realization of a transparent communication (cf. Section 5.3.4.3). Furthermore, they provide communication services and message buffering with a defined QoS (cf. Section 5.3.4.3), lifecycle management of communication links (cf. Section 5.3.4.3), and message

(un)marshaling (cf. Section 5.3.4.3). Communication middlewares, like CORBA [CORBA], OPC UA [MLD09], and DDS [DDS] are heavily used within industrial applications [PG14]. Programmers benefit from using communication middlewares as they only have to define which information of the application are exchanged in which quality. They do not have to cope with establishing, managing, and encoding communication.

The configuration of communication middlewares for CPSs that have to fulfill real-time properties is a complex task [GBK+08]. Gokhale *et al.* introduced the software approach called *model-driven middleware* [GBK+08] to face the challenge to configure complex communication middlewares. We adopted the model-driven middleware approach for our automated software construction approach and implemented it using DDS [DDS]. DDS serves as an abstraction of the underlying communication protocol and media. DDS uses the publish-subscribe communication pattern [EFGK03]. In contrast to MechatronicUML that uses an exclusive pair message passing and queuing communication pattern [*DPP+16]. We need a connector between both communication patterns to realize the MechatronicUML communication via DDS. The implementation of the connector to DDS is used as a proof of our container concept.

We generate a DDS middleware configuration model based on the OpenDDS [Mar12] metamodel from models on the basis of MechatronicUML. The DDS model is used by tools provided by OpenDDS to generate a DDS Interface Definition Language (IDL) file that can be compiled into a target language, like C [Mar12]. Figure 5.19 shows the generation template for creating a DDS middleware for MechatronicUML message types, components, and Real-Time Coordination Protocol QoS assumptions. The container implementation uses the generated code from this configuration file and its interfaces to realize the communication.

In the following, we describe how to realize a MechatronicUML-DDS communication connector by means of how to represent components, ports, connectors, messages, message buffers, and connector QoS properties of MechatronicUML in DDS [DDS; SPF04].

First of all, we need a representation of messages used in MechatronicUML within the DDS *data model*. Therefore, we create a corresponding struct for each message type. The struct contains a field for each parameter of the message type. Data types provide type safety and provide structural information needed to manipulate the data. Furthermore, DDS uses *topics* to identify data items uniquely within the *global data space*. Therefore, we create a topic for each message type that references the corresponding data type. Domain participants represent communication partners within DDS. A domain represents a virtual network. Domain participants that share the same domain can communicate. Therefore, we create a domain element for the *least common ancestor* of connect components and set the domain of the domain participant to the corresponding domain element. DDS requires a *publisher* as a part of a domain participant to distribute data and a subscriber to receive data. We create for each port that sends messages a publisher and for each port that receives messages a subscriber. The publisher may publish data of different data types and the subscriber may subscribe data of different topics. A *data writer* and a *data reader* are required for each topic. We create a data writer for each raise message and a data reader for each receiver message of each port. Providing these DDS elements is sufficient for DDS to establish a communication. Nevertheless, we need additional QoS policies to represent the communication semantics of MechatronicUML.

Real-Time Coordination Protocols assume a certain connector quality. They require a reliable communication if message lost is not allowed. Therefore, we create a reliability QoS policy and set the kind to RELIABLE if message lost is not allowed. As a result, DDS waits for receiver acknowledgments and re-sends messages in the case of message loss. The connection status of a port is set to UNCONNECTED if a message gets lost finally. We set the reliability kind to BEST_EFFORT if message lost is allowed. DDS does not wait for receiver acknowledgments and does not resend messages. The reliability property of data writer and data reader is set

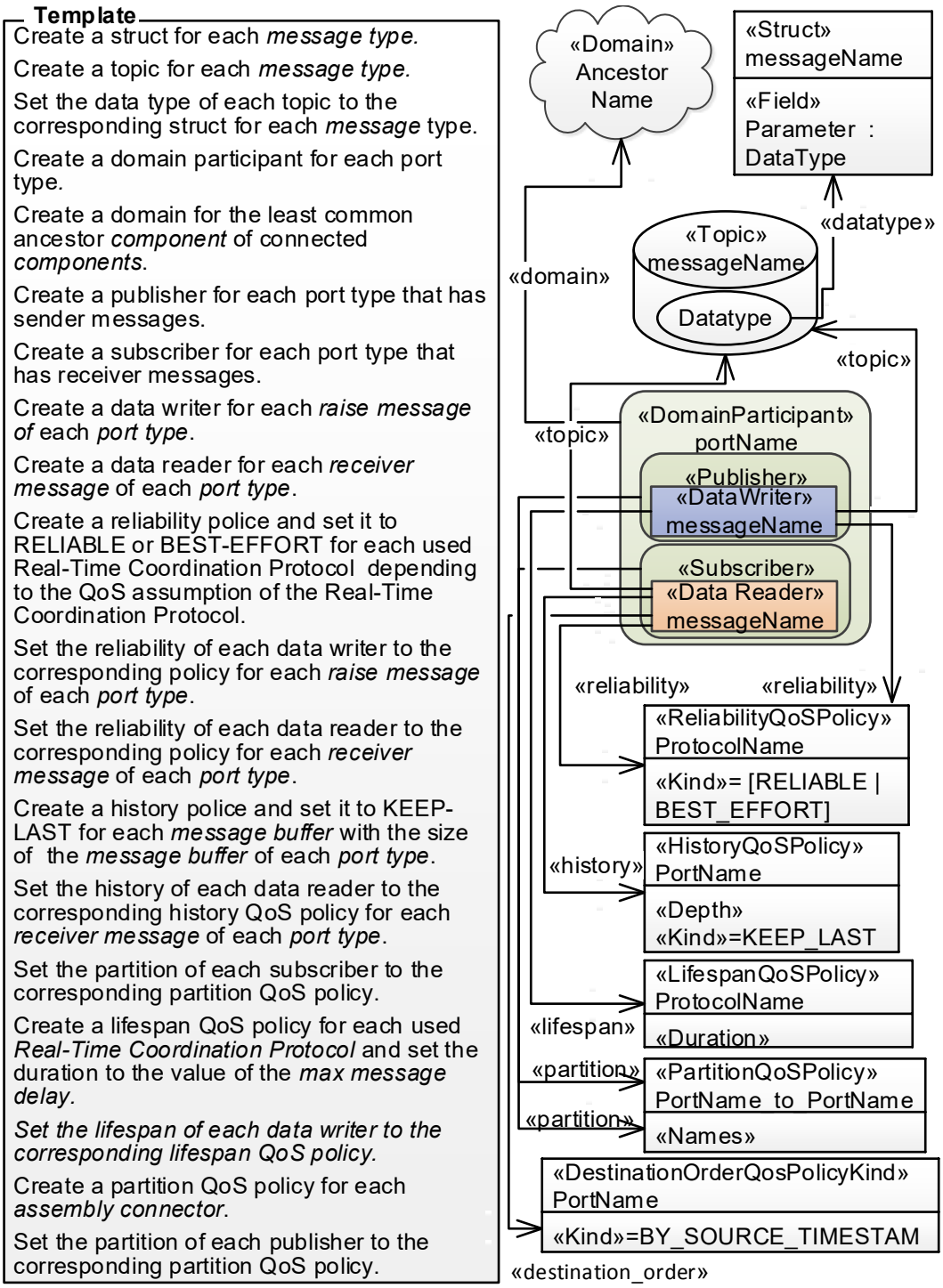


Figure 5.19: Generation Template for Creating a DDS Middleware Configuration Model for a MechatronicUML Component

to its corresponding reliability QoS policy. Ports that receive messages may have message buffers that store messages until they are consumed. Therefore, we create for each message buffer a history QoS policy. The kind is set to KEEP_LAST to store always the most recent messages. Furthermore, the depth of the history QoS policy is set to the buffer size. As a result, the buffer can store at least the required amount of messages. The history property of a data reader and a data writer is set to the corresponding history QoS policy. Real-Time Coordination Protocols assume that a message requires a maximum time to be consumed or is lost. We reflect this behavior by creating the lifespan QoS policy and set the duration to the value of the protocols max delay. Thereby, DDS delivers messages only until the duration expires. The lifespan property of a data writer is set to the corresponding lifespan QoS policy. Furthermore, it is possible in MechatronicUML that a component of the same type is instantiated multiple times. In this case, only connected port instances are allowed to communicate with each other and not all ports of the corresponding port types. In DDS only data readers and data writers that share the same *partition* can communicate. We create a partition QoS policy and set the partition property of data readers and data writer to the corresponding partition QoS policy. The partition QoS policy has the names property that defines the identifier of the partition. The component container knows for each port instance the partition id. The container creates instances of domain participants. It sets the value of the names property when creating a domain participant for a concrete port instance. Lastly, Real-Time Coordination Protocols assume that the message order is preserved. We reflect this behavior by creating the destination order QoS policy and set the kind to the value BY_SOURCE_TIMESTAMP. Thereby, we ensure that the sending order is preserved. The destination_order property of a data reader is set to the corresponding destination order QoS policy.

Figure 5.20 shows an example of a transformation of a MechatronicUML component into a DDS middleware configuration. The example represents the communicator component of the overtakee. We omit some ports and messages to reduce the complexity of the example.

The upper part of Figure 5.20 shows a model specified using MechatronicUML. The component communicator has the two ports overtakeeP and informSectionCtrP. The port overtakeeP sends the messages accept and decline and receives the messages request and finish. The buffer size of each received message is five. Furthermore, the maximum allowed delay for receiving and consuming messages of overtakeeP is 500 ms and of informSectionCtrP is 200 ms. The protocol that the port overtakeeP implements has no deadlocks even if messages get lost. The port informSectionCrtP sends the message enteredSection with the message parameters SecId and CarId of type INT32. The protocol that the port informSectionCrtP implements has no deadlocks even if messages get lost.

The lower part of Figure 5.20 shows a model specified using OpenDDS [Mar12]. The model is the result of the M2M transformation that implements the generation template for creating a DDS middleware configuration model (cf. Figure 5.19). The message types enteredSection, accept, decline, request, and finished are transformed into corresponding structs and topics. The data type INT32 of the parameters SecId and CarId is transformed into the DDS data type Long.

The port overtakeeP is transformed into the domain participant overtakeeP. It contains a data writer for both send messages (accept and decline). Furthermore, it contains a data reader for both received messages (request and finished). The *message lost* property of the used Real-Time Coordination Protocol OvertakingCoordination is transformed into the reliability QoS policy with the same name, the kind BEST_EFFORT, the references from the data writer accept and decline, and the data reader request and finished. The message buffer is transformed into the history QoS policy overtakeeP with the depth 5 and the kind KEEP_LAST. The max delay protocol assumption is transformed into the lifecycle QoS policy OvertakeeCoordination with the duration of 500 ms and the corresponding references from the data writer accept and decline. The message order assumption is transformed into the destination order QoS policy overtakeeP with the

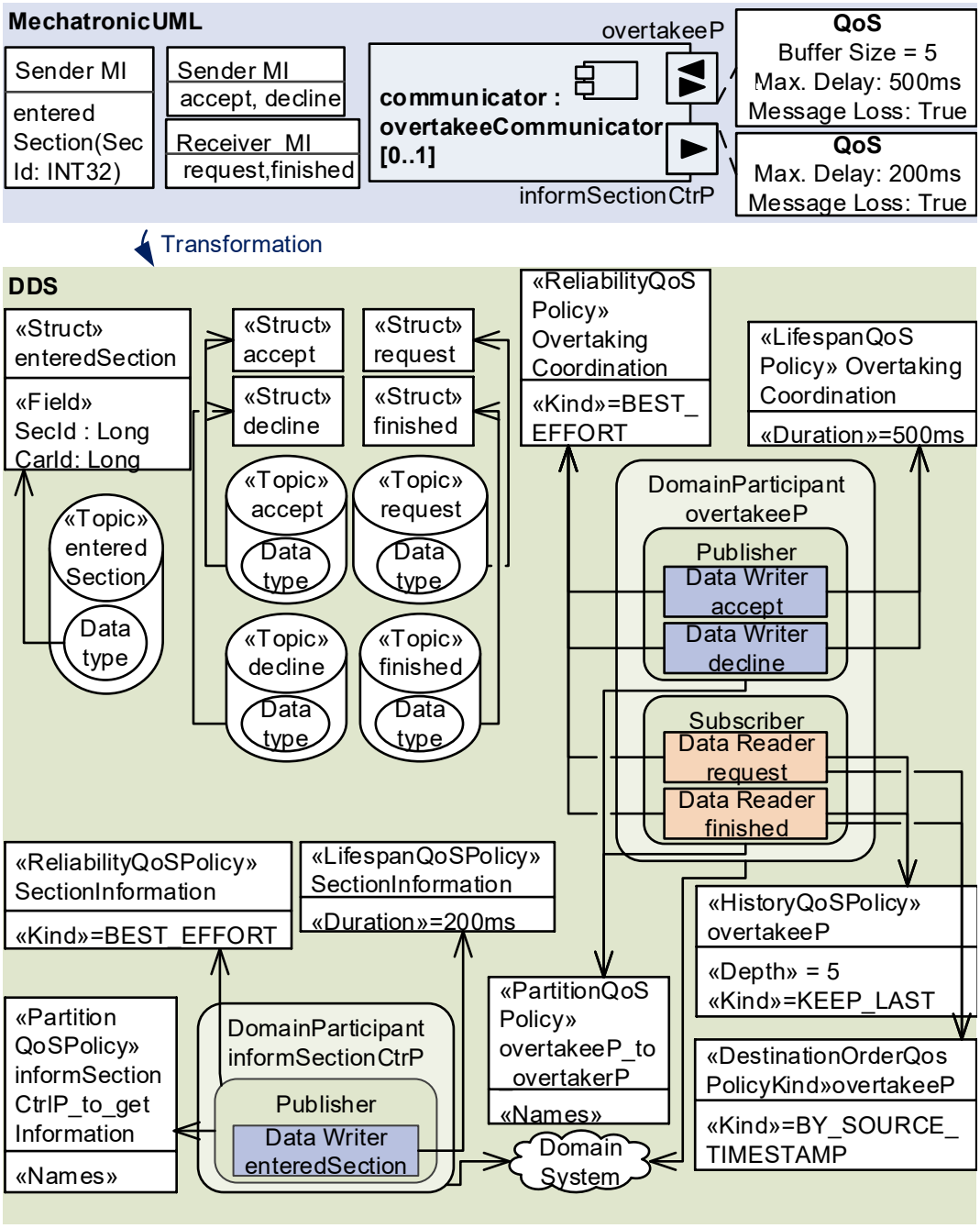


Figure 5.20: Transformation of a MechatronicUML Component into a DDS Middleware Configuration

kind BY_SOURCE_TIMESTAMP. The MechatronicUML connector that connects the port overtakeeP with the port overtakerP (not shown in the upper part) is transformed into the domain System, the partition QoS policy overtakeeP_to_overtakerP, and corresponding references.

The port informSectionCtrP is transformed into the domain participant informSectionCtrP. It contains for the send message enteredSection a data writer. The message loss property of the used Real-Time Coordination Protocol SectionInformation is transformed into the reliability QoS policy with the same name, the kind BEST_EFFORT, and the reference from the data writer enteredSection. The max delay protocol assumption is transformed into the lifecycle QoS policy SectionInformation with the duration of 200 ms and the corresponding reference from the data writer enteredSection. The MechatronicUML connector that connects the port informSectionCtrP with the port getInformation (not shown in the upper part) is transformed into a reference from the domain participant to the already existing domain System and the partition QoS policy informSectionCtrP_to_getInformation and the corresponding reference from the publisher.

5.3.5 Build

The source code of the system has to be converted into executable software artifacts that can run on distributed, connected ECUs. The process is called build. Mainly, a build consists of two tasks. Firstly, a compiler translates, i.e., compiles the source code into *machine/object code* for a specific CPU [AU72]. We use the C-compiler for the language C [Rit93] from GCC [GCC]. The code is stored by the C-compiler [Sta01; ISO9899] within an *object file* (o-file). Secondly, a linker takes all required object files and combines them into an executable program if they are linked statically [PW72].

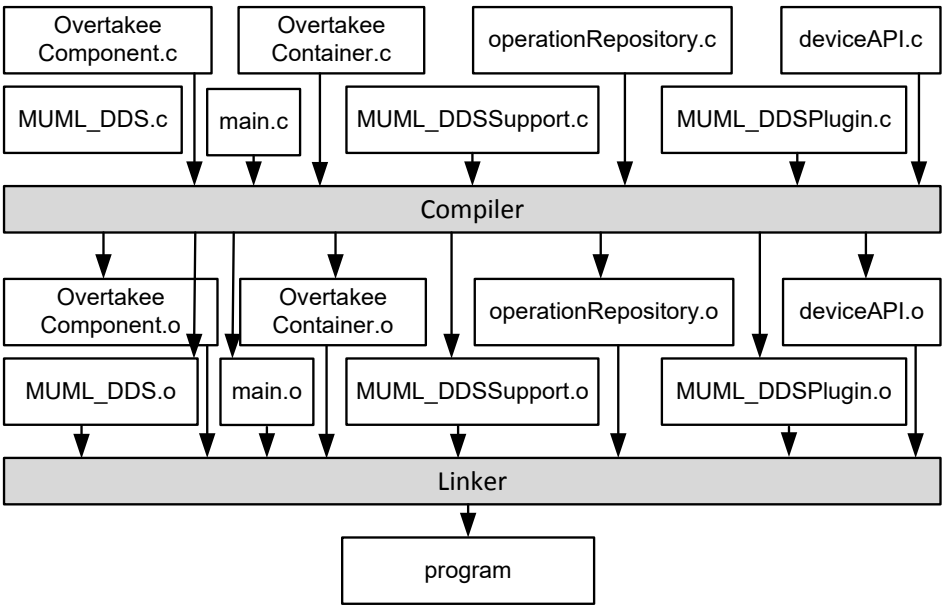


Figure 5.21: MechatronicUML Build Process

Figure 5.21 shows the build of the executable program of the Overtakee under the assumption that it has only the component type Overtakee and that it is executed on a single ECU. The source files `OvertakeeComponent.c`, `OvertakeeContainer.c`, `operationRepository.c`, `deviceAPI.c`, `MUML_DDS.c`, `MUML_DDSSupport.c`, and `MUML_DDSPlugin.c` are compiled one by one into the corresponding object files. These files are statically linked to the executable program.

We automatize the build process by using Make [Fel79]. Make is a build management tool that coordinates compiling and linking by running commands if a file changes. We generate a GNU *makefile* for each ECU that configures which commands are executed. The Appendix D.6 shows and describes the concrete generated makefile.

5.4 Tooling Implementation

We implement our automated software construction approach prototypically. We integrate the tooling within the Eclipse-based MechatronicUML Tool Suite [*DGB+14]. For quality assurance, we implement and use the same plugin `example.test` as for Modelica testing. This plugin provides a unit test that executes the platform-independent MechatronicUML implementation transformation for all MechatronicUML `example` plugins on which it depends. As a minimum set, we use the same six simple MechatronicUML examples as for the Modelica tests, which use message-based communication between components, simple and hierarchical RTSCs using entry and exit points, clock-based behavior, and operations. Thereby, we cover the main concepts of the platform-independent implementation that this chapter describes. The unit test executes the generated make file and fails if the transformation or the build process of any example fails. Otherwise, the test passes. Furthermore, we use during the development a ticket system for structuring the development and manual testing tasks. Different persons perform the development and testing tasks independently. Currently, we only provide a unit test for the platform-independent implementation and not for the platform-specific transformation. We create a reference implementation manually for an example before the implementation of the platform-specific transformation and check manually if the created code corresponds to our reference implementation.

We perform test executions of the compiled software artifacts manually using the "all states" coverage criteria [UPL12] for the RTSC of each discrete software component. We do this manually for the standalone platform-independent implementation and for the distributed platform-specific implementation. Furthermore, we coordinate the development using weekly meetings. We use a Jenkins continuous integration server [Ber12] to build all plugins automatically and to run the unit tests automatically if any registered MechatronicUML example plugin or any MechatronicUML plugin changes.

Figure 5.22 shows the main plugins and their dependencies. The plugin `codegen.componenttype.c.ui` implements the integration of the C code generation as an M2T transformation for MechatronicUML component types (cf. Section 5.3.2) into the user interface. It enables to start the transformation on a `.muml` file from the context menu. The user interface plugin uses the plugin `codegen.componenttype.c` to perform the transformation. This plugin implements the transformation as an M2T transformation using the tool Acceleo [Acc]. It uses the meta-model plugin `pim` to load MechatronicUML models, which are the source of each transformation. Furthermore, the transformation plugin `codegen.componenttype.c` uses the transformation plugin `psm.transformation` to create RTSCs for continuous components.

The plugin `pm.software.apilanguage.xtext` implements the textual editor for modeling signatures of APIs using the ApiMl (cf. Section 5.3.3.1). We implement textual editors by using Xtext [EB10]. The plugin `pm.software.apilanguage.xtext` uses the meta-model plugin `pm.software`. This plugin implements the ApiMl meta-model. Meta-model plugins use the Eclipse modeling framework [SBMP08] for defining the meta-model. Additionally, the plugin `psm.apimappinglanguage.xtext` implements the textual editor for modeling the access of continuous port instances to device APIs using the APIMappingMl (cf. Section 5.3.3.1). The

plugin `pm.software.apilanguage.xtext` uses the meta-model plugin `psm.portapimapping`. This plugin implements the APIMappingMl meta-model.

Furthermore, the plugin `container.codgen.c.ui` implements the integration of the deployment configuration generation (cf. Section 5.3.3.2), container C code generation (cf. Section 5.3.4.3), and middleware configuration generation (cf. Section 5.3.4.4) into the user interface.

The plugin `container.codgen.c.ui` uses the transformation plugin `psm.container.transformation` for generating deployment configurations and middleware configurations. The transformation plugin provides M2M transformations implemented using Query View Transformation Operational (QVTo) [QVT] The transformation plugin `psm.container.transformation` uses the meta-model plugin `psm.allocation` to load a system allocation and to generate a deployment configuration. Moreover, it uses the meta-model plugin `psm.deploymentconfigration` to create a corresponding deployment configuration model and it uses the 3rd-party meta-model plugin `org.opendds.modeling.model` [OPENDDS] to create a DDS configuration. The 3rd-party transformation plugin `org.opendds.modeling.sdk` uses DDS models to create a DDS IDL file.

The plugin `container.codgen.c.ui` uses the transformation plugin `container.codegen.c`. The plugin `container.codegen.c` implements the container C code generation (cf. Section 5.3.4.3) as an M2T transformation using the tool Acceleo [Acc]. The plugin uses the meta-model plugin `psm.deploymentconfiguration` to load a deployment configuration and the meta-model plugin `psm.portapimapping` to load the mappings of ports to device API calls.

5.5 Case Study

This section describes the evaluation of our approach for the automated software construction of distributed, connected CPSs. We perform the evaluation as a case study on the basis of the guidelines defined by Kitchenham, Pickard, and Pfleeger [KPP95] and Runeson and Höst [RH08]. We perform our evaluation using a variant of the autonomous cooperating overtaking example (cf. Figure 1.2) that Section 1.3 introduces.

5.5.1 Context and Cases

The aim of our case study is to evaluate the applicability and effectiveness of our automated software construction approach. Therefore, we evaluate the following questions:

1. Is it possible to generate C code for a cooperative CPS with a clear separation of platform-independent and platform-specific implementations?
2. Is it possible to generate C code for a cooperative CPS scenario that can be deployed on distributed, connected ECUs and runs correctly by means of state-based and message exchange behavior?
3. Does our automated software construction approach reduce the effort for software engineers to implement cooperative CPSs by means of handwritten and generated Lines of Code (LOC)?

Cooperating Overtaking Cars and Section Control: We perform the case study considering the communication between the overtaker, the section controller, and the overtakee in a cooperative overtaking scenario from the automotive domain. In contrast to our running example, both cars do not communicate with each other directly but via the section controller.

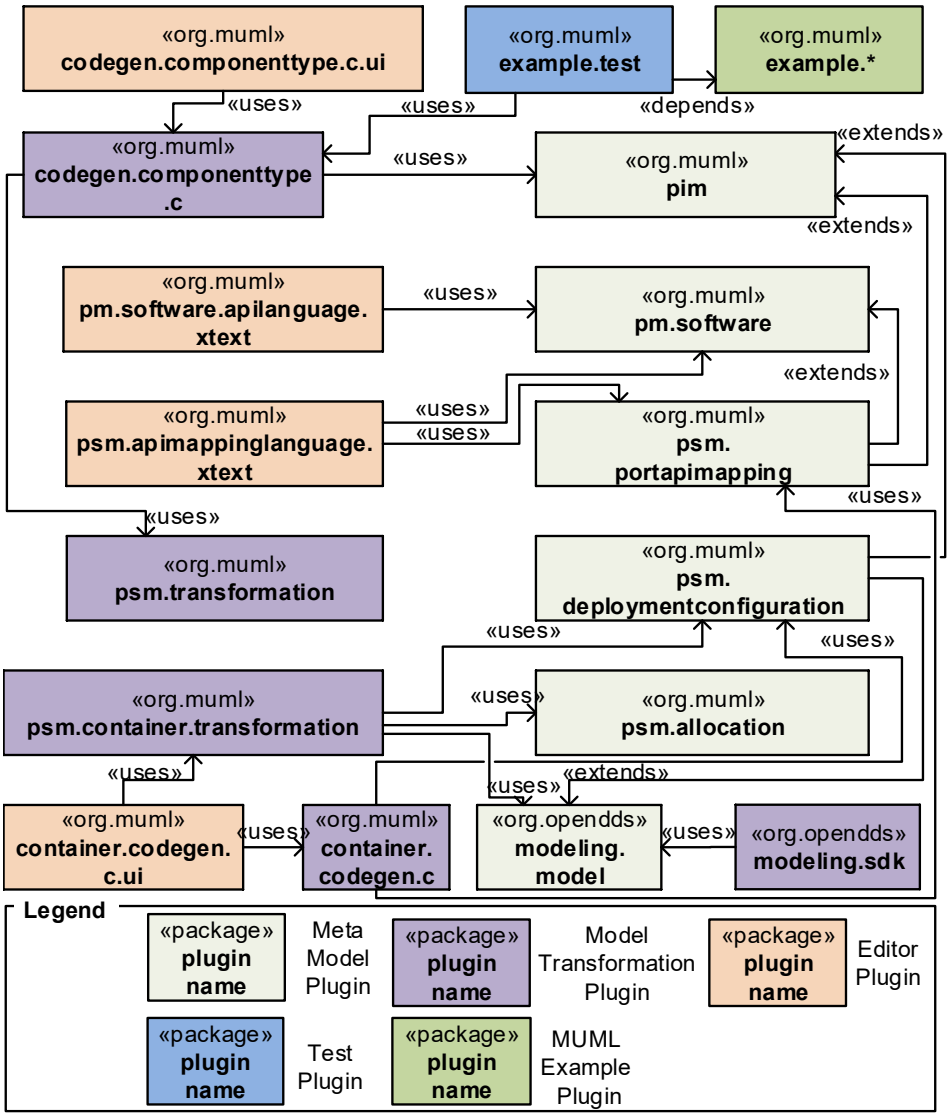


Figure 5.22: Plugins Implementing the Automated Model-Driven Software Construction of MechatronicUML Models

Both cars have the same component type. As a result, each car can become an overtakee or an overtaker.

Figure 5.23 shows the component instance configuration of the Cooperating Overtaking scenario. The configuration consists of the two Car structured component instances that are named yellow and red. A car component type consists of the discrete, atomic parts C1:Communicator, C2:Controller, and the continuous, atomic part C3:Distance. Both cars can send and receive overtaking requests by using the communicator component. The component C2:Controller interacts with the powertrain, which we omit in this example. The component C3:Distance accesses the distance sensor. Furthermore, the scenario consists of the SectionController component instance that is named C7. The section controller represents the role of a broker that acts as an intermediary between the overtaker and the overtakee and delegates the overtaking requests and responses. The Real-Time Coordination Protocols OvertakeRequest and Delegate OvertakeRequest are realized by the two ports between each car and the section controller. Figure E.15 and Figure E.16 in Appendix E.2 show the behavior

specification of these Real-Time Coordination Protocols. For simplicity, our scenario does not cover the usage of third party operation (cf. Section 5.1.1).

The platform model that we use in our scenario is quite simple. Each car and the section controller has an own ECU typed over a Raspberry Pi [Rasa] named ECURed, ECUYellow, and ECUSectionControl. Furthermore, the red car has a second Raspberry Pi ECU named ECURedDistance. Each ECU runs a Raspbian Debian Linux [Rasb]. We assume that either the target platform provides the RTI Connext DDS libraries [RTIa; RTIb] or engineers have to link the corresponding DDS libraries statically to the executable. Furthermore, we use the Wiring Pi API [Wir] of the Raspberry Pi for accessing the distance sensor via its GPIO pins. Listing E.6 in Appendix E.4 shows the required part of this device API for accessing the connected distance sensor using the ApiMl (cf. Section 5.3.3). Listing E.7 shows the specification for initializing and accessing the ultrasonic distance sensor pdistC1 that delivers the sensor values for the continuous software component Distance of the red car using the APIMappingMl (cf. Section 5.3.3). The specification of the port instance pdistC4 is similar as the yellow car uses the same operating system, libraries, hardware platform, and electrical wiring.

The allocation constraints are as follows: The component instances C1 and C2 are restricted to be allocated to the ECURed and the component instance C3 is restricted to be allocated to the ECURedDistance of the red car. The component instances C4, C5, and C6 are restricted to be allocated to the ECUYellow of the yellow car. Finally, the component instance C7 is restricted to be allocated to the ECUSectionControl of the section controller. Listing E.5 in Appendix E shows the formal allocation specification using the ASL.

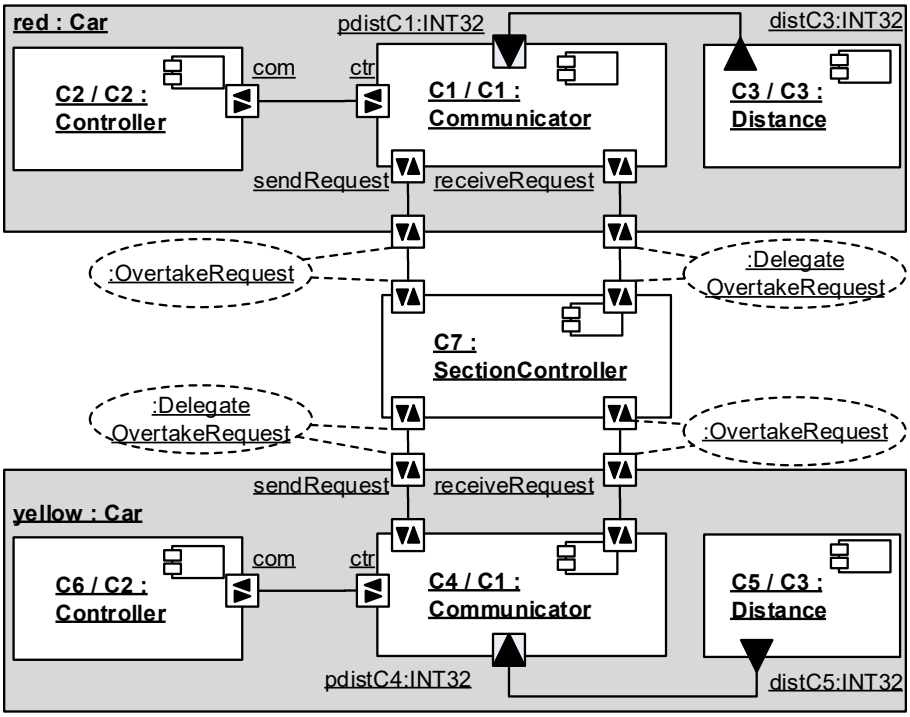


Figure 5.23: Cooperating Overtaking Cars Component Instance Configuration

5.5.2 Hypotheses

We set three evaluation hypotheses for this case study.

H1 The generated code from models specified using MechatronicUML has a clear separation of platform-independent and platform-specific code.
H2 The generated code from models specified using MechatronicUML can be executed correctly on distributed, connected ECUs by means of state-based and message exchange behavior.
H3 The development effort for implementing a cooperative CPS scenario using MechatronicUML is less than implementing it manually using C by means of writing LOC.

5.5.3 Analysis Procedure

The main input for our analysis procedure is the generated code. Therefore, we generate the software artifacts for evaluating our hypotheses on the basis of the MechatronicUML input models that realize the described case. The generated artifacts are: 1. The platform-independent code that represents the structure and behavior of component types. 2. The container model and on the basis of the model the corresponding container code and makefile 3. The OpenDDS model [OPENDDS] and on the basis of the model the corresponding IDL file. We generate DDS code on the basis of the IDL file using the tools of RTI [RTIa]. 4. The code that handles the device access.

Our analysis procedure consists of four parts. The first part is the generation itself. We require it to examine our hypotheses H1, H2, and H3. Additionally, we analyze the execution times of the transformation to examine if it can be executed in acceptable time. This analysis is not connected to any hypothesis but may be used in the future as a basis for a scalability analysis. The second part is a manual code review where we check the separation of platform-independent and platform-specific implementation within the generated code for analyzing our hypothesis H1. We define the separation of the term platform-independent and platform-specific for C code during performing the case study and after finishing the implementation of the transformation. Therefore, the manual review is necessary to check if we fulfill the conditions that Section 5.3.2.1 defines. The third part is a runtime analysis of the code by debugging the code and executing it on a local and distributed target platform for analyzing our hypothesis H2. We examine the execution runs using the coverage criteria “all states” [UPL12]. The fourth part of our analysis is comparing the size of the input models with the size of the C code by means of LOC that we generate by ourselves. The input model is stored as a normalized XML text file defined by the Eclipse XMI serializer and the C code is normalized using the standard Eclipse C code formatter. We use the tool cloc [CLOC] for counting the LOC of the C code and the XML model code.

We rate the first hypothesis as fulfilled if the C code that implements the functionality is platform-independent (cf. Section 5.3.2.1). We rate the second hypothesis as fulfilled if we can debug/log that each state becomes active at least once for each implemented role and we receive and consume all messages with the expected parameter values. Lastly, we rate the third hypothesis as fulfilled if the generated C code (header file, source file) that implements data types, message types, component types, component behaviors, and containers is at least twice (relative change = 100%) the size of the input model (manually created + generated). We do not take into account the LOC of the used library files into our rating. The tool cloc [CLOC] counts the physical lines of code (statements and brackets) without considering empty LOC and comments.

5.5.4 Preparation of the Data Collection

In preparation for the case study, we model the message types, the coordination protocols, the component types and their behavior, the hardware platform model, the allocation constraints, the Raspberry Pi API model, and the device access model manually. Furthermore, we create a component instance configuration that instantiates of each component part the maximum cardinality and calculate a feasible system allocation by using a model transformation.

We use the MechatronicUML model checking [GDHS15] to verify formally that the modeled Real-Time Coordination Protocols fulfill the safety properties and liveness properties shown in Appendix E.3. As a final model-based validation, we use the transformation to Modelica and perform a Model-in-the-Loop (MiL) simulation of the behavior of the component instance configuration. We perform a simulation run where we set the distance signal of the yellow car to the constant value `120` and where we set the distance signal of the red car to the alternating values `120` and `90` with a time interval of ten seconds. During the simulation run the red car becomes the overtakee.

5.5.5 Data Collection Procedure

We collect the data for our case study by using four different techniques that we describe in the following Sections 5.5.5.1-5.5.5.4. Firstly, we measure the execution times of the transformations. Secondly, we conduct a manual, static code review. Thirdly, we perform a runtime analysis and fourthly, we measure the development time by means of specifying LOC. We use the extended MechatronicUML Tool Suite [*DGB+14] for specifying all models and executing the automated software construction process. Section 5.4 describes our used extensions of the MechatronicUML Tool Suite.

5.5.5.1 Measure Execution Time for M2M and M2T Transformation

The data collection starts by creating the code artifacts and collecting the execution time information of the M2M and M2T transformations that we provide to realize our approach. We execute the M2M and M2T transformations ten times for a statistical analysis. We measure the execution times on a Lenovo Thinkpad with an Intel Core i7-3540M CPU, which runs at a speed of 2.9 GHz. The computer has 8GB DDR 3 RAM with a speed of 1600MHz and a Samsung 840 EVO hard disk with a capacity of 500GB. We use Windows 7 as the software platform for executing the model transformations and executing the code generations. Appendix F.3 describes the different tasks for executing the different transformations to generate the code, DDS artifacts, and to collect the data.

Table 5.1 shows the statistical relevant *locations* of the measured execution times. According to Devore, these *locations* "serve to characterize the data set and convey some of its salient features"[Dev15]. The first row shows the aspect of each column and the first column shows the corresponding location that each row presents. The statistics locations are based on the performance. The second column shows the execution times of the component type code generation executed by the Acceleo runtime engine. The generation takes between 1273-2691 milliseconds for our case. The third column shows the execution times of the deployment configuration model generation executed by the QVTo runtime engine. The generation takes between 25-186 milliseconds for our case. The fourth column shows the execution times of the OpenDDS model generation executed by the QVTo runtime engine. The generation takes between 125-296 milliseconds for our case. The fifth column shows the execution times of the container code generation executed by the Acceleo runtime engine. The generation takes 862-2348 milliseconds for our case. The sixth column shows the execution times of the device access code generation executed by the Acceleo runtime engine. The generation takes

565-1594 milliseconds for our case. The first transformation run is always an outlier. QVTo and Acceleo make some technical optimizations and compilations during the first run, which takes significant time. Acceleo also caches some results, which improves the performance after the first run. Table F.9 in Appendix F.4 shows the detailed measurement results of the different transformation tasks and transformation runs.

Table 5.1: Software Construction Transformation: Execution Time Measurement Descriptive Statistics

Location	Component Code	Deployment Model	OPENDDS Model	Container Code	Mapping Code
Average	1552.7 ms	55.4 ms	173.9 ms	1289.3 ms	865.1 ms
Minimum	1273 ms	25 ms	125 ms	862 ms	565 ms
1st Quartile	1301.75 ms	29.5 ms	143.25 ms	1060 ms	632 ms
Median	1356 ms	44 ms	157 ms	1166.5 ms	757 ms
3rd Quartile	1598 ms	52.25 ms	196.75 ms	1247.5 ms	924 ms
Maximum	2691 ms	186 ms	296 ms	2348 ms	1594 ms

5.5.5.2 Manual, Static Code Review

We check manually if the generated type code contains dependencies to platform-specific functions within the implementation of the state-based behavior for accessing timing information, network device access, or sensor and actuator access. If this is the case, we have to change the implementation if the component type code should be ported to another target platform.

The code review showed that all of the used regular functions but the `Clock_getTime` and `Clock_Reset` functions were either provided by our container implementation, or by our manually written, self-contained, platform-independent libraries.. The clock function within the header file clock.h is realized by a #define declaration, which means that the preprocessor will replace the calls by the provided implementation of the define itself. In our case, this are, e.g., `(systick_get_ms() - (aClock))`, which depends on the library time.h that is not supported by each embedded system. As a result, the compiled code depends on an operating system that supports the time.h library. Nevertheless, the time.h implementation is clearly separated from the behavior code and has to be implemented by software engineers manually. Anyway, the time.h code should be refactored in the future by replacing the #define statements by regular function calls. Furthermore, the code contains functions from the C standard library stdio for user output by using the `printf` function if it is compiled with the compiler flag -DDEBUG. Nevertheless, the include is separated in a special debug file and not mixed with the productive code.

5.5.5.3 Runtime Analysis

We create the executables by executing the generated make files (cf. Appendix D.6). The executables can be started by starting the command "./app" within each ECU folder. It is possible to run the overall system's behavior by starting the app command in separate terminal window or on different via Ethernet connected machines. For the second case, DDS requires that the machines share the same *subnetwork*. After starting the execution, we check manually if the state-based behavior and the message exchange work as expected by inspecting the printed debug output of the four executable programs. We use the debugger Nemiver [NEMIVER] on a Linux system for debugging the software by starting the command

"nemiver ./app". By stepping through the three programs, we check manually if the state-based behavior and the message exchange work as expected.

During the conformance test runs and debug runs [LY96; Gar05], we check manually that both cars communicate with each other and with the section controller as expected during the overtaking maneuver. We check if the cars behave safely in the case when the asked car allows the overtaking and in the case when the asked car rejects being overtaken. In detail, we check if the sent and received messages are equal and if only the intended receiver receives the correct message. Furthermore, we check if correct state changes happened. The observed behavior corresponds to the intended MechatronicUML behavior. We fulfill the coverage criteria "all states" [UPL12] during the manual testing.

5.5.5.4 Measure Development Effort

The data collection for measuring the development effort counts the LOC of the input model file and of the generated files. The XML code is normalized by using the standard Eclipse XMI serialization and the C code is normalized using the standard Eclipse C formatter. The data collection starts by splitting the input MechatronicUML model file ".muml" into several files. For each model aspect we create one XML file ".xml" so that cloc [CLOC] is able to detect them as XML files. Cloc has specific XML parsing rules to detect empty lines and comments. Thereby, we are able to measure the size of the input model for the following aspects (model categories): data types, message types, component types, behavior, and component instance configuration. Furthermore, we have already separate model files for the deployment configuration, the ApiMl, the APIMappingMl, and the DDS middleware configuration. The deployment configuration file and the DDS middleware configuration files also have to be renamed to the file extension name ".xml". The ApiMl and APIMappingMl syntax is close to C. We rename this files to ".c" so that cloc measures their size with the specific C rules. Afterward, we execute the tool cloc on all input model files and log the results for each file to a single log file. We also execute the tool cloc on the generated source C files, header C files, and the handwritten libraries. The results are printed to a separate log file.

Table 5.2 shows the LOC measurement results for the input model specified using the MechatronicUML and the output code implemented using C. We use the LOC metric to compare the development effort of the MechatronicUML input model and the generated code. The table is split into the upper development aspects part and lower analysis part.

In the upper part, Table 5.2 shows the development aspects of the system and the measured number of files and LOC. We distinguish between the following development aspects: data type, message type, component type, behavior, Component Instance Configuration (CIC), deployment, and device access. The columns 2-5 show the results for the MechatronicUML input model. We count the number of files and LOC of the XML file serialized using the XMI standard [ISO19503] for all aspects but the device access. We count the LOC of the device access using the concrete textual syntax specification (cf. Appendix C.3). We distinguish between manually created model parts and generated model parts. Manually created means that they are created manually within the MechatronicUML Tool Suite. Generated means they are created using M2M transformations within the MechatronicUML Tool Suite. We only count the LOC of the model that is used for the code generation. Model aspects that are specified during the model creation for further analysis and automation tasks are not considered in Table 5.2. Table F.10 in Appendix F shows the measurement results for these model parts. The columns 6-9 show the results for the generated C code. We distinguish between library code and generated code. Library code is code that is manually handwritten code, which is for every project the same. Therefore, it is only copied from the generation plugin to the destination folder. Generated code is created individually for each project based

on generation templates. We only count the LOC of the code that we created by our own M2T transformations. The metrics for the code that is generated for the middleware DDS based on the RTI DDS [RTIa] code generator are shown in Table F.10 in Appendix F.

In the lower part, Table 5.2 shows our analysis results. We calculate the individual sums for the number of files and LOC. Furthermore, we count the number of files of the input MechatronicUML model and the code required for implementing the case. The modeling aspects message type, component type, behavior, and CIC are stored within a single file. Furthermore, the data type aspect, deployment aspect, and the device access aspect are specified within a separate file. Overall, the MechatronicUML input model of the case is split up to four separate files. The generated code is split up to 46 files (12 library file, 34 generated files). The files split up to C header files and C source files. The MechatronicUML input model has 1404 LOC. The generated C code has 6381 LOC. Additionally, the libraries provide 1147 LOC. The relative change from the input model compared to the generated code is about 355%.

