Verification and Simulation of Self-Adaptive Mechatronic Systems

by

Christian Heinzemann

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Zukunftsmeile 1
33102 Paderborn

Verification and Simulation of Self-Adaptive Mechatronic Systems

PhD Thesis

to obtain the degree of

"Doktor der Naturwissenschaften (Dr. rer. nat.)"

by

CHRISTIAN HEINZEMANN

Referee:
Prof. Dr. Wilhelm Schäfer

Paderborn, September 2015

Abstract

Self-adaptive mechatronic systems automatically adapt their behavior to a changing environment by reconfiguring their software architecture at runtime. In particular, this includes to dynamically form systems of systems at runtime, where several systems collaborate with each other using message-based communication protocols. Often, these systems are safety-critical and need to satisfy hard real-time constraints, i.e., any (timing) error in their behavior may put lives at risk. As a consequence, the software of a mechatronic system needs to meet high quality standards. In particular, it needs to be guaranteed that reconfigurations of the software architecture do not lead to an unsafe behavior or a violation of the real-time constraints. Testing alone cannot prove the correctness and thereby the safety of the mechatronic system. Existing approaches for model-driven development and analysis of mechatronic systems either provide support for analyzing real-time constraints or for analyzing reconfigurations of the software architecture, but none of the existing approaches supports both.

In this thesis, we present a combination of constructive and analytical techniques that can be used by software engineers as part of a model-driven software engineering method for assuring the correctness of the software of a self-adaptive mechatronic system. As a key novelty, our approach combines formal verification and simulation-based testing for achieving a scalable analysis of the system's software. As a basis, our component-based software architecture explicitly separates discrete event-based software components from time-continuous feedback controllers. This enables to verify the software components using a compositional model checking approach that we extended by a refinement check for message-based communication protocols. The correct integration of software components and feedback controllers is assessed by a testing-based approach based on model-in-the-loop simulation. Finally, we define an approach for specifying and verifying the reconfiguration behavior of software components that, in particular, separates the reconfiguration behavior from the functional behavior for improving scalability of the verification.

We evaluated all of our contributions based on the RailCab system. In particular, we specified a component-based software architecture including reconfigurations for a RailCab and conducted two case studies. These case studies demonstrate the viability of our techniques.

Zusammenfassung

Selbstadaptive mechatronische Systeme passen ihr Verhalten über die Rekonfiguration ihrer Softwarearchitektur zur Laufzeit automatisch an eine sich verändernde Umwelt an. Dies ermöglicht insbesondere die Bildung von sogenannten „Systems-of-Systems“ zur Laufzeit, in denen mehrere eigenständige Systeme unter Verwendung nachrichtenbasierter Kommunikationsprotokolle miteinander kollaborieren. Dabei müssen die einzelnen Systeme in der Regel harten Echtzeitanforderungen genügen und sind häufig sicherheitskritisch, d.h. jegliche Fehler im funktionalen oder zeitlichen Verhalten können Menschenleben gefährden. Nicht zuletzt deshalb muss die Software eines komplexen mechatronischen Systems hohen Qualitätsstandards genügen. Die besondere Kritikalität dieser Systeme bedingt, dass eine Rekonfiguration der Softwarearchitektur nicht zu einem undefinierten bzw. gefährdenden Verhalten oder einer Verletzung der Echtzeitanforderungen führt. Durch die Anwendung testbasierter Verfahren alleine kann die Korrektheit und damit auch die Sicherheit des mechatronischen Systems nicht garantiert werden. Existierende Ansätze für eine modellgetriebene Entwicklung und Analyse mechatronischer Systeme ermöglichen entweder die Analyse von Echtzeitanforderungen oder die Analyse von Rekonfigurationen der Softwarearchitektur zur Laufzeit. Bisher existiert jedoch kein Ansatz der beides unterstützt.

Im Rahmen dieser Arbeit wird eine Kombination aus konstruktiven und analytischen Verfahren vorgestellt. Sie kann von Softwareentwicklern im Rahmen einer modellgetriebenen Softwareentwicklungsmethode eingesetzt werden, um die Korrektheit der Software eines selbstadaptiven mechatronischen Systems zu verifizieren. Die Neuartigkeit des vorgestellten Konzepts liegt in der gezielten Kombination formaler Verifikationsverfahren mit simulationsbasierten Testverfahren mit dem Ziel, einen skalierbaren Ansatz für die Analyse der Software eines mechatronischen Systems zu erhalten. Als Grundlage für diesen Ansatz wird ein Komponentenmodell vorgestellt, das explizit zwischen ereignisdiskreten Softwarekomponenten und zeitkontinuierlichen Reglern unterscheidet. Es erlaubt die formale Verifikation der Softwarekomponenten mit einem kompositionalen Model Checking Verfahren, das um eine Verfeinerungsüberprüfung für nachrichtenbasierte Kommunikationsprotokolle erweitert wurde. Die fehlerfreie Integration von Softwarekomponenten und Reglern wird anschließend mit einem Testverfahren unter Verwendung von Model-in-the-Loop Simulationen überprüft. Ergänzend wird ein Konzept für die Spezifikation und Verifikation von Rekonfigurationen vorgestellt. Dieser Ansatz trennt explizit die Spezifikation und Analyse des funktionalen Verhaltens vom Rekonfigurationsverhalten, um die Skalierbarkeit der Verifikation zu verbessern.

Alle Beiträge dieser Arbeit wurden auf Basis des RailCab Systems evaluiert. Dazu wurde eine komponentenbasierte Softwarearchitektur für das RailCab inklusive der notwendigen Rekonfigurationen entwickelt. Weiterhin wurden zwei Fallstudien durchgeführt, die die praktische Anwendbarkeit der vorgestellten Verfahren aufzeigen.

Acknowledgements

Writing this PhD thesis has been an intense experience. Throughout all the time that I spend for researching and writing, I was surrounded by many people that supported me in various ways. Without these people, this work would not have been possible.

First of all, I want to thank my supervisor Prof. Dr. Wilhelm Schäfer for giving me the opportunity to work in the inspiring environment of his research group and for the scientific guidance along the way. I thank Prof. Dr. Betty H.C. Cheng and Prof. Dr. Steffen Becker for writing their reports and Prof. Dr. Falko Dressler, Dr. Matthias Meyer, Prof. Dr. Ansgar Trächtler for attending my exam.

I thank Prof. Dr. Steffen Becker for the many inspiring discussions and a lot of valuable advice. This thesis would not be the same without you. Further more, I would like to thank all of my (former) colleagues at the software engineering group and the Fraunhofer project group mechatronic systems design for the scientific discussions, but also for the enjoyable social activities that made my time at Paderborn a very pleasant and valuable experience. These colleagues are Anas Anis, Christian Brenner, Christopher Brink, Nicola Danielzik, Tobias Eckardt, Markus Fockel, Jens Frieben, Christopher Gerking, Faezeh Ghassemi, Dr. Joel Greenyer, Dr. Stefan Henkler, Jörg Holtmann, Thorsten Koch, Sebastian Lehrig, Renate Löffler, Ahmet Mehic, Sven Merschjohann, Dr. Jan Meyer, Faruk Pasic, Marie Christin Platenius, Uwe Pohlmann, Dr. Jan Rieke, David Schmelter, Christian Stritzke, Julian Suck, Oliver Sudmann, Dr. Matthias Tichy, Dietrich Travkin, Dr. Markus von Detten, Benedict Wohlers, Jinying Yu.

Special thanks go to three of my colleagues. First, to Matthias Becker for being a very valuable office mate and for the many, not always scientific discussions. Second, to Dr. Claudia Priesterjahn for the excellent cooperation in many scientific topics and a lot of valuable advice along the way. Finally, to Stefan Dziwok for the many discussions and the excellent cooperation in the development of MECHATRONICUML and the MECHATRONICUML Tool Suite.

In addition I thank all of the people with whom I collaborated in the SFB 614, in the RailCab project, and in the E-Mobil project for providing me with the opportunity to work in a very interesting, interdisciplinary environment. I highly appreciate our discussions and the excellent interdisciplinary collaboration.

Special thanks go to Jutta Haupt and Jürgen Maniera of the software engineering group, Daniela Peine, Meike Steffen, and Andreas Knoke of the Fraunhofer project group as well as Astrid Canisius and Eckhard Steffen of the International Graduate School "Dynamic Intelligent Systems" for their excellent administrative and technical support. You always knew what to do.

Moreover, this work would not have been possible without the students that contributed to the ideas and their implementation as part of their Bachelor's and Master's theses as well as their jobs as student workers. In particular, I would like to thank David Schubert and Andreas

Volk who have been my student workers for several years. It was always fun working with you.

I thank Matthias Becker, Stefan Dziwok, Marie Christin Platenius, Claudia Priesterjahn, and Aindrila Basak for their proof-reading.

Finally, I would like to thank my family and friends for always being there. Especially, I thank my parents for always supporting me and for giving me the opportunity of starting a scientific career. Foremost, I thank my beloved life companion Nicola Buth for her extraordinary support, for cheering up my bad moods when necessary, and for enduring all the time that I was "away" writing this thesis.

Contents

1 Introduction

Today's technical systems mostly consist of mechanical, electrical, and software parts. Examples of such systems include modern cars, trains, or airplanes. We call those systems mechatronic systems [VDI04, GKP08]. New functionality in such systems is increasingly realized by embedded software. In particular, embedded software interconnects previously isolated software parts of a system [PBKS07, SW07]. In addition, embedded software enables to build systems of systems where several systems collaborate with each other using application-level communication protocols [WA13]. An example of such systems of systems is given by car platoons. In a car platoon, cars drive closely behind each other for reducing the energy consumption and increasing the throughput on a highway [RCC10, HESV91].

Realizing advanced functionality such as car platooning often requires an adaptation of the software architecture at runtime [CdLG+09]. Such adaptation is called structural reconfiguration [OMT98]. As an example, cars need to adapt their software architecture for driving in a platoon. Followers, for example, need to take the distance to the preceding car into account. The leading car needs to manage communication links to a varying number of followers because cars may join or leave the platoon at any time.

Despite the fact that the software architecture and communication links may change during runtime, self-adaptive mechatronic systems need to be safe [ISO10, p. 316]. Especially systems like cars or public transportation systems require high-quality software because any software failure in such a system may put lives at risk. In particular, reconfigurations can put a system into an unsafe state if they are executed incorrectly or only partially.

Ensuring high quality of the software is further complicated by hard real-time constraints that apply to such systems [But05]. That means that the correctness of the software does not only depend on the implemented functionality but also on the correct timing of the executed operations. That holds, in particular, for the interaction of systems by communication protocols. In our example, a braking maneuver of a car platoon requires that the platoon leader notifies all followers to brake at the correct point in time.

A common approach for achieving the necessary quality and mastering the inherent complexity of such software is model-driven development [Sch06, SV06]. When applying model-driven development, the developers build models of the software instead of implementing it directly. If these models have a defined (formal) semantics, they enable to build the software correct by construction [Cha06], i.e., models may be analyzed in order to find errors already at design time. In particular, such models enable to apply analysis techniques like model checking [BK08] and simulation [ÅEM98] for guaranteeing correctness of the software. However, current approaches for the model-driven development of mechatronic systems provide no or only very limited support for adapting the software architecture of a system at runtime and for reasoning over the adaptation at design time.

Existing standards like UML [Gro11c] or SysML [Gro10] support modeling the software of a mechatronic system, but neither provide support for specifying real-time constraints and runtime reconfiguration nor define a formal semantics enabling to analyze these mod-

els. Formal models like timed automata [AD94, BY04] that are specifically dedicated to the formal analysis of real-time systems also fail with respect to specifying runtime reconfiguration. The same restriction holds for commercial tools like MATLAB/Simulink [Matg] or Dymola/Modelica [Das, Mod09] that are used in industry for developing automotive software [PBKS07, KSHL12]. In contrast, graph-based approaches like graph transformation systems [Roz97] enable the specification of runtime reconfiguration but provide no means for specifying their real-time properties. Component models for real-time systems either provide limited support for self-adaptation behavior or they support formal verification [CSVC11, HPB+10]. Architecture description languages for self-adaptive systems [BCDW04] do not consider the real-time properties that apply to mechatronic systems. Hence, none of the approaches provides the necessary modeling and analysis capabilities that are required for self-adaptive mechatronic systems.

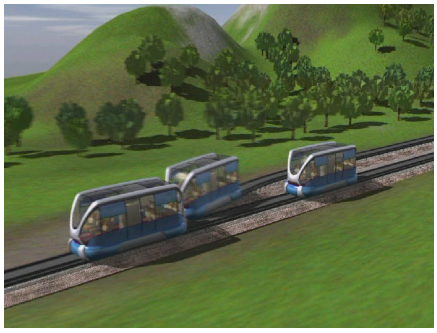
The goal of this thesis is to extend a model-driven software engineering method by techniques for guaranteeing the correctness of the software of a self-adaptive mechatronic system. In this thesis, we will use the model-driven software engineering method MECHATRONICUML [GTB+03, BDG+14a] as a basis for developing and illustrating our contributions. MECHATRONICUML adapts concepts of the UML [Gro11c] for supporting the model-driven development of self-adaptive mechatronic systems. In particular, it provides a domain-specific modeling language with a formal semantics that enables formal verification of software models. Previous works on MECHATRONICUML contributed, for example, the specification of feedback controller exchange [BGO06, Hir08, OMT+08], the specification of software reconfiguration [THHO08, Tic09], and the verification of communication protocols [GTB+03, EHH+13]. As a result, MECHATRONICUML supports the specification of a software architecture including state-based real-time behavior and reconfiguration operations for self-adaptive mechatronic systems. In this thesis, we extend the support of MECHATRONICUML for specifying and analyzing reconfigurations and message-based communication protocols that are necessary for realizing self-adaptive mechatronic systems. As an application example, we use the RailCab system that is introduced in the next section.

1.1 The RailCab System

The RailCab system [HTS+08a, NBP] is one representative example of a whole class of self-adaptive mechatronic systems that applies runtime reconfiguration for adapting their software to their changing environment [HSD+15]. The vision of the RailCab system is a new kind of railway transportation system where autonomous vehicles, the *RailCabs*, transport people and goods directly to their destination without the need for changing trains. RailCabs drive autonomously, only controlled by software using the existing track systems. Figure 1.1(a) shows a RailCab prototype in scale 1:2.5 on the test track at the University of Paderborn.

One feature of the RailCab system is the convoy mode which is similar to the aforementioned car platoons. In convoy mode, two or more RailCabs agree on driving behind one another at small distances in order to reduce the energy consumption [HTS+08a, HP14]. Figure 1.1(b) shows an illustration of the build-up of a convoy of three RailCabs. Joining a convoy requires the RailCabs to adapt their software, e.g., for handling the necessary communication inside the convoy.

(a) RailCab Prototype in Scale 1:2.5 on Test Track



(b) Build-up of a Convoy of Three RailCabs at a Switch [NBP]

Figure 1.1: The RailCab System

In addition, building a convoy includes electing a so-called *coordinator* [HTS+08a, HP14]. The coordinator is responsible for providing reference data and announcing acceleration and braking maneuvers to all other RailCabs in the convoy, which we call *members*. The necessary communication between the RailCabs is formally defined by a message-based communication protocol called ConvoyCoordination. This communication protocol is safety-critical because collisions are inevitable if RailCabs are not notified correctly about acceleration or braking maneuvers.

1.2 Problem Statement

The software of a self-adaptive mechatronic system like the RailCab is complex, i.e., it consists of a large number of concurrent, interconnected functions. A common approach for building such software is a component-based approach where the software architecture is specified by hierarchical, interconnected components [SGM02]. MECHATRONICUML follows this approach as well. Components interact via message-based communication protocols. In MECHATRONICUML, a reconfiguration of the system is specified by a modification of the software architecture, i.e., adding and removing component instances and connectors. Communication protocols and reconfigurations are formally verified for safety properties using model checking [GTB+03, GS13] and inductive analyses [BBG+06]. In this section, we outline three particular problems regarding the specification and analysis of models including runtime reconfiguration that are currently not sufficiently solved by MECHATRONICUML and other related approaches.

1. Reconfiguration of Mechatronic Systems In a component-based system, a reconfiguration often affects several components of the software architecture. As an example, consider two RailCabs that reconfigure their software architecture for building a convoy as shown in Figure 1.2. Before building the convoy, both RailCabs only execute their a VelocityController that controls their speed as shown in the upper part of Figure 1.2. For driving in the convoy, the member RailCab needs to instantiate the software component MemberControl that implements the communication protocols for driving in a convoy. In addition, this

RailCab needs to replace its VelocityController by a DistanceController that takes the distance to the preceding RailCab into account as shown in the lower part of Figure 1.2. The coordinator only needs to instantiate the software component ConvoyCoordination that implements the communication protocols for communicating with the convoy members.

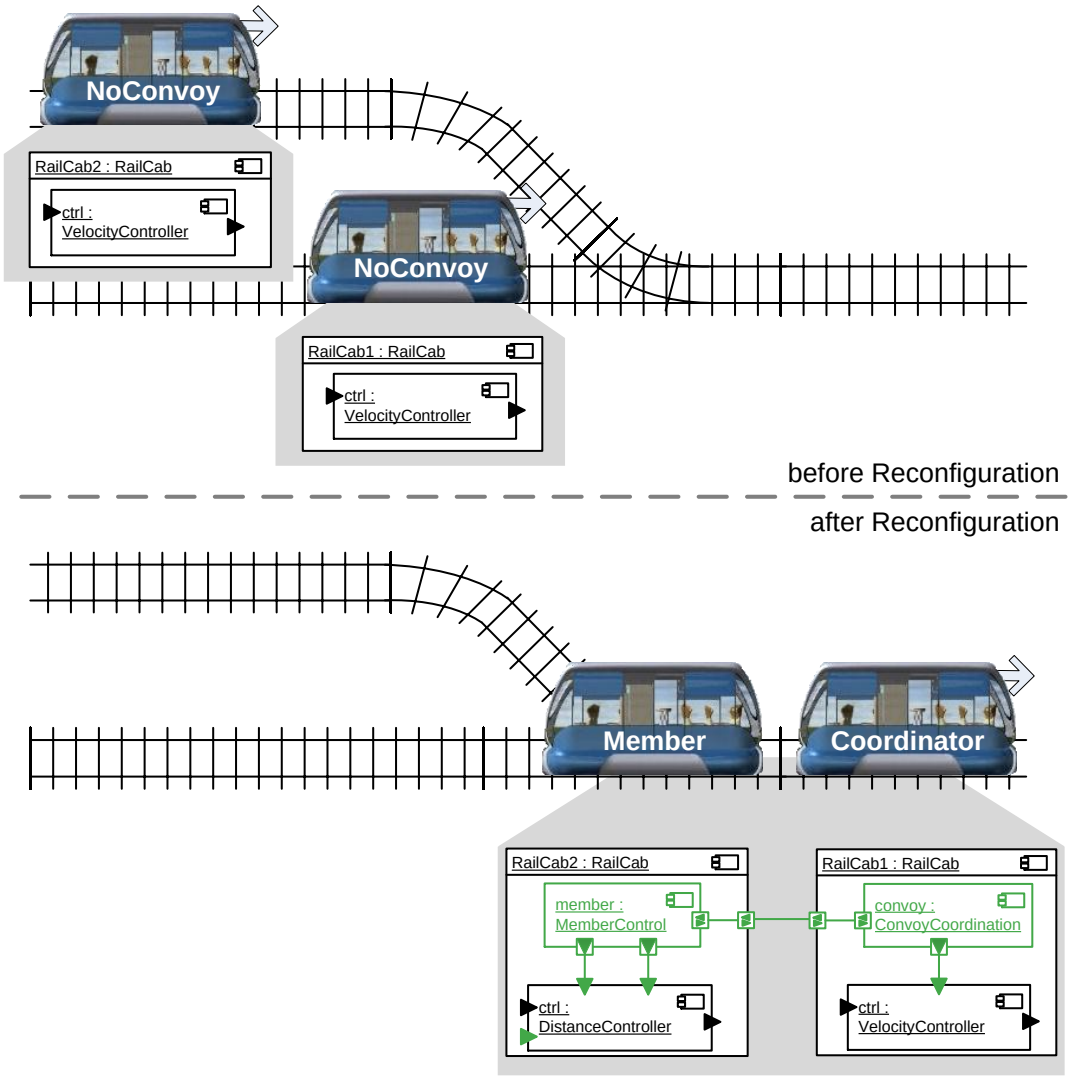


Figure 1.2: Illustration of the Software Reconfiguration of RailCabs for Building a Convoy

Thus, the reconfiguration for becoming a convoy member affects the RailCab software component as well as the VelocityController. In such cases, all of the affected components need to reconfigure correctly such that the intended result can be established. If reconfigurations are only executed partially, the system may become unsafe. If the RailCab only instantiates the MemberControl component but does not switch the feedback controller, the system is unsafe. In this case, the RailCab does not consider the distance to the preceding RailCab while driving in a convoy. If the RailCab only switches the feedback controller but does not instantiate the MemberControl component, the system is unsafe as well. In this case, the feedback

controller does not receive the reference speed for driving in the convoy and, as a result, the RailCab may drive too fast and cause a collision.

To be safe, a reconfiguration approach needs to ensure, on the one hand, that the reconfigurations can never produce an inconsistent software architecture as, e.g., executing the DistanceController without executing the MemberControl. On the other hand, the reconfiguration approach must ensure that a reconfiguration can be executed completely in time before actually starting it. That, in turn, requires to take the real-time constraints of the system into account while deciding whether a reconfiguration should be executed or not, and to verify that no reconfiguration violates the real-time constraints. Since the component model of a self-adaptive mechatronic system is hierarchical in most cases, the reconfiguration approach needs to support reconfiguration across different levels of hierarchy while preserving the encapsulation of components [SGM02]. At present, no existing approach for specifying reconfiguration of component-based systems considers all of the aforementioned properties.

2. Refining Communication Protocols In a self-adaptive mechatronic system, whose software is implemented in a component-based fashion, communication between the components is essential for realizing the functionality of the system [SW07]. This includes both, the communication between components inside a single system but also the communication between systems as part of a system of systems [WA13]. As a result, the correctness of the software of a self-adaptive mechatronic system does not only depend on the correctness of a single component but also on the correctness of the application-level communication protocols that define the interaction between components and between different system.

Due to the safety critical nature of self-adaptive mechatronic systems, it is desirable to apply formal verification methods like model checking [CGP00, BK08] to ensure correctness of their component-based software. Model checking gives a mathematical proof that safety and liveness properties, which have been specified for the system, hold. However, formal verification techniques like model checking suffer from the so-called state-explosion problem [CGP00]. It denotes the fact that the number of runtime states of a software grows exponentially in both, the number processes and the number of states of each process, if the system has concurrent executions [CKNZ12]. This makes the verification of component-based systems with concurrent components quickly infeasible. Compositional verification approaches [BCC98] tackle the state-explosion problem by verifying single components of a component-based system in isolation. Many of these approaches are based on the assume/guarantee principle [CGP00, ch. 12], i.e., they verify the correctness of a component based on assumptions that must be guaranteed by the component's environment. One of the main difficulties of assume/guarantee approaches is deriving good assumptions automatically from the software model [CAC08].

One example of a compositional verification approach based on the assume/guarantee principle is given by the compositional verification approach of MECHATRONICUML [GTB+03, GS13]. This approach prevents the computation of assumptions by providing a syntactic decomposition of the system. In particular, MECHATRONICUML separately defines components and communication protocols that define the interaction of components. Then, components may be verified under the assumption that the interaction via the communication protocol is correct. Guaranteeing this assumption requires two steps. First, we need to verify the communication protocol using model checking [EHH+13]. At this point, the assume/guar-

antee principle requires that the protocol is independent of the component. Second, we need to guarantee that the component correctly implements the communication protocol without invalidating the verification results obtained for the communication protocol in the first step.

However, implementing the communication protocol in the component requires to modify it. In particular, we need to integrate the protocol with the internal behavior of the component, for example, for accessing data and triggering computations. As a result, we need to verify that the component implementation of the communication protocol is a correct refinement of the verified protocol behavior according to a refinement definition. The refinement definition guarantees that the component implementation of a communication protocol does not invalidate the verification result that has been obtained for the communication protocol. In particular, a refinement definition formally defines how the component implementation (refined protocol) may deviate from the communication protocol (abstract protocol) without invalidating a particular set of verified properties. Thus, a suitable definition of refinement is essential for a compositional verification approach as the one used by MECHATRONICUML.

In literature, many different refinement definitions and according verification procedures exist [BK08, WL97, JLS00]. "Examples include timed simulation and timed bisimulation [WL97]. Depending on the particular type of protocol that is refined, all refinement definitions might be useful when building a system. A *suitable* refinement definition for a compositional approach needs to be as weak as possible for enabling reuse of an abstract protocol in as many different components as possible but as strong as necessary for guaranteeing that all verified properties hold for the refined protocol. If the refinement definition is too weak, it is not guaranteed that verified properties still hold for the refined protocol. If the refinement definition is too strong, the refinement check might reject the refined protocol although it fulfills all properties. This may happen, for example, if the refined protocol removes behavior that is irrelevant for the properties, but which is checked by the too strong refinement definition. The existing refinement definitions provide different compromises between reuse and preserved properties. As a consequence, there does not exist one refinement definition that is suitable for all possible protocols. Instead, a compositional verification approach should support several refinement definitions where each of which may be suitable for a particular abstract and refined protocol and a set of verified properties." [HBDS15]

As a result, we need an integrated approach that automatically selects a suitable refinement definition and, in particular, verifies it for a given pair of abstract and refined protocols.

3. Simulation of Self-adaptive Mechatronic Systems The safe and correct operation of a mechatronic system depends on the correct integration of the time-continuous feedback controllers and the discrete software components. This includes, in particular, reconfiguration of the software architecture at runtime. As discussed before, reconfiguration of the software architecture of a mechatronic system may require to exchange feedback controllers. Exchanging feedback controllers involves the specification and execution of potentially complex fading functions [BGO06, OMT$^+$08] that guarantee that safe meaningful values are applied on the physical machine at any time during the exchange. Therefore, it is absolutely mandatory to ensure correctness of the reconfigurations.

A major objective of MECHATRONICUML is to prove the correctness of such system models by applying formal verification. However, the integration of time-continuous feedback controllers whose behavior is defined by differential equations hardens formal verifi-

cation significantly. In literature, it is often referred to as the hybrid model checking problem [Hen96]. Hybrid model checking approaches either only use very simple models of time-continuous behavior or they apply overapproximation techniques [HHMWT00]. "A primary reason for adopting overapproximation is that a precise model, or a practical engineering model at hand, incorporates elements that no verification tool can handle in combination. This is often the case for hybrid system models due to their rich vocabulary. Analysis of such models can only commence after a chain of approximation steps, some of which can be achieved automatically, others – the majority in practice – requiring manual reformulation of the model under inspection. Each of these approximations may cause a loss of precision in the model, e.g., when capturing nonlinear behavior by a linear model, making the analysis less likely to succeed with a positive certificate as outcome. At the same time, as these approximations often have to be done manually, they require extremely skilled staff, are tedious and have to be repeated when the original model changes." [ERNF12] In addition, even the most recent techniques can only handle models that are "still of academic nature in the size of problems solvable." [ERNF12]. As a result, it is not yet possible to verify correctness for large and complex reconfigurable mechatronic systems such as the RailCab.

A different approach to assess the correctness of the operations of a mechatronic system is testing by using a model-in-the-loop (MIL) simulation [Plu06]. In a MIL simulation, the developer tests a model of the mechatronic system against a model of its environment. The model of the mechatronic system always includes the feedback controllers and the discrete software components, but it may also include models of the mechanic, electric, or hydraulic parts of the system. This approach is already used in the automotive industry [BB08, SHS12].

MIL simulation of mechatronic systems is supported by commercial-of-the-shelf simulators such as MATLAB®/Simulink® [Matg] or Dymola [Das] for the specification language Modelica [Fri04, Mod09]. These tools, however, require models to be static, i.e., once specified, components and connections may not change while running a simulation. In addition, they do not provide native support for asynchronous, message-based communication with message buffers.

As a result, we need an approach that supports MIL simulation of self-adaptive mechatronic systems that communicate via asynchronous, message-based communication protocols. This approach needs to be integrated into our component-based development approach such that model checking the event-discrete software components using the compositional verification approach mentioned above remains possible.

1.3 Contribution

The contribution of this thesis is a combination of constructive and analytical techniques that support the component-based specification and analysis of self-adaptive mechatronic systems as part of a model-driven approach. As a key novelty compared to related approaches, we combine formal verification and simulation-based testing for achieving a scalable analysis for ensuring correctness of the software of a self-adaptive mechatronic system. In particular, we contribute a transactional execution of hierarchical reconfigurations including an approach for their verification ($\mathbf{C_1}$), a verification procedure for showing correct refinements of communication protocols ($\mathbf{C_2}$), and support for simulating self-adaptive mechatronic systems in MATLAB/Simulink ($\mathbf{C_3}$). We integrate our contributions into the MECHATRONICUML

method. As a result, our contributions enhance the existing development process of MECHATRONICUML [HSST13, BDG+14b] as outlined in Figure 1.3. All of our contributions have been implemented as part of the MECHATRONICUML Tool Suite [DGB+14].

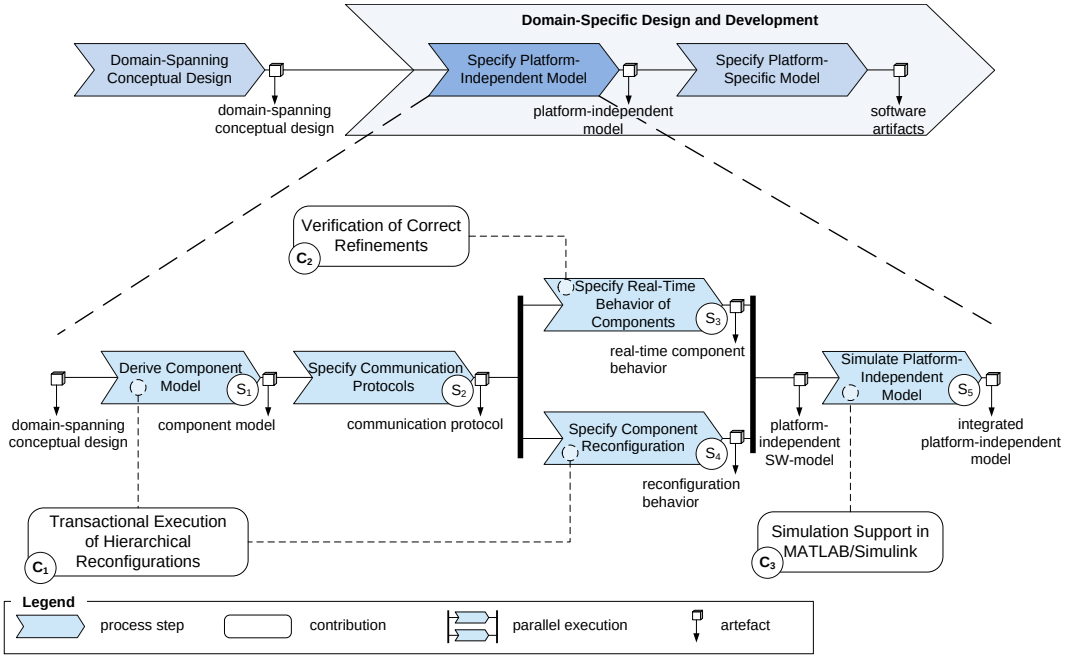


Figure 1.3: Excerpt of the Design Process for the Development of Self-Adaptive Mechatronic Systems (cf. [HSST13, GV14])

The starting point for the process, shown in Figure 1.3, is the domain-spanning conceptual design [GFDK09, GSG+09] that has been created collaboratively by experts from all disciplines involved in building the mechatronic system, e.g., mechanical engineering, control engineering, and software engineering. It includes all information about use cases, functions, and system elements that affect more than one discipline. Based on the domain-spanning conceptual design, each of the involved disciplines starts the domain-specific design and development phase. In this phase, the software engineers execute the MECHATRONICUML process [HSST13, BDG+14b], which consists of two main phases in accordance to the model-driven architecture approach [Gro14]. Thus, the process starts by creating a platform-independent model of the software. Then, the software engineers derive a platform-specific model of the software and define a deployment of the software to the hardware platform. The contributions of this thesis address the specification of the platform-independent model.

The software engineer starts specifying the platform-independent model in **Step S_1** by deriving an initial component model from the domain-spanning conceptual design. In this thesis, we unify the existing component models of MECHATRONICUML and provide an extension that enables a concise, declarative specification of hierarchical reconfigurations. This specification forms the basis for a transactional execution of reconfigurations (**C_1**) that respects ACI-properties of database systems [BHG87]. These are *atomicity*, i.e., either all component instances reconfigure or none does, *consistency*, i.e., each reconfiguration pro-

duces a consistent component instance configuration, and *isolation*, i.e., reconfigurations do not interfere with each other.

In **Step S_2**, the software engineer specifies a communication protocol for each interaction between components. This includes a formal verification of the protocol behavior using model checking [GTB+03, EHH+13, Ger13].

After specifying the communication protocols, the software engineer needs to specify the real-time behavior for each component of the component model. This real-time behavior needs to include the communication protocols that have been specified and verified in Step S_2 such that the verified safety and liveness properties are not invalidated. We support the software engineer in this step by an integrated verification procedure that verifies whether the real-time behavior of a component correctly refines a communication protocol according to a formal refinement definition (**C_2**). As a byproduct, our approach automatically selects a suitable refinement definition out of a set of possible refinement definitions.

In **Step S_4**, the software engineer specifies the reconfiguration behavior of the components using our aforementioned extensions of the component model. In addition, we extend this step by an approach for verifying that the reconfiguration behavior fulfills the required ACI-properties and meets all hard real-time deadlines (**C_1**). The result of Steps S_3 and S_4 is a platform-independent model of the software.

Finally, the software engineer needs to analyze whether event-discrete software and time-continuous feedback controllers have been integrated correctly by using a MIL simulation in **Step S_5**. We support the software engineer in this step by automatically deriving a simulation model that includes both, the real-time behavior and the reconfiguration behavior of the components. The simulation model is then extended by the implementations of the feedback controllers and the environment model. The MIL simulation may then be carried out using MATLAB/Simulink. It enables the engineers of the different disciplines to validate the whole self-adaptive mechatronic system by simulation and enables to use the code generation facilities of MATLAB/Simulink for deriving source code for the system.

1.4 Overview

The remainder of this thesis is structured as follows. Chapter 2 introduces the foundations that are required for understanding the contributions of this thesis. In Chapter 3, we define a new component model for MECHATRONICUML. The MECHATRONICUML component model forms the basis for the remaining contributions of this thesis. Along with the component model, we continue our RailCab example from Section 1.1. We use this example throughout the remainder of this thesis. Chapter 4 introduces our concept for transactional execution of reconfigurations. In addition, we explain our concept for verifying reconfigurable components for ACI and timing properties. Thereafter, Chapter 5 presents our approach for verifying that communication protocols have been correctly refined by the components in our component model. Next, we present our approach for MIL simulation of self-adaptive mechatronic systems in Chapter 6. In particular, we define how simulation models in MATLAB/Simulink can be derived automatically from a MECHATRONICUML model. Finally, we summarize the contributions of this thesis and give a perspective on future works in Chapter 7. We discuss the implementation and evaluation of our concepts as well as related works along with our contributions as part of the main chapters of this thesis.

The appendices provide additional, more technical information that supplement our contributions. First, Appendix A presents additional parts of the RailCab model that we use as a running example. Appendix B contains a formal definition of the semantics of Real-Time Statecharts that we use for defining state-based behavior. Finally, we describe our framework for performing reachability analyses (Appendix C) and the metamodels that have been created as part of this thesis (Appendix D).

2 Foundations

This chapter introduces the foundations for understanding the concepts presented in the remainder of this thesis. We start by reviewing concepts and terminology related to self-adaptive mechatronic systems in Section 2.1 that we will use in the following. Thereafter, Section 2.2 introduces timed model checking including timed automata and the timed computation tree logic. The latter two provide the formal basis for specifying and verifying state-based behavior models of a self-adaptive mechatronic system. Section 2.3 introduces graphs and corresponding graph transformations that form the basis for specifying and verifying reconfiguration operations of a self-adaptive mechatronic system. Finally, Section 2.4 introduces MECHATRONICUML, which is a domain-specific language based on timed automata and graph transformations, that enables to specify software models for a self-adaptive mechatronic system on a higher level of abstraction. We will integrate all of the contributions of this thesis into MECHATRONICUML.

2.1 Self-Adaptive Mechatronic Systems

Self-adaptive mechatronic systems automatically adapt their software architecture to a changing environment without human intervention. That requires to integrate and associate the software with the constituent parts of the mechatronic system such that the system may reason about itself and its behavior in its current environment. In this section, we introduce basic concepts and corresponding terminology related to self-adaptive mechatronic systems that we use throughout the remainder of this thesis. In Section 2.1.1, we describe how self-adaptive mechatronic systems may be structured hierarchically. Thereafter, we introduce the operator-controller-module as a reference architecture that enables to realize self-adaptive behavior in mechatronic systems (cf. Section 2.1.2). Finally, Section 2.1.3 describes how the concept of models@runtime may be used for executing reconfigurations.

2.1.1 Structuring

Self-adaptive mechatronic systems can be structured hierarchically as shown in Figure 2.1. In particular, they can be structured into mechatronic function modules, autonomous mechatronic systems, and networked mechatronic systems (cf. [GRS14, pp. 8-10]).

On the lowest level, a mechatronic system consists of several *mechatronic function modules* (MFM). An MFM embodies part of the mechanical system including sensors, actuators, and software for controlling the mechanical system. An example is given by the drive module [HZ14] or the active suspension module [KT14] of a RailCab. MFMs may, again, be composed of other MFMs.

The overall mechanical structure is represented by the *autonomous mechatronic system* (AMS). It consists of the MFMs of the mechatronic system and includes additional sensors

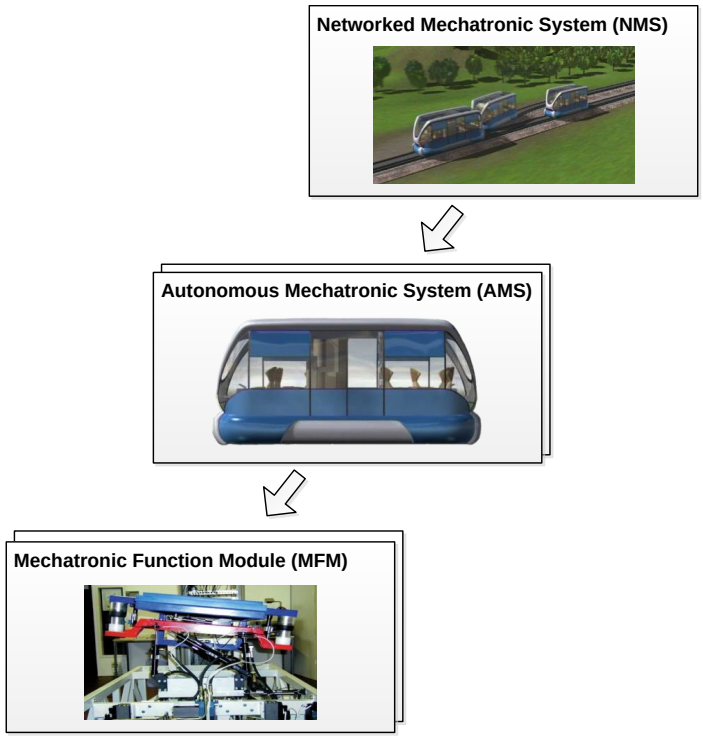


Figure 2.1: Structuring of Self-Adaptive Mechatronic Systems (cf. [GRS14, p. 9])

and software components for realizing self-adaptive behavior. An example of an AMS is a single RailCab.

Finally, AMS' may collaborate and form *networked mechatronic systems* (NMS). Networked mechatronic systems usually have no physical representation but are only virtually created by the AMS by using message-based communication protocols. Then, each AMS fulfills a particular role in the NMS. An example is given by convoys of RailCabs.

2.1.2 Operator-Controller-Module

The *operator-controller-module* (OCM) is a reference architecture for self-adaptive mechatronic systems that separates the behavior specification into three conceptual levels [HOG04, GRS09, GRS14]. As part of this thesis, we relate our contributions to these three conceptual levels of the OCM. The different levels are the cognitive operator, the reflective operator, and the controller as shown in Figure 2.2.

The *controller* level is the lowest. It contains the feedback controllers that control the physical system that is also called the controlled system [Kil05]. A feedback controller continuously receives the current value of the controlled variable from the physical sensors. Based on a reference value for the controlled variable, it tries to reduce the difference between the current value and reference value to zero by computing signals for the system's actuators.

The *reflective operator* forms the middle layer of the OCM. It contains event-discrete software that is required for the operations of the mechatronic system as, for example, the behavior for operating as a coordinator or member of a convoy. As a key element of this

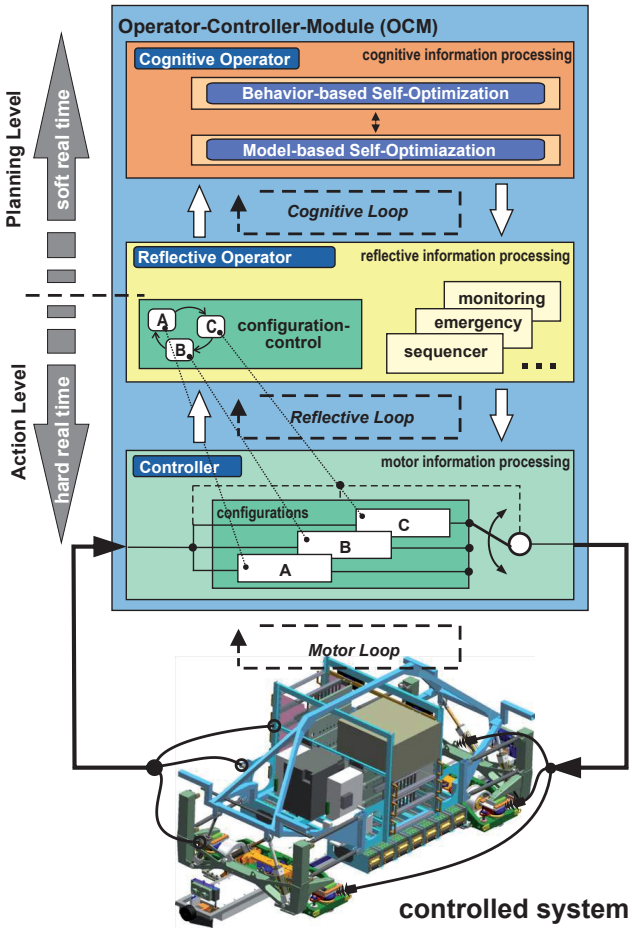


Figure 2.2: Overview of the Operator-Controller-Module [GRS14, p. 11]

behavior, the reflective operator executes communication protocols for interacting with other AMS.

In addition, the reflective operator provides the ability to reconfigure its own software architecture including the feedback controllers on the controller level. In accordance to Allen et al. [ADG98] and Zhang et al. [ZC06], we distinguish between *steady-state behavior* and *reconfiguration behavior*. The steady-state behavior defines the behavior that is executed by the feedback controllers on the controller level and by the reflective operator based on a particular software architecture without considering reconfigurations. The reconfiguration behavior defines possible modifications of the software architecture. Using the reflective loop, self-adaptive systems continuously monitor their own behavior for deciding whether, when, and how they need to reconfigure. During a reconfiguration, the system switches from one steady-state behavior to another steady-state behavior [ADG98, ZC06]. In the following, we refer to the different steady-behaviors of a self-adaptive mechatronic system as *functional behavior* [MSKC04].

The *cognitive operator* provides the self-optimization capabilities to the system. Therefore, it typically manages a set of weighted goals that the AMS shall achieve at runtime [vL01, GSB+08]. As a result, the cognitive operator contains functionality for reasoning about the system's behavior and the environment for optimizing the system's behavior according to the given goals.

The software on the controller level and part of the reflective operator are executed with respect to hard real-time constraints [Kop97]. This includes, in particular, the reconfiguration of the software being executed on these levels and the communication between different systems. The cognitive operator and the remainder of the reflective operator are executed in soft real-time [Kop97].

2.1.3 Models@Runtime

A *model@runtime* is a model of a system that it manages and uses by itself during runtime [BBF09]. In contrast to a reflective system model [Mae87], it is typically specified on a higher level of abstraction as denoted by Blair et al. [BBF09]. Both approaches, however, require that the model and the system are causally connected. That means that any change in the running system is reflected into the model@runtime and, what is even more important, any change of the model@runtime changes the running system in the same way. As a consequence, the system can be modified by modifying the model@runtime instead of modifying the running system directly. In this thesis, we use this approach for defining and executing reconfigurations based on a model@runtime of the software architecture [GS04].

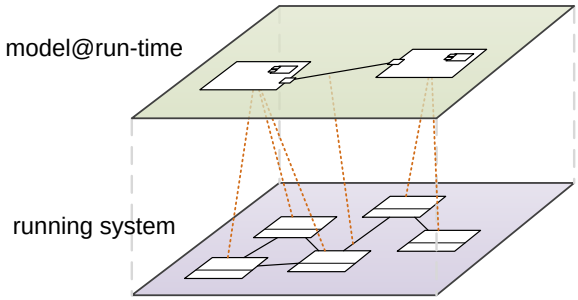


Figure 2.3: Illustration of a Model@Runtime

Figure 2.3 illustrates the principle of a model@runtime. The running system on the lower level consists of a set of objects. The model@runtime abstracts these objects to a component representation and associates the objects and connectors of the running system to the model elements of the model@runtime. Changing the component structure in the model@runtime will also change the object structure in the running system.

2.2 Timed Model Checking

Model checking [CGP00, BK08] is an automated formal verification procedure that gives a mathematical proof whether a software model fulfills a given set of formal requirements. Thus, it may guarantee the absence of errors in a model, in contrast to testing, which may only show the presence of errors. Timed model checking [ACD93, HNSY94] additionally

considers the real-time characteristics that apply to a model of a mechatronic system's software. In this thesis, we use (timed) model checking as an analytical method for showing the correctness of functional and reconfiguration behavior of a self-adaptive mechatronic system.

Timed model checking uses timed automata as introduced in Section 2.2.1 as a behavioral model. Therefore, timed automata provide the formal basis for specifying state-based real-time behavior in MECHATRONICUML (cf. Section 2.4). The timed computation tree logic as introduced in Section 2.2.2 enables the specification of formal requirements. These formal requirements express the safety and liveness properties that the timed automata need to fulfill. We require knowledge about such formal requirements for defining our refinement approach in Chapter 5. Finally, a timed model checking algorithm decides whether the timed automata fulfill the formal requirements given based on the timed computation tree logic (cf. Section 2.2.3).

2.2.1 Timed Automata

A *timed automaton* [AD94, BY04] is a state-based model for specifying real-time behavior. Timed automata extend finite automata [Mea55] by a set of real-valued variables called *clocks* and constraints over these clocks. Clocks measure the progress of time in a system. Time progresses constantly and uniformly in all clocks. In literature, there exist many variants of timed automata (see Waez et al. [WDR11] for a recent survey). In this thesis, we will use timed safety automata [HNSY94] as they are defined for the UPPAAL model checker by Bengtsson and Wang [BY04]. For the remainder of this thesis, we will refer to timed safety automata simply as timed automata.

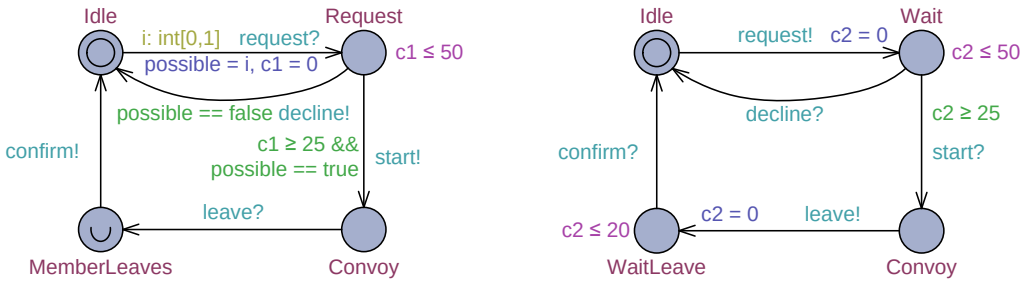
Like finite automata, timed automata have a set of inputs, a set of outputs, and a set of integer variables. Each transition may consume an input and produce an output. Integer variables may be used for guard conditions of the transitions and may be changed using assignments while the transition fires.

In addition, timed automata support three modeling elements based on clocks. These are invariants, time guards, and resets. An invariant is assigned to a location. The timed automaton may only rest in a location as long as the invariant is true for the current clock values. A time guard restricts the firing of a transition to a time interval. A reset sets a clock back to zero while a transition fires.

Timed automata can be composed to networks of timed automata (NTA). In an NTA, the single automata interact via their inputs and outputs using so-called *channels*. Then, they use the channels as inputs (marked with ?) and outputs (marked with !).

Figure 2.4 shows an NTA consisting of two timed automata that specify a simple convoy behavior. The member automaton in Figure 2.4b requests to start a convoy. The coordinator automaton in Figure 2.4a either starts the convoy or declines the request. Finally, the member automaton may choose to leave the convoy and the coordinator automaton confirms.

The two timed automata use five channels named request, start, decline, leave, and confirm for realizing the convoy behavior. Each of the timed automata has four locations and five transitions. As an example, the coordinator automaton in Figure 2.4a has locations Idle, Request, Convoy, and MemberLeaves. The transition from Idle to Request specifies a reset on clock c1, i.e., the coordinator automaton resets c1 any time it receives a request from the member automaton. The location Request has an invariant $c1 \leq 50$, i.e., the coordinator automaton needs to send an answer to the member automaton while c1 is less or equal 50.



(a) Coordinator (b) Member

Figure 2.4: Network of Timed Automata Specifying a Simple Convoy Behavior

However, the transition from Request to Convoy may only fire if c1 is greater of equal to 25 as specified by the time guard. Thus, the coordinator automaton may only start the convoy 25 time units after receiving the request.

Timed automata may be nondeterministic. In particular, they may select a value nondeterministically from a given range using selections [BDL06a]. This value may be assigned to a variable. In our example, the coordinator automaton in Figure 2.4a uses a selection at the transition from Idle to Request. It selects the value of i from an integer range from 0 to 1. The value of i is then assigned to the variable possible that defines whether it is possible to start the convoy.

The *state* of an NTA is defined by the discrete locations of all timed automata including the values of their variables and their clocks. Since clocks are real-valued and time increases continually, timed automata always have an infinite number of states. Therefore, the semantics of an NTA is usually defined by means of *symbolic states* based on clock zones [Alu99, BY04]. Clock zones store intervals of clock values and enable to represent the state space of an NTA using a finite *zone graph*. The rules for computing the zone graph define the semantics of NTAs. We refer to paths of the zone graph as *traces*. Figure 2.5 shows the zone graph of the NTA in Figure 2.4.

The execution of an NTA starts in the initial states of all timed automata (Idle in both automata in Figure 2.4) with all clocks being zero and all variables set to their initial values. The corresponding symbolic state S1 is the initial state of the zone graph in Figure 2.5. Further symbolic states and transitions in the zone graph are inferred by the following rules:

1. **Delay**
   An NTA may delay, i.e., the values of all clocks increase, but neither the active locations nor the values of the integer variables change. The values of the clocks may increase as long as no invariant of an active location restricts them. In the zone graph, these transitions are labeled with δ. As an example, consider the transition from S1 to S2. In S2, all clocks have an unbounded value greater or equal to 0 because the Idle locations do not define invariants.
2. **Single Transition**
   A timed automaton in an NTA may fire a transition that does not use a channel. In this case, the active location and, potentially, the values of the integer variables change. The active locations of all other timed automata remain unchanged. Furthermore, fir-

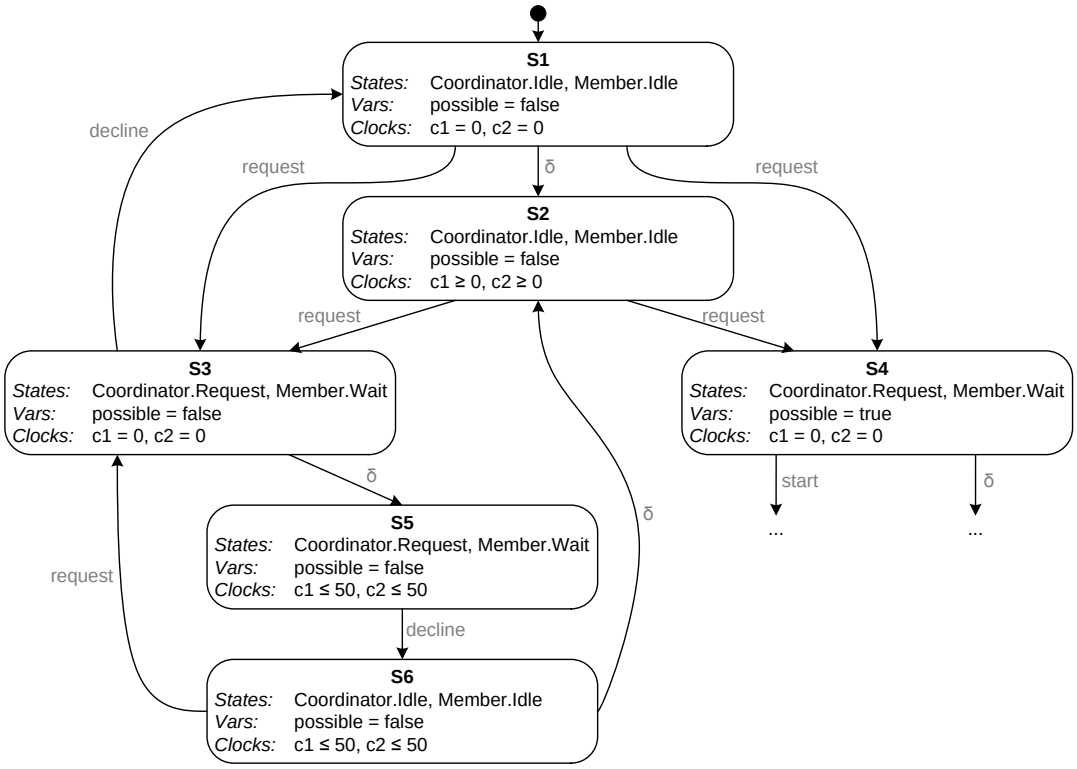


Figure 2.5: Excerpt of a Zone Graph of the NTA in Figure 2.4

ing transitions in an NTA takes no time and, thus, the values of the clocks do not change except for resets that are performed by the transition. In the zone graph, the corresponding transitions are labeled with τ.

3. **Synchronized Transitions**
   Two timed automata in an NTA may synchronize over a channel and synchronously fire one transition each. The synchronization is defined by the CCS parallel composition operator [Mil82]. As a result, synchronization is realized by hand-shake synchronization, i.e., the two timed automata move to the target locations of their transitions synchronously. For synchronizing transitions, the assignments and resets of the transition with the output (!) are executed prior to the assignments and resets of the transition with the input (?). In the zone graph, the corresponding transitions are labeled with the name of the channel. As an example, consider the transition from S2 to S3 that corresponds to a synchronization via the channel request. As a consequence, both timed automata change their active locations and, due to the resets, both clocks are set back to 0.

Transitions of a timed automaton do not need to fire if they are enabled unless they are forced to. If a location specifies an invariant, the timed automaton is forced to fire before the invariant expires. In addition, timed automata provide urgent channels as well as urgent locations for forcing immediate progress in an NTA. If two transitions that synchronize over an urgent channel are enabled, they need to fire instantly without delay. In urgent locations,

no delay is allowed as well. In the timed automaton in Figure 2.4a, MemberLeaves is an urgent location.

2.2.2 Timed Computation Tree Logic

The *timed computation tree logic* (TCTL, [ACD93]) is a timed temporal logic for real-time systems. It enables to specify formal safety and liveness properties [Lam77] for a given real-time behavior model as, e.g., an NTA. "A safety property is one which states that something will *not* happen" [Lam77], e.g., that it may never happen that one RailCab assumes to be a member of a convoy after the coordinator has declined the request. "A liveness property is one which states that something *must* happen" [Lam77], e.g., that a RailCab always answers to a proposal on building a convoy. For specifying timing properties, TCTL extends the *computation tree logic* (CTL, [CES86, HR04]) that was developed for finite-state models by quantitative timing constraints. While CTL may only specify that some condition will be true at some point in the future, TCTL may define a quantitative time bound by defining, for example, that the condition must be fulfilled within 5 ms.

TCTL uses a textual syntax. We may define the syntax of a TCTL formula ϕ inductively via a Backus Naur form

$$\phi ::= \texttt{true} \mid \texttt{false} \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \mathbf{AG}_{\sim c}\ \phi \mid \mathbf{EG}_{\sim c}\ \phi \mid \mathbf{AF}_{\sim c}\ \phi \mid \mathbf{EF}_{\sim c}\ \phi \mid \mathbf{A}(\phi_1\ \mathbf{U}_{\sim c}\ \phi_2) \mid \mathbf{E}(\phi_1\ \mathbf{U}_{\sim c}\ \phi_2)$$

where p is an element of a set AP of atomic formulas, $c \in \mathbb{N}$, and $\sim$ represents one of the binary relations $<, \leq, =, \geq, >$ (cf. [HR04, ACD93]).

The set AP of atomic propositions refers to facts of an NTA that may be evaluated to true or false for any (symbolic) state of the NTA. Considering the NTA shown in Figure 2.4, we use the atomic proposition `Coordinator.Request` to express the fact that the state Request of the coordinator automaton is active.

The temporal connectives **AG**, **EG**, **AF**, **EF**, **AU**, and **EU** define which atomic propositions need to hold in the future. As its name indicates, TCTL is a branching time logic where formulas are evaluated based on a computation tree. That means, TCTL acknowledges the fact that computations of an NTA may branch. Therefore, the temporal connectives always consist of a path quantifier (**A** or **E**) and a temporal operator (one of **G**, **F**, **U**).

Using the path quantifiers, the developer may express that a formula shall hold either for all paths of a computation tree (**A**) or for at least one path (**E**). For an NTA, the computation tree is given by its zone graph where all loops are unrolled. The temporal operator **G** defines that the subformula ϕ must hold for any (symbolic) state of a given path. The temporal operator **F** defines that ϕ needs to eventually hold for a (symbolic) state in the future. The binary temporal operator **U** defines that ϕ_1 needs to hold for any (symbolic) state on a path until eventually ϕ_2 becomes true.

In TCTL, the temporal connectives may be restricted with a time bound $\sim c$. Based on the NTA in Figure 2.4, we may specify, for example, the following formula:

$$\varphi_1 \ = \ \mathbf{AG}(\texttt{Coordinator.Request} \rightarrow \mathbf{AF}_{\leq 50}\ (\texttt{Member.Idle or Member.Convoy}))$$

It specifies that on all execution paths it globally holds that whenever the coordinator automaton is in state Request, then on all execution paths the member automaton reaches

either the Idle state or the Convoy state within 50 time units. In the above formula, we omitted the time bound ≥ 0 of **AG** in the concrete syntax because it does not restrict the temporal connective. In this case, the temporal connectives of TCTL become semantically equivalent to the temporal connectives of CTL [ACD93]. Thus, any TCTL formula that only uses the time bound ≥ 0 is a valid CTL formula.

In accordance to UPPAAL [BDL04], we use a dedicated atomic proposition `deadlock` that refers to a deadlock state. A deadlock state is a state that does not have any successors. Using this atomic proposition, we may express that an NTA is free of deadlocks:

$$\varphi_2 \ = \ \mathbf{AG}\ \neg\texttt{deadlock}$$

In addition to the regular temporal connectives of TCTL introduced above, we use the weak-until connective (**AW** and **EW**) as part of our example in Chapter 5. Weak-until is a variation of the until connectives **AU** and **EU**. In contrast to until, weak-until does not require that ψ holds eventually, i.e., ϕ may hold globally. A weak-until, however, may be expressed in terms of the regular temporal connectives of TCTL [BK08, p. 327].

An alternative for specifying safety and liveness properties is given by *linear-time temporal logic* (LTL, [Pnu77, HR04]). LTL uses a linear time model based on paths that do not consider branching. Therefore, the temporal connectives of LTL only use a temporal operator but no path quantifier. We discuss preservation of LTL formulas for our refinement check, but do not use LTL for specifying safety and liveness properties as part of this thesis because timed variants of LTL like the metric temporal logic (MTL, [Koy90]) are not decidable for the dense time model used by NTAs presented above [AH92] and, thus, no model checkers exist.

2.2.3 Model Checking Procedure

A timed model checking procedure decides whether a given timed automata or NTA fulfills a given TCTL property [HNSY94, BDM$^+$98, BY04]. Therefore, the model checking procedure computes a zone graph for the timed automaton or the NTA. Then, it successively labels the resulting states with the atomic propositions and the subformulas that hold for a given state. The timed automaton or the NTA fulfills the TCTL property if and only if the formula is true for the initial state of the zone graph. If the TCTL property is not fulfilled, the model checker returns a counterexample. A `counterexample` is a trace of the zone graph that caused that the TCTL property is not fulfilled. In the course of this thesis, we use the model checker UPPAAL [LPY95, BDL$^+$06b] for verifying TCTL properties for NTAs.

2.3 Graph-Based Specifications

The theory of *graph transformation systems* (GTS, [Roz97]) is based on graphs. Intuitively, graphs consist of nodes and edges connecting the nodes. GTS define a language consisting of words where each word is a graph. Productions of a GTS are given by graph transformation rules that formally specify how one graph may be transformed into another one. In this thesis, we use GTS as a basic formalism for formalizing reconfiguration operations of MECHATRONICUML that modify the software architecture of a self-adaptive mechatronic

system. This, in turn, enables to formally verify reconfiguration operations of MECHATRONICUML, which we exploit in our reconfiguration approach introduced in Chapter 4.

In a general GTS, the nodes and edges of a graph do not have a predefined correspondence to a real-world or software entity. For defining reconfigurations of a software architecture by graph transformations, we need to define a correspondence between nodes and edges of a graph and the components and connectors of the software architecture. This a achieved by using typed attributed GTS that we introduce in Section 2.3.1. Story diagrams as introduced in Section 2.3.2 extend typed attributed graph transformation rules by the ability to specify control flow. This enables to specify more complex reconfiguration operations. Therefore, story diagrams are the basis for specifying reconfiguration operations in MECHATRONICUML as defined in Section 3.3.

2.3.1 Typed Attributed Graph Transformations Systems

Typed attributed GTS [EEPT06] extend GTS by a type graph and node attributes. The *type graph* defines which types of nodes exist and by which types of edges they may be connected. Node attributes enable to store values like integers or strings inside a node. Both features are essential for modeling reconfiguration (cf. Section 3.3).

Figure 2.6 shows an example of a simple type graph that defines two nodes types and six edge types. The two nodes types, RailCab and TrackSection, represent RailCabs and track sections. The node type RailCab defines an attribute of type Integer that stores the size of the convoy that the RailCab is currently driving in. The edge types define how nodes of type RailCab and TrackSection may be connected.

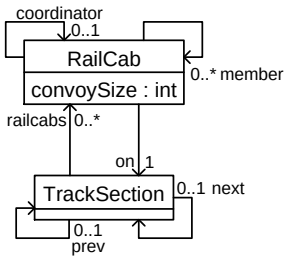


Figure 2.6: Example of a Type Graph

A TrackSection has a next and a prev TrackSection. These edge types define the outline of the track system. In addition, the edge type railcabs is used to refer to all RailCabs currently driving on a track section. The edge type on enables to define on which TrackSection a RailCab is currently located. Finally, coordinator and member enable a RailCab to refer to its coordinator or its members.

A *typed attributed graph* is always typed over exactly one type graph, while an arbitrary number of typed attributed graphs may use the same type graph. All nodes and edges of a typed attributed graph must be typed over exactly one node type or edge type, respectively, of the type graph. The type of a node or edge is immutable. In the course of this thesis, we will assume that the type graph is given by a metamodel [Küh06].

Figure 2.7 shows an example of a typed attributed graph that is typed over the type graph in Figure 2.6. It contains five nodes and five edges. It specifies the situation where two

RailCabs drive on two consecutive track sections. The RailCabs are not yet driving in a convoy because they are not connected with each other by a coordinator or member edge.

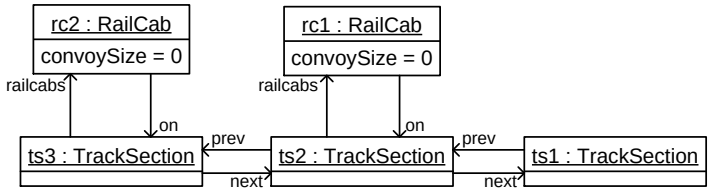


Figure 2.7: Typed Attributed Graph

Each node of a typed attributed graph has values for all of its attributes. In Figure 2.7, the nodes rc1 and rc2 both have value 0 for their convoySize attribute.

A typed attributed GTS consists of a type graph, an initial graph that is typed over the type graph, and a set of graph transformation rules that define how graphs may be modified. A graph transformation rule specifies a left hand side (LHS), a right hand side (RHS), a so-called rule morphism, and a set of negative application conditions (NAC) [EEPT06]. LHS, RHS, and NACs are typed attributed graphs based on the type graph. The rule morphism associates nodes in the LHS to nodes in the RHS to denote which nodes are the same. In addition, each NAC defines its own morphism that associates nodes in the LHS to nodes in the NAC.

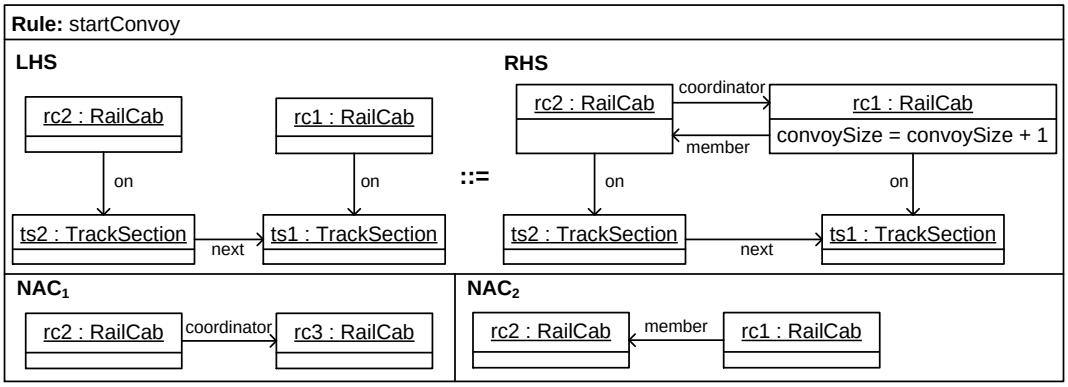


Figure 2.8: Graph Transformation Rule for Starting a Convoy

Figure 2.8 shows an example of a graph transformation rule startConvoy that starts a convoy between two RailCabs that are positioned on two consecutive track sections. The LHS specifies this situation. The RHS specifies the same situation but rc2 is now a member of a convoy that is coordinated by rc1. In our example, we implicitly define the rule morphism by using the same names, e.g., rc1 and rc2 for nodes in the LHS and RHS. The graph transformation rule startConvoy defines two NACs. NAC_1 defines a situation where rc2 is already a member of a convoy that is coordinated by a different RailCab rc3. NAC_2 defines a situation where rc2 is already a member of a convoy that is coordinated by rc1.

The application of a graph transformation rule such as startConvoy to a typed attributed graph is performed in three steps. In the first step, we search a *match* of the LHS to the typed attributed graph, the so-called *host graph*. Basically, a match is an occurrence of the LHS

in the host graph. The match needs to consider both, the type of the node and the attribute values of the node. That means, nodes of type RailCab in the LHS may only be matched to nodes of type RailCab in the host graph. If a node in the LHS specifies an attribute value, the matched node in the host graph needs to have the same attribute value. Attributes that are not used in the LHS are ignored while searching the match. A match for a graph transformation rule is only valid, if no NAC of the graph transformation rule can be matched to the host graph. In the second step, we remove all nodes and edges that occur in the LHS but not in the RHS of the graph transformation rule. To determine this set of nodes, we use the rule morphism. In the third step, we add all nodes and edges that occur in the RHS but not in the LHS. In this step, we also modify attribute values if necessary.

When applying the graph transformation rule startConvoy in Figure 2.8 to the typed attributed graph in Figure 2.7, we proceed as follows. For obtaining a match, we search for an occurrence of the LHS in the graph. The occurrence is given by the nodes rc1, rc2, ts2, and ts3 in Figure 2.7. Next, we need to check whether this match may be extended such that any NAC is completely matched. This is not the case because rc2 in Figure 2.7 is not yet member of a convoy. Thus, startConvoy has been successfully matched and we perform the graph rewriting. Therefore, we create a coordinator edge from rc2 to rc1 and a member edge from rc1 to rc2. In addition, we update the value of the attribute convoySize of rc1 by incrementing it by 1.

For the application of graph transformation rules, we follow the single pushout approach with injective matches (SPO, [Roz97]). In essence, that means that different nodes of the LHS need to be matched to different nodes in the host graph. For example, rc1 and rc2 in the LHS of startConvoy need to be matched to different RailCab nodes in the host graph. In addition, if the graph transformation rule specifies to delete a node without deleting all of its incident edges, then the incident edges are implicitly deleted as well to avoid dangling edges.

2.3.2 Story Driven Modeling

Story driven modeling (SDM, [Zün01]) is an approach for the object-oriented and model-driven software development. One essential part of SDM are *story diagrams* [FNTZ00, Zün01] that are used in the design phase for formally specifying operations of an object-oriented program. They combine an imperative control flow specification based on UML Activity Diagrams [Gro11c] with a formal, declarative specification of object manipulation based on typed attributed graph transformations, called *story patterns*. Story diagrams may also be used as an endogenous in-place model transformation language [CH06] and form the basis for defining reconfiguration operations in our approach as described in Section 3.3. In the following, we give a brief overview of story patterns (cf. Section 2.3.2.1) and story diagrams (cf. Section 2.3.2.2) based on the latest version by von Detten et al. [vDHP+12a].

2.3.2.1 Story Patterns

A story pattern consists of object variables and link variables that correspond to the nodes and edges of a typed attributed graph transformation rule. Object variables and link variables are typed over a metamodel [Küh06]. Story patterns use a concise notation of the typed attributed graph transformation rule that visualizes LHS, RHS, and NACs in a single graph. As an

example, consider the story pattern in Figure 2.9 that is equivalent to the typed attributed graph transformation rule in Figure 2.8.

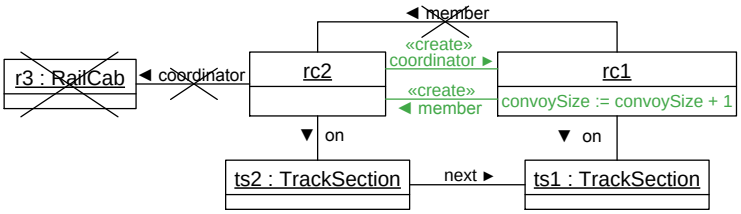


Figure 2.9: Story Pattern

Objects variables and link variables that shall be created by the story pattern are labelled with a «`create`» annotation such as the link variables coordinator and member in Figure 2.9. Object variables and link variables that shall be deleted are labeled with «`destroy`». All object variables and link variables not carrying an annotation are not changed by the graph rewriting.

Furthermore, object variables have a binding state. In particular, we distinguish between bound and unbound object variables. An unbound object variable needs to be matched by the graph matching when the story pattern is applied. A bound object variable has already been matched to an object of the host graph during the application of another story pattern. This matching is not changed while matching the unbound object variables of the story pattern. In our example, the object variables ts1, ts2, and r3 are unbound, while rc1 and rc2 are bound. In the concrete syntax, unbound variables visualize both, their name and their type, whereas bound variables only visualize their name.

Story patterns that are embedded in a story diagram always need to have at least one bound object variable. In addition, any unbound object variable must be reachable from at least one bound object variable by traversing link variables. The objective of this restriction is to reduce the matching effort for story patterns compared to typed attributed graph transformation rules. In general, deriving a matching for a typed attributed graph transformation rule is equivalent to the NP-complete subgraph isomorphism problem and, thus, requires exponential runtime. The bound object variables, however, provide starting points for the graph matcher and, in combination with the type graph, reduce the number of possible matchings and, thus, the runtime for deriving a valid matching significantly [SWZ95, Zün95, pp. 195ff.].

NACs are represented by so-called negative variables that are crossed out in the concrete syntax. In our example, the negative object variable rc3 and the negative link variable coordinator between rc2 and rc3 correspond to NAC_1 in Figure 2.8. They denote that rc2 does not have a coordinator reference to another RailCab. The negative link variable member from rc1 to rc2 corresponds to NAC_2 and defines that rc2 must not already be a member of rc1.

Object variables may contain conditions on and assignments to object attributes as in typed attributed graph transformation rules. In our example, the value of the attribute convoySize of rc1 is set to one. As in typed attributed graph transformation rules, conditions on object attributes are part of the LHS, while assignments to attributed values are part of the RHS.

2.3.2.2 Story Diagrams

A story diagram consists of activity nodes and activity edges for specifying the control flow such as sequential and conditional execution as well as loops. As part of this thesis, we use different kinds of activity nodes and activity edges that we illustrate below.

The kinds of activity nodes that we consider are initial nodes, final nodes, story nodes, decision nodes, activity call nodes, and statement nodes. Each story diagram contains exactly one initial node that marks the starting point of its execution. In addition, each story diagram has at least one final node that marks the end of its execution. A story node contains a story pattern and, thus, defines a modification of an object structure. Decision nodes enable to define complex branch and merge structures for the control flow. An activity call node [BvDHR11] enables to invoke another story diagram. Finally, statement nodes contain source code and may be used, for example, to define local counter variables.

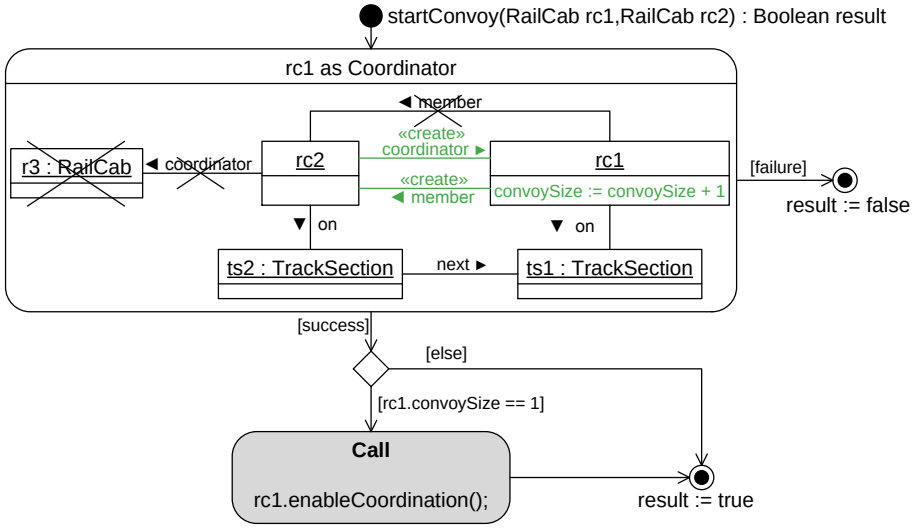


Figure 2.10: Story Diagram with Control Flow

As an example, consider the story diagram shown in Figure 2.10. The story diagram specifies the behavior of starting a convoy. It embeds the story pattern shown in Figure 2.9 in the story node named rc1 as Coordinator. If rc2 is the first member of rc1 as specified by the decision node below the story node, we additionally invoke enableCoordination on rc1 via the activity call node at the bottom of the figure.

Activity edges connect the activity nodes and define how the execution of the story diagram proceeds after executing an activity node. Story diagrams support different kinds of activity edges that are distinguished by their labels in the concrete syntax. The kind and number of outgoing activity edges depend on the kind of the source activity node.

Initial nodes and activity call nodes always have exactly one outgoing default activity edge but no other outgoing activity edges. A default activity edge has no label. Final nodes have no outgoing edges. A story node may either have one outgoing default activity edge or it may have one outgoing success activity edge, identified by the label [success], and one outgoing failure activity edge, identified by the label [failure]. The success activity edge is taken if the story pattern in the story node has been matched successfully. The failure activity edge is

taken if the story pattern could not be matched. Finally, a decision node may have either one outgoing default activity edge (merge node) or it has n outgoing activity edges where $n \geq 2$ (branch node). In the latter case, $n - 1$ of the outgoing activity edges must carry a Boolean condition, while the remaining one is an else activity edge that has the label [else]. In this case, the activity edge with a satisfied condition is executed or, if none of the Boolean conditions is fulfilled, the else activity edge is executed.

In addition to defining the control flow, the activity edges define how matchings are propagated through a story diagram. An initial matching of a story diagram is provided by the input parameters. The story diagram in Figure 2.10 has two input parameters rc1 and rc2 of type RailCab. This matching is propagated to the story node via the default activity edge. The story pattern, which is embedded in the story node, uses rc1 and rc2 as bound variables. If the story pattern can be applied successfully, the matching is extended by all matched and created variables. Destroyed object variables are removed from the matching. Then, the matching is propagated via the success activity edge to the subsequent node. If the story pattern cannot be matched, the matching is propagated unmodified via the failure activity edge. Decision nodes never change a matching. The Boolean conditions at the outgoing activity edges may refer to any object variables in the current matching and to their attributes. At a final node, object variables contained in the current matching can be assigned to the output parameters. In our example, however, we only assign the literals true and false to the output parameter result depending on whether the creation of the convoy was successful.

Story diagrams may be defined as an implementation of an operation of a class of the metamodel. In this case, the story diagram may be invoked by calling the operation for an object of the corresponding type. In our example in Figure 2.10, the activity call node invokes the operation enableCoordination on the object rc1 of type RailCab. In this case, rc1 serves as an implicit parameter for the story diagram and may be used as a bound variable with the name this in the embedded story patterns.

For defining loops, story nodes may be marked as *for-each story nodes*. A for-each story node is iteratively applied to any matching that may be obtained for the embedded story pattern in the host graph but guarantees that no matching is used twice. A for-each story node always has one outgoing end activity edge that is taken if no further matching may be obtained for the story pattern in the for-each story node (labeled with [end]). Optionally, a for-each story node may have an additional each time activity edge (labeled with [each time]) that is taken for each matching of the embedded story pattern.

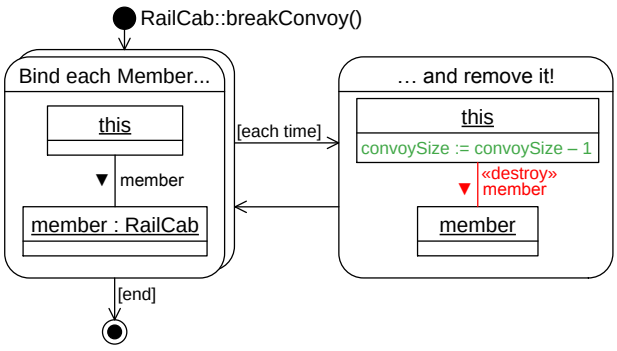


Figure 2.11: Story Diagram with for-each Activity Node

Figure 2.11 shows the story diagram breakConvoy. The story diagram may be invoked on an object of type RailCab, represented by the this variable, for dissolving a convoy. For the RailCab the for-each story node Bind each member..., which is visualized with a cascaded border line, matches any member of the RailCab. For each match, we exit the for-each story node via the each time activity edge. The second story node destroys the link to the member and decreases the convoySize by 1.

2.4 MechatronicUML

MECHATRONICUML [GTB+03, EHH+13, BDG+14a] is a model-driven software engineering method for developing event-discrete software of self-adaptive mechatronic systems. It adapts the concepts of UML 2.4 [Gro11c] for defining a component-based software architecture, state-based behavior, and runtime reconfiguration of a self-adaptive mechatronic system. In the course of this thesis, we integrate all of our contributions into MECHATRONICUML and provide an example model for the RailCab system based on MECHATRONICUML in Appendix A.

In the following, we briefly review the most important parts for specifying platform-independent models based on MECHATRONICUML that we use as part of this thesis. In particular, we introduce Real-Time Coordination Protocols (Section 2.4.1), Real-Time Statecharts (Section 2.4.2), and the assumptions on quality-of-service characteristics that are employed by MECHATRONICUML. For a detailed description of these parts of MECHATRONICUML, we refer to the MECHATRONICUML language specification [BDG+14b]. We discuss the component model of MECHATRONICUML and the specification and execution reconfigurations in detail in Chapters 3 and 4.

2.4.1 Real-Time Coordination Protocols

MECHATRONICUML uses *Real-Time Coordination Protocols* (RTCPs) for formally specifying asynchronous message-based communication between two communication partners [GTB+03, EHH+13]. RTCPs may be used in the reflective operator and in the cognitive operator of the OCM for defining message-based communication between different AMS but also between different components inside a single AMS.

An RTCP defines a name and two named roles that represent the communication partners. Each role has a behavior specification in terms of a Real-Time Statechart (cf. Section 2.4.2) that defines its behavior. The roles are connected by a role connector that specifies requirements to the physical connection such as the maximum transmission delay for a message and the possibility of message loss (cf. Section 2.4.3).

Figure 2.12 shows the declaration of a RTCP named DistanceTransmission that is used by a convoy coordinator for periodically transmitting new reference data to the members [HH11a, EHH+13]. The RTCP has two roles named provider and receiver. The RTCP is represented by the dashed ellipse, while the roles are represented by the dashed squares including the connection to the pattern ellipse. In our example, the role connector specifies a transmission delay of 1 ms for each message.

Each role defines a set of message types that it may send or receive. Message types are used to type the messages that are exchanged at runtime. They have a name and an optional list of

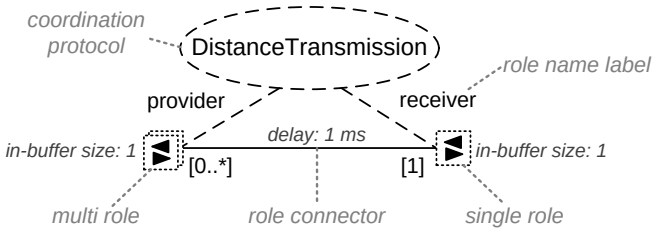


Figure 2.12: Declaration of the RTCP DistanceTransmission

typed, named parameters. Which messages may be sent or received at a particular point in time is defined by the Real-Time Statecharts of the roles, which we present in Section 2.4.2. If a role only receives messages, it is an in-role. If it only sends messages, it is an out-role. If it both sends and receives messages, it is an in/out-role [BDG+14b]. In our example in Figure 2.12, both roles are in/out-roles which is denoted by the two triangles inside the squares.

Received messages are stored in a message buffer that we call *in-buffer*. In this work, we restrict ourselves to FIFO-queues as in-buffers where all the received messages are stored in the same queue. Each role specifies a buffer size for its in-buffer [BDG+14b]. In our example, both roles specify a buffer size of 1, i.e., they can store at most one message in their in-buffer.

In addition, each role specifies a cardinality using a Min-Max-Notation as defined by Coad and Yourdon [CY90, p. 127]. Thus, the cardinality of a role defines with how many instances of the other role it may communicate at least and at most. If the upper bound of the cardinality equals 1, then we call it a single role. If the upper bound is greater than 1, we call it a multi role [EHH+13, BDG+14b]. In our example, the role receiver defines a cardinality of $[1]$ while provider defines a cardinality of $[0..*]$. Consequently, an instance of the multi role provider may communicate with 0 to many receiver's while any instance of the single role receiver may only communicate with exactly one provider.

At runtime, an instance of a multi role contains of a set of subrole instances as shown in Figure 2.13. Each subrole instance is connected via a single-cast connector to one instance of the single role and manages the communication with it. Due to the single-cast connectors, each subrole instance may only exchange messages with one particular single role instance.

Multi role instances are ordered, i.e., there exists a total order of the subrole instances. One subrole instance is the first one in the ordering, another one is the last one in the order. Adjacent subrole instances in the order have a successor-predecessor relationship. In our example in Figure 2.13, we may assume that the top most subrole instance is the first one while the bottom most subrole instance is the last one. The subrole instance in the middle is the successor of the first one and the predecessor of the last one.

2.4.2 Real-Time Statecharts

Real-Time Statecharts (RTSCs) as defined by Becker et al. [BDG+14b] are a combination of UML statemachines [Gro11c] and timed automata (cf. Section 2.2.1). Thus, they enable to specify hierarchical, state-based real-time behavior.

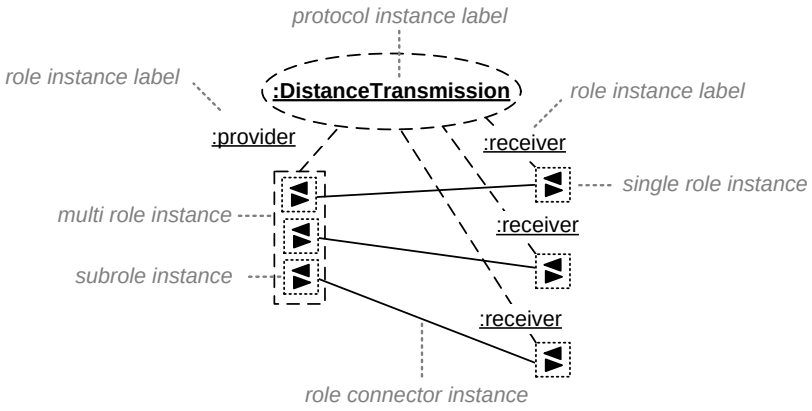


Figure 2.13: Instance of the RTCP DistanceTransmission (cf. [BDG+14b])

Basically, RTSCs consist of states and transitions. They use clocks with corresponding invariants, time guards, and resets as defined for timed automata (cf. Section 2.2.1). In addition, they may use variables for storing data and operations for encapsulating complex computations. As in UML statemachines, states may define actions that are executed upon entering (entry event) or leaving (exit event) a state.

As an example, we provide the RTSCs of the roles receiver and provider of the RTCP DistanceTransmission in Figures 2.14 and 2.15, respectively. They implement the behavior that the provider periodically sends an update message with a new reference distance and speed to all receivers. Each receiver acknowledges the receipt with an ack message.

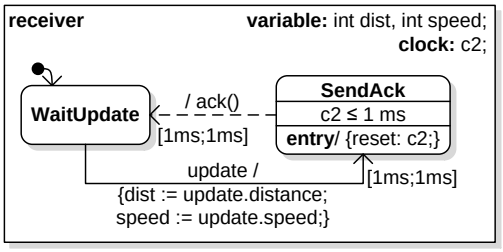


Figure 2.14: RTSC of Role receiver of DistanceTransmission

The RTSC of receiver shown in Figure 2.14 contains two states WaitUpdate and SendAck that are connected by two transitions. The state WaitUpdate is the initial state. The state sendAck defines an invariant based on c2 and an entry event that resets c2 upon entering the state. Thus, SendAck may be active for 1 ms. In contrast to timed automata, RTSCs use SI-units [BIPM06] for defining time values used in time guards, invariants, and deadlines.

A transition of a RTSC defines an enabling condition, an effect, and optionally a deadline. The enabling condition is a Boolean condition that defines whether a transition is *enabled*. If a transition is enabled, it may *fire* and thereby cause a state change in the RTSC. Upon firing, the effect of the transition is established. The deadline provides a lower and an upper bound on how long it takes to establish the transition effect. Thus, transitions of a RTSC are time-consuming in contrast to timed automata.

The enabling condition consists of a time guard, a guard condition using the variables of the RTSC, a synchronization, and a trigger message. Synchronizations based on synchronization channels are used in the same way as in timed automata. The trigger message defines the type of message that needs to be located at the head of the message buffer. All parts of the enabling condition are optional. In the concrete syntax, the enabling condition is placed before the "/" of the transition label.

The effect consists of an action, a raise message, and a clock reset. All are optional and executed in the given order. The action may modify the variables of the RTSC, call operations, and, in particular, invoke story diagrams that define reconfiguration behavior. The raise message defines a message that is sent including values for the parameters.

In the RTSC in Figure 2.14, the transition from WaitUpdate to SendAck requires an update message to be at the first position in the message buffer. The transition action assigns the values contained in the parameters distance and speed of the message update to the integer variables dist and speed. Executing the effect takes at least 1 ms and at most 1 ms as denoted by the deadline. The transition from SendAck back to WaitUpdate defines no enabling condition and sends a message ack as a part of its effect. Thus, this transition is always enabled.

In contrast to timed automata, RTSCs define urgency based on transitions rather than synchronization channels. In Figure 2.14, the transition from WaitUpdate to SendAck is urgent as denoted by the solid line, i.e., it fires as soon as it is enabled. The transition from SendAck to WaitUpdate is non-urgent, i.e., it fires at some point in time after it was enabled and before the invariant in SendAck expires.

Figure 2.15 shows the RTSC of the multi role provider. RTSCs of multi roles have a fixed form. They consist of one hierarchical state with two regions [BDG+14b]. The adaptation region contains the *adaptation RTSC* while the subrole region contains the *subrole RTSC* [EHH+13]. At runtime, each multi role instance executes exactly one instance of the adaptation RTSC. In addition, it executes one instance of the subrole RTSC for each subrole instance.

Hierarchical states optionally define a set of synchronization channels as, e.g., send and done in state Provider_Main. Then, transitions may specify synchronizations based on these synchronization channels as in timed automata. A synchronization always synchronizes two transitions whose enabling conditions are fulfilled. These transitions fire in an atomic fashion where the effect of the initiating transition (denoted by !) is executed before the effect of the receiving transition (denoted by ?). As in UPPAAL, the initiating transition is blocked if no receiving transition is enabled. Synchronizing transitions only fire urgently if both transitions are urgent. Otherwise, they fire non-urgently.

In RTSCs, synchronization channels may optionally use selectors that generalize the concept of channel arrays used in UPPAAL timed automata [BDL04]. Then, a synchronization channel defines a type for the selector while synchronizations provide a selector expression in square brackets that evaluates to the given type. A selector expression is either of type integer or, if it is used in a multi role, of type role. In both cases, the selector expressions define an additional condition for enabling the transition. Two transitions may only synchronize if they specify the same value in their selector expressions. Synchronizations that do not use a selector are called *plain synchronizations*.

As an example, consider the synchronization channel send in Figure 2.15 that specifies a selector of type role. If a selector of type role is used, two transitions may synchronize if they refer to the same subrole instance in their selector expressions. We support five dedicated

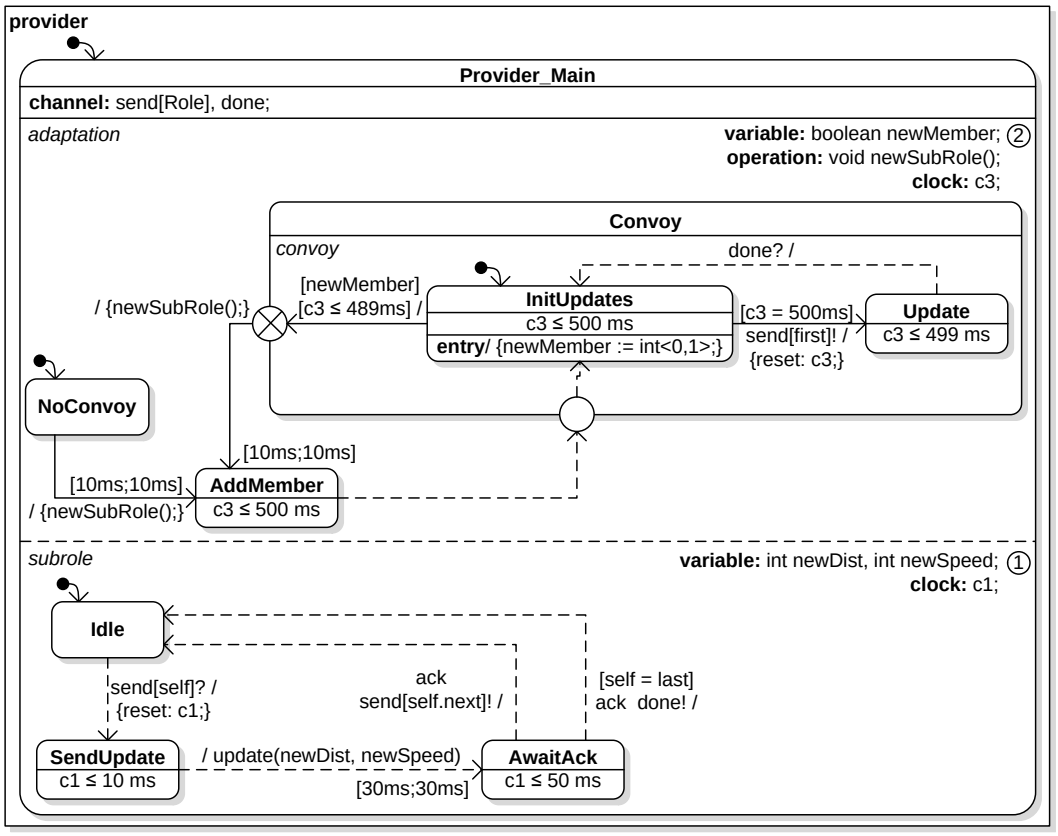


Figure 2.15: RTSC of Multi Role provider of DistanceTransmission

keywords for referring to subrole instance with respect to the order of the multi role instance. These are self, first, last, next, and prev. Self refers to the subrole instance that executes the RTSC. First (last) refers to the first (last) subrole instance of the multi role instance. Both return null if the multi role has no subrole instances. The keywords next and prev may only be applied to a subrole instance. Then, next (prev) returns the next (previous) subrole instance with respect to the order of the multi role instance. Next (prev) returns null if it is applied on the last (first) subrole instance.

In our example, the adaptation RTSC periodically triggers the first subrole RTSC of the ordered multi role instance via send at the transition from InitUpdates to Update to send an update to the receiver. This transition synchronizes with transition Idle to SendUpdate of the first subrole instance. Upon receiving the ack, the subrole instance either synchronizes via send with the next subrole instance in the multi role instance or, if it is the last one as expressed by the guard condition, it synchronizes via done with the adaptation RTSC.

RTSCs are deterministic except for so-called non-deterministic choice expressions. A non-deterministic choice expression defines an integer range and non-deterministically selects one value out of this range similar to a selection in a timed automaton (cf. Section 2.2.1). The value may be assigned to a variable as a part of an action. In our example, the entry action of InitUpdates uses a non-deterministic choice expression for assigning a value to newMember. newMember indicates that a new member wants to join the convoy. This decision is made

outside the RTCP and, therefore, we non-deterministically choose whether a new subrole shall be created.

The hierarchical state Convoy of the adaptation RTSC uses an entry-point and an exit-point. An entry point enables to activate particular substates upon entering a hierarchical state. If Convoy is entered via the entry-point at its bottom, the substate InitUpdates becomes active. An exit-point enables to define that a hierarchical state may only be left if a particular substate is active. In our example, Convoy may only be left if InitUpdates is active. When using entry- and exit-points, only the transition entering the entry- or exit-point may carry an enabling condition. In addition, only the transition leaving the entry- or exit-point may carry an effect [BDG+14b]. In this thesis, we require that RTSCs of single roles as well as the adaptation RTSC and the subrole RTSC of a multi role only use hierarchical states with one embedded region. Since hierarchical RTSCs with more regions may be flattened [DMY03, Ger13], this is no general limitation but eases the descriptions of our contributions.

In our example, the operation newSubRole that is called in the adaptation RTSC implements a reconfiguration rule. This reconfiguration rule adds a new subrole instance to an instance of the multi role provider. It may be formalized by a story diagram [EHH+13].

A *snapshot*[1] of an RTSC is defined by the active discrete state of the RTSC including the values of their variables and their clocks. As for timed automata, there exists an infinite number of snapshots for an RTSC. Therefore, we define the operational semantics of RTSCs by a zone graph using symbolic states as for an NTA. In particular, we define the operational semantics based on a network of flat timed automata as described in Section 2.2.1. Hierarchical states of RTSCs may be flattened to NTAs [DMY02, DMY03, Ger13]. Asynchronous communication using buffers may be mapped to additional timed automata representing the connector and buffer using shared integer variables for storing messages [KMR02, Ger13]. Deadlines as well as entry and exit actions may be resolved by intermediate states and transitions [GB03, DMY03]. Urgent transitions may be mapped to urgent channels using an additional automaton [DMY03]. Then, the rules for computing the zone graph are the same as those described in Section 2.2.1 with two exceptions. First, RTSCs use time guards at urgent transition, i.e., during a delay, time may only progress as long as no urgent transition becomes enabled. Second, urgent transitions have precedence over non-urgent transitions.

We provide a full formalization of the semantics of RTSCs based on NTAs in Appendix B that forms the basis of our refinement check in Chapter 5. Since our refinement check does not yet support reconfiguration of multi roles such as provider described above, we do not consider reconfiguration in our formalization. We refer to [EHH+13, HH11b] for a formal definition of the operational semantics of multi roles with reconfiguration.

2.4.3 Assumptions on Quality-of-Service Characteristics

In this thesis, we only consider platform-independent models of the discrete software of the self-adaptive mechatronic system. Thus, the underlying hardware resources and the network infrastructure [PMDB14] that are used for executing the software and for transporting messages are not part of our model. Nevertheless, our models cannot entirely ignore the timing and quality-of-service characteristics of the underlying networking infrastructure. We capture these quality-of-service (QoS) characteristics by a set of assumptions that we call QoS

[1] NTAs as introduced in Section 2.2.1 use the term *state*. We use the term *snapshot* in accordance to Gerking [Ger13] to avoid confusion with the states that are part of the syntax of an RTSC.

assumptions for the remainder of this thesis. Then, a RTCP may be safely executed using a networking infrastructure if this networking infrastructure guarantees to fulfill the QoS assumptions [HBDS15]. We present our assumptions in the following.

A fundamental assumption for our approach is that the clocks within the two roles of a RTCP run synchronously at the same rate. This also holds for any two ports of components that communicate according to the RTCP. This assumption is realistic because there exist standards like the precision clock synchronization protocol [IEE08] that may synchronize clocks with a precision of a few microseconds. Such a precision is sufficient because time constraints for mechatronic systems like cars are typically specified in the order of magnitude of milliseconds [SLT09, p. 7]. There are approaches existing as well for clock synchronization regarding heterogeneous and adaptive hardware platforms [BK13].

In addition, we consider several assumptions regarding the transmission of messages. We introduce them by following a message from the sender to the receiver. For a firing transition that defines a sender message, we assume that this message is immediately handed over to the underlying network layer. This layer sends the message and — if needed — buffers it before sending. Thus, we do not use buffers for outgoing messages on the level of MECHATRONICUML.

As stated in Section 2.4.1, the transmission of a message from the sender to the receiver takes time. Therefore, the developer has to define a message delay. We assume that the delay defines the time between sending the message from the level of MECHATRONICUML and storing the message within the receiver's in-buffer. Thus, this delay is not just the transmission via the physical medium but also the transport through the underlying network layers. As a consequence, we allow that a message is retransmitted via the physical medium if it gets lost during the transmission as long as it arrives in the receiver's in-buffer within the delay. However, if the underlying network layer assumes by mistake that a message got lost, duplicate messages may arrive at the underlying network layer. We assume that the underlying network layer detects and deletes such messages using duplicate message detection mechanisms [Kiz05]. If the in-buffer is full and another message arrives, we assume that the incoming message is dropped. If a connector of the RTCP guarantees that no messages get lost during communication, all messages arrive at the receiver's in-buffer within the transmission delay. In addition, we assume that messages are never reordered during transmission.

3 MechatronicUML Component Model

MECHATRONICUML follows a component-based approach for defining the software architecture of a system. "The cornerstone of any component-based development methodology [SGM02] is its underlying *component model*, which defines what components are, how they can be constructed and represented, how they can be composed or assembled, how they can be deployed and how to reason about all these operations on components." [Lau06] In addition, the component model defines how components interact if they are composed or assembled [HC01, p. 11]. Therefore, the component model is the central artifact for designing the software of a (mechatronic) system. Consequently, we need a precise definition of a component model for MECHATRONICUML that serves as a basis for formal analyses and transformations to other languages like MATLAB/Simulink [Matg].

A component model for defining software architectures of self-adaptive mechatronic systems needs to consider the properties of these systems. It needs to support message-based communication between components and their reconfiguration at runtime (cf. Section 1.1). This includes, in particular, to establish connections between AMS that were previously not connected with each other such that they may collaborate in an NMS. In addition, the component model needs to enable the specification of real-time behavior using, for example, RTSCs for coping with the hard real-time requirements of mechatronic systems. Moreover, the component model needs to enable the integration of feedback controllers into the software architecture because only the integration of feedback controllers and the software components enables advanced functionality as the convoy mode of the RailCab system [HTS+08a]. Finally, the component model shall facilitate the specification and formal verification of reconfiguration operations such that the software architecture remains syntactically and semantically correct after a reconfiguration.

In previous works, two component models have been developed for MECHATRONICUML based on the requirements introduced in the previous paragraph. The component model by Burmester, Giese, and Hirsch [GTB+03, GBSO04, HHG08, GS13] focuses on integrating feedback controllers into the software architecture and reconfiguring them at runtime. Their component model uses a state-based formalism called hybrid reconfiguration charts that enumerates all possible software architectures that the system may use at runtime and how the system may switch between them. The component model by Tichy [THHO08, Tic09] focuses on a formal, flexible, and concise specification of reconfiguration operations using a domain-specific variant of typed attributed graph transformations called component story diagrams. In this approach, the components of the component model define the type graph for the component story diagrams. Then, the type graph defines syntactical restrictions based on the components that guarantee syntactical correctness of the software architecture after a reconfiguration. In addition, it enables the formal verification of component story diagrams for proving correctness of the reconfigurations.

Both existing component models do not fulfill all of the aforementioned requirements. The component model by Burmester, Giese, and Hirsch provides no support for instantiat-

ing embedded components more than once. This and the fact that all software architectures need to be enumerated lead to large models, in particular, if a component may have several architectures at runtime. This makes the models hard to handle for a developer. The component model by Tichy does not distinguish between software components and feedback controllers in the type graph that is used for specifying component story diagrams. As a result, component story diagrams cannot specify the reconfiguration of feedback controllers. In addition, both component models do not enable to connect software components to feedback controllers and to establish connections between different AMS [HB14].

In this chapter, we derive a consolidated component model for MECHATRONICUML that combines the features of the two existing component models. In particular, we extend the concept of the type graph used by Tichy such that the component model may include feedback controllers and such that they may interact with software components. As a result, we can specify software architectures on the reflective operator and controller levels of the OCM. In addition, we extend component story diagrams such that they can reconfigure feedback controllers as defined by Burmester and Giese [GBSO04, Bur06, BGO06]. Finally, we provide a concept for establishing connections between AMS. As a result, our new component model enables for concise and formal specifications of components, their integration with feedback controllers, and their reconfiguration behavior.