Table 5.2: Comparing Effort to Develop the Case Study Using LOC Metric: MechatronicUML Model vs. C Code Implementation

Development Aspect	**MechatronicUML Model**				**C Code Implementation**			
	Manually Created		Generated		Library		Generated	
	Files	LOC	Files	LOC	Files	LOC	Files	LOC
Data Type	0	0	1	14	1	22	0	0
Message Type	1	21	0	0	0	0	1	34
Component Type	1	241	0	0	3	49	4	441
Behavior	1	772	0	0	0	0	3	1194
CIC	0	0	1	150	8	1076	24	4584
Deployment	0	0	1	167				
Device Access	1	39	0	0	0	0	2	28
Analysis								
Sum per Category	2	1073	3	331	12	1147	34	6381
Sum	4 Files		1404 LOC		46 Files		7528 LOC	
Relative Change from Input Model to Generated Code	$\frac{(6381-1404)*100}{1404} \approx 355\%$							

5.5.6 Interpreting the Results

We check our first hypothesis H1 that the generated code from models specified using MechatronicUML has a clear separation of platform-independent code and platform-specific code by generating platform-independent component code and conducting a manual code review. The review showed that the platform-independent code dependents on functions provided by the container for sending and receiving messages and the dedicated interface for accessing the distance sensor. Only accessing timing information was not completely separated as discussed within Section 5.5.5.2. Nevertheless, this issue is a result of the manual implementation of the header file time.h and not a conceptual issue. It can be fixed in the future. Overall, we rate our first evaluation hypothesis H1 as fulfilled.

We check our second hypothesis H2 that the generated files from models specified using MechatronicUML can be executed correctly on distributed, connected ECUs by means of state-based and message exchange behavior by using our automated software construction approach and performing a runtime analysis. The execution and debugging of our case study

show that we are able to execute the code locally on using multiple terminals and distributed using multiple via Ethernet and a router connected ECUs. The system's behavior works as designated. We use the coverage criteria "all states" during the test runs [UPL12]. Additionally, we check what happened when we cut the connection during the overtaking. In this case, the timeout transitions fired correctly. Currently, the state-based behavior does not react to the network status, e.g., it tries to send a message even though no network connection is established. This is a result of the current implementation limitations and not a conceptual limitation as the state-based behavior has the conceptual ability to access the port connection status. Summarizing, we were able to log the state-based behavior for different paths for each implemented role from the initial state back again to the initial state and we receive and consume all messages with the correct parameter values. The observed behavior works correctly. Therefore, we rate our second evaluation hypothesis H2 as fulfilled.

We check our third hypothesis H3 that the development effort for implementing a cooperative CPS scenario using MechatronicUML is less than implementing it manually using C code by counting and comparing LOC of the MechatronicUML model and the C implementation. Furthermore, we measure the execution time of the transformations that are part of the automated software construction approach. The analysis shows that all transformations can be executed within at most three seconds for our case. The percentage increase from the whole MechatronicUML input model to the generated C code is 355%, which is clearly larger than our evaluation criteria that the generated code has to be at least twice the size of the input model (relative change = 100%). Therefore, we rate also our third evaluation hypothesis H3 as fulfilled. Nevertheless, the hypothesis is not fulfilled for each development aspect. The relative change from the message type model to its code correspondent is $\approx 62\%$, from component type model to component code $\approx 83\%$, and from behavior to behavior code is $\approx 55\%$. Anyway, model analyses like (formal) model checking or MiL simulation require and justify using model-driven development approaches.

5.5.7 Threats to Validity

Empirical case studies have limitations and the results rely to a large extent on the research design. Each empirical study has to face threats to validity [RH08]. We use the systematics of Runeson and Höst for analyzing and reporting these threats by splitting the threats into the construct validity, internal validity, external validity, and reliability. Furthermore, we report the corrective actions that were performed to reduce the impact of the threats.

5.5.7.1 Construct validity

The case study was designed and conducted by the same researcher and the same student that developed the approach. Therefore, a threat is that the construction of the case study by the same people has a bias towards the already developed approach. The case study design and the research questions have been discussed with other researchers as a corrective action. LOC as a valid metric for measuring the effort of a task is a certain threat. Especially, if the compared languages are not the same. Additionally, we faced the challenge to find a metric for measuring the complexity of creating our input model. Selecting the LOC of the Extensible Markup Language (XML) input file may not reflect the complexity of creating that file because software engineers use engineering tools. These engineering tools only serialize the models as XML files and software engineers do not directly write this code within a text editor. The size of the XML depends on the number of used modeling elements. Nevertheless, measuring LOC is a metric for measuring the effort that is used often and has a long history [AG83]. It can be measured automatically and delivers reproducible results. Lind and Vairavan showed that

LOCs reflect the programming effort [LV89]. We discussed to measure the effort for editing model elements as a metric. This metric may reflect the required modeling effort in a better way but we discarded this because the different parts of the input model are heterogeneous and thereby the modeling effort to create them differs a lot. We need a clear definition of what an atomic model element is, e.g., a state transition or a state transition guard/action and how much modeling effort an engineer needs in average for creating it. It is hardly possible to get these numbers without performing extensive user studies. User studies require a lot of effort, the results depend on the knowledge of the user, and the results are difficult to reproduce. Durisic *et al.* report that using simple metrics, like the number of elements, helps in practice to predict the effort even though that they have certain threats to the construct validity [DSTH17]. Furthermore, the coverage criteria "all transitions" or "all transition-pairs" for testing the state-based behavior provide a more reliable result than "all states" but result in a higher effort [BLW04].

5.5.7.2 Internal validity

Currently, we do not judge about a causal relation between the model size and the generation time or the size of the generated code. Our input model consists of different modeling aspects and each aspect it transformed by a different transformation. We could state that there is a relation between the code size and the number of transitions within the RTSC as we generate new statements for each transition. However, we have a threat to the internal validity for the considered sums because we have made no investigation how large is the distribution on a percentage basis. The measured metrics just give a first impression of the relation between the input model and the output model. We have split up measuring the metrics for different model aspects as a corrective action to this threat.

5.5.7.3 External validity

The conducted empirical evaluation that uses a case study that considers a new approach cannot ensure external validity. Nevertheless, the case study design may help another researcher to define a systematic guideline how to evaluate model-driven development approaches. Especially, defining standardized non-trivial modeling cases in the field of connected CPS for different domains (automotive, productions systems, railway, ...) would help to compare modeling approaches with each other.

5.5.7.4 Reliability

The manual code review depends on the subjective opinion of the reviewer. In Section 5.3.2 we define for this thesis and especially for the reviewer who performs the manual code review how to rate C code as platform-independent or as platform-specific. This definition is our corrective action for minimizing the subjective opinion of the reviewer. A further threat to the reliability is that the investigated case is implemented using a research tool, which may not be available in the future. Therefore, we describe the modeled protocols and components in detail in Appendix E. Furthermore, counting the LOC depends on the formatting of the code and the used language metrics. We use the standard Eclipse formatter and serialization algorithm to get reproducible results. Furthermore, we describe which language metric we use by executing cloc. The execution time depends on the computer that executes the transformation. Therefore, we describe the characteristics of the used computer, perform ten transformation runs, calculate the average, and calculate the metrics for a box and whiskers plot [Dev15]. The interpretation of the results could vary as it is a subjective interpretation of the researcher and depends on the personal preferences and lessons learned during the development of CPSs. We discussed

the interpretation of the results with several researchers and master students from our software engineering research group.

5.6 Limitations

Our approach for the automated software construction underlies the following limitations:

1. Currently, the generation of RTOS scheduling artifacts is not part of the automated software construction approach that this thesis defines. Runnables, threads, and tasks as well as the definition of its specific real-time properties (period, priority, deadline, Worst-Case Execution Time (WCET)) have to be defined to execute the generated implementation on an RTOS. We present in [*GPS17] first ideas of how to generate these artifacts automatically. The challenge is to generate the real-time properties automatically without violating the model checking assumptions [Dzi17], without losing too much performance, and without relying on strong modeling restrictions.
2. We do not support the generation of *urgent transitions* [GDHS15] that have to be executed immediately when the condition is logically fulfilled [*DPP+16]. Currently, we do not control the scheduling of tasks and execute the behavior of components using an arbitrary but fixed order. Therefore, we cannot guarantee at which time a component is scheduled/executed. As a result, the firing of a transition may be delayed due to the execution of another component. This corresponds to the behavior of *non-urgent transitions*.
3. The port behavior does not react to the status of the physical network connection (port status). Using the component context pattern (cf. Appendix D.1) enables the component to access the current status of the physical network. Currently, we do not have defined how the port behavior that realized an Real-Time Coordination Protocol (RTCP) should behave in the case of a port instance that has the status INACTIVE, UNCONNECTED, or CONNECTIONLOST. It is possible to change the port behavior in the future, e.g., to block the execution or to execute a fallback behavior.
4. We do not support the generation of code for reconfigurable software components [HB13; Hei15; SHG16]. It is up to future research how architectural reconfigurations can be supported by our component-container-based architecture. Containers that handle the lifecycle of component instances have to interact with the MechatronicUML *reconfiguration controller* [Hei15]. Furthermore, we do not support optional ports and multi ports [EHH+13]. Currently, port instances are not activated and deactivated during runtime. Furthermore, we do not support multi ports because they use subport behavior for communicating with a flexible number of communication partners.
5. We do not check formally that the generated platform-specific code fulfills the safety properties or is a valid refinement of the verified models. Model checking of platform-specific models [BC11] or automata-based refinement checking [HBDS15] are promising approaches for verifying the properties for the resulting code. Furthermore, static code analysis [BCC+03] and testing [KBE16] approaches are also promising for validating safety properties on the code level. Nevertheless, we implemented and tested our code generator carefully as described in Section 5.4.
6. The initialization and the termination of devices are currently not supported by the platform-specific code generation. This is a technical limitation and can be integrated into the tooling in the future.
7. The deployment configuration currently supports only a shared memory port instance configuration and a DDS port instance configuration for communication. Realizing a

cooperative overtaking with cars requires a special communication channel for vehicle applications [HL09]. This is a technical limitation and can be extended in the future if required.

8. The generated code is not optimized considering the memory footprint on the target platform. This is a technical limitation and can be optimized in the future.
9. Currently, we only provide a C code generator. C is a very common language in the domain of embedded systems. C-compilers are available for the most hardware platforms. Nevertheless, languages like ADA in combination with the Ravenscar profile [Rog09] or new languages like RUST [MK14] may fit better than C for the properties of safety-critical systems. This is a technical limitation. Our concepts can be used to provide code generators for other languages in the future.

5.7 Related Work

This section aims to provide an overview of existing approaches for developing distributed CPSs. We also consider approaches for embedded systems and real-time systems. The approaches are compared to the MechatronicUML approach that this chapter describes. The related work has a special focus on component-based development approaches that are strongly related to MechatronicUML. Pop *et al.* provide a survey of these approaches [PHH+14; HPB+10]. We distinguish between approaches that focus on the engineering of components representing functional concerns, i.e., the application layer, and approaches that focus on the engineering of components representing technical concerns, .i.e., the middleware layer. Both sets are not disjoint, some approaches like ProCom [BCC+08; CFMS10], have a strong focus on designing functional concerns but also provide an own middleware/runtime environment. Other approaches, like the Flex-eWare component model (FCM) [RCGT09; JJK+12] and its middleware [FRGT10] reuse mainly existing component models with slight changes and focus on providing a middleware layer as an abstraction to the underlying platform.

We describe for each approach the following four general criteria: (C1a) Existence of a component model with the support of (C1a) hierarchical nesting and the support of (C1b) communication styles; (C2) Existence of a formal (real-time) behavior model; (C3) Existence of a hardware/device model; (C4) Support of verification and validation.

Furthermore, we discuss the following two platform-specific modeling criteria and four code generation criteria in comparison to our approach: (C5a) Coupling with sensors and actuators; (C5b) Modeling real-time attributes (e.g., period, deadline) and scheduling; (C6a) Code generation of functional concerns (e.g., component interfaces / behavior); (C6b) Code generation of technical concerns (e.g., lifecycle management, communication); (C6c) Separation of the platform-independent code from the platform-specific code; (C6d) Providing a runtime container or a runtime environment.

5.7.1 Component-based Application Engineering

In the following, we examine the component-based approaches that focus on the development of the application layer. We only consider approaches that support the realization of distributed applications [PHH+14], i.e., applications that run distributed on multiple ECUs and communicate with each other to realize their functionality.

SOFA HI [PWT+08; HPM+10] is an extension of the SOFA 2.0 component model [PBH06] for spacecraft onboard systems. These systems are reconfigurable embedded real-time systems. SOFA HI supports hierarchical component models (C1a) and different communication styles (request-reply calls, message passing, etc.) (C1b). It uses so-called *behavior protocols* [PV02] to model behavior formally (C2). Behavior protocols describe a verifiable finite state space

without considering timing behavior. SOFA HI does neither support the modeling of a hardware platform (C3) nor an allocation. The verification is done by examining the state space [PV02] (C4). SOFA HI supports method invocation as a communication style. The coupling to APIs of sensors and actuators can be represented explicitly by a method invocation for a specific platform (C5a). Furthermore, an active component has real-time properties like a priority, a period, and a deadline [HPM+10] (C5b). During the software construction, the functional behavior is implemented manually in C. No code generator for implementing functional concerns is provided currently (C6a). Code for the technical concerns is generated with respect to the communication and the binding of components [Bur06a] (C6b). The engineer has to separate platform-independent components from platform-specific components manually. The component model does not distinguish between PIM and PSM on the functional layer (C6c). A SOFA HI component may have several so called controllers for realizing different technical concerns (C6d). A controller manages a certain technical concern of a normal component. A controller is considered during the code generation like a normal component. For managing the lifecycle a LifecycleController is created, which is in charge of controlling the component lifecycle, a BindingController is in charge of managing the connections to other components. A component is deployed to a container that provides several services to the component. SOFA-HI provides a general abstraction layer that components should use. This abstraction layer has to be provided for each target platform (C6d). In contrast to our approach, this abstraction layer is static for each platform and is not customized for each component type, e.g., omitting unused functions or using different communication middlewares for different ports. As a result, SOFA HI is less flexible compared to our approach.

The PROGRESS component model ProCom [BCC+08; CFMS10] is designed for embedded real-time systems. Actually, the ProCom component model distinguishes between the two component models ProSys and ProSave. The ProSys component model is comparable to the MechatronicUML component model and defines hierarchical, active components (C1a) that communicate via message passing (C1b). ProSys is used on a higher level than ProSafe for defining software architecture. The ProSafe component model defines passive components that have ports for specifying control flow and data flow. The ProSafe component model is an ancestor of the SaveCCM component model [HÅCT04; CHP06; ÅCF+07]. A passive component is comparable to a MechatronicUML operation (cf. Section 5.1.1), a MechatronicUML API command (cf. Section 5.3.3.1), or the representation of the MechatronicUML components on the C-level as C-functions. The behavior of ProCom is defined by a variant of timed automata [VSC+09] (C2). ProCom supports a simple hardware platform model that contains physical nodes and network links [CFMS10] (C3). Åkerholm *et al.* propose to use general tools for model checking timed automata as the semantics is defined by timed automata [ÅCF+07] (C4). Modeling the platform-specific device access is done by defining platform-specific device mappings [Led15] (C5a). ProCom supports the modeling of periods, priorities, and deadlines (C5b). Borde and Carlson generate from ProCom C code for component interfaces and their behavior [BC11] (C6a). Additionally, Lednicki generates glue code to devices [Led15] (C6b). In an earlier version, Lednicki, Crnković, and Zagar present a code synthesis algorithm that weaves existing implementations of software components together [LCZ12] (C6b). The generated application code, without the glue code, is independent of an operating system or hardware (C6c). ProCom provides a ProCom specific runtime environment called runnable virtual node (RVN) [IS12] (C6d). An RVN has an own scheduler, provides an API to the ProSys component, and has an API to a middleware that realizes the inter RVN communication. The deployment of ProCom components is done by mapping ProSys components and its required ProSafe components to an RVN [ICSK14]. An RVN is allocated to an ECU. A platform-specific implementation of the RVN must exist. Furthermore, Borde and Carlson define a platform-specific representation of their generated code using UPPAAL

models and define verification properties to verify that certain required properties are still fulfilled on the code level [BC11]. In contrast to our component-container-based architecture, RVNs are general-purpose containers. We generate specific containers for each component type that only require specific middleware API functions for the required communication. In our approach, a container can rely on multiple middlewares for different purposes or does not need a middleware at all if the components do not need an intersystem communication. Thereby, we are more flexible by using our deployment configuration (cf. Section 5.3.3.2) for the generation of the platform-specific implementation (cf. Section 5.3.4.3).

The DEECo (Distributed Emergent Ensembles of Components) component model [MKM+16; BGH+13] is designed for cooperative CPSs. Components communicate by exchanging typed data structures, so-called knowledge, in a fixed time interval if the components build a so called *ensemble*. An ensemble is a flat component composition (C1a) that is constructed dynamically during runtime if certain constraints are fulfilled. DEECo only specifies when knowledge is exchanged via publish-subscribe (C1b). It does not model the behavior of the functions that a component provides and that can be invoke based on a timer event or a data change event (C2). Furthermore, DEECo does not provide a hardware or a platform model (C3). The overall system's behavior is verified either by verifying certain properties on the code level [BGH+13] or by performing a networked-based simulation to validate the timing behavior during runtime [MKM+16] (C4). Model checking is not used because DEECO does not specify protocols, which form a state space that a model checker can examine. The access to sensors and actuators is modeled via function calls. The functions itself are platform-specific and have to be implemented for each platform manually (C5a). A set of processes with priority, period, and deadline information belong to a DEECO component (C5b). DEECo generates a C++ implementation of the component definitions as classes and the exchanged knowledge as structures. The implementation of the functions has to be done manually (C6a). This implementation requires the generic DEECo runtime environment (C6b). DEECo provides no mechanism for separating platform-independent code from platform-specific code. The implementation of the component and the knowledge definition itself are platform-independent but not the used functions (C6c). A DEECo component can be deployed on systems that provide the DEECO runtime environment. The runtime environment has to be implemented for each used operating system manually. It handles the release of processes and the detection which communication connections are established with other components. It also handles the communication itself (C6d). In contrast to MechatronicUML, the communication of different ports, attributes, or even knowledge cannot be handled by different communication protocols or middlewares. The generic DEECo runtime environment is less flexible than our component type specific generated containers. The communication QoS cannot be configured based on the components requirements. Furthermore, the portability is more difficult because the functionality has to be implemented for each platform if it depends on sensors or actuators. Developers have to take care by themselves to design a wise separation.

The CHESS modeling language [CSCS13] is a UML-based component model for real-time embedded systems. It is implemented using the UML profile mechanism [UML]. The CHESS modeling language is based on the three UML profiles MARTE [MARTE], SysML [SysML], and EAST-ADL [EAST-ADL]. Furthermore, it uses the UML action language ALF [ALF]. Ciccozzi *et al.* model the software architecture by using UML/SysML component diagrams (C1a). Communication is supported by using function calls (request-reply) and message passing (C1b). The software behavior is modeled by tailored UML state machines that are using MARTE stereotypes for modeling real-time, UML activities, and ALF actions (C2). CHESS uses the UML profile MARTE [MARTE] for modeling resources, the hardware platform, and the allocation [Cic13]. CHESS focuses on dependability analysis [GP11] for validation but as it uses UML models the approach of Knapp, Merz, and Rauh can be used to verify the

UML behavior [KMR02] without considering the used stereotypes. MARTE enables to model and use concrete platform-specific API commands to access sensors and actuators by using *SWAccessServices* (C5a). MARTE offers also a rich set of real-time attributes, like period, deadline, priority, time slot, and offset (C5b). CHESS provides a code generation [CS12; CCM+12; CSCS13; CCS13] for components and behavior specification (C6a). It does not generate infrastructure code that manages the component lifecycle, the resources, and the connection status of ports if it is not modeled explicitly (C6b). CHESS does not differ between platform-independent and platform-specific software (C6C). Everything has to be modeled as a component explicitly. Therefore, CHESS does not rely on a runtime environment if it is not modeled explicitly within the software architecture as a component (C6d). In contrast to our approach, CHESS does not separate platform-independent and platform-specific parts. CHESS does not provide the ability to generate containers automatically that realize the technical concerns of components and its interactions. The modeler has to provide these components manually. The code generation of CHESS adds traceability links to the code, which we currently do not support. The traceability links enable the analysis of extra-functional properties during runtime and using this information to improve the software design.

AutoFocus3 [AVT+15] is a component model for distributed embedded systems. It supports a hierarchical component model (C1a) and message passing for communication between components (C1b). Behavior is specified using automata, tables, and imperative code (C2). The hardware platform is described by ECUs, actuators, and sensors that are connected via a bus (C3). Formal verification is supported via model checking using NUXMV [CCD+14] and validation by a simulation of the software (C4). Ports of software components can be mapped to hardware sensors or actuators to define a coupling (C5a). Real-time attributes and a scheduling can be synthesized [VS13] (C5b). AutoFocus3 generates functional concerns to an intermediate representation called C0, which can be translated to C [Höl09] (C6a). Another generator has to generate the technical concerns for a specific platform. It is not defined which functionality the platform-specific code has to provide. It is stated that the generator must uphold the semantics of the model (C6b). The separation of the platform-independent code from the platform-specific code is fulfilled as long as the imperative code within the model does not use any platform-specific functions (C6c). An RT CORBA [CORBAe] middleware generation is described in [AKPM05] (C6d). In contrast to MechatronicUML, AutoFocus3 relies on a general-purpose container/runtime environment on each platform and does not provide a component-individual deployment configuration. Therefore, the container is less flexible and may have a larger memory footprint. Furthermore, real-time functional behavior cannot be modeled by the automata formalism. The semantics of the sensor and actuator access and the reuse of existing libraries is not clearly defined.

5.7.2 Component-based Middleware Engineering and Deployment Frameworks/Specifications

The flex-eWare component model (FCM) [RCGT09; JJK+12] defines a component model as a UML profile for distributed, embedded, real-time systems. FCM is used for specifying functional concerns. It unifies the Fractal component model [MS08] with is an extended CORBA component model (CCM) [CORBA]. FCM provides a hierarchical component model, like the UML [UML] and offers several synchronous and asynchronous communication styles (C1). The definition of functional real-time behavior (C2) and its validation and verification (C4) is not in the scope of FCM. The eC3M [FRGT10](embedded component container middleware) defines a component-container architecture for realizing the technical concerns of the FCM. Radermacher *et al.* propose to use the hardware resource model of MARTE [MARTE] to define the hardware/platform model (C3). The coupling of software components to sensors

and actuators can be defined by normal ports. A coupling of software components to hardware devices is not addressed explicitly but may be realized by using MARTE stereotypes (C5a). Real-time attributes are also modeled by using MARTE stereotypes (C5b). The code generation of components and its containers is conceptually close to the generation that this chapter presents (C6a, C6b) because eC3M generates component-specific containers based on the allocation and the used context objects. It distinguishes between a component type, a component instance, and a component implementation. eC3M separates the platform-independent code from the platform-specific code to a great extent but it does not provide any abstraction layer for accessing platform-specific device APIs (C6c). eC3M provides a runtime container for a specific component. It may be composed of different connectors to realize the technical concerns of the communication (C6d). In contrast to MechatronicUML, eC3M does not consider the role of sensors and actuators to separate the platform-specific concerns from the platform-independent concerns. Furthermore, it does not cover the behavior part of the software architecture and its requirements for the implementation of the communication by means of QoS. Verified safety-critical properties may be violated by the deployment if the assumptions of the model checking are not fulfilled.

MyCCM-HI [BHP09; BFHP09] is a framework for reconfigurable embedded real-time systems. It is based on the Architecture Analysis and Design Language (AADL) [FG12] component model and the lightweight CORBA component model [CORBA]. It supports a hierarchical component model during the design-time (C1a) and component parameters (so-called attributes) for defining different component configurations. A component has, like in MechatronicUML, ports for sending and receiving asynchronous events, and a required interface for defining operation dependencies. Additionally, it can provide callback operations (C1b). Behavior is not modeled but provided as functional source code (C2). The hardware model is described using the AADL [FG12] (C3). MyCCM-HI offers verification and validation of timing behavior of task scheduling and not of functional concerns (C4). It uses AADL for the coupling of software and hardware (C5a) and describing the real-time attributes of tasks and processes [FG12] (C5b). The code generation [DHPZ08; DPK08] generates code stubs for the software architecture of the AADL source model (C6a). MyCCM-HI generates technical code for synchronous communication between components by means of a component wrapper, synchronization code for shared data access within the component implementation, and mode switch management code [BFHP09] (C6b). The separation of the platform-independent code from the platform-specific code is not in the scope of MyCCM-HI (C6c). MyCCM-HI uses the CORBA implementation for distributed communication. Therefore, MyCCM-HI depends on CORBA containers [CORBA] (C6d). In contrast, we generate a platform-specific component-container architecture with less overhead compared to the general-purpose CORBA containers. Our functional application code is separated by specific containers from the underlying middleware. Lastly, we configure the QoS of the communication on the basis of safety-critical application requirements. MyCCM-HI cannot provide such a configuration due to the lack of information on the application requirements.

MICOBS (Multi-platform multI-model COmponent-Based Software development framework) [PPK+11] defines a component model for the deployment of component-based systems. It supports multiple communication styles and hierarchical component models (C1). Behavior models are not considered (C2) as MICOBS focuses on the integration and composition of components from heterogeneous component models. The platform can be modeled by describing the deployment target (C3). The operating system, its API, the processor architecture, its instruction set, and the ECU itself can be specified. Network connections, sensors, and actuators are not considered. MICOBS allows the specification of architectural properties that can be verified formally (C4). The coupling to software that accesses sensors and actuators has to be described by using synchronous ports (C5a). A connection to the platform-specific

component is required for each platform. Modeling real-time attributes is not addressed but user specific analysis can be added to the framework (C5b). The framework offers the ability to provide M2M and M2T transformations to generate code for functional concerns (C6a) and technical concerns (C6b). The functional generation is based on a so-called application-oriented domain, which is comparable to a component instance configuration. The technical generation is based on a so-called implementation-oriented domain that is comparable to a deployment configuration. The shown case-study generates some code for realizing adapters to MyCCM components [BHP09] and EDROOM components [SPG+06]. The component developer has to take care that the platform-independent parts and the platform-specific parts are separated (C6c). The separation of the platform-independent code and the platform-specific code can be done by using the application-oriented and implementation-oriented domain correctly. A container can be generated by the implementation-oriented domain (C6d). MICOBS focuses on providing a framework for the deployment of heterogeneous component models using heterogeneous middlewares and operating systems. In contrast, MechatronicUML focuses on the deployment of MechatronicUML components. The MICOBS component model could be an alternative solution to reach this target, instead of providing an own deployment configuration model and transformations. Nevertheless, MICOBS has not been actively maintained since 2013 and would add another dimension of complexity by using it during the deployment without providing a significant benefit when deploying MechatronicUML models.

OMG's Deployment & Configuration (D&C) specification [OMG06] defines how to deploy and configure UML [UML] components. It does not target embedded and real-time system directly but as it is implemented as a UML profile it can be extended with corresponding concepts easily. The UML component model defines a hierarchical component model and supports several communication styles [UML] (C1a/b). Behavior can be defined by state machines, activities, or opaque behavior (C2). Clocks are not first class entities but timing behavior can be specified by using time events or timing elements from the MARTE specification [MARTE]. The hardware model of MARTE can also be used (C3). Model checking can be used to verify UML behavior [KMR02] (C4). The D&C specification does not specify how to couple regular software components to components that access sensors and actuators. Such a coupling has to be modeled explicitly by using regular components (C5a). Modeling real-time attributes, like periods and deadlines, are not considered (C5b). Also, the generation of the component implementation is not in the scope of the specification (C6a/b). The D&C specification distinguishes between a PIM and a PSM (C6c). The PIM describes the functional concepts, like that a network communication, has to be provided. The PSM describes how the network communication has to be provided, e.g., by using CORBA [CORBA]. The PIM definition is less restrict than our definition of a PIM because in D&C a component on the PIM level can already have dependencies to specific operating systems and their libraries. The D&C specification defines a set of managers that defines the execution environment (C6d). In contrast to our approach, the D&C specification defines a framework for the deployment and the configuration of UML components, where we define a concrete realization. The focus of the D&C specification is the installation, configuration, planning, preparation, and launch of components during runtime. In contrast, MechatronicUML performs these tasks offline or during the initialization of an ECU to meet the requirements of real-time, embedded systems. Nevertheless, the concepts that this chapter presents may be mapped to a D&C specific representation in the future if a target provides a corresponding runtime environment and requires such a description.

Autosar [AUTOSAR14c] defines an architecture description as an industry standard for automotive systems. Autosar supports hierarchical components and several communication styles (sender–receiver, client–server, trigger event) [AUTOSAR12a] (C1). The behavior

specification is not part of the Autosar standard (C2). The hardware platform including ECUs, sensors, and actuators can be specified using the system template [AUTOSAR12c] (C3). Validation is supported on the architectural level, without considering functional behavior. Software components that communicate synchronously with other components can be used to model the coupling to sensors and actuators. Real-time properties are supported for so-called runnables and tasks [AUTOSAR12b] (C4). The generation of code is not in the scope of the Autosar specification. Approaches like [Mey15] generate configuration files, which are used to generate Autosar stubs and specific code for configuring the runtime environment (C6a, C6b). Autosar separates the functional code from the platform-specific part by specifying the Autosar architecture (C6c). The functional concerns are deployed to the application layer [AUTOSAR14b]. An Autosar runtime environment (RTE) [AUTOSAR14e] delegates and adapts the service requests of components to the Autosar operating system or underlying services, like the virtual functional bus for realizing the communication between components (C6d). In contrast to our approach, the Autosar basis software [AUTOSAR14a] has to run on each ECU where an Autosar component should be deployed. Furthermore, Autosar requires an Autosar conform operating system. This requirement does not hold for CPSs in general but only for CPSs from the automotive domain, i.e., cars. The RTE can be configured but cannot implement missing functions by itself, like our container for MechatronicUML. Therefore, the RTE is less flexible compared to our approach. However, we may support the deployment to the Autosar RTE in the future by adding corresponding connectors to our container. The MechatronicUML component type code does not need to be changed to run on top of Autosar.

RoboCop [MC04; Maa05] is a component model for consumer devices. It targets the development of the middleware layer. Components have required and provided interfaces and define 3rd-party dependencies and dynamic bindings (C1). The development of the application layer, its behavior (C2), and its validation (C4) is not in the scope of RoboCop (C2). RoboCop also does not define a hardware model (C3). The coupling to sensors and actuators has to be done by using the normal interfaces (C5a). Real-time properties are provided by the resource management framework (C5b). Code generation (C6a, C6b), and the separation of the platform-independent software from the platform-specific software (C6c) is out of the scope of RoboCop. RoboCop defines that a runtime environment that offers specific services has to be implemented for each target environment (C6d) to realize technical concerns. The runtime environment manages the creation of components during runtime. RoboCop provides no abstraction from the underlying platform. In contrast, our proposed component-container-based architecture separates functional from technical concerns clearly. As a result, we do not have to reimplement the functional concerns if we deploy a component to another platform. The runtime environment is less flexible than our generated container as it does not adapt to the requirements of the deployed components.

DREMS [BDO+15] defines a component model for distributed, embedded, real-time systems. It targets the deployment of component-based systems. It defines a flat component model that supports several communication styles (C1). Each component has a component implementation. The behavior definition of functional concerns is not in the scope of DREMS (C2). DREMS defines a flat hardware platform model without considering sensors and actuators explicitly (C3). Validations and analysis can be performed on the architectural aspects, without considering the business logic (C4). Sensors and actuators can be accessed via ports. The couplings of sensors and actuators are represented as software components (C5a). Real-time attributes can be associated with software components and can be analyzed (C5b). The code generation of components and its containers is conceptually close to the generation that this chapter presents (C6a, C6b) because DREMS generates component-specific containers. The separation of the platform-independent implementation and the platform-specific implementation is out

of the scope of the approach (C6c). DREMS defines its own runtime environment in the form of an embedded RTOS and a specific DREMS middleware (C6d). In contrast to our approach, DREMS has no concept of component types and component instances. It only requires that each component implementation has a component specification. In DREMS it is not possible that multiple instances of the same component type share their code on the same ECU. We share the code and reduce the memory footprint on an ECU. The deployment of DREMS components and containers always require the DREMS operating system and the DREMS middleware. This requirement can hardly be realized on every CPSs, like cars that require AUTOSAR. Nevertheless, the DREMS operating system and its middleware are potential candidates for supported platforms by MechatronicUML in the future.

5.8 Summary

This section summarizes to what degree the related work and MechatronicUML in combination with the results of this thesis fulfill the requirements R.5.1-R.5.8, which the introduction of this chapter defines for a model-driven software construction approach for CPSs. We claim that the requirements are necessary but not sufficient because we focus on safety but not, e.g., on security, which is mandatory for connected systems. Table 5.3 shows a summary of our opinion about the fulfillment of the requirements. In the following paragraphs, we discuss the realated work in the context of each requirement.

Table 5.3: Comparison of the Requirements Fulfillment for Software Construction Approaches for CPSs (Legend: ✓: fulfilled; ◑: fulfilled partially; ∅: not fulfilled; ?: unknown; n.a.: not applicable)

Approach	R.5.1	R.5.2	R.5.3	R.5.4	R.5.5	R.5.6	R.5.7	R.5.8
SOFA HI [PWT+08; HPM+10]	✓	✓	◑	✓	◑	∅	◑	◑
ProCom [BCC+08; CFMS10]	✓	✓	◑	✓	✓	✓	◑	◑
DEECo [MKM+16; BGH+13]	✓	∅	∅	∅	◑	◑	✓	◑
CHESS [CSCS13]	✓	✓	✓	?	✓	✓	∅	∅
AutoFocus3 [AVT+15]	✓	✓	✓	✓	✓	✓	◑	◑
Flex-eWare/eC3M [RCGT09; JJK+12]	✓	∅	n.a.	n.a.	n.a.	◑	◑	✓
MyCCM-HI [BHP09; BFHP09]	✓	∅	✓	n.a.	✓	◑	◑	◑
MICOBS [PPK+11]	✓	∅	◑	n.a.	◑	◑	◑	✓
OMG's D&C [OMG06]	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	◑	◑
Autosar [AUTOSAR14c]	✓	∅	◑	n.a.	◑	◑	◑	◑
RoboCop [MC04; Maa05]	✓	∅	∅	n.a.	◑	n.a.	∅	◑
DREMS [BDO+15]	✓	∅	◑	n.a.	n.a.	◑	◑	✓
MechatronicUML including this thesis	✓	✓	✓	✓	✓	✓	✓	✓

Requirement R.5.1 forms the basis for a model-driven development approach by defining a software architecture. Therefore, all approaches fulfill this requirement directly or by using another technology. OMG D&C does not provide a component-based approach by itself but refers to the UML. MechatronicUML provides a sophisticated hierarchical component model (cf. Section 2). DREMS supports only a flat software architecture.

Only 5 of the 13 compared approaches provide a modeling approach for the behavior and fulfill the requirement R.5.2. CHESS supports a subset of the UML behavior constructs, e.g., it forbids composite states with orthogonal regions. OMG D&C refers again to the UML. MechatronicUML provides RTSCs for modeling the functional behavior, which refine RTCPs that represent the coordination behavior between system parts or different CPSs.

The hardware platform and the allocation are supported only by 4 approaches and partially by another 5 approaches. Flex-eWare and OMG D&C refer to MARTE without showing how it should be used. SOFA HI, ProCom, MICOBS, and DREMS provide only a rudimentary hardware model that mostly do not consider sensors and actuators (cf. Section 4.12.1.2). Within Autosar many properties are defined using unstructured text or the concrete specification is left to the car manufacturers or respectively to the tool vendors (cf. Section 4.12.1.3). MechatronicUML provides the hardware platform model that Section 4.3 describes. Additionally, Section 5.3.3 introduces the APIMappingMl to describe the interfaces of the software platform.

The verification and validation that the requirement R.5.4 necessitates can only be fulfilled by approaches that fulfill the requirement R.5.2. Otherwise, we stated "n.a." because the preconditions are not fulfilled. All approaches that provide a behavior specification besides CHESS also provide mechanisms to verify and validate the behavior. CHESS does not discuss this topic. Therefore, we set the cell in the table to the value "?" but we propose that the general UML verification of Knapp, Merz, and Rauh is usable. SOFA HI does not support timed model checking. MechatronicUML provides CPS-specific verification properties and CPS-specific timed model checking for RTCPs [Dzi17]. Furthermore, it supports a refinement approach for the transformation of protocols roles to component ports [HBDS15; Hei15], which is the basis for our code generation. Furthermore, the overall system's behavior can be validated using MiL simulation using Simulink [Hei15], Functional Mockup Units (FMUs) [*PSR+12], and Modelica (cf. Section 3.1).

Concerning the requirement R.5.5, the approaches ProCom, CHESS, AutoFocus3, and MyCCM-HI support a model coupling with platform functionalities directly. Furthermore, SOFA HI, DEECo, MICOBS, Autosar, and RoboCop enable to use general function calls of other software components to access functionality of the platform. Due to the focus on the infrastructure and not on the application layer we state Flex-eWare, OMG's D&C, and DREMS with "n.a.". MechatronicUML provides a platform-specific coupling to devices (cf. Section 5.3.3). Thereby, it is possible to access the same type of a device with different parameters or function calls on different target platforms. Additionally, we model access constraints to restrict the time access, which avoids device failures during runtime.

In addition to the modeling and quality assurance capabilities, a model-driven development approach becomes usable if it supports to generate code from the specification, which avoids manual errors during the implementation using the defined specification. The requirement R.5.6 necessitates the generation for the software artifacts for the functional architecture and the behavior. Only the approaches ProCom, CHESS, and AutoFocus3 fulfill this requirement completely. SOFA HI does not support code generation although it provides a behavior specification. DEECo, MyCCM, Autosar, eC3M, MICOBS, and DREMS support code generation for the functional architecture and fulfill therefore the requirement partially. OMG's D&C and RoboCop provide frameworks or runtime containers but do not support code generation. The approaches may be combined with other approaches. Therefore, we judge them by using "n.a.". The requirement R.5.7 requires advanced concepts for intra- and inter-system communication, including fulfilling complex QoS policies. The only other approach that supports sophisticated QoS policies for the communication is DEECo. Nevertheless, it does not take care of verified behavior-based QoS assumptions like message buffering because it does not consider the application logic. All other approaches except CHESS and RoboCop do support the generation of connectors for supporting intra- and inter-system communication. CHESS requires that communication is explicitly modeled by a software engineer as separate software components. Section 5.3.4 describes how MechatronicUML handles the intra- and inter-system communication, including message buffering and different QoS policies that are required by the different QoS properties of the application. Concerning

the requirement R.5.8, the deployment of the application software to different platforms requires at least an abstraction layer, framework, or dedicated runtime environment. Flex-eWare, MICOBS, and DREMS generate customized code for the requirements of the components and fulfill the requirement. Most of the other related work partially fulfills this requirement by either providing their own runtime environment or building up on general runtime environments like CORBA. The drawback is that these environments must be provided and maintained for the different platforms. CHESS requires modeling of the functionality for accessing the platform explicitly. Therefore it does not fulfill the requirement currently. MechatronicUML generates customized technical infrastructure code for the requirements of the components. Furthermore, MechatronicUML separates platform-independent code from platform-specific code, which enables to deploy the functional part to further platforms without changing it. Only the infrastructure code has to be extended or adapted.

6

Conclusion

6.1 Summary

Using communication in distributed Cyber-Physical Systems (CPSs) increases the overall engineering complexity. Heterogeneous, distributed platforms add another level of complexity and constitute additional engineering tasks where engineers may introduce new errors that can lead to catastrophic failures during runtime. Currently, connected technologies like, Internet of Things (IoT), 5G network, or Vehicle-to-Everything (V2X), have a large growth and are the driving force for rapid changes in our daily life [aca11]. Thereby, the improvement of the engineering itself and the system's safety becomes much more important. The contributions of this thesis enable engineers of connected CPSs to retain the safety the development phases of the simulation-based validation, the resource allocation, and the final software construction phase itself to detect and avoid systematic errors of the constructed software. We implement and evaluate all our contributions based on the MechatronicUML approach. During the development of this thesis, we define the MechatronicUML approach on the basis of former work of our colleagues. Furthermore, we integrate our contributions into the MechatronicUML approach to enable a seamless and efficient engineering. Nevertheless, our contributions may also help to improve other model-driven and component-based approaches. All our contributions are part of the MechatronicUML tool suite and are licensed under the open-source Eclipse public license (EPL).

As our first contribution, we provide a Model-in-the-Loop (MiL) simulation approach for CPSs. Our approach enables to test the state-based behavior, including its real-time and communication logic, and the correct integration with feedback controllers, physical parts, and the environment as a safe virtual prototype. Especially, the integrated behavior cannot be verified by timed model checking. Software engineers can detect and remove logical errors before a physical prototype or computing platform exist by using our MiL simulation approach. It is even possible to engineer the software and to test the overall system's behavior iteratively, starting with a simple behavior and building up a more and more complex behavior successively. In particular, we contribute the Real-Time Coordination Modelica library that supports and eases the modeling of message-based, real-time communication within Modelica tools, like Dymola [Dym]. The Modelica StateGraph2 library misses a built-in support for these concepts that are required for modeling and simulating real-time coordination behavior. Furthermore, we contribute a transformation of formally verified software specifications defined by MechatronicUML to Modelica. The MechatronicUML model contains a definition of the software architecture as a component model. The behavior of discrete components are specified using Real-Time Statecharts (RTSCs) and formally verified by using UPPAAL. The transformation uses our Real-Time Coordination Modelica library and is fully automatic. As a result, engineers can first formally verify that the discrete software specification fulfills the safety requirements by using the concepts of Dziwok [Dzi17] and then use our concept and the same model to simulate the integration with the continuous software specification

and physical behavior. The input MechatronicUML models are less complex than the resulting output Modelica models because MechatronicUML provides higher-level modeling constructs for the state-based behavior and the message-based asynchronous communication. Using a case study, we evaluate that the generated code is syntactical correct Modelica code, contains all expected Modelica model elements, and that the simulation runs conform to the MechatronicUML semantics. The case study contains one case from the automotive domain and one case from the production/automation domain. We fulfill our six self-defined requirements but no related approach.

As our second contribution, we provide a model-driven allocation engineering approach. Our approach enables to plan a feasible allocation specification automatically based on the software component model, hardware platform model, and the specified constraints. An advantage of this approach is its flexibility regarding adding new constraints by allocation engineers without the knowledge of encoding and solving them as a formal constraint satisfaction problem. In particular, we firstly contribute an approach for the multi-view modeling of hardware platforms for specifying them with respect to the information required for allocation engineering in an easy and expressive way. The approach contains a systematic process and a formal hardware platform description language. Secondly, we identify and provide modeling extensions for the MechatronicUML component model to specify its real-time execution and resource consumption properties with respect to required properties for allocation engineering. Thirdly, we contribute the domain-specific Allocation Specification Language (ASL) for specifying allocation constraints. The ASL integrates the Object Constraint Language (OCL) to query existing models, to calculate subsets, and to calculate complex resource consumptions. Fourthly, our allocation engineering approach provides an automatic transformation of the allocation constraints, which are specified using the ASL, into an Integer Linear Programming (ILP) specification for solving this constraint satisfaction problem automatically by existing solvers. We prove that our automatic transformation preserves the formally defined semantics of the ASL. We illustrate our contributions based on the constraint definitions for CPSs and MechatronicUML. However, our allocation engineering approach is not limited to MechatronicUML but also usable for other EMF-based languages and allocation problems, like allocating tasks to processor cores in multi-processor environments. We evaluate the reliability of the implementation of our model-driven allocation engineering approach, the efficiency of the allocation constraint specification approach in comparison to specifying the same constraints as an ILP specification manually, and the scalability of the automatic transformations and planning. The evaluation consists of a case study and two cases from the automotive domain. We get a relative change of -0.9792 Lines of Code (LOC) from an ILP specification to our ASL specification. The scalability evaluation shows that we can plan the allocation of 104 component instances and 72 Electronic Control Units (ECUs) in less than 12 minutes. Finally, we fulfill our eight self-defined requirements but no other related approach fulfills them all.