In the following, we illustrate our component model based on a software architecture for the driving module of a RailCab that includes the behavior for building convoys. The requirements for the convoy behavior have been presented in our technical report [Hei12]. In our example, we use ideas presented by Hirsch [Hir08], Tichy [Tic09], and Flaßkamp et al. [FHK+13]. These ideas have been significantly extended as part of this thesis.

The remainder of this chapter is structured as follows. We start by defining how components (Section 3.1), component instances (Section 3.2), and reconfiguration operations (Section 3.3) are specified. Thereafter, we introduce our concepts for establishing connections between AMS (Section 3.4) and for specifying architectural constraints (Section 3.5). Next, we describe how the new component model has been implemented as part of the MECHATRONICUML Tool Suite (Section 3.6). Finally, we discuss related approaches (Section 3.7) and summarize the chapter (Section 3.8).

3.1 Modeling Components

"A [..] *component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard." [HC01, p. 7] In accordance to UML [Gro11c], components are either implemented directly or they are assembled from other components. We refer to the former as *atomic components* and to the latter as *structured components*.

In both cases, the internals of a component are hidden from the outside world. This is denoted as component encapsulation [SGM02]. Access to the capabilities or data of a component is only allowed via its ports. This enables to replace one component by another one with a compatible interface without affecting any other component in a system. In addition, component encapsulation is one of the key enablers of compositional verification [BCC98, GTB+03] because it guarantees that there may not exist more dependencies to other components than those captured by ports. We will exploit this in Chapter 5.

Our component model explicitly distinguishes between component types and component instances. The component types are instantiated to component instances for representing the software architecture of a system. In the following, we refer to component types simply as *components*. We introduce component instances in detail in Section 3.2.

We illustrate the specification of components and component instances based on examples given in concrete syntax. A formalization of the component model is given by a metamodel [SV06, ch. 4] whose abstract syntax is defined in Appendix D.1. The static semantics has been formalized based on constraints in the object constraint language (OCL, [Gro12]) that are contained in the metamodel. The OCL constraints are listed in the MECHATRONICUML language specification [BDG+14b].

In the following, we first introduce the different kinds of ports that we support in our component model (Section 3.1.1). They differ in the kind of information they process and in their purpose. Based on the different kinds of ports, we define different kinds of atomic components (Section 3.1.2) and structured components (Section 3.1.3). Thereafter, we define how components may be connected via their ports using connectors (Section 3.1.4). As an extension to the previous component models, our component model supports that a component may expose a set of component properties (Section 3.1.5) that we need for our reconfiguration concept presented in Chapter 4. The concepts presented in this section have successively been integrated into the MECHATRONICUML language specifications [BDG+11, BBD+12, BBB+12, BDG+14b].

3.1.1 Ports

In our component model, we distinguish between six kinds of ports based on their purpose and the kind of data they process. We use discrete and continuous ports as defined by Burmester and Giese [GBSO04, Bur06, BGO06]. Additionally, we use hybrid ports that enable to connect discrete software components and feedback controllers. Furthermore, we use broadcast ports for instantiating RTCPs between AMS. Finally, we use two kinds of reconfiguration ports, namely reconfiguration message ports (RM ports) and reconfiguration execution ports (RE ports), that enable to execute reconfigurations involving several component instances. Figure 3.1 summarizes the concrete syntax of the different kinds of ports.

Each port defines a cardinality that defines how many instances of it may be created in one component instance. The previous component models only supported three fixed cardinalities that defined that the port can be instantiated at most once ($[0..1]$), exactly once ($[1]$), or arbitrary often ($[0..*]$). We extend the concept of cardinalities by enabling to specify precise cardinalities using an integer for lower and upper bound. Again, we allow $*$ as an upper bound to indicate that the port may be instantiated arbitrary often. If the cardinality has a lower bound of 0, we call it an *optional port* and visualize it with unfilled triangles (cf. Figures 3.1b and 3.1d) according to Giese and Schäfer [GS13]. If the lower bound of the cardinality is greater or equal to 1, we call it a *mandatory port* and visualize it with filled triangles (cf. Figures 3.1a and 3.1c). Ports with an upper bound of 1 are called *single ports* while ports with an upper bound greater than 1 are called *multi ports* in accordance to Hirsch [Hir08, HHG08]. We visualize multi ports with a cascaded border line as shown in Figures 3.1c and 3.1d [Hir08, HHG08, Tic09].

In the following, we introduce the different kinds of ports in more detail.

	in-port	out-port	in/out-port
discrete			
continuous			n/A
hybrid			n/A
RM / RE	n/A	n/A	
broadcast	n/A	n/A	

(a) Mandatory Single Ports

	in-port	out-port	in/out-port
discrete			
continuous			n/A
hybrid			n/A
RM / RE	n/A	n/A	n/A
broadcast	n/A	n/A	

(b) Optional Single Ports

	in-port	out-port	in/out-port
discrete			
continuous	n/A	n/A	n/A
hybrid	n/A	n/A	n/A
RM / RE	n/A	n/A	
broadcast	n/A	n/A	n/A

(c) Mandatory Multi Ports

	in-port	out-port	in/out-port
discrete			
continuous	n/A	n/A	n/A
hybrid	n/A	n/A	n/A
RM / RE	n/A	n/A	n/A
broadcast	n/A	n/A	n/A

(d) Optional Multi Ports

Figure 3.1: Kinds of Ports (cf. [BDG+14b])

3.1.1.1 Discrete Port

Discrete ports send and receive asynchronous messages. Therefore, each discrete port defines a set of message types that it may send or receive. A message type has a name and an ordered set of typed, named parameters. In contrast to the existing component models, we do not provide explicit interfaces in terms of required and provided interfaces. Instead, we use a port-based specification of the interface where we directly assign message types to the ports [CSVC11]. This approach introduces flexibility that we need for introducing hierarchical reconfiguration in Chapter 4.

If a discrete port only sends messages, it is an *in-port* which is denoted by a small triangle pointing "into" the component similar to the notation of Koala [vOvdLKM00]. If it only sends messages, it is an *out-port* as denoted by the small triangle pointing "outside" the component. If it both sends and receives messages, it is an *in/out-port* denoted by two embedded triangles. Discrete ports define a message buffer in the same fashion as a role of a RTCP (cf. Section 2.4.1).

Each discrete port needs to refine a role of an RTCP (cf. Section 2.4.1). Then, the discrete port needs to send and receive the same message types as the role. In our concrete syntax, we enable to visualize the role that is refined by a port by a dashed line that is attached to the port as shown in Figure 3.5 on Page 42. Visualizing the refined role of a port is optional.

A discrete port has a behavior specification in terms of an RTSC. The behavior that is defined by the port's RTSC needs to be compliant to the behavior that is defined by the role. We describe how the RTSC of a role may be refined to an RTSC of a port in detail in Chapter 5. The RTSC of a multi port has the same structure as the RTSC of a multi role (cf. Section 2.4.2).

3.1.1.2 Reconfiguration Message Port and Reconfiguration Execution Port

Reconfiguration message ports (RM ports) and *reconfiguration execution ports* (RE ports) are special kinds of discrete ports. We use these kinds of ports for realizing the communication that is necessary for our concept of transactional execution of reconfiguration in structured components as described in Chapter 4. In the concrete syntax, we visualize RM ports and RE ports by squares that embed the letter "RM" and "RE", respectively.

Compared to discrete ports, RM ports and RE ports have extended interface specifications that provide additional information for the messages types that may be sent or received. We introduce the interface specification in detail in Section 4.3.

RM ports and RE ports have message buffers and their behavior is defined by a RTSC as for discrete ports. However, RM ports and RE ports are always mandatory in/out ports. Both may be used as multi ports as we explain in Section 4.1.

3.1.1.3 Broadcast Port

Broadcast ports are a special kind of discrete port. We use broadcast ports only for instantiating RTCPs between different AMS as explained in Section 3.4. In the concrete syntax, we visualize broadcast ports by squares that embed the letter "B".

Analogously to discrete ports, broadcast ports define a set of message types that they may send and receive as well as a message buffer. Their behavior is defined by a RTSC. In contrast to discrete ports, broadcast ports are always in/out-ports and may only be used as single ports. In addition, they do not refine a role of a RTCP.

3.1.1.4 Continuous Port

Continuous ports send (out-port) or receive (in-port) a signal value. "A *signal* is a time varying quantity that has values at all points in time" [Matf]. The data type of the signal must be a primitive data type or an array of primitive types.

Continuous ports are either in-ports or out-ports. In addition, continuous ports may be optional, but we currently do not support continuous multi ports.

In the concrete syntax, continuous ports are visualized as isosceles triangles where the top of the triangle either points into the component (in-port) or outside the component (out-port).

3.1.1.5 Hybrid Port

Hybrid ports send (out-port) or receive (in-port) a signal value similar to a continuous port. They enable that a discrete component sends a signal to or receives a signal from a feedback controller. As for a continuous port, the data type of the signal must be a primitive data type or an array of primitive types. Thus, we define a new semantics of hybrid ports compared to Burmester [Bur06]. Burmester introduced hybrid ports as "multiple discrete and continuous ports as syntactic construct to reduce visual complexity" [Bur06, p. 56] but did not define them.

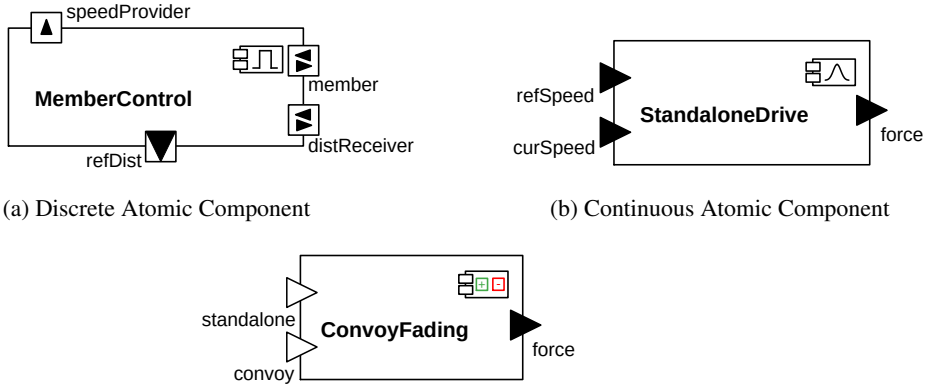
Hybrid ports are either in-ports or out-ports. Then, the RTSC of the component may read (in-port) or write (out-port) the value of the hybrid port like a normal variable. In addition, hybrid ports may be optional, but we currently do not support hybrid multi ports.

For keeping the behavior specification of a discrete component discrete, hybrid ports define a sampling interval. Then, the value of the signal only changes at the rate of the sampling interval and we do not make any assumptions on how the value may change.

In the concrete syntax, hybrid ports are visualized as squares that embed an isosceles triangle. The top of the triangle either points into the component (in-port) or outside the component (out-port).

3.1.2 Atomic Components

An *atomic component* directly contains a behavior specification and does not embed other components. Our component model distinguishes three kinds of atomic components that differ in their purpose and their behavior specification. In accordance to Burmester and Giese [GBSO04, Bur06, BGO06], we distinguish between discrete and continuous atomic components. Discrete atomic components define discrete, event-based behavior while continuous atomic components represent the feedback controllers of the system. In addition, we introduce a new kind of atomic component: the fading component [Vol13].



(a) Discrete Atomic Component

(b) Continuous Atomic Component

(c) Fading Component

Figure 3.2: Kinds of Atomic Components

Figure 3.2 illustrates the concrete syntax of the different kinds of atomic components. In accordance to the UML [Gro11c], components are represented by rectangles with a com-

ponent icon in the upper right corner and a name label in the center. At the border of the component, we visualize the ports of the component.

In contrast to the previous component models, we distinguish the different kinds of atomic components by using different component icons to increase semiotic clarity. Semiotic clarity requires that different semantic constructs of a language need to be represented by different graphical symbols for reducing the potential for misinterpretation [Moo09]. In the following, we introduce all three kinds of atomic components in more detail.

3.1.2.1 Discrete Atomic Component

Discrete atomic components define the discrete, event-based real-time behavior of the system. As a result, a discrete component operates on time-discrete values and implements message-based communication. Thus, discrete atomic components are used for defining the behavior of the reflective operator of the OCM.

A discrete atomic component may use discrete ports for interacting with other components based on RTCPs. In addition, it may use hybrid ports for interacting with continuous atomic components (cf. Section 3.1.2.2) and broadcast ports if it implements the instantiation of RTCPs between AMS. If the component is reconfigurable, it has one RM port and one RE port.

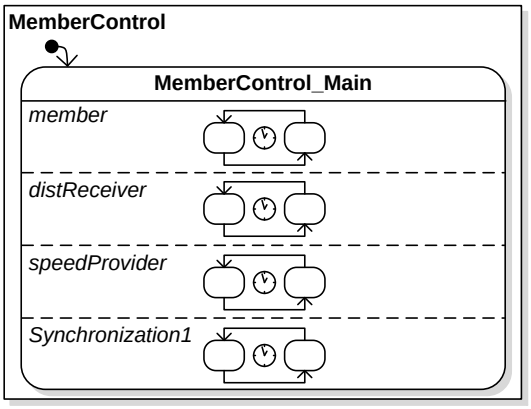


Figure 3.3: Structure of a RTSC of a Discrete Atomic Component

The behavior of a discrete atomic component is defined by a RTSC that has a fixed, hierarchical structure [Hir08, p. 133]. Figure 3.3 illustrates this structure for the component MemberControl shown in Figure 3.2a. The RTSC always contains one hierarchical state. This state contains one region for each discrete port that embeds the port's RTSC. In the example, we obtain regions for the discrete ports member, distReceiver, and speedProvider. In addition, the RTSC may contain an arbitrary number of so-called *synchronization RTSCs* that may be used to synchronize the port RTSCs [GTB+03].

3.1.2.2 Continuous Atomic Component

Continuous atomic components represent the feedback controllers of the system that are located of the controller level of the OCM. They operate on time-continuous values that are

represented by signals. Their behavior is typically defined "by block-diagrams, differential equations, or transfer functions" [Bur06, p. 56].

A continuous atomic component may only use continuous ports for exchanging signals with other components. In accordance to Burmester et al. [BGH+07], we only specify the interface of the continuous component based on its ports but not the component's behavior. The behavior of continuous atomic components is specified in a control engineering tool such as MATLAB/Simulink [Matg].

As an example, consider the continuous atomic component StandaloneDrive shown in Figure 3.2b. It implements a feedback controller that lets a RailCab drive at a constant speed. It receives a reference speed via refSpeed and the current speed of the RailCab via curSpeed. By modifying force of the electric drive emitted via force, it modifies the speed of the RailCab such that curSpeed eventually equals refSpeed. This control strategy, however, needs to be implemented in MATLAB/Simulink.

3.1.2.3 Fading Component

A fading component enables to switch between continuous component instances as part of a reconfiguration if the continuous component instances produce the same output signal. Thus, fading components operate on time-continuous values like continuous atomic components and are located on the controller level of the OCM.

As an example, consider that the continuous component StandaloneDrive shown in Figure 3.2b is to be replaced by a continuous component ConvoyDrive as illustrated in Figure 3.4. The ConvoyDrive component implements a feedback controller that additionally considers a reference distance (refDist) and the current distance (curDist) to the preceding RailCab. It needs to be used by all RailCabs that are convoy members. Thus, any RailCab that wants to join a convoy needs to perform this replacement at runtime as part of a reconfiguration.

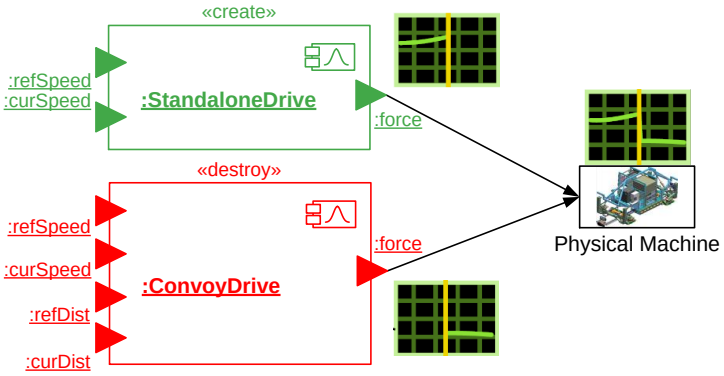


Figure 3.4: Illustration of Exchanging a Controller without Fading Function

In general, continuous component instances must not be replaced instantaneously if they produce the same output signal such as force in Figure 3.4. In the figure, the green graphs illustrate the computed value of force over time while the vertical yellow bar denotes the point in time where the continuous component instances are replaced instantaneously. In this case, a jump in the value of the controlled variable force occurs at the engine and may damage it.

For preventing such jumps, previous works integrated fading functions based on cross fading [BGO06] and flatness-based switching [OMT+08] into MECHATRONICUML. These are two strategies for smoothing the output signal while replacing continuous component instances. The actual behavior of the fading function or the flatness-based switching is specified in a control engineering tool such as MATLAB/Simulink [Matg].

In our component model, we encapsulate fading functions and flatness-based switching in fading components such as ConvoyFading shown in Figure 3.2c. The fading component has one continuous out-port for the output signal and one continuous in-port for any continuous component that may produce this output signal. Thus, ConvoyFading has one out-port force and in-ports standalone and convoy for the two continuous components StandaloneDrive and ConvoyDrive, respectively.

In addition to the ports, the fading component defines a set of fading functions. Each fading function fades from the input signal of one in-port to the input signal of another in-port. In the example in Figure 3.4, the ConvoyFading would need to fade from standalone to convoy. At this point, we do not need to distinguish whether the fading function implements a cross fading [BGO06] or flatness-based switching [OMT+08]. We only need to specify how long it takes to execute the fading function. If the fading component does not execute a fading function, it forwards the input signal unmodified to its out-port.

3.1.3 Structured Components

A *structured component* embeds other component types by means of component parts as defined in the component model by Tichy [Tic09]. Component parts are defined as an association to another component [Gro11c], i.e., the same component may be embedded multiple times in a structured component. Component parts define a name and a cardinality.

Structured components only define a reconfiguration behavior but no functional behavior. This enables separation of concerns between reconfiguration behavior and functional behavior. According to McKinley et al. [MSKC04], this is one of the three key enablers for successfully developing self-adaptive systems. With respect to the OCM given in Section 2.1.2, structured components belong to the reflective operator.

In contrast to the existing component models, our component model distinguishes between two kinds of structured components based on the kinds of components they embed. These are discrete structured components (cf. Section 3.1.3.1) and hybrid structured components (Section 3.1.3.2). The differentiation between two kinds of structured components is helpful for defining our transactional reconfiguration approach in Chapter 4.

3.1.3.1 Discrete Structured Component

A discrete structured component (recursively) embeds discrete components only. Consequently, a discrete structured component may use all kinds of ports except continuous ports analogous to discrete atomic components (cf. Section 3.2a).

Figure 3.5 shows an example of a discrete structured component named ConvoyCoordination. It contains the behavior of a convoy coordinator, i.e., it provides behavior for adding and removing RailCabs to/from the convoy and for announcing all acceleration and breaking maneuvers to the convoy members. ConvoyCoordination embeds two components ConvoyMan-

agement and RefGen using two component parts named man and refGen, respectively. Both of which are discrete components.

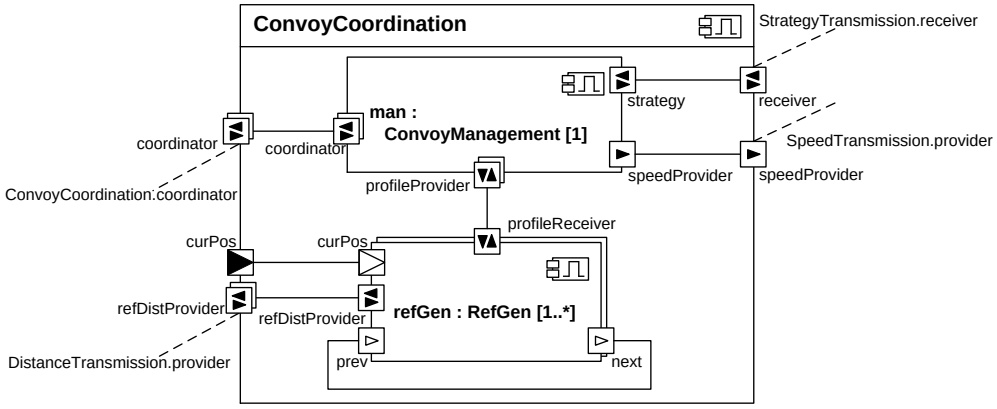


Figure 3.5: The structured component type ConvoyCoordination

Our component model allows for a precise specification of cardinalities using integers for lower and upper bound, but still enables to use an asterisk to support an arbitrary number of instances. Supporting precise cardinalities is especially useful for simulations in a simulation tool as MATLAB/Simulink as presented in Chapter 6. In Figure 3.5, the component part man has a cardinality of $[1]$. That means any instance of ConvoyCoordination contains exactly one instance of ConvoyManagement. We call this a *single part*. refGen has a cardinality of $[1..*]$ such that an instance of ConvoyCoordination has arbitrary many but at least one instance of RefGen. In accordance to Tichy, we call this a *multi part*. In the concrete syntax, multi parts are visualized by a cascaded border line [Tic09, p. 38].

In ConvoyCoordination, the ConvoyManagement is responsible for adding and removing convoy members to the convoy and for negotiating the maximum speed of the convoy. The interaction with the convoy members is implemented in the port coordinator that refines the role coordinator of the RTCP ConvoyCoordination [FHK+13, FHK+14]. We present the RTCP ConvoyCoordination in Appendix A.1.2.

For each convoy member, the ConvoyCoordination has one instance of the RefGen multi part that generates reference data for the convoy member (cf. [Tic09]). RefGen receives information about the corresponding convoy member and the negotiated speeds from ConvoyManagement via profileReceiver. The information about the convoy members is encapsulated in so-called profiles [Hir08, FHK+13, FHK+14]. The RefGen instance for the first convoy member additionally receives the position of the coordinator RailCab via curPos. Then, RefGen computes a reference distance to the preceding RailCab based on the position of this preceding RailCab and the profile of the RailCab. This can be used to adapt the distances between RailCabs within the convoy to changing environmental conditions such as higher speeds, strong wind, or slopes. RefGen sends the new reference distance to the convoy members using the RTCP DistanceTransmission introduced in Section 2.4.

3.1.3.2 Hybrid Structured Component

A hybrid structured component embeds a mixture of discrete, hybrid, and continuous components. Hybrid structured components may use all kinds of ports. Component parts and their cardinalities are used in the same fashion as in discrete structured components.

Figure 3.6 shows the hybrid structured component RailCabDriveControl that implements the driving functions of the RailCab. It embeds eight component parts that implement different parts of the behavior. In the following, we provide a detailed description of the RailCabDriveControl component because it forms the basis of our running example that we use in the remainder of this thesis for illustrating our concepts.

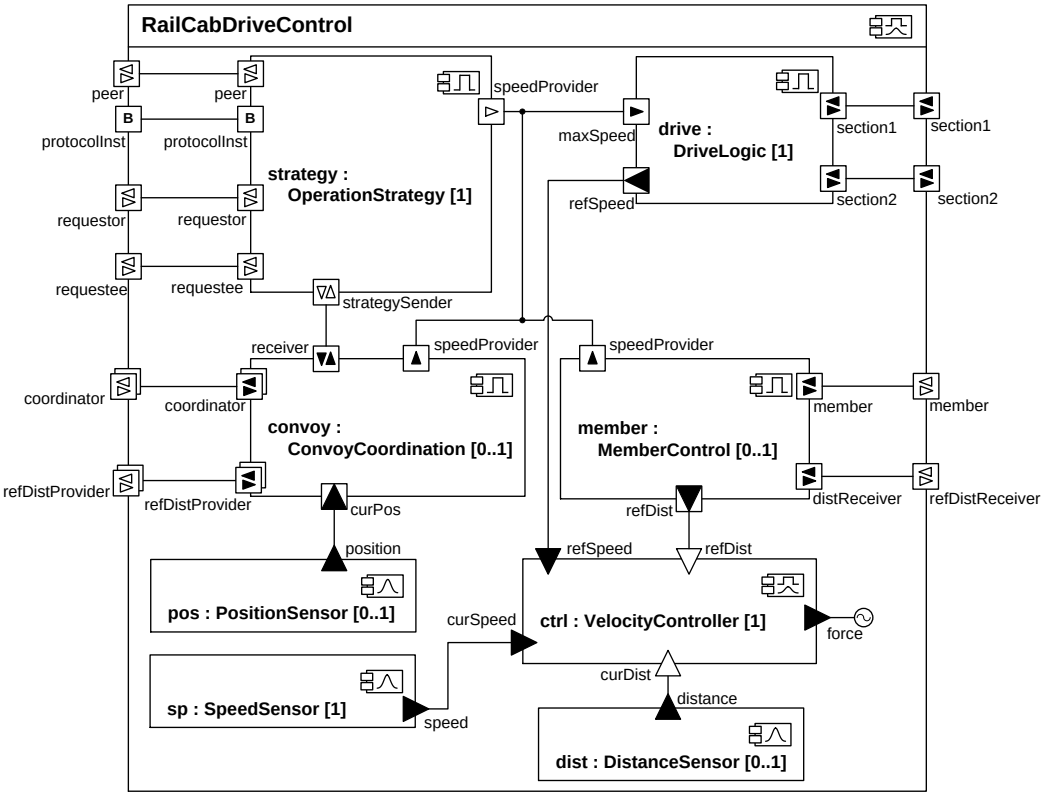


Figure 3.6: The component type RailCabDriveControl

The component OperationStrategy implements the operation strategy that defines, for example, the maximum speed for the RailCab. In addition, it contains the logic for deciding whether to build a convoy or not. Via the broadcast port protocolInst, it establishes connections to other RailCabs that are eligible for building convoys (cf. Section 3.4.1). The ports requestor and requestee implement both roles of the RTCP ProtocolInstantiation introduced in Section 3.4.2. The ProtocolInstantiation protocol implemented in OperationStrategy only instantiates the RTCP ConvoyEntry that is refined by the peer port. This RTCP specifies the message exchange for negotiating whether to build a convoy or not and which RailCab will serve as the coordinator for the convoy. We provide a description of this RTCP in Appendix A.1.1.

The component DriveLogic defines the current speed of the RailCab that it sends via the refSpeed port to the VelocityController. The current speed is either defined by the OperationStrategy

if the RailCab drives alone or it depends on the maximum speed that has been negotiated for the convoy. In addition, the DriveLogic contains the two ports section1 and section2. These ports are used for communicating with the current and the next track section for gaining admission to drive onto a track section. This is necessary for avoiding collisions between RailCabs that want to drive onto the same track section. In addition, this communication may be used to obtain further information about the track characteristics (cf. [BGO06, Hir08]) or a track specific maximum speed [Hei12]. We introduce the associated behavior in Chapter 5.

The component part convoy is typed by the component ConvoyCoordination shown in Figure 3.5. It is connected to the operation strategy because it needs to be informed about the information that has been negotiated with new convoy members. The continuous port curPos is connected to the PositionSensor that provides the current position of the RailCab.

The component MemberControl, also shown in Figure 3.2a, implements the behavior for operating as a convoy member. The ports member and distReceiver implement the complementary roles of the RTCPs ConvoyCoordination and DistanceTransmission for communicating with the coordinator. In particular, MemberControl receives the reference speed and reference distances for driving in the convoy. It sends the reference speed via speedProvider to the DriveLogic and the reference distance via refDist to the VelocityController.

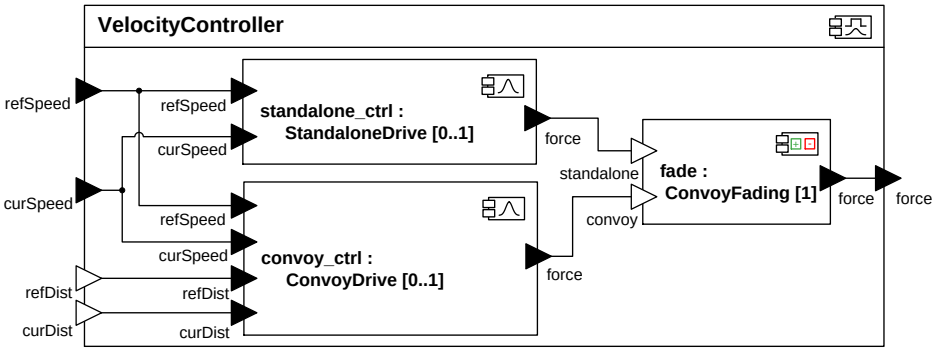


Figure 3.7: The component type VelocityController

Finally, the VelocityController shown in Figure 3.7 contains the feedback controllers that control the electric motors of the RailCab. The continuous component StandaloneDrive contains the feedback controller that is used if the RailCab drives alone or as a convoy coordinator. Based on the current speed obtained from the SpeedSensor and the reference speed provided by the DriveLogic it computes a force to be applied by the electric motor. The port force is directly connected to the actuator and, therefore, remains unconnected in RailCabDriveControl. If the RailCab operates as a convoy member, it needs to execute the feedback controller implemented in the continuous component ConvoyDrive. It additionally considers the current distance obtained from the DistanceSensor and the reference distance provided by the MemberControl for computing the force. In addition, the VelocityController contains the fading component ConvoyFading shown in Figure 3.2c for switching between the two continuous components.

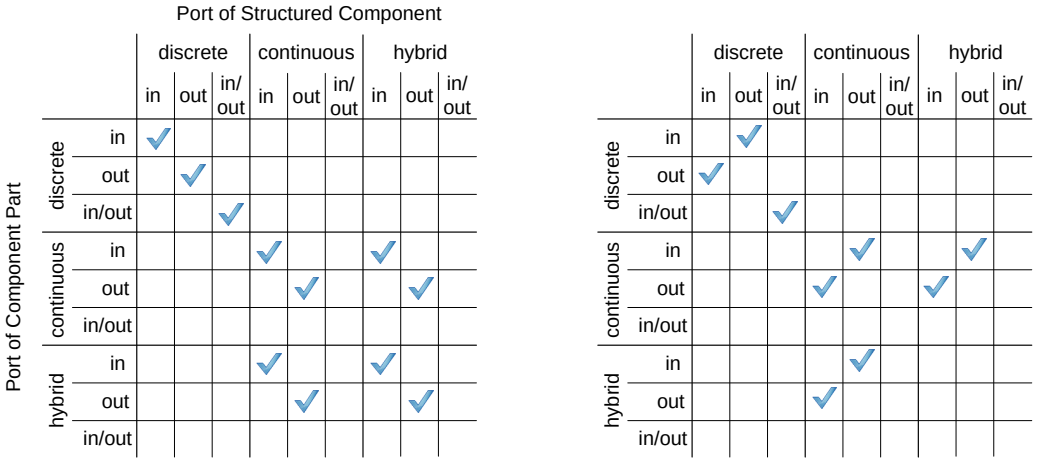
3.1.4 Connectors

Components in our component model are connected via their ports using connectors. In accordance to UML [Gro11c] and to the existing component models by Burmester and Giese as well as Tichy, we distinguish between two kinds of connectors: assembly connectors and delegation connectors. An *assembly connector* connects two ports of component parts inside the same structured component. *Delegation connectors* connect ports of structured components to the ports of component parts of the same structured component.

As an example, consider the structured component RailCabDriveControl shown in Figure 3.6. The discrete ports strategySender of strategy and receiver of convoy are connected by an assembly connector because both are ports of component parts. The ports coordinator of RailCabDriveControl and coordinator of convoy are connected by a delegation connector. We use the concrete syntax defined by Burmester and Giese [GBSO04, GS13] and visualize both kinds of connectors by solid lines.

Whether two ports may be connected by a connector depends on three conditions. First, they need to be structurally compatible. Second, they need to have matching interface specifications. Third, they need to have matching endpoint cardinalities. We define these conditions in detail in the following.

To be structurally compatible, the ports need to be of compatible kinds and have compatible directions. Discrete ports may only be connected to discrete ports. The same holds for RM ports and RE ports. Continuous and hybrid ports may be connected with each other. Broadcast ports may only by delegated to broadcast ports but not connected by assembly connectors (cf. Section 3.4). For delegation connectors, both ports need to have the same direction. For assembly connectors, they need to have inverse directions. Figure 3.8 summarizes the combinations of structurally compatible ports for discrete, hybrid, and continuous ports. Only combinations marked with a checkmark are allowed. We explain how RM ports and RE ports may be connected in more detail in Section 4.1.



(a) Delegation Connectors

(b) Assembly Connectors

Figure 3.8: Structurally Compatible Ports Allowing for a Connector (cf. [BDG+14b])

As the second condition, ports need to have compatible interfaces. For continuous and hybrid ports, we require that they send or receive a signal value with the same data type. For discrete ports, we require that they refine the same role of the same RTCP (delegation connector) or different roles of the same RTCP (assembly connector).

Finally, we require that the endpoint cardinalities of a connector match as defined by Tichy [Tic09, p. 39]. The cardinality of an endpoint is the product of the port cardinality and the component part cardinality. In essence, that means that a single port may only be delegated to a single port of a single part as, e.g., the port member of RailCabDriveControl. A multi port may either be delegated to a multi port as, e.g., coordinator of RailCabDriveControl, or to a single port of a multi part as, e.g., refDistProvider of ConvoyCoordinator (cf. Figure 3.5). The same conditions hold for assembly connectors.

For a structured component, we require that all of its ports are connected by at least one delegation connector to a port of a component part. In addition, we require that all ports of the component parts are attached to at least one connector. The only exception to this rule are continuous ports of component parts if they are directly connected to a hardware component that is not part of the MECHATRONICUML component model. An example of such port is given by the port force of the component part ctrl in RailCabDriveControl. This port is directly connected to the electric motors. In order to prevent unconnected ports in our component model, we visualize such ports as shown in Figure 3.6. We use a graphical symbol that is inspired by a pin in a digital circuit diagram as defined in IEC60617 [IEC96] to represent the hardware pin. Then, we connect this pin by a connector to the continuous port. The notation may be used for both, in-ports and out-ports.

3.1.5 Component Properties

A component may define a set of so-called *component properties*. They enable that a component exposes information about its inner state or configuration to its parent component. A component property has a name and a primitive data type similar to attributes of components as defined by Tichy [Tic09, p. 36] or to attribute controllers in Fractal [BCL+06]. In contrast to attributes, component properties may only be read by the parent component but not modified. We forbid modifications because any change that is applied to the inner state of a component instance needs to be made through one of its ports. This ensures encapsulation and correctness of the compositional verification approach (cf. Chapter 5). In addition, the value of a component property is derived (cf. [SBPM08, p. 108]), i.e., it is computed from the inner state or configuration of a component instance but does not contribute to it.

We present a modeling language, called component story decision diagrams, for expressing component properties based on the current configuration in Section 3.5. In our concrete syntax, we enable to optionally visualize component properties for component instances as illustrated in Figure 3.9, but we do not visualize component properties for components.

3.2 Component Instances

The components introduced in Section 3.1 are instantiated to *component instances* for defining a software architecture of a system. Components may be instantiated multiple times in a system. In particular, each structured component instance creates its own instances for the

components that are embedded by the component parts (cf. [Tic09]). Upon instantiation, the variable parts of the component need to be determined. That means, the number of port instances for each port, the number of embedded component instances for each component part, and the connector instances need to be determined. By default, all ports and component parts are instantiated with minimum cardinality.

Each component instance has a configuration that is defined by its currently instantiated port instances, embedded component instances, and connector instances. If a component is reconfigurable, as in our approach, then a component instance may switch between different configurations at runtime by executing reconfigurations (cf. Section 3.3 and Chapter 4). The current configurations of all component instances in the software architecture of the system define the configuration of the system itself. We refer to this as the *component instance configuration* (CIC) of the system.

As on the type level, we distinguish between atomic component instances and structured component instances. An atomic component instance is typed over an atomic component and executes its behavior specification at runtime. A structured component instance is typed over a structured component and embeds a CIC that contains all embedded component instances and connector instances. Any component instance has a name and, in case that it is embedded in a structured component instance, refers to its component part (cf. [Tic09]).

Figure 3.9 shows the CIC of RailCabDriveControl for a RailCab driving alone. The standaloneRC only embeds four component instances: os of type OperationStrategy, dl of type DriveLogic, vc1 of type VelocityController, and sp of type SpeedSensor. Consequently, the maximum speed for the RailCab is defined by os and provided to dl. dl sets the reference speed for the VelocityController vc1. vc1 controls the force of the electric motor only based on this reference speed and the current speed of the RailCab provided by sp. Consequently, vc1 only uses the feedback controller implemented in StandaloneDrive (cf. Figure 3.7).

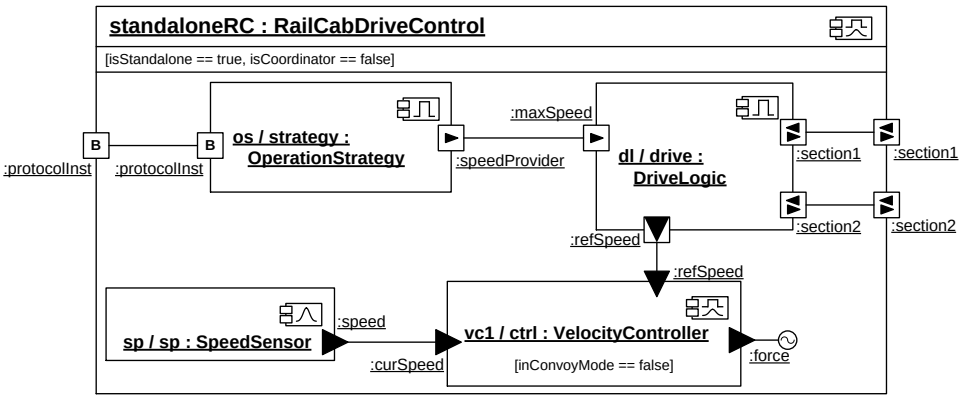


Figure 3.9: Component Instance of Component RailCabDriveControl for a RailCab Driving Alone

The visualization of component properties for component instances is optional. For structured component instances, we visualize component properties in an additional compartment as shown in Figure 3.9. The compartment contains a comma-separated list of component properties with their values in square brackets. In the example, the value of the component property isStandalone is true while the value of the component property isCoordinator is false.

For an atomic component instance or an embedded component instance of a structured component, we visualize component properties in square brackets below the name label of the component instance. In Figure 3.9, the embedded component instance vc1 visualizes the component property inConvoyMode, which is false in the example.

A CIC is syntactically correct under the following conditions. First, the number of port instances of a component instance complies with the cardinality of the corresponding port of the component. For a structured component, we additionally require that the number of embedded component instances for a given component part complies to the cardinality of the component part. Furthermore, port instances of structured component instances may only be delegated to port instances of embedded component instances if the corresponding ports are connected by a delegation connector in the structured component. Analogously, port instances of embedded component instances may only be connected by assembly connector instances if the corresponding ports are connected by an assembly connector in the structured component. If the component instances are not embedded in a structured component, then they may be connected via their port instances using assembly connector instances by applying the same rules that we defined for connectors in Section 3.1.4.

Each discrete port instance is either connected to exactly one other port instance by an assembly connector instance or it is delegated to exactly one port instance of the parent component instance. Port instances of structured component instances have an additional delegation connector instance to a port instance of an embedded component instance. Continuous and hybrid port instances need to fulfill the same properties, but two exceptions apply. A continuous or hybrid out-port may have more than one outgoing connector instance, i.e., the signal value may be send to several other component instances. In addition, continuous in-ports of a structured component instance may be delegated to several embedded component instances. A continuous or hybrid port instance may also have no connector instance if it is directly attached to hardware as, e.g., instances of the port force of VelocityController. In any case, component instances that are embedded in a structured component instance may only be connected by connector instances if the corresponding port types are connected by a connector in the structured component type.

In addition, our component model uses three implicit composite aggregations for component instances. First, a component instance that is embedded in a structured component instance cannot exist without its parent. Second, a port instance cannot exist without its surrounding component instance. Third, a connector instance cannot exist without being attached to exactly two port instances [Tic09, p. 42].

Figure 3.10 shows an instance of ConvoyCoordination that is executed in a coordinator RailCab with one member. It contains an instance cm of type ConvoyManagement and, since the convoy has one member, one instance rg1 of type RefGen. Since rg1 is associated to the first convoy member, it receives the current position of the coordinator RailCab via curPos. Furthermore, cc has instances of the coordinator and refDistProvider multi ports for communicating with the member.

Hirsch [Hir08] and Tichy [Tic09] did not distinguish between single port instances and multi port instances in the concrete syntax. However, a multi port instance has the same structure as a multi role instance (cf. Section 2.4.1), i.e., it contains several subport instances that belong together. Therefore, we propose to visualize the multi port instance by a dashed square that groups its subport instances as shown for the instances of coordinator,

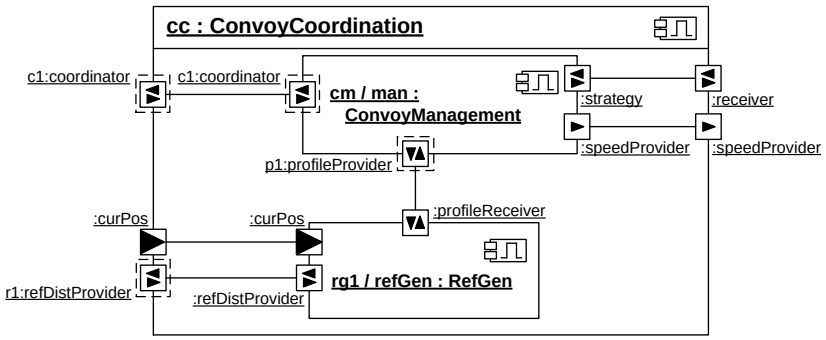


Figure 3.10: Component Instance of Component ConvoyCoordination for a Convoy with 1 Member

refDistProvider, and profileProvider in Figure 3.10. The subport instances are visualized as port instances as before.

We present additional component instances for coordinator RailCabs and member RailCabs in Appendix A.4.

3.3 Modeling Reconfiguration

The existing component models defined two modeling languages for specifying reconfiguration behavior of reconfigurable structured components. The component model by Tichy uses *component story diagrams* (CSDs, [THHO08, Tic09]), which enable a rule-based specification of reconfiguration behavior based on story diagrams (cf. Section 2.3.2). They enable formal, modular, and concise models. In contrast, *hybrid reconfiguration charts* as proposed by Burmester and Giese [GBSO04, BGO06, Bur06] provide a state-based model where each state contains one configuration of a structured component instance. Hybrid reconfiguration charts quickly become very large and unmaintainable if a component instance has several configurations. This is the case, for example, for the component ConvoyCoordination in Figure 3.5 where we have a sequence of RefGen instances that reflects the order of the convoy members on track. As a result, we chose to use CSDs for specifying reconfiguration of structured and atomic component instances in our component model. We refer to Schubert [Sch12] and our technical report [HB14] for a detailed comparison of hybrid reconfiguration charts and CSDs.

In the following, we first introduce CSDs as they have been defined by Tichy (cf. Section 3.3.1). Thereafter, we introduce three extensions to CSDs that we developed as part of this thesis. These are controller exchange nodes (cf. Section 3.3.2), constraints for multi port variables (cf. Section 3.3.3), and CSDs for atomic components (cf. Section 3.3.4). These extensions add features to CSDs that are necessary for specifying reconfiguration behavior in our component model. We illustrate these features based on examples given in concrete syntax. A formalization of CSDs is given by a metamodel [SV06, ch. 4] whose abstract syntax is defined in Appendix D.2. The static semantics has been formalized based on OCL constraints [Gro12]. The operational semantics of CSDs has already been defined by Tichy [Tic09, pp. 71ff] in form of a translational semantics [SK95] by defining a transfor-

mation of CSDs to story diagrams. Our extensions only extend the type system that is used for the story diagrams and do not require a new definition of the operational semantics.

3.3.1 Component Story Diagrams

In our component model, we use CSDs [THHO08, Tic09] for modeling reconfiguration of component instances. Each component contains a set of CSDs that define how instances of the component may be reconfigured at runtime. We define how and when CSDs are executed for a component instance in Chapter 4.

CSDs are based on story diagrams (cf. Section 2.3.2.2) and support the same constructs for specifying control flow including a set of input and output parameters. The story nodes of a CSD, however, contain *component story patterns* instead of story patterns [Tic09].

A component story pattern defines the modification of a component instance and, in case of a structured component instance, its embedded CIC. We use the components that are defined in our component model as a type graph to type the variables of the component story pattern. Then, all variables and links of the component story pattern are typed by the components, ports, and connectors that are defined by the component model. Thereby we can ensure that component instances remain syntactically correct after applying a component story pattern. In particular, we can ensure that a component story pattern can only be executed if its modifications do not violate the cardinalities of ports and component parts.

Each component story pattern contains exactly one this component variable. The this variable is typed by the component that contains the corresponding CSD. At runtime, the this variable is automatically bound to the component instance that invoked the CSD on itself. Thus, it is a implicit input parameter of any CSD [Tic09].

Figure 3.11 shows a CSD becomeMember of the component RailCabDriveControl. The CSD reconfigures an instance of RailCabDriveControl of a RailCab driving alone (cf. Figure 3.9) to an instance of a RailCab driving as a member of a convoy (cf. Figure A.31).

The CSD has two story nodes. In the first story node, we match the embedded component instances of types OperationStategy, DriveLogic, and VelocityController. We destroy the assembly connector instance between os and dl. In addition, we invoke the reconfiguration applyMemberStrategy on os that destroys the speedProvider port instance. We explain this CSD in more detail in Section 3.3.4. In addition, we invoke the reconfiguration switchToConvoy on vc that reconfigures the feedback controllers for driving as a convoy member. We introduce this CSD in more detail in Section 3.3.2. In the second story node, we create an instance of MemberControl. In addition, we create an instance of DistanceSensor and connect it to vc by an assembly connector instance. Finally, we create port instances of member and refDistReceiver on this and connect all port instances of mc.

A CSD may specify invocations of further CSDs on embedded component instances. The invocation is directly attached to the corresponding component variable [Tic09, p. 62] as shown in the first story node of the CSD becomeMember in Figure 3.11. We define how such invocations are executed with respect to the component hierarchy in Chapter 4.

In our component model, we restrict CSDs such that they respect component encapsulation. In particular, we forbid that a CSD directly creates or destroys port instances of embedded component instances as it is allowed by Tichy [Tic09, p. 55]. Such port instances may only be created by the embedded component instance itself. The corresponding CSD

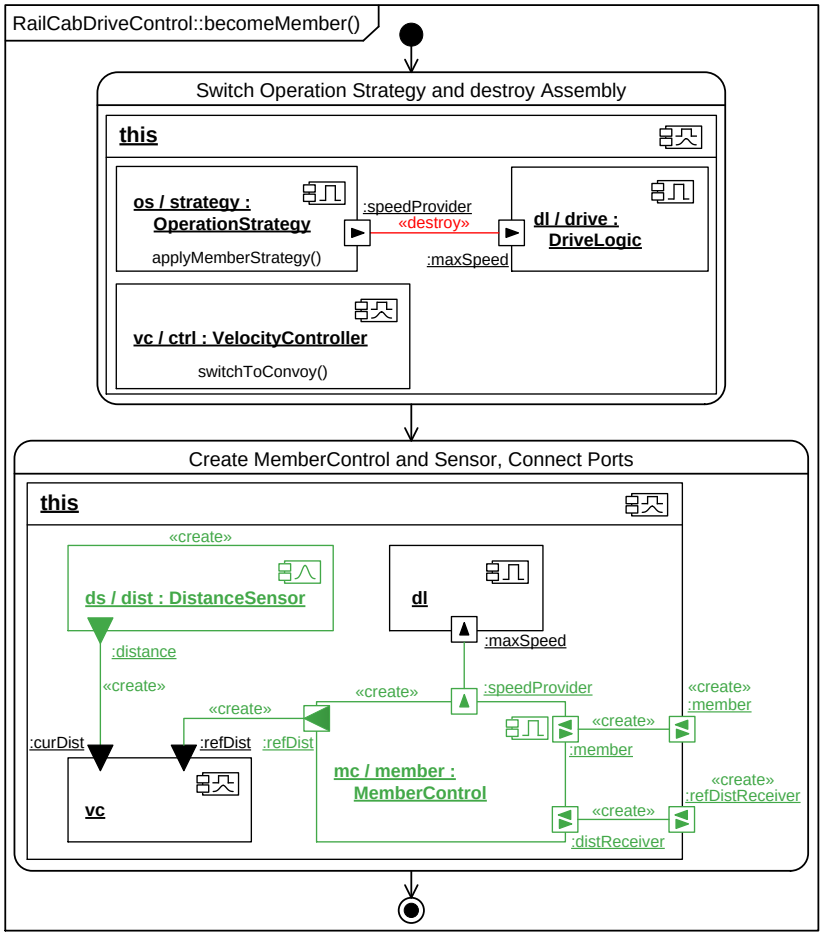


Figure 3.11: CSD for Component RailCabDriveControl that Reconfigures the Component Instance to Serve as a Member

that creates the port instance needs to be invoked on the embedded component instance as shown in Figure 3.11.

In accordance to Tichy, a component may define one or more constructor CSDs. A constructor CSD defines how instances of the component are initialized upon instantiation. Then, a component variable with stereotype «`create`» may invoke a constructor [Tic09, p. 62]. In addition, every component defines an implicit constructor that instantiates all ports and embedded components according to their minimum cardinality. The implicit constructor is always used if no explicit constructor is invoked for a coponent variable with stereotype «`create`». In Figure 3.11, we used implicit constructors for both, ds and mc. We provide an example of an explicit constructor in Appendix A.6.2.

3.3.2 Controller Exchange Nodes

The component model by Tichy does not distinguish between different kinds of components [Tic09]. Consequently, it does not enable to use fading functions, which are typically required when replacing continuous component instances as explained in Section 3.1.2.3.

As a solution, Schubert [Sch12] introduced *controller exchange nodes* for enabling the reconfiguration of continuous components. A controller exchange node is a special kind of story node that enables safe execution of fading functions. It has a fixed structure that consists of exactly three component variables. Two of which reference continuous components where one is destroyed and one is created. The third component variable refers to the fading component that is connected to the two continuous components. The component variable referring to the fading component additionally specifies which fading function needs to be executed.

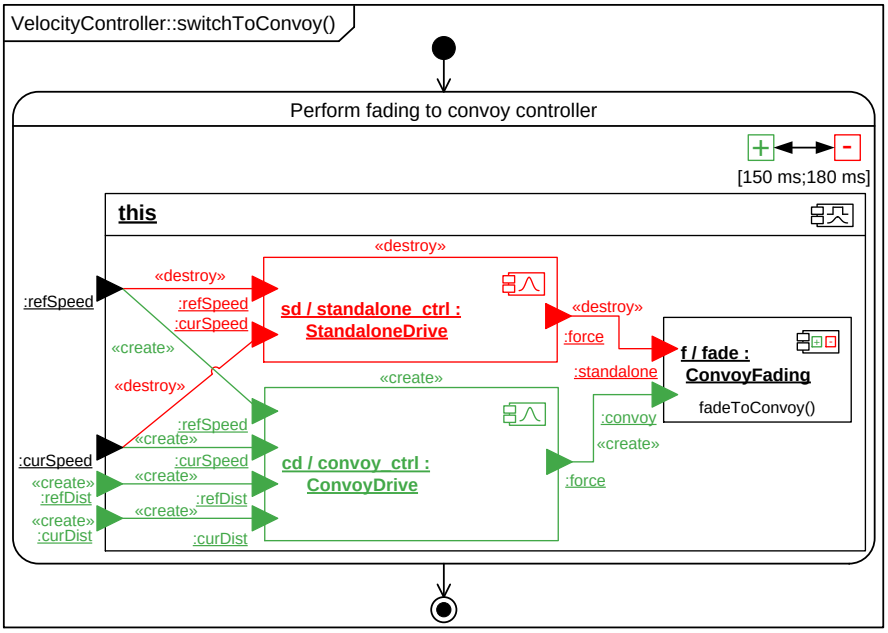


Figure 3.12: CSD for Component VelocityController that Reconfigures the Component Instance to Serve as a Member

Figure 3.12 shows the CSD switchToConvoy that uses a controller exchange node. It is invoked by becomeMember and reconfigures an instance of VelocityController such that it uses an instance of ConvoyDrive instead of StandaloneDrive. Consequently, it destroys the latter and creates a new instance of the former. Since both continuous components provide a signal force, we need to use the fading component ConvoyFading to fade between both signals. In the controller exchange node of switchToConvoy, we therefore select the fading function fadeToConvoy for the reconfiguration as specified within the fading component variable. As indicated in the upper right corner of the controller exchange node, the fading takes between 150 ms and 180 ms.

3.3.3 Constraints for Multi Port Variables

Multi ports are ordered, i.e., the subport instances are arranged in a sequence as it has been defined for multi roles (cf. Section 2.4.1). Up to now, this order cannot be used in component story pattern, for example, for creating a subport instance as a successor of another subport instance.

As an example, consider the component ConvoyCoordinator shown in Figure 3.5 on Page 42. The subport instances of an instance of the multi port coordinator shall have the same order as the corresponding convoy members on track. Thus, if a new member joins the convoy at a particular position, we need to insert a subport instance at the same position into the multi port instance.

Therefore, we extend component story patterns by constraints for multi port variables that refer to the order of the subport instances [Hei14]. They are inspired by so-called link constraints of story diagrams [vDHP+12a]. In accordance to these link constraints, we distinguish between *multi port position constraints* and *multi port order constraints*. Both of which are illustrated in Figure 3.13.

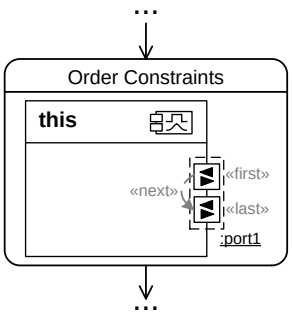


Figure 3.13: Order Constraints for Multi Port Variables

A multi port position constraint enables to refer to the first (or last) subport instance of a multi port instance. In our concrete syntax, we visualize it by attaching a stereotype «first» (or «last») to the corresponding subport variable. In Figure 3.13, the upper subport variable matches the first subport instance while the lower subport variable matches the last subport instance.

A multi port order constraint enables to refer to the relative order of the subport instances. We enable to define that a subport instance is a direct successor (or predecessor) of another subport instance. In our concrete syntax, we visualize these constraints by a dashed arrow annotated with the stereotype «next» (or «prev») to denote that the target subport instance is a direct successor (or direct predecessor) of the source subport instance. In Figure 3.13, the lower subport instance has to be a direct successor of the upper subport instance of the multi port instance of type port1 for successfully matching the component story pattern.

Multi port position constraints and multi port order constraints may be part of the LHS or RHS of the component story pattern. A multi port position constraint is part of the LHS if it is attached to a subport variable with no stereotype or with stereotype «destroy». In this case, the matched subport variable needs to fulfill the multi port position constraint for a successful matching. A multi port position constraint is part of the RHS if it is attached to a subport variable with stereotype «create». In this case, the created subport instance is inserted at the specified position into the multi port, i.e., either at first or last position. A multi port order constraint is part of the RHS if at least one of the attached subport variables carries a «create» stereotype. It is part of the LHS in all other cases. It is not allowed to connect a subport variable stereotyped with «create» and a subport variable stereotyped with «destroy» by a multi port order constraint.

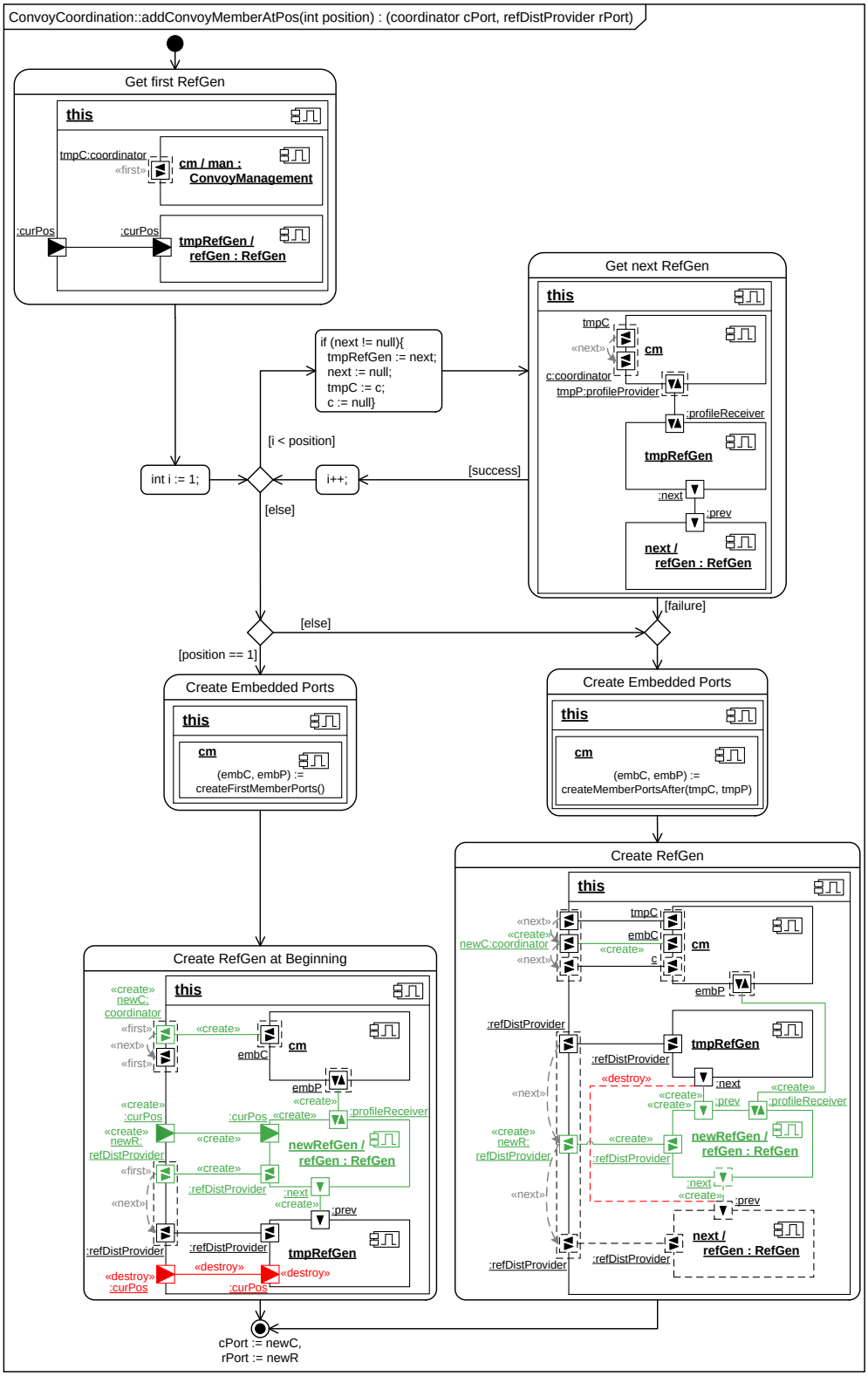


Figure 3.14: CSD for Adding a Convoy Member (cf. [Tic09, Sch12])

Figure 3.14 shows an example of a complex CSD of the component ConvoyCoordination that implements the aforementioned use case of adding a new convoy member at a specific position of the convoy. The CSD takes a position as its input and returns the subport instances that have been created for the multi ports coordinator and refDistProvider of ConvoyCoordination. In addition to the subport instances, the CSD also creates an instance of the RefGen multi part.

The behavior of the CSD addConvoyMemberAtPosition is as follows. The first story node matches the first RefGen instance which is the only one having an instance of the curPos port. Matching this story node will always succeed because the corresponding component parts are both mandatory. Thereafter, the statement node initializes a counter variable i that is used for iterating over the list of RefGen instances until the instance at the position given as parameter has been found. The iteration is performed via the story node and the two statement nodes in the upper right corner. The component story pattern in the story node uses a multi port order constraint for iterating over the subport instances of coordinator.

If the correct position has been found, the story node in the lower left corner inserts an instance of refGen at the beginning of the list, while the story node in the lower right inserts an instance of refGen at any other position. Along with the instance of RefGen, we create subport instances for the multi ports coordinator and refDistProvider of this and insert them at the corresponding positions. In the story node Create RefGen at Beginning, we use the «first» stereotype two times within the same multi port variable. This is allowed because one is part of the LHS for matching the previously first subport instance while the other one is part of the RHS. Finally, the final node assigns the created subport instances newC of type coordinator and newR of type refDistProvider to the output parameters cPort and rPort, respectively.

3.3.4 Reconfiguration of Atomic Components

In our component model, we use CSDs for reconfiguring atomic components as well. Tichy neither explicitly defined nor restricted CSDs in that way [Tic09]. The only difference to a CSD of a structured component is that we visualize the this component variable as an atomic component. Then, the this component variable may only contain port variables but no embedded component variables (cf. [BDG+14b]).

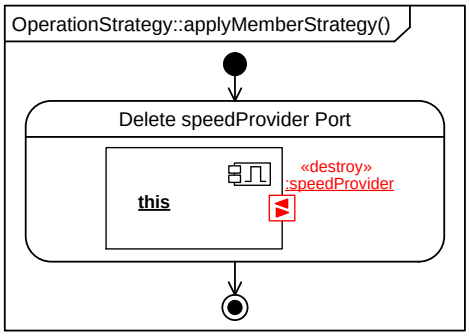


Figure 3.15: CSD for Component OperationStrategy that Reconfigures the Ports for Being Member

Figure 3.15 shows the CSD applyMemberStrategy that is invoked in the first story node of becomeMember shown in Figure 3.11. OperationStrategy is an atomic component and, therefore, the this variable has no embedded component variables. The CSD deletes the speedProvider port instance because the reference speed of a member is defined by the coordinator and not by its own operation strategy.

3.4 Instantiating Real-Time Coordination Protocols on System Level

RTCPs on the system level define the communication between different AMS while they collaborate in an NMS. Since NMS are virtual, there does not exist a component that contains both AMS and that may instantiate an RTCP between the AMS. Consequently, the instantiation cannot be described by CSDs, but the AMS need to agree on instantiating a particular RTCP via message-based communication. This, however, requires at least one of the AMS to know about the existence of the other AMS. Then, one of the AMS may initiate the communication for agreeing on the instantiation. This communication, however, cannot be handled by the discrete ports and connectors introduced in Section 3.1 because their instances already require a connector with a RTCP.

For solving this problem, we need to relax the strict requirement of MECHATRONICUML that all communication between AMS is exclusively handled by RTCPs with single-cast connectors [GTB+03, EHH+13]. In particular, we use broadcast ports as introduced in Section 3.1.1.3 that enable for broadcast communication. Whenever an AMS sends a message via a broadcast port, the message is received by all other broadcast ports "in reach" that can process this message. Which broadcast ports are in reach depends on the spatial distribution of the AMS as well as the transmission medium. In general, it is not known at design time which ports will receive a message and which will not. In order to retain the safety guarantees provided by the use of RTCPs, we only allow the use of broadcast ports for two special purposes. First, for gaining knowledge about the existence of other systems and, second, for instantiating one particular RTCP called ProtocolInstantiation. This RTCP then enables to instantiate further RTCPs. No further broadcast ports are allowed in a MECHATRONICUML model.

Following the terminology of Baresi et. al. [BDNG06], gaining knowledge about other systems is only required in so-called *open-world* scenarios. In an open-world scenario, systems do not know each other in advance and the possible communication partners change frequently over time. For example, RailCabs move along the track system and do not know when or where they meet which other RailCab. In this case, we may need to use a broadcast port that executes a so-called discovery protocol [NNSS07] that detects and gathers knowledge about the systems in the environment. This information needs to be stored in an environment model that may be used by the cognitive operator of the OCM for deciding which other systems are suitable for which cooperation. We present a simple discovery protocol and environment model in Appendix A.2.1, but we do not consider this use case in detail as part of this thesis.

In the following, we illustrate how we use a broadcast port for instantiating the RTCP ProtocolInstantiation for two AMS (Section 3.4.1). Thereafter, we show how to use the RTCP ProtocolInstantiation for instantiating further RTCPs (Section 3.4.2).

3.4.1 Instantiating the RTCP ProtocolInstantiation

An AMS may use its protocolInst broadcast port for contacting another AMS for instantiating a connector including the ProtocolInstantiation RTCP. ProtocolInstantiation is the only RTCP that may be instantiated via broadcast communication. All other RTCPs need to be instantiated via ProtocolInstantiation or as part of another, user-defined RTCP.

Assumptions For instantiating the RTCP ProtocolInstantiation, the AMS that initiates the instantiation needs to know the other system in its environment model. Based on this, we apply the following assumptions to the instantiation process:

1. Each AMS has a unique ID that is known to the RTSC of the broadcast port.
2. There may exist different versions of this message exchange that only differ in their timing constraints as, e.g., timeouts, where each version has a unique identifier. The broadcast port knows the ID of the version that it implements.
3. The IDs are represented using a data type that can be sent as a parameter of a message.
4. No message loss occurs during the interaction.
5. No eavesdropper tries to compromise or prevent the connection setup.

Since the different versions only differ in their timing constraints, the message exchange for instantiating the RTCP ProtocolInstantiation may be used in different systems without modification.

Behavior Figure 3.16 shows the message exchange for instantiating the RTCP ProtocolInstantiation as a modal sequence diagram (MSD, [HM08a]). For better readability of the figure, we omitted all timing constraints.

In the MSD, sys1 initiates the instantiation of ProtocolInstantiation with sys2. It sends a connectionRequest via the broadcast port that includes the ID of sys2 as the first parameter and its own ID as the second parameter. If an AMS receives such message, it checks whether its ID is contained in the first parameter. If so, it evaluates the message and checks whether another instance of the port implementing the RTCP ProtocolInstantiation may be created. If not, it answers with a connectionDenial. If the port instance can be created, it sends a connectionApproval. sys1 then checks whether the connectionApproval has been sent by sys2 and continues with the instantiation by sending a startProtocolInstantiation message. This message contains the version ID in addition to the IDs of the two systems. After receiving the startProtocolInstantiation message, sys2 checks whether it supports the desired version. If not, it sends an abortProtocolInstantiation message and the instantiation fails. If it supports the desired version, sys2 creates a port instance that implements the RTCP ProtocolInstantiation. Then, it answers with a confirmProtocolInstantiation message that includes the created port instance. If sys1 receives this message, it creates a port instance including the connector instance to the port instance contained in the message. Finally, it sends a completedProtocolInstantiation message to sys2 including the port instance it just created. Finally, sys2 receives this message and creates the connector instance itself to the port instance provided by sys1. At this point, both systems successfully instantiated the RTCP ProtocolInstantiation and may continue to instantiate further RTCPs.

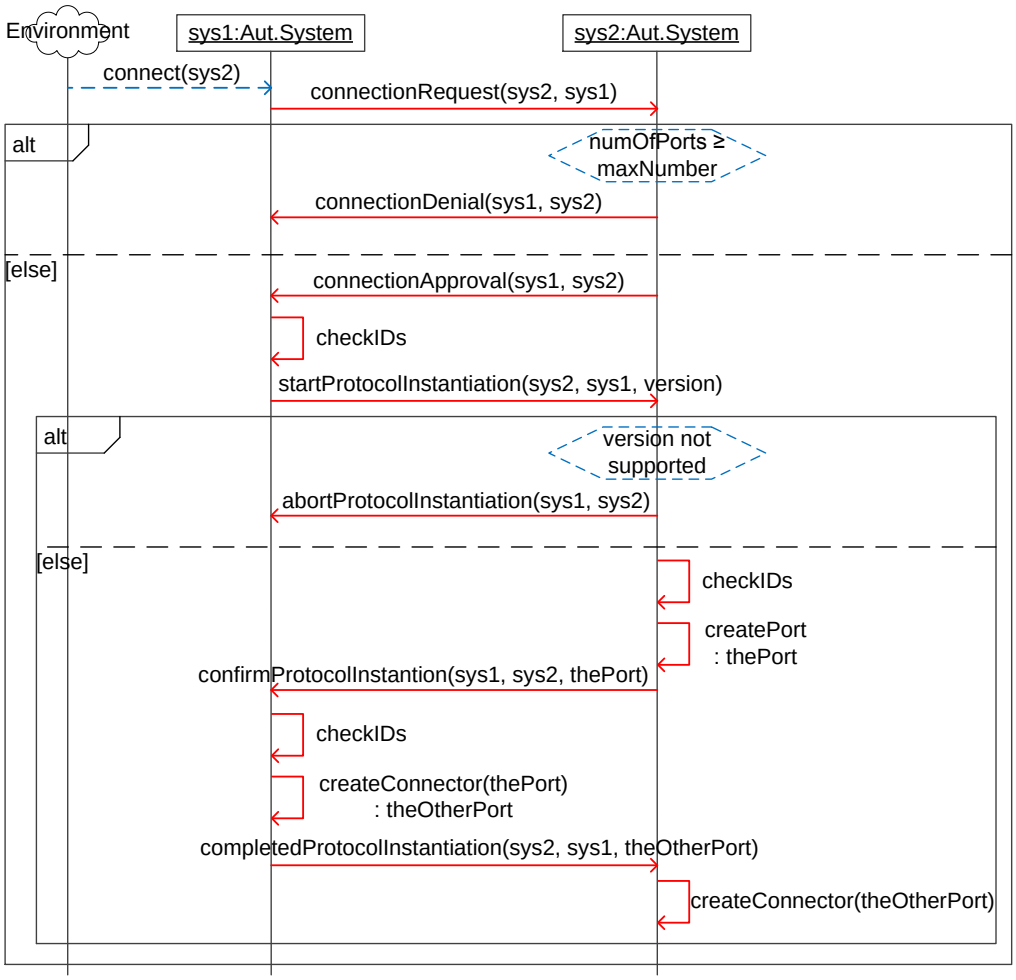


Figure 3.16: MSD Specifying the Broadcast Message Exchange for Instantiating the RTCP ProtocolInstantiation

Please note that connector instances between two AMS are virtual, i.e., there exists no shared connector instance object between the two systems. Instead, each of the AMS knows the other one as an external system and maintains its own connector instance object to that external system. Therefore, the connector is created two times: once in each system.

We specified the behavior of the broadcast port by a RTSC and verified it using the UPPAAL model checker [BDL+06b]. The RTSC is contained in Appendix A.2.2. The behavior is free of deadlocks and ensures that if sys1 created the port instance, then also sys2 created the port instance. Furthermore, it ensures that a third system may not accidentally create a port instance.

3.4.2 The RTCP ProtocolInstantiation

The RTCP ProtocolInstantiation, whose declaration is shown in Figure 3.17, is a special purpose RTCP that is intended to be used in combination with the protocolInst broadcast port. It has two roles requestor and requestee. The system that initiated the instantiation of ProtocolIn-

stantiation (cf. Section 3.4.1) is always the requestor. It requests the requestee to instantiate a particular role of a RTCP.

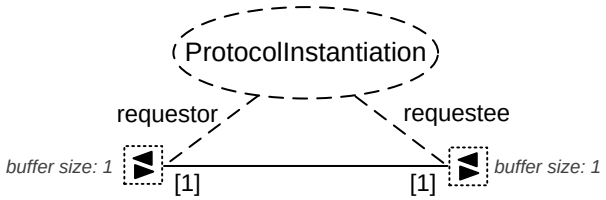


Figure 3.17: Declaration of the RTCP ProtocolInstantiation

Assumptions The specification of ProtocolInstantiation as described below underlies the following assumptions:

1. Each RTCP has a unique ID.
2. Each role of a RTCP has a unique ID within its RTCP.
3. The IDs are represented using a data type that can be sent as a parameter of a message.
4. The component instance that executes the requestor role at one of its port instances is able to instantiate a port that implements the other role of the requested RTCP.
5. No message loss occurs during the interaction.

Behavior The message exchange between the two roles of the RTCP ProtocolInstantiation is defined by the MSD in Figure 3.18.

The interaction starts with a request that is sent by the requestor. It requests the requestee to instantiate a particular role (roleID) of the RTCP identified by the protocolID. The requestee checks whether it supports the requested protocol and role. If so, it requests its environment to create a port instance that implements the requested role. If not, it answers with a protocolNotSupported message and the instantiation fails. In the context of this RTCP, the environment is the atomic component instance that contains the port instance executing one of its roles. This atomic component instance then either creates the port instance itself or it triggers a reconfiguration using our approach presented in Chapter 4.

If the requestee requested to create a port instance, the environment either answers with success or failed depending on whether the port could be created successfully. In the latter case, the requestee sends a declineInstantiation message to the requestor and the instantiation fails. In the former case, the requestor sends a confirmInstantiation message to the requestor. This message includes the port instance that was created by the requestee. The requestor then advises its environment to create a port instance as well and provides the port instance of the requestor as a parameter. According to our assumptions, the creation of the port instance succeeds. The requestor then sends a finalize message including the port instance that it created. Then, the requestee passes this port instance to its environment to create the connector instance, which finishes the connection setup. The environment acknowledges that by sending finished. Then the requestee sends completed to the requestor which completes the instantiation.

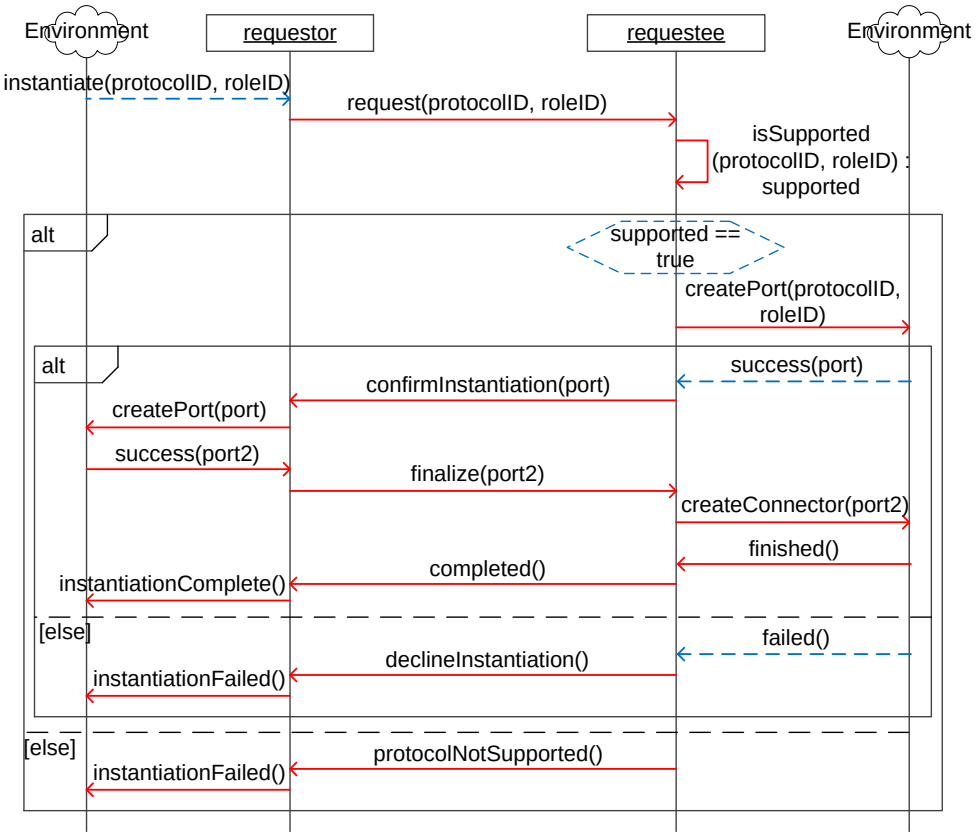


Figure 3.18: MSD Specifying the Message Exchange for Instantiating an RTCP

We present RTSCs implementing the behavior for both roles in Appendix A.2.3. The RTSCs have been specified and verified using a connector delay of 25 ms. The verification has been carried out using UPPAAL [BDL+06b]. The RTSCs guarantee that the protocol is deadlock free and that either both roles succeed or both roles fail.

Usage The message exchange shown in Figure 3.18 is independent of the RTCPs that an AMS wants to instantiate. They only appear as parameters of messages. If the developer assigns the requestee role to a port of a component of the AMS, the developer needs to implement the isSupported function. In addition, the port that implements requestor needs to be integrated with the behavior that decides when to ask another system to start collaborating. Both ports need to be integrated with the reconfiguration behavior such that the necessary reconfigurations may be triggered.

Although the purpose of ProtocolInstantiation is instantiating RTCPs, we do not want to instantiate all RTCPs that an AMS supports via this protocol. In general, only RTCPs with a predefined role assignment should be instantiated via ProtocolInstantiation. As an example, consider the RTCPs that are necessary for driving as part of a convoy. Driving in a convoy includes the election of a coordinator and complex negotiation about the speed for the convoy, maximum accelerations and decelerations, and the ongoing route. As a result, we do not want to instantiate these RTCPs, but only a RTCP that can be used to elect a coordinator for

the convoy that can manage the necessary negotiations. In our example, we only want to instantiate the RTCP ConvoyEntry that is refined by the port peer of RailCabDriveControl shown in Figure 3.6. We introduce this RTCP in Appendix A.1.1.

3.5 Modeling Component Properties by Architectural Constraints

In this thesis, we define component properties (cf. Section 3.1.5) by architectural constraints. An architectural constraint defines a condition on the configurations of a (structured) component instance [GMW00]. In addition, we enable to define such component properties as *invariants*. An invariant needs to evaluate to true for any configuration of the component, whereas other component properties may evaluate to false for some configurations. Invariants enable to define valid configurations of a component, which we exploit for verifying the correctness of the reconfiguration behavior in Section 4.5.1.

We define a new language called *component story decision diagrams* (component SDDs, [Hei14]) for modeling architectural constraints. Component SDDs combine the constraint specification of story decision diagrams (SDDs, [KG07, Sta08]) with component story patterns (cf. Section 3.3.1) for referring to components. They have a name and always apply to one component of our component model. This component defines the type of the this variable which is used in the component story patterns.

In the following, we give a brief overview of component SDDs using examples in concrete syntax and an informal description of their semantics. Component SDDs have been formalized by a metamodel [SV06, ch. 4] whose abstract syntax is given in Appendix D.2.4. Their operational semantics has been formally defined in form of a translational semantics [SK95] by defining a transformation to SDDs. We describe this transformation in our technical report [Hei14].

Component SDDs use the same syntactic elements as SDDs [KG07]. They consist of one initial node, a set of pattern nodes and leaf nodes, and a set of directed edges connecting the nodes. The initial node denotes the starting point for evaluating the component SDD. A pattern node contains a component story pattern. Leaf nodes mark the end for evaluating the component SDD. There exist two kinds of leaf nodes: (0)-nodes and (1)-nodes. The nodes are connected by two kinds of edges: then-edges (also called high-edges) and else-edges (also called low-edges). The initial node has exactly one outgoing then-edge and no incoming edge. Each pattern node has exactly one outgoing then-edge and one outgoing else-edge, while leaf nodes have no outgoing edges. The number of incoming edges is not restricted for pattern nodes and leaf nodes, but nodes and edges need to form a directed acyclic graph [Sta08, p. 60].

Figure 3.19 shows an example of a simple component SDD called isCoordinator. The initial node is visualized as a filled black circle. It shows the name of the component and the name of the component SDD. (1)-nodes are visualized as green circles containing a 1, while (0)-nodes are visualized as red circles containing a 0. Pattern nodes are visualized as squares and visualize the component story pattern that they contain. It the upper left corner, they have a label that enumerates all unbound variables of the component story pattern. In our example, the component story pattern defines that RailCabDriveControl embeds instances of

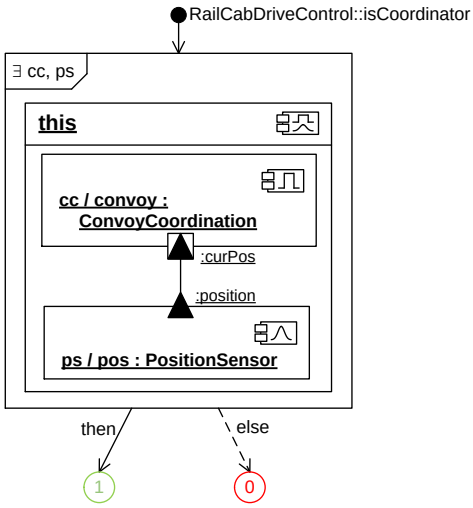


Figure 3.19: Component SDD isCoordinator for Component RailCabDriveControl

ConvoyCoordination and PositionSensor. The pattern node is connected by a solid then-edge to the (1)-node and by dashed else-edge to the (0)-node.

The semantics of a component SDD is defined analogously toSDDs [Sta08]. The evaluation starts at the initial node with a variable binding that only assigns the this variable of the component story pattern to the component instance on which the component SDD should be evaluated. In our example, this is an instance of RailCabDriveControl. This variable binding is passed to the first pattern node. If the contained component story pattern can be matched on RailCabDriveControl, then the variable binding is extended with bindings for all unbound variables (cc and ps) of the component story pattern and passed down the then-edge. Otherwise, the variable binding remains unchanged and is passed down the else-edge. If the evaluation ends at a (1)-node, then the component SDD is fulfilled. If the evaluation ends at a (0)-node, then the component SDD is not fulfilled. Thus, the component SDD in Figure 3.19 is fulfilled for an instance of RailCabDriveControl if it embeds instances of ConvoyCoordination and PositionSensor that are connected by an assembly connector instance.

Since component SDDs are a constraint language, the component story patterns used in pattern nodes may not use «`create`» or «`destroy`» annotations. In addition, we do not allow to use optional or negative variables. Optional variables have no influence on a successful matching and, thus, cannot be referenced in subsequent nodes. Thus, they have no semantics in component SDDs. In accordance to Stallmann [Sta08], we do not use negative variables either but express negation by switching the then- and else-edges. As an example, consider the component SDD isStandalone shown in Figure 3.20.

The component SDD isStandalone denotes that RailCabDriveControl neither operates as a coordinator nor as a member of a convoy. It is fulfilled if the component story patterns in both pattern nodes cannot be matched on an instance of RailCabDriveControl. If one of the component story pattern could be matched, the evaluation would proceed via the then-edge and end at a (0)-node.

The pattern nodes that are used in Figures 3.19 and 3.20 are so-called existential pattern nodes. A component SDD containing only existential pattern nodes is fulfilled if and only if

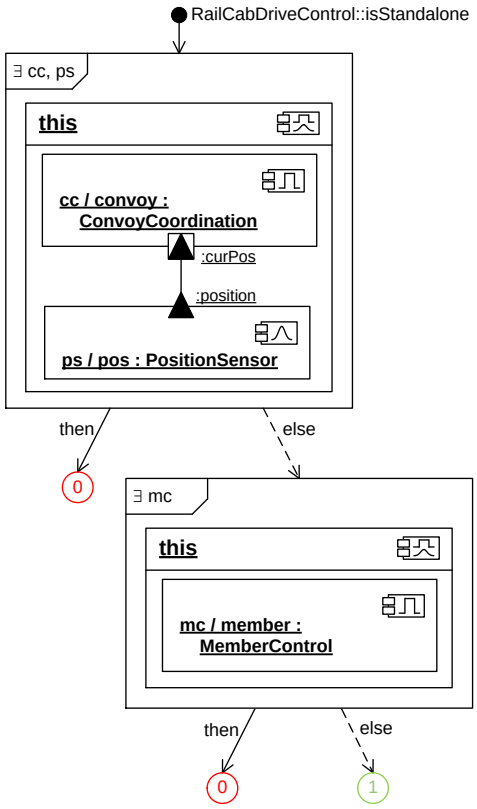


Figure 3.20: Component SDD isStandalone for Component RailCabDriveControl

there exists one variable binding for the pattern nodes such that the evaluation terminates at a (1)-node. In addition, there exist so-called universal pattern nodes as shown in Figure 3.21. They are visualized with a cascaded border line [Sta08] in accordance to CSDs. If a component SDD contains a universal pattern node, then the evaluation needs to terminate at a (1)-node for any matching that can be obtained for the universal pattern node.

The invariant component SDD convoyOrder shown in Figure 3.21 specifies that any two successive subport instances of refDistProvider are delegated to successive subport instances of ConvoyCoordination. This ensures that reference speeds and distances can always be distributed within the convoy in the right order. In the component SDD, the first pattern node matches an instance of ConvoyCoordination that is embedded in RailCabDriveControl. The second pattern node is a universal pattern node that matches all pairs of successive refDistProvider port instances. For any match that may be obtained for this pattern node, the third pattern node needs to be matched as well such that the execution terminates at the (1)-node. If there exists no matching for the universal pattern node, then the component SDD is fulfilled. For this reason, we only visualize one outgoing then-edge for an universal pattern node as a shorthand notation [Sta08, p. 63].

A component SDD may require that an embedded component instance has a component property with a particular value. This component property may be specified, again, using a component SDD. This enables to connect architectural constraints through the different hierarchy levels of the component model. In Figure 3.21, the part variable cc in the universal

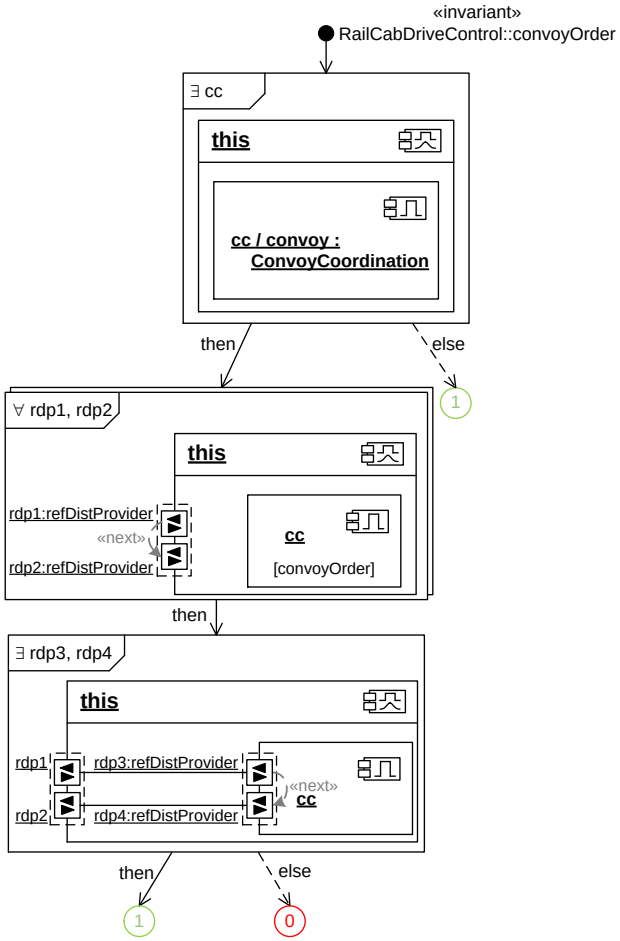


Figure 3.21: Component SDD convoyOrder for Component RailCabDriveControl

pattern node requires that the component property convoyOrder is true for the component instance matched to cc.

3.6 Implementation

The concepts presented in this chapter have been implemented as part of the MECHATRONICUML Tool Suite[1] [DGB+14]. We have built a metamodel (cf. Appendix D) and several diagram editors for creating models based on our component model. In contrast to the previous version of the tool, called Fujaba Real-Time Tool Suite [PTH+10], the metamodel has been developed using the Eclipse Modeling Framework (EMF, [SBPM08]). The static semantics has been completely encoded in the metamodel using OCL [Gro12]. Figure 3.22 provides a conceptual overview of the Eclipse plugins that have been created.

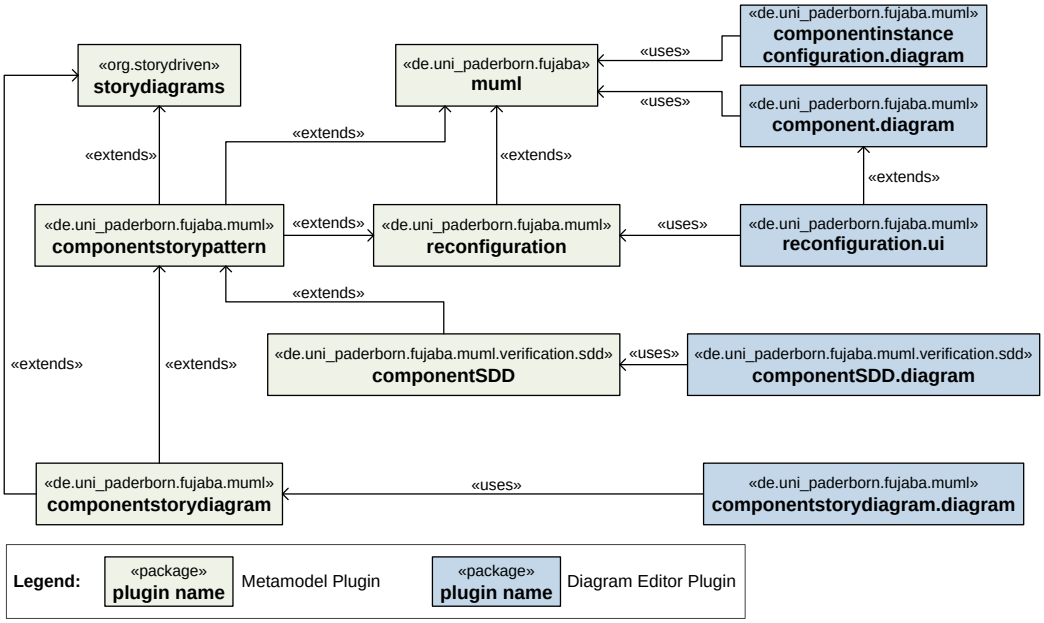


Figure 3.22: Plugins Implementing the Concepts of the Component Model

The plugin muml contains the core metamodel of MECHATRONICUML. It enables to specify components and CICs including RTCPs and RTSCs. In muml, we only support non-reconfigurable components in order to have a core language that can be used for simple, non-adaptive systems. Reconfigurable components are modeled in the plugin reconfiguration. This plugin also contains the metamodel for broadcast ports. The metamodel for CSDs has been split into two plugins. The first one, called componentstorypattern, enables to specify component story patterns based on reconfigurable components. The componentstorydiagram plugin adds the story nodes that are special for CSDs. The metamodel for CSDs extends the story diagram metamodel [HRvD+11] contained in the plugin storydiagrams and reuses as much of the implementation as possible. In addition, we reuse the componentstorypattern metamodel for specifying component SDDs. The corresponding plugin componentSDD only

[1] https://trac.cs.upb.de/mechatronicuml

defines the nodes and edges of component SDDs and uses the same component story pattern implementation as CSDs. We present class diagrams for our metamodels in Appendix D.

Based on the metamodels, we developed four diagram editors using the graphical modeling framework (GMF, [Gro09]). In particular, we created editors for specifying components, CICs, CSDs, and component SDDs. The reconfiguration.ui plugin extends the component editor such that it may also be used for modeling reconfigurable components.

At present, our implementation does not yet support the specification component properties for a component. Component SDDs are currently only used as part of our transactional reconfiguration approach as discussed in Chapter 4. Statement nodes of CSDs are not yet supported as well.

3.7 Related Work

This section relates our new component model for MECHATRONICUML to other approaches for defining software architectures. First, we compare it to other software component models (Section 3.7.1). Second, we relate it to architecture description languages (ADLs, [MT00]) for self-adaptive systems (Section 3.7.2). Finally, we discuss related works regarding the specification of architectural constraints for a component-based system (Section 3.7.3).

3.7.1 Software Component Models

The surveys by Lau [LW07] and Crnković et al. [CSVC11] review different kinds of component models. They distinguish between general purpose component models as, for example, CORBA [Gro11a] and EJB [Ora13], and specialized component models for particular domains. The latter usually address business information systems or embedded real-time systems. In this section, we focus primarily on component models for embedded real-time systems and on component models that support runtime reconfiguration.

Hošek et al. [HPB+10] surveyed component models for embedded real-time systems. Only few of which support runtime reconfiguration including SOFA-HI, MyCCM-HI, ProCom, BlueArX, and AUTOSAR. All of these component models are restricted to mode changes [HKMU06] where a component instance moves from one implementation to another one. SOFA-HI [PWT+08, PKH+11] is an extension of the SOFA 2.0 [HP06, HB07] component model for real-time systems. It enables to specify hierarchical components that are considered to be implemented manually in C. In contrast to SOFA 2.0, reconfigurations in SOFA-HI cannot change connectors at runtime [PWT+08, HPB+10]. MyCCM-HI [BFHP09] extends the OMG CORBA Component Model (CCM, [Gro11a]) for specifying adaptive embedded systems with a textual syntax. It enables a detailed specification of tasks and their activation but does not define how the behavior of tasks is implemented. A mode change of a structured component may reconfigure connectors. The BlueArX component model by Bosch [KKH+08, KRKH09] also provides the specification of hierarchical components with mode changes. In BlueArX, the behavior of a component is defined by signal flows. In addition, each component defines a set of tasks including a scheduling of these tasks. A mode switch may either change the signal flow inside a component or it may change the task scheduling. Mode changes cannot be composed hierarchically. The ProCom component model [VSC+09] has recently been extended to support hierarchical

reconfiguration based on mode changes as well [HQCH13]. We discuss this approach in detail in Section 4.8 along with our transactional reconfiguration approach. For automotive systems, the AUTOSAR standard [FMB+09] defines a component model for specifying hierarchical components. AUTOSAR does not specify how application software components are implemented [AUT14c]. As of version 4.0, AUTOSAR supports modes [AUT14a] and a timing specification [AUT14b]. Modes only enable to activate and deactivate trigger events in atomic software components thereby changing their behavior. The timing specification enables to define periods and orders for events as well as end-to-end deadlines for chains of events. All of these component models have in common that they do not provide means to specify and verify asynchronous message-based communication with real-time properties and that they only provide limited reconfiguration capabilities. In contrast, CSDs of MECHATRONICUML provide a more powerful and flexible specification of reconfigurations that includes control flow and reconfigurations across different levels of hierarchy (see also Chapter 4).

Fractal [BCL+06, LLC10] provides the definition of hierarchical components including runtime reconfiguration of structured components. Each component consists of a membrane and a content area. The content area embeds other components while the membrane contains so-called controllers that enable introspection and reconfiguration. Although Fractal provides a C-implementation called Think [AHJ+09], it does not provide the ability to specify clock-based real-time properties for components or to verify the functional behavior or the reconfiguration specification.

The DEECo component model [BGH+13] provides non-hierarchical components for soft real-time systems based on ensembles. While being in an ensemble, components may communicate and exchange knowledge. The communication, however, is not explicitly modeled in their approach. Components declaratively specify conditions for being part of an ensemble and a shared runtime framework automatically constructs and dissolves ensembles based on these conditions. De Nicola et al. [DNFLP13] present a textual language named SCEL that enables to express these conditions as policies including the necessary modifications of the software architecture for establishing the ensemble. In contrast to MECHATRONICUML, both approaches neither provide further reconfigurations of components and nor real-time constraints in their behavior specification. In [BBCP13], Barnat et al. introduce DCCL that is a formal component specification implementing the concepts of DEECo. They provide an LTL model checking of ensembles proving properties concerning the knowledge of components but not their behavior or the structure of the ensembles.

CompoSE [KKTS09, ASTPH10] defines a hierarchical component model for modeling embedded systems. Atomic components may be implemented in a different language comparable to our continuous components. Each atomic component defines a set of configurations each consisting of a set of ports and a computation that defines the behavior. Structured components specify configurations based on combinations of ports and embedded component instances. At runtime, a component may switch between configurations. In contrast to MECHATRONICUML, the approach does not support using fading functions and message-based communication.

EAST-ADL2 [CFJ+10] is an architecture description language targeted to the development of automotive systems. It provides a component specification where components refer to an external implementation, e.g., specified in MATLAB/Simulink [Matg]. The component model can be mapped to the AUTOSAR component model but does not yet support modes.

Similar to MECHATRONICUML, it focuses on the integration of feedback controllers but provides no means for formal verification or runtime reconfiguration.

Other component models for embedded real-time systems like Koala [vOvdLKM00], Robocop [Maa05], SaveCCM [CHP06, ÅCF+07], Rubus [HMTN+08], COMDES-II [KSA07], PECOS [GCW+02], and CHESS [PV14] provide the ability to specify real-time behavior on a low level of abstraction. They support formal analysis as our component model but neither support message-based communication (except COMDES-II) nor runtime reconfiguration.

All of the mentioned approaches except DEECo do not provide a concept for instantiating connectors on system level.

3.7.2 ADLs for Self-Adaptive Systems

ADLs [MT00] specify software architectures based on components and connectors, although the term *component* is less strictly defined as for component models. Connectors define the interaction of components, constraints define restrictions that the architecture needs to follow while it evolves, and architectural styles are families of related architectures [GMW00].

Bradbury et al. [BCDW04] survey ADLs that enable runtime reconfiguration of the software architecture. They classify these ADLs into three categories: graph-based, process algebra-based, and formal logic-based. Graph-based approaches define an initial configuration that is modified by graph rewriting rules. Examples include CHAM [IW95] and the approaches by Le Métayer [LM98] and Hirsch et al. [HIM98]. MECHATRONICUML also belongs to this category. Process algebra-based approaches like Dynamic Wright [ADG98], Darwin [KM98], or the approach by Bartels and Kleine [BK11] specify processes for each configuration using a process algebra like the π-calculus [MPW92] (Darwin) or CSP [Hoa85] (Dynamic Wright, Bartels and Kleine). At runtime, components switch between processes to execute reconfigurations. Formal-logic-based approaches like the approach by Aguirre and Maibaum [AM02] or GeReL [EW92] declaratively specify component behavior and constraints based on first-order logic. All of the mentioned approaches rely on a textual specification and enable checking for architectural constraints. However, they do not support real-time constraints for functional or reconfiguration behavior. Most approaches discussed above (except Darwin and GeReL) do not support structured components.

The approach by Kacem et al. [KKJ12] specifies a system model using UML 2.0 components [Gro05]. They specify reconfigurations by graph transformations using the concrete syntax of components that are guarded by OCL constraints [Gro12]. In contrast to MECHATRONICUML, they do not support control flow in their rules. They support verifying constraints by translating their specification to Z [Spi92]. In contrast to MECHATRONICUML, they do not support hierarchical components and real-time properties.

3.7.3 Constraint Languages

We compare our approach for modeling architectural constraints by component SDDs to two kinds of constraints languages. First, we compare it to object-based constraints languages that are defined based on classes and objects (cf. Section 3.7.3.1). Second, we compare component SDDs to constraint languages that were defined based on components, mostly as part of an architecture description language (cf. Section 3.7.3.2).

3.7.3.1 Object-Based Constraint Languages

Approaches in this category enable to specify constraints for approaches that are based on classes, references, and objects. Probably the most well-known example is OCL, [Gro12]. OCL supports the textual specification of complex structural properties for classes, e.g., using iterators, sets, and selections.

In [FHTW05], Fish et.al. compare two visualizations of OCL: visual OCL [BKPPT01, KTW02] and constraint diagrams [Ken97]. Visual OCL uses a graph-based syntax for visualizing OCL constraints which is derived from the UML 1.4 notation [Gro01]. Constraint diagrams [Ken97] use a visual notation that is inspired by UML and Venn diagrams [Ven80]. In [FFH05], constraint diagrams are extended by a partial order for quantifiers and their semantics is defined based on first-order predicate logic.

All of these approaches support the specification of architectural constraints based on the abstract syntax of the component model. That requires the developer, who specifies components based on concrete syntax, to translate the constraints to the abstract syntax of the component model. This introduces additional complexity for a developer that keeps him from effectively specifying constraints.

3.7.3.2 Component-Based Constraint Languages

Approaches in this category enable to specify constraints based on a component specification either provided by an architecture description language or a component model.

The ADL Dynamic Wright [ADG98] supports the specification of constraints based on first-order logic using a textual notation. The constraints directly refer to the components and connectors defined by the architectural style. Armani [Mon01] is a constraint language for the Acme ADL [GMW00]. It allows to specify architectural constraints in a first-order predicate logic using a textual concrete syntax. In contrast to our approach, neither Dynamic Wright nor Armani enable to refer to properties of embedded components.

FPath [DLLC09] is a textual query language based on the Fractal component model. It is inspired by XPath [W3C10] and allows to select a set of embedded components in a hierarchical Fractal component across different levels of hierarchy. Therefore, it requires knowledge of the implementation of all components thereby breaking encapsulation. FPath is not explicitly defined as a constraint language, but may be used like one.

The ACL family of architecture constraint languages [TFS10, TSDF11] enables the specification of constraints for components independent of a concrete component model. It supports two levels of abstraction: an object level using an OCL-like language called CCL (core constraint language) and an architecture-level constraint language. In the latter, constraints are modeled as special constraint components that are connected to the functional components by special non-functional or constraint ports even across different levels of hierarchy. The constraints are then verified at design-time to ensure that components are correctly assembled and implemented. In contrast to our approach, they do not enable to evaluate their constraints during runtime.

3.8 Summary

In this chapter, we introduce a consolidated component model for MECHATRONICUML that enables to specify software architectures for self-adaptive mechatronic systems. Thereby, our component model primarily addresses the reflective operator of the OCM reference architecture but also includes the interface to the feedback controllers on the controller level. Therefore, it combines and enhances two existing component models for MECHATRONICUML that have been created by Burmester, Giese, and Hirsch [GTB+03, GBSO04, BGO06, HHG08, GS13] as well as Tichy [THHO08, Tic09]. In a little more detail, we use the necessary distinction of discrete atomic components and continuous atomic components representing feedback controllers from Burmester and Giese [GBSO04, BGO06]. In addition, we use the specification of structured components by means of component parts from Tichy [Tic09]. Compared to the previous component models, our new component model guarantees component encapsulation, enforces a separation of concerns between functional and reconfiguration behavior, and improves semiotic clarity [Moo09] of the concrete syntax. In our component model, we use CSDs [THHO08, Tic09] for specifying reconfigurations of components because they enable for a more concise specification compared to hybrid reconfiguration charts [GBSO04, BGO06]. Furthermore, we introduce a concept for instantiating RTCPs between AMS that are not yet connected with each other. Finally, we defined component SDDs that enable to specify architectural constraints and component properties based on the software architecture.

We underpin the suitability of our component model for specifying software architectures of self-adaptive mechatronic systems by providing a software architecture of the RailCab system (cf. Section 1.1) focussing on the convoy mode. Our example includes the component definitions and the necessary CSDs for realizing RailCab convoys. Additional CICs and CSDs of the example scenario are given in Appendix A. We use this example as a basis for illustrating the further contributions of this thesis in the subsequent chapters.