As our third contribution, we provide an approach for the automatic software construction of distributed CPSs specified and validated using MechatronicUML. We enable software engineers and developers to implement the software effectively and efficiently by using our approach. The approach uses, combines, and enhances the three approaches MDA [OMG03], architecture-centric model-driven software development [VSK05; SVC06], and model-driven middleware [GBK+08] in the context of distributed CPSs specified using MechatronicUML. In particular, we contribute an extended MechatronicUML component model, a component-container-based code generation infrastructure, and the domain-specific languages ApiMl and APIMappingMl for specifying device APIs and the integration with the MechatronicUML component model. Additionally, we contribute a code generation of C code for the platform-independent MechatronicUML component model and for the newly developed platform-

specific deployment model that considers the allocation specification. Furthermore, the approach automates the building of executables from the generated implementation by providing a makefile. The generated implementation separates platform-independent functional concerns from platform-specific technical concerns. Our approach fulfills the verified Real-Time Coordination Protocol (RTCP)'s assumptions by generating and configuring the communication middleware DDS properly. Using our approach enables the generated software to handle the lifecycle of distributed, deployed components. Additionally, it enables to handle and use different communication middlewares, protocols, and media. Software engineers can reuse and configure software components on different platforms and can integrate 3rd-party software artifacts. We illustrated our contributions on the basis of MechatronicUML and C but the concept is not limited to these languages. On the one hand, it can be used for component-based approaches like AutoFocus3, ProCom, or CHESS. On the other hand, it can be used for different target languages like Java, Rust, and C#. We evaluate the correctness and efficiency of our software construction approach conducting a case study from the automotive domain. We show that for this case the relative change of the LOC from the input model to the generated code is about 3.55. Finally, we provide a summary that compares our approach with related work in the context of eight requirements that are based on the systematic literature review of Dann [Dan16]. Our approach is the only approach that fulfills all eight requirements.

In combination, our contributions retain safety with respect to systematic errors during constructing the software after designing and verifying a platform-independent software specification of CPSs. Moreover, our integrated engineering approach and the corresponding tooling enable engineers, e.g., software application engineers, system engineers, software platform engineers, hardware platform engineers, allocation engineers, safety engineers, and software developer, to create and validate CPSs together in a close collaboration in a model-driven way. Using MechatronicUML, we provide the concepts and tooling for a seamless engineering from requirements to running, distributed, cooperating CPSs. Thereby, we consolidate the safety and reliability of the system's behavior and enable the effective and efficient development. Nevertheless, during the development of this thesis, new challenges come up and the contributions of this thesis enable further possibilities for research on its topics. The next section describes the possible future work.

6.2 Future Work

This section highlights an extract of the unsolved problems that are valuable for further research. Furthermore, based on the contributions of this thesis, it is worth to overcome its limitations and to mitigate its assumptions. Additionally, the concepts of this thesis have to be transferred from research to application. Therefore, the concepts have to be evaluated in the context of industrial projects and industrial users.

View-based and Domain-driven Design: Our development approach MechatronicUML supports the specification of cooperative CPSs. CPSs are diverse and heterogeneous. The requirements of different domains, like railway, production, automotive, (home) automation, space, energy, and telecommunication differ to a large extent. Nevertheless, the different systems have interfaces and interact with each other. In the future, the energy grid has to communicate with electric cars and production plants to be resilient. We need domain-specific design views to meet the special requirements of each domain efficiently. Furthermore, we need explicit interaction points to other domain-specific design methods and models and to systems engineering methods and models. The consistency must be maintained if domain models or even the domain meta-models evolve. Future work should investigate which commonalities exist within different

domains, which differences exist, and how domain-spanning interactions are handled. Existing development approaches should be extended according to these findings and new approaches should be developed, e.g., by using the view-based model-driven development approach [Bur14] and model-driven consistency management [Rie14].

Model-Driven Testing: The contributions of the thesis support the creation of a Modelica simulation model and the creation of an implementation. Despite the fact that the fulfillment of the safety and liveness properties of the design models is formally verified errors could exist within the integration of continuous and discrete software parts and within the implementation of the software. Platform-specific properties like task periods, task priorities, and different worst-case execution times on different platforms can introduce further errors. Therefore, future work should provide testing support within the MiL simulation task and during the software installation and software commissioning task. Test cases may be generated automatically from the scenario-driven MechatronicUML requirements engineering method [Gre11; HFK+16; HBM+16; Bre16]. Furthermore, testing metrics for unit tests and integration tests have to be developed for hybrid CPSs [JFA+07]. The metrics should combine classical software coverage criteria [YLW09] and testing criteria from control theory like controller stability [ÅK14]. Furthermore, an automated test case execution could reduce the manual effort for testing and may increase the application of tests, which should result in systems with a higher quality.

Deployment and Lifecycle Handling: The contributions of this thesis end with the creation of deployable software artifacts. Nevertheless, challenges exist within CPSs during the deployment of the software on the target systems [Dea07]. Currently, the installation on the target platform is performed manually. This is a challenging task if not all required libraries are statically linked to the executable. Software engineers have to make sure that all dependencies and licenses are available on all distributed target platforms. Currently, no standardized deployment platform or software ecosystem exists for CPSs like app stores for mobile devices. Therefore, future work should take care of developing an open ecosystem for the deployment of CPSs and creation of description languages that specify the software requirements [Kri13; KCO10]. Furthermore, the lifecycle of CPSs is complex. The commissioning that brings the system into operation is a challenging task because the system has to be connected and configured correctly. For example, systems provide the possibility to deactivate communication interfaces like wireless LAN or Bluetooth to save energy. Additionally, during the start-up phase of a system, the different software parts have to be started in the correct order to bring the system into operation. Also the termination has to be handled correctly because otherwise a car may trap passengers or a person may get caught by a press. Therefore, future work should investigate how the lifecycle of CPSs can be designed and handled safely.

(Fail-Safe) Operation and Resilience: The development of CPSs is not finished with the creation of the software. The operation itself must fulfill high-quality requirements because they are deployed within critical infrastructures, like transportation of energy supply. Resilient systems must provide their service in conformance with the expected, predictable behavior even if the conditions of operation changes [Crn11]. Furthermore, a system must operate safely in cases of failures like the defect of an ECU [Sin11]. Therefore, future work should take into account how to derive resilient and fail-safe systems. A key factor for these system properties is to monitor the current status of the system during runtime because a system cannot be improved without measuring the current status [WH07]. A research question that should be answered is how to integrate monitor frameworks like Kieker [vHWH12] or runtime testing frameworks like Proteus [FC15] into the infrastructure that this thesis contributes. Furthermore, safety cases have to be integrated that specify the safety requirements explicitly [BB10]. Additionally, new features and updates must be integrated seamlessly from the design into the deployed system. The term *DevOps* is used in recent years to close the gap between development and operations in order to continuously deploy stable software [BvHW+15]. The *DevOps* research addresses

mainly enterprise software systems and explicitly does not take into account the requirements of embedded systems and CPSs [BvHW+15]. Future work should consider how to use DevOps concepts for safety-critical CPSs.

Product Lines and Variant Handling: The software industry uses software product lines to manage and maintain its software products [VSR07]. The goal is a large-scale software reuse. Software components are artifacts that are reused [ABB+07]. However, this thesis shows that the software components have varying requirement dependencies on the basis of the software platform, hardware platform, sensors, and actuators. Furthermore, this thesis enables to configure components to be reused in different contexts. In previous work, Brink et al. contribute how product lines, their variants, and dependencies that depend on software and hardware parts are modeled [BKPS14; BHS15]. Future work should investigate how these product line models can be used to allocate, construct, and deploy the software of CPSs automatically.

Logic Programming, Constraints, and Optimization: The application of model-driven engineering in different domains and its rising usage in industry triggers the needs to solve constraint satisfaction problems and optimizations [ABG+13]. This thesis contributes a new method how to combine model-driven engineering and logic programming to solve allocation constraint problems of component instances and ECUs. However, this concept is not constrained to be used only for component instance to ECU allocation. Constrained mappings and allocations have to be calculated for different domains and use cases. For example, engineering tasks have to be mapped to engineers who are able to work on them or cloud applications have to be mapped to a server that may be restricted to be located within certain countries. Furthermore, the Quality of Service (QoS) of systems' services depends on mappings and allocations. The mappings and allocations should consider the QoS to optimize the systems. Previous work considers how to extend the ASL to describe services and its quality and how to define optimization goals [BCD+14; *PHM16; *HP17]. Future work should investigate how to extend the ASL to be used on different context models and how to define the mappings or allocations that should be calculated. This includes also cross dependencies between different mappings and allocations. For example, the allocation of a connector between software components to a network link depends on the allocation of a component instance to an ECU. Furthermore, optimization criteria should be defined that improve the QoS of services from different domains.

Real-Time Scheduling for Many- and Multi-Core ECUs: Currently, more and more systems use many- and multi-core ECUs to handle the complexity of the software [KKLR13]. Furthermore, Real-Time Operating Systems (RTOSs) manage the execution of program parts by using tasks [Kop11]. A task that is able to be executed is in the state ready and a task that is executed is in the state running. A task that waits for a shared resource and that is able to be executed is in the state blocked. Additionally, tasks may have properties like a period, priority, deadline, and Worst-Case Execution Time (WCET). A scheduler decides based on its strategy and the task properties which task is executed. Switching the currently executing task takes execution time because the old execution context has to be saved and the new execution context has to be loaded into the processing memory. Software engineers have to decide to split up the software into tasks and which ECU and ECU core executes the software. They have to find a trade-off between the performance of the system and ensuring that all hard real-time requirements are met during runtime. It is a complex task to decide how to map components to tasks and to choose the task properties in a way that guarantees correct real-time behavior without wasting too much performance by blocking the processor without using it efficiently. Our previous work, which is not part of this thesis, shows how to calculate task properties automatically and how to split up components' behavior automatically using certain modeling restrictions and a time-triggered scheduling [*GPS17]. Future work should

investigate how to use reachability analysis to get rid of the modeling limitations and still be able to calculate modeling tasks and tasks properties. Furthermore, it should investigate how to combine time-triggered behavior with event-triggered behavior to improve the system's performance

Self-Adaption by Reconfiguration: Self-adaptive systems have the ability to reconfigure their software structure or behavior in response to their environment, goals, or internal state [dLGMS13]. In previous work, we identified the following three use cases for self-adaption: use case 1: changing communication partners due to a changing environment, use case 2: changing mode of operation, use case 3: increasing availability of the system [*HSD15]. This thesis contributes a simulation-based analysis, an allocation analysis, and a code generation only for static MechatronicUML software architectures. Previous work on reconfiguration for MechatronicUML [HBV17; Hei15; SHG16; EHH+13; HB13; THHO08] defines concepts for modeling and analyzing reconfiguration behavior. Future work should investigate how to consider these concepts for the named use cases for planning software allocation and constructing the software. For example, the concepts of Heinzemann [HBV17; Hei15; HRS13] allow computing the state space of the system configurations. Thereby, the maximum system resources can be calculated and considered for allocation planning. Additionally, component instances, which are never active at the same time, can be identified. These component instances may share resources without getting into conflicts during runtime. The state space of the system configurations may also be used to activate and deactivate component instances or tasks that execute them respectively. Furthermore, previous work of Heinzemann et al. shows how reconfiguration can be considered during the simulation of self-adaptive systems using MATLAB Simulink [Hei15; HRS13]. Different Modelica-based tools offer different possibilities to consider reconfiguration behavior [ZLB07; EMK15]. Future work should investigate how to combine the Modelica capabilities for considering reconfiguration in combination with the reconfiguration approach of Heinzemann [Hei15] and the real-time coordination Modelica library that this thesis contributes.

Security: Distributed systems have communication interfaces to connect the systems for realizing and improving the overall system's behavior. However, the communication interfaces are also used as an attack vector [HH05]. Engineers and operators of CPSs have the following security objectives: authentication, authorization, confidentially, integrity, auditability, and availability [OPCUAPart2]. Furthermore, they have to struggle with security threats like message flooding, spoofing, alteration, and replay [OPCUAPart2]. Currently, our approach only analyses and considers safety requirements during software construction. It does not consider security requirements [ISO27001] and common criteria for security evaluation [ISO15408] at all. Nevertheless, this thesis contributes an approach for allocation engineering and software construction that can be used as a basis for engineering secure, distributed CPSs. Therefore, future work should investigate how model-driven security engineering approaches [NAY17] can be combined with the concepts of this thesis. Firstly, a threat model that considers all parts of the system, even the platform model, has to be developed [NAY17]. Secondly, considering security requirements explicitly can be used to improve our allocation engineering approach. Thereby, critical software components may only be allocated to hardened platforms that may provide functionalities like secure boot to defeat the manipulation of the boot loader and thereby the manipulation of the running software [DW08]. Thirdly, the generation of the implementation and configuration of the communication middleware itself have to take security requirements of the coordination protocol into account by adding authentication or encryption [SLCS12].

Bibliography

Contributions of the Papers to this Thesis

The main contributing publications to this thesis are the following: [*PDS+12; *PDM+14; *PHM+14; *PMDB14; *PH15]. I am the main author of the texts and contributor to these papers. The concept and detailed solutions of each paper and the evaluations were developed by myself. Furthermore, I coordinated the whole writing process of each paper and presented all papers on the corresponding conferences personally. The papers benefited from the collaboration with my co-authors, who contributed smaller amounts of texts, and the external reviewers, who provided suggestions for improvements. In comparison to the papers, the content of this thesis has been significantly detailed and integrated into a coherent solution.

Own Peer-Reviewed Papers

[*BDG+14] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann, "The MechatronicUML method: Model-driven software engineering of self-adaptive mechatronic systems," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion '14, Hyderabad, India: ACM, 2014, pp. 614–615, ISBN: 978-1-4503-2768-8. DOI: http://doi.acm.org/10.1145/2591062.2591142 (cit. on pp. 2, 4, 11, 84).

[*DGB+14] S. Dziwok, C. Gerking, S. Becker, S. Thiele, C. Heinzemann, and U. Pohlmann, "A tool suite for the model-driven software engineering of cyber-physical systems," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, Hong Kong, China: ACM, 2014, pp. 715–718, ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2661665 (cit. on pp. 2, 5, 12, 70, 94, 131, 139, 161, 196, 201).

[*GPS17] J. Geismann, U. Pohlmann, and D. Schmelter, "Towards an automated synthesis of a real-time scheduling for cyber-physical multi-core systems," in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, ser. MODELSWARD '17, Setúbal, Portugal: Scitepress, 2017, pp. 285–292, ISBN: 978-989-758-210-3. DOI: 10.5220/0006117702850292. [Online]. Available: http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=ADc9X1W/tIE=&t=1 (cit. on pp. 14, 103, 145, 151, 207, 223).

[*HPR+12] C. Heinzemann, U. Pohlmann, J. Rieke, W. Schäfer, O. Sudmann, and M. Tichy, "Generating Simulink and Stateflow models from software specifications," in *Proceedings of DESIGN 2012, the 12th International Design Conference*, ser. DESIGN '12, Glasgow, Scotland: The Design Society, 2012, pp. 475–484, ISBN: 978-953-7738-17-4 (cit. on pp. 3, 77, 79, 81).

[*PDM+14] U. Pohlmann, S. Dziwok, M. Meyer, M. Tichy, and S. Thiele, "A Modelica coordination pattern library for cyber-physical systems," in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '14, Lisbon, Portugal: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 76–85, ISBN: 978-1-63190-007-5. DOI: 10.4108/icst.simutools.2014.254640 (cit. on pp. 22, 30, 38–40, 225).

[*PDS+12] U. Pohlmann, S. Dziwok, J. Suck, B. Wolf, C. C. Loh, and M. Tichy, "A Modelica library for real-time coordination modeling," in *Proceedings of the 9th International Modelica Conference*, ser. Modelica '12, DLR - Robotics and Mechatronics Center, Modelica Association, Linköping, Sweden: Linköping University Electronic Press, 2012, pp. 365–374, ISBN: 978-91-7519-826-2. DOI: 10.3384/ecp12076365 (cit. on pp. 5, 22, 26, 27, 30–37, 79, 80, 225).

[*PH15] U. Pohlmann and M. Hüwe, "Model-driven allocation engineering (t)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, Lincoln, NE, USA: IEEE, Nov. 2015, pp. 374–384. DOI: 10.1109/ASE.2015.18 (cit. on pp. 6, 86, 88, 93, 98, 104–113, 115–119, 123, 134–137, 140, 144, 148–150, 225, 301).

[*PHM+14] U. Pohlmann, J. Holtmann, M. Meyer, and C. 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*, ser. SEAA '14, New York, USA: IEEE, Aug. 2014, pp. 191–198. DOI: 10.1109/SEAA.2014.18 (cit. on pp. 6, 7, 15, 21, 22, 29, 30, 43, 46, 68, 69, 72, 75–77, 79, 225).

[*PMDB14] U. Pohlmann, M. Meyer, A. Dann, and C. 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*, ser. VAO '14, York, United Kingdom: ACM, 2014, pp. 23–30, ISBN: 978-1-4503-2900-2. DOI: 10.1145/2631675.2631682 (cit. on pp. 6, 12, 86, 88, 92–102, 146, 147, 225).

[*Poh13] U. Pohlmann, "Safe deployment for reconfigurable cyber-physical systems," in *Proceedings of the 18th international doctoral symposium on components and architecture*, ser. WCOP '13, Vancouver, British Columbia, Canada: ACM, 2013, pp. 31–36, ISBN: 978-1-4503-2125-9. DOI: http://doi.acm.org/10.1145/2465498.2465503 (cit. on pp. 5, 88).

[*PSR+12] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner, "Generating functional mockup units from software specifications," in *Proceedings of the 9th International Modelica Conference*, ser. Modelica '12, DLR - Robotics and Mechatronics Center, Modelica Association, Linköping, Sweden: Linköping University Electronic Press, 2012, pp. 765–774, ISBN: 978-91-7519-826-2. DOI: 10.3384/ecp12076765 (cit. on pp. 6, 20, 22, 78, 79, 81, 216).

[*PT+11] U. Pohlmann and M. Tichy, "Modelica code generation from ModelicaML state machines extended by asynchronous communication," in *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, ser. EOOLT '12, Linköping, Sweden: Linköping University Electronic Press, 2011, pp. 75–84, ISBN:

978-91-7519-825-5. [Online]. Available: http://www.ep.liu.se/ecp/056/009/ecp1105609.pdf (cit. on pp. 6, 22, 26, 76, 79, 80).

[*PTD+14] U. Pohlmann, H. Trsek, L. Dürkop, S. Dziwok, and F. Oestersotebier, "Application of an intelligent network architecture on a cooperative cyber-physical system: An experience report," in *Proceedings of the 19th IEEE International Conference on Emerging Technology and Factory Automation*, ser. ETFA '14, New York, USA: IEEE, Sep. 2014, pp. 1–6. DOI: 10.1109/ETFA.2014.7005358 (cit. on pp. 2, 22, 68, 69).

[*PW+12] U. Pohlmann and R. Wagner, "Einsatz des FMI/FMU-Standards zur frühzeitigen Simulation von Software- und Hardwaremodellen komplexer mechatronischer Systeme," in *Berichtsband des Tag des Systems Engineering 2012*, ser. TdSE '12, München, Germany: Carl Hanser Verlag, 2012, pp. 229–238, ISBN: 978-3-446-43435-6. DOI: 10.3139/9783446436039.023 (cit. on pp. 6, 22, 78).

[*SPF+10] W. Schamai, U. Pohlmann, P. Fritzson, J. C. Paredis, P. Helle, and C. Strobel, "Execution of UML state machines using Modelica," in *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Linköping, Sweden: Linköping University Electronic Press, 2010, pp. 1–10, ISBN: 978-91-7519-824-8. [Online]. Available: http://www.ep.liu.se/ecp/047/001/ecp4710001.pdf (cit. on pp. 6, 26, 76, 80).

[*TJD+12] H. Teichrieb, V. Just, S. Dziwok, U. Pohlmann, T. Schierbaum, and A. Trächtler, "Modellbasierter Entwurf zweier kooperierender Delta-Roboter," in *Berichtsband des Tag des Systems Engineering 2012*, ser. TdSE '12, München, Germany: Carl Hanser Verlag, 2012, pp. 217–228, ISBN: 978-3-446-43435-6. DOI: 10.3139/9783446436039.022 (cit. on p. 2).

Own Non-Peer-Reviewed Technical Reports and Book

[*DP14] A. P. Dann and U. Pohlmann, "The MechatronicUML hardware platform description method – process and language," Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-14-336, Feb. 2014, v. 0.1 (cit. on pp. 6, 94, 277).

[*DPP+16] S. Dziwok, U. Pohlmann, G. Piskachev, D. Schubert, S. Thiele, and C. Gerking, "The MechatronicUML design method - process and language for platform-independent modeling," Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Tech. Rep. tr-ri-16-352, Dec. 2016, Version 1.0 (cit. on pp. 2, 4, 11, 12, 14–17, 21, 28, 30, 34, 63, 68, 70, 71, 78, 84, 160–162, 164, 189, 191, 207, 277).

[*GST14] H. Anacker, F. Bauer, H. Borcherding, S. Dziwok, U. Frank, R. Herden, G. Hoppe, V. Just, M. Kiele-Dunsche, D. Kruse, F. Oestersötebier, J. Papenfort, U. Pohlmann, H. Reddehase, J. Rieke, T. Schierbaum, L. Seifert, H. Stichweh, H. Teichrieb, R. Wagner, and S. Wessels, *Semantische Technologien im Entwurf mechatronischer Systeme*, J. Gausemeier, W. Schäfer, and A. Trächtler, Eds. München, Germany: Carl Hanser Verlag, 2014, ISBN: 978-3-446-43630-5 (cit. on pp. 2, 22, 68, 69, 71).

[*HP17] M. Hüwe and Pohlmann, "Formal definition and proofs for the MechatronicUML Allocation Specification Language," Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Tech. Rep. tr-ri-17-353, Apr. 2017 (cit. on pp. 86, 88, 116–118, 155, 223).

[*HSD15] C. Heinzemann, D. Schubert, S. Dziwok, U. Pohlmann, C. Priesterjahn, C. Brenner, and W. Schäfer, "Railcab convoys: An exemplar for using self-adaptation in cyber-physical systems," Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-15-344, Jan. 2015 (cit. on pp. 2, 21, 38, 160, 224).

[*PH17] U. Pohlmann and M. Hüwe, "Model-driven allocation engineering – abridged version," in *Software Engineering 2017, 21. – 24. Februar 2017, Hannover, Proceedings*, ser. LNCS, vol. P-267, Bonn, Germany: Köllen Druck+Verlag GmbH, 2017, pp. 73–74, ISBN: 978-3-88579-661-9 (cit. on pp. 6, 88).

[*PHM16] U. Pohlmann, J. Holtmann, and M. Meyer, "Das Erwachen der Macht – automatische Softwareverteilung," in *Tagungsband des Embedded Software Engineering Kongress 2016*, ser. ESE '16, Würzburg, Germany: ELEKTRONIKPRAXIS, Dec. 2016, pp. 587–592, ISBN: 978-3-8343-2504-4 (cit. on pp. 6, 88, 223).

[*Poh10] U. Pohlmann, "A UML based modeling language with operational semantics defined by Modelica," Master's Thesis, Paderborn University, Department of Computer Science, Software Engineering Group, 2010. [Online]. Available: `https://www.hni.uni-paderborn.de/publikationen/publikationen/?tx_hnippview_pi1[publikation]=7943` (cit. on p. 76).

[*SDP15] W. Schäfer, S. Dziwok, U. Pohlmann, J. Bobolz, M. Czech, A. P. Dann, J. Geismann, M. Hüwe, A. Krieger, G. Piskachev, D. Schubert, and R. Wohlrab, "Seminar Theses of the project group Cybertron," Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Projektgruppenbericht tr-ri-15-345, Jul. 2015 (cit. on p. 88).

Contributions of the Supervised Theses to this Thesis

During the development of the concepts of this thesis, I supervised the theses that the next section lists. I defined the topic, including the motivation, problem statements, and goals, for each thesis. During the term of each thesis, I discussed with each student their ideas and progress at least for one hour per week. During that period, I significantly developed each concept and gave detailed feedback to each thesis. The main contributing supervised thesis to this thesis are the following: [Wol11; Dan13; Hüw13; Ros14; BCD+14; Dan16]. In comparison to the supervised theses, the concepts and their descriptions within this thesis have been significantly detailed, extended, and integrated into a coherent solution.

Supervised Theses

[BCD+14] J. Bobolz, M. Czech, A. P. Dann, J. Geismann, M. Hüwe, A. Krieger, G. Piskachev, D. Schubert, and R. Wohlrab, "Project group cybertron 2013/2014 final document," Project Group Documentation, Software

Engineering Group, Heinz Nixdorf Institute, Paderborn University, Oct. 2014. [Online]. Available: https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/PG_Cybertron/PG_Cybertron_Final_Document.pdf (cit. on pp. 84, 88, 145, 162, 223, 228).

[Dan13] A. P. Dann, "Ein Plattform-Beschreibungsmodell für die Software-Verteilungsplanung mechatronischer Systeme," Bachelor's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Apr. 2013 (cit. on pp. 5, 84, 88, 228).

[Dan15] ——, "Survey: Real-time distribution middleware," Seminar's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Oct. 2015 (cit. on p. 172).

[Dan16] ——, "Model-driven deployment of cyber-physical systems," Master's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Mar. 2016 (cit. on pp. 5, 159, 162, 221, 228, 307).

[Gei15] J. Geismann, "Multi-core execution of safety-critical component-based software," Master's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Dec. 2015 (cit. on pp. 5, 14, 103, 145, 151, 162).

[Hüw13] M. Hüwe, "Das Verteilungsproblem für mechatronische systeme," Bachelor's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Oct. 2013 (cit. on pp. 5, 83, 84, 88, 114, 228).

[Ros14] M. Rose, "Modellgetriebene Generierung von API-Konnektoren für eingebettete Sensoren und Aktuatoren," Bachelor's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Mar. 2014 (cit. on pp. 162, 228).

[Win14] A. Winkler, "Modellgetriebene Softwareentwicklung zur induktiven Energieübertragung," Bachelor's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Mar. 2014 (cit. on p. 41).

[Wol11] B. Wolf, "Automatische Transformation von Modellen der MechatronicUML nach Modelica," Master's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Nov. 2011 (cit. on pp. 5, 21, 22, 228).

Literature

[ABB+07] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Peach, J. Wust, and J. Zettel, *Component-Based Product Line Engineering with UML*. Boston, USA: Addison-Wesley Professional, 2007, ISBN: 978-0201737912 (cit. on p. 223).

[ABDM01] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, Eds., *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, USA: IEEE Press, 2001, ISBN: 0769510000 (cit. on p. 85).

[ABG+13] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transaction on Software Engineering*, vol. 39, no. 5, pp. 658–683, May 2013, ISSN: 0098-5589. DOI: 10.1109/TSE.2012.64 (cit. on pp. 104, 144–147, 223).

[ABGM09] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for architecture optimization of aadl models," in *Prococeedings of the ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, ser. MOMPES '09, May 2009, pp. 61–71. DOI: 10.1109/MOMPES.2009.5069138 (cit. on pp. 3, 83, 86, 148, 152, 153).

[aca11] acatech, Ed., *Cyber-Physical Systems: Driving Force for Innovations in Mobility, Health, Energy and Production*. Berlin Heidelberg, Germany: Springer, 2011, ISBN: 978-3-642-29090-9. DOI: 10.1007/978-3-642-29090-9 (cit. on pp. 1, 83, 219).

[ÅCF+07] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The SAVE approach to component-based development of vehicular systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 655–667, 2007, Component-Based Software Engineering of Trustworthy Embedded Systems, ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.08.016 (cit. on p. 209).

[AD94] R. Alur and D. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994. DOI: 10.1016/0304-3975(94)90010-8 (cit. on pp. 3, 20–22, 26, 30, 34, 35, 79).

[AEH+10] L. Apvrille, R. El Khayari, O. Henniger, Y. Roudier, H. Schweppe, H. Seudié, B. Weyl, and M. Wolf, "Secure automotive on-board electronics network architecture," in *Proceedings of the FISITA 2010, World Automotive Congress*, B, May 2010. [Online]. Available: http://www.eurecom.fr/publication/3132 (cit. on p. 109).

[AFM+04] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: A tool for schedulability analysis and code generation of real-time systems," in *In Proceedings of the 1st International Workshop Formal Modeling and Analysis of Timed Systems*, K. G. Larsen and P. Niebert, Eds., ser. FORMATS '03. Marseille, France: Springer, 2004, pp. 60–72, ISBN: 978-3-540-40903-8. DOI: 10.1007/978-3-540-40903-8_6 (cit. on p. 158).

[AG83] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: A software science validation," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 6, pp. 639–648, Nov. 1983, ISSN: 0098-5589. DOI: 10.1109/TSE.1983.235271 (cit. on p. 205).

[AK03] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36–41, Sep. 2003, ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231149 (cit. on pp. 85, 94).

[ÅK14] K. J. Åström and P. Kumar, "Control: A perspective," *Automatica*, vol. 50, no. 1, pp. 3–43, 2014, ISSN: 0005-1098. DOI: 10.1016/j.automatica.2013.10.012 (cit. on pp. 2, 19, 23, 27, 77, 78, 162, 222).

[AKPM05] J. Ahluwalia, I. H. Krüger, W. Phillips, and M. Meisinger, "Model-based run-time monitoring of end-to-end deadlines," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05, New York, USA: ACM, 2005, pp. 100–109, ISBN: 1-59593-091-4. DOI: 10.1145/1086228.1086248 (cit. on p. 211).

[AL98] P. S. Addison and D. J. Low, "A novel nonlinear car-following model," *Chaos*, vol. 8, no. 4, pp. 791–799, 1998. DOI: 10.1063/1.166364 (cit. on p. 19).

[Ale15] A. Aleti, "Designing automotive embedded systems with adaptive genetic algorithms," *Automated Software Engineering*, vol. 22, no. 2, pp. 199–240, 2015, ISSN: 0928-8910. DOI: 10.1007/s10515-014-0148-0 (cit. on pp. 88, 90, 134–136, 144).

[All09] R. Allan, "Automotive networks strive to satisfy safety and bandwidth needs," *Electronic Design*, vol. 57, no. 21, pp. 28–33, 2009, ISSN: 0013-4872. [Online]. Available: http://electronicdesign.com/communications/automotive-networks-strive-satisfy-safety-and-bandwidth-needs (cit. on p. 94).

[ÅM08] K. J. Åström and R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton, USA: Princeton University Press, 2008, ISBN: 0691135762 (cit. on p. 20).

[AR04] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of Computer Organization and Architecture (Wiley Series on Parallel and Distributed Computing)*. Hoboken, USA: John Wiley & Sons, Inc, 2004, ISBN: 0471467413 (cit. on p. 98).

[Arb04] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, pp. 329–366, 03 Jun. 2004, ISSN: 1469-8072. DOI: 10.1017/S0960129504004153 (cit. on pp. 76, 79, 80).

[ARMS02] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Generic ilp versus specialized 0-1 ilp: An update," in *Proceedings of the IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '02, San Jose, California: ACM, 2002, pp. 450–457, ISBN: 0-7803-7607-2. DOI: 10.1145/774572.774638 (cit. on p. 114).

[AU72] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*. Upper Saddle River, USA: Prentice-Hall, Inc., 1972, ISBN: 0-13-914556-7 (cit. on p. 195).

[AVT+15] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz, "Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems," in *In the Joint Proceedings of the 8th International Workshop on Model-based Architecting of Cyber-physical and Embedded Systems and 1st International Workshop on UML Consistency Rules*, ser. ACES-MB-WUCOR '15, vol. 1508, Aachen, Germany: CEUR-WS, 2015, pp. 19–26. [Online]. Available: http://ceur-ws.org/Vol-1508/ (cit. on pp. 150, 153, 154, 211, 215).

[BB10] R. Bloomfield and P. Bishop, "Safety and assurance cases: Past, present and possible future – an adelard perspective," in *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*, C. Dale and T. Anderson, Eds. London, England: Springer, 2010, pp. 51–67, ISBN: 978-1-84996-086-1. DOI: 10.1007/978-1-84996-086-1_4 (cit. on p. 222).

[BBP+16] P. Buonocunto, A. Biondi, M. Pagani, M. Marinoni, and G. Buttazzo, "Arte: Arduino real-time extension for programming multitasking applications," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16, New York, USA: ACM, 2016, pp. 1724–1731, ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851672 (cit. on p. 179).

[BC11] E. Borde and J. Carlson, "Towards verified synthesis of procom, a component model for real-time embedded systems," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11, New York, USA: ACM, 2011, pp. 129–138, ISBN: 978-1-4503-0723-9. DOI: 10.1145/2000229.2000248 (cit. on pp. 207, 209, 210).

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03, New York, USA: ACM, 2003, pp. 196–207, ISBN: 1-58113-662-5. DOI: 10.1145/781131.781153 (cit. on p. 207).

[BCC+08] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. V. Feljan, "Procom - the progress component model reference manual, version 1.0," Mälardalen University, Tech. Rep. MDH-MRTC-230/2008-1-SE, Jun. 2008. [Online]. Available: http://www.es.mdh.se/publications/1279- (cit. on pp. 4, 158, 208, 209, 215).

[BDO+15] D. Balasubramanian, A. Dubey, W. Otte, T. Levendovszky, A. Gokhale, P. Kumar, W. Emfinger, and G. Karsai, "DREMS ml: A wide spectrum architecture design language for distributed computing platforms," *Science of Computer Programming*, vol. 106, pp. 3–29, 2015, Special Issue: Architecture-Driven Semantic Analysis of Embedded Systems, ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.04.002 (cit. on pp. 214, 215).

[BEH+04] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg, "Verification of plc programs given as sequential function charts," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science, H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, Eds., vol. 3147, Berlin Heidelberg, Germany: Springer, 2004, pp. 517–540, ISBN: 978-3-540-23135-6. DOI: 10.1007/978-3-540-27863-4_28 (cit. on p. 77).

[Ber12] A. Berg, *Jenkins Continuous Integration Cookbook*. Birmingham, UK: Packt Publishing Ltd, 2012, ISBN: 978-1-849517-40-9 (cit. on pp. 67, 132, 196).

[BFHP09] E. Borde, P. H. Feiler, G. Haïk, and L. Pautet, "Model driven code generation for critical and adaptative embedded systems," *ACM SIGBED Review - Special Issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems*, vol. 6, no. 3, 10:1–10:5, Oct. 2009, ISSN: 1551-3688. DOI: 10.1145/1851340.1851352 (cit. on pp. 212, 215).

[BFS13] M. Bonfè, C. Fantuzzi, and C. Secchi, "Design patterns for model-based automation software design and implementation," *Control Engineering Practice*, INCOM '09, vol. 21, no. 11, pp. 1608–1619, 2013, Advanced Software Engineering in Industrial Automation, ISSN: 0967-0661. DOI: 10.1016/j.conengprac.2012.03.017 (cit. on p. 177).

[BGH+07] S. Burmester, H. Giese, S. Henkler, M. Hirsch, M. Tichy, A. Gambuzza, E. Munch, and H. Vocking, "Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camel-view," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, May 2007, pp. 801–804. DOI: 10.1109/ICSE.2007.88 (cit. on pp. 77, 81).

[BGH+13] T. Bureš, I. Gerostathopoulos, P. Hnětynka, J. Keznikl, M. Kit, and F. Plasil, "Deeco: An ensemble-based component system," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '13, New York, USA: ACM, 2013, pp. 81–90, ISBN: 978-1-4503-2122-8. DOI: 10.1145/2465449.2465462 (cit. on pp. 4, 158, 210, 215).

[BGO06] S. Burmester, H. Giese, and O. Oberschelp, "Hybrid uml components for the design of complex self-optimizing mechatronic systems," in *INFORMATICS IN CONTROL, AUTOMATION AND ROBOTICS I*, J. BRAZ, H. ARAÚJO, A. VIEIRA, and B. ENCARNAÇÃO, Eds. Dordrecht, Netherlands: Springer, 2006, pp. 281–288, ISBN: 978-1-4020-4543-1. DOI: 10.1007/1-4020-4543-3_34 (cit. on pp. 2, 20).

[BGS05] S. Burmester, H. Giese, and W. Schäfer, "Model-driven architecture for hard real-time systems: From platform independent models to code," in *Proceedings of the First European Conference on Model Driven Architecture – Foundations and Applications*, A. Hartman and D. Kreische, Eds., ser. Lecture Notes in Computer Science. Berlin Heidelberg, Germany: Springer, 2005, vol. 3748, pp. 25–40, ISBN: 978-3-540-32093-7. DOI: 10.1007/11581741_4 (cit. on pp. 4, 159).

[BHK06] S. Becker, J. Happe, and H. Koziolek, "Putting components into context - supporting qos-predictions with an explicit context model," in *Proceedings of the Eleventh International Workshop on Component-Oriented Programming*, R. Reussner, C. Szyperski, and W.Weck, Eds., ser. WCOP '06, 2006, pp. 1–6. [Online]. Available: http://research.microsoft.com/en-us/um/people/msr%20web%20page%20(cszypers)/events/wcop2006/WCOP06-Becer.pdf (cit. on pp. 4, 158, 159).

[BHP09] E. Borde, G. Haïk, and L. Pautet, "Mode-based reconfiguration of critical software component architectures," in *Proceedings of the Design, Automation Test in Europe Conference & Exhibition*, ser. DATE '09, 2009, pp. 1160–1165. DOI: 10.1109/DATE.2009.5090838 (cit. on pp. 212, 213, 215).

[BHS07] F. Buschmann, K. Henney, and D. C. Schmidt, *A Pattern Language for Distributed Computing*, ser. Pattern-Oriented Software Architecture. The Atrium Southern Gate, England: John Wiley & Sons, 2007, vol. 4, ISBN: 978-0-470-05902-9 (cit. on pp. 158, 172, 173, 188, 304–306).

[BHS15] C. Brink, P. Heisig, and S. Sachweh, "Using cross-dependencies during configuration of system families," in *Product-Focused Software Process Improvement: 16th International Conference, Proceedings*, P. Abrahamsson, L. Corral, M. Oivo, and B. Russo, Eds., ser. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2015, vol. 9459 2015, pp. 439–452, ISBN: 978-3-319-26844-6. DOI: 10.1007/978-3-319-26844-6_32 (cit. on p. 223).

[BKN+05] J. Baber, J. Kolodko, T. Noel, M. Parent, and L. Vlacic, "Cooperative autonomous driving: Intelligent vehicles sharing city roads," *IEEE Robotics Automation Magazine*, vol. 12, no. 1, pp. 44–49, Mar. 2005, ISSN: 1070-9932. DOI: 10.1109/MRA.2005.1411418 (cit. on pp. 1, 6, 7).

[BKPS14] C. Brink, E. Kamsties, M. Peters, and S. Sachweh, "On hardware variability and the relation to software variability," in *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, ser. SEAA '14, New York, USA: IEEE, Aug. 2014, pp. 352–355. DOI: 10.1109/SEAA.2014.15 (cit. on p. 223).

[BLL+08] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-28, Apr. 2008. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.html (cit. on p. 79).

[BLW04] L. C. Briand, Y. Labiche, and Y. Wang, "Using simulation to empirically investigate test coverage criteria based on statechart," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, May 2004, pp. 86–95. DOI: 10.1109/ICSE.2004.1317431 (cit. on pp. 73, 206).

[BMH08] T. Bureš, M. Malohlava, and P. Hnětynka, "Using dsl for automatic generation of software connectors," in *Proceedings of the Seventh International Conference on Composition-Based Software Systems*, ser. ICCBSS '08, New York, USA: IEEE, Feb. 2008, pp. 138–147. DOI: 10.1109/ICCBSS.2008.17 (cit. on p. 179).

[BNOW93] A. Birrell, G. Nelson, S. Owicki, and E. Wobber, "Network objects," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, ser. SOSP '93, Asheville, North Carolina, USA: ACM, 1993, pp. 217–230, ISBN: 0-89791-632-8. DOI: 10.1145/168619.168637 (cit. on p. 190).

[BP02] D. Bálek and F. Plášil, "Software connectors and their role in component deployment," in *Proceedings of the Third International Working Conference on Distributed Applications and Interoperable Systems*, K. Zieliński, K. Geihs, and A. Laurentowski, Eds. Boston, USA: Springer, 2002, pp. 69–84, ISBN: 978-0-306-47005-9. DOI: 10.1007/0-306-47005-5_6 (cit. on pp. 172, 188).

[BR03] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, CSUR '03, vol. 35, no. 3, pp. 268–308, Sep. 2003, ISSN: 0360-0300. DOI: 10.1145/937503.937505 (cit. on pp. 3, 145, 148–150).

[Bre16] C. Brenner, "Szenariobasierte synthese verteilter mechatronischer systeme," Phd's Thesis, Paderborn University, Paderborn, Germany, 2016. [Online]. Available: http://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466:2-17734 (cit. on p. 222).

[BS06] J. Boardman and B. Sauser, "System of systems - the meaning of of," in *Proceedings of the IEEE/SMC International Conference on System of Systems Engineering*, ser. SoSE '06, New York, USA: IEEE, Apr. 2006, pp. 118–123. DOI: 10.1109/SYSOSE.2006.1652284 (cit. on pp. 1, 160).

[BSKA12] C. Brandt, F. Santini, N. Kokash, and F. Arbab., "Modeling and simulation of selected operational it risks in the banking sector," in *Proceedings of the European Simulation and Modelling Conference*, M. Klumpp, Ed., 2012, pp. 192–200, ISBN: 978-90-77381-73-1 (cit. on pp. 76, 79, 80).

[BU10] U. Brinkschulte and T. Ungerer, *Mikrocontroller und Mikroprozessoren*, 3rd ed., ser. eXamen.press. Berlin Heidelberg, Germany: Springer, 2010, ISBN: 978-3-642-05397-9. DOI: 10.1007/978-3-642-05398-6 (cit. on p. 96).

[Bur06a] T. Bureš, "Generating connectors for homogeneous and heterogeneous deployment," Phd's Thesis, Charles University in Prague, Sep. 2006 (cit. on pp. 179, 209).

[Bur06b] S. Burmester, "Model-driven engineering of reconfigurable mechatronic systems," Phd's Thesis, Paderborn University, Berlin, Germany, 2006, ISBN: 3-8325-1298-5. [Online]. Available: http://d-nb.info/98104803X (cit. on pp. 2, 4, 159).

[Bur13] E. J. Burger, "Flexible views for view-based model-driven development," in *Proceedings of the 18th International Doctoral Symposium on Components and Architecture*, ser. WCOP '13, New York, USA: ACM, 2013, pp. 25–30, ISBN: 978-1-4503-2125-9. DOI: 10.1145/2465498.2465501 (cit. on p. 85).

[Bur14] E. Burger, "Flexible views for view-based model-driven development," Phd's Thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, Jul. 2014, ISBN: 978-3-7315-0276-0. DOI: 10.5445/KSP/1000043437. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043437 (cit. on p. 222).

[BV15] K. Becker and S. Voss, "Analyzing graceful degradation for mixed critical fault-tolerant real-time systems," in *Proceedings of the 18th International IEEE Symposium on Real-Time Distributed Computing*, ser. ISORC '15, 2015, pp. 110–118. DOI: 10.1109/ISORC.2015.10 (cit. on p. 150).

[BvHW+15] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert, "Performance-oriented DevOps: A research agenda," SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), Tech. Rep. SPEC-RG-2015-01, Aug. 2015. [Online]. Available: http://research.spec.org/fileadmin/

user_upload/documents/wg_devops/endorsed_publications/SPEC-RG-2015-001-DevOpsPerformanceResearchAgenda.pdf (cit. on pp. 222, 223).

[BY04] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, J. Desel, W. Reisig, and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 3098, Berlin Heidelberg, Germany: Springer, 2004, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3 (cit. on pp. 3, 21, 30, 34, 158).

[CB10] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2009.10.017 (cit. on p. 190).

[CB13] M. Canale and S. C. Brunet, "A Lego Mindstorms NXT experiment for model predictive control education," in *Proceedings of the 2013 European Control Conference*, ser. ECC '13, 2013, pp. 2549–2554. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6669633 (cit. on p. 179).

[CCD+14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Proceedings of the 26th International Conference on Computer Aided Verification*, ser. CAV '14. Cham, Swiss: Springer, 2014, pp. 334–342, ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_22 (cit. on p. 211).