4 Transactional Execution of Hierarchical Reconfigurations

Reconfigurations in a hierarchical component model often require the reconfiguration of several components that are located on different levels of hierarchy. As an example, the reconfiguration of a structured component instance may require the upfront reconfiguration of one or more of its children as it has been shown in the example of Figure 3.11. In this example, creating the instance mc of MemberControl requires to reconfigure the instances of OperationStrategy and VelocityController first. Then, the port instances created by these reconfigurations are connected by RailCabDriveControl. In general, we distinguish two use cases for such reconfigurations.

In **Use Case 1**, an embedded component instance, in the following referred to as *child*, detects a situation that requires a reconfiguration that it cannot handle by itself. In our example in Figure 3.6, the OperationStrategy component negotiates that the RailCab enters a convoy, but it does not know how to do this itself. Thus, it needs to send a request to the embedding structured component instance of type RailCabDriveControl to handle that situation and to execute the necessary reconfiguration. We will refer to the embedding structured component instance as *parent* in the following.

In **Use Case 2**, a structured component instance executes a reconfiguration that requires the reconfiguration of one or more of its children. In our example, becoming a member of a convoy requires a reconfiguration of the RailCabDriveControl (cf. Figure A.31). Executing this reconfiguration, however, requires that the OperationStrategy changes its port instances and that the VelocityController switches to the ConvoyDrive component instance (cf. Figure 3.12). Therefore, RailCabDriveControl needs to trigger the corresponding reconfigurations on its children.

For both use cases, executing such reconfigurations safely demands that all component instances, which are required to reconfigure, perform their reconfiguration in a coordinated way. The necessary conditions for executing a hierarchical reconfiguration safely are given by the *ACI-properties* (atomicity, consistency, and isolation) of database systems [BHG87, LLC10] and a correct timing. *Atomicity* requires that either all or no component instances, which need to reconfigure, execute their reconfiguration. If reconfigurations are only executed partially, the system is usually unsafe. *Consistency* requires that any component instance has a valid architecture before and after each reconfiguration. *Isolation* ensures that reconfigurations do not interfere with each other. Interference of reconfigurations results in invalid architectures. A correct *timing* demands that if a hard deadline for executing a reconfiguration exists, the system needs to make sure that it meets the deadline before starting to reconfigure. For the remainder, we refer to these properties as *ACI-T properties*. If a reconfiguration is executed according to ACI-T properties, we denote this as *transactional execution*.

For guaranteeing ACI-T properties for a distributed execution of reconfigurations, we provide an approach that adapts the 2-phase commit protocol for distributed database systems [BHG87, ch. 7] to the domain of mechatronic systems. In accordance to the 2-phase-commit protocol, a structured component instance asks all children that are required to reconfigure whether they can execute the required reconfiguration. If all children confirm and if the reconfiguration can be finished in time, the children are notified to execute their reconfiguration. Additionally, we need to check whether the reconfiguration can be finished in time, which is not part of the original 2-phase-commit protocol.

Figure 4.1 summarizes our process for specifying reconfigurations based on our variant of the 2-phase-commit protocol [HSST13]. This process specifies Step S_4 of our overview process in Figure 1.3 on Page 8 in more detail. In the first Step $S_{4.1}$, the developer specifies the reconfiguration rules using CSDs as introduced in Section 3.3. Thereafter, the developer creates a declarative, table-based specification of hierarchical reconfigurations in Step $S_{4.2}$. This specification defines in which situation which CSD is to be executed, but it relieves the developer from specifying how the reconfiguration is carried out [HB13]. Then, we automatically generate an operational behavior specification based on RTSCs from the declarative table-based specification in Step $S_{4.3}$. The operational behavior specification additionally specifies *how* reconfigurations are executed based on the 2-phase-commit protocol. In Step $S_{4.4}$, the developer specifies architectural invariants based on component SDDs (cf. Section 3.5) that define the set of valid configurations for instances of a component. In Step $S_{4.5}$, we use the component SDDs as well as the generated RTSCs that form the operational behavior specification for verifying that the reconfiguration specification fulfills ACI-T properties.

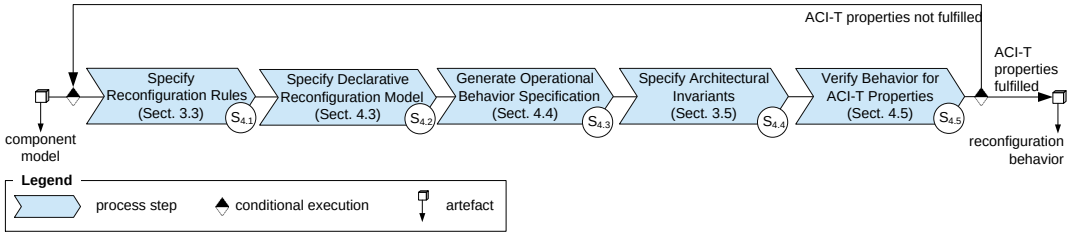


Figure 4.1: Process for Specifying Reconfiguration Behavior (cf. [HSST13])

In the remainder of this section, we first introduce the MECHATRONICUML reconfiguration controller with its constituent elements that contains our declarative, table-based reconfiguration model (Section 4.1). Thereafter, we explain how reconfigurations are executed with respect two the 2-phase-commit protocol (Section 4.2). Section 4.3 defines how we specify these reconfigurations declaratively based on tables in our reconfiguration controller. This specification is the basis for generating operational behavior models as described in Section 4.4. Section 4.5 describes our approach for verifying ACI-T properties for the reconfiguration specification for guaranteeing its safety. We describe our implementation of the hierarchical reconfiguration approach in Section 4.6 and discuss the assumptions and limitations of our approach in Section 4.7. Section 4.8 presents related work regarding transactional execution of reconfigurations before we summarize the contributions of this chapter in Section 4.9.

4.1 MechatronicUML Reconfiguration Controller

In the MECHATRONICUML component model, a developer defines a set of CSDs that specify the possible reconfigurations of a reconfigurable component. This does not enable to specify in which situation which reconfiguration is to be executed. In addition, the MECHATRONICUML component model as introduced in Chapter 3 does not offer means to execute a reconfiguration hierarchically according to ACI-T properties while preserving component encapsulation.

As a solution, we syntactically extend each reconfigurable discrete or hybrid component with a dedicated *reconfiguration controller* that is inspired by the reconfiguration controller of the Fractal component model [BCL+06, BHR09]. Our reconfiguration controller as shown in Figure 4.2 introduces two syntactic elements, namely a *manager* and an *executor*, that encapsulate the necessary behavior for deciding when to execute which reconfiguration and for executing a particular reconfiguration. Optionally, we may add a *risk manager* to the reconfiguration controller that decides whether it is safe to execute a particular reconfiguration [PHST12].

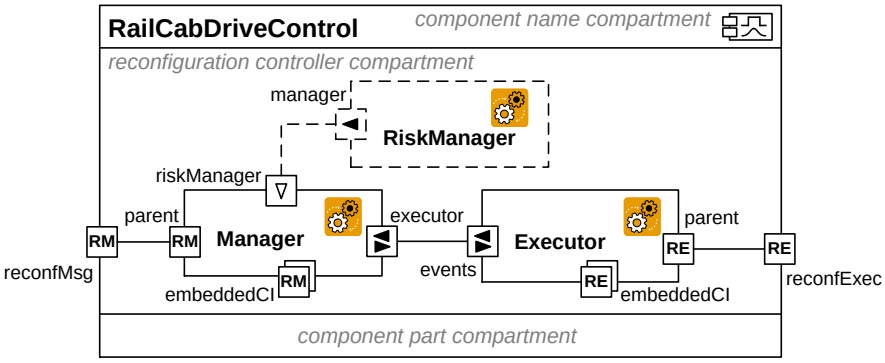


Figure 4.2: Reconfiguration Controller of a Structured Component (cf. [HB13])

By using a dedicated reconfiguration controller, we retain separation of concerns between functional behavior and reconfiguration behavior as advised by McKinley et al. [MSKC04]. We use the RM ports and RE ports as introduced in Section 3.1.1.2 in our reconfiguration controller for providing the necessary interfaces for executing reconfigurations across different levels of hierarchy without violating component encapsulation.

In our approach, the executor is responsible for executing reconfigurations respecting hierarchy and ACI-T properties based on the 2-phase-commit protocol. Thus, it is similar to the script interpreter of Fractal [BHR09]. The manager decides which reconfiguration is executed in which situation, which is not supported by the Fractal reconfiguration controller. The RM ports and RE ports provide the bottom-up and top-down message flow for initiating and executing reconfigurations.

In a little more detail: A component uses its RM ports for sending information on situations that may require a reconfiguration to its parent. Consequently, RM ports are used for bottom-up information provision and to provide the necessary message flow for realizing Use Case 1. A component uses its RE port for offering reconfigurations to its parent. The parent may trigger a reconfiguration on a child by sending a message to the RE Port of that child.

Thus, RE ports are primarily used for top-down reconfiguration initiation and to provide the necessary message flow for realizing Use Case 2. As of [BHR09], Fractal only supports Use Case 2.

For enabling message flow across different levels of hierarchy at runtime, we connect the manager (and executor) to the parent and all embedded component instances using the RM ports (or RE ports). Figure 4.3 illustrates these connections for an instance of the RailCabDriveControl component for driving alone.

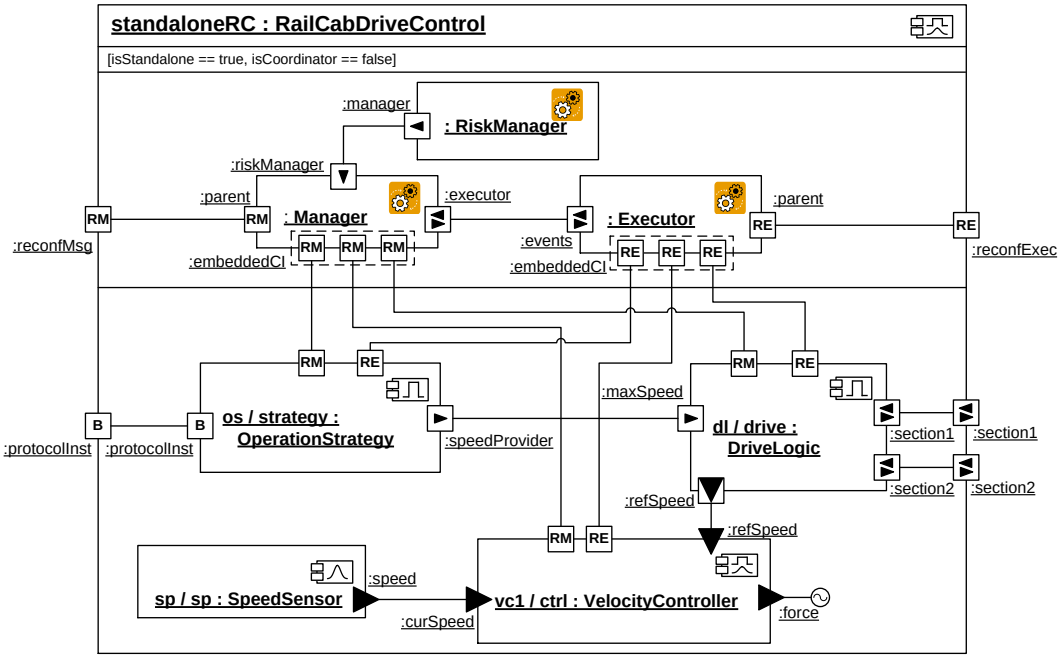
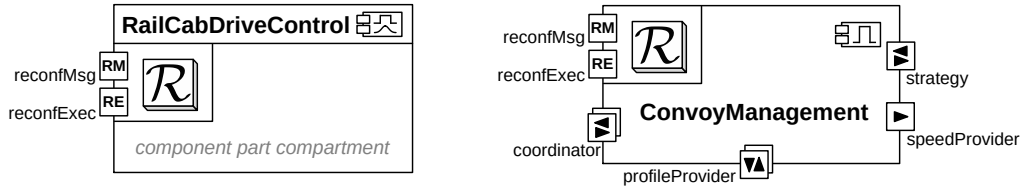


Figure 4.3: Component Instance RailCabDriveControl with Reconfiguration Controller

As shown in Figure 4.2, the manager specifies two RM ports named parent and embeddedCI. The RM port parent implements the RM port of the structured component and is used for sending messages to the parent. The RM multi port embeddedCI connects the manager to the RM port instances of the embedded component instances for receiving their messages. At runtime, one subport instance of this port exists for each child of the structured component instance as shown in Figure 4.3. Since dc:DriveControl has three embedded component instances, the embeddedCI port of the manager contains three subport instances. The executor is connected to the parent and the embedded component instances in the same fashion.

Since the reconfiguration controller has the same structure for any structured component and introduces additional visual complexity, we typically use the short-hand notation shown in Figure 4.4 for visualizing reconfigurable structured and atomic components [HPB12].

Initial ideas regarding the introduction of a manager and executor including dedicated ports for handling reconfiguration have been presented by Dreising [Dre11]. These ideas have been refined and extended significantly in our publications [HPB12] and [HB13].



(a) Reconfigurable Structured Component

(b) Reconfigurable Atomic Component

Figure 4.4: Short-hand Notation for Reconfigurable Components

4.2 Executing Reconfigurations

Using our reconfiguration controller, we can execute reconfigurations with respect to hierarchy considering our two use cases. As mentioned above, we provide a variant of the 2-phase-commit protocol [BHG87, ch. 7]. The 2-phase-commit protocol starts with a *voting phase*. In the voting phase, a structured component instance queries all of its children, which are required to participate in the reconfiguration, whether they actually can reconfigure. The children then evaluate in parallel whether they can execute the requested reconfiguration or not. If so, they commit otherwise they abort. Only if all queried children have committed to the reconfiguration, it can be executed in the *execution phase* as explained below.

In MECHATRONICUML, we need to use such 2-phase-commit approach because we may only start the reconfiguration, if we can ensure that the reconfiguration can be executed completely in time. This is necessary because the reconfiguration controllers are executed as part of the reflective operator of the OCM of the mechatronic system that underlies hard real-time constraints. Therefore, we must not try to reconfigure optimistically and roll-back to a preexisting configuration if the reconfiguration fails as, for example, proposed for reliable reconfiguration of Fractal components in [LLC10]. This is for two reasons: first, the system might come into an inconsistent state that causes it to malfunction if a reconfiguration is only executed partially. Second, it is not guaranteed that returning to the configuration before reconfiguration has started is even possible and safe.

For executing a reconfiguration in the execution phase of our 2-phase-commit protocol, we need to distinguish between purely discrete reconfigurations and reconfigurations that involve continuous components. In the former case, all affected children need to be quiescent as explained in Section 4.2.3 and, therefore, we may reconfigure the system bottom-up in a single pass as explained in Section 4.2.1. We refer to this as *single-phase execution*. If the reconfiguration replaces continuous components, we need to execute fading functions (cf. Section 3.1.2.3). These fading functions require that all port instances of the destroyed and created continuous component instance are properly connected. This requires to split the execution phase into three sub-phases. We refer to this as *three-phase execution* as explained in Section 4.2.2. In general, single phase execution is faster and requires less messages to be exchanged between the executor of a structured component instance and the executors of the children. Therefore, single phase execution should be preferred whenever possible.

4.2.1 Single-Phase Execution

Using single phase execution, the reconfiguration of a structured component instance is performed in a single, bottom-up pass over the component hierarchy. That means, we start at the children that are nested most deeply. The reconfiguration of a structured component instance is then executed after the parallel execution of the reconfigurations of all children. In the following, we describe the message flow and responsibilities in our reconfiguration controller for realizing the two use cases mentioned above with our 2-phase-commit protocol and single-phase execution.

Figure 4.5 illustrates Use Case 1 (i.e., a reconfiguration triggered by a child component) for an instance dc of RailCabDriveControl (cf. Figure 4.3). First, the OperationStrategy component instance sends a message via its RM port to the Manager, requesting, for example, a reconfiguration for adding new a convoy member. Then, the Manager decides whether to execute the reconfiguration and, if so, triggers the executor. The executor initiates the 2-phase-commit protocol and collects the votes of the children. In our example, only the ConvoyCoordination instance is affected by the reconfiguration. If at least one child sends an abort, the reconfiguration will be aborted. If a child commits, it provides a commit time. The commit time denotes how long the child can assure to execute the reconfiguration. After that time, the child is no longer bound to its commit (cf. Section 4.3.4). If all children have committed, the executor computes the minimum of all commit times provided by the children. If the time needed for executing the reconfiguration is less than the minimum commit time, then the executor queries all children to execute their reconfiguration. After all children have finished, the executor performs the reconfiguration of the structured component instance and reports the result to the manager. Since the reconfiguration originated from a request of a child, the result is reported to OperationStrategy.

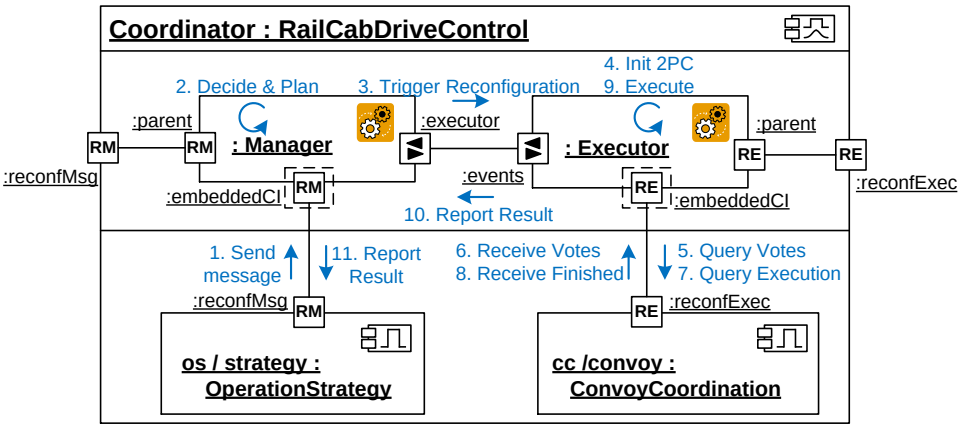


Figure 4.5: Use Case 1: Reconfiguration after Child Request (cf. [HB13])

Figure 4.6 illustrates Use Case 2 in the same fashion. Continuing our example, we consider the ConvoyCoordination instance that is triggered by its parent for adding a new member to the convoy (cf. Figure 4.5). The ConvoyCoordination receives the message via its RE port. The message is propagated to the manager which decides upon the request. The manager then reports the decision to the executor. If the manager has decided not to execute the reconfiguration, the executor immediately sends an abort to the parent. If the manager has

decided to execute the reconfiguration, the executor initiates the 2-phase-commit as in Use Case 1. After it has collected the votes of the children, it checks whether the reconfiguration can be executed in time using the commit times of the children. Then, it sends the resulting vote to the parent. After sending the vote, the executor waits for the answer of the parent, but no longer then the minimum commit time. If the parent decides to execute (or abort), the executor queries the execution (or abortion) of the reconfiguration on the children. After all children have finished, the executor performs the reconfiguration of the structured component and reports to its parent that the reconfiguration has been finished.

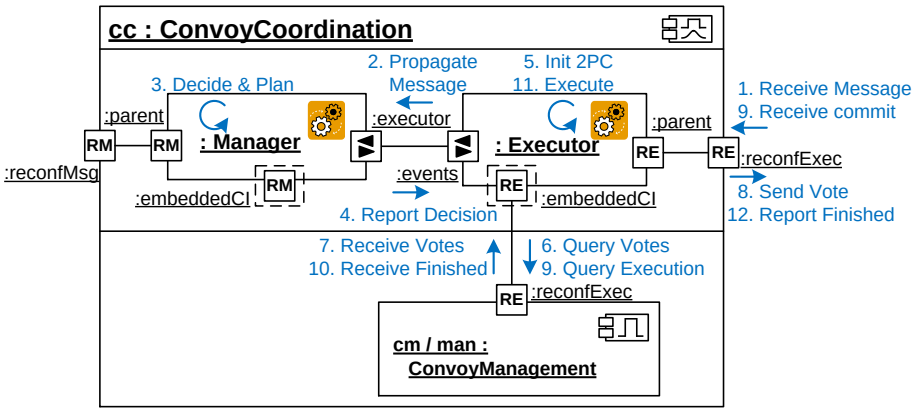


Figure 4.6: Use Case 2: Reconfiguration as Part of 2-Phase-Commit (cf. [HB13])

The use cases for reconfiguration are designed such that they propagate recursively for children. If a structured component reconfigures based on Use Case 1 and invokes a child, that child reconfigures based on Use Case 2. If the child is a structured component instance itself and needs to invoke reconfigurations of its children as well, Use Case 2 propagates recursively. For an atomic component, only Use Case 2 may occur.

4.2.2 Three-Phase Execution

Single-phase execution of reconfigurations as described in the previous section cannot be applied if the reconfiguration involves replacing continuous components. As an example, consider the reconfiguration for becoming a convoy member shown in Figure 3.11. The reconfiguration requires that the VelocityController switches from the StandaloneDrive to the ConvoyDrive feedback controller (cf. Figure 3.7).

Figure 4.7 shows an intermediate CIC that would occur if we executed this reconfiguration according to single-phase execution. As part of the execution phase, rc1 already queried the execution on vc1. As a result, vc1 started executing the CSD shown in Figure 3.12. vc1 already created the ConvoyDrive instance and currently executes the fadeToConvoy fading function in the fading component. At this point of time, both continuous component instances are executed in parallel. However, the ConvoyDrive instance will not properly work because the input ports refDist and curDist have no defined values. The reason is that these port instances are delegated by vc1 and needed to be connected in rc1 before starting to execute the fading. In particular, we needed to create instances of the SpeedSensor and the MemberControl in rc1 prior to executing the fading function.

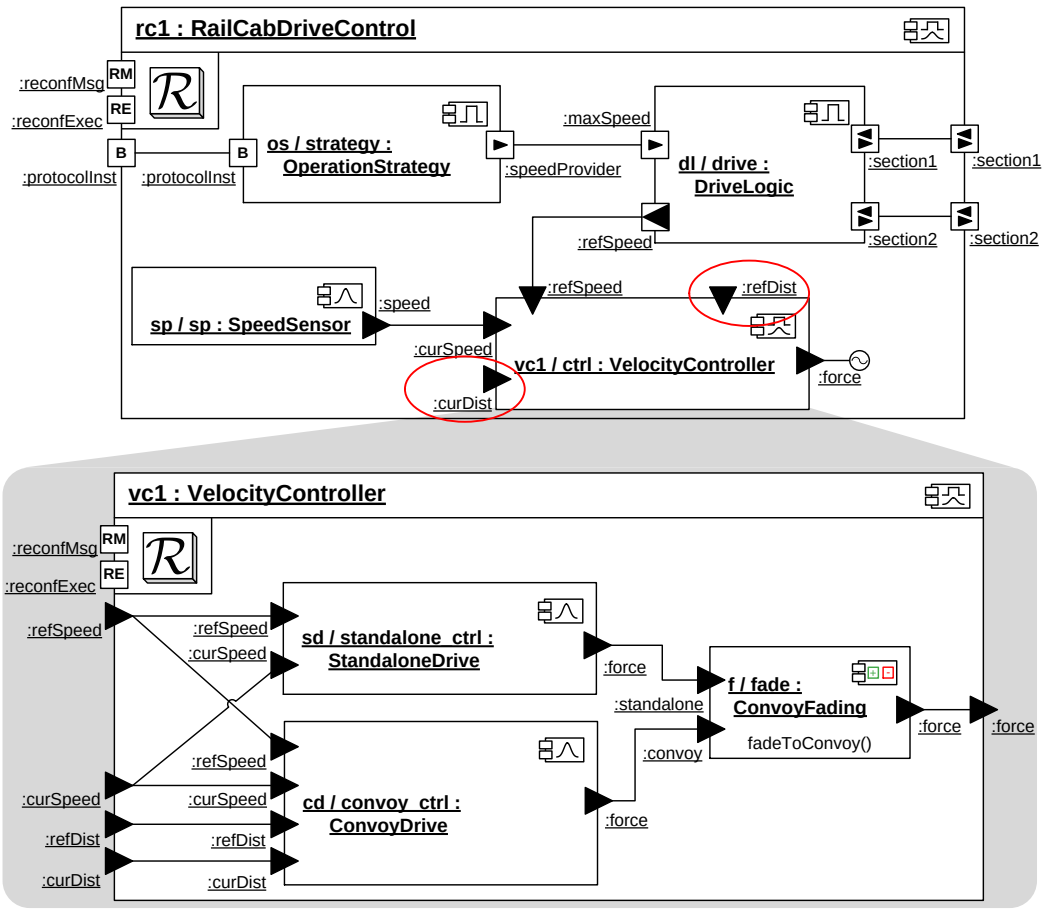


Figure 4.7: Problems when Replacing Continuous Component Instances using Singe-Phase Execution

As a solution to this problem, we split the execution phase of our 2-phase-commit protocol into three sub-phases as proposed by Volk [Vol13] if the reconfiguration replaces continuous components. These sub-phases are *setup*, *fading*, and *teardown*. Each of these sub-phases executes part of the reconfiguration. Figure 4.8 illustrates how these sub-phases are executed in a structured component instance. A filled bar denotes that the instance is currently executing reconfiguration behavior, while an unfilled bar denotes that the instance is idle. We explain this figure in more detail along with the different phases in the following Sections 4.2.2.1 to 4.2.2.3. The voting phase of the 2-phase-commit is executed as for single-phase execution (cf. Section 4.2.1) and will not be described here.

4.2.2.1 Setup

The three-phase execution starts with the setup phase. The setup phase (hierarchically) reconfigures a component instance such that all preconditions for executing the fading functions are established. Therefore, it changes the software architecture of the mechatronic system, but it does not change the exhibited behavior of the mechatronic system. The setup phase

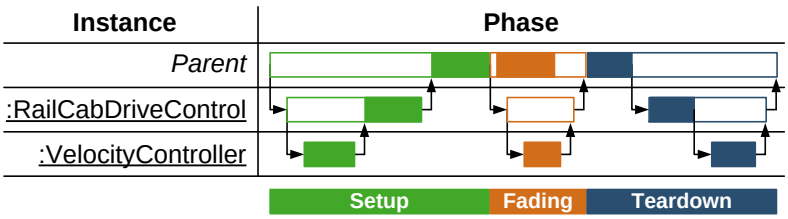


Figure 4.8: Illustration of Three-Phase Execution [Vol13]

is executed bottom-up as shown in Figure 4.8, i.e., a component first triggers its children in parallel and executes its own setup behavior after all children are finished.

In the setup phase, each component instance that is affected by the reconfiguration creates all discrete, continuous, and hybrid port instances as specified by the reconfiguration rule. In addition, structured component instances create all embedded component instances and all connector instances between continuous and hybrid port instances. Discrete component instances and ports are kept in a suspended mode, i.e., their RTSCs are not being executed and their clocks do not yet progress. All hybrid port instances that have been created during setup already emit their default value though. All affected fading components still forward the unmodified value of the continuous component that is to be replaced.

Figure 4.9 shows component instances of RailCabDriveControl and VelocityController after performing the setup phase for the reconfiguration becomeMember shown in Figure 3.11. The corresponding CSD is applied on the component instance of RailCabDriveControl for driving alone shown in Figure 3.9.

Since the setup phase is executed bottom-up, the execution starts at vc1. vc1 creates an instance of ConvoyDrive including the port instances refDist and curDist. In addition, it delegates all in-ports to the corresponding port instances of the new component instance cd. Finally, it creates the port convoy at the fading component including the assembly instance. As a result, vc1 contains all necessary component instances, port instances, and connector instances for executing the fading function.

After vc1 finished, rc1 executes its setup phase. According to the CSD in Figure 3.11, rc1 creates instances of DistanceSensor and MemberControl. Since MemberControl is a discrete component, the instance mc remains suspended and only emits the default reference distance via its hybrid refDist port instance. In addition, rc1 creates the port instances member and refDistReceiver, but it does not yet create the corresponding delegation instances. Finally, rc1 creates the assembly connector instances between the continuous and hybrid port instances for connecting DistanceSensor and MemberControl to vc1.

As it can be inferred from Figure 4.9, all in-ports of vc1 are properly connected. Thus, the fading function can now be executed and provide a meaningful result. Up to now, the behavior of rc1 and vc1 has not been changed because vc1 still emits the force value of sd and because the discrete connector instances in rc1 have not yet been modified and MemberControl is still suspended.

4.2.2.2 Fading

In the fading phase, the behavior of the mechatronic system changes, but its software architecture does not change. In particular, we execute the fading functions of all affected fading

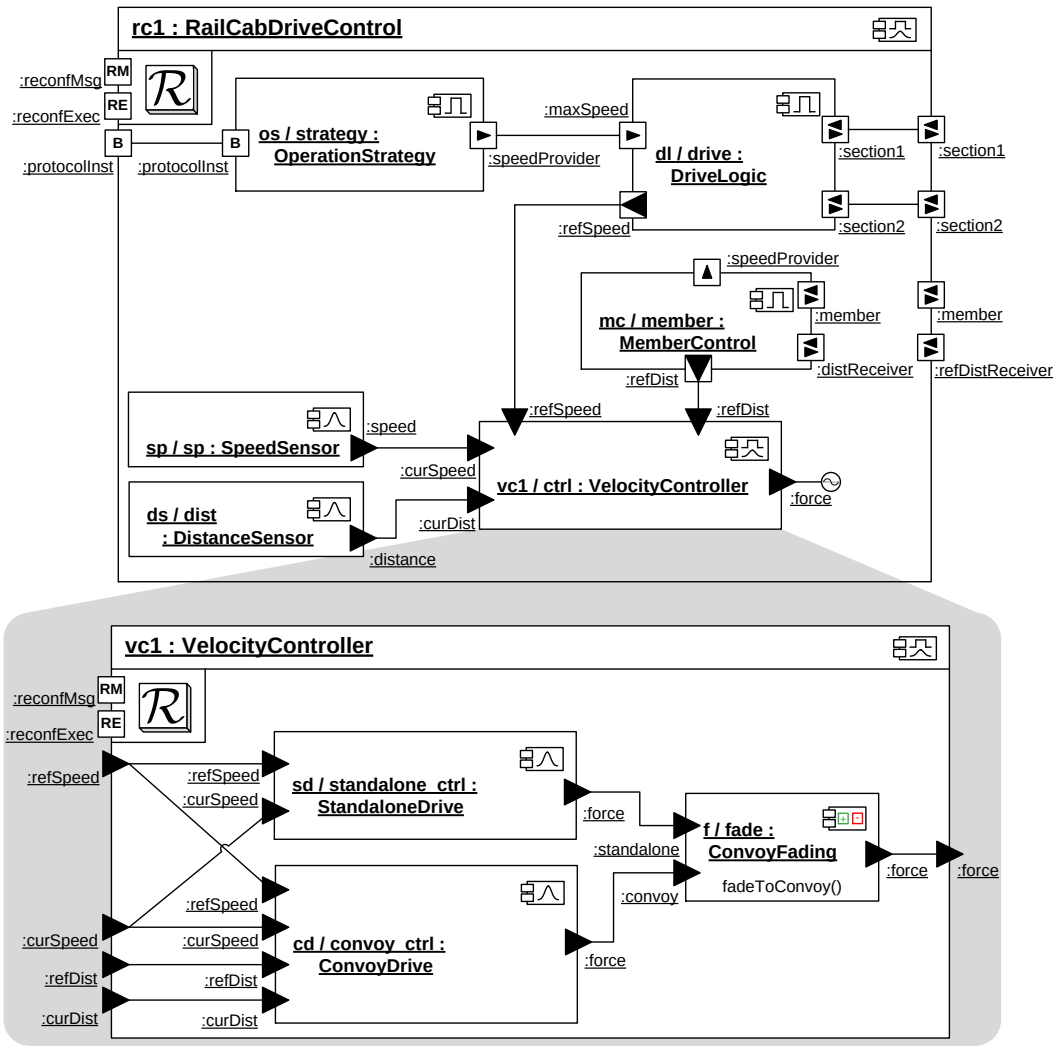


Figure 4.9: RailCabDriveControl after Executing the Setup Phase for the Reconfiguration becomeMember

components. As shown in Figure 4.8, we execute all fading functions in parallel, i.e., the fading components now emit the values of the fading functions that combine their input values. Discrete components remain idle during this phase.

In our example in Figure 4.9, the fading component f executes the switchToConvoy fading as specified by the CSD in Figure 3.12.

4.2.2.3 Teardown

The execution of the reconfiguration finishes with the teardown phase. In this phase, both, the behavior and the software architecture of the mechatronic system change. The teardown phase is executed top-down as shown in Figure 4.8, i.e., a component first executes its own teardown behavior before it triggers its children in parallel.

In the teardown phase, we destroy all component instances, port instances, and connector instances as specified by the reconfiguration rule. Furthermore, we activate all discrete component instances and port instances that were created in the setup phase including the connector instances between discrete port instances. The fading components now forward the unmodified value of the continuous component instance that has been created.

Continuing our example in Figure 4.9, we now destroy the StandaloneDrive instance including all of its port instances and adjacent connector instances. In rc1, we destroy the assembly between speedProvider of os and maxSpeed of dl. Additionally, os destroys its speedProvider port instance. Furthermore, we create delegation connector instances that delegate the port instances member and distReceiver of mc to the corresponding port instances of rc1. Finally, we create the assembly connector instance between speedProvider of mc and maxSpeed of dl. The result is, as expected, equivalent to the component instance Member shown in Figure A.31.

4.2.3 Quiescence

Component instances and port instances may not be deleted at any point in time. In particular, they may not be deleted if they currently perform a computation or if they are engaged in executing a communication protocol that is required for the safe operation of the system. *Quiescence* [KM98, ZC06] defines whether it is safe to delete a component instance or one of its port instances at a certain point of time. Then, executing a reconfiguration safely demands that all affected component instances are quiescent.

As an example, consider a member RailCab that leaves a convoy. As a consequence, the RailCab will destroy its instance of MemberControl and it will switch back to the StandaloneDrive feedback controller (cf. Section 4.2.2). However, the RailCab may not perform this reconfiguration if it is still driving closely behind another RailCab. If it performs the reconfiguration, it will not be notified about braking maneuvers of the convoy and, thus, a crash is likely to occur.

Therefore, we need a concept for defining quiescence of component instances in MECHATRONICUML. In particular, we need to define quiescence of discrete atomic component instances. For continuous atomic component instances, the fading functions define how they may be safely replaced. A structured component instance is quiescent with respect to a particular reconfiguration if all children that are affected by this reconfiguration are quiescent.

The concept for quiescence of discrete atomic component instances needs to answer the following three questions for being usable in our 2-phase-commit protocol.

1. Is the component instance quiescent?
2. If the component instance is quiescent, how long will it remain quiescent?
3. If the component instance is not quiescent, when will it be quiescent again?

These questions need to be answered by the discrete atomic component instance during the voting phase of the 2-phase-commit protocol. Question 1 and 3 are important for deriving the voting result. A discrete atomic component instance may only vote for commit if it is presently quiescent or if it will become quiescent early enough. Question 2 is important for deriving the commit time that defines how long the component instance will stick to its commit. However, the component instance will only vote for commit if the commit time is above a threshold that is defined by the developer as we discuss in Section 4.3.4.

An approach that may answer the three questions given above for a self-adaptive mechatronic system has been developed as part of a Master's thesis [Sch15]. We will sketch its core ideas in the following. Our ideas are inspired by the approach of Zhang and Cheng [ZC06]. Their approach considers a state-based functional behavior specification of components based on petri nets (cf. [ZC06]) or UML Statecharts (cf. [RC08]), but they do not consider real-time constraints or properties of the physical system in their specification. For performing a reconfiguration, the system switches between source and target functional behaviors by executing an adaptation behavior (cf. Section 2.1.2). In our approach, the source and target functional behaviors correspond to the CICs before and after executing a reconfiguration, while the adaptation behavior is represented by our 2-phase-commit protocol.

For guaranteeing quiescence, Zhang and Cheng define a set of global invariants using temporal logic that need to be fulfilled during the adaptation process. Then, a state s of the source functional behavior is quiescent if there exists a state t in the target functional behavior such that the adaptation from s to t does not violate any global invariant [ZC06]. This is ensured at design time by model checking the functional behaviors and the adaptation behavior [ZGC09]. Then, all states s of the source functional behavior are marked as quiescent with respect to a given adaptation.

In a self-adaptive mechatronic system, the state of a component instance is not only determined by the active state of its RTSC but also by the current clock values of the RTSC and, potentially, the physical state of the mechatronic system. The physical state of a mechatronic system is given, for example, by its current spatial position, its speed, or its acceleration. Consider a member RailCab that wants to leave a convoy as an example. There, we need to consider the RailCab's distance to the preceding RailCab and its current speed for deciding whether the component instance is quiescent. Therefore, it is not possible to simply mark states of an RTSC as quiescent as proposed by Zhang and Cheng.

As an additional problem, considering the clock values and the physical state of the system induces a so-called hybrid model checking problem [Hen96]. Such model checking problems cannot be solved efficiently with current techniques [ERNF12] as we discuss in more detail in Chapter 6. As a possible solution, we can use our approach of motion profiles [FHK$^+$13, FHK$^+$14] for avoiding hybrid verification. A motion profiles gives an assertion on the limits of a change of the physical parameters of the mechatronic system in the future. Each motion profile is defined with respect to a particular control strategy, with respect to the current driving maneuver, e.g., braking or accelerating, and with respect to optimization criteria, e.g., braking strongly vs. braking smoothly. As a result, each system is equipped with a multitude of motion profiles. However, even in this case the state-space that needs to be explored is significantly larger compared to Zhang et al. [ZGC09] because we need to consider clocks and all possible motion profiles of the RailCab based on each possible point in time of the maneuver that is defined by the motion profile.

Therefore, our idea is to identify quiescent states at runtime as a part of the voting phase of our 2-phase-commit protocol. This is more efficient than computing all possible symbolic states at design time [GCZ08] because we only need to check a few symbolic states. In particular, we only need to consider symbolic states that are reachable from the current snapshot of the component instance in a short period of time. In addition, we only need to consider the currently applied motion profile instead of considering all n available motion profiles which reduces the state space by factor n. Figure 4.10 summarizes the idea of our approach.

At design time, the developer needs to specify a set of conditions for quiescence. These conditions refer to the different parts of the atomic component instance, e.g., an active state of the RTSC, messages that are located in the message buffer of a port instance, or values regarding the physical state of the system that are received via a hybrid port instance. In our example, we might require that the distance of the member RailCab to the RailCab directly driving in front of it must be larger than 50 m. Then, any symbolic state of the RTSC that fulfills all of the imposed conditions at runtime is considered to be quiescent. Thus, the conditions correspond to the invariants used by Zhang et al. [ZGC09]. For supporting the developer, we provide him with a checklist for typical influence factors that need to be considered for quiescence. The checklist will be derived by analyzing influence factors on quiescence in different self-adaptive mechatronic systems such as RailCabs or self-coordinating cars [PHMG14].

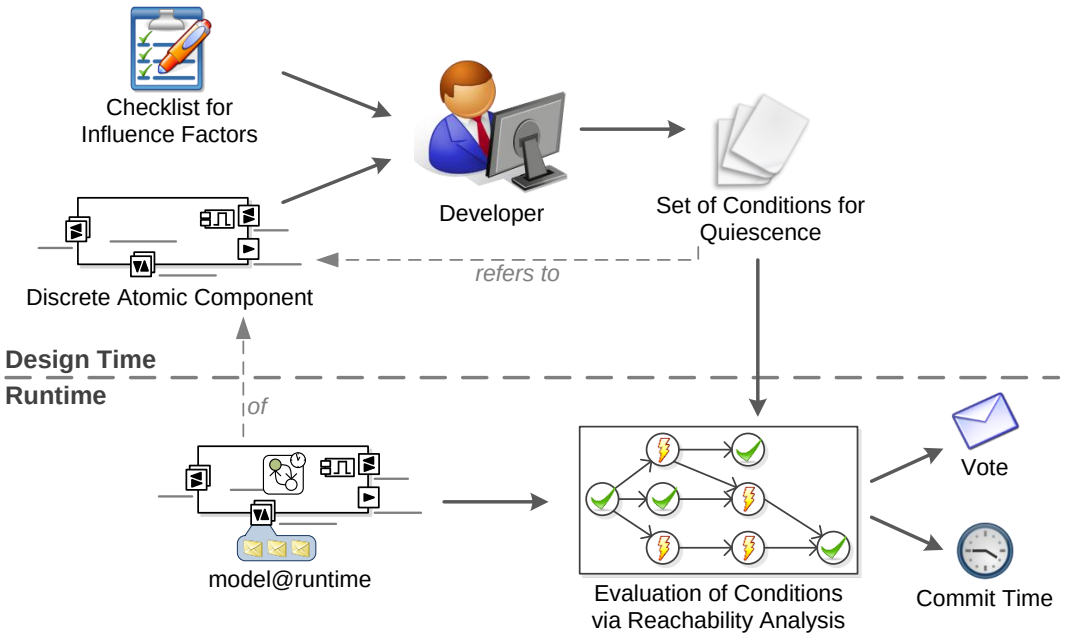


Figure 4.10: Approach for Identifying Quiescent States in MECHATRONICUML

At runtime, we use our model@runtime of the atomic component instance for evaluating the conditions as a part of the voting phase of the 2-phase-commit protocol. If the atomic component instance is requested to execute a reconfiguration by its parent, we start a reachability analysis on the current snapshot of the model@runtime. Then, we calculate the symbolic states that the atomic component instance may reach in a short time frame starting from the current snapshot. The time frame that needs to be considered is defined by the time for execution of our 2-phase-commit protocol and the threshold for the commit time. For each of the symbolic states, we evaluate the conditions that the developer has specified at design time. The result is a zone graph (cf. Section 2.2.1) where each symbolic state is marked as quiescent or non-quiescent. Thereby, we only need to consider the currently active motion profile and the current physical state of the system. Based on the clock values of the symbolic states, we may utilize the paths of the zone graph for calculating whether the component in-

stance is quiescent and how long it will remain quiescent. From this information, we derive the voting result and the commit time that are passed to the parent.

The reachability analysis may be carried out by a variant of the reachability analysis for RTSCs introduced in Appendix C.3 that is optimized for being executed on embedded computing devices. This rechability analysis needs to be implemented such that its runtime is predictable. This is necessary for guaranteeing that the component instance will obtain a voting result within a given time that the component asserts to its parent as we describe in more detail in Section 4.3.4. Predictability may be achieved, e.g., by limiting the number of symbolic states that are investigated for each trace of the zone graph as proposed by bounded model checking techniques [BCC+03].

4.3 Declarative, Table-based Specification of the Reconfiguration Controller

In our approach, we provide a declarative specification of the behavior of the reconfiguration controller based on tables. These tables extend the component model introduced in Chapter 3 by additional syntactical elements that are tailored the 2-phase-commit protocol. More technically speaking, the tables relieve the developer from manually specifying RTSCs for RM ports, RE port, manager, and executor.

In a little more detail, the developer needs to specify one table for each RM port and for each RE port of a reconfigurable component. This table enhances the interface of the port, i.e., which messages the port may send or receive, with timing constraints for the messages that are relevant for executing the 2-phase-commit protocol. In addition, the developer needs to specify one table for the manager and for the executor of each reconfigurable structured component. The entries in these tables define conditions that express when to execute which reconfiguration, but they do not specify how the conditions are checked and how reconfigurations are executed according to the 2-phase-commit protocol.

The timing constraints that are contained in our declarative, table-based specification are requirements for an execution of the reconfiguration behavior on a hardware platform. For a self-adaptive mechatronic system, these requirements originate from three sources. First, they originate from conditions that are imposed by the physical environment. As an example, consider a convoy build-up at a switch. In this case, the reconfigurations for becoming a coordinator or member, respectively, need to be finished before coming too close to the switch. Second, timing requirements are defined by the functional safety specification. In case of a hardware failure, a reconfiguration that implements a self-healing operation needs to be finished within a particular time in order to prevent a hazard. This particular time may be obtained by performing a timed hazard analysis [PST13]. Third, timing requirements originate from the quiescence criteria. The component instances that are affected by a reconfiguration need to remain quiescent throughout the reconfiguration (cf. Section 4.2.3). As a result, the reconfiguration needs to be finished before the component instance needs to execute some non-quiescent behavior that is necessary for safely operating the system.

In the following, we provide details regarding our declarative, table-based specifications of manager and executor as well as of the interfaces of RM ports and RE ports in Sections 4.3.1 to 4.3.3.

4.3.1 Interface Specification of RM Ports

An RM port is a special kind of discrete port (cf. Section 3.1.1.2) that is solely used for communication between the managers of reconfigurable components. Its interface is defined by a table with four columns. The first column defines the message types that may be sent to the parent. The second column gives additional information on the semantics of the message using a type. We distinguish two types of messages: info messages and requests. An info message is only provided for information and does not necessarily require a reconfiguration. A request is sent in situations where a reconfiguration is necessary from the perspective of the sending component and where it cannot solve the situation itself. In case of a request, the developer of a component may specify an expected response time in the third column. It defines the point in time where the component needs the information whether a reconfiguration has been executed by the parent. The fourth column optionally contains a human readable description of the reported situation for a developer. Each interface entry corresponds to one row in the table.

Message Type	Type	Expected Response Time	Description
drivingAtHighSpeed	info	---	RailCab travels at high speed.
drivingAtNormalSpeed	info	---	RailCab travels at normal speed.
distanceSensorFailure	request	200 ms	Distance sensor is broken.

Figure 4.11: RM Port Specification of the RailCabDriveControl Component (cf. [HB13])

Figure 4.11 shows an example of an RM port specification for the RM port of the RailCabDriveControl component shown in Figure 4.3. It contains three entries. First, the RailCabDriveControl informs its parent about its speed profile using the messages drivingAtHighSpeed and drivingAtNormalSpeed of type info. This information may be used to adapt the sensing of obstacles depending on the speed. If the speed is high, obstacles need to be sensed in larger distances to brake early enough. In addition, RailCabDriveControl sends a request positionSensorFailure, which denotes that the distance sensor is broken. This request triggers a self-healing operation (cf. [Pri13, PST13]) and needs to be finished in 200 ms for guaranteeing the convoy safety.

4.3.2 Manager Specification

The behavior of a manager is specified declaratively using a table with eight columns. We refer to each row of the table as an entry of the manager specification. The entries of the manager specification define how the manager needs to react if it receives a particular message. In our approach, the manager only reacts to messages that it receives from the children or from the executor. We did not yet include dedicated monitoring capabilities for structured components in our approach. The manager specification needs to contain exactly one entry for each message that the manager may receive from the children or from the executor.

In the manager specification, the first column contains a message type that the manager may receive either from a child or from the executor. The second and third column define whether the manager treats the message or whether it propagates the message to its parent. A message that is received from the executor always needs to be treated. We allow, however, that the manager operates as a sink with respect to messages sent by a child by neither

treating nor propagating them. We do not allow to treat and propagate a message at the same time because that may lead to conflicting reconfiguration decisions on different levels of hierarchy in the component model. All messages that are specified as propagated in the manager specification need to appear in the interface specification of the RM port parent of the corresponding reconfigurable component.

If the specification defines that a message is treated, the developer must specify a reconfiguration rule to be executed by the executor in the fourth column. Whether a reconfiguration may be executed at runtime depends on three conditions that are specified in columns five to seven: (1) whether it is *allowed* to execute the reconfiguration (column Structural Condition), (2) whether it is *safe* to execute the reconfiguration (column Safety Relevant), and (3) whether it is *useful* to execute the reconfiguration (column Invoke Planner). Only if all three conditions evaluate to true during runtime, the manager will trigger the executor to execute the reconfiguration. We explain these conditions in more detail in the following.

For each entry of the manager specification, the developer needs to define a structural condition. The structural condition specifies a condition on the embedded component instances that must be fulfilled for executing the reconfiguration. Currently, we only support specifying structural conditions based on component SDDs (cf. Section 3.5). It is only *allowed* to execute a reconfiguration if the structural condition is fulfilled. If the execution of the reconfiguration shall not be restricted, true may be used as a structural condition as for Entries 4, 6, 7, and 8.

A reconfiguration may affect the functional safety [IEC10, ISO11a] of the system. An example for such reconfiguration is joining a convoy as a member. The functional safety specification puts a limit on the risk that a dangerous situation may occur during runtime. If a RailCab joins a convoy, the risk of a collision rises due to the small distances between the RailCabs. If, in addition, one of the sensors necessary for a convoy drive is broken, the risk of a collision may become too high to be acceptable. In such cases, we need to use a runtime risk manager in the reconfiguration controller as shown in Figure 4.2. Then, the runtime risk manager decides whether it is *safe* to execute the reconfiguration [PHST12, TSL13]. If it is not safe, the reconfiguration will be blocked and not executed. If the reconfiguration does not affect the functional safety, we do not need to invoke the runtime risk manager. Then, it is always safe to execute the reconfiguration. In addition, the runtime risk manager may only be invoked if the message of the corresponding entry is treated. The runtime risk manager introduced in [PHST12] calculates in advance which reconfigurations need to be blocked based on the current system configuration. Therefore, we do not need to account for its runtime in our specification.

Finally, we account for the usage of a planner in our manager specification. The planner will be contained in the cognitive operator of the OCM and decides whether it is *useful* to execute the reconfiguration based on the goals of the system [ZW14]. Although we have not explicitly added a planner to our approach, yet, we enable the developer to specify whether to invoke a planner or not. A planner may only be invoked if the message is treated. If a planner is invoked, the developer needs to provide the maximum time that the planner may run in the eighth column of the table. If no planner shall be invoked, it is always considered to be useful to execute the reconfiguration if it is requested.

Figure 4.12 shows the manager specification of the RailCabDriveControl component in Figure 4.3. The messages in the Entries 1 to 3 are sent by the child OperationStrategy that negotiates with other RailCabs whether to form a convoy. These messages are treated and not

	Message Type	Treat	Propagate to parent	Reconfiguration Rule	Structural Condition	Safety Relevant	Invoke Planner	Time For Planning
1	becomeCoordinator	Yes	No	becomeCoordinator()	isStandalone()	Yes	Yes	20 ms
2	newMember	Yes	No	addConvoyMember()	isCoordinator()	No	No	---
3	becomeMember	Yes	No	becomeMember()	isStandalone()	Yes	Yes	20 ms
4	noConvoyMode	Yes	No	disableConvoyMode()	true	No	No	---
5	enableConvoyMode	Yes	No	enableConvoyMode()	convoyDisabled()	No	No	---
6	distanceSensorFailure	No	Yes	---	true	No	No	---
7	drivingAtHighSpeed	No	Yes	---	true	No	No	---
8	drivingAtNormalSpeed	No	Yes	---	true	No	No	---

Figure 4.12: Manager Specification of the RailCabDriveControl Component (cf. [HB13])

propagated. Therefore, we specify a reconfiguration rule that is contained in the executor specification (cf. Section 4.3.3) for each of them. Each of the reconfigurations specifies a structural condition by means of a component SDD (cf. Section 3.5). A RailCab may only become coordinator of a convoy, if it is not yet engaged in a convoy. This is expressed by the component SDD isStandalone in Figure 3.20. The same condition needs to be fulfilled for becoming a member of a convoy. New members can only be added if RailCab already is the coordinator of a convoy. This is formally specified by the component SDD isCoordinator in Figure 3.19. The reconfigurations becomeCoordinator and becomeMember are safety relevant and may be blocked by the runtime risk manager because building a convoy affects the functional safety of the system. Adding a new member to an existing convoy is not safety relevant for the coordinator. In addition, we foresee invoking a planner before building a convoy. For both reconfigurations, we permit the planner to run for 20 ms.

The messages in Entries 4 and 5 are sent by the parent of RailCabDriveControl. If a RailCab started transporting hazardous goods, it must no longer engage in convoys and, thus, the OperationStrategy component instance will remove its broadcast port (Entry 4). After delivering the hazardous good, the convoy mode may be enabled again (Entry 5). Both reconfigurations are not safety relevant and do not require to invoke a planner.