[CCM+12] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega, "Chess: A model-driven engineering tool environment for aiding the development of complex industrial systems," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '12, Essen, Germany: ACM, 2012, pp. 362–365, ISBN: 978-1-4503-1204-2. DOI: 10.1145/2351676.2351748 (cit. on p. 211).

[CCS13] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Round-trip support for extra-functional property management in model-driven engineering of embedded systems," *Information and Software Technology*, vol. 55, no. 6, pp. 1085–1100, 2013, ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2012.07.014 (cit. on p. 211).

[CDKB11] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th. Boston, USA: Addison-Wesley Publishing Company, 2011, ISBN: 0132143011, 9780132143011 (cit. on pp. 1, 32).

[CFJ+11] P. Cuenot, P. Frey, R. Johansson, H. Lönn, Y. Papadopoulos, M.-O. Reiser, A. Sandberg, D. Servat, R. Tavakoli Kolagari, M. Törngren, and M. Weber, "11 the east-adl architecture description language for automotive embedded software," in *Model-Based Engineering of Embedded Real-Time Systems*, ser. LNCS, vol. 6100, Berlin Heidelberg, Germany: Springer, 2011, pp. 297–307, ISBN: 978-3-642-16276-3. DOI: 10.1007/978-3-642-16277-0_11 (cit. on p. 3).

[CFMS10] J. Carlson, J. Feljan, J. Maki-Turja, and M. Sjodin, "Deployment modelling and synthesis in a component model for distributed embedded systems," in *Proceedings of the 36th EUROMICRO Conference on Conference on Software Engineering and Advanced Applications*, ser. SEAA '10, 2010, pp. 74–82. DOI: 10.1109/SEAA.2010.43 (cit. on pp. 4, 86, 147, 158, 208, 209, 215).

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge and London: MIT Press, 2000, ISBN: 978-0-262-03270-4 (cit. on p. 15).

[CGZC16] C. Cara, A. Groza, S. Zaporojan, and I. Calmicov, "Assisting drivers during overtaking using car-2-car communication and multi-agent systems," in *Proceedings of the 12th International Conference on Intelligent Computer Communication and Processing*, ser. ICCP '16, New York, USA: IEEE, Sep. 2016, pp. 293–299. DOI: 10.1109/ICCP.2016.7737162 (cit. on p. 6).

[Cha06] R. Chapman, "Correctness by construction: A manifesto for high integrity software," in *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, ser. SCS '05, Darlinghurst, Australia: Australian Computer Society, Inc., 2006, pp. 43–46, ISBN: 1-920-68237-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1151816.1151820 (cit. on pp. 2, 20, 160).

[Cha10] N. V. Chawla, "Data mining for imbalanced datasets: An overview," in *Data Mining and Knowledge Discovery Handbook*, O. Maimon and L. Rokach, Eds. Boston, USA: Springer, 2010, pp. 875–886, ISBN: 978-0-387-09823-4. DOI: 10.1007/978-0-387-09823-4_45 (cit. on p. 167).

[CHP06] J. Carlson, J. Håkansson, and P. Pettersson, "Saveccm: An analysable component model for real-time systems," *Electronic Notes in Theoretical Computer Science*, vol. 160, pp. 127–140, 2006, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2006.05.019 (cit. on p. 209).

[Cic13] F. Ciccozzi, "Automatic synthesis of heterogeneous cpu-gpu embedded applications from a uml profile," in *Proceedings of the 6th International Workshop on Model Based Architecting and Construction of Embedded Systems co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems*, ser. ACESMB '13, vol. 1084, Aachen, Germany: CEUR-WS, 2013, pp. 1–10. [Online]. Available: http://ceur-ws.org/Vol-1084/paper6.pdf (cit. on p. 210).

[CJP83] H. Crowder, E. L. Johnson, and M. Padberg, "Solving large-scale zero-one linear programming problems," *Operations Research*, vol. 31, no. 5, pp. 803–834, 1983. DOI: 10.1287/opre.31.5.803 (cit. on p. 83).

[CK06] F. E. Cellier and E. Kofman, "Differential algebraic equation solvers," in *Continuous System Simulation*, New York, USA: Springer, 2006, pp. 319–396, ISBN: 978-0-387-26102-7. DOI: 10.1007/0-387-30260-3_8 (cit. on pp. 40, 71).

[Cle96] P. C. Clements, "A survey of architecture description languages," in *Proceedings of the 8th International Workshop on Software Specification and Design*, ser. IWSSD '96, Washington, USA: IEEE Computer Society, 1996, pp. 16–25, ISBN: 0-8186-7361-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=857204.858261 (cit. on p. 146).

[CM87] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. New York, USA: Springer, 1987, ISBN: 0-387-17539-3 (cit. on p. 150).

[Crn11] I. Crnković, "Predictability and evolution in resilient systems," in *Software Engineering for Resilient Systems: Third International Workshop, SERENE 2011, Geneva, Switzerland, September 29-30, 2011. Proceedings*, E. A. Troubitsyna, Ed. Berlin Heidelberg, Germany: Springer, 2011, pp. 113–114, ISBN: 978-3-642-24124-6. DOI: 10.1007/978-3-642-24124-6_11 (cit. on p. 222).

[CS12] F. Ciccozzi and M. Sjödin, "Enhancing the generation of correct-by-construction code from design models for complex embedded systems," in *Proceedings of the 17th IEEE International Conference on Emerging Technologies Factory Automation*, ser. ETFA '12, 2012, pp. 1–4. DOI: 10.1109/ETFA.2012.6489716 (cit. on p. 211).

[CSCS13] F. Ciccozzi, M. Saadatmand, A. Cicchetti, and M. Sjödin, "An automated round-trip support towards deployment assessment in component-based embedded systems," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, ser. CBSE '13, Vancouver, British Columbia, Canada: ACM, 2013, pp. 179–188, ISBN: 978-1-4503-2122-8. DOI: 10.1145/2465449.2465450 (cit. on pp. 3, 4, 146, 179, 210, 211, 215).

[CSVC11] I. Crnković, S. Sentilles, A. Vulgarakis, and M. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, Sep. 2011, ISSN: 0098-5589. DOI: 10.1109/TSE.2010.83 (cit. on pp. 162, 179).

[CZ94] A. C. H. Chow and B. P. Zeigler, "Parallel devs: A parallel, hierarchical, modular, modeling formalism," in *Proceedings of the 26th Conference on Winter Simulation*, ser. WSC '94, Orlando, Florida, USA: Society for Computer Simulation International, 1994, pp. 716–722, ISBN: 0-7803-2109-X. [Online]. Available: http://dl.acm.org/citation.cfm?id=193201.194336 (cit. on p. 75).

[Dak65] R. J. Dakin, "A tree-search algorithm for mixed integer programming problems," *The Computer Journal*, vol. 8, no. 3, pp. 250–255, 1965. DOI: 10.1093/comjnl/8.3.250 (cit. on pp. 83, 115).

[DB08] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, Budapest, Hungary: Springer, 2008, pp. 337–340, ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 150).

[DB11] ——, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011, ISSN: 0001-0782. DOI: 10.1145/1995376.1995394 (cit. on pp. 144–146, 150).

[DBBL07] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007, ISSN: 1573-1383. DOI: 10.1007/s11241-007-9012-7 (cit. on pp. 91, 96, 97, 103, 104, 128–130).

[DBHT12] S. Dziwok, K. Bröker, C. Heinzemann, and M. Tichy, "A catalog of real-time coordination patterns for advanced mechatronic systems," Paderborn University, Germany, Paderborn, Germany, Tech. Rep. tr-ri-12-319, Feb. 2012 (cit. on pp. 11, 38–41, 166).

[DBO+05] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, "Dance: A qos-enabled component deployment and configuration engine," in *Component Deployment: Third International Working Conference, CD 2005, Grenoble, France, November 28-29, 2005. Proceedings*, A. Dearle and S. Eisenbach, Eds. Berlin Heidelberg, Germany: Springer, 2005, pp. 67–82, ISBN: 978-3-540-32281-8. DOI: 10.1007/11590712_6 (cit. on p. 161).

[DDGI14] R. Dorociak, R. Dumitrescu, J. Gausemeier, and P. Iwanek, "Specification technique CONSENS for the description of self-optimizing systems," in *Design Methodology for Intelligent Technical Systems*, ser. Lecture Notes in Mechanical Engineering, Berlin Heidelberg, Germany: Springer, 2014, ch. Methods for the Domain-Spanning Conceptual Design, pp. 119–127, ISBN: 978-3-642-45435-6. DOI: 10.1007/978-3-642-45435-6 (cit. on p. 11).

[Dea07] A. Dearle, "Software deployment, past, present and future," in *Proceedings of the Future of Software Engineering*, ser. FOSE '07, Washington, USA: IEEE Computer Society, 2007, pp. 269–284, ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.20 (cit. on pp. 146, 222).

[Dec05] J. D. Decotignie, "Ethernet-based real-time and industrial communications," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1102–1117, Jun. 2005, ISSN: 0018-9219. DOI: 10.1109/JPROC.2005.849721 (cit. on p. 189).

[Dem03] M. Dempsey, "Automatic translation of simulink models into modelica using simelica and the advancedblocks library," in *Proceedings of the 3rd International Modelica Conference*, ser. Modelica '03, The Modelica Association and Programming Environments Laboratory, Institutionen för datavetenskap, Linköpings universitet, 2003, pp. 115–124. [Online]. Available: https://modelica.org/events/Conference2003/papers/h42_Dempsey.pdf (cit. on pp. 77, 79, 80).

[Dev15] J. L. Devore, *Probability and Statistics for Engineering and the Sciences*, 8th. Bostion, USA: Cengage Learning, 2015, ISBN: 978-0538-73352-6. [Online]. Available: http://202.74.245.22:8080/xmlui/handle/123456789/591 (cit. on pp. 72, 201, 206, 316).

[DHBN08] U. Donath, J. Haufe, T. Blochwitz, and T. Neidhold, "A new approach for modeling and verification of discrete control components within a Modelica environment," in *Proceedings of the 6th Modelica Conference*, ser. Modelica '08, 2008, pp. 269–276, ISBN: 978-91-7393-513-5 (cit. on pp. 76, 79, 80).

[DHPZ08] J. Delange, J. Hugues, L. Pautet, and B. Zalila, "Code generation strategies from AADL architectural descriptions targeting the high integrity domain," in *Proceedings of the 4th European Congress Embedded Real Time Software*, ser. ERTS '08, 2008 (cit. on p. 212).

[DHT12] S. Dziwok, C. Heinzemann, and M. Tichy, "Real-time coordination patterns for advanced mechatronic systems," in *Coordination Models and Languages*, ser. Lecture Notes in Computer Science, M. Sirjani, Ed., vol. 7274, Berlin Heidelberg, Germany: Springer, 2012, pp. 166–180, ISBN: 978-3-642-30828-4. DOI: 10.1007/978-3-642-30829-1_12 (cit. on pp. 11, 38, 39).

[dLGMS13] "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle*, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. Berlin Heidelberg, Germany: Springer, 2013, pp. 1–32, ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_1 (cit. on p. 224).

[DPK08] J. Delange, L. Pautet, and F. Kordon, "Code generation strategies for partitioned systems," in *Proceedings of the 29th IEEE Real-Time Systems Symposium*, ser. RTSS '08, Barcelona, Spain: IEEE Computer Society, 2008, pp. 53–56. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01303791 (cit. on p. 212).

[DPST11] T. Dillon, V. Potdar, J. Singh, and A. Talevski, "Cyber-physical systems: Providing quality of service (qos) in a heterogeneous systems-of-systems environment," in *Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies*, ser. (DEST '11), New York, USA: IEEE, May 2011, pp. 330–335. DOI: 10.1109/DEST.2011.5936595 (cit. on pp. 1, 161).

[Dre04] I. Dressler, "Code generation from JGrafchart to Modelica," Master's Thesis, Department of Automatic Control, Lund University, Mar. 2004. [Online]. Available: http://www.control.lth.se/documents/2004/5726.pdf (cit. on pp. 77, 79, 80).

[DSTH17] D. Durisic, M. Staron, M. Tichy, and J. Hansson, "Assessing the impact of meta-model evolution: A measure and its automotive application," *Software & Systems Modeling*, pp. 1–27, 2017, ISSN: 1619-1374. DOI: 10.1007/s10270-017-0601-1 (cit. on p. 206).

[DW08] K. Dietrich and J. Winter, "Secure boot revisited," in *Proceedings of thethe 9th International Conference for Young Computer Scientists*, ser. ICYCS '08, New York, USA: IEEE, Nov. 2008, pp. 2360–2365. DOI: 10.1109/ICYCS.2008.535 (cit. on p. 224).

[Dzi17] S. Dziwok, "Specification and verification for real-time coordination protocols of cyber-physical systems," Phd's Thesis, Paderborn University, Paderborn, Germany, 2017. DOI: 10.17619/UNIPB/1-196 (cit. on pp. 2, 4, 11, 21, 32, 41, 160, 207, 216, 219, 314).

[EB10] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 307–309, ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869625 (cit. on pp. 132, 134, 196).

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, Jun. 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857078 (cit. on p. 191).

[EGMD12] H. Elmqvist, F. Gaucher, S. E. Matsson, and F. Dupont, "State machines in Modelica," in *Proceedings of the 9th International Modelica Conference*, ser. Modelica '12, Linköping University Electronic Press; Linköpings universitet, 2012, pp. 37–46, ISBN: 978-91-7519-826-2. DOI: 10.3384/ecp1207637 (cit. on pp. 76, 80).

[EHH+13] T. Eckardt, C. Heinzemann, S. Henkler, M. Hirsch, C. Priesterjahn, and W. Schäfer, "Modeling and verifying dynamic communication structures based on graph transformations," *Computer Science - Research and Development*, vol. 28, no. 1, pp. 3–22, 2013, ISSN: 1865-2042. DOI: 10.1007/s00450-011-0184-y (cit. on pp. 207, 224).

[EMK15] D. G. Esperon, A. Mehlhase, and T. Karbe, "Appending variable-structure to modelica models (wip)," in *Proceedings of the Conference on Summer Computer Simulation*, ser. SummerSim '15, San Diego, USA: Society for Computer Simulation International, 2015, pp. 1–6, ISBN: 978-1-5108-1059-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2874916.2874981 (cit. on pp. 74, 224).

[Emm00] W. Emmerich, "Software engineering and middleware: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00, Limerick, Ireland: ACM, 2000, pp. 117–129, ISBN: 1-58113-253-0. DOI: 10.1145/336512.336542 (cit. on p. 172).

[FC14] J. Feljan and J. Carlson, "Task allocation optimization for multicore embedded systems," in *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, ser. SEAA 2014, 2014, pp. 237–244. DOI: 10.1109/SEAA.2014.22 (cit. on pp. 150, 153, 155).

[FC15] E. M. Fredericks and B. H. C. Cheng, "Automated generation of adaptive test plans for self-adaptive systems," in *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '15, Florence, Italy: IEEE Press, 2015, pp. 157–168. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821357.2821385 (cit. on p. 222).

[FCS12] J. Feljan, J. Carlson, and T. Seceleanu, "Towards a model-based approach for allocating tasks to multicore processors," in *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications*, ser. SEAA 2012, 2012, pp. 117–124. DOI: 10.1109/SEAA.2012.56 (cit. on pp. 150, 153, 155).

[Fel79] S. I. Feldman, "Make — a program for maintaining computer programs," *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979, ISSN: 1097-024X. DOI: 10.1002/spe.4380090402 (cit. on p. 196).

[FG12] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, ser. SEI Series in Software Engineering. Boston, USA: Addison-Wesley Professional, 2012, ISBN: 978-0-321-88894-5 (cit. on pp. 3, 212).

[FGH06] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (aadl): An introduction," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2006-TN-011, Feb. 2006. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=7879 (cit. on p. 146).

[FGP14] Y. A. Feldman, L. Greenberg, and E. Palachi, "Simulating Rhapsody SysML blocks in hybrid models with FMI," in *Proceedings of the 10th International Modelica'2014 Conference, Lund, Sweden*, Linköping University Electronic Press; Linköpings universitet, 2014, pp. 43–52, ISBN: 978-91-7519-380-9. DOI: 10.3384/ecp1409643 (cit. on pp. 78, 80).

[FH03] M. Fränzle and C. Herde, "Proceedings of the 10th international conference on logic for programming, artificial intelligence, and reasoning," in, M. Y. Vardi and A. Voronkov, Eds., ser. LPAR '03. Berlin Heidelberg, Germany: Springer, 2003, ch. Efficient SAT Engines for Concise Logics: Accelerating Proof Search for Zero-One Linear Constraint Systems, pp. 302–316, ISBN: 978-3-540-39813-4. DOI: 10.1007/978-3-540-39813-4_22 (cit. on p. 149).

[FLD+11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," English, in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806, Berlin Heidelberg, Germany: Springer, 2011, pp. 379–395, ISBN: 978-3-642-22109-5. DOI: 10.1007/978-3-642-22110-1_30 (cit. on pp. 3, 77, 81).

[Fog95] D. Fogel, "Phenotypes, genotypes, and operators in evolutionary computation," in *Proceedings of the IEEE International Conference on Evolutionary Computation*, vol. 1, 1995, pp. 193–198, ISBN: 0-7803-2759-4. DOI: 10.1109/ICEC.1995.489143 (cit. on p. 148).

[FPK+12] K. Fischer, D. Panfilenko, J. Krumeich, M. Born, and P. Desfray, "Viewpoint-based modeling-towards defining the viewpoint concept and implications for supporting modeling tools," in *Proceedings of the Entwicklungsmethoden für Informationssysteme und deren Anwendung Fachgruppentreffen (EMISA)*, ser. Lecture Notes in Informatics, vol. 201, Gesellschaft für Informatik, Bonn, 2012, pp. 123–136, ISBN: 978-3-88579-600-8. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings206/article6773.html (cit. on pp. 12, 92).

[FRGT10] M. Fredj, A. Radermacher, S. Gerard, and F. Terrier, "Ec3m: Optimized model-based code generation for embedded distributed software systems," in *Proceedings of the 10th Annual International Conference on New Technologies of Distributed Systems*, ser. NOTERE '10, May 2010, pp. 279–284. DOI: 10.1109/NOTERE.2010.5536628 (cit. on pp. 208, 211).

[Fri14] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*, 2nd edition. The Atrium Southern Gate, England: John Wiley & Sons, 2014, ISBN: 978-1-118-85912-4 (cit. on pp. 23, 27, 64).

[FT11] J. Frieben and M. Tichy, "Automatic deployment of iec 61499 function blocks onto interconnected devices," in *In Proceedings of the SPS IPC DRIVES 2011 : Elektrische Automatisierung, Systeme und Komponenten*, ser. SPS IPC DRIVES "11, 2011, pp. 141–150. [Online]. Available: https://www.tichy.de/publications/2011/FT11.pdf (cit. on pp. 152–154).

[Gam10] A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language*, 1st. New York, USA: Springer, 2010, ISBN: 978-1-4419-0941-1 (cit. on p. 79).

[Gar05] A. Gargantini, "4 conformance testing," in *Model-Based Testing of Reactive Systems: Advanced Lectures*, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds. Berlin Heidelberg, Germany: Springer, 2005, pp. 87–111, ISBN: 978-3-540-32037-1. DOI: 10.1007/11498490_5 (cit. on pp. 70, 73, 203).

[GBB12] T. Goldschmidt, S. Becker, and E. Burger, "Towards a tool-oriented taxonomy of view-based modelling," in *Proceedings of the Modellierung 2012*, ser. Lecture Notes in Informatics, vol. 201, Gesellschaft für Informatik, Bonn, 2012, pp. 59–74, ISBN: 978-3-88579-295-6. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings201/article6680.html (cit. on pp. 12, 85, 92).

[GBK+08] A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "Model driven middleware: A new paradigm for developing distributed real-time and embedded systems," *Science of Computer Programming*, vol. 73, no. 1, pp. 39–58, Sep. 2008, ISSN: 0167-6423. DOI: 10.1016/j.scico.2008.05.005 (cit. on pp. 160, 188, 191, 220).

[GDHS15] C. Gerking, S. Dziwok, C. Heinzemann, and W. Schäfer, "Domain-specific model checking for cyber-physical systems," in *Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation*, ser. MoDeVVa '15, vol. 1514, Aachen, Germany: CEUR-WS, 2015. [Online]. Available: http://ceur-ws.org/Vol-1514/ (cit. on pp. 2, 11, 15, 20, 21, 34, 201, 207).

[Ger13] C. Gerking, "Transparent UPPAAL-based verification of MechatronicUML models," Master's Thesis, Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, 2013 (cit. on p. 15).

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-63361-2 (cit. on p. 95).

[GP11] B. Gallina and S. Punnekkat, "Fi4fa: A formalism for incompletion, inconsistency, interference and impermanence failures' analysis," in *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, ser. SEAA '11, Aug. 2011, pp. 493–500. DOI: 10.1109/SEAA.2011.80 (cit. on p. 210).

[Gre11] J. Greenyer, "Scenario-based design of mechtronic system," Phd's Thesis, Paderborn University, Paderborn, Germany, 2011. [Online]. Available: http://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466:2-7690 (cit. on p. 222).

[Gro09] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, 1st ed. Addison-Wesley Professional, 2009, ISBN: 978-0-3215-3407-1 (cit. on p. 132).

[GTB+03] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake, "Towards the compositional verification of real-time uml designs," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE '03, Helsinki, Finnland: ACM Press, Sep. 2003, pp. 38–47 (cit. on p. 11).

[HÅCT04] H. Hansson, M. Åkerholm, I. Crnković, and M. Törngren, "SaveCCM - a component model for safety-critical real-time systems," in *Proceedings of the 30th EUROMICRO Conference*, ser. EUROMICRO '04, Washington, USA: IEEE Computer Society, 2004, pp. 627–635, ISBN: 0-7695-2199-1. DOI: 10.1109/EURMIC.2004.1333431 (cit. on p. 209).

[Har13] F. Hartwich, "Bit time requirements for can fd," in *Proceedings of the International CAN Conference (iCC)*, 2013, pp. 4–17. [Online]. Available: http://www.can-cia.org/fileadmin/resources/documents/proceedings/2013_hartwich_v2.pdf (cit. on p. 130).

[Har87] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987. DOI: 10.1016/0167-6423(87)90035-9 (cit. on p. 26).

[HB13] C. Heinzemann and S. Becker, "Executing reconfigurations in hierarchical component architectures," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '13, Vancouver, British Columbia, Canada: ACM, 2013, pp. 3–12, ISBN: 978-1-4503-2122-8. DOI: 10.1145/2465449.2465452 (cit. on pp. 2, 74, 207, 224).

[HBDS15] C. Heinzemann, C. Brenner, S. Dziwok, and W. Schäfer, "Automata-based refinement checking for real-time systems," *Computer Science - Research and Development*, vol. 30, no. 3-4, pp. 255–283, 2015, ISSN: 1865-2034. DOI: 10.1007/s00450-014-0257-9 (cit. on pp. 11, 163, 207, 216).

[HBM+16] J. Holtmann, R. Bernijazov, M. Meyer, D. Schmelter, and C. Tschirner, "Integrated and iterative systems engineering and software requirements engineering for technical systems," *Journal of Software: Evolution and Process*, vol. 28, no. 9, pp. 722–743, 2016, ISSN: 2047-7481. DOI: 10.1002/smr.1780 (cit. on pp. 93, 222).

[HBV17] C. Heinzemann, S. Becker, and A. Volk, "Transactional execution of hierarchical reconfigurations in cyber-physical systems," *Software & Systems Modeling*, pp. 1–33, 2017, ISSN: 1619-1374. DOI: 10.1007/s10270-017-0583-z (cit. on p. 224).

[Hei15] C. Heinzemann, "Verification and simulation of self-adaptive mechatronic systems," Phd's Thesis, Paderborn University, Paderborn, Germany, 2015. [Online]. Available: https://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466:2-16778 (cit. on pp. 2–4, 12, 16, 20, 21, 28, 32, 68–70, 74, 76–78, 160–162, 164, 207, 216, 224, 277).

[HEJ15] D. Henneke, M. Elattar, and J. Jasperneite, "Communication patterns for cyber-physical systems," in *Proceedings of the 20th IEEE Conference on Emerging Technologies and Factory Automation*, ser. ETFA '15, 2015, pp. 1–4. DOI: 10.1109/ETFA.2015.7301623 (cit. on p. 183).

[Hen00] T. A. Henzinger, "The theory of hybrid automata," in *Verification of Digital and Hybrid Systems*, ser. NATO ASI Series, M. Inan and R. Kurshan, Eds., vol. 170, Berlin Heidelberg, Germany: Springer, 2000, pp. 265–292, ISBN: 978-3-642-64052-0. DOI: 10.1007/978-3-642-59615-5_13 (cit. on pp. 3, 19, 81).

[Hen12] S. Henkler, "Ein komponentenbasierter, modellgetriebener Softwareentwicklungsansatz für vernetzte, mechatronische Systeme," Phd's Thesis, Paderborn University, Paderborn, Germany, 2012. [Online]. Available: http://d-nb.info/1036552160 (cit. on pp. 4, 159).

[HFK+16] J. Holtmann, M. Fockel, T. Koch, D. Schmelter, C. Brenner, R. Bernijazov, and M. Sander, "The MechatronicUML requirements engineering method: Process and language," Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Tech. Rep. tr-ri-16-351, Dec. 2016. [Online]. Available: https://www.hni.uni-paderborn.de/pub/9479 (cit. on p. 222).

[HH05] S. Hansman and R. Hunt, "A taxonomy of network and computer attacks," *Computers & Security*, vol. 24, no. 1, pp. 31–43, 2005, ISSN: 0167-4048. DOI: http://dx.doi.org/10.1016/j.cose.2004.06.011 (cit. on p. 224).

[HH11] C. Heinzemann and S. Henkler, "Reusing dynamic communication protocols in self-adaptive embedded component architectures," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11, New York, USA: ACM, 2011, pp. 109–118, ISBN: 978-1-4503-0723-9. DOI: 10.1145/2000229.2000246 (cit. on p. 11).

[HHL+11] R. v. Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, and et al., "Wcet tool challenge 2011: Report," University of North Texas Libraries Government Documents Department to Digital Library, Tech. Rep. LLNL-CONF-488591, 2011. [Online]. Available: https://digital.library.unt.edu/ark:/67531/metadc864381/ (cit. on p. 96).

[HHW97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1254, Berlin Heidelberg, Germany: Springer, 1997, pp. 460–463, ISBN: 978-3-540-63166-8. DOI: 10.1007/3-540-63166-6_48 (cit. on p. 19).

[Hir08] M. Hirsch, "Modell-basierte Verifikation von vernetzten mechatronischen Systemen," Phd's Thesis, Paderborn University, Paderborn, Germany, 2008. [Online]. Available: http://digital.ub.uni-paderborn.de/hsmig/content/titleinfo/5221 (cit. on p. 2).

[HK04] D. Harel and H. Kugler, "The rhapsody semantics of statecharts (or, on the executable core of the uml)," in *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, Eds. Berlin Heidelberg, Germany: Springer, 2004, pp. 325–354, ISBN: 978-3-540-27863-4. DOI: 10.1007/978-3-540-27863-4_19 (cit. on p. 48).

[HL09] H. Hartenstein and K. P. Laberteaux, Eds., *VANET: Vehicular Applications and Inter-Networking Technologies*. The Atrium Southern Gate, England: John Wiley & Sons, Ltd, 2009, ISBN: 9780470740637. DOI: 10.1002/9780470740637 (cit. on p. 208).

[Höl09] F. Hölzl., "The autofocus 3 c0 code generator," Institut für Informatik. Technische Universität München, Tech. Rep. TUM-I0918, 2009. [Online]. Available: http://www4.in.tum.de/~hoelzlf/publications/TUM-I0918.pdf (cit. on p. 211).

[HP14] C. Heinzemann and C. Priesterjahn, "Convoy mode," in *Design Methodology for Intelligent Technical Systems*, J. Gausemeier, F.-J. Rammig, and W. Schäfer, Eds., Berlin Heidelberg, Germany: Springer, Jan. 2014, ch. 2.1.7, pp. 49–50, ISBN: 978-3-642-45435-6. DOI: 10.1007/978-3-642-45435-6 (cit. on p. 39).

[HPB+10] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava, "Comparison of component frameworks for real-time embedded systems," in *Proceedings of the 13th International ACM Sigsoft Symposium on Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, L. Grunske, R. Reussner, and F. Plasil, Eds., vol. 6092, Berlin Heidelberg, Germany: Springer, 2010, pp. 21–36, ISBN: 978-3-642-13237-7. DOI: 10.1007/978-3-642-13238-4_2 (cit. on p. 208).

[HPM+10] P. Hošek, T. Pop, M. Malohlava, P. Hnětynka, and T. Bureš, "Supporting real-time features in a hierarchical component system," Charles University in Prague, Technical Report 2010/5, 2010. [Online]. Available: http://d3s.mff.cuni.cz/publications/download/HosekPopMalohlavaHnetynkaBures-Technical2010-12-SOFAHi.pdf (cit. on pp. 4, 158, 208, 209, 215).

[HR07] G. Hamon and J. Rushby, "An operational semantics for stateflow," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 447–456, 1, 2007, ISSN: 1433-2787. DOI: 10.1007/s10009-007-0049-7 (cit. on p. 80).

[HRS13] C. Heinzemann, J. Rieke, and W. Schäfer, "Simulating self-adaptive component-based systems using MATLAB/Simulink," in *Proocedings of the 7th International Conference on Self-Adaptive and Self-Organizing Systems*, ser. SASO '13, Sep. 2013, pp. 71–80. DOI: 10.1109/SASO.2013.17 (cit. on pp. 3, 77, 224).

[HSST13] C. Heinzemann, O. Sudmann, W. Schäfer, and M. Tichy, "A discipline-spanning development process for self-adaptive mechatronic systems," in *Proceedings of the 2013 International Conference on Software and System Process*, ser. ICSSP '13, New York, USA: ACM, 2013, pp. 36–45, ISBN:

978-1-4503-2062-7. DOI: 10.1145/2486046.2486055 (cit. on pp. 4, 11, 12).

[Hwa12] M. H. Hwang, "Qualitative verification of finite and real-time devs networks," in *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, ser. TMS/DEVS '12, Orlando, Florida: Society for Computer Simulation International, 2012, 43:1–43:8, ISBN: 978-1-61839-786-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2346616.2346659 (cit. on p. 75).

[ICSK14] R. Inam, J. Carlson, M. Sjödin, and J. Kunčar, "Predictable integration and reuse of executable real-time components," *Journal of Systems and Software*, vol. 91, pp. 147–162, 2014, ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.12.040 (cit. on p. 209).

[IS12] R. Inam and M. Sjödin, "Implementing and evaluating communication-strategies in the procom component technology," *ACM SIGBED Review - Special Issue on the 24th Euromicro Conference on Real-Time Systems*, vol. 9, no. 4, pp. 41–44, Nov. 2012, ISSN: 1551-3688. DOI: 10.1145/2452537.2452545 (cit. on p. 209).

[IT97] N. Imlig and A. Tsutsui, "Performance estimation of embedded software with pipeline and cache hazard modeling," in *High Performance Computing*, ser. LNCS, vol. 1336, Berlin Heidelberg, Germany: Springer, 1997, pp. 131–142, ISBN: 978-3-540-63766-0. DOI: 10.1007/BFb0024211 (cit. on p. 98).

[JÅ99] C. Johnsson and K.-E. Årzén, "Grafchart and Grafcet: A comparison between two graphical languages aimed for sequential control applications," in *Proceedings of the 14th World Congress of the International Federation of Automatic Control (IFAC)*, vol. A, Beijing, P.R.China: IFAC, 1999, pp. 19–24. [Online]. Available: http://lup.lub.lu.se/record/7761428 (cit. on p. 77).

[JFA+07] A. A. Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas, "Robust test generation and coverage for hybrid systems," in *Proceedings of the 10th International Workshop on Hybrid Systems: Computation and Control*, A. Bemporad, A. Bicchi, and G. Buttazzo, Eds., ser. HSCC '07, Berlin Heidelberg, Germany: Springer, 2007, pp. 329–342, ISBN: 978-3-540-71493-4. DOI: 10.1007/978-3-540-71493-4_27 (cit. on p. 222).

[JJK+12] M. Jan, C. Jouvray, F. Kordon, A. Kung, J. Lalande, F. Loiret, J. Navas, L. Pautet, J. Pulou, A. Radermacher, and L. Seinturier, "Flex-eware: A flexible model driven solution for designing and implementing embedded distributed systems," *Software: Practice and Experience*, vol. 42, no. 12, pp. 1467–1494, 2012, ISSN: 1097-024X. DOI: 10.1002/spe.1143 (cit. on pp. 208, 211, 215).

[JL87] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87, Munich, West Germany: ACM, 1987, pp. 111–119, ISBN: 0-89791-215-2. DOI: 10.1145/41625.41635 (cit. on pp. 3, 83, 104, 144, 145, 150).

[JM94] J. Jaffar and M. J. Maher, "Special issue: Ten years of logic programming constraint logic programming: A survey," *The Journal of Logic Programming*, vol. 19, pp. 503–581, 1994, ISSN: 0743-1066. DOI: 10.1016/0743-1066(94)90033-7 (cit. on pp. 3, 104, 144).

[KAA+11] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter, "Miplib 2010," *Mathematical Programming Computation*, vol. 3, no. 2, p. 103, Jun. 7, 2011, ISSN: 1867-2957. DOI: 10.1007/s12532-011-0025-9 (cit. on p. 143).

[Kar72] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, Eds. Boston, USA: Springer, 1972, pp. 85–103, ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9 (cit. on p. 115).

[KBE16] A. Knauss, C. Berger, and H. Eriksson, "Towards state-of-the-art and future trends in testing of active safety systems," in *Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems*, ser. SEsCPS '16, New York, USA: ACM, 2016, pp. 36–42, ISBN: 978-1-4503-4171-4. DOI: 10.1145/2897035.2897037 (cit. on p. 207).

[KCO10] D. Kramer, T. Clark, and S. Oussena, "Mobdsl: A domain specific language for multiple mobile platform deployment," in *Proceedings of the 2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*, ser. NESEA '10, Nov. 2010, pp. 1–7. DOI: 10.1109/NESEA.2010.5678062 (cit. on p. 222).

[KK12] K. D. Kim and P. R. Kumar, "Cyber-physical systems: A perspective at the centennial," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1287–1308, May 2012, ISSN: 0018-9219. DOI: 10.1109/JPROC.2012.2189792 (cit. on p. 1).

[KKLR13] J. Kim, H. Kim, K. Lakshmanan, and R. (Rajkumar, "Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ser. ICCPS '13, New York,USA: ACM, 2013, pp. 31–40, ISBN: 978-1-4503-1996-6. DOI: 10.1145/2502524.2502530 (cit. on p. 223).

[KLMJ10] S. Kumar, M. K. Luhandjula, E. Munapo, and B. C. Jones, "Fifty years of integer programming: A review of the solution approaches," *Asia Pacific Business Review*, vol. 6, no. 3, pp. 5–15, 2010. DOI: 10.1177/097324701000600301 (cit. on pp. 3, 83, 115).

[Klo14] K. Klobedanz, "Towards the design of fault-tolerant distributed real-time systems," Phd's Thesis, Paderborn University, Paderborn, Germany, Mar. 2014. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:hbz:466:2-13767 (cit. on pp. 151, 153, 155).

[KMR02] A. Knapp, S. Merz, and C. Rauh, "Model checking timed uml state machines and collaborations," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, W. Damm and E.-R. Olderog, Eds., vol. 2469, Berlin Heidelberg, Germany: Springer, 2002, pp. 395–414, ISBN: 978-3-540-44165-6. DOI: 10.1007/3-540-45739-9_23 (cit. on pp. 76, 210, 211, 213, 216).

[Kop11] H. Kopetz, "Simplicity," in *Real-Time Systems*, ser. Real-Time Systems Series, New York, USA: Springer, 2011, pp. 29–50, ISBN: 978-1-4419-8236-0. DOI: 10.1007/978-1-4419-8237-7_2 (cit. on pp. 1, 91, 173, 223).

[KPP+15] S. Kugele, G. Pucea, R. Popa, L. Dieudonné, and H. Eckardt, "On the deployment problem of embedded systems," in *Proceedings of the ACM/IEEE International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE, 2015, pp. 158–167. DOI: 10.1109/MEMCOD.2015.7340482 (cit. on pp. 3, 84, 148, 153, 155).

[KPP95] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Software*, vol. 12, no. 4, pp. 52–62, Jul. 1995, ISSN: 0740-7459. DOI: 10.1109/52.391832 (cit. on pp. 38, 67, 134, 197).

[KR11] A. Koziolek and R. Reussner, "Towards a generic quality optimisation framework for component-based system models," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11, Boulder, Colorado, USA: ACM, 2011, pp. 103–108, ISBN: 978-1-4503-0723-9. DOI: 10.1145/2000229.2000244 (cit. on pp. 149, 153).

[Kri13] C. Krintz, "The appscale cloud platform: Enabling portable, scalable web application deployment," *IEEE Internet Computing*, vol. 17, no. 2, pp. 72–75, Mar. 2013, ISSN: 1089-7801. DOI: 10.1109/MIC.2013.38 (cit. on p. 222).

[Kru92] C. W. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, Jun. 1992, ISSN: 0360-0300. DOI: 10.1145/130844.130856 (cit. on p. 162).

[Kuc01] K. Kuchcinski, "Constraints-driven design space exploration for distributed embedded systems," *Journal of Systems Architecture*, vol. 47, no. 3-4, pp. 241–261, 2001, ISSN: 1383-7621. DOI: 10.1016/S1383-7621(00)00048-5 (cit. on pp. 151, 153).

[LCP+05] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys Tutorials*, vol. 7, no. 2, pp. 72–93, Feb. 2005, ISSN: 1553-877X. DOI: 10.1109/COMST.2005.1610546 (cit. on pp. 160, 190).

[LCZ12] L. Lednicki, I. Crnković, and M. Zagar, "Automatic synthesis of hardware-specific code in component-based embedded systems," in *Proceedings of the Seventh International Conference on Software Engineering Advances*, ser. ICSEA '12, 2012, pp. 563–570, ISBN: 978-1-61208-230-1. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=icsea_2012_20_20_10463 (cit. on p. 209).

[Led15] L. Lednicki, "Software and hardware models in component-based development of embedded systems," Phd's Thesis, Mälardalen University, HÖGSKOLEPLAN 1, 721 23 Västerås, Schweden, Jan. 2015, ISBN: 978-91-7485-180-9. [Online]. Available: http://www.es.mdh.se/publications/3884- (cit. on p. 209).

[Lee08] E. Lee, "Cyber physical systems: Design challenges," in *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, ser. ISORC '08, May 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25 (cit. on p. 21).

[Lee09] E. A. Lee, "Finite state machines and modal models in Ptolemy II," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-151, Nov. 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html (cit. on p. 77).

[Lee10] ——, "Disciplined heterogeneous modeling," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, D. Petriu, N. Rouquette, and Ø. Haugen, Eds., vol. 6395, Berlin Heidelberg, Germany: Springer, 2010, pp. 273–287, ISBN: 978-3-642-16128-5. DOI: 10.1007/978-3-642-16129-2_20 (cit. on pp. 20, 76, 77, 79–81).

[LFF10] L. Liu, F. Felgner, and G. Frey, "Comparison of 4 numerical solvers for stiff and hybrid systems simulation," in *Proceedings of the 5th Conference on Emerging Technologies Factory Automation*, ser. ETFA '10, Sep. 2010, pp. 1–8. DOI: 10.1109/ETFA.2010.5641330 (cit. on p. 61).

[LGHT08] M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich, "Efficient symbolic multi-objective design space exploration," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '08, Seoul, Korea: IEEE Computer Society Press, 2008, pp. 691–696, ISBN: 978-1-4244-1922-7. DOI: 10.1109/ASPDAC.2008.4484040 (cit. on p. 146).

[LGRT11] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4j: A modular framework for meta-heuristic optimization," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '11, Dublin, Ireland: ACM, 2011, pp. 1723–1730, ISBN: 978-1-4503-0557-0. DOI: 10.1145/2001576.2001808 (cit. on pp. 115, 150).

[Lil00] D. J. Lilja, *Measuring Computer Performance: A Practitioner's Guide*. New York, USA: Cambridge University Press, 2000, ISBN: 0-521-64105-5 (cit. on pp. 96, 98).

[LL73] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973, ISSN: 0004-5411. DOI: 10.1145/321738.321743 (cit. on pp. 91, 126, 127).

[LM95] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, ser. DAC '95, San Francisco, California, USA: ACM, 1995, pp. 456–461, ISBN: 0-89791-725-1. DOI: 10.1145/217474.217570 (cit. on p. 98).

[LMW95] Y.-T. S. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, ser. RTSS '95, Washington, USA: IEEE Computer Society, 1995, pp. 298–307, ISBN: 0-8186-7337-0. DOI: 10.1109/REAL.1995.495219 (cit. on p. 96).

[LP10] D. Le Berre and A. Parrain, "The sat4j library, release 2.2, system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00868136 (cit. on p. 149).

[Luk10] M. Lukasiewycz, "Modeling, analysis, and optimization of automotive networks," Phd's Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Göttingen, Germany, 2010, ISBN: 978-3-86955-414-3. [Online]. Available: http://d-nb.info/1005711844 (cit. on pp. 83–85, 97, 104, 105, 130, 143, 149, 153, 155).

[LV89] R. K. Lind and K. Vairavan, "An experimental investigation of software metrics and their relationship to software development effort," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 649–653, May 1989, ISSN: 0098-5589. DOI: 10.1109/32.24715 (cit. on pp. 205, 206).

[LW07] K. K. Lau and Z. Wang, "Software component models," *Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, Oct. 2007, ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70726 (cit. on pp. 161, 162).

[LY96] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, Aug. 1996, ISSN: 0018-9219. DOI: 10.1109/5.533956 (cit. on pp. 70, 73, 203).

[LZ07] E. A. Lee and H. Zheng, "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems," in *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, ser. EMSOFT '07, Salzburg, Austria: ACM, 2007, pp. 114–123, ISBN: 978-1-59593-825-1. DOI: 10.1145/1289927.1289949 (cit. on p. 19).

[Maa05] H. Maaskant, "A robust component model for consumer electronic products," in *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, ser. Philips Research, P. van der Stok, Ed., vol. 3, Dordrecht, Netherlands: Springer, 2005, pp. 167–192, ISBN: 978-1-4020-3453-4. DOI: 10.1007/1-4020-3454-7_7 (cit. on pp. 147, 214, 215).

[Mah05] Q. Mahmoud, Ed., *Middleware for Communications*, 1st. The Atrium Southern Gate, England: John Wiley & Sons, 2005, ISBN: 0-470-86206-8. DOI: 10.1002/0470862084 (cit. on pp. 1, 160, 172, 183).

[Mar12] M. Martinez, "A survey of operating systems infrastructure for embedded systems," OBJECT COMPUTING INC (OCI), SAINT LOUIS, USA, Tech. Rep. ADA557628, Feb. 2012. [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA557628 (cit. on pp. 191, 193).

[MBAG11] I. Meedeniya, B. Buhnova, A. Aleti, and L. Grunske, "Reliability-driven deployment optimization for embedded systems," *Journal of Systems and Software*, vol. 84, no. 5, pp. 835–846, 2011, ISSN: 0164-1212. DOI: 10.1016/j.jss.2011.01.004 (cit. on pp. 85, 105).

[MC04] J. Muskens and M. Chaudron, "Prediction of run-time resource consumption in multi-task component-based software systems," in *Proceedings of the 7th International ACM Sigsoft Symposium on Component-Based Software Engineering*, ser. CBSE '04, Berlin Heidelberg, Germany: Springer, 2004, pp. 162–177, ISBN: 978-3-540-24774-6. DOI: 10.1007/978-3-540-24774-6_16 (cit. on pp. 214, 215).

[MEO98] S. E. Mattsson, H. Elmqvist, and M. Otter, "Physical system modeling with Modelica," *Control Engineering Practice*, vol. 6, no. 4, pp. 501–510, 1998, ISSN: 0967-0661. DOI: 10.1016/S0967-0661(98)00047-1 (cit. on p. 23).

[Mey15] J. Meyer, "Eine durchgängige modellbasierte Entwicklungsmethodik für die automobile Steuergeräteentwicklung unter Einbeziehung des AUTOSAR Standards," Phd's Thesis, Paderborn University, Paderborn, Germany, Mar. 2015. [Online]. Available: http://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466:2-15395 (cit. on p. 214).

[MFG+17] S. J. Maher, T. Fischer, T. Gally, G. Gamrath, A. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig, "The scip optimization suite 4.0," eng, Zuse Institute Berlin, Berlin, Germany, Tech. Rep. 17-12, 2017 (cit. on pp. 115, 140).

[MH06] A. Metzner and C. Herde, "Rtsat– an optimal and efficient approach to the task allocation problem in distributed architectures," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, ser. RTSS '06, 2006, pp. 147–158. DOI: 10.1109/RTSS.2006.44 (cit. on pp. 149, 153).

[MH11] D. Martinec and Z. Hurák, "Vehicular platooning experiments with lego mindstorms nxt," in *Proceedings of the 2011 IEEE International Conference on Control Applications*, ser. CCA '11, Sep. 2011, pp. 927–932. DOI: 10.1109/CCA.2011.6044393 (cit. on p. 180).

[MK14] N. D. Matsakis and F. S. Klock II, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14, Portland, Oregon, USA: ACM, 2014, pp. 103–104, ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188 (cit. on p. 208).

[MKM+16] A. Masrur, M. Kit, V. Matěna, T. Bureš, and W. Hardt, "Component-based design of cyber-physical applications with safety-critical requirements," *Microprocessors and Microsystems*, vol. 42, pp. 70–86, 2016, ISSN: 0141-9331. DOI: 10.1016/j.micpro.2016.01.007 (cit. on pp. 1, 4, 158, 210, 215).

[MLD09] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*, 1st. Berlin Heidelberg, Germany: Springer, 2009, ISBN: 978-3-540-68899-0. DOI: 10.1007/978-3-540-68899-0 (cit. on pp. 172, 183, 186, 191).

[MMM12] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," *IEEE Transaction on Software Engineering*, vol. 38, no. 1, pp. 73–100, Jan. 2012, ISSN: 0098-5589. DOI: 10.1109/TSE.2011.3 (cit. on pp. 3, 83, 85, 86, 96, 105, 117, 144, 147, 148, 153).

[MP12] P. Manolios and V. Papavasileiou, "ILP modulo theories," *Computing Research Repository (CoRR)*, vol. abs/1210.3761, 2012. [Online]. Available: http://arxiv.org/abs/1210.3761 (cit. on p. 146).

[MS08] P. Merle and J.-B. Stefani, "A formal specification of the fractal component model in alloy," INRIA, Research Report RR-6721, 2008, p. 44. [Online]. Available: https://hal.inria.fr/inria-00338987 (cit. on p. 211).

[Muc97] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ISBN: 1-55860-320-4 (cit. on p. 96).

[MWC10] S. P. Miller, M. W. Whalen, and D. D. Cofer, "Software model checking takes off," *Communications of the ACM*, vol. 53, no. 2, pp. 58–64, Feb. 2010, ISSN: 0001-0782. DOI: 10.1145/1646353.1646372 (cit. on p. 80).

[NAY17] P. H. Nguyen, S. Ali, and T. Yue, "Model-based security engineering for cyber-physical systems: A systematic mapping study," *Information and Software Technology*, vol. 83, pp. 116–135, 2017, ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2016.11.004 (cit. on p. 224).

[NGC08] M. Nita, D. Grossman, and C. Chambers, "A theory of platform-dependent low-level software," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08, San Francisco, USA: ACM, 2008, pp. 209–220, ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328465 (cit. on p. 174).

[NT03] I. A. Niaz and J. Tanaka, "Code generation from uml statecharts," in *Proceedings of the 7th IASTED International Conference on Software Engineering and Application*, ser. SEA '03, 2003, pp. 315–321 (cit. on p. 177).

[NTAM16] A. Nazari, D. Thiruvady, A. Aleti, and I. Moser, "A mixed integer linear programming model for reliability optimisation in the component deployment problem," *JOURNAL OF THE OPERATIONAL RESEARCH SOCIETY*, vol. 67, no. 3, pp. 1–11, Mar. 2016, ISSN: 0160-5682. DOI: 10.1057/jors.2015.119 (cit. on pp. 145, 146).

[OME+09] M. Otter, M. Malmheden, H. Elmqvist, S. Mattsson, and C. Johnsson, "A new formalism for modeling of reactive and hybrid systems," in *Proceedings of the 7th Modelica Conference*, ser. Modelica '09, 2009, pp. 364–377, ISBN: 978-91-7393-513-5. DOI: 10.3384/ecp09430108 (cit. on pp. 22, 24, 25, 35–37, 75, 80).

[PBB+10] C. J. Paredis, Y. Bernard, R. M. Burkhart, H.-P. de Koning, S. Friedenthal, P. Fritzson, N. F. Rouquette, and W. Schamai, "5.5.1 an overview of the sysml-modelica transformation specification," *INCOSE International Symposium*, vol. 20, no. 1, pp. 709–722, 2010, ISSN: 2334-5837. DOI: 10.1002/j.2334-5837.2010.tb01099.x (cit. on pp. 77, 79, 80).

[PBG05] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005, ISSN: 1433-2787. DOI: 10.1007/s10009-004-0183-4 (cit. on pp. 3, 83, 115).

[PBH06] F. Plasil, T. Bureš, and P. Hnětynka, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, ser. SERA '06, Washington, USA: IEEE Computer Society, 2006, pp. 40–48, ISBN: 0-7695-2656-X. DOI: 10.1109/SERA.2006.62 (cit. on p. 208).

[PCT13] E. Palachi, C. Cohen, and S. Takashi, "Simulation of cyber physical models using SysML and numerical solvers," in *Proceedings of the 2013 IEEE International Systems Conference*, ser. SysCon '13, Apr. 2013, pp. 671–675. DOI: 10.1109/SysCon.2013.6549954 (cit. on pp. 78, 80).

[Pet82] L. R. Petzold, "A description of dassl: A differential/algebraic system solver," in *Proocedings of the 10th IMACS World Congress*, 1982, pp. 430–432 (cit. on pp. 40, 71).

[PG14] H. Pérez and J. J. Gutiérrez, "A survey on standards for real-time distribution middleware," *ACM Computing Surveys*, vol. 46, no. 4, 49:1–49:39, Mar. 2014, ISSN: 0360-0300. DOI: 10.1145/2532636 (cit. on pp. 1, 160, 190, 191).

[PHH+14] T. Pop, P. Hnětynka, P. Hošek, M. Malohlava, and T. Bureš, "Comparison of component frameworks for real-time embedded systems," *Knowledge and Information Systems*, vol. 40, no. 1, pp. 127–170, 2014, ISSN: 0219-3116. DOI: 10.1007/s10115-013-0627-9 (cit. on pp. 21, 159, 208).

[Plu06] A. R. Plummer, "Model-in-the-loop testing," *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 220, no. 3, pp. 183–199, 2006. DOI: 10.1243/09596518JSCE207 (cit. on pp. 5, 20, 27).

[PPK+11] P. Parra, O. R. Polo, M. Knoblauch, I. Garcia, and S. Sanchez, "Micobs: Multi-platform multi-model component based software development framework," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11, New York, USA: ACM, 2011, pp. 1–10, ISBN: 978-1-4503-0723-9. DOI: 10.1145/2000229.2000231 (cit. on pp. 212, 215).

[PR11] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals*, 1^{st} edition. Santa Barbara, USA: Rocky Nook, 2011, ISBN: 978-1-933952-81-9 (cit. on p. 93).

[PV02] F. Plasil and S. Visnovsky, "Behavior protocols for software components," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1056–1076, Nov. 2002, ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1049404 (cit. on pp. 208, 209).

[PW72] L. Presser and J. R. White, "Linkers and loaders," *ACM Computing Surveys*, vol. 4, no. 3, pp. 149–167, Sep. 1972, ISSN: 0360-0300. DOI: 10.1145/356603.356605 (cit. on p. 195).

[PWT+08] M. Prochazka, R. Ward, P. Tuma, P. Hnetynk, and J. Adamek, "A component-oriented framework for spacecraft on-board software," in *Proceedings of the DASIA 2008 Data Systems In Aerospace*, ser. ESA Special Publication, vol. 665, Aug. 2008, ISBN: 978-92-9221-229-2. [Online]. Available: http://adsabs.harvard.edu/abs/2008ESASP.665E..39P (cit. on pp. 4, 86, 147, 158, 179, 208, 215).

[QCG+09] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: An open-source robot operating system," in *Proceedings of the Workshop on Open Source Software*, ser. ICRA '09, 2009. [Online]. Available: http://www.robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf (cit. on p. 172).

[RCGT09] A. Radermacher, A. Cuccuru, S. Gerard, and F. Terrier, "Generating execution infrastructures for component-oriented specifications with a model driven toolchain: A case study for marte's gcm and real-time annotations," *SIGPLAN Notices*, vol. 45, no. 2, pp. 127–136, Oct. 2009, ISSN: 0362-1340. DOI: 10.1145/1837852.1621628 (cit. on pp. 208, 211, 215).

[RH08] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2008, ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8 (cit. on pp. 38, 41, 67, 73, 74, 134, 144, 197, 205).

[Rie14] J. Rieke, "Model consistency management for systems engineering," Phd's Thesis, Paderborn University, Paderborn, Germany, 2014. [Online]. Available: https://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466:2-15597 (cit. on pp. 11, 222).

[Rit93] D. M. Ritchie, "The development of the c language," in *Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL-II, Cambridge, Massachusetts, USA: ACM, 1993, pp. 201–208, ISBN: 0-89791-570-4. DOI: 10.1145/154766.155580 (cit. on p. 195).

[RLW+14] S. Ran, J. Lin, Y. Wu, J. Zhang, and Y. Xu, "Converting Ptolemy II models to SpaceEx for applied verification," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, X.-h. Sun, W. Qu, I. Stojmenovic, W. Zhou, Z. Li, H. Guo, G. Min, T. Yang, Y. Wu, and L. Liu, Eds., vol. 8630, Cham, Switzerland: Springer, 2014, pp. 669–683, ISBN: 978-3-319-11196-4. DOI: 10.1007/978-3-319-11197-1_52 (cit. on pp. 19, 77).

[Rog09] P. Rogers, "Embedded, hard, real-time systems with ada," in *Proceedings of the ACM SIGAda Annual International Conference on Ada and Related Technologies*, ser. SIGAda '09, Saint Petersburg, Florida, USA: ACM, 2009, pp. 17–18, ISBN: 978-1-60558-475-1. DOI: 10.1145/1647420.1647430 (cit. on p. 208).

[RRRW15] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, "Language and code generator composition for model-driven engineering of robotics component & connector systems," *Journal of Software Engineering for Robotics*, vol. 6, no. 1, pp. 33–57, 2015, ISSN: 2035–3928. [Online]. Available: http:

//joser.unibg.it/index.php?journal=joser&page=article&op=view&path[]=87 (cit. on p. 179).

[Sam09] M. Samek, *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*, 2nd Edition, M. Samek, Ed. Burlington, USA: Newnes Press, 2009, ISBN: 978-0-7506-8706-5. [Online]. Available: http://www.sciencedirect.com/science/book/9780750687065 (cit. on p. 177).

[SBB+02] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proceedings of the Eighth IEEE Symposium on Software Metrics*, 2002, pp. 249–258. DOI: 10.1109/METRIC.2002.1011343 (cit. on p. 20).

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse modeling framework*, 2nd edition. Boston, USA: Addison-Wesley Professional, 2008, ISBN: 978-0321331885 (cit. on pp. 67, 132, 196).

[ŠC15] I. Švogor and J. Carlson, "Scall: Software component allocator for heterogeneous embedded systems," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ser. ECSAW '15, Dubrovnik, Cavtat, Croatia: ACM, 2015, 66:1–66:5, ISBN: 978-1-4503-3393-1. DOI: 10.1145/2797433.2797501 (cit. on pp. 84, 151, 153).

[Sch06] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006, ISSN: 0018-9162. DOI: 10.1109/MC.2006.58 (cit. on p. 2).

[SCH08] D. C. Schmidt, D. A. Corsaro, and H. V. Ha, "Addressing the challenges of tactical information management in net-centric systems with dds," *Crosstalk, The Journal of Defence Software Engineering*, vol. 21, no. 3, pp. 24–29, 2008. [Online]. Available: http://www.crosstalkonline.org/storage/issue-archives/2008/200803/200803-0-Issue.pdf (cit. on pp. 189, 190).

[Sch09] W. Schamai, "Modelica modeling language (ModelicaML): A UML profile for Modelica," Linköping: Linköping University Electronic Press, Tech. Rep. tr-ri-14-337, 2009. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-20553 (cit. on pp. 76, 80).

[Sch86] A. Schrijver, *Theory of linear and integer programming*. New York, USA: John Wiley & Sons, 1986, ISBN: 0-471-90854-1 (cit. on pp. 114, 115).

[ŠCV13a] I. Švogor, I. Crnković, and N. Vrček, "Multi-criteria software component allocation on a heterogeneous platform," in *Proceedings of the 35th International Conference on Information Technology Interfaces*, ser. ITI '13, 2013, pp. 341–346. DOI: 10.2498/iti.2013.0558 (cit. on pp. 151, 153).

[ŠCV13b] I. Švogor, I. Crnković, and N. Vrček, "An extended model for multi-criteria software component allocation on a heterogeneous embedded platform," *Journal of Computing & Information Technology*, vol. 21, no. 4, pp. 211–222, 2013. DOI: 10.2498/cit.1002284 (cit. on pp. 83, 85, 105, 151, 153).

[SDD+04] J. M. Stecklein, J. Dabney, B. Dick, B. Haskins, R. Lovell, and G. Moroney, "Error cost escalation through the project life cycle," in *Proceedings of the 14th Annual International Symposium*, Houston, USA: NASA Johnson Space Center, 2004, pp. 1–11. [Online]. Available: https://ntrs.nasa.gov/search.jsp?R=20100036670 (cit. on p. 20).

[Sel03] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003, ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146 (cit. on p. 157).

[SFP13] W. Schamai, P. Fritzson, and C. J. Paredis, "Translation of UML state machines to Modelica: Handling semantic issues," *SIMULATION*, vol. 89, no. 4, pp. 498–512, 2013. DOI: 10.1177/0037549712470296 (cit. on p. 76).

[SH96] M. Saksena and S. Hong, "An engineering approach to decomposing end-to-end delays on a distributed real-time system," in *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, ser. WPDRTS '96, 1996, pp. 244–251. DOI: 10.1109/WPDRTS.1996.557688 (cit. on p. 91).

[SHG16] D. Schubert, C. Heinzemann, and C. Gerking, "Towards safe execution of reconfigurations in cyber-physical systems," in *Proceedings of the 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, ser. CBSE '16, Apr. 2016, pp. 33–38. DOI: 10.1109/CBSE.2016.10 (cit. on pp. 2, 74, 166, 207, 224).

[SHL10] B. Schätz, F. Hölzl, and T. Lundkvist, "Design-space exploration through constraint-based model-transformation," in *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, ser. ECBS'10, 2010, pp. 173–182. DOI: 10.1109/ECBS.2010.25 (cit. on pp. 150, 152–155).

[Sin11] P. Sinha, "Architectural design and reliability analysis of a fail-operational brake-by-wire system from ISO 26262 perspectives," *Reliability Engineering & System Safety*, vol. 96, no. 10, pp. 1349–1359, 2011, ISSN: 0951-8320. DOI: 10.1016/j.ress.2011.03.013 (cit. on p. 222).

[Sip06] M. Sipser, *Introduction to the Theory of Computation*, 2nd. Boston, USA: Thomson Course Technology, 2006, ISBN: 0-534-95097-3 (cit. on pp. 83, 115).

[SLCS12] M. Saadatmand, T. Leveque, A. Cicchetti, and M. Sjödin, "Managing timing implications of security aspects in model-driven development of real-time embedded systems," *International Journal On Advances in Security*, vol. 5, no. 3&4, Dec. 2012. [Online]. Available: http://www.es.mdh.se/publications/2692- (cit. on p. 224).

[SNW08] J. Seward, N. Nethercote, and J. Weidendorfer, *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Godalming, UK: Network Theory Ltd., 2008, ISBN: 0954612051, 9780954612054 (cit. on p. 102).

[SPF04] J. M. Schlesselman, G. Pardo-Castellote, and B. Farabaugh, "Omg data-distribution service (dds): Architectural update," in *Proceedings of the IEEE 2004 Military Communications Conference*, ser. MILCOM '04, vol. 2, Oct. 2004, pp. 961–967. DOI: 10.1109/MILCOM.2004.1494965 (cit. on pp. 189–191).

[SPG+06] A. V. Sánchez, O. R. Polo, O. L. Gómez, M. K. Revuelta, S. S. Prieto, and D. M. Luna, "Edroom: A free tool for the UML2 component based design and automatic code generation of tiny embedded real time system," in *Proceedings of the 3rd European Congress on Embedded Real-Time Software*, ser. ERTS '06, 2006, pp. 1–5 (cit. on p. 213).

[SSC13] G. Sapienza, T. Secelanu, and I. Crnković, "Modelling for hardware and software partitioning based on multiple properties," in *Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications*, ser. SEAA '13, Sep. 2013, pp. 189–194. DOI: 10.1109/SEAA.2013.56 (cit. on p. 147).

[Sta01] R. M. Stallman, *Using and Porting the GNU Compiler Collection*. Boston, USA: Free Software Foundation, 2001, ISBN: N 1-882114-37-X (cit. on pp. 173, 174, 195).

[SUD08] V. Sanz, A. Urquia, and S. Dormido, "Introducing messages in Modelica for facilitating discrete-event system modeling," in *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, ser. EOOLT '08, Linköping University Electronic Press; Linköpings universitet, 2008, pp. 83–93, ISBN: 978-91-7519-823-1. [Online]. Available: http://www.ep.liu.se/ecp_article/index.en.aspx?issue=029;article=009 (cit. on pp. 22, 75).

[SUD09] V. Sanz, A. Urquia, and S. Dormido, "Parallel DEVS and process-oriented modeling in Modelica," in *Proceedings of the 7th Modelica'2009 Conference, Como, Italy*, 2009, pp. 96–107, ISBN: 978-91-7393-513-5. DOI: 10.3384/ecp09430104 (cit. on pp. 75, 79, 80).

[SVC06] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. The Atrium Southern Gate, England: John Wiley & Sons, 2006, ISBN: 978-0-470-02570-3 (cit. on pp. 157, 160, 161, 168, 220).

[SW07] W. Schäfer and H. Wehrheim, "The challenges of building advanced mechatronic systems," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 72–84. DOI: 10.1109/FOSE.2007.28 (cit. on p. 157).

[SW10] H. Saadawi and G. Wainer, "Rational time-advance devs (rta-devs)," in *Proceedings of the 2010 Spring Simulation Multiconference*, ser. SpringSim '10, Orlando, Florida: Society for Computer Simulation International, 2010, 143:1–143:8, ISBN: 978-1-4503-0069-8. DOI: 10.1145/1878537.1878686 (cit. on p. 75).

[SWYS11] J. Shi, J. Wan, H. Yan, and H. Suo, "A survey of cyber-physical systems," in *Proceedings of the International Conference on Wireless Communications and Signal Processing*, ser. WCSP '11, Nov. 2011, pp. 1–6. DOI: 10.1109/WCSP.2011.6096958 (cit. on p. 73).

[Szy02] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd. Boston, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0-2017-4572-0 (cit. on p. 162).

[TB14] S. Tripakis and D. Broman, "Bridging the semantic gap between heterogeneous modeling formalisms and FMI," Department of Informatics, Faculdade de Ciências, University of Lisbon, Tech. Rep. ADA605553, Apr. 2014. [Online]. Available: http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA605553 (cit. on pp. 22, 78–81).

[TBW92] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating hard real-time tasks: An np-hard problem made easy," *Real-Time Systems*, vol. 4, no. 2, pp. 145–165, 1992, ISSN: 1573-1383. DOI: 10.1007/BF00365407 (cit. on p. 149).

[TBW95] K. Tindell, A. Burns, and A. J. Wellings, "Analysis of hard real-time communications," *Real-Time Systems*, vol. 9, no. 2, pp. 147–171, 1995, ISSN: 1573-1383. DOI: 10.1007/BF01088855 (cit. on p. 14).

[THHO08] M. Tichy, S. Henkler, J. Holtmann, and S. 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 (OMER 4), Paderborn, Germany*, ser. Verlagsschriftenreihe des Heinz Nixdorf Instituts, Paderborn, vol. 236, Paderborn, Germany: Heinz Nixdorf Institut, 2008, pp. 27–38 (cit. on pp. 2, 224).

[Tic09] M. Tichy, "Gefahrenanalyse selbstoptimierender systeme," Phd's Thesis, Paderborn University, Paderborn, Germany, Mar. 2009. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:hbz:466-20090508447 (cit. on pp. 2, 164).

[TJ14] A. Theorin and C. Johnßon, "Extending JGrafchart with support for FMI for co-simulation," in *Proceedings of the 10th International Modelica Conference*, ser. Modelica '14, Linköping University Electronic Press; Linköpings universitet, 2014, pp. 1257–1263, ISBN: 978-91-7519-380-9. DOI: 10.3384/ecp140961257 (cit. on pp. 22, 78–80).

[TSZ09] R. Toledo-Moreo, J. Santa, and M. Á. Zamora-Izquierdo, "A cooperative overtaking assistance system," in *Proceedings of the 3rd Workshop on Planning, Perception and Navigation for Intelligent Vehicles*, ser. PPNIV '09, Nantes, France: IRCCyN-CNRS Laboratory, 2009, pp. 50–55. [Online]. Available: ants.inf.um.es/~josesanta/doc/IROS09.pdf (cit. on pp. 1, 6, 7).

[UMMW12] A. Urquia, C. Martin-Villalba, M. Moallemi, and G. A. Wainer, "DEVS graph in Modelica for real-time simulation," in *Proocedings of the 26th European Conference on Modelling and Simulation*, ser. ECMS '12, 2012, pp. 157–163. DOI: 10.7148/2012-0157-0163 (cit. on p. 75).

[UPL12] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012, ISSN: 1099-1689. DOI: 10.1002/stvr.456 (cit. on pp. 67, 70, 73, 132, 196, 200, 203, 205).

[VBD+13] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth, *Designing, Implementing and Using Domain-Specific Languages*, 1st. CreateSpace Independent Publishing Platform, 2013, ISBN: 1-4812-1858-1 (cit. on pp. 2, 91).

[VBK10] K. Venkatesh Prasad, M. Broy, and I. Krueger, "Scanning advances in aerospace & automobile software technology," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 510–514, Apr. 2010, ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2041835 (cit. on p. 1).

[VEH14] S. Voss, J. Eder, and F. Hölzl, "Design space exploration and its visualization in AUTOFOCUS3," in *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering*, vol. 1514, Aachen, Germany: CEUR-WS, 2014, pp. 57–66. [Online]. Available: http://ceur-ws.org/Vol-1129/paper33.pdf (cit. on pp. 3, 83, 145, 150).

[vGH17] R. van Glabbeek and P. Höfner, "Split, send, reassemble: A formal specification of a CAN bus protocol stack," in *2nd Workshop on Models for Formal Analysis of Real Systems*, ser. MARS '17, 2017, pp. 14–52. [Online]. Available: 10.4204/EPTCS.244.2 (cit. on p. 103).

[vHWH12] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12, Boston, USA: ACM, 2012, pp. 247–248, ISBN: 978-1-4503-1202-8. DOI: 10.1145/2188286.2188326 (cit. on p. 222).

[VS12] S. Voss and B. Schätz, "Scheduling shared memory multicore architectures in af3 using satisfiability modulo theories," in *Dagstuhl-Workshop: Modellbasierte Entwicklung eingebetteter Systeme VIII, Schloss Dagstuhl, Germany, 2012, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, ser. MBEES '12, 2012, pp. 49–56. [Online]. Available: https://www.in.tu-clausthal.de/fileadmin/homes/GI/Documents/MBEES12Proceedings.pdf (cit. on p. 150).

[VS13] S. Voss and B. Schätz, "Deployment and scheduling synthesis for mixed-critical shared-memory applications," in *Proceedings of the 20th IEEE International Conference and Workshops on the Engineering of Computer Based Systems*, ser. ECBS ' 13, Apr. 2013, pp. 100–109. DOI: 10.1109/ECBS.2013.23 (cit. on p. 211).

[VSC+09] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson, "Formal semantics of the procom real-time component model," in *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, ser. SEAA '09, Aug. 2009, pp. 478–485. DOI: 10.1109/SEAA.2009.53 (cit. on p. 209).

[VSK05] M. Völter, C. Salzmann, and M. Kircher, "Component-based software development for embedded systems: An overview of current research trends," in, C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. Model Driven Software Development in the Context of Embedded Component Infrastructures, pp. 143–163, ISBN: 978-3-540-31614-5. DOI: 10.1007/11591962_8 (cit. on pp. 157, 158, 160, 172, 220).

[VSR07] F. J. Van der Linden, K. Schmid, and E. Rommes, *Software product lines in action: The best industrial practice in product line engineering*. Springer, 2007, ISBN: 978-3-540-71436-1 (cit. on p. 223).

[VSW02] M. Völter, A. Schmid, and E. Wolff, *Server Component Patterns: Component Infrastructures Illustrated with EJB*, 1st, ser. Wiley Software Patterns Series. The Atrium Southern Gate, England: John Wiley & Sons, 2002, ISBN: 978-0-470-84319-2 (cit. on pp. 158, 170, 172, 173, 306, 307).

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–52, May 2008, ISSN: 1539-9087. DOI: 10.1145/1347375.1347389 (cit. on pp. 96, 103).

[WH07] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *IET Software*, vol. 1, no. 5, pp. 172–179, Oct. 2007, ISSN: 1751-8806. DOI: 10.1049/iet-sen:20060076 (cit. on p. 222).

[Wil11] E. D. Willink, "Re-engineering eclipse mdt/ocl for xtext," *Electronic Communications of the EASST*, vol. 36, 2011, ISSN: 1863-2122. DOI: 10.14279/tuj.eceasst.36.444 (cit. on p. 134).

[WMS04] S. Wang, J. R. Merrick, and K. G. Shin, "Component allocation with multiple resource constraints for large embedded real-time software design," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS 2004, 2004, pp. 219–226. DOI: 10.1109/RTTAS.2004.1317267 (cit. on pp. 151, 153).

[WTM09] T. L. Willke, P. Tientrakool, and N. F. Maxemchuk, "A survey of inter-vehicle communication protocols and their applications," *IEEE Communications Surveys Tutorials*, vol. 11, no. 2, pp. 3–20, 2009, ISSN: 1553-877X. DOI: 10.1109/SURV.2009.090202 (cit. on p. 189).

[WW98] H. P. Williams and J. M. Wilson, "Connections between integer linear programming and constraint logic programming—an overview and introduction to the cluster of articles," *INFORMS Journal on Computing*, vol. 10, no. 3, pp. 261–264, 1998, ISSN: 1526-5528. DOI: 10.1287/ijoc.10.3.261 (cit. on pp. 104, 144).

[XMDS08] F. Xia, L. Ma, J. Dong, and Y. Sun, "Network QoS management in cyber-physical systems," in *Proceedings of the International Conference on Embedded Software and Systems Symposia*, ser. ICESS '08, Jul. 2008, pp. 302–307. DOI: 10.1109/ICESS.Symposia.2008.84 (cit. on p. 190).

[YLW09] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009. DOI: 10.1093/comjnl/bxm021 (cit. on p. 222).

[YRBK15] L. H. Yoong, P. S. Roop, Z. E. Bhatti, and M. M. Kuo, *Model-Driven Design Using IEC 61499–A Synchronous Approach for Embedded and Automation Systems*, 1st. Springer, 2015, ISBN: 978-3-319-10521-5. DOI: 10.1007/978-3-319-10521-5 (cit. on p. 144).

[ZKP00] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd. Orlando, FL, USA: Academic Press, Inc., 2000, ISBN: 0127784551 (cit. on pp. 75, 79, 80).

[ZLB07] G. Zauner, D. Leitner, and F. Breitenecker, "Modeling structural - dynamics systems in modelica/dymola; modelica/mosilab and anylogic," in *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, ser. EOOLT '07, Linköping, Sweden: Linköping University Electronic Press, 2007, pp. 99–110 (cit. on pp. 74, 224).

[ZP13] M. Zeller and C. Prehofer, "Modeling and efficient solving of extra-functional properties for adaptation in networked embedded real-time systems," *Journal of Systems Architecture*, vol. 59, no. 10, Part C, pp. 1067–1082, 2013, Embedded Systems Software Architecture, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2012.11.003 (cit. on pp. 83, 85, 91, 96, 102, 104, 105, 115, 124, 126, 128, 155).

[ZPW+11] M. Zeller, C. Prehofer, G. Weiss, D. Eilers, and R. Knorr, "Towards self-adaptation in real-time, networked systems: Efficient solving of system constraints for automotive embedded systems," in *Proceedings of the 5th International IEEE Conference on Self-Adaptive and Self-Organizing Systems*, ser. SASO '11, 2011, pp. 79–88. DOI: 10.1109/SASO.2011.19 (cit. on pp. 148, 149, 153).

[ZS14] W. Zimmermann and R. Schmidgall, "Bussysteme in der Fahrzeugtechnik," in, 5th, ser. ATZ/MTZ-Fachbuch. Wiesbaden, Germany: Springer, 2014, ch. Software-Standards: OSEK und HIS, pp. 331–365, ISBN: 978-3-658-02419-2. DOI: 10.1007/978-3-658-02419-2_7 (cit. on p. 83).

Norms and Specifications

[ALF] OMG ALF, *Action language for foundational uml (alf)*, Specification, version 1.0, Object Managment Group, 2011. [Online]. Available: http://www.omg.org/spec/ALF/1.0/Beta2/ (cit. on pp. 15, 210).

[AUTOSAR12a] AUTOSAR development cooperation, *Software component template*, Specification, version 4.2, Part of Release 4.0, AUTOSAR development cooperation, 2012. [Online]. Available: https://www.autosar.org/fileadmin/files/standards/classic/4-0/methodology-templates/templates/standard/AUTOSAR_TPS_SoftwareComponentTemplate.pdf (cit. on p. 213).

[AUTOSAR12b] ——, *Specification of timing extensions*, Specification, version 1.2, Part of Release 4.0, AUTOSAR development cooperation, 2012. [Online]. Available: https://www.autosar.org/fileadmin/files/standards/classic/4-0/methodology-templates/templates/standard/AUTOSAR_TPS_TimingExtensions.pdf (cit. on p. 214).

[AUTOSAR12c] ——, *System template*, Specification, version 4.2, Part of Release 4.0, AUTOSAR development cooperation, 2012. [Online]. Available: https://www.autosar.org/fileadmin/files/standards/classic/4-0/methodology-templates/templates/standard/AUTOSAR_TPS_SystemTemplate.pdf (cit. on p. 214).

[AUTOSAR14a] ——, *General specification of basic software modules*, Specification, version Part of Release 4.2, AUTOSAR development cooperation, 2014. [Online]. Available: https://www.autosar.org/fileadmin/files/standards/classic/4-2/software-architecture/general/standard/AUTOSAR_SWS_BSWGeneral.pdf (cit. on p. 214).

[AUTOSAR14b] ——, *Layered software architecture*, Specification, version Part of Release 4.2, AUTOSAR development cooperation, 2014. [Online]. Available: https://www.autosar.org/fileadmin/files/standards/classic/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (cit. on p. 214).

[AUTOSAR14c] ——, *Release 4.2 overview and revision history*, Specification, version 4.2, AUTOSAR development cooperation, 2014. [Online]. Available: http://www.autosar.org/specifications/release-42/ (cit. on pp. 4, 158, 172, 183, 186, 213, 215).

[AUTOSAR14d] ——, *Specification of ecu resource template*, Specification, version Part of Release 4.2, AUTOSAR development cooperation, 2014. [Online]. Available: https://www.autosar.org/fileadmin/files/standards/classic/4-2/methodology-and-templates/templates/standard/AUTOSAR_TPS_ECUResourceTemplate.pdf (cit. on p. 147).

[AUTOSAR14e] ——, *Specification of rte*, Specification, version Part of Release 4.2, AUTOSAR development cooperation, 2014. [Online]. Available: https://www.autosar.org/fileadmin/files/standards/classic/4-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf (cit. on p. 214).

[BPMN] OMG BPMN, *Business process model and notation*, Specification, version 2.0.2, Object Managment Group, 2013. [Online]. Available: http://www.omg.org/spec/BPMN/2.0.2/PDF (cit. on pp. 11, 28, 92, 113, 168).

[CORBA] OMG CORBA, *Common object request broker architecture*, Specification, version 3.3, Object Managment Group, 2012. [Online]. Available: http://www.omg.org/spec/CORBA/3.3/ (cit. on pp. 158, 172, 190, 191, 211–213).

[CORBAe] ——, *Common object request broker architecture for embedded*, Specification, version 1.0, Object Managment Group, 2008. [Online]. Available: http://www.omg.org/spec/CORBAe/1.0/ (cit. on pp. 158, 172, 211).

[DDS] OMG DDS, *Data distribution service*, Specification, version 1.4, Object Managment Group, 2015. [Online]. Available: http://www.omg.org/spec/DDS/1.4/ (cit. on pp. 161, 172, 173, 183, 184, 186, 189–191).

[EAST-ADL] MODELISAR Consortium, *East-adl, domain model specification*, version 2.1.12, EAST-ADL Association, 2013. [Online]. Available: http://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf (cit. on pp. 146, 147, 153, 210).

[EJB06] EJB 3.0 Expert Group, *Jsr 220: enterprise javabeans*, Specification, version 3.0, Sun Microsystems, 2006. [Online]. Available: http://download.oracle.com/otndocs/jcp/ejb-3_0-fr-eval-oth-JSpec/ (cit. on pp. 158, 172).

[FMI4CS] MODELISAR Consortium, *Functional mock-up interface for co-simulation*, version 1.0, Modelica Association, 2010. [Online]. Available: https://svn.modelica.org/fmi/branches/public/specifications/v1.0/FMI_for_CoSimulation_v1.0.pdf (cit. on pp. 75, 78).

[FMI4ME] ——, *Functional mock-up interface for model exchange*, version 1.0, Modelica Association, 2010. [Online]. Available: https://svn.modelica.org/fmi/branches/public/specifications/v1.0/FMI_for_ModelExchange_v1.0.pdf (cit. on pp. 75, 78).

[IEC61131] IEC 61131-3, *Programmable controllers - part 3: Programming languages*, Norm, Geneva, Switzerland: International Electrotechnical Commission, 2013. [Online]. Available: https://www.iec-normen.de/219666/iec-61131-3-2013-02-ed-3-0-zweisprachig.html (cit. on p. 77).

[IEC61499] IEC 61499-1, *Function blocks - part 1: Architecture*, Norm, Geneva, Switzerland: International Electrotechnical Commission, 2012. [Online]. Available: https://www.iec-normen.de/219342/iec-61499-1-2012-11-ed-2-0-zweisprachig.html (cit. on p. 144).

[IEC61508] IEC 61508, *Functional safety of electrical/ electronic/ programmable electronic safety-related systems - part 3: Software requirements*, Norm, Geneva, Switzerland: International Electrotechnical Commission, 2010. [Online]. Available: http://www.iec-normen.de/217179/iec-61508-3-2010-04-ed-2-0-zweisprachig.html (cit. on pp. 1, 19).

[ISO11898] ISO 11898-1:2003, *Road vehicles – controller area network (CAN) – part 1: Data link layer and physical signalling*, Norm, Geneva, Switzerland: International Organization for Standardization, 2003. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=33422 (cit. on pp. 97, 190).

[ISO15408] ISO/IEC 15408-1:2009, *Information technology – security techniques – evaluation criteria for it security – part 1: Introduction and general model*, Norm, Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, 2009. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=50341 (cit. on p. 224).

[ISO17458] ISO 17458-2:2013, *Road vehicles – FlexRay communications system – part 2: Data link layer specification*, Norm, Geneva, Switzerland: International Organization for Standardization, 2013. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=33422 (cit. on pp. 97, 130).

[ISO19503] ISO/IEC 19503:2005, *Information technology – xml metadata interchange (xmi)*, Norm, Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, 2005. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=32622 (cit. on p. 203).

[ISO25010] ISO/IEC 25010:2011, *Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models*, Norm, Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, 2011. [Online]. Available: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733` (cit. on pp. 2, 19, 180).

[ISO26262-1] ISO 26262-1:2011, *Road vehicles – functional safety – part 1: Vocabulary*, Norm, Geneva, Switzerland: International Organization for Standardization, 2011. [Online]. Available: `http://www.iso.org/iso/catalogue_detail?csnumber=43464` (cit. on pp. 83, 109).

[ISO26262-6] ISO 26262-6:2011, *Road vehicles – functional safety – part 6: Product development at the software level*, Norm, Geneva, Switzerland: International Organization for Standardization, 2011. [Online]. Available: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733` (cit. on pp. 1, 19, 145).

[ISO27001] ISO/IEC 27001:2013, *Information technology – security techniques – information security management systems – requirements*, Norm, Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, 2013. [Online]. Available: `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=54534` (cit. on p. 224).

[ISO42010] ISO/IEC/IEEE 42010:2011, *Systems and software engineering – architecture description*, Norm, Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, 2011. [Online]. Available: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=50508` (cit. on p. 92).

[ISO9899] ISO/IEC 9899:2011, *Information technology – programming languages – C*, Norm, Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, 2011. [Online]. Available: `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=57853` (cit. on pp. 173, 174, 195).

[ISO9945] POSIX, *Ieee std 1003.1, iso/iec 9945:2009*, Norm, The IEEE, The Open Group, International Organization for Standardization, and International Electrotechnical Commission, 2013. [Online]. Available: `http://pubs.opengroup.org/onlinepubs/9699919799/toc.htm` (cit. on p. 173).

[MARTE] OMG MARTE, *Uml profile for marte: modeling and analysis of real-time embedded systems*, Specification, version 1.1, Object Managment Group, 2011. [Online]. Available: `http://www.omg.org/spec/MARTE/1.1/PDF` (cit. on pp. 3, 4, 84, 86, 95, 96, 146, 153, 179, 180, 210, 211, 213).

[MODELICA] Modelica, *Modelica - a unified object-oriented language for physical systems modeling, language specification*, Specification, version 3.2, Modelica, Association, 2010. [Online]. Available: `http://www.modelica.`

org/documents/ModelicaSpec32.pdf (cit. on pp. 3, 20, 23, 29, 61, 68, 70).

[OCL] OMG OCL, *Object constraint language (ocl)*, Specification, version 2.4, Object Managment Group, 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4/PDF/ (cit. on pp. 105, 107, 132).

[OMG03] OMG MDA, *Mda guide*, ed. by J. Miller and J. Mukerji, Specification, version 1.0.1, Object Managment Group, 2003. [Online]. Available: www.omg.org/cgi-bin/doc?omg/03-06-01 (cit. on pp. 157, 158, 160, 168, 220).

[OMG06] OMG DEPL, *Deployment and configuration of component-based distributed applications*, Specification, version 4.0, Object Managment Group, 2006. [Online]. Available: http://www.omg.org/spec/DEPL/4.0/PDF (cit. on pp. 146, 213, 215).

[OPCUAPart2] OPC Foundation, *Opc unified architecture specification part 2: Security model*, Specification, version 1.1, Scottsdale, USA: OPC Foundation, 2015. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-2-security-model (cit. on p. 224).

[SysML] OMG SysML, *Systems modeling language (sysml)*, Specification, version 1.4, Object Managment Group, 2015. [Online]. Available: http://www.omg.org/spec/SysML/1.4/PDF/ (cit. on pp. 21, 77, 210).

[SysML4Modelica] OMG SysML-Modelica, *Sysml-modelica transformation*, Specification, version 1.0, Object Managment Group, 2012. [Online]. Available: http://www.omg.org/spec/SyM/1.0/PDF/ (cit. on pp. 77, 79, 80).

[TCP] J. Postel, Ed., *Rfc793: Transmission control protocol*, Information Sciences Institute, University of Southern California, California, USA, 1981 (cit. on pp. 97, 172, 183, 189, 190).

[UDP] J. Postel, Ed., *Rfc 768: User datagram protocol*, Information Sciences Institute, University of Southern California, California, USA, 1980 (cit. on pp. 97, 172, 183, 189).

[UML] OMG UML, *Unified modeling language (uml)*, Superstructure Specification, version 2.4.1, Object Managment Group, 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/ (cit. on pp. 2, 4, 20, 48, 77, 78, 84, 130, 210, 211, 213).

Tools, Software Platforms, and Hardware Platforms

[Acc] Acceleo. [Online]. Available: http://www.eclipse.org/acceleo/ (cit. on pp. 30, 67, 196, 197).

[Ard] Arduino. [Online]. Available: https://www.arduino.cc/ (cit. on pp. 88, 97, 179).

[CAM] CAMeLView. [Online]. Available: http://www.ixtronics.de/ (cit. on p. 77).

[Chi] T. Chikamasa. [Online]. Available: http://lejos-osek.sourceforge.net/ecrobot_c_api_frame.htm (cit. on p. 180).

[CLOC] A. Danial. [Online]. Available: https://github.com/AlDanial/cloc (cit. on pp. 200, 203).

[CPLEX] IBM's ILOG CPLEX Optimization Studio. [Online]. Available: http://ibm.com/software/products/ (cit. on p. 152).

[DOT] DOT. [Online]. Available: http://www.graphviz.org/content/dot-language (cit. on p. 29).

[Dym] Dymola. [Online]. Available: http://www.3ds.com/products-services/catia/products/dymola (cit. on pp. 20, 21, 26, 27, 30, 39–41, 73, 219).

[GCC] GCC. [Online]. Available: https://gcc.gnu.org/ (cit. on pp. 178, 195).

[GRAPHVIZ] Graphviz. [Online]. Available: http://www.graphviz.org/ (cit. on pp. 29, 67).

[GUR] GUROBI. [Online]. Available: https://www.gurobi.com (cit. on p. 115).

[LPS] LPSolve. [Online]. Available: http://lpsolve.sourceforge.net/5.5/ (cit. on pp. 115, 118, 134, 145).

[Mic] Microsoft. [Online]. Available: https://developer.microsoft.com/en-us/windows/iot (cit. on p. 179).

[NEMIVER] Nemiver. [Online]. Available: https://wiki.gnome.org/Apps/Nemiver (cit. on p. 202).

[Ope] OpenDSE. [Online]. Available: http://opendse.sourceforge.net/ (cit. on p. 149).

[OPENDDS] Object Computing Incorporated. [Online]. Available: http://opendds.org/ (cit. on pp. 197, 200, 322).

[OPT4J] Opt4J. [Online]. Available: http://http://opt4j.sourceforge.net// (cit. on pp. 115, 118, 134, 145, 146).

[PMAP] A. Cahalan. [Online]. Available: https://linux.die.net/man/1/pmap (cit. on p. 102).

[PT] U. Pohlmann and S. Thiele. [Online]. Available: https://github.com/modelica-3rdparty/RealTimeCoordinationLibrary/blob/%20master/CoordinationPattern.mo (cit. on p. 38).

[PWD] U. Pohlmann, B. Wolf, and S. Dziwok. [Online]. Available: https://github.com/modelica-3rdparty/RealTimeCoordinationLibrary (cit. on pp. 30, 39, 41).

[QVT] QVTo. [Online]. Available: http://projects.eclipse.org/projects/modeling.mmt.qvt-oml (cit. on pp. 29, 67, 118, 134, 197).

[Rasa] RaspberryPi. [Online]. Available: https://www.raspberrypi.org/ (cit. on pp. 161, 179, 199).

[Rasb] Raspbian. [Online]. Available: https://www.raspbian.org/ (cit. on pp. 179, 199).

[RTIa] RTI. [Online]. Available: https://www.rti.com/products/dds/ (cit. on pp. 199, 200, 204, 309, 310, 322, 323).

[RTIb] ——, [Online]. Available: https://community.rti.com/content/forum-topic/howto-run-rti-connext-dds-raspberry-pi (cit. on pp. 199, 310).

[SCI] SCIP. [Online]. Available: `http://scip.zib.de/` (cit. on pp. 115, 118, 134, 140, 145).