Finally, the messages in Entries 6 to 8 are sent by the VelocityController. These messages are propagated to the parent and not treated. Consequently, we neither specify a reconfiguration rule nor one of the three conditions for executing the reconfiguration.

4.3.3 Executor Specification

The behavior of an executor is specified declaratively using a table with three columns. Again, we refer to each row of the table as an entry of the executor specification. The entries of the executor specification define an integer ID for each reconfiguration rule in the first column. The second column contains a reference to the reconfiguration rule. In our approach, we use CSDs as introduced in Section 3.3 for specifying reconfiguration rules. The third column defines the maximum worst-case execution time (WCET, [Kop97, ch. 4.5]) for executing the reconfiguration rule on a platform. Please note that this is not the actual WCET of the CSD on a particular platform but a requirement how large the WCET may be as described at the beginning of Section 4.3.

ID	Reconfiguration Rule	WCET
1	becomeCoordinator()	50 ms
2	addConvoyMember()	10 ms
3	becomeMember()	50 ms
4	disableConvoyMode()	5 ms
5	enableConvoyMode()	5 ms

Figure 4.13: Executor Specification of the RailCabDriveControl Component (cf. [HB13])

Figure 4.13 shows the executor specification of the component RailCabDriveControl (cf. Figure 4.3). RailCabDriveControl supports five reconfiguration rules. The first one creates the necessary components, ports, and connectors for operating as a convoy coordinator. It is formally specified by the CSD in Figure A.53. If the component already operates as a coordinator, the second reconfiguration rule adds another member to the convoy. It is shown in Figure A.57. The third reconfiguration rule creates the necessary components, ports, and connectors for operating as a convoy member as specified by the CSD in Figure 3.11. Finally, reconfiguration rules four and five enable to remove or create the broadcast port of OperationStrategy including the broadcast port and delegation connector instance in RailCabDriveControl. The corresponding CSDs are given in Figures A.60 and A.62.

4.3.4 Interface Specification of RE Ports

An RE port is a special kind of discrete port (cf. Section 3.1.1.2) that is solely used for communication between the executors of reconfigurable components. Its interface is defined by a table with five columns. The first column defines a message type that it accepts from its parent. The second column contains a human readable description of the effect of sending a corresponding message to the component. The remaining columns contain time values that define timing requirements towards the execution of the 2-phase-commit protocol.

The third column contains the time for decision. This time value provides an upper bound for the time that the component needs for deriving a decision whether it may execute a reconfiguration or not. This may include the time that is necessary for moving into a quiescent state (cf. Section 4.2.3). The fourth column contains a timing specification that defines an upper bound on how long the component needs for executing the reconfiguration that is associated with this message in the manager specification (cf. Section 4.3.2). If the reconfiguration may be executed according to single-phase execution, the time for execution is a single value. If the reconfiguration needs to be executed according to three-phase execution, then the timing specification contains separate time values for setup, fading, and teardown. We need to provide distinct time values for each phase for correctly computing how much time a hierarchical reconfiguration needs for being executed after deploying the component as explained below. Finally, the fifth column provides the minimum commit time that defines a lower bound on how long the component may stick to its decision of executing the reconfiguration (cf. Section 4.2).

Figure 4.14 shows the interface specification of the RE port reconfExec of the RailCabDriveControl component in Figure 4.3. The component offers two reconfigurations to its parent that correspond to the two entries in the table. The first one uses the message type noConvoyMode. Sending this message to the RE port of RailCabDriveControl causes the RailCab not

Message Type	Description	Time for Decision	Time for Execution	Minimum Commit Time
noConvoyMode	The RailCab will not engage in convoys anymore.	25 ms	20 ms	200 ms
enableConvoyMode	The RailCab will try to join convoys if possible and useful.	25 ms	50 ms	200 ms

Figure 4.14: RE Port Specification of the RailCabDriveControl Component (cf. [HB13])

to drive in convoys any longer. This reconfiguration will be triggered by the RailCab if it transports hazardous goods. The second one uses the message type enableConvoyMode and causes RailCabDriveControl to enable the convoy mode again. Since both reconfigurations involve discrete components only, the entries in the fourth column of the RE port interface specification only provide a single time value for the time for execution.

4.4 Generating Operational Behavior Specifications

The declarative, table-based specification of the reconfiguration controller introduced in the previous section cannot be verified or implemented directly. In order to verify or implement the reconfiguration behavior, we need a formal and operational behavior specification. Therefore, we automatically generate a RTSC for both, manager and executor, because RTSCs are formal and operational.

Using RTSCs for specifying the operational behavior of manager and executor enables to reuse the existing tool chain for MECHATRONICUML. That includes model checking support (cf. Section 4.5), WCET analyses [Bur06, BGST05], export to simulation environments as MATLAB/Simulink (cf. Section 6) or Modelica [PSR+12, PHMG14], and code generation [BGS05, AAB+11, Gei13].

In this section, we provide generation templates for the RTSCs of manager and executor [HB13]. The generation templates define the 2-phase-commit protocol implementation as outlined in Section 4.2 and contain placeholders for the entries of the tables of our declarative, table-based reconfiguration specification introduced in Section 4.3. The placeholders are automatically filled by the information given in each row of the tables. By using the generation templates and an automatic generation process, we relieve the developer from specifying a large and complicated behavior specification for the 2-phase-commit protocol by himself. In summary, this means saving 18 states and 30 transitions for the manager RTSC given in Section 4.4.1 plus 2 states and 8 transitions for each entry of the manager specification. For the executor RTSC given in Section 4.4.2, we save another of 71 states and 102 transitions of manual work plus 1 state and 4 transitions for each entry in the RE port specification, 2 states and 5 transitions for each reconfiguration rule assuming single-phase execution, and 6 states and 7 transition for each reconfiguration assuming three-phase execution.

4.4.1 Manager Specification

Figure 4.15 shows the generation template for the manager RTSC. The RTSC template is complex and provides many variation points that depend on the manager specification. By using our generation template, however, we hide the complexity of the RTSC from the developer who can reuse the template for all of his reconfigurable structured components.

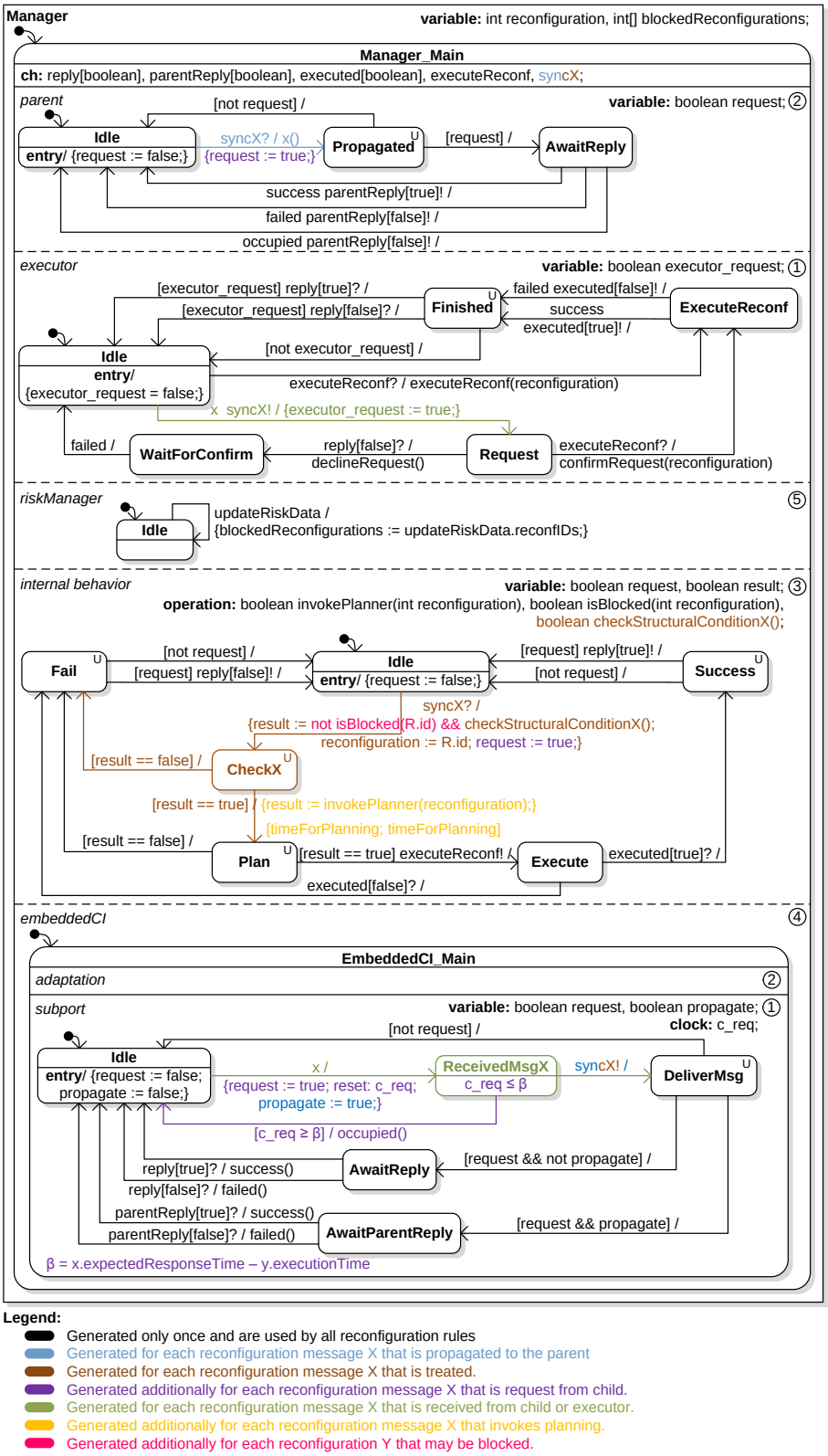


Figure 4.15: Generation Template for the Manager RTSC (cf. [HB13])

In the RTSC, all black states and transitions form the general frame of the RTSC. They are always present and will only be generated once for every manager RTSC. The colored parts are variable and are generated based on the entries of the manager specification. The green parts are generated of each entry of the manager specification. They are used to handle an incoming message. If the message is a request, we additionally generate the purple parts for the corresponding entry. The brown parts are generated for each message that is treated. They specify the behavior for checking whether to execute the reconfiguration. If the message is propagated to the parent, we generate the blue parts. The yellow and pink parts provide the functionality for invoking a planner and checking whether a reconfiguration is blocked if this is specified by the corresponding manager specification entry.

The resulting manager RTSC consists of one state Manager_Main with four or five regions. The region parent implements the RM port parent of the manager that is used for communicating with the parent. The region executor implements the executor port for communicating with the executor. The region riskManager contains the communication with the runtime risk manager and is only present if a runtime risk manager is actually used in the reconfiguration controller. The region internal behavior specifies the behavior of deciding whether to execute which reconfiguration. The region embeddedCI implements the behavior of the multi port embeddedCI that is used for communicating with the children. The RTSC follows the standard structure of a multi port RTSC (cf. Section 2.4.2), although we only need the *subport* RTSC for communicating with the children in this case.

The information flow through the manager RTSC depends on the use cases. In Use Case 1, messages reach the manager via the embeddedCI port and are processed by the subport. If the message is treated, the subport triggers the internal behavior which checks whether to execute a reconfiguration. Then, the internal behavior triggers the executor region to send a message to the executor. The executor region waits for an answer from the executor and reports the result to the internal behavior which, in turn, propagates the result to the subport.

If the message is propagated to the parent, the subport triggers the parent directly. In case of a request, the region parent waits for the answer of the parent and reports this answer back to the subport.

In Use Case 2, the executor sends a message to the manager which is processed by the executor region. Then, the executor RTSC triggers the internal behavior and the execution proceeds as for Use Case 1.

If several requests reach the manager at the same time, for example, from different children, we need to serialize these messages to ensure isolation of the reconfiguration operations. This is achieved by the internal behavior that ensures that only one message is treated at a time.

In the following, we provide a detailed, technical description of the generation template and explain how Use Cases 1 and 2 are encoded in the template. An example of a generated manager RTSC is given in Appendix A.6.3.1 for the component RailCabDriveControl.

Use Case 1 starts with a message from a child. Therefore, we start explaining the subport region. For each message x that may be sent by a child, we generate one state ReceivedMsgX and a transition from Idle to that state. This transition receives the message x. If the message is a request, we reset a clock c_req at this transition and add an invariant c_req $\leq \beta$ to ReceivedMsgX. The invariant ensures that the RTSC will return to Idle if the reconfiguration can no longer be executed. This is the case if the time needed for executing the reconfiguration

exceeds the expected response time. In this case, the subport sends an occupied message to the child indicating that the reconfiguration is currently not possible.

The state ReceivedMsgX may also be left via the transition to DeliverMsg. That transition initiates a synchronization via the synchronization channel syncX. The synchronization channel syncX is generated for each message x that is either propagated or treated by the manager. The transition synchronizes either with the internal behavior if the message is treated or with the parent if the message is propagated.

If the message is propagated, the subport synchronizes with the parent. Then, the parent switches from Idle to Propagated and sends the message x to the parent. If the message is a request, the parent switches to AwaitReply while the subport switches to AwaitParentReply. When the parent answers, either by success or failure or occupied. Then, the parent uses the synchronization channel parentReply to report the result back to the subport which, in turn, sends the result back to the child.

If the message received by the subport is treated, the synchronization via syncX causes the internal behavior to switch to CheckX. As its transition action, the transition checks the structural condition of the message x by calling the operation checkStructuralConditionX. This operation implements the structural condition that is specified in the manager specification (cf. Section 4.3.2). If the reconfiguration is safety relevant, then the operation isBlocked checks whether the reconfiguration with the given id is currently blocked by the runtime risk manager. Therefore, it uses the variable blockedReconfigurations that is set by the riskManager any time the runtime risk manager provides new data via updateRiskData. If the structural condition is not fulfilled or the reconfiguration is currently blocked, then the internal behavior immediately switches to Fail. Then it reports the result via the synchronization channel reply to the subport if the message is a request. If it is not a request, both RTSCs return to their Idle states without synchronization. If the structural condition is fulfilled and the reconfiguration is not blocked, the internal behavior switches to Plan and optionally invokes a planner. If the reconfiguration should be executed, the internal behavior synchronizes with the executor region using the synchronization channel executeReconf. Then, the internal behavior waits in state Execute for the result of the execution.

The synchronization via executeReconf causes the executor region to switch from Idle to ExecuteReconf. The corresponding transition sends a message executeReconf to the executor. The reconfiguration to be executed is referred by its ID from the executor specification and encoded by an integer parameter of the message. Then, the executor performs the 2-phase commit protocol and reports the result, either success or failed, to the manager. The executor region reports the result to the internal behavior using the synchronization channel executed. Then, the executor region takes the lower transition from Finished back to Idle. If the message has been a request, the internal behavior reports the result to the instance of the subport that initiated the reconfiguration via the synchronization channel reply. The instance of the subport waits for that synchronization in the state AwaitReply and sends the result, either success or failed, back to the child. This finishes Use Case 1.

In Use Case 2, the executor sends a message x to the manager. This message is processed by the executor RTSC at the transition from Idle to Request. Such transition is generated for each message offered by the RE port of the structured component. This transition, however, may only fire if a synchronization via the synchronization channel syncX with the internal behavior is possible. That, in turn, is only possible if currently no other reconfiguration is executed. Thus, we use the synchronization channel syncX for serializing the messages inside

the manager. If the synchronization is possible, the execution proceeds as for Use Case 1. If the executor region reaches state Finished, however, it takes one of the upper two transitions back to Idle. These transitions synchronize with the transitions from Success to Idle and Fail to Idle in the internal behavior. These transitions enable to treat Use Cases 1 and 2 identical within the internal behavior. This finishes Use Case 2.

4.4.2 Executor Specification

Figures 4.16 and 4.17 show the generation template for the executor RTSC. The template includes the behavior for both, single-phase execution and three-phase execution. The template implements the 2-phase-commit protocol including many variation points that depend on the executor and RE port specification.

In the RTSC, all black states and transitions form the general frame of the RTSC. As for the manager generation template, they are always present and will only be generated once for every executor RTSC. The colored parts are variable and depend on the executor and RE port specification. We generate the blue parts for each message that is offered by the RE port of the component. The purple parts are generated for each reconfiguration rule that the executor may execute. Finally, the brown parts are generated for every reconfiguration rule that is offered by a child in its RE port.

For realizing Use Case 1, the information flows as follows through the executor RTSC: The executor is initially triggered by the manager and receives the request in the events region. The events region triggers the internal behavior that initializes the 2-phase-commit protocol. The implementation of the 2-phase-commit protocol is mainly located in the adaptation region of embeddedCI. The adaptation computes the children that are affected by the reconfiguration. Then it performs the voting by triggering the corresponding subport instances that are connected to the affected children. Then, the reconfiguration is executed. In case of single-phase execution, the adaptation region triggers the subport instances again. After the execution of the child reconfigurations is finished, the adaptation reports the result to the internal behavior. Then, the internal behavior executes the reconfiguration for the structured component. In case of three-phase execution, the adaptation and the internal behavior execute the three phases of the reconfiguration. The adaptation triggers the subport instances that are connected to the affected children while the internal behavior executes the local reconfiguration operations. After the reconfiguration has been completely executed, the internal behavior notifies the events region that the reconfiguration is completed. Then, the events region notifies the manager. This finishes Use Case 1 for the executor.

In Use Case 2, the parent region receives a message from the parent. This message is forwarded to the events region which, in turn, forwards the message to the manager. Then, the manager answers with the decision and, if the request is confirmed, with the reconfiguration to be executed. Then, the 2-phase-commit protocol is executed as in Use Case 1 except for one difference. After finishing the voting phase, the adaptation triggers the parent region for sending the voting result back to the parent. Then, the parent region waits for the answer of the parent and triggers the adaptation region after the answer has been received. In case of three-phase execution, this is repeated for each of the three phases. Finally, the parent region informs the parent that the reconfiguration has been completed. This finishes Use Case 2.

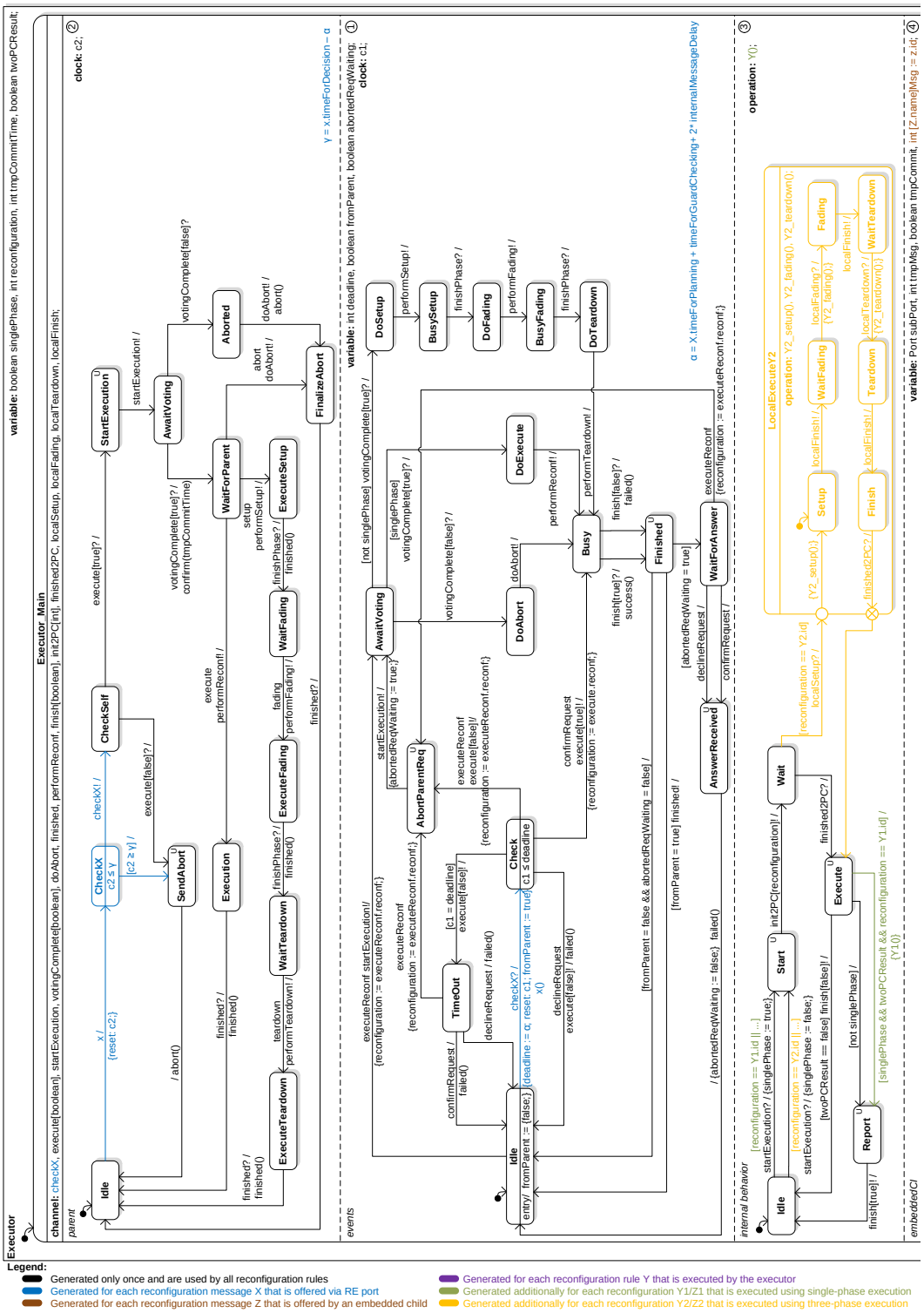


Figure 4.16: Generation Template for the Executor RTSC (Pt. 1)

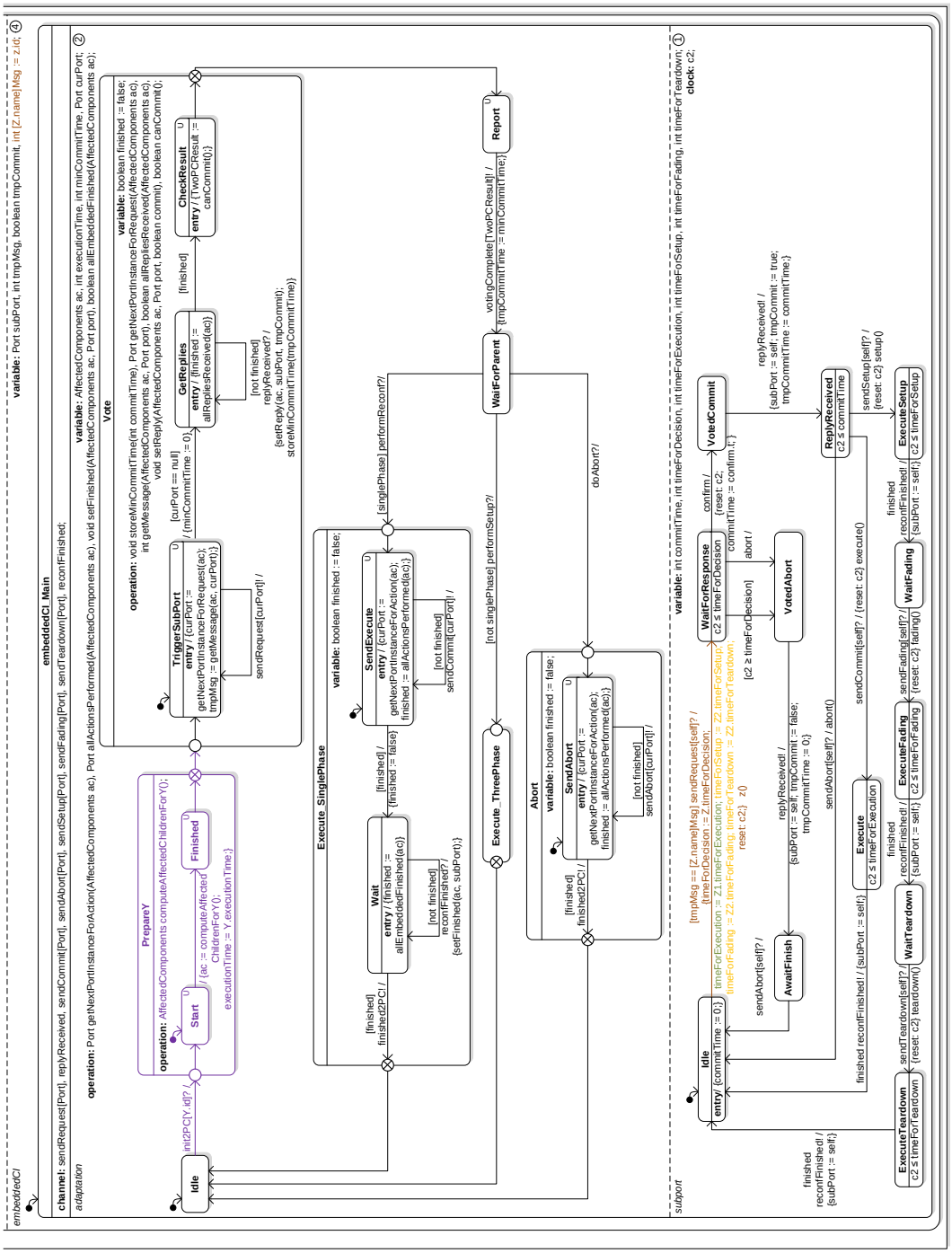


Figure 4.17: Generation Template for the Executor RTSC (Pt. 2)

In the following, we provide a detailed, technical description of the generation template and explain how the Use Cases 1 and 2 are encoded in the template. An example of a generated executor RTSC is given in Appendix A.6.3.2 for the component RailCabDriveControl.

In Use Case 1, the events region receives a message executeReconf from the manager. This message is processed by the transition from Idle to AwaitVoting. The message contains the ID of the reconfiguration rule to be executed as a parameter. In addition, the transition from Idle to AwaitVoting synchronizes with the internal behavior via startExecution. The corresponding transitions from Idle to Start set the variable singlePhase to true if the reconfiguration needs to be executed with single-phase execution or false if the reconfiguration needs to be executed with three-phase execution. Then, the internal behavior starts the 2-phase-commit protocol using the transition from Start to Wait by synchronizing with the adaptation RTSC in embeddedCI via init2PC. The reconfiguration to be executed is encoded in the selector expression.

The RTSC of the multi-port embeddedCI encodes the main logic for executing the 2-phase-commit protocol and controlling its different stages. We use the adaptation RTSC for synchronizing the communication with the children that are affected by the reconfiguration. The actual communication with the children is contained in the subport RTSC.

If the adaptation RTSC is triggered via init2PC, it enters the PrepareY state. We generate such state for each reconfiguration Y that can be executed by the executor. In this state, we compute which children are affected by the reconfiguration using the function computeAffectedChildrenY(). The result is saved in a temporary data structure of type AffectedComponents that contains the information which reconfiguration needs to be executed on which child. By using this data structure, we achieve that the remainder of the adaptation RTSC is independent of the actual reconfiguration that is executed. We present the definition of AffectedComponents in Appendix A.6.4.1 and an example for computeAffectedChildrenY() in Appendix A.6.4.2. All functions that are recursively contained in embeddedCI are formally specified using story diagrams that we present in Appendix A.6.4.3.

After PrepareY, the adaptation switches to the Vote state. In the Vote state, the votes of all affected children for executing their reconfiguration are requested and collected. In TriggerSubPort, all subport instances communicating with an affected child are triggered by the self-transition using the synchronization channel sendRequest.

The subport RTSC contains one transition from Idle to WaitForResponse for each message z that is offered by a child. The message z to be sent to the particular child is stored in the variable tmpMsg of embeddedCI. The adaptation RTSC stores this message in the variable tmpMsg during its entry action in state TriggerSubPort. Upon synchronization via sendRequest, the subport RTSC uses this variable in its guard for sending the corresponding message z to the child. If the child does not answer in time or answers abort, the subport switches to VotedAbort. If the child answers commit, the subport switches to VotedCommit and stores the commit time in an internal variable.

The adaptation switches from TriggerSubPort to GetReplies after all subports have been triggered. In GetReplies, the adaptation synchronizes with the subport, again, to receive their voting results. We use the synchronization replyReceived for that purpose and transfer the voting results using the variables tmpCommit and tmpCommitTime. After collecting all votes, the adaptation switches to CheckResult and calls the function canCommit() upon entry. The function determines whether all children voted for executing the reconfiguration and whether the minimum commit time sent by the children is greater than the time needed for executing the reconfiguration. If so, it is possible to execute the reconfiguration.

After the voting phase has been finished, the adaptation reports the voting result via votingComplete to the events region. The RTSC either switches to DoAbort or DoExecute, respectively, and triggers either the execution via performReconf or the abortion via doAbort. Then, it waits in state Busy until the execution of the 2-phase-commit protocol has been finished.

The further behavior of the adaptation depends on the voting result and whether the reconfiguration is executed with single-phase or three-phase execution. If the reconfiguration is aborted, adaptation enters the Abort state and triggers all affected subport instances, again, for sending the message to the corresponding children. The subport sends abort at the transition from ReplyReceived to Idle.

If the reconfiguration is executed with single-phase execution, the adaptation enters the Execute_SinglePhase state. In this case, the adaptation triggers all affected subport instances for sending execute to the corresponding children at the transition from ReplyReceived to Execute. If the child executed, it answers with finished after successfully executing the reconfiguration. Then, the subport reports to the adaptation that the child has finished executing the reconfiguration using the synchronization finished. The adaptation waits for these synchronizations in substate Wait of Execute. After all child replies have been received, the adaptation synchronizes with the internal behavior via the synchronization channel finished2PC to report that all child reconfigurations have been completed.

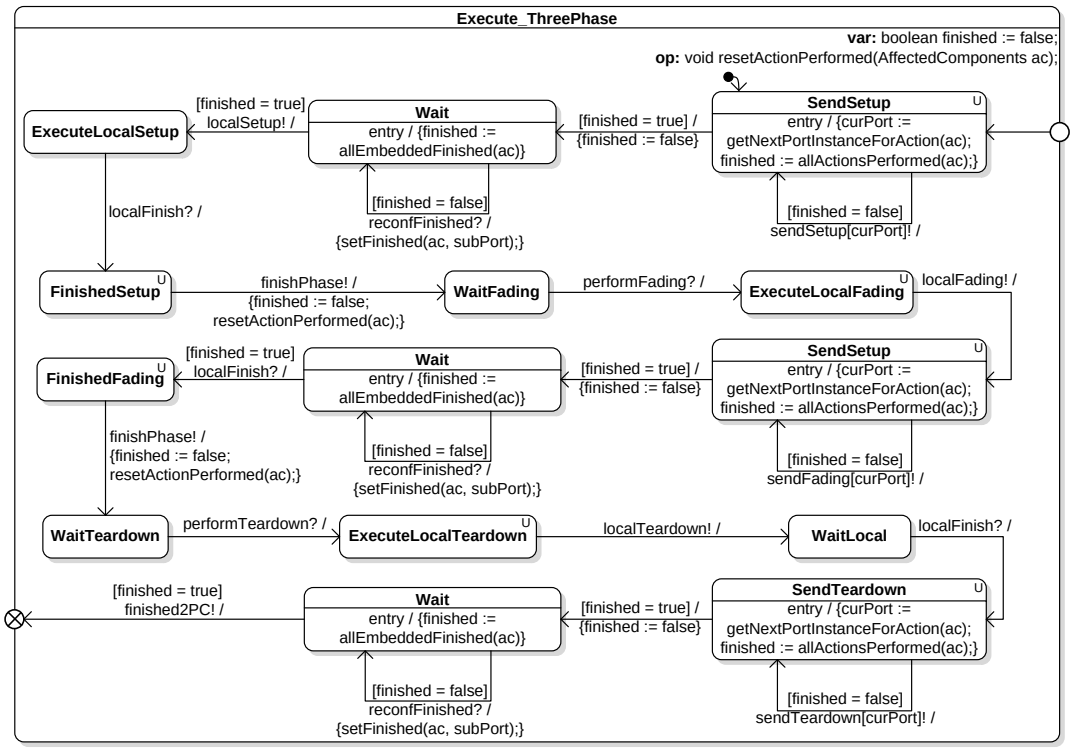


Figure 4.18: Internal Structure of the Execute_ThreePhase State

If the reconfiguration is executed with three-phase execution, the adaptation enters the Execute_ThreePhase state. The internal structure of this state is shown in Figure 4.18. The execution starts in Execute_Setup for executing the setup phase. First, the adaptation triggers

all affected subport instances for sending setup to the corresponding children at the transition from ReplyReceived to ExecuteSetup. If the child executed, it answers with finished after successfully executing the setup phase. Again, the adapatation waits in substate Wait of Execute_Setup for the replies of the subport instances. After all children have performed their setup, the adaptation synchronizes with the internal behavior via localSetup. This causes the internal behavior to enter the LocalExecuteY2 state and to execute the local setup. After it is finished, it synchronizes via localFinished and the adaptation finishes the setup phase. It reports that the phase has been finished to the events region via finishedPhase and waits for the next phase to start. The remaining phases are executed in the same fashion. There are only two notable differences. In the fading phase, the adaptation first triggers the local fading. Without waiting for the finish of the local fading, it triggers the subport instances such that all fading functions are executed in parallel. In the teardown phase, the adaptation also triggers the local teardown first, but waits for the local teardown to finish. After the local teardown is finished, it triggers the subport instances for the last time. After all children have performed their teardown, the adaptation synchronizes via finished2PC with the internal behavior to report that the reconfiguration has been executed successfully.

In case of three-phase execution, the synchronization via finished2PC causes the internal behavior to switch from Finished to Execute. Since singlePhase is false in this case, it immediately proceeds to Report. In case of single-phase execution, the synchronization via finished2PC causes the internal behavior to switch from Wait to Execute. If twoPCResult, which stores the decision on whether to execute or not, is false, the internal behavior switches back to Idle. If twoPCResult is true, then the internal behavior switches to Report and calls the own reconfiguration rule in the transition action. The transition back to Idle synchronizes with the events region to indicate that the execution has been finished.

The synchronization finish causes the events RTSC to switch from Busy to Finished. This transition also sends a message success or failed to the manager in case that the reconfiguration has been executed or aborted, respectively. Finally, the RTSC fires the upper transition from Finished to Idle which finishes Use Case 1.

In Use Case 2, messages reach the executor via the parent port and are processed by the corresponding RTSC in region parent. For each message x that the component offers via its RE port, we generate one state CheckX including transitions from Idle to CheckX and from CheckX to CheckSelf and SendAbort. The transition from Idle to CheckX receives the message x. In CheckX, the parent tries to synchronize via checkX with the RTSC in the events region. The state CheckX contains an invariant $c2 \leq \gamma$. γ is the time for decision specified in the RE port minus the time actually needed for deriving a decision in the manager. If γ is exceeded, it is no longer possible to check whether the reconfiguration can be executed within the time for decision and the RTSC switches to SendAbort. This case will usually happen if the executor is already executing a reconfiguration when the message x arrives. In this case, a synchronization with the events region via checkX is not possible.

If the synchronization via checkX is possible, the parent triggers the events region and switches to CheckSelf. The RTSC in the events region switches from Idle to Check and forwards message x to the manager. We generate such transition from Idle to Check for each message x in the RE port specification. In Check, the events region waits for the decision of the manager. If the manager sends declineRequest, the events region reports the result via the synchronization channel execute and the parent switches to SendAbort. If the manager sends confirmRequest, then the reconfiguration may be executed. The events region reports that re-

sult via execute to the parent which switches to AwaitVoting and triggers the internal behavior. Then, the voting phase is performed as in Use Case 1.

In contrast to Use Case 1, the voting result is returned to the parent that sends the voting result to the parent component. If the component needs to abort, the parent switches via Aborted to FinalizeAbort. In FinalizeAbort, it synchronizes via finished with the events region and both RTSCs return to their Idle states. If the component has committed the reconfiguration, the parent waits in WaitForParent for the decision of the parent component. If the parent component aborts the reconfiguration, the parent FinalizeAbort. The transition from WaitForParent to FinalizeAbort synchronizes with the adaptation RTSC of embeddedCI via doAbort to abort the child reconfigurations. Afterwards, it synchronizes with the events region via finished as described above. If the parent decided to execute the reconfiguration, the parent region switches from WaitForParent to either Execution or to ExecuteSetup depending on whether the reconfiguration is executed with single-phase or three-phase execution. The corresponding transitions synchronize via performReconf with the adaptation RTSC of embeddedCI. Then, the child reconfigurations are triggered as in Use Case 1. In three-phase execution, the adaptation synchronizes with parent via finishPhase after completely executing one of the phases. The parent then reports that the phase has been finished to the parent. After all child reconfigurations and the own reconfiguration have been performed, the internal behavior synchronizes with the events region that informs the manager about the result of the execution. Finally, events takes the lower transition from Finished to Idle and synchronizes with parent via finished. That synchronization causes the transition from Execution to Idle to fire. This transition sends finished to the parent component which finishes Use Case 2.

In the executor it may happen that two requests arrive at the same time: one from the parent component, the other one from the manager. These interleavings are handled in the events region using the states AbortParentReq, WaitForAnswer, and AnswerReceived as well as the variable abortedReqWaiting. If the events region is in state Check as part of Use Case 2, it may happen that the manager sends executeReconf instead of confirmRequest. In this case, the manager already treated a child request according to Use Case 1 when the executor forwarded the parent request. Then, the events region first treats the reconfiguration that was requested by the manager according to Use Case 1. Therefore, it switches from Check to AbortParentReq. This transition aborts the parent request by synchronizing via execute with the parent. As a result, the parent switches to SendAbort and finishes the request. However, the request by the parent still resides in the message queue of the manager. Therefore, after finishing the reconfiguration, the events region does not return to Idle, but it switches to WaitForAnswer. In this state, it waits for the confirmRequest or declineRequest message from the manager. If one of these messages is received, the events RTSC switches to AnswerReceived and immediately replies failed to the manager. In WaitForAnswer, it may also happen that another executeReconf message arrives. Then, the manager has treated another child request before the request of the executor. Then, the events RTSC switches back to AbortParentReq and the procedure repeats as described before.

4.5 Verifying the Reconfiguration Specification

We need to verify that the specified reconfiguration behavior of a structured component fulfills all of the ACI-T properties of the 2-phase-commit protocol. These properties guarantee

that the reconfiguration behavior of the structure component is correct and, thus, safe. In our approach, formal verification is enabled by the operational behavior specifications for manager and executor in terms of RTSCs.

For verifying the ACI-T properties of the 2-phase-commit protocol, we need to verify the following yet informal properties:

1. If the executor decides to execute (abort), then all affected children execute (abort) (**Atomicity**).
2. The reconfiguration rules cannot produce an inconsistent CIC (**Consistency**).
3. The executor will execute no other reconfiguration than the one requested by the manager (**Consistency**).
4. At any time, at most one reconfiguration is executed (**Isolation**).
5. The RTSCs of manager and executor are free from deadlocks (**Timing**).
6. Each reconfiguration is executable (**Timing**).

Properties 1, 3, and 4 can already be guaranteed by the correctness of the generation templates given in Section 4.4. Therefore, they do not need to be verified again for a particular structured component. The correctness of the generation templates with respect to these three properties has been verified using UPPAAL [HB13, Vol13].

Property 2 specifies that reconfigurations may not produce an inconsistent CIC. A CIC may either be syntactically inconsistent or semantically inconsistent. A CIC is syntactically inconsistent if it violates the conditions for syntactical correctness that we introduced in Section 3.2. In our approach, the CSDs guarantee that CICs remain syntactically consistent after a reconfiguration due to syntactic restrictions. Thus, no further check is necessary. A CIC is semantically inconsistent if the instantiated component instances, port instances, and connector instances do not constitute a desired functional behavior. In the worst case, the component may even be unsafe. As an example, consider a RailCab that drives as part of a convoy as a member but which does not have an instance of MemberControl (cf. Figure 4.7) although it switched the controller. Such situations cannot be prevented by syntactic rules but need to be verified for each structured component as we describe in Section 4.5.1.

Finally, Properties 5 and 6 specify the conditions for a correct timing specification. In a platform-independent model, we may verify whether the timing requirements provided in our declarative, table-based specification are satisfiable. If they are satisfiable, there may exist a hardware platform that enables to execute the reconfiguration behavior without violating the timing requirements. After deriving a platform-specific model that includes a platform model and a deployment of components to hardware nodes [PMDB14], we may already check at design-time whether the execution of the reconfigurations on the hardware platform fulfills the imposed requirements. We describe the verification of the timing specification in detail in Section 4.5.2.

In combination, both verification steps and the verified generation templates enable to verify the reconfiguration behavior of our components completely with respect to the ACI-T properties. The only part of the reconfiguration behavior that cannot be formally verified using model checking is given by the implementations of the fading functions. Their correctness needs to be determined using our approach for MIL simulation introduced in Chapter 6.

4.5.1 Consistency

We ensure consistency by verifying that the reconfiguration behavior cannot produce a semantically inconsistent CIC. However, it is not possible to automatically derive from the component model which CICs are semantically inconsistent and which are not. Therefore, a developer needs to provide this information explicitly. In the following, we introduce three possibilities for specifying semantically inconsistent CICs. These are forbidden CICs (Section 4.5.1.1), architectural invariants (Section 4.5.1.2) and properties in temporal logic (Section 4.5.1.3). For each of these, we describe an approach for formal verification.

4.5.1.1 Forbidden CICs

A forbidden CIC is a particular CIC or part of a CIC that may never occur for a given component. As an example, consider the CIC in Figure 4.19 that defines an excerpt of an instance inconsistent of the component RailCabDriveControl. It specifies the situation where both, MemberControl and ConvoyCoordination, are instantiated. In our example, we only allow RailCabs either to be the coordinator or a member but not both at the same time. Therefore, we consider this CIC as semantically inconsistent and, thus, as forbidden.

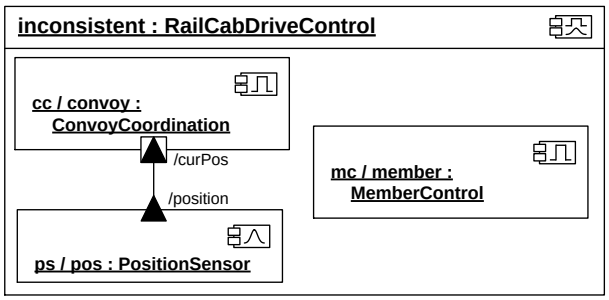


Figure 4.19: Example of a Forbidden CIC

We have two alternatives for verifying that a forbidden CIC may not occur. First, we may perform a reachability analysis [HSE10] using our framework described in Appendix C. In this approach, we compute all possible configurations of a component starting from the initial configurations. Then, we may check whether the forbidden CIC occurs in any of these configurations. Second, we can use the inductive invariant approach by Becker et al. [BBG+06]. This approach provides a proof that a forbidden CIC cannot have been produced out of a semantically consistent CIC by applying a backward application of typed attributed graph transformation rules. Backward application means that they match the RHS and enforce the LHS of the typed attributed graph transformation rule. This approach has been extended towards supporting story diagrams with few branches in the control flow by Meyer [Mey09]. The benefit of this approach is that it may even be applied if the number of configurations of a component instance is unbounded.

Applying the inductive invariant approach requires to translate the CSDs to normal story diagrams. This translation has been defined by Tichy [Tic09, p. 77ff]. The basic idea is using the metamodel of our component model (cf. Appendix D.1) as a type graph for the story diagrams.

4.5.1.2 Architectural Invariants

Our component model enables to specify architectural invariants based on component SDDs as described in Section 3.5. Any CIC that violates an architectural invariant is considered as semantically inconsistent. An example is given by the component SDD in Figure 3.21 that ensures a correct ordering of the subport instances of the multi port instance refDistProvider of a coordinator RailCab. This constraint ensures that updates of reference speed and distance are distributed in the correct order among the members. Component SDDs are more expressive than forbidden CICs because they enable specifying conditional constraints and support quantification based on first-order logic.

Component SDDs may be verified by a reachability analysis on the CSDs. First, the CSDs need to be translated to normal story diagrams as defined by Tichy [Tic09, p. 77ff]. Then, the component SDDs first need to be translated to normal SDDs [KG07] by applying the transformation by Tichy to the component story patterns that are contained in the pattern nodes. Thereafter, the resulting normal SDDs are translated to story diagrams by applying the concept of Ahmadian et al. [AAB+11, p. 38ff]. Then, the SDD is fulfilled if and only if the resulting story diagram can be executed successfully on each configuration of the component. We can check this condition by performing a reachability analysis on the resulting set of story diagrams using the initial configuration of our structured component. In the reachability analysis, we check whether there exists a configuration to which the story diagram resulting from the SDD cannot be matched. We may utilize the reachability analysis introduced in [HSJZ10, HSE10] for this purpose.

4.5.1.3 Properties in Temporal Logic

Temporal logic constraints based on CTL and LTL (cf. Section 2.2.2) enable to specify constraints on the evolution of a CIC. In our example, we want to specify that a RailCab may not directly switch from being coordinator to being member. Such properties may be expressed by graph-based variants of CTL and LTL such as quantified CTL (QCTL, [Ren06]), graph-based LTL (GLTL, [Ren08]), or first-order TCTL (FO-TCTL, [Suc11, SHS11]).

Verifying such properties requires a graph-based model checking, for example, based on GROOVE [KR06, Ren08] or CheckVML [SV03]. Applying GROOVE on CSDs requires to translate the component model and CSDs into a typed attributed GTS. Then, the component model defines the type graph and the initial configuration of the structured component defines the initial graph. The CSDs need to be translated in two steps. First, they need to be translated to story diagrams as defined by Tichy[Tic09, p. 77ff]. Second, the story diagrams need to be translated to typed attributed graph transformation rules as defined by Reineke [Rei07].

4.5.2 Timing

We ensure a correct timing specification by applying timed model checking on the platform-independent component model as described in Section 4.5.2.1. If the model checking encounters a deadlock or a reconfiguration rule that cannot be executed, the timing requirements in our declarative, table-based specification are not satisfiable. After deriving a platform model including a deployment of component instances to hardware nodes [PMDB14], we need to check whether the timing requirements are satisfied by the platform model as described in Section 4.5.2.2.

4.5.2.1 Ensuring Correct Timing by Timed Model Checking

We apply timed model checking on the RTSCs of manager and executor for guaranteeing that they are free from deadlocks and that they enable to execute each reconfiguration rule. In order to achieve an efficient and scalable verification approach, we verify the timing requirements separately for each structured component. This is enabled by using stubs for the parent component as well as the children. These stubs abstract from the internal behavior of the parent and the children. They only implement the relevant behavior based on the interfaces of the RM and RE ports that is necessary for checking a correct vertical integration of the component with respect to timing.

For applying timed model checking, we need to translate the RTSCs of manager and executor into an NTA as illustrated in Figure 4.20. In particular, we obtain one timed automaton for each region of the manager RTSC and of the executor RTSC. In addition, we need one automaton that defines the behavior of the connector between manager and exeuctor. Finally, we add two parent stubs for the parent component and two child stubs for each embedded component part. The number of child stubs is equal to the number of timed automata that are generated for the subport RTSCs of the embeddedCI ports of manager and executor (cf. Section 4.4). The arrows illustrate the information flow between the timed automata that results from the synchronizations used in the RTSCs and the messages being sent.

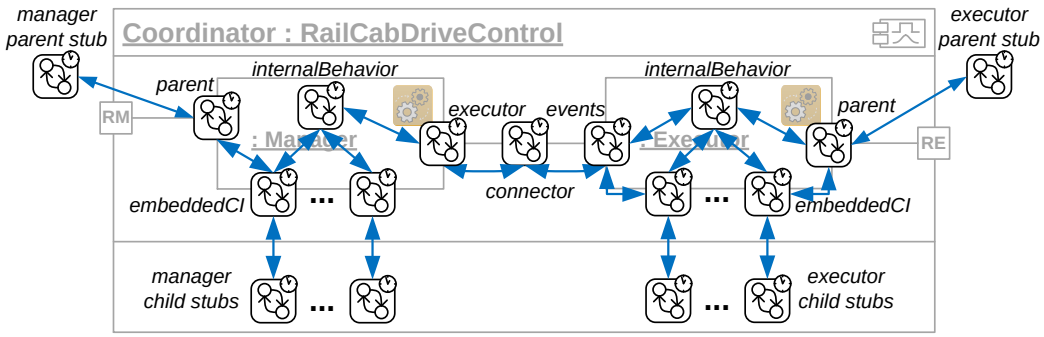


Figure 4.20: Sketch of the Generated NTA

Figure 4.21 shows an executor child stub that was generated based on the RE port specification of ConvoyCoordination for verifying the correct timing of RailCabDriveControl.

The behavior of the executor child stub is as follows. It waits in Idle for being triggered by RailCabDriveControl for executing a reconfiguration. In this case, only AddConvoyMemberAtPos may be triggered by RailCabDriveControl. This corresponds to the channel childAddConvoyMemberAtPos and the executor child stub switches to ReceivedAddConvoyMemberAtPos. As part of the transition, the executor child stub nondeterministically chooses whether it will commit or abort the request. The result is stored in doCommit. In addition, we assign the time values for the timeForDecision, the timeForExecution, and the minCommitTime that are contained in the RE port interface specification to the eponymous variables. The child stub now waits in ReceivedAddConvoyMemberAtPos until the timeForDecision has expired. Then, it either synchronizes via msgChildAbort and returns to Idle or it synchronizes via msgChildCommit and switches to Committed. In Committed, the invariant ensures that the executor child stub will only rest in the Committed state until the minCommitTime expires. As a result, a deadlock occurs if RailCabDriveControl does not sent a decision whether to execute or abort the reconfiguration in

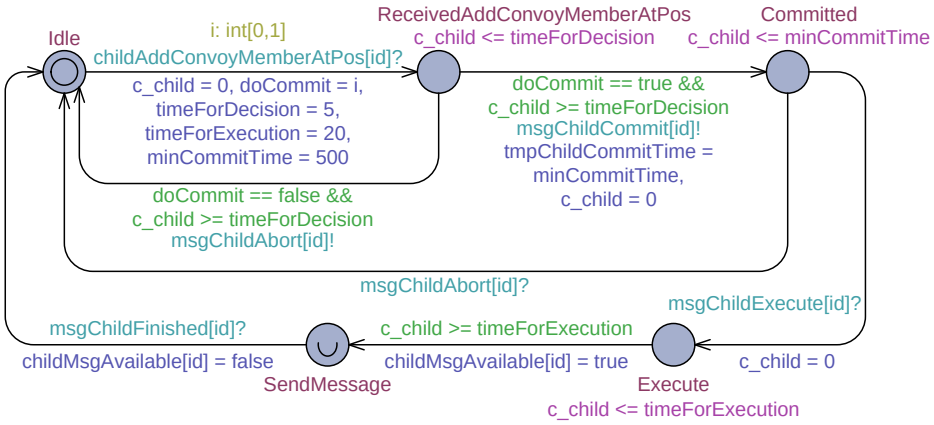


Figure 4.21: Child Stub Representing ConvoyCoordination for the Verification of RailCabDrive-Control

time. Based on the decision by RailCabDriveControl, the executor child stub either switches to Idle (abort) or to Execute (execute). In Execute, the executor child stub rests as long as the timeForExecution has not expired. Then, it switches to SendMessage to check whether RailCabDriveControl may accept the result message, which is then send at the transition from SendMessage to Idle.