[Sim] Simulink. [Online]. Available: `http://www.mathworks.com/products/simulink/` (cit. on pp. 3, 20, 150).

[Sym] Symisc Systems. [Online]. Available: `https://unqlite.org` (cit. on p. 184).

[UPP] UPPAAL. [Online]. Available: `http://www.uppaal.org/` (cit. on pp. 11, 15, 69, 158).

[Wir] WiringPi. [Online]. Available: `http://wiringpi.com/` (cit. on p. 199).

[Xte] Xtext. [Online]. Available: `https://eclipse.org/Xtext/` (cit. on p. 29).

List of Figures

List of Tables

List of Acronyms

AADL Architecture Analysis and Design Language
ABS Anti-lock Braking System
ADL Architecture Description Language
API Application Programming Interface
ASL Allocation Specification Language
AUTOSAR AUTomotive Open System ARchitecture
BPMN Business Process Model and Notation
CACC Cooperative Adaptive Cruise Control
CAN Control Area Network
CIC Component Instance Configuration
CISC Complex Instruction Set Computer
CLP Constraint Logic Programming
CORBA Common Object Request Broker Architecture
COTS Commercial Off-The-Shelf
CPI Cycles Per Instruction
CPN Coloured Petri Net
CPS Cyber-Physical System
DAE Differential Algebraic Equation
DDS Data Distribution Service
DEVS Discrete EVent System specification
DSL Domain-Specific Language
DSE Design Space Exploration
EBNF Extended Backus–Naur Form
ECU Electronic Control Unit
EDF Earliest Deadline First
EJB Enterprise JavaBeans
EMF Eclipse Modeling Framework
FIFO First in, First Out
FMI Functional Mockup Interface
FMU Functional Mockup Unit
FPGA Field Programmable Gate Array
GCC GNU Compiler Collection
GMF Graphical Modeling Framework
GPIO General-Purpose Input/Output
HPDL Hardware Platform Description Language
IEC International Electrotechnical Commission
LIN Local Interconnect Network
LOC Lines of Code
IDL Interface Definition Language
ILP Integer Linear Programming
ISO International Organization for Standardization
M2M Model-to-Model
M2T Model-to-Text

MARTE Modeling and Analysis of Real-Time Embedded Systems
MDA Model-Driven Architecture
MDE Model-Driven Engineering
MiL Model-in-the-Loop
MIPS Million Instructions Per Second
MOST Media Oriented Systems Transport
OCL Object Constraint Language
ODE Ordinary Differential Equation
OMG Object Management Group
OSEK Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
PIM Platform-Independent Model
PLC Programmable Logic Controller
PM Platform Model
POSIX Portable Operating System Interface
PSM Platform-Specific Model
QoS Quality of Service
QVTo Query View Transformation Operational
RAM Random-Access Memory
RISC Reduced Instruction Set Computer
ROS Robot Operating System
RTA Response-Time Analysis
RTCP Real-Time Coordination Protocol
RTOS Real-Time Operating System
RTSC Real-Time Statechart
SAT Boolean Satisfiability Problem
SFC Sequential Function Chart
SIL Safety Integrity Level
SIMD Single Instruction Multiple Data
SMT Satisfiability Modulo Theory
TCP Transmission Control Protocol
UDP User Datagram Protocol
UML Unified Modeling Language
VDX Vehicle Distributed Executive
VLIW Very Long Instruction Word
WCET Worst-Case Execution Time
XMI XML Metadata Interchange
XML Extensible Markup Language

Appendices

A Supplementary Material for the Hardware Platform Description Language

A.1 Hardware Platform Description Meta-Model

We separate the meta-model of the MechatronicUML hardware model into five packages. This section describes the class diagrams for these packages. The packages depend on the MechatronicUML core package [*DPP+16; Hei15], which provides the general base classes NamedElement, ExtendableElement, CommentableElement. These classes enable to provide names, extensions, and comments. Furthermore, the base class Expression of all specific expressions allows to embed these specific expressions within our MechatronicUML action language [*DPP+16]. We omit these dependencies within the class diagrams to provide a better readability. The description is based on the meta-model documentation itself. A more detailed description of a predecessor is given in [*DP14].

A.1.1 Hardware Value Types Meta-Model

Figure A.1 shows the class diagram of the hardware value type package to specify attributes of hardware resources. It defines the classes DataRate, DataSize, and Frequency that have a value property of type EDouble and a unit property referring to the corresponding enumerations DataRateUnit, DataSizeUnit, or FrequencyUnit. We use binary multiples of bits and bytes also known as kibibits and kibibytes. Additionally, the class diagram specifies the class TimeInterval referring to a TimeUnit and a lower and upper bound. We refer to the Java TimeUnit enumeration[1] for defining the time unit constants. Summarizing, the contained classes are represent the following concepts: DataRate represents the data rate of a resource,e.g., the bandwidth of a bus. DataSize represents the data size of an element or the size of a memory resource. TimeInterval represents an interval to specify time bounds.

A.1.2 Hardware Resource Meta-Model

Figure A.2 shows the class diagram of the hardware resource package to define the resource type view. It defines the classes ResourceRepository, CommunicationProtocolRepository, CommunicationProtocol, BusProtocol, LinkProtocol, Resource, CommunicationResource, Device, AtomicResource, StructuredResource, ComputingResource, ProgrammableDevice, Processor, MemoryResource, Cache. Furthermore, the class diagram shows the enumerations MemoryKind, ReplacementPolicy, ProcessorArchitecture, LinkProtocolKind, BusProtocolKind, DeviceKind, MemoryAccessKind, WritePolicy, and CommunicationKind.

ResourceRepository represents a repository containing several resource types. CommunicationProtocolRepository represents a repository containing several CommunicationProtocols, which are used in the hardware models. CommunicationProtocol represents as an abstract class the communication protocol used by the HWPorts and CommunicationMedia. It is used to match HWPorts and CommunicationMedia. BusProtocol represents a bus protocol used by bus media

[1] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html

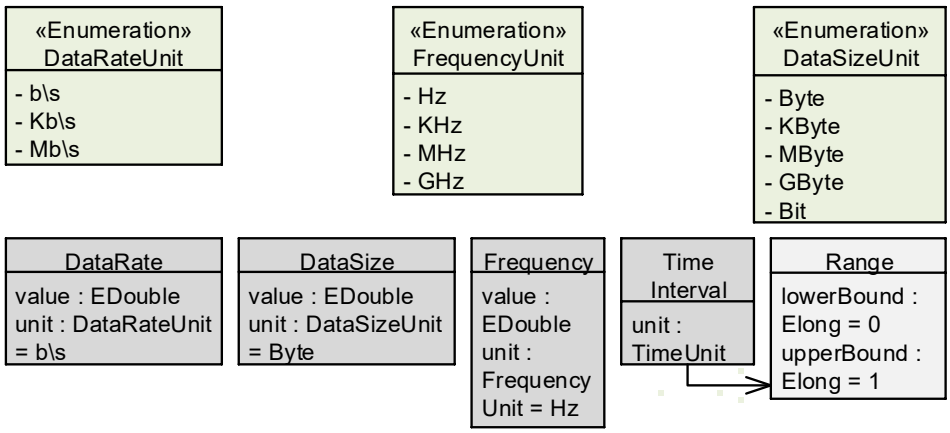


Figure A.1: Class Diagram of the Hardware Value Types Meta-Model

and bus ports. LinkProtocol represents a link protocol used by link media and link ports. Resource represents as an abstract class the super class of all resource types. Communication-Resource represents all resources which are able to transmit data. Device represents a device. Devices are resources (sensors, actuators) that interact with the environment. AtomicResource represents as an abstract class the atomic resource types ComputingResource and Memory-Resource. AtomicResources can not be further sub-divided. StructuredResource represents all StructuredResource (i.e. ECUs, Server,etc.). A StructuredResource consists of several Atomic-Resources. ComputingResource represents as an abstract class all resources which are able to execute code. ProgrammableDevice represents. Processor is a ComputingResource. It represents different kind of a processor specified by its family and architecture. MemoryResource represents an AtomicResource that is capable of storing data. Cache represents a cache used by a Processor.

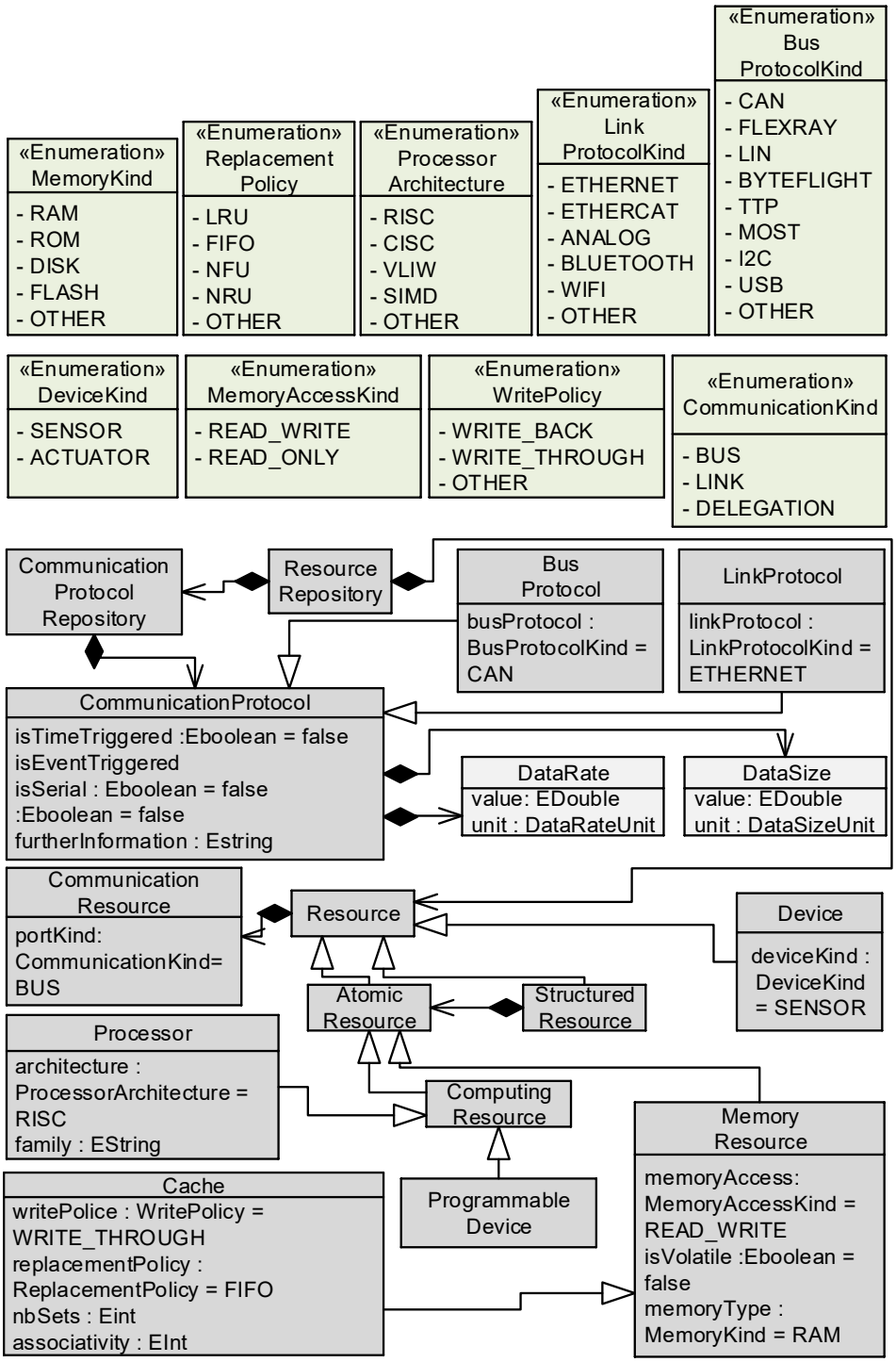


Figure A.2: Class Diagram of the Hardware Resource Meta-Model

A.1.3 Hardware Resource Instance Meta-Model

Figure A.3 shows the class diagram of the hardware resource instance package to define the resource instance view. It defines the classes ResourceInstance, ComputingResourceInstance, MemoryResourceInstance, DeviceInstance, StorageMemoryInstance, ProcessingMemoryInstance, CacheInstance, ProgrammableLogicDeviceInstance, ProcessorInstance, AtomicResourceInstance, StructuredResourceInstance, SensorInstance, ActuatorInstance, ResourceInstanceRepository, and HWPort.

HWPort represents a hardware port of a resource instance. ResourceInstanceRepository represents a ResourceInstanceRepository. A ResourceInstanceRepository contains all ResourceInstances to build a HWPlatform. The ResourceInstances are derived from a ResourceTypeRepository. ActuatorInstance represents an actuator device at the resource instance level. SensorInstance represents a sensor device at the resource instance level. StructuredResourceInstance represents a structured resource at instance level. A StructuredResourceInstance is derived from its resource type. The embedded AtomicResourceInstances of this StructuredResourceInstance are derived from its type. AtomicResourceInstance represents as an abstract class all atomic resource instances. ProcessorInstance represents a processing unit at the hardware resource level. ProgrammableLogicDeviceInstance CacheInstance represents a processor cache at resource instance level. ProcessingMemoryInstance represents fast and volatile processing memory, e.g., SRAM. StorageMemoryInstance represents an instance of non-volatile memory, which is capable of storing data. DeviceInstance represents as an abstract class the super class of all sensors and actuators at instance level. MemoryResourceInstance represents as an abstract class the super class of all memory instances. ComputingResourceInstance represents an instance of a ComputingResource. ComputingResources are resources that are able to execute code. ResourceInstance

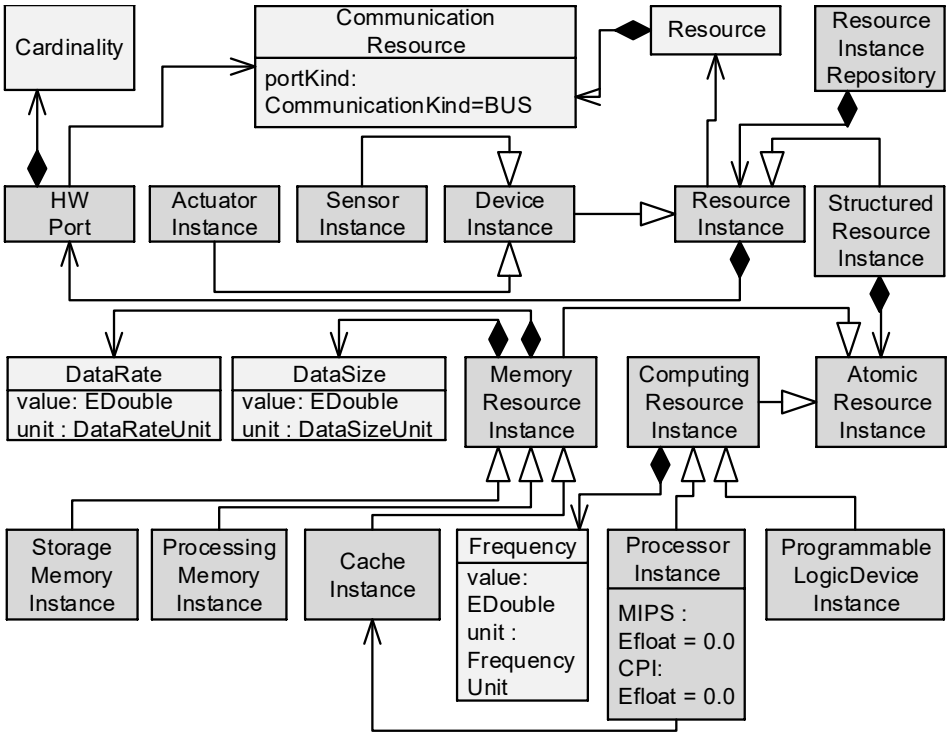


Figure A.3: Class Diagram of the Hardware Resource Instance Meta-Model

A.1.4 Hardware Platform Meta-Model

Figure A.4 shows the class diagram of the hardware platform package to define the platform type view.

It defines the classes HWPlatform, PlatformPart, HWPlatformPart, ResourcePart, DelegationHWPort, HWPortPart, Bus, NetworkBridge, NetworkingHardware, and NetworkConnector.

HWPlatform represents a hardware platform at the type level. A HWPlatform consists of several embedded PlatformParts and several CommunicationResources to connect the embedded PlatformParts. PlatformPart represents a PlatformPart. PlatformParts are used to specify the inner structure of a HWPlatform. A PlatformPart can be a HWPlatformPart or a ResourceInstancePart. HWPlatformPart represents a HWPlatformPart. HWPlatformParts are used to specify the structure of a HWPlatform. A HWPlatformPart is embedded in a HWPlatform and it is typed over a HWPlatform. ResourcePart represents a ResourceInstancePart. ResourceInstanceParts are used to specify the structure of a HWPlatform. A ResourceInstancePart is embedded in a HWPlatform and it is typed over a ResourceInstance. DelegationHWPort a DelegationPort of a HWPlatform on instance level. In order to be consistent with the MechatronicUML meta-model, this class inherits from pim::ConnectorEndpointInstance. HWPortPart represents a hardware port of a PlatformPart and is derived from a HWPort. The purpose of a HWPortPart is to enable the reuse of defined HWPorts in different HWPlatforms and in different configurations. For Instance, HWPortParts are derived from one HWPort but can be attached to different NetworkConnectors. Bus represents a bus. A bus is used to connect several BusPortInstances. NetworkBridge represents a bridge or a gateway. NetworkBridges are used to connect two or more Networks. NetworkingHardware represents as an abstract class the super class of all hardware devices, which are used to connect ECUs, e.g., NetworkBridge or Bus. NetworkConnector represents a network connection between hardware resources. This class inherits from pim::Connector to be consistent with the MechatronicUML meta-model.

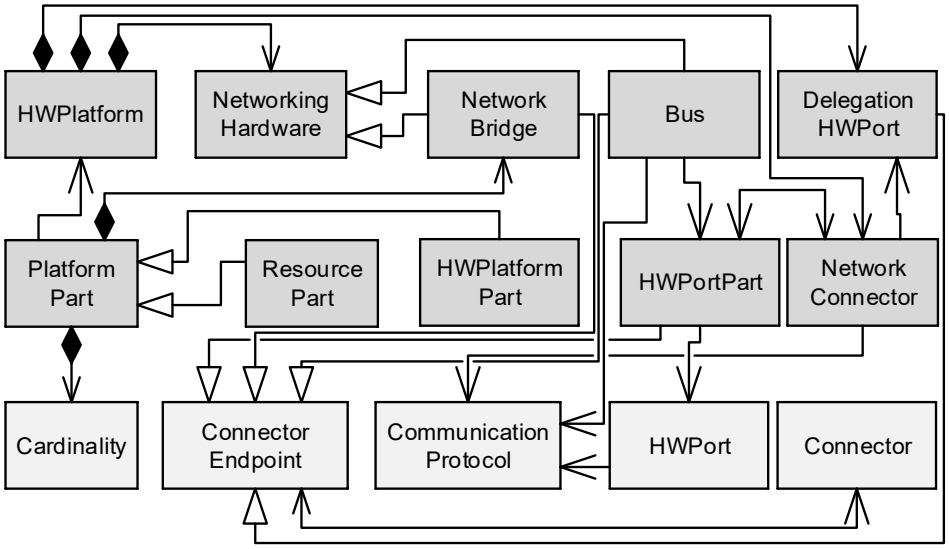


Figure A.4: Class Diagram of the Hardware Platform Meta-Model

A.1.5 Hardware Platform Instance Meta-Model

Figure A.5 shows the class diagram of the hardware platform instance package to define the platform instance view. It defines the classes HWPlatformInstance, HWPlatformInstanceConfiguration,

HWPortInstance, BusInstance, NetworkBridgeInstance, NetworkingHardwareInstance, DelegationHWPortInstance, and NetworkConnectorInstance.

HWPlatformInstance represents a hardware platform at the platform instance level. HWPlatformInstanceConfiguration represents a concrete configuration of a hardware platform instance. A HWPlatformInstanceConfiguration consists of several HWPlatformInstances. The HWPlatformInstances are connected via NetworkConnectorInstances. HWPortInstance represents a hardware port of a hardware platform instance. BusInstance represents a bus at the platform instance level. A BusInstance is used to connect several BusPortInstances. NetworkBridgeInstance represents a bridge or a gateway at the platform instance level. NetworkBridgeInstances are used to connect two or more CommunicationMedia. NetworkingHardwareInstance represents an abstract class hardware devices, which are used to connect ECUs, e.g., bridge, router, bus, on instance level. DelegationHWPortInstance represents a DelegationPort of a HWPlatform at the hardware platform instance level. In order to be consistent with the MechatronicUML meta-model, this class inherits from the class pim::ConnectorEndpointInstance. NetworkConnectorInstance represents a network connection between hardware resources at the hardware platform instance level. This class inherits from the class pim::ConnectorInstance to be consistent with the MechatronicUML meta-model.

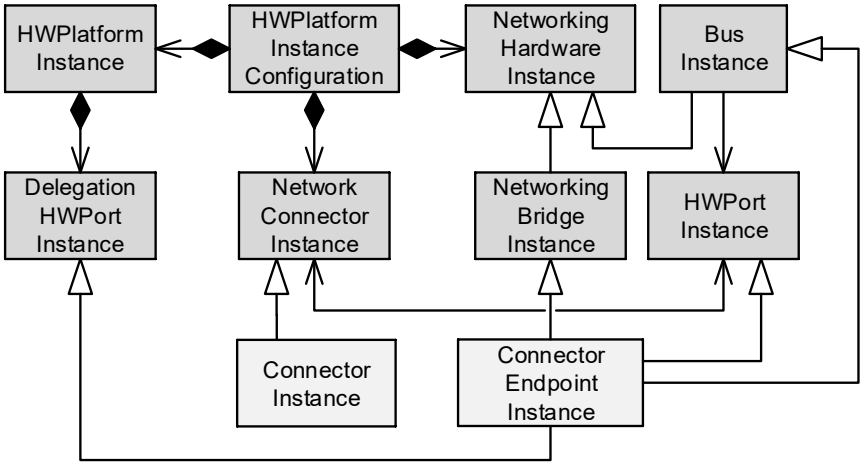


Figure A.5: Class Diagram of the Hardware Platform Instance Meta-Model

B Supplementary Material for the Allocation Specification Language

B.1 Preamble of Allocation Constraint Specification

Our Allocation Specification Language (ASL) can be used for any allocation problem. In this part, we focus on the allocation specification for MechatronicUML component instances to MechatronicUML Electronic Control Units (ECUs). Therefore, we have to specify some properties within the preamble of the allocation constraint specification that we describe in the following paragraphs. Listing E.5 in Appendix E.1 shows a complete preamble in the context of a constraint specification.

Meta-Model Import, Service Provider, OCL Library Include, and Relation

Our model-driven allocation engineering approach works on every EMF-based meta-model. Before we are able to specify allocation constraint for a concrete model we have to

import the corresponding meta-model. In our ASL we use the keyword **import** to refer to the meta-model. For example, for MechatronicUML we import the meta-models 'http://www.muml.org/pm/hardware/hwresourceinstance/1.0.0', 'http://www.muml.org/pim/1.0.0', and 'http://www.muml.org/psm/allocation/language/oclcontext/1.0.0'.

The ASL requires unique names for the transformation to an ILP and a service for storing the results. Therefore, we provide the technical Java service classes that are referenced by the ASL keyword **nameProvider** 'org.muml.psm.allocation.language.xtext.provider.MUMLNameProvider' and **storageProvider** 'org.muml.psm.allocation.language.xtext.provider.MUMLStorageProvider' for MechatronicUML. Appendix B.3 describes them in more detail.

Furthermore, we use OCL libraries to ease the constraint specification. Section 4.5.2 describes this in more detail. We use the keyword **include** to refer to the corresponding OCL library. For example, for MechatronicUML we include the OCL library 'platform:/plugin/org.muml.psm.allocation.language.xtext/operations/OCLContext.ocl'.

An allocation of a component instance to an resource instance is a relation. Within our ASL we have to define which model elements of a concrete type should be allocated to which model elements of a concrete type. Therefore, engineers have to define this concrete relation. Listing B.1 shows the relation specification. Firstly, the relation defines the types of the elements that should be allocated. In our case model elements of type component instances should be allocated to model elements of type ResourceInstance. Secondly, the relation uses an OCL query to specify the concrete set of tuples that describe each possible allocation of a component instance to a resource instance. The query gets all component instances by using the OCL library operation getAllSWInstances and builds the Cartesian product with all structured hardware resources by using the OCL library operation getAllStructuredHWInstances and by using the default OCL operation product. The default OCL operation product has the following return type Set(Tuple(first : T, second : T2)). As a result, the OCL query in Listing B.1 evaluates in the context of our software component instance configuration in Figure 4.3 and hardware platform instance configuration in Figure 4.10 to a tuple like Set{ Tuple{first = overtakeeHMI, second = hmi_board}, Tuple{first = overtakeeDisplay, second = hmi_board}, Tuple{first = overtakeeCommunicator, resourceInstance = hmi_board},...}. In the following of this chapter we assume that the shown constraints always refer to this input set.

```
1 relation {
    descriptors (first:instance::ComponentInstance, second:hwresourceinstance::ResourceInstance);
        ocl self.getAllSWInstances()->product(self.getAllStructuredHWInstances());
    }
```

Listing B.1: The Relation Definition for the MechatronicUML Allocation Planning Problem

B.2 Allocation Specification Language Meta-Model

Figure B.6 shows the class diagram of the allocation package to define the allocation constraint view. It defines the classes SpecificationCS, JavaImplementationProviderCS, NameProviderCS, StorageProviderCS, RelationCS, OCLContextCS, EvaluatableElementCS, ConstraintCS, CoherenceConstraintCS, LocationConstraintCS, ResourceConstraintCS, TupleDescriptorCS, BoundedWeightTupleDescriptorCS, TypedPairCS, and TypedNamedPartCS. Furthermore, the class diagram shows the enumeration CoherenceConstraintType. SpecificationCS represents the complete allocation constraint specification. JavaImplementationProviderCS represents the class path to Java classes that the allocation specification references and which provides the Java implementation of a

name provider and a storage provider. A NameProvider is used to compute a unique name for a given model element, which we require to encode an allocation specification as an Integer Linear Programming (ILP). Therefore, a NameProvider represents the interface for a concrete name provider that requires the operation getName that returns a EString for a given Object. A StorageProvicerCS is used to store the results of the allocation planning within a model. StorageProviderCS represents the interface for a concrete storage provider that requires the operations initialize, store, and noRelationFound. The operation initialize is used to change a given object before the allocation planning results are stored. The operation store is used to store the relation of the first parameter to the second parameter. The operation noAllocationFound is used to provide actions if no feasible allocation is found during the planning. RelationCS represents the elements that are potential candidates for allocations. It is an EvaluableElementCS. EvaluableElementCS represents as an abstract class the super type of all classes for which an evaluable OCL expression can be specified. It refers to a TupleDescriptorCS that defines the type of the pairs that should be allocated. TupleDescriptorCS represents the concrete type description of the allocation tuples. Therefore, it contains TypedPairCS. TypedPairCS represents concrete 2-tuples that contain the first TypedNamedPartCS and the second TypedNamedPartCS. TypedNamedPartCS represents typed model elements. OCLContextCS refers to the evaluation context of all OCL queries within an allocation specification. ConstraintCS represents as an abstract class the super type of all constraints. It is an EvaluatableElementCS CoherenceConstraintCS represents the constraints where two sources either should be collocated or allocated to separate locations. Therefore, it has the property type that is typed over the enumeration CoherenceConstraintType with the two mentioned options. CoherenceConstraintCS is a ConstraintCS. LocationConstraintCS represents all constraints where sources should be allocated to targets with specific properties. LocationConstraintCS is a ConstraintCS. CoherenceConstraintCS and LocationConstraintCS refer to the class TupleDescriptorCS to specify how many 2-tuples have to be considered by the constraints. ResourceConstraintCS represents the constraints where only a specific amount of sources can be allocated to a target due to resource limitations. It contains a BoundedWeightTupleDescriptor that represents the resource bound of the target, the weight of each tuple, and the tuple itself. BoundedWeightTupleDescriptor is a WeightTupleDescriptorCS, which is a TupleDescriptorCS.

B.3 Name Provider and Storage Provider

Each element must have a unique name that does not contain special characters like "-" because an ILP would interpret this characters as mathematical instructions. A developer has to provide a Java class that provides the getName(Object) operation, which returns this name for all model elements that are considered during the allocation planning. Within the allocation specification this class can be referenced after the keyword **nameProvider**. For example, we provide the name provider 'org.muml.psm.allocation.language.xtext.provider.MUMLNameProvider' for MechatronicUML. Additionally, after the planning the planning result should be stored within a model. Therefore, the software engineer or allocation engineer must provide a Java class that provides the operations initialize(Object), store(Object,Object):Object and noAllocationFound(Object). Initialize is used to prepare the model before storing all allocations, store is used to store each concrete allocation, and noAllocationFound is used if no feasible allocation can be calculated. Within the allocation specification this class can be referenced after the keyword **storageProvider**. For example, we provide the storage provider 'org.muml.psm.allocation.language.xtext.provider.MUMLStorageProvider' for MechatronicUML.

B.4 OCL-based ASL Library

```
1 import 'http://www.muml.org/pm/hardware/hwresource/1.0.0'
```

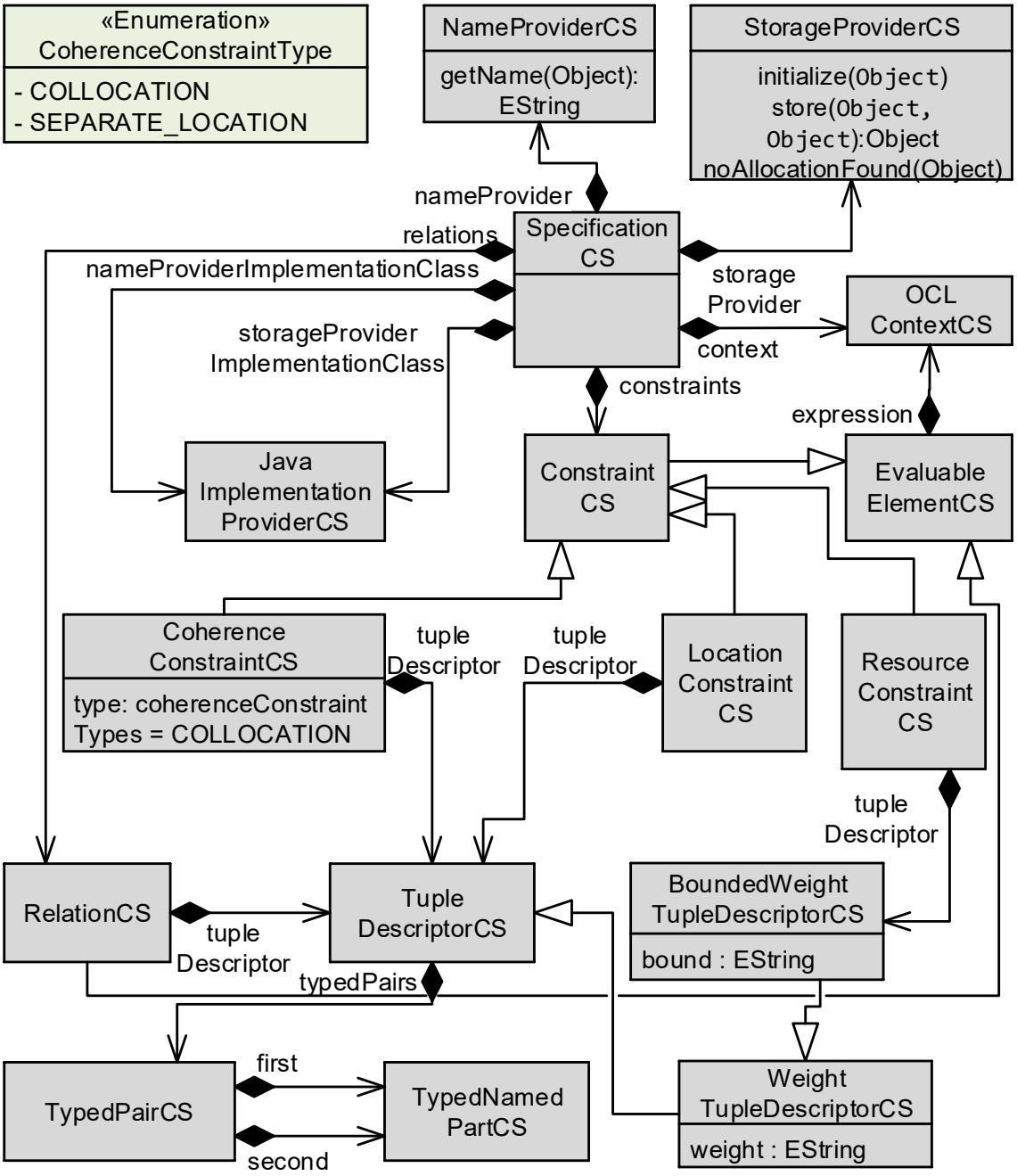


Figure B.6: Class Diagram of the Allocation Specification Language Meta-Model

```
import 'http://www.muml.org/pm/hardware/hwplatforminstance/1.0.0'
import 'http://www.muml.org/pm/hardware/hwresourceinstance/1.0.0'
import 'http://www.muml.org/pim/instance/1.0.0'
import 'http://www.muml.org/pim/connector/1.0.0'
import 'http://www.muml.org/pim/valuetype/1.0.0'
import 'http://www.muml.org/psm/properties/1.0.0'
import 'http://www.muml.org/core/expressions/common/1.0.0'
import 'http://www.muml.org/psm/allocation/context/muml/oclcontext/1.0.0'

context oclcontext::OCLContext

def: getECUs(ecuNames : Set(String)) : Sequence(hwresourceinstance::
    ResourceInstance) =
    self.hardwarePlatformInstanceConfiguration.resources->
    select(ecuNames->includes(name))->asSequence()

def: getECU(ecuName : String) : hwresourceinstance::ResourceInstance =
    self.getAllHWInstances()->any(name = ecuName)

def: getAllSWInstances() : Set(instance::ComponentInstance) =
    self.componentInstanceConfiguration.getAllEmbeddedInstances()

def: getSWInstance(instanceName : String) : instance::ComponentInstance
    =
    self.getAllSWInstances()->any(name = instanceName)

def: getAllHWInstances() : Set(hwresourceinstance::ResourceInstance) =
    self.getAllHWPlatformInstances()->collect(embeddedHPIC.resources)->
        asSet()

def: getAllHWPlatformInstances() : Set(hwplatforminstance::
    HWPlatformInstance) =
    self.hardwarePlatformInstanceConfiguration.
        getAllEmbeddedPlatformInstances()

def: getHWPlatformInstance(platformInstanceName : String) :
    hwplatforminstance::HWPlatformInstance =
    self.getAllHWPlatformInstances()->any(name = platformInstanceName)

def: allocateToECU(instanceName : String, ecuName : String)
    : Set(Tuple(first : instance::ComponentInstance, second :
        hwresourceinstance::ResourceInstance))
    = self.allocateToECU(self.getSWInstance(instanceName), ecuName)

def: allocateToECU(componentInstance : instance::ComponentInstance,
    ecuName : String)
    : Set(Tuple(
    first : instance::ComponentInstance,
    second : hwresourceinstance::ResourceInstance)) =
    componentInstance->asSet()->
    product(self.getECU(ecuName).resolveToStructuredResourceInstances())
        ->asSet()

def: allocateComponentToPlatform(swInstanceName : String,
    platformInstanceName : String)
```

```
    : Set(Tuple(component : instance::ComponentInstance, allowedResource
        : hwresourceinstance::ResourceInstance))
    = self.getSWInstance(swInstanceName).getAllEmbeddedInstances()->
        product(
    self.getHWPlatformInstance(platformInstanceName).embeddedHPIC.
        resources
    ->select(oclIsKindOf(hwresourceinstance::StructuredResourceInstance))
        )

  def: allocateToDifferentECUs(instance1 : String, instance2 : String)
    : Set(Tuple(firstComponent : instance::ComponentInstance,
        secondComponent : instance::ComponentInstance))
    = Set{Tuple{firstComponent = self.getSWInstance(instance1),
        secondComponent = self.getSWInstance(instance2)}}

  def: allocateToSameECU(instance1 : String, instance2 : String)
    : Set(Tuple(firstComponent : instance::ComponentInstance,
        secondComponent : instance::ComponentInstance))
    = self.allocateToDifferentECUs(instance1, instance2)

  def: allocateCommunicatingComponentsToConnectedECUs() : Set(Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance)) =
    let allHWInstances : Set(hwresourceinstance::ResourceInstance) =
    self.getAllHWInstances()->select(oclIsKindOf(hwresourceinstance::
        StructuredResourceInstance))
    in
    self.componentInstanceConfiguration.getCommunicatingComponents()->
        collect(s |
    self.allocateComponentsToConnectedECUs(s->at(1), s->at(2),
    allHWInstances))->asSet()

  def: allocateComponentsToConnectedECUs(c1 : instance::ComponentInstance,
     c2 : instance::ComponentInstance,
    allowedECUs : Set(hwresourceinstance::ResourceInstance)) : Set(Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance)) =
    allowedECUs->collect(e1 |
    e1.getConnectedECUs()->union(Set{e1})
    ->select(e | allowedECUs->includes(e))
    ->collect(e2 |
    Set{
    Tuple{c1 = c1, e1 = e1, c2 = c2, e2 = e2},
    Tuple{c1 = c1, e1 = e2, c2 = c2, e2 = e1}
    }))->asSet()

    def: colocateSCIAndEmbeddedParts(componentName : String) :
    Set(Tuple(first : instance::ComponentInstance, second : instance::
        ComponentInstance)) =
    let ci : instance::ComponentInstance = self.getSWInstance(
        componentName)
    in
```

```
    ci->asSet()->product(ci.getAllEmbeddedInstances()->reject(c | c = ci)
        )

  def: getSWInstancesWithRequiredMemory() : Set(instance::
      ComponentInstance) =
      self.getAllSWInstances()->select(ci |
      ci.getMemoryExtensions()->notEmpty())

  def: getHWInstancesWithMaxMemory() : Set(hwresourceinstance::
      StructuredResourceInstance) =
      self.getAllHWInstances()->select(
      oclIsKindOf(hwresourceinstance::StructuredResourceInstance)
      )->collect(
      oclAsType(hwresourceinstance::StructuredResourceInstance)
      )->select(
      getMemoryResourceInstances()->notEmpty())->asSet()

  def: maxMemoryConsumption() : Set(
      Tuple(
      requiredMemory : Set(
      Tuple(
      componentInstance : instance::ComponentInstance,
      resourceInstance : hwresourceinstance::ResourceInstance,
      requiredMemory : Real
      )),
      maxMemory : Real
      )) =
      let swInstances : Set(instance::ComponentInstance) = self.
          getSWInstancesWithRequiredMemory()
      in
      self.getHWInstancesWithMaxMemory()->collect(ecu |
      Tuple{
      maxMemory = ecu.getMemoryResourceInstances()->any(true).memorySize._'
          value',
      requiredMemory = swInstances->product(ecu->asSet())->collect(t |
      Tuple {
      componentInstance = t.first,
      resourceInstance = t.second,
      requiredMemory = t.first.getMemoryExtensions()->any(true).
          requiredMemory._'value'
      })->asSet()})->asSet()

  def: getSWInstancesWithWCET(instances : Set(instance::ComponentInstance)
      ,
      ecu : hwresourceinstance::ResourceInstance) : Set(instance::
          ComponentInstance) =
          instances->select(not getWCETExtensionForECU(ecu).oclIsUndefined()
              )

  def: getSWInstancesForSchedulingOnECU(instances : Set(instance::
      ComponentInstance),
      ecu :     hwresourceinstance::ResourceInstance) : Set(instance::
          ComponentInstance) =
      self.getSWInstancesWithWCET(instances, ecu)->
      select(not getSchedulingExtension().oclIsUndefined())
```

```
context connector::ConnectorInstance
def: getReachableConnectors() : OrderedSet(OrderedSet(connector::
    ConnectorInstance)) =
    OrderedSet{OrderedSet{self}}->
    closure(s : OrderedSet(connector::ConnectorInstance) | s->last()->
        collect(c |
    c.connectorEndpointInstances->collect(connectorInstances)
    )->select(c | if s->size() >= 2 then
    s->at(s->size() - 1).connectorEndpointInstances
    else
    OrderedSet{}
    endif
    ->intersection(c.connectorEndpointInstances)->isEmpty()
    )->collectNested(c | s->append(c)))

context hwplatforminstance::HWPlatformInstanceConfiguration
def: getAllEmbeddedPlatformInstances() : Set(hwplatforminstance::
    HWPlatformInstance) =
    self.hwplatformInstances->closure(h |
    h->asSet()->union(h.embeddedHPIC.hwplatformInstances)
    )->asSet()


context hwresourceinstance::ResourceInstance
def: resolveToStructuredResourceInstances() : Set(hwresourceinstance::
    ResourceInstance) =
    if self.oclIsKindOf(hwresourceinstance::DeviceInstance) then
    self.getConnectedECUs()->select(oclIsKindOf(hwresourceinstance::
        StructuredResourceInstance))
    else
    self->asSet()
    endif

def: getConnectedECUs() : Set(hwresourceinstance::ResourceInstance) =
    self.getConnectors()->collect(c |
    c.connectorEndpointInstances->select(oclIsKindOf(hwresourceinstance::
        HWPort))
    ->collect(oclAsType(hwresourceinstance::HWPort).
        parentResourceInstance)
    )->asSet()->reject(r | r = self)

def: getConnectors(other : hwresourceinstance::ResourceInstance) :
    OrderedSet(OrderedSet(connector::ConnectorInstance)) =
    self.getConnectors()->select(s |
    s->last().connectorEndpointInstances
    ->select(oclIsKindOf(hwresourceinstance::HWPort))
    ->collect(oclAsType(hwresourceinstance::HWPort).
        parentResourceInstance)
    ->includes(other))

def: getConnectors() : OrderedSet(OrderedSet(connector::
    ConnectorInstance)) =
    self.hwports->select(oclIsKindOf(connector::ConnectorEndpointInstance
        ))
    ->collect(oclAsType(connector::ConnectorEndpointInstance))
    ->collect(connectorInstances)
```

```
    ->collectNested(c | c.getReachableConnectors())
    ->iterate(it : OrderedSet(OrderedSet(connector::ConnectorInstance));
    acc : OrderedSet(OrderedSet(connector::ConnectorInstance)) =
        OrderedSet{} |
    acc->union(it))

def: getReachableECUs() : Set(hwresourceinstance::ResourceInstance) =
    self->asSet()->closure(
    getConnectedECUs()
    )->reject(r | r = self)

context hwresourceinstance::StructuredResourceInstance
def: getMemoryResourceInstances() : Set(hwresourceinstance::
    MemoryResourceInstance) =
    self.embeddedAtomicResourceInstances->select(
    oclIsKindOf(hwresourceinstance::MemoryResourceInstance)
    )->collect(
    oclAsType(hwresourceinstance::MemoryResourceInstance)
    )->select(
    memoryType = hwresource::MemoryKind::RAM
    )->asSet()

context instance::ComponentInstanceConfiguration
def: getAllEmbeddedInstances() : Set(instance::ComponentInstance) =
    self.componentInstances->closure(c |
    if c.oclIsKindOf(instance::StructuredComponentInstance) then
    c->asSet()->union(c.oclAsType(instance::StructuredComponentInstance).
        embeddedCIC.componentInstances)
    else
    c->asSet()
    endif
    )->asSet()

def: getCICs() : Set(instance::ComponentInstanceConfiguration) =
    self->asSet()->closure(
    componentInstances->select(oclIsKindOf(instance::
        StructuredComponentInstance))
    ->collect(oclAsType(instance::StructuredComponentInstance))
    ->collect(embeddedCIC))

def: getConnectors() : Set(instance::PortConnectorInstance) =
    self.getCICs()->collect(portConnectorInstances)->select(portInstances
        ->size() = 2)->asSet()

def: getCommunicatingComponents() : Set(OrderedSet(instance::
    ComponentInstance)) =
    self.getCICs()->collect(getConnectors())
    ->collectNested(
    OrderedSet{
    portInstances->at(1).componentInstance,
    portInstances->at(2).componentInstance
    })->asSet()

context instance::ComponentInstance
def: getAllEmbeddedInstances() : Set(instance::ComponentInstance) =
    if self.oclIsKindOf(instance::StructuredComponentInstance) then
```

```
    self->asSet()->union(self.oclAsType(instance::
        StructuredComponentInstance).
    embeddedCIC.getAllEmbeddedInstances())
    else
    self->asSet()
    endif

def: getMemoryExtensions() : Set(properties::RequiredMemory) =
    self.extension->select(oclIsKindOf(properties::RequiredMemory))
    ->collect(oclAsType(properties::RequiredMemory))->asSet()

def: getSchedulingExtensions() : Set(properties::Scheduling) =
    self.extension->select(oclIsKindOf(properties::Scheduling))
    ->collect(oclAsType(properties::Scheduling))->asSet()

def: getSchedulingExtension() : properties::Scheduling =
    self.getSchedulingExtensions()->any(true)

def: getWCETExtensions() : Set(properties::WCET) =
    self.extension->select(oclIsKindOf(properties::WCET))
    ->collect(oclAsType(properties::WCET))->asSet()

def: getWCETExtensionForECU(ecu : hwresourceinstance::ResourceInstance)
    : properties::WCET =
    let wcets : Set(properties::WCET) = self.getWCETExtensions()
    in
    let wcet : properties::WCET = wcets->any(resourceInstance = ecu)
    in
    if wcet.oclIsUndefined() then
    wcets->any(resourceInstance.oclIsUndefined())
    else
    wcet
    endif

context valuetype::TimeValue
def: getLiteralVal() : Real =
    if self.'value'.oclIsKindOf(common::LiteralExpression) then
    self.'value'.oclAsType(common::LiteralExpression).'value'.toReal()
    else
    -1
    endif


-- CAN RTA
context oclcontext::OCLContext

def: busUtilization(busName : String) : Set(
    Tuple(utilization : Set(
    Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance,
    utilization : Real)),
    messageDeadline : Real)) =
    let canBus = self.getCANBus(busName)
```

```
    in
    let busECUPairs : Set(Tuple(first : hwresourceinstance::
        ResourceInstance,
    second : hwresourceinstance::ResourceInstance)) =
    canBus.getConnectedECUs()->product(canBus.getConnectedECUs())->reject
        (t |
    t.first = t.second
    or not t.first.oclIsKindOf(hwresourceinstance::
        StructuredResourceInstance)
    or not t.second.oclIsKindOf(hwresourceinstance::
        StructuredResourceInstance))
    in
    Set{Tuple{
    utilization = self.getSendingPortInstances()->collect(
    getMessageExtension()
    )->collect(msg |
    let senderPortInstance : instance::PortInstance =
    msg.extendableBase.oclAsType(instance::PortInstance)
    in
    let
    senderComponent : instance::ComponentInstance = senderPortInstance.
        componentInstance
    in
    letreceiverComponent : instance::ComponentInstance =
    senderPortInstance.getReceiverPortInstance().componentInstance
    in
    busECUPairs->collect(t |
    Tuple{
    c1 = senderComponent,
    e1 = t.first,
    c2 = receiverComponent,
    e2 = t.second,
    utilization = canBus.getUtilization(Set{msg})
    }))->asSet(),messageDeadline = 1}}

def: canRTA(busName : String) : Set(
    Tuple(messageResponseTime : Set(
    Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance,
    messageResponseTime : Real)),messageDeadline : Real)) =
    let canBus = self.getCANBus(busName)
    in
    let possibleMessages : Set(Set(properties::CANMessageFrame)) =
    canBus.getPossibleMessages(
    self.getSendingPortInstances()->collect(getMessageExtension())->asSet
        ())
    in
    possibleMessages->collect(messages |
    messages->collect(msg |
    let blockingTime : Real = canBus.getBlockingTime(msg.
        getLowerPriorityMessages(messages, true))
    in
```

```
    let busyPeriod : Real = msg.calculateBusyPeriod(msg.
        getHigherPriorityMessages(messages, false),
    canBus, blockingTime)
    in
    msg.calculateWorstCaseResponseTimeTuples(
    msg.getHigherPriorityMessages(messages, true),
    canBus, blockingTime, busyPeriod,
    self.getAllHWInstances()->select(
    oclIsKindOf(hwresourceinstance::StructuredResourceInstance)))))->
        asSet()


def: getCANBusses() : Set(hwplatforminstance::BusInstance) =
    self.getAllHWPlatformInstances()->collect(embeddedHPIC)->collect(
    networkingHardwareInstances
    )->select(oclIsKindOf(hwplatforminstance::BusInstance)
    )->collect(oclAsType(hwplatforminstance::BusInstance))->asSet()

def: getCANBus(busName : String) : hwplatforminstance::BusInstance =
    self.getCANBusses()->any(name = busName)

def: getSendingPortInstances() : Set(instance::PortInstance) =
    self.getAllSWInstances()->collect(getSendingPortInstances())->asSet()

def: getPowerset(portInstances : Set(instance::PortInstance)) : Set(Set(
    instance::PortInstance)) =
    Set{portInstances}->closure(s |
    s->collectNested(c |
    s->reject(elm | elm = c)))->asSet()

context hwplatforminstance::BusInstance
def: getConnectedECUs() : Set(hwresourceinstance::ResourceInstance) =
    self.getConnectors()->select(path |
    -- path should contain other busses or bridges
    path->forAll(c |
    c.connectorEndpointInstances->forAll(ce |
    ce.oclIsKindOf(hwplatforminstance::NetworkingHardwareInstance)
        implies ce = self
    )))->collect(path |
    path->collect(connectorEndpointInstances)->select(
    oclIsKindOf(hwplatforminstance::HWPortInstance)
    )->collect(oclAsType(hwplatforminstance::HWPortInstance).
        parentResourceInstance)
    )->asSet()


def: getConnectors() : Set(OrderedSet(connector::ConnectorInstance)) =
    self.connectorInstances
    ->collectNested(c | c.getReachableConnectors())
    ->iterate(it : OrderedSet(OrderedSet(connector::ConnectorInstance));
    acc : OrderedSet(OrderedSet(connector::ConnectorInstance)) =
        OrderedSet{} |
    acc->union(it))

def: getPossibleMessages(messages : Set(properties::CANMessageFrame)) :
    Set(Set(properties::CANMessageFrame)) =
```

```
    Set{messages}->closure(msgs |
385 if self.getUtilization(msgs) < 1 then
    Set{msgs}
    else
    msgs->collectNested(msg |
    msgs->reject(elm | elm = msg))
390 endif)->select(msgs | self.getUtilization(msgs) < 1)

def: getUtilization(messages : Set(properties::CANMessageFrame)) : Real
    =
    messages->collect(msg |
    self.getTransmissionTime(msg) / msg.period.getLiteralVal())->sum()
395
def: getTransmissionTime(message : properties::CANMessageFrame) : Real =
    message.size._'value' / self.bandwidth._'value'

def: getBlockingTime(lmessages : Set(properties::CANMessageFrame)) :
    Real =
400 lmessages->collect(msg | self.getTransmissionTime(msg))
   ->union(Set{0})->_'max'()

def: getBitTransmissionTime() : Real =
    1 / self.bandwidth._'value'
405
context instance::ComponentInstance
    def: getSendingPortInstances() : Set(instance::PortInstance) =
    self.portInstances->select(
    not getMessageExtension().oclIsUndefined())->asSet()
410
context instance::PortInstance
def: getReceiverPortInstance() : instance::PortInstance =
    self.connectorInstances->any(true).connectorEndpointInstances->
    any(p | p <> self).oclAsType(instance::PortInstance)
415
def: getMessageExtensions() : Set(properties::CANMessageFrame) =
    self.extension->select(oclIsKindOf(properties::CANMessageFrame))
   ->collect(oclAsType(properties::CANMessageFrame))->asSet()

420 def: getMessageExtension() : properties::CANMessageFrame =
    self.getMessageExtensions()->any(true)

context valuetype::NaturalNumber
    def: getVal() : Integer =
425 self._'value'

context properties::CANMessageFrame
    def: getLowerPriorityMessages(msgs : Set(properties::CANMessageFrame)
        ,
    strict : Boolean) : Set(properties::CANMessageFrame) =
430 msgs->select(msg |
    if strict then
    msg.priority.getVal() > self.priority.getVal()
    else
    msg.priority.getVal() >= self.priority.getVal()
435 endif)
```

```
    def: getHigherPriorityMessages(msgs : Set(properties::CANMessageFrame
        ),
    strict : Boolean) : Set(properties::CANMessageFrame) =
    msgs - self.getLowerPriorityMessages(msgs, not strict)

def: ceil(val : Real) : Integer =
    let floor : Integer = val.floor()
    in
    if floor = val then
    floor
    else
    floor + 1
    endif

def: calculateBusyPeriod(hemessages : Set(properties::CANMessageFrame),
    canBus : hwplatforminstance::BusInstance, blockingTime : Real) : Real
        =
    OrderedSet{canBus.getTransmissionTime(self)}->closure(t : Real |
    hemessages->collect(msg |
    self.ceil(t / msg.period.getLiteralVal()) *
    canBus.getTransmissionTime(msg)
    )->sum() + blockingTime)->last()

def: calculateWaitTimeForInstanceq(
    hmessages : Set(properties::CANMessageFrame),
    canBus : hwplatforminstance::BusInstance,
    blockingTime : Real, q : Integer) : Real =
    let transmissionQ : Real = q * canBus.getTransmissionTime(self)
    in
    OrderedSet{blockingTime + transmissionQ}->closure(w : Real |
    hmessages->collect(msg |
    self.ceil(
    (w + canBus.getBitTransmissionTime()) / msg.period.getLiteralVal()
    ) * canBus.getTransmissionTime(msg)
    )->sum() + blockingTime + transmissionQ)->last()

    def: calculateWorstCaseResponseTimeForInstanceq(
    hmessages : Set(properties::CANMessageFrame), canBus :
        hwplatforminstance::BusInstance,
    blockingTime : Real, q : Integer
    ) : Real =
    self.calculateWaitTimeForInstanceq(hmessages, canBus, blockingTime, q
        )
    - q * self.period.getLiteralVal() + canBus.getTransmissionTime(self)

def: calculateWorstCaseResponseTime(
    hmessages : Set(properties::CANMessageFrame), canBus :
        hwplatforminstance::BusInstance,
    blockingTime : Real, busyPeriod : Real
    ) : Tuple(worstCaseResponseTime : Real, instance : Integer) =
    let maxInstances : Integer = self.ceil(busyPeriod / self.period.
        getLiteralVal())
    in
    let worstCaseResponseTimes : OrderedSet(Real) = OrderedSet{0 ..
        maxInstances - 1}->collect(q |
    self.calculateWorstCaseResponseTimeForInstanceq(
```

```
    hmessages, canBus, blockingTime, q ))->asOrderedSet()
    in
    let worstCaseResponseTime : Real = worstCaseResponseTimes->_'max'()
    in
    Tuple{
    worstCaseResponseTime = worstCaseResponseTime,
    instance = worstCaseResponseTimes->indexOf(worstCaseResponseTime) -
        1}

  def: calculateInterferenceTimeForInstanceQ(
    hmessages : Set(properties::CANMessageFrame), canBus :
        hwplatforminstance::BusInstance,
    blockingTime : Real, q : Integer, interferingMessage : properties::
        CANMessageFrame
    ) : Real =
    let
    interferenceTime : Real =
    self.calculateWaitTimeForInstanceq(hmessages, canBus, blockingTime, q
        )
    - blockingTime - q * self.period.getLiteralVal() - q * canBus.
        getTransmissionTime(self)
    in
    self.ceil(
    interferenceTime / interferingMessage.period.getLiteralVal()
    ) * canBus.getTransmissionTime(interferingMessage)

  def: calculateWorstCaseResponseTimeLvalTuples(
    hmessages : Set(properties::CANMessageFrame), canBus :
        hwplatforminstance::BusInstance,
    blockingTime : Real, busyPeriod : Real, ecus : Set(hwresourceinstance
        ::ResourceInstance)
    ) : Set(
    Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance,
    messageResponseTime : Real
    )) =
    let t : Tuple(worstCaseResponseTime : Real, instance : Integer) =
    self.calculateWorstCaseResponseTime(hmessages, canBus, blockingTime,
        busyPeriod)
    in
    let senderPortInstance : instance:: PortInstance =
    self.extendableBase.oclAsType(instance::PortInstance)
    in
    let senderComponent : instance::ComponentInstance =
    senderPortInstance.componentInstance
    in
    let receiverComponent : instance::ComponentInstance =
    senderPortInstance.getReceiverPortInstance().componentInstance
    in
    let senderReceiverInterference : Set(Tuple(senderComponent : instance
        ::ComponentInstance,
    receiverComponent : instance::ComponentInstance, interferenceTime :
        Real)) =
```

```
    hmessages->collect(msg |
    let senderPortInstance : instance::PortInstance
    =
    msg.extendableBase.oclAsType(instance::PortInstance)
    in
    Tuple{
    senderComponent = senderPortInstance.componentInstance,
    receiverComponent = senderPortInstance.getReceiverPortInstance().
        componentInstance,
    interferenceTime = self.calculateInterferenceTimeForInstanceQ(
    hmessages, canBus, blockingTime, t.instance, msg)})->asSet()
    in
    let busECUPairs : Set(Tuple(first : hwresourceinstance::
        ResourceInstance,
    second : hwresourceinstance::ResourceInstance))
    =
    canBus.getConnectedECUs()->product(canBus.getConnectedECUs())->reject
        (t |
    t.first = t.second
    or not t.first.oclIsKindOf(hwresourceinstance::
        StructuredResourceInstance)
    or not t.second.oclIsKindOf(hwresourceinstance::
        StructuredResourceInstance))
    in
    busECUPairs->collect(tb |
    Tuple{
    c1 = senderComponent,
    e1 = tb.first,
    c2 = receiverComponent,
    e2 = tb.second,
    messageResponseTime = t.worstCaseResponseTime}
    )->union(
    senderReceiverInterference->collect(t |
    ecus->collect(ecu |
    Tuple{
    c1 = t.senderComponent,
    e1 = ecu,
    c2 = t.receiverComponent,
    e2 = ecu,
    messageResponseTime = -1 * t.interferenceTime
    }))    )->asSet()

  def: calculateWorstCaseResponseTimeTuples(
    hmessages : Set(properties::CANMessageFrame), canBus :
        hwplatforminstance::BusInstance,
    blockingTime : Real, busyPeriod : Real, ecus : Set(hwresourceinstance
        ::ResourceInstance)
    ) : Set(
    Tuple(
    messageResponseTime : Set(
    Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance,
    messageResponseTime : Real)), messageDeadline : Real))=
```

```
    Set{
    Tuple{
    messageResponseTime = self.calculateWorstCaseResponseTimeLvalTuples(
    hmessages, canBus, blockingTime, busyPeriod, ecus),
    messageDeadline = self.deadline.getLiteralVal()}}

-- EDF
context oclcontext::OCLContext
def: minimum(a : Real, b : Real) : Real =
    if a - b < 0 then
    a
    else
    b
    endif

def: EDFOnECU(instances : Set(instance::ComponentInstance),
    ecu : hwresourceinstance::ResourceInstance)
    : Set(Tuple(computationDemand : Set(
    Tuple( componentInstance : instance::ComponentInstance,
    resourceInstance : hwresourceinstance::ResourceInstance,
        computationDemand : Real
    )), upperBound : Real)) =
    Set {
    Tuple {
    computationDemand = self.getSWInstancesForSchedulingOnECU(instances,
        ecu)->
    collect(swInstance : instance::ComponentInstance |
    let scheduling : properties::Scheduling = swInstance.
        getSchedulingExtension()
    in
    let period : Real = scheduling.period.getLiteralVal()
    in
    let deadline : Real = scheduling.deadline.getLiteralVal()
    in
    let wcet : Real = swInstance.getWCETExtensionForECU(ecu).wcet.
        getLiteralVal()
    in
    Tuple{
    componentInstance = swInstance,
    resourceInstance = ecu,
    computationDemand = wcet / self.minimum(period, deadline)
    })->asSet(), upperBound = 1}}

def: EDF()
    : Set(Tuple(computationDemand : Set(Tuple(
    componentInstance : instance::ComponentInstance,
    resourceInstance : hwresourceinstance::ResourceInstance,
    computationDemand : Real    )),    upperBound : Real ))
    =
    let instances : Set(instance::ComponentInstance) = self.
        getAllSWInstances()
    in
    self.getAllHWInstances()->collect(ecu : hwresourceinstance::
        ResourceInstance |
    self.EDFOnECU(instances, ecu)
    )->asSet()
```

```
-- RTA
context oclcontext::OCLContext
    def: RTAForInstanceAndECU(swInstance : instance::ComponentInstance,
    instances : Set(instance::ComponentInstance), ecu :
        hwresourceinstance::ResourceInstance)
    : Set(Tuple(responseTime : Set(  Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance,
    responseTime : Real  )),   upperBound : Real )) =
    let deadline : Real = swInstance.getSchedulingExtension().deadline.
        getLiteralVal()
    in
    Set{Tuple { responseTime = Set{  Tuple{
    c1 = swInstance,
    e1 = ecu,
    c2 = swInstance,
    e2 = ecu,
    responseTime = swInstance.getWCETExtensionForECU(ecu).wcet.
        getLiteralVal()}}
    ->union(
    instances->select(
    getSchedulingExtension().priority._'value' > swInstance.
        getSchedulingExtension().priority._'value'
    )->collect(other : instance::ComponentInstance |
    let otherPeriod : Real = other.getSchedulingExtension().period.
        getLiteralVal()
    in
    let otherWCET : Real = other.getWCETExtensionForECU(ecu).wcet.
        getLiteralVal()
    in
    Tuple{
    c1 = swInstance,
    e1 = ecu,
    c2 = other,
    e2 = ecu,
    responseTime = otherWCET - otherWCET * otherWCET / otherPeriod +
        otherWCET / otherPeriod * deadline
    })->asSet()), upperBound = deadline}}

def: RTAOnECU(instances : Set(instance::ComponentInstance), ecu :
    hwresourceinstance::ResourceInstance)
    : Set(
    Tuple(
    responseTime : Set(
    Tuple(
    c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance,
    responseTime : Real
    )), upperBound : Real)) =
    let schedInstances : Set(instance::ComponentInstance) =
        getSWInstancesForSchedulingOnECU(instances, ecu)
```

```
    in
    schedInstances->collect(swInstance : instance::ComponentInstance |
680 self.RTAForInstanceAndECU(swInstance, schedInstances, ecu))->asSet()

def: RTA()
    : Set(Tuple( responseTime : Set(
    Tuple(
685 c1 : instance::ComponentInstance,
    e1 : hwresourceinstance::ResourceInstance,
    c2 : instance::ComponentInstance,
    e2 : hwresourceinstance::ResourceInstance,
    responseTime : Real
690 )), upperBound : Real)) =
    let instances : Set(instance::ComponentInstance) = self.
        getAllSWInstances()
    in
    self.getAllHWInstances()->collect(ecu : hwresourceinstance::
        ResourceInstance |
    self.RTAOnECU(instances, ecu))->asSet()
```

B.5 Linear Program Meta-Model

Figure B.7 shows the corresponding meta-model for linear programs. A LinearProgram is composed of Variables, ConstraintExpressions, and an ObjectiveFunctionExpression. A Variable has a name and a data type. The meta-model provides the data types BINARY, INTEGER, and REAL. A ConstraintExpression is a BinaryExpression and is composed of a left and an right expression. As its operator a ConstraintExpression has a ComparingOperator. The meta-model provides the comparing operators LESS LESS_OR_EQUAL, EQUAL, GREATER_OR_EQUAL, and GREATER. An expression may be an ArithmeticExpression, a UnaryExpression, a VariableExpression, or a LiteralExpression. An ArithmeticExpression is a BinaryExpression with an ArithmeticOperator. The meta-model provides the arithmetic operators PLUS, MINUS, and TIMES. A UnaryExpression is a expression with one enclosed expression and the UnaryOperator MINUS. A VariableExpression references a variable and a LiteralExpression has a value. Lastly, an ObjectiveFunctionExpression defines a goal. The meta-model provides the objective goals MIN and MAX.

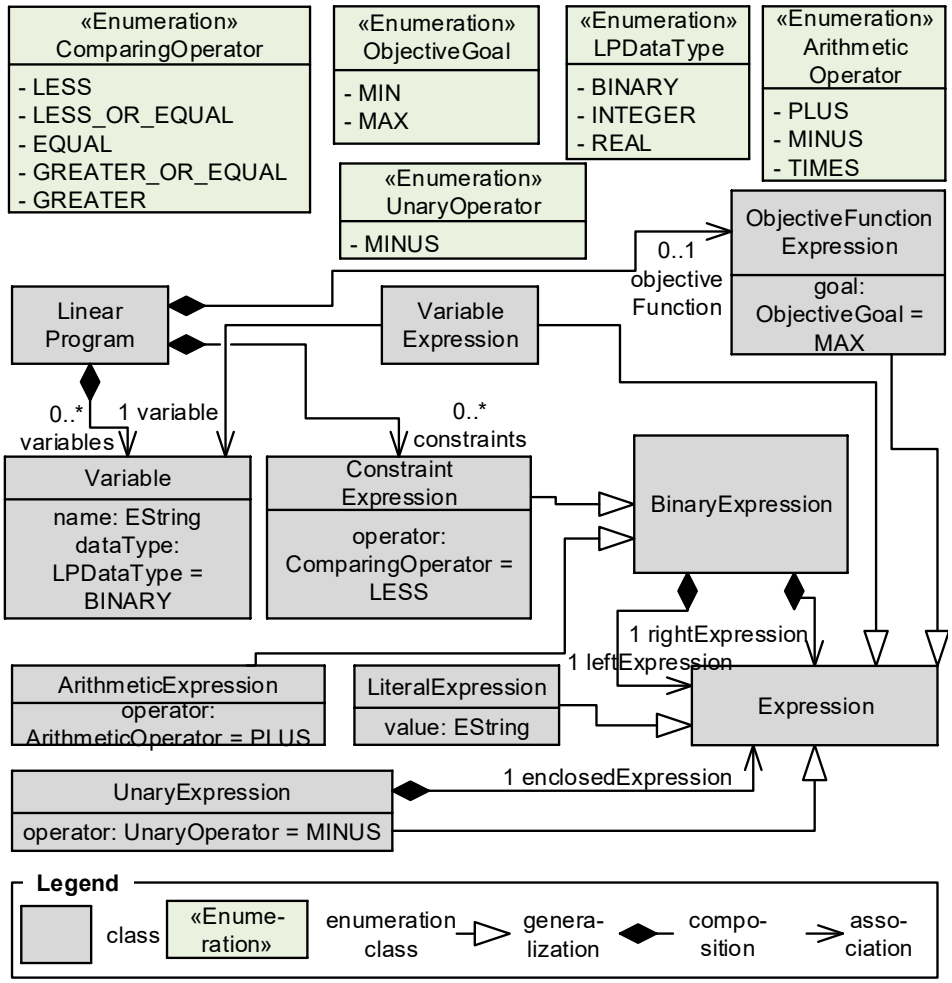


Figure B.7: Linear Program Meta-Model (Adapted from [*PH15])

B.6 Concrete LPSolve Syntax for Linear Programs

```
1
grammar de.uni_paderborn.fujaba.muml.allocation.ilp.lpsolve.xtext.LPSolve
    with org.eclipse.xtext.common.Terminals
import "http://www.muml.org/psm/allocation/ilp/1.0.0" as ilp
5 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.muml.org/core/1.0.0" as core
import "http://www.muml.org/core/expressions/1.0.0" as expressions
import "http://www.muml.org/core/expressions/common/1.0.0" as common

10 IntegerLinearProgram returns ilp::IntegerLinearProgram:
    objectiveFunction=ObjectiveFunctionExpression
    constraints+=ConstraintExpression*
    variables+=Variable*;

15 ObjectiveFunctionExpression returns ilp::ObjectiveFunctionExpression:
    (('min' | 'max') ':' ';') |
    goal=ObjectiveGoal':' objectiveFunction=LinearExpression ';';

enum ObjectiveGoal returns ilp::ObjectiveGoal:
20    MIN = 'min' | MAX = 'max';
```

```
ConstraintExpression returns ilp::ConstraintExpression:
    (comment=VariableID ':')? leftExpression=SimpleLinearExpression
    operator=ComparingOperator  rightExpression=SimpleLinearExpression ';';

enum ComparingOperator returns common::ComparingOperator:
    EQUAL='=' | GREATER='>' | GREATER_OR_EQUAL='>=' |
    LESS='<' | LESS_OR_EQUAL='<=';

Variable returns ilp::Variable:
    dataType=ILPDataType name=VariableID ';';

VariableID returns ecore::EString:
    ID ('.' (ID | INT ID?))*;

enum ILPDataType returns ilp::ILPDataType:
    BINARY = 'bin' | INTEGER = 'int' | REAL = 'real';

LinearExpression returns expressions::Expression:
    SimpleLinearExpression;

SimpleLinearExpression returns expressions::Expression:
    ArithmeticExpression;

enum ArithmeticOperator returns common::ArithmeticOperator:
    PLUS='+' | MINUS='-' | TIMES='*';

enum UnaryOperator returns common::ArithmeticOperator:
     MINUS='-' ;

UnaryExpression returns expressions::UnaryExpression:
    operator=UnaryOperator enclosedExpression=LinearExpression
;

ArithmeticExpression returns expressions::ArithmeticExpression:
    Operand (
        {common::ArithmeticExpression.leftExpression = current}
        operator=ArithmeticOperator rightExpression=LinearExpression)?;


Operand returns expressions::Expression:
    NumberLiteralExpression | VariableExpression | UnaryExpression;

NumberLiteralExpression returns common::LiteralExpression:
    value=Number;

Number returns ecore::EString:
    '-'? (Decimal | INT);

Decimal returns ecore::EString:
    INT '.' INT;

VariableExpression returns ilp::VariableExpression:
    variable=[ilp::Variable|VariableID];
```

C Supplementary Material for the Deployment Configuration Language

C.1 MechatronicUML Deployment Configuration Meta-Model

Figure C.8 shows the class diagram of the deployment package to define the deployment configuration view. It defines the classes DeploymentConfiguration, ECUConfiguration, ComponentContainer, ContainerComponentInstanceConfiguration, PortInstanceConfiguration, PortInstanceConfiguration_Local, and PortInstanceConfiguration_DDS.

DeploymentConfiguration represents the deployment configuration for the code generation for a specific application model and hardware platform model. It contains for each used ECU an ECUConfiguration. ECUConfiguration represents the concrete deployment configuration for one ECU. Therefore, it references a StructuredResourceInstance. Component instances may be allocated to an ECU. For each component type of the allocated component instances the ECUConfiguration contains a ComponentContainer. ComponentContainer represents the container that handles the technical concerns and manages the lifecycle of the referenced ComponentType. Furthermore, it references all ComponentInstances for which it is responsible, For each component instance its contains a ContainerComponentInstanceConfiguration. ContainerComponentInstanceConfiguration represents the configuration of a concrete component instance on a concrete ECU. It contains for each of its port instances a PortInstanceConfiguration. PortInstanceConfiguration represents as an abstract class the super type of all concrete PortInstanceConfugurations that exists for different communication realizations. it references the corresponding port instance that it represents and optional the hardware port that should be used for realizing the communication. PortInstanceConfiguration_Local is a PortInstanceConfiguration and represents the realization of the communication via shared memory on a single ECU. It has two properties that are used to realize the communication locally. PortInstanceConfiguration_DDS is another PortInstanceConfiguration. It represents the concrete configuration for realizing the communication of a port instance via Data Distribution Service (DDS). Therefore, it also has two required properties that are used to realize this communication.

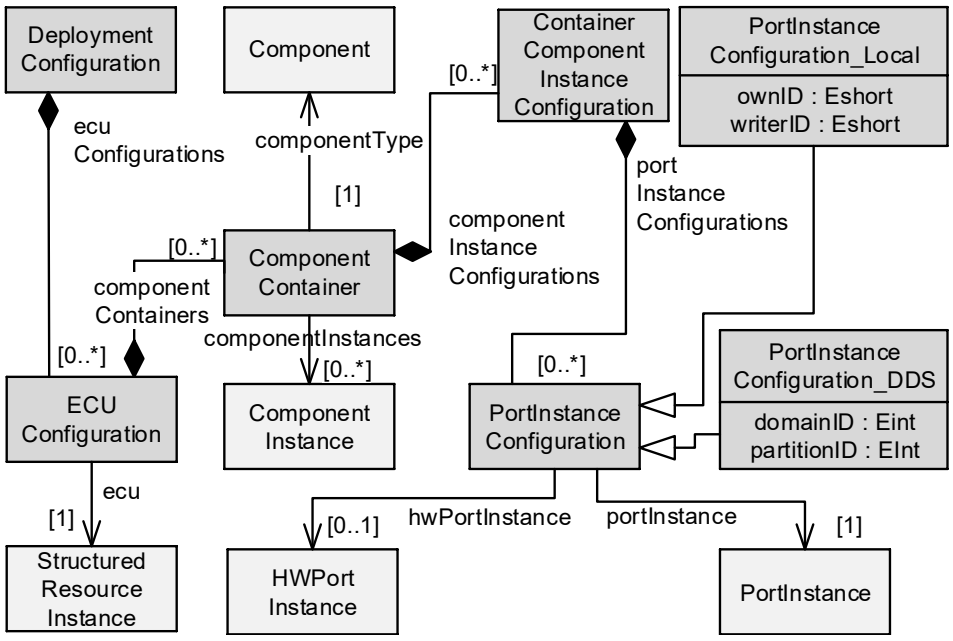


Figure C.8: Class Diagram of the Deployment Configuration Meta-Model

C.2 Concrete ApiMl Syntax

Listing C.2 shows the Xtext grammar for the ApiMl language within MechatronicUML.

```
OperatingSystem = 'OperatingSystem:' ID '{' APIRepository*'}' ';'
APIRepository = 'Device_API_Commands:' ID '{' APICommand* '}'';'
APICommand = DataType ID '('Parameter (','Parameter)* ')' [ '['TimeConstraint']']';'
Parameter = DataType ID
TimeConstraint = Number JAVA::TIMEUNIT
```

Listing C.2: Grammar: API Modeling Language

C.3 Concrete APIMappingMl Syntax

Listing C.3 shows the Xtext grammar for the ApiMl language within MechatronicUML.

```
MappingRepository = 'MappingRepository:' ID
    '{'PortApiMapping (','PortApiMapping)*'}'
PortApiMapping = 'PortInstance:'
    PortInstance '{''execCommand:' Actionlanguage::EntryRule'}'
    ['initCommand:' Actionlanguage::EntryRule]
    Actionlanguage::Expression (APICallExpression| Actionlanguage::LogicalExpression)
APICallExpression =
    APIML::APICommand '(' Actionlanguage::ParameterBinding (','Actionlanguage::
        ParameterBinding)*')'';'
```

Listing C.3: Grammar: APIMappingMl

D Supplementary Material for the Software Construction Explanation

D.1 Component Context Object Pattern

The Context Object Pattern is useful if information should be shared, e.g., about the system status without using a global coupling, e.g., by using global variables. As a result of using a context object, the included information is encapsulated and can be provided to functions that need the information. The pattern is used in the context of loosely coupled systems where different parts share information about their execution context. In our case, the component container has the knowledge about the network connection and the port connection status. Each component instance needs this information for executing its behavior that depends on the correct connection, e.g., the port behavior. Propagating many items of fine-grained information, like the connection status of each port instance, as individual global variables and functions does not scale well and is getting unmanageable. Instead, we introduce a context object for each component instance that encapsulates the context information and provides functions to read context information and to manipulate the context information. The object can be passed to functions that need information about the context. The benefit of the context object is that "it is possible to modify the execution context of a component without necessarily having to modify the configuration or code of other parts of a system. It is also possible to run with multiple, different contexts within the same program, perhaps in different threads." [BHS07]

Figure D.9 shows the *Context Object Pattern* for container and components. The pattern provides the port status via several functions. The instances of components and the corresponding context objects belong to a container that takes care of the objects. The

component context is in our case a struct (cf. Listing 5.1 on Page 174) of the component. The context provides functions for setting and getting the port status and for setting and getting component parameter values. The component has references to the corresponding elements and can read each port status and each parameter value. The component container sets each status and each parameter value during runtime. The status of a port is defined by the PortStatus enumeration. A port can have the status INACTIVE, CONNECTED, UNCONNECTED, and CONNECTIONLOST.

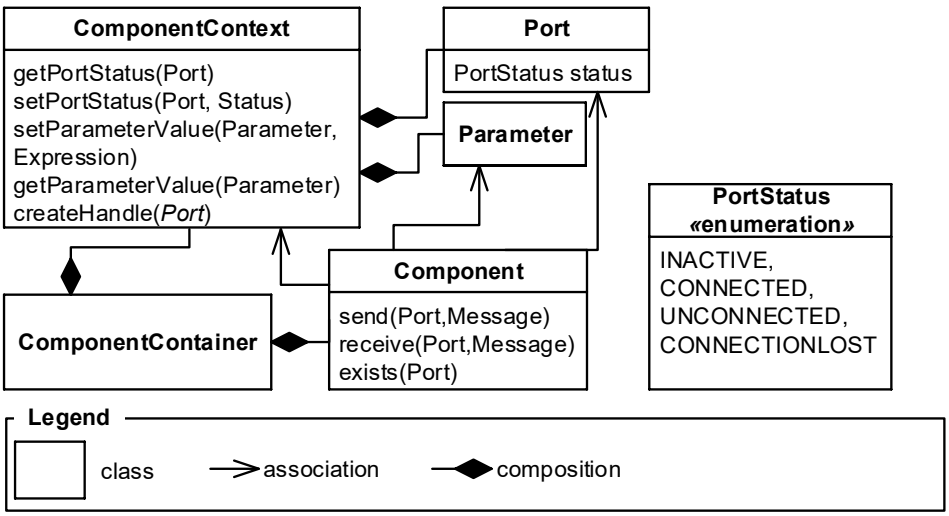


Figure D.9: Component Context Pattern in the Context of Containers and Components

D.2 Handle Pattern

The Handle Pattern is useful if different kinds of information should be stored and used for the same purpose, e.g., describing a communication partner as an IP address, socket address, or memory address. As a result of using a concrete handle object that refines an abstract data type, the included information and implementation are encapsulated. The pattern is used if an object should be shielded from dealing with the concrete manifestation. In our case, the component code should be shielded from dealing with low-level communication issues for each port. The same type of a component and moreover the same type of a port can be instantiated and connected several times. For each instance may be a different kind of communication has to be established, e.g., via publish-subscribe, shared memory, or request-reply. The problem is, functions for sending and receiving must be provided without knowing how to handle the sending and receiving on the concrete platform when constructing the component type. Therefore, the solution is to introduce a port handle as a variable. The variable represents a reference to an abstract data type, which is unknown for the concrete component code. The benefit of the port handle is that the component container can provide the actual implementation. The component container can provide for each component instance and respectively for each port instance an individual handle that contains any needed information." [BHS07]

Figure D.10 shows the Handle Pattern in the context of containers, component contexts, components, and ports. The ComponentContainer provides the concrete port handles. The Port has a reference to the abstract data type PortHandlePtr. A PortHandle is a PortHandlePtr and contains specific information for communication, e.g., a socket number, socket address, or a type identification number that defines the kind of port handle.

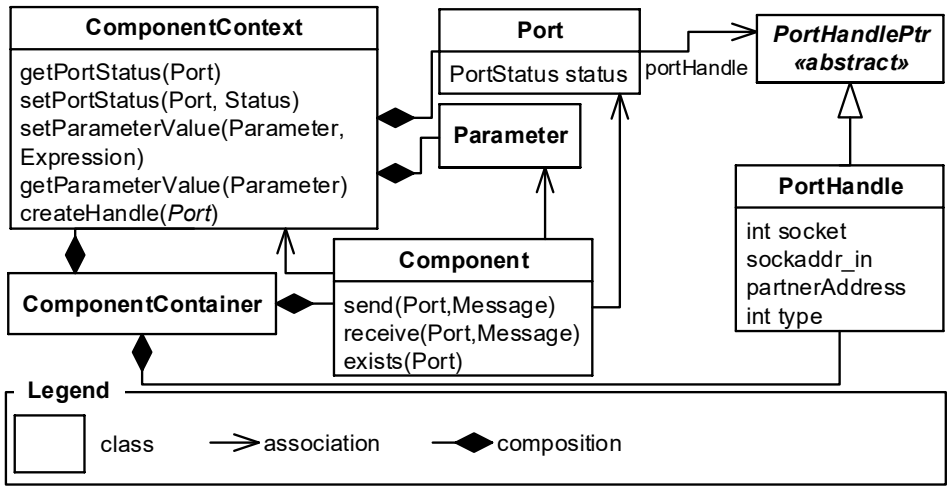


Figure D.10: Handle Pattern in the Context of Containers, Component Contexts, Components, and Ports

D.3 Builder Pattern

The Builder Pattern is useful if the construction of a complex object should be separated from the representation of a complex object. As a result, different instances can be created via the same construction/builder function. The Builder Pattern is used in the context, when someone wants to create a specific 'product' but does not want to take care about how to create this product. In our case, the component container has to create some instances of a specific component. The instances may differ in the number of ports, parameter bindings, context status, or communication partner. The problem is that the building process may be very complex and may differ between different instances. Therefore, the solution is to introduce a builder that provides functions for creating specific parts and a director for creating a concrete object by using the builder. The benefit of the separation is that it supports a flexible variation of an object's creation by encapsulating the basic creation mechanisms of the object's different parts. [BHS07]

Figure D.11 shows the Builder Pattern in the context of components and respectively in the context of creating concrete component instances. The container calls the createInstance function of the ComponentDirector. The director performs everything in the correct order that is required to create an instance of the component type, e.g., creating the component structure, creating structure and handles for each port type, and creating the component context. It uses the ConcreteComponentInstanceBuilder, which encapsulates the logic how to build the concrete parts of a concrete component instance. The result of calling the create function is an instance of the component type.

D.4 Lifecycle Callback Pattern

Lifecycle Callback Pattern is useful if two objects have to cooperate with each other via a well-defined interface in order to be used efficiently [VSW02]. As a result, an object can control another object via the known callback functions. The Lifecycle Callback Pattern is used in the context when an object controls the lifecycle of another object. That means the object controls the construction and destruction of the other objects and the activation and deactivation of the other objects if they can be suspended. In our case, each component container controls the lifecycle of the component instances of a specific component type. The problem is that

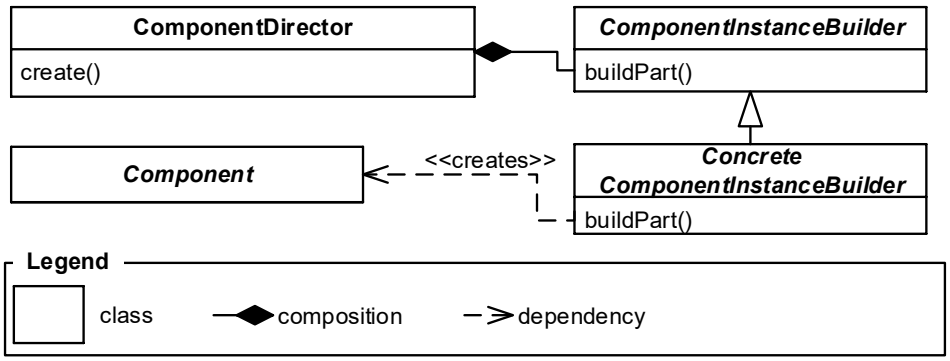


Figure D.11: Builder Pattern in the Context of Components and Component Instances (Adapted from [Dan16])

the container depends on the correct implementation of certain lifecycle functions of each component, which it can call. Therefore, the solution is that the component lifecycle callback interface defines that each component has to provide the functions for creating, initializing, and for destroying component instances and to free corresponding memory. Furthermore, the interface defines functions for activation and deactivation to support reconfiguration in the future. The benefit of the lifecycle callback pattern is that the component executes the tasks that the container expects [VSW02].

Figure D.12 shows the Lifecycle Callback Pattern in the context of containers and components. The ComponentContainer calls the functions initialize(), destroy(), activation(), and deactivation() at well-defined points ins time during the lifecycle of an component instance. In our case, the lifecycle callback pattern is combined with the builder pattern. The initialize function calls the create function if the component directory, which uses the concrete builder. It ensures that the component instance is correctly created and initialized. The result of using the lifecycle callback pattern is that the containers manage the lifecycle of each component instance and the implementation of the component provides the logic of the corresponding lifecycle functions.

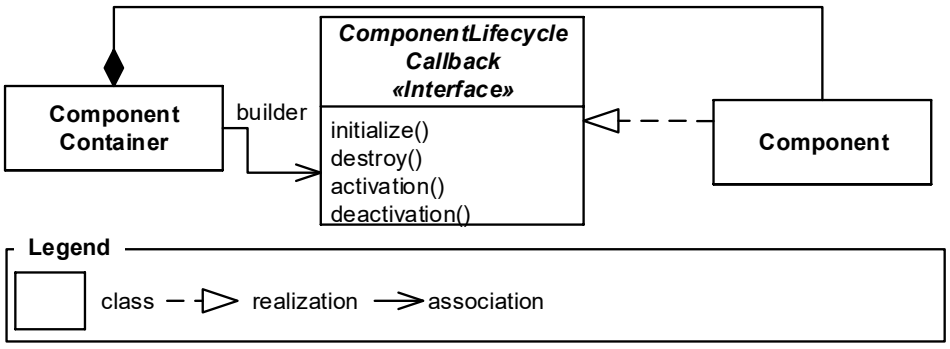


Figure D.12: Lifecycle Callback Pattern in the Context of Containers and Components

D.5 Supplementary Material for the Hybrid Port Semantics Definition

D.5.1 In-Port

Figure D.13 shows the parametrized Real-Time Statechart that represents the semantics of a hybrid in-port. A new region PeriodicTansmission_receive_hybrid_in is added to the component Real-Time Statechart and the original behavior is unchanged. This region is the counterpart of the region PeriodicTransmission_send. The region PeriodicTransmission_receive_hybrid_in periodically receives the message message(DOUBLE), within the sampling time of the hybrid port and assigns the received value to the variable apiValue. If no message is received in time or if the message failure is received, the state Failure gets active until a correct message is received again.

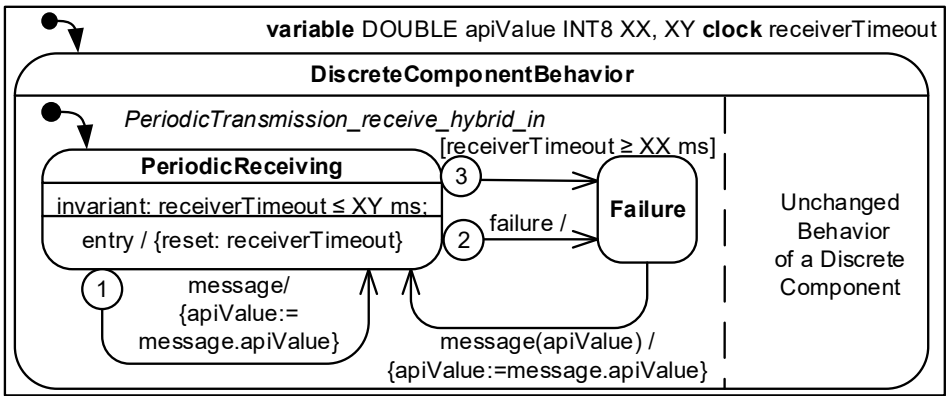


Figure D.13: Semantics Definition of Hybrid In-Ports

D.5.2 Out-Port

Figure D.14 shows the parametrized Real-Time Statechart that represents the semantics of a hybrid out-port. A new region PeriodicTansmission_send_hybrid_out is added to the component Real-Time Statechart and the original behavior is unchanged. This region is the counterpart of the region PeriodicTransmission_receive. The region PeriodicTransmission_send_hybrid_out periodically sends the message message(apiValue), within the sampling time of the hybrid port, i.e. 30 ms. If the statechart receives a failure message it changes to the state Failure. In the next step, the statechart sends the message message(apiValue) again and changes back to the state PeriodicSending.