Examples for specifying parent stubs and manager child stubs may be found on the website [Hei13] that accompanies our paper [HB13]. The resulting NTA may then be verified using UPPAAL [BDL+06b]. In particular, we need to check that the NTA contains to deadlock and that the state Report of the internal behavior region of the executor RTSC (cf. Figure 4.16) may be reached for each reconfiguration rule that is specified in the executor specification.

4.5.2.2 Ensuring Correct Timing after Deployment

After creating a platform model including a deployment, we need to check whether the deployment satisfies the timing requirements. As a basis, we need to compute WCETs for the behavior of manager and executor as well as for the execution of the CSDs on the given platform. Since the behavior of manager and executor is defined by RTSCs, we may use the WCET analysis defined by Burmester [Bur06, BGST05] for this purpose. In addition, we need to apply the WCET analysis for CSDs presented by Tichy et al. [TGS06, THHO08] for checking whether the hardware platform satisfies the WCET requirements for the CSDs given in the executor specification. Thereafter, we need to check that the vertical integration of the components with respect to timing is still correct.

Expected Response Time First, we need to check whether the timing of child requests that are propagated to the parent by the manager is correct. In this case, the interface specifications of the RM ports of the structured component and of the child that sends the request need to be consistent. For being consistent, the expected response time t_{resp} of the structured component must be smaller than the expected response time t_{resp}^{sub} of the child such that the

response arrives at the child within time as defined by the formula:

$$t_{resp} \leq t_{resp}^{sub} - 2 \cdot m^{sub} - e_{overhead}$$

where m^{sub} is the message delay for a message sent to the child and $e_{overhead}$ denotes the WCET for executing the internal behavior of the manager.

Time For Voting Second, we need to check whether the time for decision that is contained in the interface specification of the RE port is still satisfied. In particular, if the reconfiguration that is associated with the interface entry of the RE port involves reconfigurations of one or more children, the time for decision needs to be large enough to include the execution of the voting phase by the children.

Based on the execution of the voting phase illustrated in Figure 4.6, four time values contribute to the time for decision d_d. First, we need to consider the maximum time for decision d_d^{sub} among the children that are affected by the reconfiguration if all children are executed in parallel. Otherwise, we first need to sum up the times for decision of all children that are executed sequentially on the same hardware node before calculating the maximum. Here, we additionally need to consider the message delay m^{sub} for sending the request to the child and the message delay for the voting decision being sent back. Second, we need to consider the time for planning t_{plan} (cf. Section 4.3.2) that is specified in the manager specification. Third, we need to add two times the message delay m for a message that is exchanged between manager and executor. Finally, we need to consider $e_{overhead}$, which is the WCET for executing the internal behavior of manager and executor that includes, for example, checking the structural condition in the manager and initializing the 2-phase-commit protocol in the executor. Then, the time for decision of the structured component must be greater or equal to

$$d_d \geq \max_i \{d_{d,i}^{sub} + 2{\cdot}m_i^{sub}\} + t_{plan} + 2{\cdot}m + e_{overhead}$$

where $d_{d,i}^{sub}$ refers to the time for decision of the i^{th} child that is affected by this reconfiguration and m_i^{sub} is the message delay for a message sent to that child. We assume that a message sent from parent to child takes as long as a message in the opposite direction. The computation of d_d is the same for single-phase and three-phase execution.

Time For Execution Third, we need to check whether the time for execution that is contained in the interface specification of the RE port is still satisfied. In particular, if the reconfiguration that is associated with the interface entry of the RE port involves reconfigurations of one or more children, the time for execution needs to be large enough to include the child reconfigurations.

The time for execution d_e denotes the maximum time that the component needs to execute the reconfiguration in the execution phase of the 2-phase-commit protocol. For single phase execution, three time values contribute to the time for execution. First, we need to consider the maximum time for execution d_e^{sub} among the children that are affected by the reconfiguration. This includes, again, two times the message delay m^{sub} for sending a message to a child. Second, we need to consider the WCET of the reconfiguration rule executed by

the component itself. Finally, we need to consider the WCET $e_{overhead}$ for managing the 2-phase-commit protocol in the executor. Then, the time for execution d_e of the structured component must be greater or equal to

$$d_e \geq \max_i \{d_{e,i}^{sub} + 2 \cdot m_i^{sub}\} + d_{reconf} + e_{overhead}$$

where $d_{e,i}^{sub}$ refers to the time for execution of the i^{th} child that is affected by this reconfiguration and m_i^{sub} is the message delay for a message sent to that child.

If we execute reconfigurations based on three phase execution, we need to apply the above formula separately for each execution phase. We cannot provide a single value for three-phase execution because a phase can only be finished after all children have finished their executions. As a result, the duration of a phase for a single child may be extended by a waiting period where it waits for another child to finish its execution as illustrated in Figure 4.8. Thus, we need to compute the maximum duration separately for each phase.

Minimum Commit Time Finally, the minimum commit time d_{ct} denotes the minimum time that the component sticks to a commit. A structured component instance may only stick to the commit at most as long as the children do. Thus, we need to consider the minimum among the minimum commit times d_{ct}^{sub} of all affected children. In addition, we need to subtract two times the message delay m^{sub} because the commit time starts after the child sent the commit and the query to execute needs to reach the child before the commit time expires. Thus, the minimum commit time d_{ct} of a structured component must be less or equal to

$$d_{ct} \leq \min_i \{d_{ct,i}^{sub} - 2 \cdot m_i^{sub}\}$$

where $d_{ct,i}^{sub}$ refers to the minimum commit time of the i^{th} child that is affected by this reconfiguration and m_i^{sub} is the message delay for a message sent to that child.

4.6 Implementation

We implemented the concepts introduced in this chapter as part of the MECHATRONICUML Tool Suite. In particular, we integrated the implementation into the plugins reconfiguration and reconfiguration.ui shown in Figure 3.22 on page 65.

We extended the metamodel in plugin reconfiguration such that it includes our reconfiguration controller including the declarative, table-based specification. A class diagram of this metamodel is presented in Appendix D.2.1.

The plugin reconfiguration.ui extends the component editor such that it enables to specify reconfigurable components including their reconfiguration controllers. In addition, it contains the generator that enables to generate RTSCs for manager and executor based on the generation templates given in Section 4.4. The generator has been implemented in QVT Operational [Gro11b]. In addition, we support to convert a non-reconfigurable component into a reconfigurable component.

4.7 Assumptions and Limitations

Our approach for the transactional execution of reconfiguration underlies the following assumptions and limitations:

- Any reconfiguration that has been started can be finished successfully. In particular, we assume that no hardware failures occur while executing a reconfiguration.
- All monitoring is performed by atomic components that accumulate the monitoring data and provide accumulated data to the manager of the parent component.
- The reconfiguration controller and the generation templates for deriving an operational behavior specification for manager and executor have only been defined for structured components because of the missing concept of quiescence for atomic components in MECHATRONICUML (cf. Section 4.2.3).
- We may only trigger at most one reconfiguration on each child of a structured component instance when executing a CSD for the structured component instance.

In addition, our implementation underlies the following limitations:

- The generation templates do not support input and output parameters of CSDs as they are used, e.g., by the CSD in Figure 3.14 on Page 54.
- The concept for verifying consistency introduced in Section 4.5.1 has not yet been implemented.

4.8 Related Work

Section 3.7 reviewed component models and architecture description languages that support reconfiguration of the software architecture at runtime. Only few of them consider reconfiguration of hierarchical components supporting a transactional execution of reconfigurations. We review their reconfiguration capabilites in Section 4.8.1. Thereafter, we discuss related approaches for achieving quiescence in a system in Section 4.8.2.

4.8.1 Approaches Supporting Reconfiguration of Hierarchical Components

Our approach is inspired by the reconfiguration concepts of Fractal [BCL+06, LLC10] which has been extended to distributed execution in [BHR09]. Their concept extends each reconfigurable component with a reconfiguration interface and a reconfiguration executor for executing reconfiguration scripts. We have adopted the concept of a reconfiguration executor and extended the remote reconfiguration invocation. In contrast to our approach, Fractal starts reconfigurations optimistically and performs a roll-back in case that the reconfiguration is not possible. As described in Section 4.2, this is not safe in mechatronic systems. Their approach achieves ACI properties as well, but does not consider timing of reconfigurations. In addition, we support a higher level modeling language for modeling reconfigurations rather than implementing them as a script. The approach by Boyer et al. [BGP13] also follows a roll-back approach for achieving reliable reconfiguration, but their approach neither treats

hierarchy nor achieves any of the ACI-T properties. The SOFA 2.0 component uses component controllers similar to Fractal and to our approach, called micro-components, but does not provide a transactional execution of reconfigurations [HB07].

The architecture description language GeReL [EW92] supports a separation of concerns between functional and reconfiguration behavior. It uses a first-order logic to determine whether a reconfiguration can be executed or not. This ensures consistency of the modified system, while their execution model guarantees atomicity of reconfigurations. Their approach, however, does explicitly support hierarchical components. In addition, they do not consider real-time properties.

Pop et al. [PPO+12] introduce a mode change operation of embedded real-time systems based on the SOFA-HI [PWT+08] component model. In their approach, each mode of a component instance corresponds to a configuration. They also separate functional and reconfiguration behavior and enable mode changes across different levels of hierarchy. Consistent modes of a component and its children are specified by property networks. In contrast to our approach, they cannot ensure atomicity if a child is currently not able to reconfigure and they do not provide a formal verification support for checking for a correct timing of reconfigurations.

The approach by Hang et al. [HCH12, HQCH13] implements a composable mode change operator based on the ProCom component model [VSC+09]. As in [PPO+12], modes correspond to component configurations. Similar to our approach, they use dedicated reconfiguration components that are hierarchically connected. Reconfiguration requests may traverse the hierarchy bottom-up or top-down. In [HH13], they adopted our approach for executing reconfigurations in two phases and use our verification approach introduced in Section 4.5.2. In contrast to our approach, they do not provide explicit real-time properties regarding the execution of reconfigurations in their specification.

The framework by de Oliveira et al. [DOLS13] uses several autonomic managers for adapting cloud applications in a coordinated fashion. Their autonomic managers have a similar purpose as our reconfiguration controller but are horizontally composed. They share information using event-based coordination protocols for improving adaptation decisions but do not consider transactional execution or real-time properties.

The Rainbow framework [GCH+04, CGS09] provides an implementation of the reference architecture MAPE-K [IBM06] that targets business information systems. Their concept defines an adaptation manager and an adaptation executor that closely correspond to the manager and executor in our approach. However, their approach does not respect component encapsulation for a hierarchical component architecture. In addition, their approach does neither support real-time properties nor guarantee ACI-T properties.

Similarly, Zhang et al. [ZCYM05] provide an approach for safe adaptation of component-based systems. Their approach uses one central adaptation manager that orchestrates the adaptation process, and several agents that are attached to the components and perform their modification. If an adaptation cannot be finished successfully, they perform a roll-back to the previous configuration. Thus, their approach guarantees ACI properties for the execution of reconfigurations but no real-time properties. In addition, their approach does not explicitly consider hierarchical components.

The approach by Edwards et al. [EGT+09] uses meta-level components for implementing a self-adaptation control loop similar to MAPE-K [IBM06] based on a hierarchical component model. The meta-level components fulfill a similar purpose as our reconfiguration con-

troller by monitoring the components on the hierarchy level below, by evaluating whether and how to adapt, and by executing the resulting adaptation plan. Similarly, Vromant et al. [VWMA11] connect several MAPE control loops that are located on the same hierarchy level following a master-slave pattern. Then, the control loops communicate for deriving a consistent adaptation strategy. Both approaches do not explicitly connect meta-level component or MAPE control loops, respectively, on different hierarchy levels such that hierarchical execution are not supported and ACI-T properties cannot be guaranteed.

EUREMA [VG14] supports the specification of self-adaptation feedback loops based on MAPE-K [IBM06] using a graphical notation called feedback loop diagrams. The approach supports to use and to coordinate multiple feedback loops in a single system. In addition, feedback loops on different architectural levels may be connected and coordinated by using layer diagrams. Weyns et al. [WSG+13] discuss different design patterns for connecting multiple MAPE feedback loops in a system. With respect to their pattern, our approach is based on the hierarchical control pattern. In contrast to our approach, the approaches do not support real-time constraints and do not explicitly consider ACI-T properties. However, EUREMA satisfies isolation of adaptations.

Finally, the fault-tolerant component model by de Lemos et al. [dLdCGFR06] partitions component behavior into normal and abnormal (exception) behavior. We follow the same idea by separating normal behavior and reconfiguration behavior. Their approach provides horizontal propagation of exceptions, but not propagation to parent components. With a similar objective, Strunk and Knight [SK06] provide a dependable reconfiguration approach for hard real-time systems where a system moves from one configuration to another one with degraded functionality in case of a failure. The approach, however, neither considers components nor hierarchy, but it ensures by formal proofs that any reconfiguration can be executed successfully.

4.8.2 Quiescence of Components

In the approach by Kramer and Magee [KM98], the conditions for quiescence require all affected component instances and all component instances that are connected to them to be passive. In essence, this means that the component instances are shut down and no longer executed. After the reconfiguration has been finished, they are started, again. Given an NMS such as a convoy of RailCabs, this is not a viable approach. In the worst case, it requires that all the RailCabs in a convoy need to shutdown if one RailCab needs to perform a reconfiguration. This, in turn, requires the RailCab to stop for each reconfiguration, which is not desirable. The concept of tranquility [VEBD07] relaxes the conditions on quiescence by Kramer and Magee [KM98]. The major drawback of their approach is that tranquility is not predictable, i.e., it cannot be decided whether a component instance will become tranquil in a given amount of time.

The approaches by Chen et al. [CHS01], Ghafari et al. [GJSH12], and Panzica La Manna [PLM12] support the evolution of a software architecture of a business information system where component instances are upgraded to a new version implements the same behavior. The new component instance either fixes bugs of the old component instance or provides quality of service that suites better to the current requirements. The approaches by Chen et al. and Ghafari et al. work similar to our fading functions. The component instance to be removed and the new component instance are executed in parallel. Then, new transactions

are handled by the new component instance while the old component instance remains active until it has processed all pending transactions. The approach by Panzica La Manna tries to transfer the complete state of the old component instance to the new one such that the new component instance may continue processing all transactions that have been started using the old component instance. If this is not possible, the approach applies a version consistent update as defined by Ma et al. [MBG+11] that applies a similar strategy as the approaches by Chen et al. and Ghafari et al. In essence, all of these approaches try to preserve the functional behavior of the system during and after the update with the exception of corrected bugs and improved quality of service characteristics. In contrast, our approach explicitly aims at modifying the functional behavior, e.g, if a RailCab joins a convoy. Therefore, we require to add or to entirely remove component instances from the software architecture, which is not supported by these approaches. In addition, they do not consider the real-time constraints and the physical movement of the system, e.g., its current speed or its distance to other vehicles.

4.9 Summary

This section introduces an approach for executing reconfigurations in a hierarchical component model for self-adaptive mechatronic systems. On the syntactic level, we extend each structured component by a dedicated reconfiguration controller that contains the reconfiguration behavior. The reconfiguration controller contains a manager, an executor, and an optional runtime risk manager. The manager defines whether and how the component shall reconfigure. The executor is responsible for executing reconfigurations with respect to hierarchy. The runtime risk manager defines which reconfigurations may be executed such that the functional safety of the system is retained. On the semantic level, our reconfiguration controller implements a variant of the 2-phase-commit protocol [BHG87, ch. 7] that has been adapted to the domain of mechatronic systems. As a result, our approach satisfies ACI-T properties for the execution of reconfigurations, i.e., atomicity, consistency, isolation, and a correct timing, even for reconfigurations spanning vertical compositions of components. Furthermore, our approach respects encapsulation of components. For the execution of the reconfiguration, our approach supports a single-phase execution for reconfiguring discrete component instance and a three-phase execution for safely replacing continuous components that contains feedback controllers.

Our approach relieves the component developer from specifying the complex implementation of the 2-phase-commit protocol by hand for each component. Instead, we provide a concise declarative specification of the behavior of manager and executor based on tables. These tables specify the conditions when to execute which reconfiguration. Then, these tables are used as an input of a generator that automatically derives an implementation of the 2-phase-commit protocol based on RTSCs. The generated RTSCs fulfill atomicity and isolation by construction. In addition, the generated RTSCs and the CSDs that define the modification of the CIC serve as inputs for our verification procedure that verifies consistency and a correct timing of the reconfiguration behavior.

5 Verifying Refinements based on Test Automata

Self-adaptive mechatronic systems are often intended to operate as part of an NMS. As an example, RailCabs are intended to operate in convoys. Then, the correct functionality of the self-adaptive mechatronic system and, in particular, its safety do not only depend on its own correctness but also on the correct interaction with other AMS inside the NMS. The interaction, in turn, is typically defined by complex application-level communication protocols. These communication protocols define which messages are needed to be exchanged and in which order and time intervals for realizing the intended functionality. Equally, defining the behavior of a single AMS requires to connect the different components of its software architecture using application-level communication protocol as well. As an example, consider the component instances of a RailCab given in Section 3.2 and Appendix A.4.

Due to the safety critical nature of self-adaptive mechatronic systems, we need to formally verify their behavior based on model checking [CGP00, BK08] for guaranteeing their correctness. On the one hand, this requires to verify the reconfiguration behavior of the components as discussed in Section 4.5. On the other hand, this requires to verify the functional behavior specification of each discrete atomic component that is used in the software architecture. However, the correctness of a single component does not only depend on its own behavior specification but also on the behavior specifications of the components that it needs to interact with. The resulting software architecture as given by a CIC consists of several interconnected component instances. Such CIC, however, cannot be verified using standard model checking tools like UPPAAL [BDL+06b] due to the state-explosion problem [CGP00].

Compositional verification approaches [BCC98] based on the assume/guarantee principle [CGP00, ch. 12] tackle the state explosion problem by decomposing the system into smaller units for verification. Previous works defined such compositional verification approach for MECHATRONICUML as well [GTB+03, Gie03, GS13]. The basic idea of MECHATRONICUML's compositional verification approach is a syntactic decomposition of the functional behavior into RTCPs and components. It requires that RTCPs are specified independent of components. Then, each discrete port of a discrete atomic component refines one role of a RTCP, which results in one RTSC for each discrete port. These port RTSCs are then composed to a component RTSC as illustrated in Figure 3.3. This allows to verify the functional behavior of large components or even of complete an NMS in three steps as illustrated in Figure 5.1.

In the following, we illustrate the three steps of MECHATRONICUML's compositional verification approach based on the RailCab system using the RTCP EnterSection. In the RailCab system, RailCabs travel on a track system that is subdivided into different types of sections including switches and railroad crossings. Before entering a section, a RailCab needs to query the section whether it is allowed to enter it using EnterSection. This is necessary for

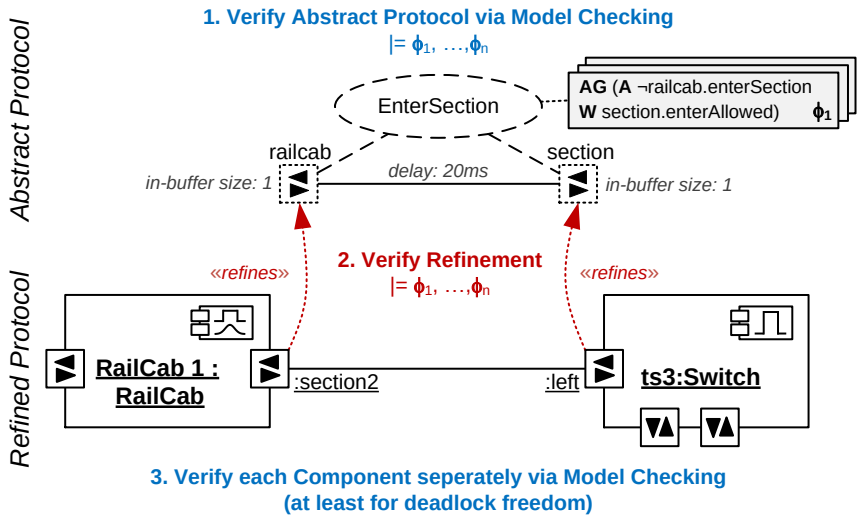


Figure 5.1: Overview of the Refinement Approach [HBDS15]

realizing collision avoidance because sensors in a RailCab may not detect other RailCabs if they are hidden behind a bend or some other obstacle. Therefore, RailCabs communicate with a section for getting permission to enter it.

In the first step of the compositional verification approach, the developer needs to verify all RTCPs that he used in the system for safety and liveness properties. The verification may either be carried out using model checking based on UPPAAL as described by Gerking [Ger13] or using a graph-based verification technique [EHH+13, SHS11]. In our RailCab example, this step includes verification of the RTCP EnterSection beside others. For the remainder of this chapter, we refer to the behavior implemented by the roles of the RTCP as the *abstract protocol*.

In the second step, we verify whether the ports of a component correctly refine the roles of the RTCP. This is necessary because the ports usually need to extend the role behavior by additional computations. In our example, a port of a track section may not decide on its own whether the track section is free. If several ports of a track section communicate with different RailCabs, the ports need to be synchronized such that only one port allows a RailCab to enter at a time. Despite the necessary modifications, the behavior of a port must be compliant to the specified role behavior, i.e., it must be a legal refinement according to a refinement definition. In the following, we refer to the behavior implemented by the discrete ports as the *refined protocol*.

In the third step, we combine the port RTSCs to a component RTSC using additional synchronization RTSCs (cf. Section 3.1.2.1). Then, we need to verify for each component RTSC that it is free of deadlocks [Gie03]. We may verify additional safety and liveness properties referring to a correct interaction of the different ports of a component if necessary. In our RailCab example, we may verify the aforementioned property that the track section gives permission to only one RailCab to enter at a time. Approaches for resolving such dependencies automatically have beeen introduced by Eckardt and Henkler [EH10] and Goschin et al. [Gos14, DGB14].

Verifying the correctness of the refinement in the second step of the compositional verification approach requires a formal refinement definition. It guarantees that all properties that have been verified for the RTCP also hold for the interaction of components via their discrete ports. A suitable refinement definition leaves the developer with as much flexibility on refining the model as possible, but is as restrictive as necessary for guaranteeing that no verified property is violated. This enables to use the same RTCP for different components. In the RailCab example, it is particularly useful to use the RTCP EnterSection for all types of track sections. Then, the type of track section is opaque for a RailCab. Each kind of track section, however, requires the behavior of a section to be refined differently.

In literature, different refinement definitions have been proposed [WL97, JLS00, HH11a]. Each of which provides a different compromise between preserved properties and allowed modifications. Depending on the particular type of RTCP that is refined, all of them might be useful when building a system. As a consequence, there does not exist one refinement definition that is suitable of all RTCPs. Instead, a compositional verification approach should support several refinement definitions. Presently, the compositional verification approach supports only one particular kind of timed simulation [Gie03], which is not sufficient to handle the example sketched above.

In this chapter, we extend the compositional verification approach of MECHATRONICUML by supporting a total of six different refinement definitions. As our main contribution, we present a refinement check that enables to verify all six refinement definitions for a given role of a RTCP and a discrete port of a component. Our refinement check extends the approach by Jensen et al. [JLS00] that is based on so-called test automata. A test automaton encodes both the behavior of the role and the conditions for a correct refinement. If (and only if) the port behavior violates the conditions for a correct refinement, the test automaton enters a special error location indicating a negative verification result. We parameterized and extended the original construction such that we may verify all refinement definitions in a single algorithm that may easily be extended to include additional refinement definitions if necessary. As a byproduct, our refinement check may automatically detect which refinement definition is suitable for a given pair of role and port behavior including the verified properties. Although the compositional verification approach of MECHATRONICUML is primarily intended for verifying the software that is used in the reflective operator of the OCM [GS13], our refinement check may also be used for checking refinements of RTCPs that are used in the cognitive operator.

In the remainder of this chapter, we first describe the behavior of the RTCP EnterSection including refined port RTSCs for the different types of track sections in Section 5.1. Section 5.2 reviews the six refinement definitions that we consider in our approach. Then, we introduce our refinement check based on test automata in Section 5.3 and its implementation in Section 5.4. Thereafter, we discuss the assumptions and limitations of our approach in Section 5.5. We evaluate our refinement check using a case study based on the RTCP EnterSection (Section 5.6). Finally, we discuss related work (Section 5.7) and summarize the results (Section 5.8).

The test automaton construction presented in Section 5.3 has been developed as part of a Master's Thesis [Bre10]. The contents of this chapter have been published in [BHSH13] and [HBDS15].

5.1 Refining Real-Time Coordination Protocols to Port Implementations

In the following, we describe the RTCP EnterSection including the RTSCs of both roles in Section 5.1.1. Then, we show in Section 5.1.2 how the role section needs to be refined for different types of track sections.

5.1.1 Real-Time Coordination Protocol EnterSection

The RTCP EnterSection has two roles named railcab and section as shown in Figure 5.1. The role railcab is to be implemented by RailCabs while the role section is to be implemented by all types of track sections. Both roles have a buffer size of 1 and a message has a propagation delay of 20 ms.

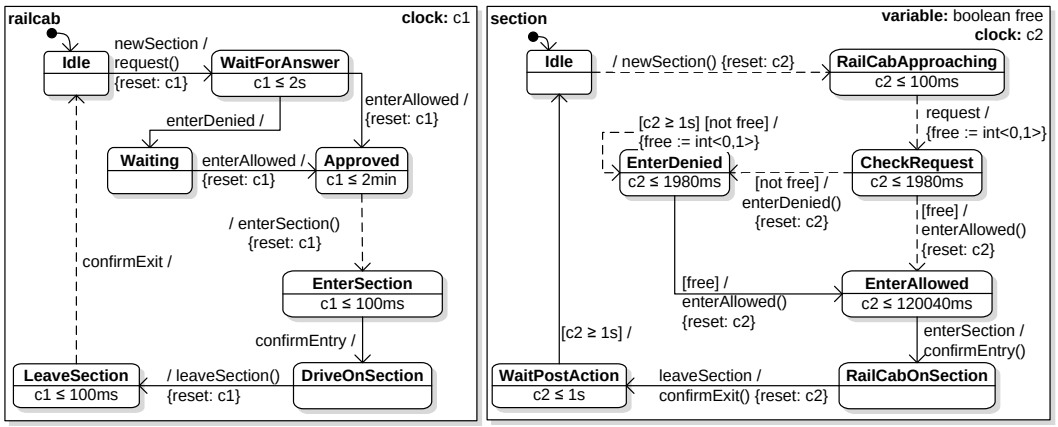


Figure 5.2: RTSCs of Role railcab and Role section of RTCP EnterSection [HBDS15]

Figure 5.2 shows the RTSCs of the two roles railcab and section. In our behavior specification, we assume that a track section senses upcoming RailCabs that are about to enter and notifies these RailCabs using a message newSection. Then, the informal behavior definition is as follows: Initially, both roles are in state Idle. As soon as role section recognizes the RailCab, it sends the message newSection to role railcab. railcab needs to answer with request within 100 ms. Then, role section switches to state CheckRequest. In this state, the section checks within 1980 ms if the RailCab may enter. This check is not part of the protocol but of the concrete component because each type of section must execute different checks. However, the result is stored in variable free and may be true or false. If the section is free, then the role section sends the message enterAllowed and switches to state EnterAllowed, else it sends the message enterDenied and switches to state EnterDenied. The RailCab expects one of these messages within 2 s. If the track section was not free, section will check repeatedly whether the track section becomes free. As soon as this is the case, section sends enterAllowed and switches to the eponymous state. For simplicity reasons, we assume that entering will eventually be allowed within 1980 ms. After receiving enterAllowed, the RailCab switches to Approved and starts entering the track section. Upon entering the track section, railcab sends enterSection. This needs to happen within 2 min. section will receive this message at most

120,040 ms (= 2 min 40 ms) after it allowed the railcab to enter and will answer with message confirmEntry. As soon as the RailCab leaves the section, it will send message leaveSection. Role section will confirm this with the message confirmExit. After one second, the interaction for this drive through is finished. Then, railcab may start a new interaction with the next track section.

The RTCP EnterSection is safety-critical. If a RailCab is allowed to enter a track section although the track section is occupied, a crash will happen. Therefore, we verify the RTCP for safety and liveness properties in Step 1 of the compositional verification approach. In particular, we verify three properties ϕ_1 to ϕ_3 that are needed to be preserved by the ports that refine the roles of this RTCP (cf. [HBDS15]).

The first property is: "The message enterSection will not be sent by role railcab until section sends enterAllowed." We may formalize this property ϕ_1 using TCTL as:

$\phi_1 =$ **AG**(**A** `not railcab.enterSectionMsg` **W** `section.enterAllowed`)

The second property is: "A RailCab may eventually enter a track section". We may formalize this property ϕ_2 using TCTL as:

$\phi_2 =$ **EF** `(section.RailCabOnSection)`

Finally, ϕ_3 ensures that the RTCP does not have a deadlock.

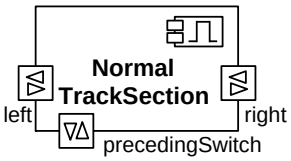
5.1.2 Refined Port Real-Time Statecharts

The roles of EnterSection now need to be refined by discrete ports of the components. "Typical refinement steps include adding data exchange between different ports of a component, adding component specific functions, and accessing shared variables inside the component. That, in turn, may require to add additional states and transitions to the RTSC of the role." [HBDS15] This refinement results in the *port RTSC*.

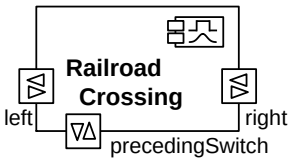
In our RailCab example, the RailCab refines the role section at its ports section1 and section2 of the embedded component DriveControl as shown in Figure 3.6. These two ports are mandatory in DriveControl. We assume that they are always reconnected by the surrounding RailCab component such that one port instance is always connected to the current track section while the other is connected to the next track section. If the RailCab has left and track section and returned to the Idle state, the port is reconnected to the next track section.

We consider three different types of track sections. These are normal track sections, railroad crossings, and switches. The corresponding components are shown in Figure 5.3. By using EnterSection for all types of track sections, we enable RailCabs to register at any type of track section without needing to distinguish them. However, each of the three types of track sections requires the behavior to be refined differently as we illustrate in the following.

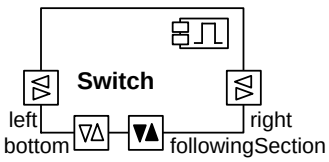
"We start with the normal track section. A normal track section is a track section that has only tracks but no switches, stations, or any kind of crossing. A normal track section may need to communicate with more than one RailCab at a time, e.g., from different directions. Then, the decision whether the track section is free does not only depend on the communication of a single port with a RailCab, but it depends on the communication of several ports with RailCabs. This is because if one port permits a RailCab to enter the track section, all other ports have to deny. Thus, the decision whether the track section is free can only be made within the component but not within the RTCP. As a consequence, we need to refine the track section behavior as shown in Figure 5.4. We highlighted the parts that were changed or added with respect to the RTSC in the right part of Figure 5.2 in blue color. In



(a) Component for Normal Track Sections



(b) Component for Railroad Crossings



(c) Component for Switches

Figure 5.3: Components for the Different Types of Track Sections [HBDS15]

particular, the RTSC for ports of the normal track section reads the variable sectionFree at the transition from RailCabApproaching to CheckRequest which is declared inside the component RTSC. If the track section is free, then the RTSC needs to synchronize via the synchronization channel acquire with the internal RTSC of the component for reserving the track section. As part of this synchronization, the internal RTSC changes the value of sectionFree to false (cf. Figure A.39 in Appendix A.5.2.1).

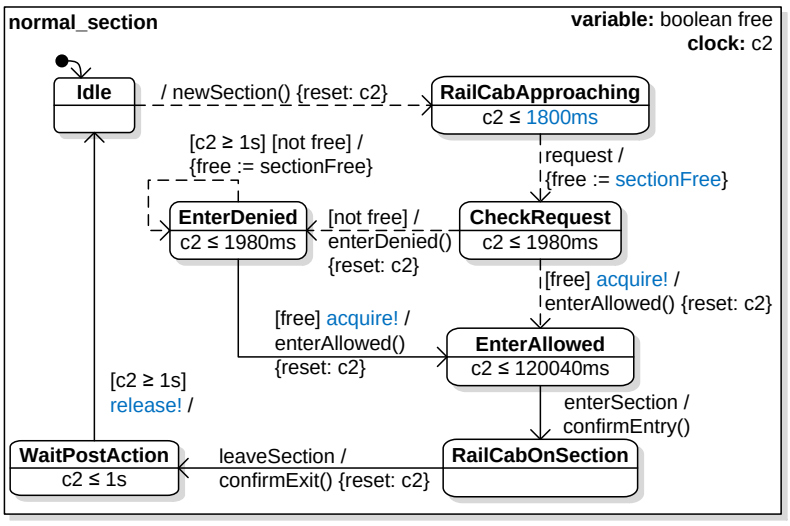


Figure 5.4: Refined Protocol Behavior for Normal Sections [HBDS15]

Having consumed the message newSection, the RTSC of the role section gives a limit of 1980 ms for deciding whether the section is free. The normal track section, however, only needs to read the variable sectionFree that is defined in the component RTSC for this decision. This only takes a small amount of time. Therefore, we relax the invariant of the state RailCabApproaching from 100 ms to 1800 ms to increase the probability that a RailCab currently

driving on the section has already left. As a result, the normal track section will store the request in the in-buffer beyond the point in time that was allowed by the abstract protocol.

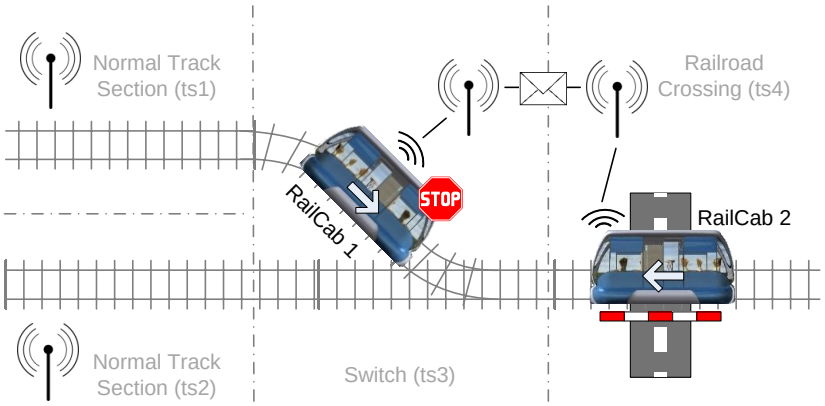


Figure 5.5: Deadlock Resulting from RailCab Stopping on a Switch [HBDS15]

The second type of track section is the switch. In addition to the requirements of a normal track section, switches have the requirement that a RailCab must not stop *on* a switch. In particular, a RailCab must stop *before* entering the switch if the subsequent section is not free. Consider the situation shown in Figure 5.5. RailCab 1 entered the switch ts3 driving to the right. It needs to stop because the subsequent section ts4 is occupied by RailCab 2 driving to the left. Since RailCab 1 blocks the switch, RailCab 2 cannot pass. If RailCab 1 waited on ts1, i.e. *before* the switch, RailCab 2 would have been able to pass. For preventing such situations, we only allow RailCabs to enter a switch if the subsequent track section is free as well. As a consequence, the switch needs to communicate with the subsequent track section if it receives a request from a RailCab.

Figure 5.6 shows the resulting RTSC for a switch. The switch also uses an additional state, which is called WaitForTrack. At the transition from RailCabApproaching to WaitForTrack, the RTSC synchronizes with the port followingSection of the switch (cf. Figure 5.3c) via the synchronization channel nextSectionFree. Then, the port followingSection communicates with the subsequent track section for checking whether that track section is free. If so, the port synchronizes via sectionFree, otherwise it synchronizes via sectionOccupied. Only if the switch itself and the subsequent track section are free, then the RTSC may switch to EnterAllowed and give permission to enter the switch to the RailCab.

Finally, we consider railroad crossings where cars and pedestrians cross the tracks. In addition to the requirements of a normal track section, we must close the gates before allowing a RailCab to enter. The gates needs to remain closed as long as the RailCab drives on the railroad crossing and need to be opened after the RailCab left." [HBDS15]

Figure 5.7 shows the resulting RTSC for a railroad crossing. We added an additional state ClosingGate where the railroad crossing waits for the gates to close. At the transition from CheckRequest to ClosingGate, the RTSC synchronizes with the internal behavior of the railroad crossing using the synchronization channel closeGate for closing the gate. After the gates have been closed, the internal behavior synchronizes via gateClosed and the RTSC switches to EnterAllowed. Although the RTSC looks fine at a first glance, it does not correctly refine the abstract protocol due to a timing error in state CheckRequest. We deliberately added this error

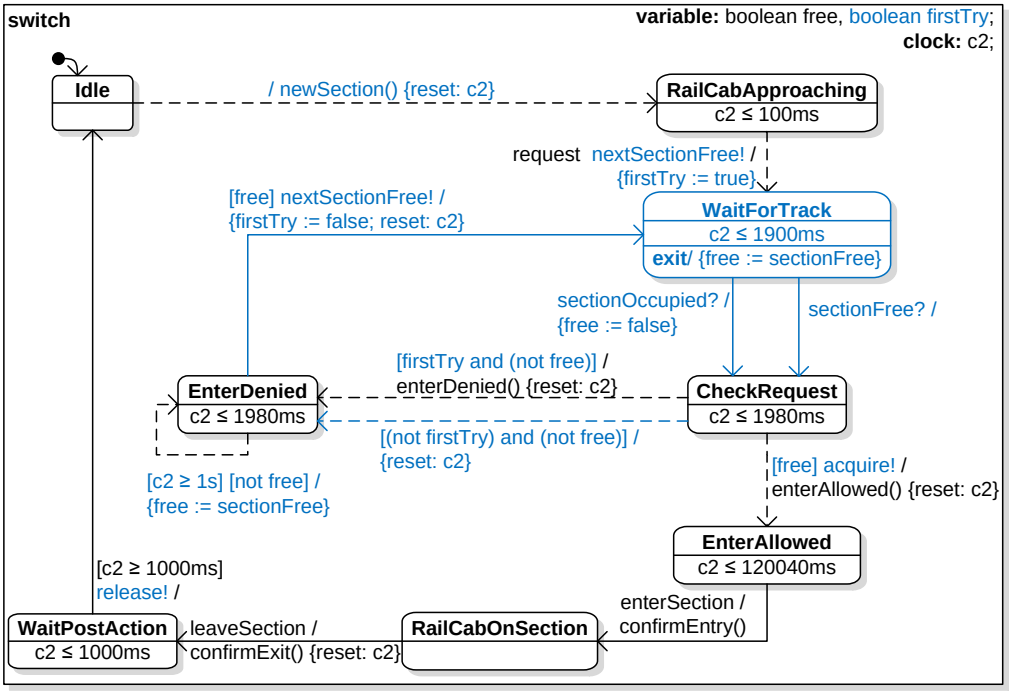


Figure 5.6: Refined Protocol Behavior for Switches [HBDS15]

for illustrating in our case study (cf. Section 5.6) how we detect such incorrect refinements using our refinement check. We present a correctly refined RTSC in Section 5.6.4.

"Although each of the three RTSCs introduced above refines the same abstract protocol behavior, the resulting refined RTSCs look quite different. Even though the RTSCs are only of medium size, it is already very hard to decide manually whether they have been refined correctly." [HBDS15] Using our refinement check, we may automatically decide whether they are correctly refined.

5.2 Considered Refinement Definitions

"A *refinement definition* relates an abstract model and a refined model of the same system. In our approach, the abstract model is given by the abstract protocol while the refined model is given by the refined protocol as shown in Figure 5.1. The refinement definition defines how the behavior defined by the refined protocol may deviate from the behavior defined by the abstract protocol such that verified properties still hold. That means, if the refinement definition is fulfilled, we can avoid any explicit verification of the refined protocol.

In a little more detail, a restrictive refinement definition guarantees that verified safety and liveness properties, like properties ϕ_1, ϕ_2, and ϕ_3 in Section 5.1.1, still hold for the refined protocol. This is crucial for our compositional verification approach. A less restrictive refinement definition leaves developers with more flexibility to adapt the abstract protocol to a component and, thus, allows for more possible refined protocols. Finding a suitable refine-

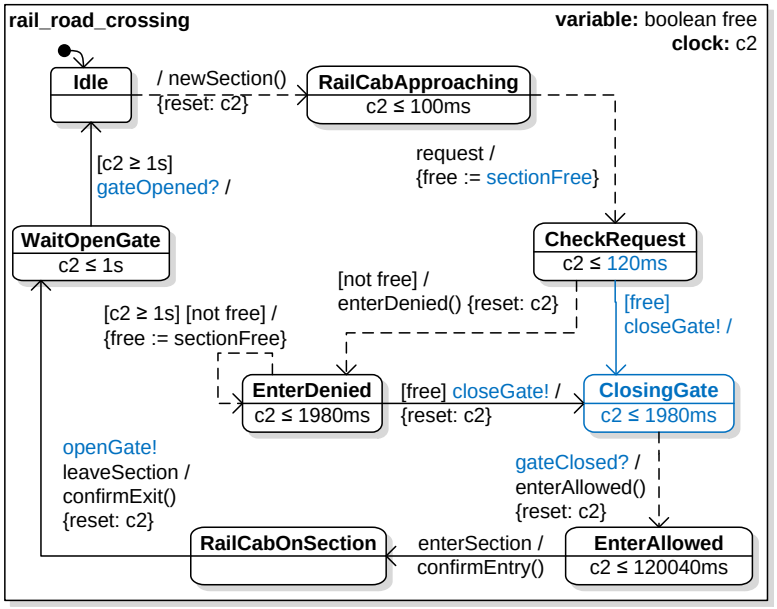


Figure 5.7: Incorrectly Refined Protocol Behavior for Railroad Crossings due to a Timing Error in State CheckRequest [HBDS15]

ment definition is, thus, a trade-off between flexibility upon building the refined protocol and properties that are preserved by the refined protocol." [HBDS15]

"In the following, we briefly explain the six most relevant refinement definitions for networked mechatronic systems. Four of these, simulation [BK08], bisimulation [BK08], timed simulation [WL97], and timed bisimulation [WL97], are especially well-known definitions. Each of them has been shown to preserve a particular class of verified properties. We additionally consider the less well-known timed ready simulation [JLS00] and relaxed timed bisimulation [HH11a, Hen12] because they are particularly useful for refining MECHATRONICUML models." [HBDS15] We restrict ourselves to informal descriptions of the refinement definitions because the formal definitions are not required for understanding the presented concepts. We refer the interested reader to the literature given above for formal definitions of all refinements.

"All of the considered refinement definitions only allow the refined protocol to include sequences of sent and received messages that are already specified in the abstract protocol. None of them allows the refined protocol to add additional sequences of messages. As a minor extension to the existing definitions, we require that the refined protocol correctly refines non-deterministic choices contained in the abstract protocol. That means after a choice, the refined protocol needs to conform to the abstract protocol that made the same choice." [HBDS15]

For being able to add behavior to the role RTSCs as described in Section 5.1.2, we consider only so-called *weak* variants of the refinement definitions [WL97]. These abstract from internal behavior that is defined by transitions not carrying a message but performing an internal computation or synchronization. Since these operations are not visible to a communication partner, they are not relevant for the message exchange defined by the protocol. In particular,

the refinement only needs to ensure that the next message after such internal computation is sent in the right time interval.

"The upper part of Figure 5.8 shows a timing diagram for an excerpt of the role behavior defined by the RTSC section in Figure 5.2. The timing diagram shows when (in which time interval) the messages newSection, request, enterAllowed, and enterDenied can be sent or received. Here, section may send the message newSection at an arbitrary point in time. After sending newSection, the clock c2 is reset to 0 and request must be received within 100 ms. Then, either enterAllowed or enterDenied must be sent until c2 reaches 1980 ms. The lower part of Figure 5.8 gives six examples for port RTSCs. Each refines the role section in a different way, resulting in different intervals for sending or receiving the aforementioned messages. Each of the examples showcases a different combination of changes to the intervals, e.g, the extension of intervals for received messages but not for sent messages. As a consequence, each example fulfills a different set of refinements as indicated in the corresponding row in the table on the right. In the following, we introduce all six refinement definitions from left to right with respect to the table in Figure 5.8. Along with the description of each refinement definition, we will refer to Figure 5.8 to illustrate the differences.

All of the refinement definitions mentioned above rely on the assumptions stated in Section 2.4.3. If a system does not fulfill these assumptions, the refinement definitions presented in this section cannot guarantee that the verified properties still hold for the refined protocol." [HBDS15]

Simulation

Simulation [CGP00, BK08] is the least restrictive refinement definition that we consider. It requires that the refined protocol only includes sequences of messages that are already specified by the abstract protocol. The refined protocol may remove sequences of messages and define a different timing of messages. As a result, simulation preserves all LTL formulas and all CTL formulas that only contain $\forall$-path quantifiers. Formulas with an $\exists$-path quantifier are not preserved because the paths fulfilling the property might be removed.

In Example 1 in Figure 5.8, the message enterDenied is removed, while enterAllowed can still be sent later. The interval for receiving the message request, on the other hand, is shortened. These changes are permitted by simulation but not by any other considered refinement definition.

Bisimulation

Bisimulation [CGP00, BK08] requires that the refined protocol contains the same sequences of messages as the abstract protocol but allowing for a different timing. As a result, bisimulation preserves all LTL and CTL formulas.

Example 2 in Figure 5.8 fulfills the conditions of bisimulation because it uses the same sequences of messages with a different timing. Example 1 does not fulfill the conditions of bisimulation because a message has been removed.

Timed Simulation

Timed simulation [WL97] imposes the same conditions as simulation but additionally imposes timing constraints. In particular, the refined protocol may only send and receive messages in the same or a restricted time interval compared to the abstract protocol. As a conse-

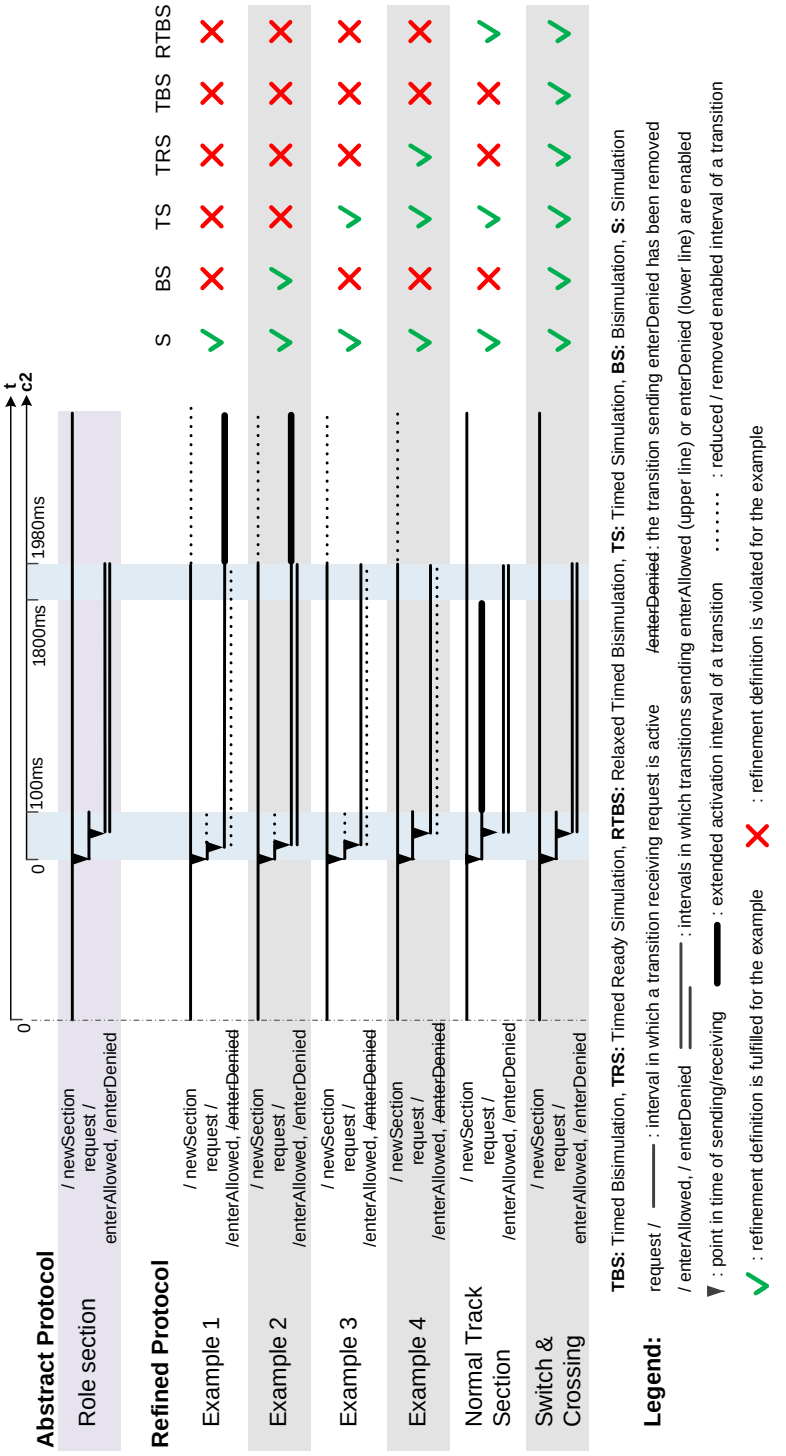


Figure 5.8: Example for Illustrating the Differences Between the Considered Refinement Definitions [HBDS15]

quence, timed simulation preserves all TCTL formulas that only contain $\forall$-path quantifiers. We refer to this as ATCTL.

In Figure 5.8, the refinement in Examples 3 and 4 fulfills the conditions of timed simulation because time intervals are only reduced but never extended. Examples 1 and 2 extend the time interval for sending enterAllowed and, therefore, do not fulfill the conditions of timed simulation.

Timed Ready Simulation

Timed ready simulation [JLS00] imposes the same conditions as timed simulation but additionally requires that the refined protocol preserves all urgent transitions including their timing. Jensen et al. [JLS00] proved that this is necessary for a compositional verification approach if the behavior contains urgent transitions. As a result, timed ready simulation preserves all ATCTL properties and ensures that the refined protocol has the same urgent behavior as the abstract protocol.

In Figure 5.8, Example 4 fulfills the timed ready simulation because the interval of the urgent transition for receiving the message request is not changed. In addition, the behavior of the switch and the crossing fulfills the timed ready simulation because it sends and receives all messages in the same time intervals as the role section.

Timed Bisimulation

Timed bisimulation [WL97] imposes the same conditions as a bisimulation but additionally requires that the refined protocol sends and receives messages in exactly the same time intervals as the abstract protocol. Therefore, it is the strictest refinement definition that we consider. Still, the timed bisimulation allows to modify the abstract protocol during the refinement step by inserting internal computations between the sent and received messages if they do not affect the timing of messages. Timed bisimulation preserves all TCTL properties.

In Figure 5.8, only the behavior of switch and crossing fulfills the conditions of timed bisimulation because it neither removes messages nor changes time intervals of messages as it is the case in all other examples.

Relaxed Timed Bisimulation

The *relaxed timed bisimulation* [HH11a, Hen12] relaxes the strict conditions of timed bisimulation. In particular, it enables to extend the time intervals for receiving message, but it still requires that the upper bounds for sent messages remain unchanged. This refinement is particularly useful for networked mechatronic systems. If two mechatronic systems coordinate on a specific task, it often does not matter when messages are received but only that the answer is on time. In our example, it is irrelevant for a RailCab at what point in time the track section processes its request. For the RailCab, it only matters that it receives the answer in time. Due to the relaxation on received messages, relaxed timed bisimulation preserves all LTL and CTL-formulas as well as all TCTL formulas only referring to the latest sending of messages.