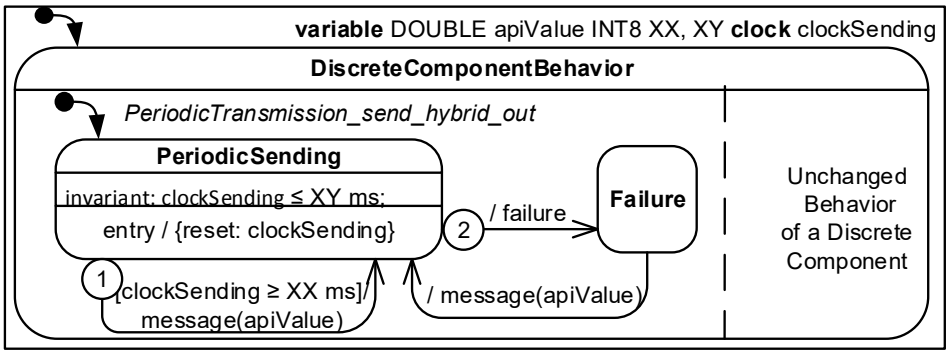


Figure D.14: Semantics Definition of Hybrid Out-Ports

D.6 Supplementary Material for the Makefile Explanation

A makefile consists of rules that define a target. After a colon (:) the prerequisites are stated in form of other targets or files. In a new line that starts with a TAB an arbitrary command can be stated.

Listing D.4 shows the generated makefile that defines the build that Figure 5.21 shows. Line 2 defines the make variable DDSHOME that contains the path to an installed DDS middleware SDK on the development computer. The compiler requires this path to compile the container that depends on DDS. Line 3 defines the make variable SYSLIBS that contains the names of required system libraries. Line 6 defines specific compile flags within the make variable DEFINES for the DDS middleware RTI Connext [RTIa] that we use. Lines 8–9 define compile flags within the make variable CFLAGS, e.g., for enabling debug support an warnings. Furthermore, they add further include paths. The variable CFLAGS includes the values of the variable DEFINES. Line 11 defines the all target. As a result, the developer can start the build by running the command *make all*. The all target depends on the program target that is defined by the Lines 13–17. The program target requires all object files and links them by using the gcc command to the output file program. Line 19–20 state the target main.o that requires the file main.c. It executes the compile command gcc on the file main.c using the options that are stored in the variable CFLAGS. We generate an own target for all source files that have to be compiled. Lastly, Line 23–14 define the clean target that deletes all object files and the program file when the command make clean is called. Afterward, the command *make all* builds all files from scratch.

```
1  # provide the correct path to your dds middleware
   DDSHOME := /home/user/ddsMiddleware
   # link libaries: dynamic link, socket, math, pthread, real time
   SYSLIBS = -ldl -lnsl -lm -lpthread -lrt
5  # DDS specific compile flags
   DEFINES = -DRTI_UNIX -DRTI_LINUX -DRTI_64BIT
   # compile flags for enable debug and warnings, and include paths
   CFLAGS = -ggdb -Wall -c  $(DEFINES) -I types -I lib
   -I $(DDSHOME)/include -I $(DDSHOME)/include/ndds
10 #all target
   all: program
   #-o link files to output file
   program : main.o OvertakeeComponent.o OvertakeeContainer.o deviceAPI.o
   operationRepository.o MUML_DDSSupport.o MUML_DDSPlugin.o MUML_DDS.o
15 [TAB]gcc main.o OvertakeeComponent.o OvertakeeContainer.o deviceAPI.o
   operationRepository.o MUML_DDSSupport.o MUML_DDSPlugin.o
   MUML_DDS.o  -o program
   #compile targets
   main.o: main.c
20 [TAB]gcc $(CFLAGS) main.c
   ...
   #clean target
   clean:
   [TAB]rm -rf *o program
```

Listing D.4: Makefile

The generated files for the type code, container code, and middleware code are located in different folders after the generation. The generated makefiles for each ECU expects a specific location for each source file and header file. We have to copy the generated files to the right location to execute the make command. Especially, we have to copy the generated type code (folders: components, lib, messages, RTSCs, types) and its libraries (folder: lib)

into each ECU folder. Furthermore, we have to copy the generated DDS code to the folder "dds" into each ECU folder. Finally, we have to copy the corresponding files for accessing the distance sensor to the folder "APImappings" for both cars. Alternatively, the originally generated APImappings code stubs can be enhanced with code for testing purposes manually and copied instead. We added for test runs on standard Linux machines the command `scanf(''%d'',pdist);` that simulates the distance sensor by user input. Note that this command blocks the execution of the program and is therefore only usable for testing purposes. Afterward, we have set the environment variable NDDSHOME in the makefile to the location of the RTI DDS installation and have to ensure that a valid RTI Connext license [RTIa] is installed on the machine where we like to compile the program. Note, detailed instructions for the cross compilation of RTI DDS programs for the Raspberry Pi is described by RTI in [RTIb]. Furthermore, the LIBS variable within the makefile have to be set to the DDS library path, e.g., "$(NDDSHOME)/lib/x64Linux3.xgcc4.6.3/". Now, everything is ready for building the software for each ECU by executing the make command within each ECU folder.

E Supplementary Case Study Model Descriptions

This section provides a detailed description of the models that we use for the case study, which Section 5.5 describes.

E.1 Allocation Constraints for the Cooperating Overtaking Scenario

Listing E.5 shows the constraint specification for the cooperating overtaking scenario that Section 5.5 uses within the case study. Line 2 includes our OCL allocation specification library (cf. Section 4.5.2). Line 3 and line 4 reference the name and storage provider for MechatronicUML (cf. Appendix B.3). Line 5 sets the OCL context to the MechatronicUML specific context class OCLContext (cf. Section 4.5.1.1). Lines 6–8 import the required MechatronicUML meta-models. Lines 10–13 define the relation, i.e., the concrete component instances that should be allocated to specific ECUs (cf. Appendix B.1). Lines 15–20 show the constraint requiredLocationYellow that enforces that all children of the structure component named yellow are allocated to the ECU named ECUYellow. Lines 21–26 show the constraint requiredLocationRed that enforces that all children of the structure component named red beside the component instance named C3 are allocated to the ECU named ECURed. Lines 27–31 show the constraint requiredLocationRedDistance that the component instance named C3 is allocated to the ECU named ECURedDistance. Thereby, we can run the component instance named C3 as an independent process and use also blocking input functions without influencing the execution of the other component instances. Lines 32–36 show the constraint requiredLocationSection that the component instance named C7 is allocated to the ECU named ECUSectionControl.

E.2 Real-Time Coordination Protocol for the Cooperating Overtaking Scenario

In the upper part, Figure E.15 shows the Real-Time Coordination Protocol OvertakeRequest, which is displayed within the dashed ellipse. The protocol describes the communication between the requestor role and the provider role. The coordination role requestor has the sender message types requestOvertaking and overtakingDone. Furthermore, it has the receiver message types startToOvertake and abortOvertake. The receiver and sender message types of the coordination role provider are vice versa. The coordination roles requestor and provider have a buffer size of 1 for each message type. Both roles store each received message type within an own First in, First Out (FIFO) queue. Additionally, both roles discard an incoming message if the message queue of the corresponding message type is full. The connector of the coordination protocol Overtaking Request requires that a message that is sent via communication medium requires at most 250 ms until it is consumed. Furthermore, the connector allows that messages can get lost.

In the middle part, Figure E.15 shows the Real-Time Statechart (RTSC) of the requestor coordination role. It consists of the simple states init, WaitForAnswer, Overtake, and GetSafeDistance. The initial state init reflects the status when a car drives in a normal mode and does not like to overtake any other car. The state has a self-transition with a non-deterministic choice expression that reflects that the value of the variable pDist can switch from 99 to 100 or vice versa at any time. The RTSC switches from the state init to the state WaitForAnswer when the value of the variable pDist is less than 100. During the state switch, the role requestor sends the message requestOvertaking to the role provider and reset its timeout clock. The state WaitForAnswer reflects the state where the requestor role waits for an answer from the provider role. It has a time invariant, which allows that the state is not allowed to be active for longer than 600 ms. The statechart switches from state WaitForAnswer to the state GetSafeDistance if the timeout clock becomes greater than 500 ms or when it receives the message abortOvertake. It switches to the state Overtake and reset the timeout clock if it receives the message startToOvertake. The state Overtake reflects the state where the requestor role is allowed to overtake. It has a time invariant, which allows that the state is not allowed to be active for longer than 15 s. Furthermore, it has a self-transition with a non-deterministic choice expression that reflects that the value of the variable overtakingDone can switch from 0 to 1 or vice versa at any time. The statechart switches from the state Overtake to the state GetSafeDistance if either the value of the timeout clock is greater than 14.5 s or the value of the variable overtakingDone is equal to 1. In the second case, it sends the message overtakingDone to the other role. The state GetSafeDistance reflects the state where the overtaking maneuver must be finished. The state has a self-transition with a non-deterministic choice expression that reflects that the value of the variable pDist can switch from 99 to 100 or vice versa at any time. The statechart switches from the state GetSafeDistance to the state init when the timeout clock has a value of at least 18 s and the value of the variable pDist is greater or equal to 100.

In the lower part, Figure E.15 shows the RTSC of the provider coordination role. It consists of the simple states init, RequestedOvertake, and Overtaken. The initial state init reflects the status when no car has sent a request to overtake to the section controller. The statechart switches to the state RequestedOvertake if it receives the message requestOvertaking, reset the timeout clock, and sets the variable choice to either 0, 1, 2 depending on the result of the non-deterministic choice expression. The state requestOvertaking reflects the status when a car has sent a request to overtake to the section controller. The statechart switches from the state RequestedOvertake to the state init and sends the message abortOvertake if the value of the variable choice is equal to 0. Otherwise, if the value of the variable choice is equal to 1, it switches from the state RequestedOvertake to the state Overtaken, sends the message startToOvertake, and reset

the timeout clock. If the value of the variable choice is equal to 2 it remains within the state RequestedOvertake until the timeout clock becomes at least 1 s. In that case it switches to the state init and sends the message abortOvertake. Thereby, the provider has at most 1 s to take the decision, which may depend on the answer of another car. The state Overtaken reflects the state when an overtaking maneuver takes place. The maneuver runs either in a timeout when the timeout clock becomes larger or equal to 16 s or when the provider receives the message overtakingDone. In both cases, the statechart switches to the state init.

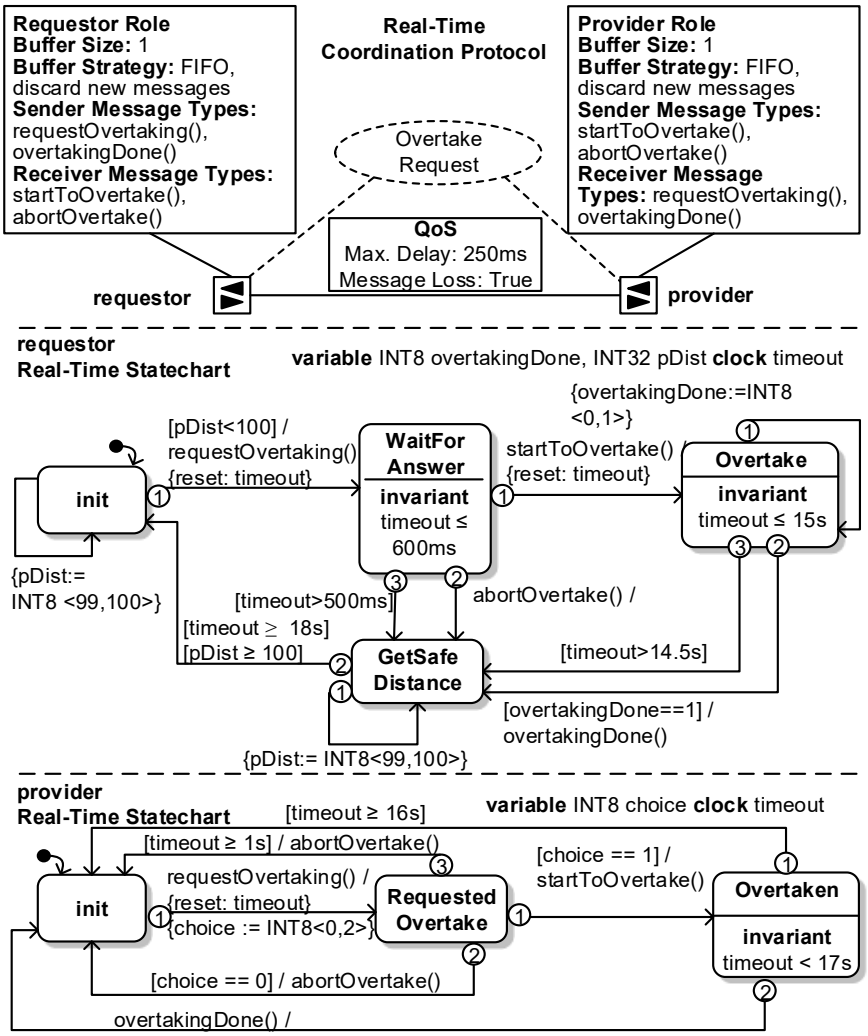


Figure E.15: Cooperating Overtaking Overtake Request Protocol

In the upper part, Figure E.16 shows the Real-Time Coordination Protocol Delegate Overtake Request, which is displayed within the dashed ellipse. The protocol describes the communication between the delegator role and the receiver role. The coordination role delegator has the sender message types askForOvertakingPermission and finishedOvertaking. Furthermore, it has the receiver message types overtakingAllow and overtakingDisallow. The receiver and sender message types of the coordination role receiver are vice versa. The coordination roles delegator and receiver have a buffer size of 1 for each message type. Both roles store each received message type within an own FIFO-queue. Additionally, both roles discard an incoming message if the message queue of the corresponding message type is full. The connector of the coordination protocol Delegate Overtake Request requires that a message that is sent via communication medium requires at

most 250 ms until it is consumed. Furthermore, the connector allows that messages can get lost.

In the middle part, Figure E.16 shows the RTSC of the delegator coordination role. It consists of the simple states init, WaitForAnswer, Allow, and FinishedRequest. The initial state init reflects the status when a section controller (delegator) do not need to delegate a request. The state has a self-transition with a non-deterministic choice expression that reflects that the value of the variable requestRequired can switch from 0 to 1 or vice versa at any time. The RTSC switches from the state init to the state WaitForAnswer when the value of the variable requestRequired becomes 1. During the state switch, the role delegator sends the message askForOvertakingPermission to the role receiver and reset its timeout clock. The state WaitForAnswer reflects the state where the delegator role waits for an answer from the receiver role. It has a time invariant, which allows that the state is not allowed to be active for longer than 600 ms. The statechart switches from state WaitForAnswer to the state FinishedRequest if the timeout clock becomes greater than 500 ms or when it receives the message overtakingDisallow. It switches to the state Allow and reset the timeout clock if it receives the message overtakingAllow. The state Allow reflects the state where the receiver allows to overtake. It has a time invariant, which allows that the state is not allowed to be active for longer than 15 s. Furthermore, it has a self-transition with a non-deterministic choice expression that reflects that the value of the variable finsihedOvertaking can switch from 0 to 1 or vice versa at any time. he statechart switches from the state Allow to the state FinishedRequest if either the value of the timeout clock is greater than 14.5 s or the value of the variable finishedOvertaking is equal to 1. In the second case, it sends the message finishedOvertaking to the other role. The state FinishedRequest reflects the state where the request is done. The statechart switches from the state FinishedRequest to the state init when the timeout clock has a value of at least 18 s.

In the lower part, Figure E.16 shows the RTSC of the receiver coordination role. It consists of the simple states idle, GetRequest, and BeingOvertaken. The initial state idle reflects the status a car drives standalone and no other car wants to overtake it. The statechart switches to the state GetRequest if it receives the message askForOvertakingPermission and sets the variable choice to either 0 or 1 depending on the result of the non-deterministic choice expression. The state GetRequest reflects the status when a car has received a request to overtake it. The statechart switches from the state GetRequest to the state init and sends the message overtakingDisallow if the value of the variable choice is equal to 0. Otherwise, if the value of the variable choice is equal to 1, it switches from the state GetRequest to the state BeingOvertaken, sends the message overtakingAllow, and reset the timeout clock. The state BeingOvertaken reflects the state when an overtaking maneuver takes place. The maneuver runs either in a timeout when the timeout clock becomes larger or equal to 16 s or when the provider receives the message finishedOvertaking. In both cases, the statechart switches to the state init.

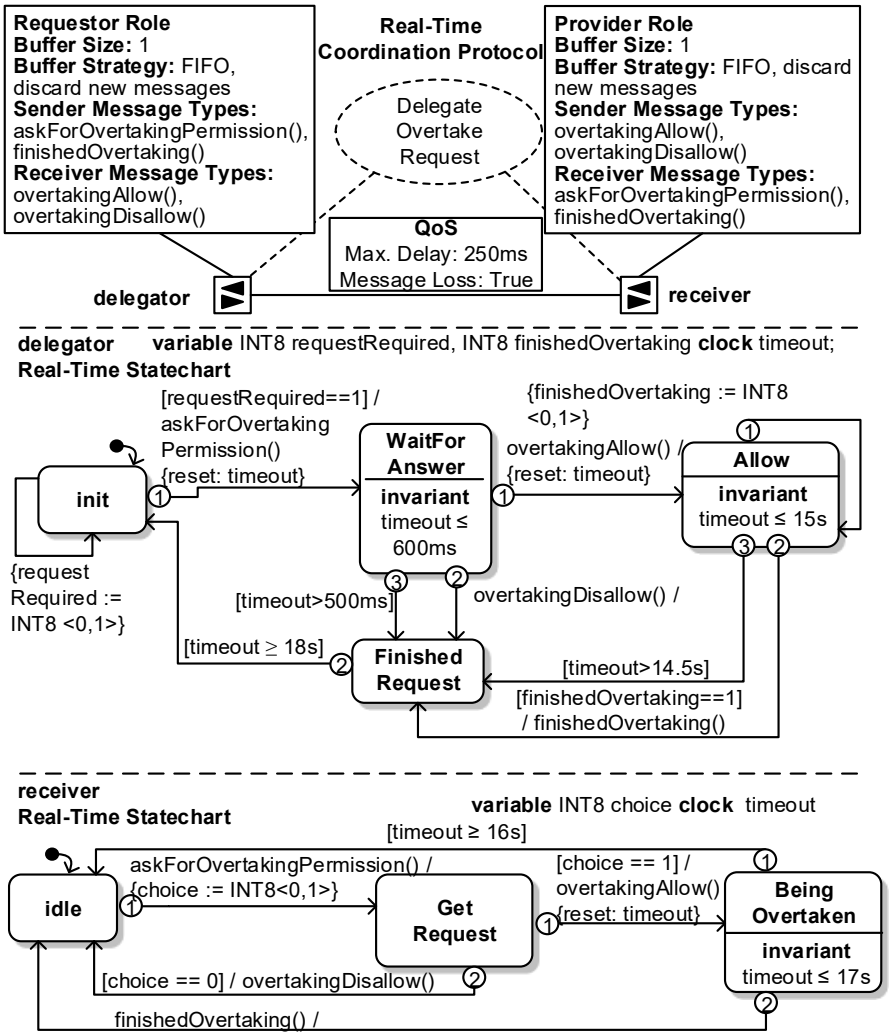


Figure E.16: Cooperating Overtaking Delegate Overtake Request Protocol

E.3 Verified Safety and Liveness Properties

The Real-Time Coordination Protocols (RTCPs) that Figure E.15 and Figure E.16 show fulfill the safety and liveness properties that we describe in this section. The properties are defined and verified by using the domain-specific timed model checking approach of Dziwok [Dzi17]. The following verification properties hold:

1. `AG (stateActive(requestor.Overtake) implies stateActive(provider.Overtaken));`
2. `AG (stateActive(delegator.Allow) implies stateActive(receiver.BeingOvertaken));`
3. `AG not deadlock;`
4. `AG not connectorOverflow;`
5. `forall(t : Transitions) EF transitionFiring(t);`
6. `forall(s : States) EF stateActive(s);`
7. `forall(m : MessageTypes) exists(b : Buffers) EF messageInBuffer(m, b);`
8. `forall(m : MessageTypes) EF messageInTransit(m);`
9. `forall(b : Buffers) EF bufferMessageCount(b) >= 1;`

Property 1 requires whenever the requestor role is in the state Overtake, the provider role must be in the state Overtaken. Property 2 requires whenever the delegator role is in the state Allow,

the receiver role must be in the state BeingOvertaken. Property 3 requires that a deadlock never occurs. Property 4 requires that a connector overflow never occurs. Property 5 requires that there is no transition that can never be fired. Property 6 requires that there is no state that can never be active. Property 7 requires that there is no message type that cannot possibly arrive in a buffer. Property 8 requires that there is no message type that cannot possibly be in transit. Finally, property 9 requires that there is no buffer that is always empty.

E.4 Distance Sensor Access Specification for the Raspberry Pi Platform

Listing E.6 shows the part of the device Application Programming Interface (API) specified using the ApiMl (cf. Section 5.3.3). The device API is provided by the platform Wiring Pi Library. Line 1 defines the operating system name raspberryPi. Line 2 defines the API repository name gpio. Line 3 defines the API command digitalWrite. The command has the parameters pin and value. Both have the data type INT32. The command digitalWrtite has a time constraint that requires that it called at most every 30 ms. Line 4 defines the API command digitalRead. It has the INT32 parameter pin and the return type INT32. Line 5 defines the API command pinMode. It has the INT32 parameters pin and mode. Line 6 defines the API command wiringPiSetup and line 7 defines the API command micros with the return type INT64. Lastly, line 7 defines the API command delayMicroseconds with the INT32 parameter delay. We store the implementation using the file api.osdsl.

Listing E.7 shows the specification for initializing and accessing the ultrasonic distance sensor pdistC1 that is specified by using the APIMappingMl (cf. Section 5.3.3). The implementation delivers the sensor values for the continuous software component Distance of the red car. The specification of the port instance pdistC4 is similar as the yellow car uses the same operating system, libraries, hardware platform, and electrical wiring. Line 1 shows the URL of the original implementation. Line 2 defines an import that refers to the context MechatronicUML model file with the name org.muml.evaluation.execution.muml. Line 3 refers to the file that stores the implementation that Listing E.6 shows. Line 4 defines the mapping repository with the name myMap. Line 5 defines the mapping for the port instance pdist. The lines 6-14 define the initCommand. It defines variables, expressions, and commands that are executed to initialize the distance sensor. Especially, the variables TRIGPIN and ECHOPIN are set to define how these pins are wired. The lines 15-38 define the execCommand. It defines the implementation for accessing the distance sensor. It sets the value of the port instance pdist by writing to the corresponding variable within the code. The digitalWrite command initiates the distance measurement. The digitalRead command is used to get a response from the distance sensor and to measure the travelTime that the ultrasonic signal requires for the distance to an object in front and back again. We use this time to calculate the distance to an object that is in front of the sensor. The speed of sound is 340 m/s or 29 microseconds per centimeter. We can use the divisor 58 because we just want to know the distance to the object. In the case of an error, we set the value of the distance port to -1.

F Supplementary Collected Data

F.1 Modelica Transformation

Table F.1 shows the results of the time measurement while transforming both cases of the Modelica case study (cf. Section 3.6) from the MechatronicUML model to the Modelica model. We perform ten transformation runs. The first run is always the outlier because Acceleo make

some technical optimizations during the first run, which takes significant time. Acceleo also caches some results, which improves the performance after the first run.

Table F.1: Modelica Transformation: Execution Time Measurement Results

Run	Cooperating Delta Robots	Cooperative Overtaking Cars
1st	15346 ms	10668 ms
2nd	8663 ms	6023 ms
3rd	8151 ms	5213 ms
4th	7836 ms	4576 ms
5th	7557 ms	4337 ms
6th	7245 ms	4097 ms
7th	6708 ms	4346 ms
8th	7462 ms	4331 ms
9th	6700 ms	4149 ms
10th	6705 ms	3844 ms

F.2 Allocation Transformation

Table F.2 shows the statistics locations of the results of the time measurement while calculating the allocation specification for the Overtakee example. The Overtakee example consists of 8 component instances, 6 ECUs, and 8 constraints. Table F.3 shows the statistics locations of the results of the time measurement while calculating the allocation specification for the Brake-by-wire system. The Brake-by-wire system consists of 16 component instances, 9 ECUs, and 7 constraints.

According to Devore, the show statistics *locations* "serve to characterize the data set and convey some of its salient features"[Dev15]. The first row shows the aspect of each column and the first column shows the corresponding location that each row presents. The following columns except the last column show the execution time considering the corresponding transformation step and the statistics location. The last column shows the time for solving the created ILPs. The statistics locations are based on the performance measurement results that Table F.4 and Table F.5 in Appendix F.2 show. The first transformation run is always an outlier. QVTo and Acceleo make some technical optimizations and compilations during the first run, which takes significant time. Acceleo also caches some results, which improves the performance after the first run.

The transformation computes on average 123 ms for creating $8 * 6 = 48$ general Overtakee constraints (cf. Equation (4.10)) and it computes on average 271 ms for creating $16 * 9 = 144$ general Brake-by-wire constraints. Furthermore, the transformation computes on average 34 ms for transforming 1 Overtakee collocation tuple and on average 140 ms for transforming 3 Brake-by-wire collocation tuples. The transformation of the single Brake-by-wire separate tuple takes on average only 34 ms. Additionally, the transformation computes on average 614 ms for the 210 Overtakee required location constraints and it computes on average 1977 ms for the 748 Brake-by-wire location constraints. Especially, handling the routing constraints between component instances takes the highest amount of computational effort. The last part of the transformation is creating the tuples for required resource constraints. It computes on average 839 ms for 17 Overtakee resource tuples and 1634 ms for transforming 27 Brake-by-wire resource tuples. Solving the final Overtakee ILP, consisting of 263 binary variables and 537 constraints, takes on average 94 ms. Solving the final Brake-by-wire ILP, consisting of 945 binary variables and 2001 constraints, takes on average 313 ms.

Table F.2: Overtakee Allocation Transformation and Solving: Execution Time Measurement Descriptive Statistics

Statistics Location	General Constraints	Collocation Constraints	Separate Location Constraints	Required Location Constraints	Required Resource Constraints	ILP Solving
Average	122.60 ms	38.30 ms	34.20 ms	614.50 ms	838.70 ms	93.70 ms
Min	71 ms	21 ms	21 ms	405 ms	647 ms	77 ms
1st Quartil	76.50 ms	22.25 ms	21.75 ms	437.50 ms	697.75 ms	78.25 ms
Median	85.50 ms	25.50 ms	29.50 ms	522.50 ms	751.5 ms	82.5 ms
3rd Quartil	90 ms	34.50 ms	35.25 ms	624.50 ms	882 ms	95 ms
Max ms	431 ms	127 ms	82 ms	1288 ms	1348 ms	152 ms

Table F.3: Brake-by-wire Allocation Transformation and Solving: Execution Time Measurement Descriptive Statistics

Statistics Location	General Constraints	Collocation Constraints	Required Location Constraints	Required Resource Constraints	ILP Solving
Average	271.09 ms	140.82 ms	1976.55 ms	1634 ms	313.18 ms
Min	172 ms	92 ms	1811 ms	1682 ms	320 ms
1st Quartil	198 ms	96.25 ms	1851.25 ms	1721.75 ms	323.5 ms
Median	216.5 ms	103.5 ms	1876 ms	1750.5 ms	335 ms
3rd Quartil	229.5 ms	163 ms	2080 ms	1795 ms	349 ms
Max	982 ms	484 ms	4229 ms	2190 ms	430 ms

Table F.4 shows the results of the time measurement while transforming and solving the overtakee case of the allocation engineering case study (cf. Section 4.10). We perform ten transformation runs. The first transformation run is always an outlier. QVTo and Acceleo make some technical optimizations and compilations during the first run, which takes significant time. Acceleo also caches some results, which improves the performance after the first run.

Table F.5 shows the results of the time measurement while transforming and solving the Brake-by-wire case of the allocation engineering case study (cf. Section 4.10). We perform ten transformation runs. The first transformation run is always an outlier. QVTo and Acceleo make some technical optimizations and compilations during the first run, which takes significant time. Acceleo also caches some results, which improves the performance after the first run.

We analyze three different scaling situations using the Brake-by-wire case of the case study that Section 4.10 describes. Firstly, Table F.6 shows the raw data of the scenario that scales the number of Brake-by-wire software component instance configurations. Secondly, Table F.7 shows the raw data of the scenario that scales the number of Brake-by-wire hardware platform instance configurations. Thirdly, Table F.8 shows the raw data of the scenario that scales the number of Brake-by-wire software component instance configurations and hardware platform

Table F.4: Overtakee Allocation Transformation and Solving: Execution Time Measurements

Measurement	General Constraints	Collocation Constraints	Separate Location Constraints	Required Location Constraints	Required Resource Constraints	ILP Solving
1st	431 ms	127 ms	82 ms	1288 ms	1348 ms	152 ms
2nd	147 ms	49 ms	45 ms	871 ms	1096 ms	77 ms
3rd	91 ms	33 ms	33 ms	644 ms	791 ms	125 ms
4th	87 ms	27 ms	21 ms	566 ms	712 ms	86 ms
5th	86 ms	35 ms	36 ms	544 ms	900 ms	98 ms
6th	85 ms	23 ms	24 ms	501 ms	666 ms	85 ms
7th	81 ms	22 ms	21 ms	481 ms	647 ms	78 ms
8th	72 ms	22 ms	32 ms	422 ms	696 ms	77 ms
9th	75 ms	21 ms	21 ms	405 ms	828 ms	79 ms
10th	71 ms	24 ms	27 ms	423 ms	703 ms	80 ms

Table F.5: Brake-by-wire Allocation Transformation and Solving: Execution Time Measurement

Measurement	General Constraints	Collocation Constraints	Required Location Constraints	Required Resource Constraints	ILP Solving
1st	982 ms	484 ms	4229 ms	2190 ms	430 ms
2nd	355 ms	184 ms	2333 ms	1774 ms	357 ms
3rd	232 ms	96 ms	1861 ms	1756 ms	349 ms
4th	222 ms	93 ms	1891 ms	1802 ms	338 ms
5th	207 ms	105 ms	2122 ms	1711 ms	322 ms
6th	215 ms	102 ms	1851 ms	1745 ms	332 ms
7th	184 ms	92 ms	1838 ms	1682 ms	349 ms
8th	218 ms	97 ms	1954 ms	1859 ms	325 ms
9th	172 ms	118 ms	1852 ms	1716 ms	323 ms
10th	195 ms	178 ms	1811 ms	1739 ms	320 ms

instance configurations equally. Additionally, to the measured transformation and solving time, the third column of all three tables shows the memory load because the memory consumption constraint is the constraint that restricts at most the feasible solution space. The solving is much faster without the memory consumption constraint. Furthermore, the last column of all three tables describes if the examined model has a feasible solution. In the cases, where the memory consumption is larger than 100% no feasible solution exists.

Table F.6: Allocation Performance Analysis for Scaling the Software Configuration and the Same Hardware Configuration

# Brake-By-Wire SW/HW Instances	# Atomic SW/HW Instances	Memory load	General Constraints	Collocation Constraints	Required Location Constraints	Required Resource Constraints	Serializing	ILP Solving	Solvable
1/1	13/9	40.83%	399 ms	345 ms	190 ms	2088 ms	1682 ms	70 ms	yes
2/1	26/9	81.67%	766 ms	314 ms	148 ms	2287 ms	2775 ms	230 ms	yes
3/1	39/9	122.50%	1167 ms	361 ms	231 ms	3842 ms	3866 ms	120 ms	no
4/1	52/9	163.33%	1555 ms	508 ms	296 ms	5575 ms	5278 ms	60 ms	no
5/1	65/9	204.17%	1969 ms	543 ms	434 ms	8270 ms	6516 ms	210 ms	no
6/1	78/9	245.00%	2319 ms	654 ms	466 ms	11339 ms	7778 ms	310 ms	no
7/1	91/9	288.83%	3102 ms	792 ms	618 ms	13600 ms	9435 ms	470 ms	no
8/1	104/9	326.67%	3159 ms	1080 ms	806 ms	16726 ms	11627 ms	530 ms	no

Table F.7: Allocation Performance Analysis for Scaling the Hardware Configuration and the Same Software Configuration

# Brake-By-Wire SW/HW Instances	# Atomic SW/HW Instances	Memory load	General Constraints	Collocation Constraints	Required Location Constraints	Required Resource Constraints	Serializing	ILP Solving	Solvable
1/1	13/9	40.83%	399 ms	345 ms	190 ms	2088 ms	1682 ms	70 ms	yes
1/2	13/18	20.42%	842 ms	447 ms	281 ms	4088 ms	2926 ms	280 ms	yes
1/3	13/27	13.61%	1101 ms	495 ms	256 ms	3921 ms	3935 ms	540 ms	yes
1/4	13/36	10.21%	1499 ms	413 ms	324 ms	5169 ms	5236 ms	790 ms	yes
1/5	13/45	8.17%	1882 ms	584 ms	427 ms	6996 ms	6625 ms	830 ms	yes
1/6	13/54	6.81%	2267 ms	603 ms	506 ms	7667 ms	7671 ms	1780 ms	yes
1/7	13/93	5.83%	2639 ms	657 ms	621 ms	9312 ms	8788 ms	1700 ms	yes
1/8	13/72	5.10%	3028 ms	706 ms	729 ms	11665 ms	10118 ms	2120 ms	yes

Table F.8: Allocation Performance Analysis for Scaling the Software Configuration and the Hardware Configuration

# Brake-By-Wire SW/HW Instances	# Atomic SW/HW Instances	Memory load	General Constraints	Collocation Constraints	Required Location Constraints	Required Resource Constraints	Serializing	ILP Solving	Solvable
1/1	13/9	40.83%	345 ms	190 ms	2088 ms	1682 ms	399 ms	70 ms	yes
2/2	26/18	40.83%	637 ms	507 ms	7855 ms	6930 ms	1810 ms	920 ms	yes
3/3	39/27	40.83%	1214 ms	969 ms	15423 ms	11709 ms	3862 ms	8110 ms	yes
4/4	52/36	40.83%	1777 ms	1378 ms	32831 ms	22763 ms	6476 ms	49850 ms	yes
5/5	65/45	40.83%	2715 ms	2417 ms	74967 ms	33050 ms	9947 ms	98910 ms	yes
6/6	78/54	40.83%	4170 ms	3664 ms	142362 ms	50101 ms	14535 ms	166600 ms	yes
7/7	91/93	40.83%	5752 ms	5113 ms	338805 ms	61315 ms	19319 ms	380080 ms	yes
8/8	104/72	40.83%	8773 ms	8056 ms	654605 ms	80989 ms	25081 ms	698120 ms	yes

```
OvertakingAllocationSpecification{
include 'platform:/plugin/org.muml.psm.allocation.language.xtext/operations/OCLContext.ocl'
nameProvider 'org.muml.psm.allocation.language.xtext.provider.MUMLNameProvider';
storageProvider 'org.muml.psm.allocation.language.xtext.provider.MUMLStorageProvider';
oclContext 'oclcontext::OCLContext'
import 'http://www.muml.org/pm/hardware/hwresourceinstance/1.0.0'
import 'http://www.muml.org/pim/1.0.0'
import 'http://www.muml.org/psm/allocation/language/oclcontext/1.0.0'

relation {
  descriptors (first:instance::ComponentInstance, second:hwresourceinstance::ResourceInstance);
      ocl self.getAllSWInstances()->product(self.getAllStructuredHWInstances());
  }

constraint requiredLocation requiredLocationYellow {
  descriptors (first:instance::ComponentInstance, second:hwresourceinstance::ResourceInstance);
  ocl   self.getSWInstance('yellow')->union(self.getSWInstance('yellow').
    getAllEmbeddedInstances())->asSet()->product(self.getECU('ECUYellow').
    resolveToStructuredResourceInstances())->asSet();
}
constraint requiredLocation requiredLocationRed {
  descriptors (first:instance::ComponentInstance, second:hwresourceinstance::ResourceInstance);
  ocl   self.getSWInstance('red')->union(self.getSWInstance('red').getAllEmbeddedInstances())
    ->
  select(inst|inst.name<>'C3')->asSet()->product(self.getECU('ECURed').
  resolveToStructuredResourceInstances())->asSet();
}
constraint requiredLocation requiredLocationRedDistance {
  descriptors (first:instance::ComponentInstance, second:hwresourceinstance::ResourceInstance);
  ocl   self.getSWInstance('C3')->asSet()->product(self.getECU('ECURedDistance').
  resolveToStructuredResourceInstances())->asSet();
}
constraint requiredLocation requiredLocationSection {
descriptors (first:instance::ComponentInstance, second:hwresourceinstance::ResourceInstance);
  ocl   self.getSWInstance('C7')->asSet()->product(self.getECU('ECUSectionControl').
  resolveToStructuredResourceInstances())->asSet();
}}
```

Listing E.5: Allocation Constraints for the Cooperating Overtaking Scenario

```
OperatingSystem:raspberryPi{
  Device_API_Calls:gpio{
    void digitalWrite(INT32 pin, INT32 value)[30 MICROSECONDS];
     INT32 digitalRead(INT32 pin);
     void pinMode(INT32 pin, INT32 mode);
     void wiringPiSetup();
     INT64 micros();
     void delayMicroseconds(INT32 delay);
  }
}
```

Listing E.6: Software Platform Specification of the Raspberry Pi and its Wiring Pi Library using the APIML

```
1  //https://ninedof.wordpress.com/2013/07/16/rpi-hc-sr04-ultrasonic-sensor-mini-project/
   import ''org.muml.evaluation.execution.muml''
   import ''api.osdsl''
   MappingRepository:myMap{
5    PortInstance:pdist{
       initCommand:{
         INT32 TRIGPIN := 5;
         INT32 ECHOPIN := 6;
         INT32 LOWVAL :=0;
10       INT32 OUTPUTMODE :=1;
         wiringPiSetup() ;
         pinMode(pin:=ECHOPIN,mode:=OUTPUTMODE) ;
         digitalWrite(pin:=TRIGPIN, value:=LOWVAL) ;
       }
15     execCommand:{
         INT32 TRIGPIN := 5;
         INT32 ECHOPIN := 6;
         INT32 HIGHVAL :=1;
         INT32 LOWVAL :=0;
20       digitalWrite(pin:=TRIGPIN, value:=HIGHVAL) ;
          delayMicroseconds(delay:=30) ;
           digitalWrite(pin:=TRIGPIN, value:=LOWVAL) ;
          INT32 startTime := -1;
           while(digitalRead(pin:=ECHOPIN) == LOWVAL) {
25           startTime := micros() ;
           }
           INT32 endTime := -1 ;
           while(digitalRead(pin:=ECHOPIN) == HIGHVAL) {
             endTime := micros() ;
30         }
           if(startTime <> -1 && endTime <> -1) {
             INT32 travelTime :=  endTime - startTime;
             pdist := travelTime / 58 ;
           }
35         else {
             pdist := -1;
           }
       }
     }
40 }
```

Listing E.7: Mapping of the Port Instance pdistC1 to Software Platform API Commands using the APIMappingMl

F.3 Generating C Code and DDS Artifacts from MechatronicUML and Collecting Execution Time Data

In the first task, we start the "Export MechatronicUML Source Code" wizard on the MechatronicUML model file of the modeled case. We select the root component instance configuration and create a new target location named TypeCode. Afterward, we select the code generator "Component Type ANSI C99" and finish the wizard. This starts the platform-independent component type code generation of all component types that are instantiated within the component instance configuration. The code generation execution time is printed in the "Eclipse Error Log" with the ID org.muml.codegen.componenttype.c.ui.

In the second task, we open the wizard "Export Container Model and DDS Model" and select the calculated system allocation. Furthermore, we create and select the target location "Container" and finish the wizard. As a result, the model files "MUML_Container.muml_container" and "MUML_DDS.opendds" are stored in the Container folder. We start the container code generation by right-clicking the container file and selecting the item "Generate Component Container Code" from the MechatronicUML context menu. As a result, a new folder named src-gen that contains for each of the four ECUs a folder is created: ECUYellow, ECURed, ECURedDistance, and ECUSectionControl. Each of the ECU folders contains the folder component_container with the contained, generated C code that represents the container structure and its behavior. Furthermore, it contains the folder container_lib that contains necessary, copied library files. Additionally, the ECU folder contains a header file named ECU_Identifier.h with unique identifiers defined for each used message instance and component instance. Additionally, the ECU folder contains a generated main.c source file and a generated makefile. The code generation execution times for the container model creation and the OpenDDS model creation is printed in the "Eclipse Error Log" with the ID org.muml.psm.container.transformation.

Up to now, we have the platform-independent type code and the platform-specific container code. The container code depends on the DDS middleware provided by RTI [RTIa]. Therefore, we generate in the third task the required DDS code. DDS requires an implementation of specific DDS data types and handling code. They provide a code generator that generates this code on the basis of a standardized IDL file. We create the IDL file by using the 3rd-party new wizard of OpenDDS [OPENDDS] creating a new OpenDDS code generator and executing its "Generate IDL" action. Finally in the third task, we generate DDS code on the basis of the IDL file using the rtiddsgen tool of RTI [RTIa]. As a result, we get the files MUML_DDS.c/.h, MUML_DDSPlugin.c/.h, MUML_DDSSupport.c/.h. We do not measure the execution time for the describe transformations in this section as we do not have implemented them by ourselves.

In the fourth task, we generate the code for accessing the distance sensor. We start this code generation by right-clicking the apimapping file and selecting the item "Generate MechatronicUML API Mapping C" from the MechatronicUML context menu. The generated code is stored to the folder with the name APImappings. The code generation execution time is printed in the "Eclipse Error Log" with the ID org.muml.psm.api.apimappinglanguage.c.ui.

F.4 Software Construction Transformation

Table F.9 shows the results of the time measurement while transforming the case of the software construction case study (cf. Section 3.6). We perform ten transformation runs. The first run is always the outlier because Acceleo make some technical optimizations during the first run, which takes significant time. Acceleo also caches some results, which improves the performance after the first run.

Table F.9: Deployment Transformation: Execution Time Measurement Results

Measurement	Component Code	Deployment Model	OPENDDS Model	Container Code	Mapping Code
1st	2691 ms	186 ms	296 ms	2348 ms	1594 ms
2nd	1880 ms	67 ms	208 ms	1761 ms	1217 ms
3th	1659 ms	43 ms	211 ms	1240 ms	952 ms
4th	1415 ms	34 ms	159 ms	1250 ms	840 ms
5th	1367 ms	28 ms	142 ms	1183 ms	668 ms
6th	1345 ms	47 ms	133 ms	1093 ms	802 ms
7th	1297 ms	54 ms	163 ms	1049 ms	620 ms
8th	1316 ms	45 ms	125 ms	1150 ms	591 ms
9th	1284 ms	25 ms	155 ms	862 ms	712 ms
10th	1273 ms	25 ms	147 ms	957 ms	565 ms

Table F.10shows the measurement results for model aspects that are specified during the model creation of the case study in Section 5.5 but which are not considered in the effort evaluation. Row 1 shows the size of the allocation specification. Row 2 shows the size of the manually implemented software libraries. The size is zero because we do not use a software library within the case. Row 3 shows the size of the software platform API. Row 4 shows the size of the DDS specific IDL file and of the generated DDS code from this IDL file. We use the C code generator that is provided by RTI [RTIa]. Row 5 shows the size of the hardware resource model and row 6 shows the size of the hardware platform model.

Table F.10: Supplementary Measures for Comparing Effort to Develop the Case Study

Development Aspect	MechatronicUML Model				C Code Implementation			
	Manually Created		Generated		Library		Generated	
	Files	LOC	Files	LOC	Files	LOC	Files	LOC
Allocation	1	45	0	0	0	0	0	0
Software Library	0	0	0	0	0	0	0	0
Software Platform API	1	11	0	0	-	-	0	0
DDS Middleware IDL	0	0	1	58	-	-	6	12277
Hardware Resource	1	35	0	0	0	0	0	0
Hardware Platform	1	97	0	0	0	0	0	0