Relaxed timed bisimulation imposes two important conditions on the RTCP being refined. First, the refined protocol needs a larger in-buffer than the abstract protocol in order to avoid buffer overflows because message are taken out of the in-buffer later. Second, the RTCP

must be bidirectional. Otherwise, the receiving protocol will not be restricted in its timing behavior at all.

In Figure 5.8, the behaviors of normal track section, switch, and crossing fulfill the conditions of relaxed timed bisimulation. The behavior of normal track section (cf. Figure 5.4) still receives request after 1800 ms which is covered by relaxed timed bisimulation but violates timed bisimulation.

5.3 Test automata-based Refinement Checking

This section introduces our approach [Bre10] for verifying correct refinements of roles to ports. Our approach is based on *test automata*. "Test automata have been introduced by Jensen et al. [JLS00] as an approach for verifying refinements for UPPAAL timed automata. The basic idea of their approach is to encode an abstract automaton A and the conditions for a correct refinement as an automaton T_A, called test automaton. Test constructs in T_A encode which changes are allowed and which are not, according to the conditions of the particular refinement definition (cf. Section 5.2). The test automaton T_A is then used to verify whether a refined automaton R is a correct refinement of A according to the given refinement definition. If and only if the conditions of the refinement definition are *not* fulfilled by R, a special state Error in T_A becomes reachable via the test constructs. Jensen et al. use a reachability analysis for deciding whether Error is reachable or not. Figure 5.9 gives an overview of our process for refinement checking based on test automata in MECHATRONICUML." [HBDS15]

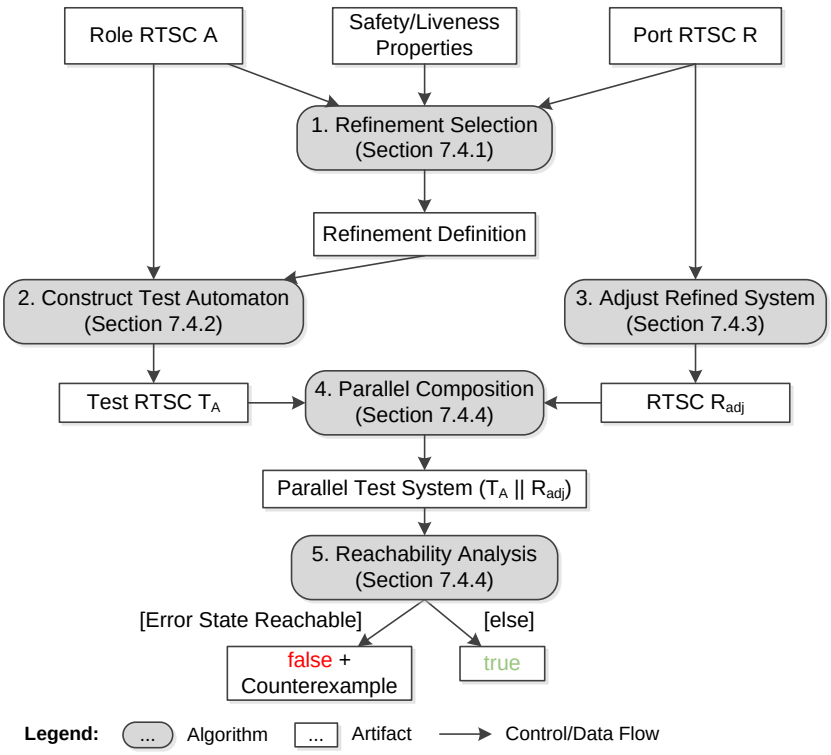


Figure 5.9: Refinement Check using Test Automata [HBDS15]

The inputs for our refinement check are the RTSC of the role serving as the abstract automaton A, the RTSC of the port serving as the refined automaton R, and the safety and liveness properties that have been verified for the abstract protocol. The refinement check is then carried out in five steps. In the first step, we automatically select the most suitable refinement definition based on the role RTSC and the safety and liveness properties (cf. Section 5.3.1). In the second step, we construct the test automaton T_A (cf. Section 5.3.2). We extend the construction by Jensen et al. [JLS00] such that it enables checking all six refinement definitions introduced in Section 5.2. In particular, we introduce new test constructs for checking relaxed timed bisimulation and bisimulation. Our new construction is parameterized such that a test automaton can be built based on the selected refinement definition. Since T_A is a RTSC in our approach, we call it the *Test RTSC*. In the third step, we adjust the port RTSC R to R_{adj} such that it may be tested by T_A (cf. Section 5.3.3). In particular, T_A needs to communicate via synchronizations with R, i.e., without the additional delay of asynchronous communication, for testing the intervals in which R sends or receives messages. Thereafter, we build the parallel test system $T_A \parallel R_{adj}$. The construction of the parallel test system is based on NTAs. We refer to Appendix B for a formal definition of the construction of an NTA for two RTSCs. Finally, we perform a reachability analysis on the parallel test system (cf. Section 5.3.4). If the special error state is reachable in $T_A \parallel R_{adj}$, the port RTSC is not refined correctly and our algorithm returns a counterexample. If the error state is not reachable, the refinement is fulfilled.

5.3.1 Refinement Selection

We summarized the characteristics of the six refinement definitions introduced in Section 5.2 in the decision tree shown in Figure 5.10. Using this decision tree, we may automatically derive the most suitable refinement definition based on the role RTSC, the port RTSC, and the verified properties. The most suitable refinement definition is the one that is least restrictive for the given kind of models while still preserving all properties that were verified for the abstract protocol. A less restrictive refinement definition allows more modifications in the refined protocol and, therefore, provides more flexibility to the developer.

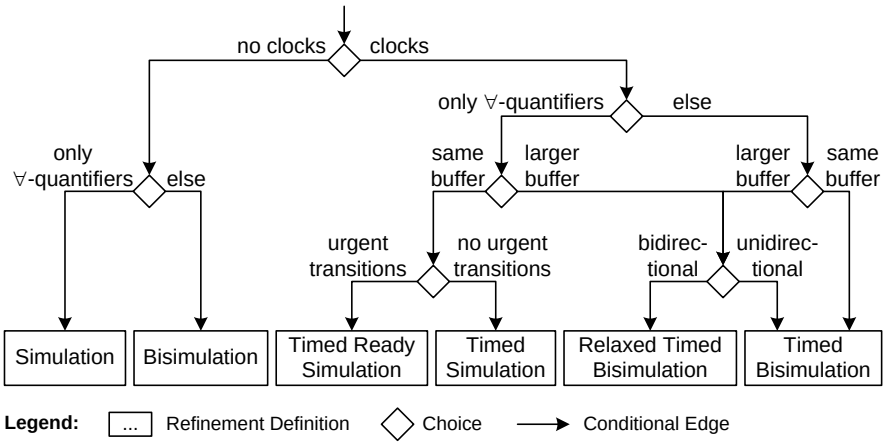


Figure 5.10: Decision Tree for Selecting a Refinement Definition [HBDS15]

We can extract the necessary information for deriving a decision based on the decision tree by a syntactical analysis of the inputs. For the first decision in the tree, we need to analyze whether the RTSCs use clocks or not. Untimed refinements are not suited for models that define time-dependent behavior. Second, we check whether the properties only contain $\forall$-path quantifiers as, e.g., Property ϕ_1 in Section 5.1.1, or whether they also contain $\exists$-path quantifiers as, e.g., Property ϕ_2. If any property uses an $\exists$-path quantifier, we need a variant of bisimulation to preserve this formula. Third, for timed refinements we need to take into account whether the port RTSC uses a larger in-buffer than the role RTSC. While our Relaxed Timed Bisimulation is less restrictive than the alternatives, it can only be used in cases where a larger buffer is available. To decide whether to select Timed Ready Simulation or Timed Simulation we also need to analyze if the role RTSC uses urgent transitions. Finally, Relaxed Timed Bisimulation is only applicable for bidirectional protocols. If the protocol is unidirectional, we need to apply the more restrictive Timed Bisimulation.

In our example in Section 5.1, all three refined RTSCs use clocks. Therefore, we take the right branch of the decision tree in all three cases. As Property φ_2 contains an $\exists$-path quantifier, we take the else-branch on the next decision. Therefore, the only suitable refinements remaining are relaxed timed bisimulation and timed bisimulation.

The refined RTSC of the normal track section (cf. Figure 5.4) uses a larger in-buffer than the role section. As a result, the decision tree selects relaxed timed bisimulation for checking this refined RTSC. The refined RTSCs of switch (cf. Figure 5.6) and railroad crossing (cf. Figure 5.7) do not use a larger buffer. Therefore, the decision tree selects timed bisimulation for these refined RTSCs.

5.3.2 Construction of the Test Automaton

"Figure 5.11 presents the schema for the construction of a part of T_A. In particular, it defines how one single transition $S \longrightarrow S'$ of A is translated to T_A. Thus, the construction schema needs to be applied to each transition of A. Figure 5.12 shows an excerpt of a test RTSC SectionTA_TBS that has been constructed by applying the construction schema to each transition of the role section (cf. Figure 5.2) for checking a timed bisimulation. In the following, we refer to this example to illustrate the constructs created as part of T_A.

T_A contains test constructs for checking the different conditions of the refinement definitions. In particular, the test constructs check for allowed communication (Case 1, cf. Section 5.3.2.1), forbidden communication (Cases 2a and 2b, cf. Section 5.3.2.2), and required communication (Cases 3a, 3b, and 3c, cf. Section 5.3.2.3) in R. Which of these test constructs are used in T_A depends on the refinement definition to be checked as shown in Table 5.1. In addition, Table 5.1 summarizes the definition of the function $widen$ used at the transitions $S_{TA} \longrightarrow S_{TA}{}'$ in T_A. We explain this function along with the labels of the transitions in the following." [HBDS15]

5.3.2.1 Test Constructs for Allowed Communication (Case 1)

Case 1 includes *allowed communication* in T_A, i.e., sequences of messages that are defined by A. We include allowed communication in T_A because all refinement definitions allow these sequences of messages to be included in R. The white states S_{TA} and S'_{TA} in the schema correspond to the states S and S', respectively, of A. The transitions between white

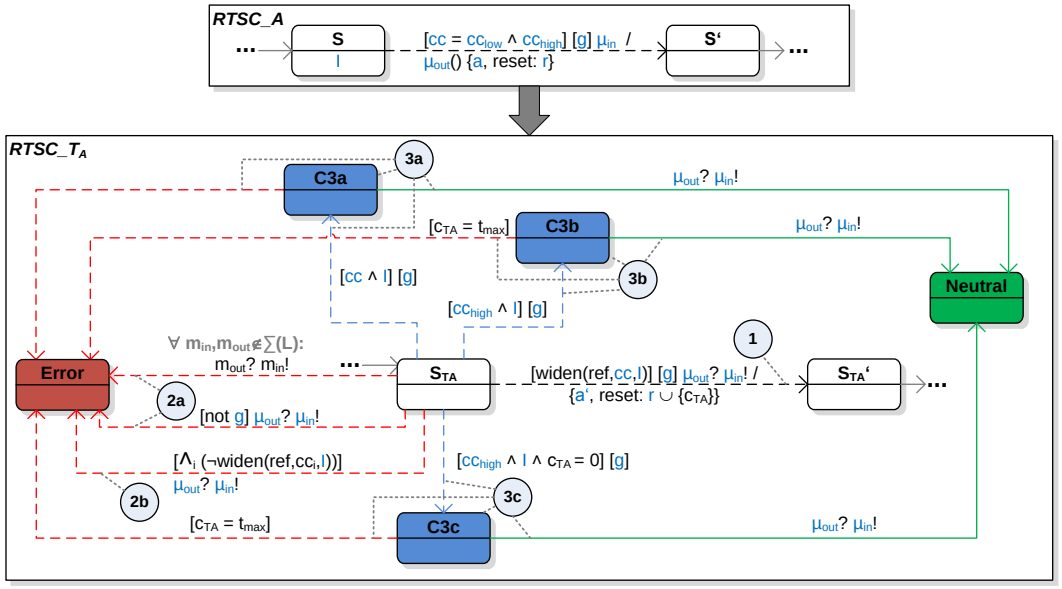


Figure 5.11: Construction Schema for our Test Automata [HBDS15]

Table 5.1: Required Cases and Definition of $widen$ Function for Each Refinement Definition [HBDS15]

Refinement Definition	**Required Cases**	**Definition of widen**
Simulation	1, 2a	$true$
Bisimulation	1, 2a, 3c	$true$
Timed Simulation	1, 2a, 2b	$cc \wedge I$
Timed Ready Simulation	1, 2a, 2b, 3a (urgent)	$cc \wedge I$
Relaxed Timed Bisimulation	1, 2a, 2b, 3b (sending), 3c (receiving)	$cc_{high} \wedge I$ (sending), $true$ (receiving)
Timed Bisimulation	1, 2a, 2b, 3a	$cc \wedge I$

states correspond to the role behavior. For each transition $S \longrightarrow S'$ in A, we add one corresponding transition $S_{TA} \longrightarrow S_{TA}{}'$ to T_A. In the example of Figure 5.12, the white states and the transitions between them have been created to handle allowed communication.

Since T_A needs to communicate synchronously with R_{adj}, we map asynchronous messages to synchronizations. If $S \longrightarrow S'$ sends (receives) a message μ_{out} (μ_{in}), then the corresponding transition $S_{TA} \longrightarrow S_{TA}{}'$ specifies a synchronization $\mu_{in}?$ ($\mu_{out}!$). That means, T_A produces the inputs for R_{adj} and receives its outputs. All transitions in T_A that carrying a synchronization are non-urgent even if the corresponding transition is urgent in A. This is necessary for entering the test constructs checking for forbidden and required communication because urgent transitions have precedence over non-urgent transitions. Then, $T_A \parallel R_{adj}$ would urgently execute the allowed behavior without being able to enter the test constructs. All other transitions T_A have the same urgency as their corresponding transitions in A.

$S_{TA} \longrightarrow S_{TA}{}'$ specifies the same guard g as the corresponding transition in A such that it may only be enabled if $S \longrightarrow S'$ is enabled. In addition, we add all clock resets of $S \longrightarrow S'$

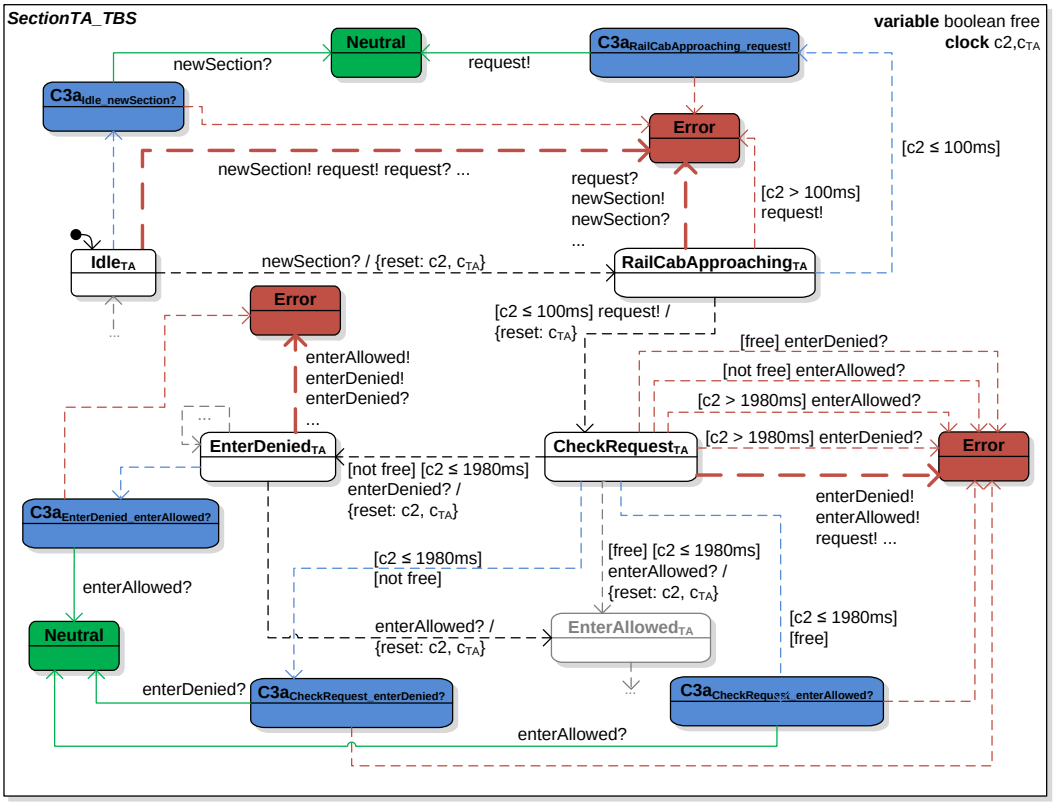


Figure 5.12: Example Test RTSC (Excerpt) for Checking the Timed Bisimulation for section [HBDS15]

to $S_{TA} \longrightarrow S_{TA}'$. The latter includes one additional clock reset for the clock c_{TA}. This clock is used by T_A for checking required communication as we explain in Section 5.3.2.3.

In Figure 5.12, the transition from Idle$_{TA}$ to RailCabApproaching$_{TA}$ defines the synchronization newSection?. It has been derived from the transition Idle to RailCabApproaching of section (cf. Figure 5.2) that sends a message newSection. In addition, the transition from Idle$_{TA}$ to RailCabApproaching$_{TA}$ specifies the clock reset for c2 and additionally resets c_{TA}.

The time guard of $S_{TA} \longrightarrow S_{TA}'$ depends on the refinement definition to be checked, the time guard cc of $S \longrightarrow S'$, and the invariant of state S in A. Based on these inputs, the function $widen$ assigns a time guard to $S_{TA} \longrightarrow S_{TA}'$. Table 5.1 summarizes the outputs of $widen$ for the different refinement definitions. In any case, the state S_{TA} does not have an invariant. This is necessary to enable that T_A may check whether R sends or receives messages in different time intervals compared to A. If S_{TA} carried an invariant, T_A would be forced to fire a transition and would not be able to identify whether R may exceed the time interval specified by A due to the semantics RTSCs (cf. Appendix B).

Simulation and bisimulation do not check timing conditions and, therefore, $widen$ assigns $true$ as a time guard to $S_{TA} \longrightarrow S_{TA}'$. The same holds for transitions receiving a message when checking for a relaxed timed bisimulation because relaxed timed bisimulation allows to delay such transitions arbitrarily.

For timed simulation, timed ready simulation, and timed bisimulation, the time guard returned by $widen$ is the conjunction of the original time guard cc and the invariant I of S. As a result, the transition $S_{TA} \longrightarrow S_{TA}{}'$ may only fire if the corresponding transition $S \longrightarrow S'$ is enabled. Since the example in Figure 5.12 has been constructed for a timed bisimulation, all time guards at transitions between white states use time guards of the form $cc \wedge I$.

Finally, the relaxed timed bisimulation uses the guard $cc_{high} \wedge I$, i.e., it conjuncts the upper bound of the original time guard and the invariant I of S. This is because relaxed timed bisimulation also allows transitions to send messages earlier compared to A but not later.

The translation of the transition action a of $S \longrightarrow S'$ depends on whether a contains a non-deterministic choice expression. If so, the non-deterministic choice expression is removed from a. If not, a is added unmodified to $S_{TA} \longrightarrow S_{TA}{}'$. All variables used with non-deterministic choice expressions are shared in $T_A \parallel R_{adj}$. The reason for the same is, R needs to refine A correctly for any choice. However, after making a choice, R should not need to correspond to an A that made a different choice. In our example, the RTSC for role section in Figure 5.2 and the RTSC for port normal_section in Figure 5.4, both non-deterministically assign the variable free. Then, A and R need to show the same behavior if the section is free (sending enterAllowed) and if the section is not free (sending enterDenied). As a consequence, only R_{adj} makes the non-deterministic choices and T_A follows these choices.

5.3.2.2 Test Constructs for Forbidden Communication (Case 2)

Case 2 checks for *forbidden communication*, i.e., messages that may not be sent or received by R if A is in state S. Therefore, we add the special state Error to T_A and create transitions from white states to Error. Here, we need to distinguish Cases 2a and 2b. Case 2a checks for illegal messages, i.e., messages sent or received by R in state S_{TA} that are not defined by any outgoing transition of S in A. These messages may also not appear in R because all refinement definitions forbid to add new sequences of messages to R. Case 2b checks for legal messages that are sent at illegal times, i.e., messages that may indeed be sent by R in state S_{TA}, but which do not comply to the timing restrictions imposed by the refinement definition. We explain both cases below in more detail. In Figure 5.12, we included several Error states to improve the readability of the figure.

Case 2a: Illegal Message

In Case 2a, we add two types of transitions to T_A. First, we add one transition $S_{TA} \longrightarrow Error$ for each message μ that *is not* specified at any outgoing transition of S in A. If R adds a forbidden message, these transitions make Error reachable. The resulting transitions only carry the synchronization corresponding to the message μ. Second, we add one transition for each message μ_{in} or μ_{out} that *is* specified at an outgoing transition of S in A. This transition checks whether R may send μ_{out} or receive μ_{in} if the guard g is not fulfilled. This is forbidden because R needs to behave in the same way as A under a given decision. The resulting transition only carries the synchronization corresponding to μ_{in} or μ_{out} and the negated transition guard $\neg g$. We add such transitions for any refinement definition that we consider in our approach (cf. Table 5.1).

In our example in Figure 5.12, the thick and dashed red transition from Idle_{TA} to Error has been constructed based on Case 2a. It represents a set of transitions to improve readability

of the figure. In particular, we obtain one transition that checks whether R may receive newSection in state Idle and additional transitions for any other message that checks whether this message may be sent or received by R. All of these messages are forbidden because R may only send newSection in state Idle. The two upper transitions from CheckRequest_{TA} to Error are also constructed based on Case 2a. The first one checks whether R may send enterDenied even though the section is free. The second one checks whether R may send enterAllowed even though the section is not free. Both are forbidden because R needs to behave in the same way as A if the section is free or not free.

Case 2b: Legal Message at Illegal Time

In Case 2b, we add one transition $S_{TA} \longrightarrow Error$ for each time interval in which μ may not be sent or received in R according to the given refinement definition. This case is only checked by refinement definitions that impose conditions on the timing of messages (cf. Table 5.1). The resulting transitions only carry a synchronization corresponding to the message μ and a time guard. The time guard encodes all time intervals where transitions $S_{TA} \longrightarrow {S_{TA}}'$ with synchronization μ may not fire.

We compute the time guard as follows. Each transition i from S_{TA} to ${S_{TA}}'$ in T_A has a time guard of the form $\bigwedge_j low_{ij} \leq c_j \leq up_{ij}$ for clocks c_j. We negate this clock constraint to obtain time intervals where the corresponding message μ may not be sent or received and yield a time guard of the form $\bigvee_j (c_j < low_{ij} \vee c_j > up_{ij})$. Then, we need to conjunct the resulting time guards for all transitions i to obtain time intervals where none of them may fire. Consequently, the transition from S_{TA} to Error has the time guard

$$\bigwedge_i \left(\bigvee_j (c_j < low_{ij} \vee c_j > up_{ij}) \right).$$

For a simple example, consider a state with two outgoing transitions that send a message a, one with the clock constraint $10 \leq c \leq 20$ and one with $50 \leq c \leq 60$. Then, the resulting clock constraint is

$$\begin{aligned} & \neg((c \geq 10 \wedge c \leq 20) \vee (c \geq 50 \wedge c \leq 60)) \\ = \; & (c < 10 \vee c > 20) \wedge (c < 50 \vee c > 60) \\ = \; & (c < 10 \wedge c < 50) \vee (c < 10 \wedge c > 60) \vee (c > 20 \wedge c < 50) \vee (c > 20 \wedge c > 60) \\ = \; & (c < 10) \vee (20 < c < 50) \vee (c > 60) \end{aligned}$$

Since clock constraints may only contain conjunctions, we create individual transitions for $c < 10$, $20 < c < 50$, and $c > 60$. If R_{adj} defines the synchronization $a!$ within one of these time intervals, then Error becomes reachable.

In Figure 5.12, the right transition from $\text{RailCabApproaching}_{TA}$ to Error has been created based on Case 2b. The transition from $\text{RailCabApproaching}_{TA}$ to CheckRequest_{TA} has the time guard $c2 \leq 100$ resulting from the invariant of state RailCabApproaching in section. Thus, the transition from $\text{RailCabApproaching}_{TA}$ to Error has the time guard $c2 > 100$ and specifies the same synchronization.

5.3.2.3 Test Constructs for Required Communication (Case 3)

Case 3 checks for *required communication*, i.e., sequences of messages that must be included in R according to the refinement definition. In particular, all variants of bisimulation and the timed ready simulation require that R contains particular sequences of messages that are contained in A. We add a neutral state Neutral to T_A that is reached when a required message is correctly sent or received by R. Case 3 is further subdivided into Cases 3a to 3c. Each case is associated with one particular kind of test state $CX(X \in [3a, 3b, 3c])$ for checking the corresponding conditions. T_A contains one such state for each required message μ that needs to be included in R. In addition, we add transitions $S_{TA} \longrightarrow CX$, $CX \longrightarrow Neutral$, and $CX \longrightarrow Error$ to T_A. If R violates the conditions imposed by the refinement definition, transitions $CX \longrightarrow Error$ are enabled and Error is made reachable. Otherwise, the transitions $CX \longrightarrow Neutral$ lead to the neutral state.

The Cases 3a to 3c check for different time intervals where μ needs to be sent or received by R. Which case is used depends on the particular refinement definition that is applied (cf. Table 5.1). We introduce the different cases in detail below.

Case 3a: Message in Same Time Interval

Case 3a with the associated state $C3a$ checks that R sends or receives μ in exactly the same time interval as A. This is needed for timed bisimulation and timed ready simulation. Timed bisimulation does not allow R to reduce the time intervals for messages that were defined in A. Timed ready simulation requires the same but only for messages defined for urgent transitions.

The transition from S_{TA} to $C3a$ has a guard and a time guard. Both are the same as for $S_{TA} \longrightarrow S_{TA}{}'$ considering the definition of $widen$ for timed bisimulation and timed ready simulation in Table 5.1. Consequently, T_A may enter $C3a$ whenever A may send or receive μ.

The transition from $C3a$ to Neutral is urgent, i.e., it has precedence over the non-urgent transition to Error. As long as R sends or receives μ, the transition to Error is never enabled and error is not reachable. If there exists a time interval in $cc \wedge I$ where R does not send or receive μ, then $CX \longrightarrow Neutral$ is not enabled and the transition to Error may fire indicating a violation of the refinement definition.

In the example in Figure 5.12, the state Neutral, the $C3a$-states and all in- and outgoing transitions of the $C3a$-states have been created according to Case 3a to check for required communication according to the timed bisimulation.

Case 3b: Message Obeys Upper Bound

Case 3b with the associated state $C3b$ checks that R does not send or receive μ later than A. In our approach, we only apply Case 3b for checking transitions that send a message μ when checking for a relaxed timed bisimulation.

The transition from S_{TA} to $C3b$ has a guard and a time guard. While the guard is the same as for $S_{TA} \longrightarrow S_{TA}{}'$, the time guard differs. In particular, the time guard conjuncts the upper bound cc_{high} of the clock constraint with the invariant I of S. Consequently, T_A may enter $C3b$ in a time interval that is restricted by the latest point in time where μ may be sent in A.

As long as the urgent transition to Neutral is enabled up to the upper bound of the time interval where $C3b$ was entered, no time may pass in $C3b$. Therefore, the transition to Error has a time guard that compares the clock c_{TA} to a maximum value that is larger than any value used in the clock constraints of A and R. Thus, the transition to Error may only become enabled if time passes in $C3b$ which may not be the case if R refines A correctly.

Case 3c: Message Eventually Sent or Received

Case 3c with the associated state $C3c$ checks that R eventually sends or receives μ not checking for time intervals. This is needed for bisimulation and for transitions receiving a message μ when checking for relaxed timed bisimulation.

The transition from S_{TA} to $C3b$ has a guard and a time guard. While the guard is the same as for $S_{TA} \longrightarrow S_{TA}{}'$, the time guard differs. In particular, the time guard conjuncts the upper bound cc_{high} of the clock constraint with the invariant I of S and requires that c_{TA} is 0. Consequently, T_A may enter $C3c$ in a time interval whose upper bound is restricted by the by the latest point in time where μ may be sent in A. The lower bound is restricted by the point in time where S_{TA} may be entered at the earliest expressed by $c_{TA} = 0$.

The transition to Neutral checks that R eventually sends or receives μ. The transition to Error is again guarded by an artificial maximum value t_{max} that shall never be reached in A or R. This transition will only become enabled if R never sends or receives μ.

5.3.3 Adjusting the Port Real-Time Statechart

"The port RTSC needs to be adjusted such that it may be combined with T_A to the parallel test system. The adjustments do not change the behavior of R but only ensure that the test constructs in T_A may correctly identify forbidden deviations of R from A. The result of this step is the adjusted port RTSC R_{adj}. Figure 5.13 shows an excerpt of an example for R_{adj} as constructed for the port RTSC of railroad crossings (cf. Figure 5.7)." [HBDS15]

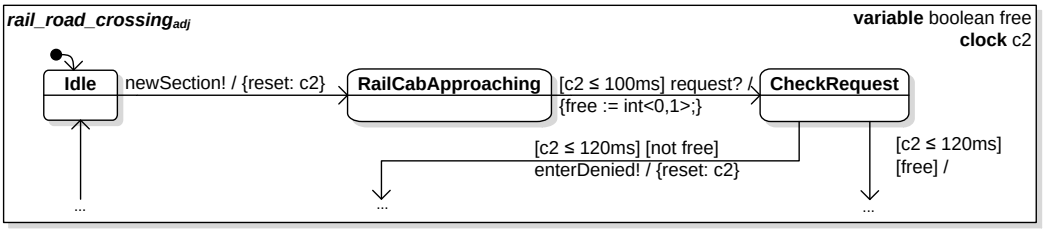


Figure 5.13: Adjusted Port RTSC for Railroad Crossings [HBDS15]

First, all sent and received messages need to be replaced by corresponding synchronizations, i.e., a sent message m_{out} is replaced with $m_{out}!$ and a received message m_{in} is replaced with $m_{in}?$. This is necessary for building the parallel test system using a network of timed automata.

Second, any invariants of states of R_{adj} need to be removed. The corresponding clock constraints are conjuncted with the time guards of the outgoing transitions. This prevents that an invariant in R_{adj} may stop time from progressing in T_A after both have been composed to the parallel test system. The modification does neither add nor remove externally visible

behavior of R because a transition is only enabled if its time guard is fulfilled and if the invariant of the source state is fulfilled. In the example in Figure 5.13, this modification affects the states RailCabApproaching and CheckRequest and their outgoing transitions.

Third, all transitions carrying a message are urgent in R_{adj} as shown in Figure 5.13. They need to synchronize urgently with the transitions from the CX states to the Neutral state in the test constructs checking for required communication (Cases 3a, 3b, and 3c). The transitions in R_{adj}, however, still synchronize non-urgently with the transitions between white states in T_A because the latter are all non-urgent (cf. Section 5.3.2.1).

Fourth, we remove all synchronizations of R with other RTSCs within the component RTSC. An example is given by the synchronization closeGate! at the transition from CheckRequest to ClosingGate in Figure 5.7. In our refinement check, we only check whether the externally visible behavior of R is correct with respect to A and the selected refinement definition. Checking that the integration with the remaining ports via these synchronizations has been done correctly is subject to Step 3 of our compositional verification approach.

"Finally, R may read and write variables of the component RTSC. An example is given by the variable sectionFree that is read by all refined protocols introduced in Section 5.1.2. From the perspective of R, such variables may change at arbitrary points in time. Verifying their correct usage by the component and all of its ports is, again, subject to Step 3 of the compositional verification approach. Therefore, we replace read accesses to component variables by non-deterministic choices, i.e., we assume that the variable may have any allowed value when it is accessed. In addition, we completely remove write accesses to component variables because they are irrelevant if the written value is never read." [HBDS15]

5.3.4 Parallel Composition and Reachability Analysis

We combine T_A and R_{adj} to the parallel test system based on an NTA as formally defined in Appendix B. The reachability analysis then computes the reachable state space in terms of a zone graph (cf. Section 2.2.1). Each path in the zone graph represents a trace consisting of a sequence of symbolic states.

In the reachability analysis, we search for a trace where the Error state of T_A is active in a symbolic state. This corresponds to verifying the formula ϕ `EF T_A.Error`. If this formula is fulfilled for $T_A \parallel R_{adj}$, the reachability analysis returns the trace that leads to the particular symbolic state where Error is active. This trace then serves as a counterexample that is provided to the developer. If ϕ is not fulfilled, the reachability analysis does not return a result and the conditions of the refinement definition are fulfilled for A and R. We provide an example of a counterexample in Section 5.6.

5.4 Implementation

We have implemented all algorithms shown in Figure 5.9 in version 0.4 of the MECHATRONICUML Tool Suite. Figure 5.14 shows the plugins that have been created as part of the implementation.

The plugin muml implements the component model of MECHATRONICUML including RTCPs and RTSCs (cf. Section 3.6). The plugin runtime enables to store the currently active states and the current values of variables for RTSCs. The plugin reachabilityGraph provides

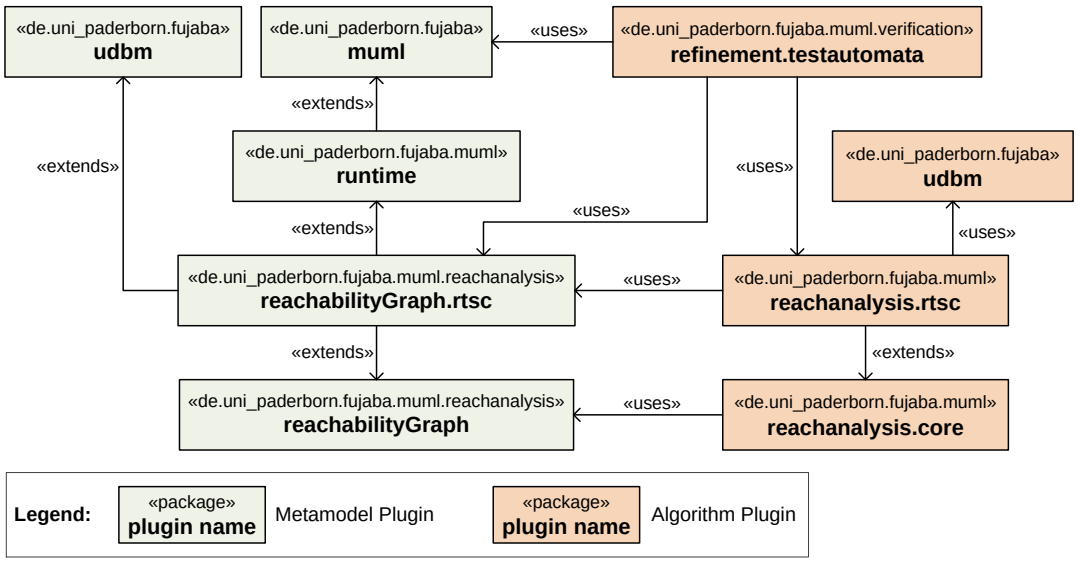


Figure 5.14: Plugins Implementing our Refinement Check

abstract superclasses for storing state spaces computed by a reachability analysis (cf. Appendix C). This plugin is extended by reachabilityGraph.rtsc for storing a zone graph that has been computed for a network of timed automata (cf. Section 2.4.2). In particular, this plugin uses the runtime plugin for representing the active states in $T_A \parallel R_{adj}$ and the current values of all variables. The clock values are stored in federations that are provided by the udbm plugin. Our udbm plugin [EH11] integreates an UPPAAL library [Dav06] for storing federations and executing operations on them using so-called difference bound matrices (DBMs, [Dil90]).

The aforementioned plugins are used by the algorithm plugins on the right side of Figure 5.14. The plugin refinement.testautomata contains the main parts of the refinement check. In particular, it implements the algorithm for selecting a suitable refinement definition (cf. Section 5.3.1), the construction of the test automaton as described in Section 5.3.2, and the adjustment of the port RTSC as described in Section 5.3.3. Our implementation underlies the assumptions and limitations discussed in Section 5.5. At present, our implementation may not yet automatically detect whether an existential quantifier has been used in the verified properties. Instead, we ask the developer using a dialog. All algorithms in this plugin have been implemented in Java based on the initial implementation provided by Brenner [Bre10].

Finally, the two plugins reachanalysis.core and reachanalysis.rtsc implement the reachability analysis on $T_A \parallel R_{adj}$. reachanalysis.core implements a state-space exploration based on a breadth-first search (BFS) while reachanalysis.rtsc implements the RTSC-specific computation of successor states. The result of the reachability analysis is a zone graph that is constructed according to the formal semantics defined in Appendix B. We provide a more detailed description of our framework for reachability analyses in Appendix C. Based on the constructed zone graph, the algorithm in refinement.testautomata may decide whether the refinement is fulfilled or not. Our algorithm exports counterexamples using the PDF or SVG file formats.

5.5 Assumptions and Limitations

Our test automaton construction underlies the following assumptions and limitations:

- Role RTSC and port RTSC are flat, i.e., they may not use hierarchical states.
- Transitions in the role RTSC may only send or receive a message but not both. This is no general limitation because transitions with two messages may be split into two transitions with one message each and an intermediate state.
- When checking for a relaxed timed bisimulation, the role RTSC must not use clock resets at transitions that receive a message.
- Messages must not have parameters.
- The RTCPs and port RTSCs to be checked underlie the assumptions on quality of service characteristics described in Section 2.4.3.

5.6 Case Study

"We evaluate our automatic refinement check based on a RTCP of the RailCab system. We conducted a case study based on the guidelines defined by Kitchenham et al. [KPP95]. In our case study, we investigate the correctness of our method for a realistic example within our domain and do not aim at generalizing this statement in this thesis."[HBDS15]

5.6.1 Case Study Context

The objective of our case study is evaluating whether our refinement check returns correct results for a realistic RTCP and corresponding correctly and incorrectly refined protocols. We conduct our case study based on the RailCab system using RTCP EnterSection and the refined protocols introduced in Section 5.1.

5.6.2 Setting the Hypothesis

"In Section 5.1.2, we presented three refined protocols. According to our expertise, two of them are correctly refined (the ones for normal sections and switches) and one is incorrectly refined (the one for railroad crossings). In addition, we consider a refined protocol of the role railcab where we did not apply any modification. This is a correct refinement with respect to any refinement definition.

For our case study, we define two evaluation hypotheses. Our *first evaluation hypothesis H1* is that our refinement check correctly identifies the correctly and incorrectly refined protocols. Our *second evaluation hypothesis H2* is that the counterexample that is produced for an incorrectly refined protocol enables to identify the reason for the violation of the conditions of the checked refinement definition.

For evaluating our hypotheses, we manually calculate the state spaces of the role section and of the three refined protocols. Based on the state spaces we manually check whether the conditions of the corresponding refinement definition are satisfied. We will compare the results of our refinement check to the results of our manual calculations. In addition, we will

derive a correctly refined protocol for railroad crossings based on the counterexample returned by the refinement check. We consider our evaluation to be successful, if the automatic refinement check returns the same results as our manual calculation and if we succeed in correcting the refined protocol for railroad crossings based on the counterexample."[HBDS15]

5.6.3 Preparing the Input Models

In preparation of the case study, we specify all models presented in Section 5.1 using our implementation described in Sections 3.6 and 5.4. The created models are available for download on our webpage [HBD13].

In particular, we define the RTCP EnterSection, the components shown in Figure 5.3, and the refined port RTSCs presented in Section 5.1.2. "We refine the role section at the ports NormalTrackSection.left, NormalTrackSection.right, Switch.left, Switch.right, Switch.bottom, RailRoadCrossing.left, and RailRoadCrossing.right. In addition, we refine the role railcab at the ports DriveLogic.section1 and DriveLogic.section2 (cf. Figure 3.6). We also specified the internal behavior of the components in Figure 5.3. We refer to Appendix A.5.2 for a description of these RTSCs.

5.6.4 Validating the Hypothesis

We apply our refinement check to the ports of section1 and section2 of RailCabDriveControl and to all ports of NormalTrackSection, Switch, and RailroadCrossing that refine the role section of protocol EnterSection.

We start by verifying the refinement for the ports of RailCabDriveControl. Our refinement check first selects timed bisimulation based on the decision tree in Figure 5.10 and the verification succeeds. Thereafter, we repeated the verification without existentially quantified formula such that the decision tree selects the timed ready simulation. Again, the verification succeeds as expected.

Concerning the ports of NormalTrackSection, our refinement check selects a relaxed timed bisimulation as stated in Section 5.3.1. The verification succeeds as expected. Next, the verification returns that the refined RTSC for the ports of the Switch is valid with respect to a timed bisimulation.

Finally, we check the ports of the RailroadCrossing. Again, our refinement check select a timed bisimulation. In this case, however, the refinement is invalid with respect to timed bisimulation. Therefore, our refinement check returns the counterexample shown in Figure 5.15 for this violation.

"The counterexample consists of six symbolic states. It has been obtained by performing a reachability analysis on the parallel composition of the test RTSC shown in Figure 5.12 and the adjusted RTSC of rail_road_crossing shown in Figure 5.13. In the counterexample, the test RTSC is denoted as SectionTA_TBS while rail_road_crossing denotes the adjusted RTSC of rail_road_crossing. In the first symbolic state S1 of the counterexample, both RTSCs are in their initial Idle states and all clocks are zero. Please note that we convert all time units to milliseconds in our implementation and, therefore, do not visualize time units in the counterexample. Then, the RTSCs synchronize via newSection and enter the RailCabApproaching states. In the next step, the RTSCs synchronize via request and enter the CheckRequest states. In both symbolic states, all clocks are still zero. In S3, the variable free receives the value

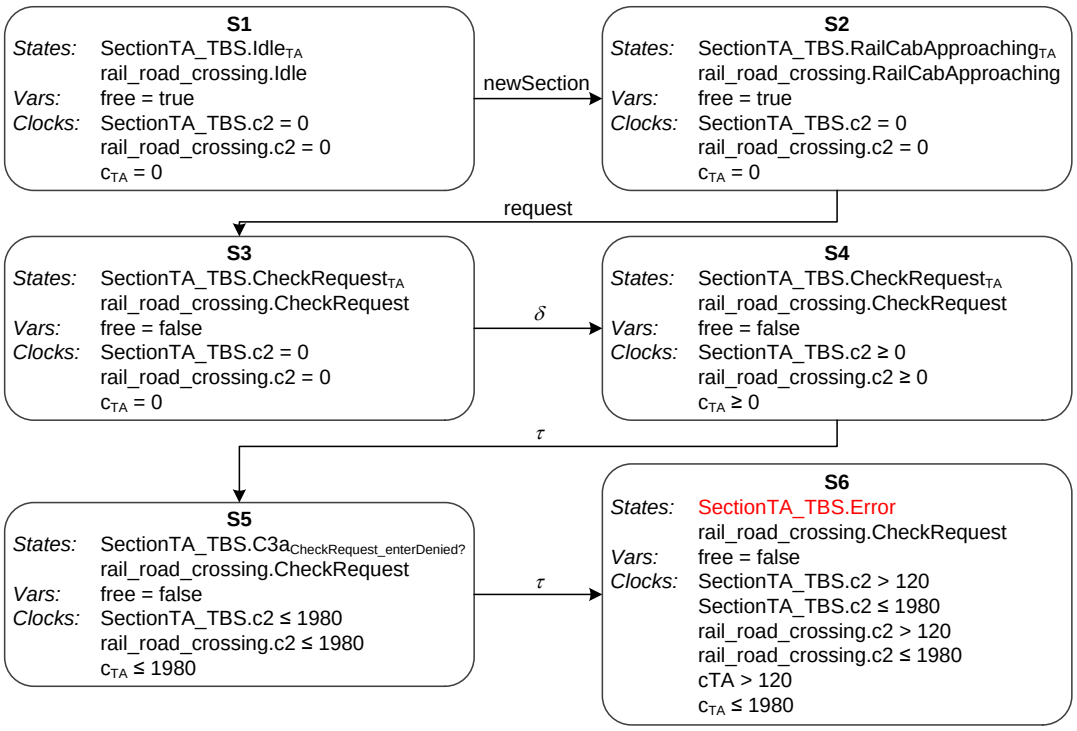


Figure 5.15: Counterexample for the Incorrectly Refined Behavior for Railroad Crossings [HBDS15]

false. The next transition in the counterexample is a delay transition (cf. Section 2.2.1). Since we removed all invariants and moved the corresponding clock constraints to the outgoing transitions (cf. Section 5.3.2), the clock values are not restricted. Thus, all clocks have an unbounded value greater or equal to zero. The transition from S4 to S5 is a so-called tau transition where the test RTSC fires a transition without synchronization (cf. Section 2.2.1). In particular, it enters the C3a$_{CheckRequest_enterDenied?}$ state that checks whether rail_road_crossing may send the message enterDenied in the same time interval as the section role. When entering the state, all clocks have a value less or equal to 1980 resulting from the clock constraint of the transition from CheckRequest$_{TA}$ to C3a$_{CheckRequest_enterDenied?}$. In the final symbolic state S6, the test RTSC is in state Error and, thus, the refinement does not hold. The clock values show that the Error state has been reached with all clocks having a value in the interval $120 < c2 \leq 1980$.

From the counterexample, we can deduce that the refinement is violated by the transition from CheckRequest to EnterDenied in rail_road_crossing that sends the message enterDenied. Since the Error state has been reached via the C3a$_{CheckRequest_enterDenied?}$ state, we also know that the refined RTSC of the railroad crossing does not support sending enterDenied in the whole time interval that is required by the section RTSC. In particular, the transition enables to send the message enterDenied only until c2 is at 120 ms, while the RTSC for role section allows sending until c2 is at 1980 ms.

Using this information, we derive a correctly refined RTSC for the railroad crossing that is shown in Figure 5.16. In particular, we correct the error by introducing the state SendDenial.

If the railroad crossing is not free, we do not immediately switch to EnterDenied, but we switch to SendDenial. This state enables to send enterDenied until c2 is at 1980 ms as it is required by the timed bisimulation. The resulting RTSC fulfills the conditions of the timed bisimulation."[HBDS15]

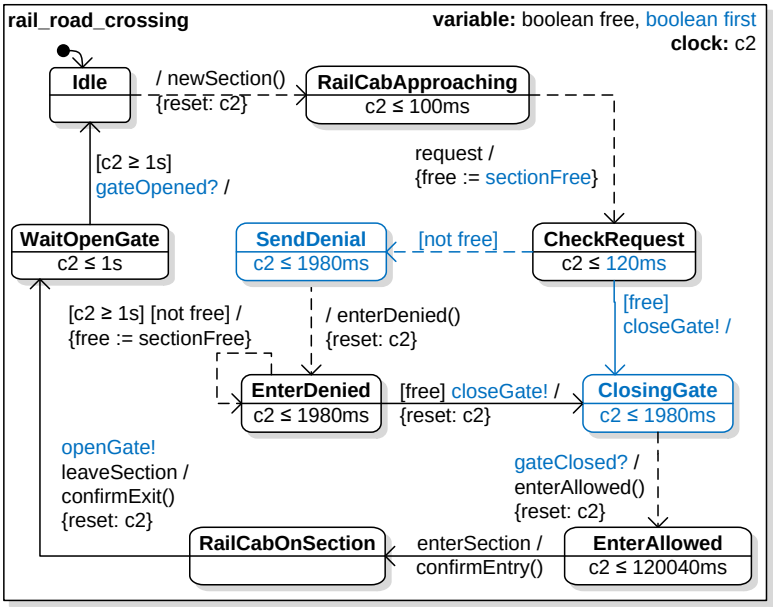


Figure 5.16: Correctly Refined Behavior for Railroad Crossings [HBDS15]

5.6.5 Analyzing the Results

"The results of our case study show that our refinement check correctly identifies for all given refined protocols whether they are refined correctly or not. Therefore, our first evaluation hypothesis H1 is fulfilled. In addition, we were able to identify the cause of the incorrect refinement based on the counterexample that was returned by our automatic refinement check. That enabled us to correct the refined protocol and, as a result, our second evaluation hypothesis H2 is fulfilled as well. This gives rise to the assumption that our approach is also applicable to other realistic examples within our domain.

In our case study, the most important *threats to validity* are as follows: (1) We might have made mistakes in the manual calculation that identified which refined protocols are a correctly refined and which are not. (2) We only considered one abstract protocol and four different refinements. Even though we consider this example as realistic, other realistic protocols could be highly different. (3) We did not check all possible refinement definitions but only timed ready simulation, timed bisimulation, and relaxed timed bisimulation. However, checking for a timed bisimulation includes checks for a (timed) simulation and a bisimulation. In particular, the refinement definitions that we checked explicitly cover all cases of our construction (cf. Section 5.1). (4) We are experts for verification and refinement checking. Therefore, we could easily derive the cause for the violation of the refinement from the counterexample which might not be true for novices." [HBDS15]

5.7 Related Work

We discuss related work from two research areas. First, we review other approaches that support refinements of real-time behavior and approaches that use multiple refinement definitions (Section 5.7.1). Second, we review related works that use test automata for verification (Section 5.7.2).

5.7.1 Refinement Checking

"Reeves and Streader [RS08a, RS08b] identify commonalities and differences of refinement definitions for process algebras and unify them in a generalized definition but provide neither a selection nor a verification algorithm. Sylla et al. [SSdR05] present a refinement definition including a refinement check where the refinement is parameterized by a particular LTL formula [BK08] such that only this particular formula is preserved. In contrast to our approach, both do not consider real-time properties.

In [Bey01], Beyer introduces timed simulation for Cottbus Timed Automata which are a special kind of timed automata. We cover this refinement definition in our refinement check. In addition, there exist refinement definitions based on (timed) I/O automata [dAH05, dAHS02] as, for example, (timed) alternating simulations and bisimulations [AHKV98, DLL+10]. These approaches use a two player game for deciding whether a refined component behaves in the same way as the abstract component in an unknown environment. In our approach, the behavior of the environment is not unknown but formally defined by the RTCPs. Therefore, we do not need to check refinement based on a two player game. Instead, we may use a simple reachability analysis using our test automaton.

The FOCUS approach [BS01] supports the specification of embedded systems. It defines the behavior of components by stream-processing functions on streams of messages [Ste97] and uses, according to [BS01], three kinds of refinements. Two of them, namely interface refinement and conditional refinement, enable to modify the input/output behavior of the system. In particular, they allow to change the number of messages, the types of parameters, and the encoding of data. These refinements support a top-down refinement of the system's behavior including, in particular, the modification of component internal behavior. In contrast, we only refine the interface behavior of our components, which is more restrictive. As a result, their approach requires to consider the internal behavior of the sending and receiving component in addition to the interface behavior. That makes refinement checking much more expensive compared to our approach and scalability becomes a problem. The third refinement, which is called behavioral refinement, defines the conditions for simulation and bisimulation. According to [HHR09], interface abstractions of component behavior are equivalent to statemachines that, in turn, correspond to I/O automata. Refinements for I/O automata have been discussed in the previous paragraph. In contrast to our approach, [HHR09] supports completely non-deterministic specifications." [HBDS15]

5.7.2 Test automata-based Verification

"Test automata are used by Aceto et al. [ABBL03] for model checking temporal properties specified in SBBL (Safety Model Property Language) on timed automata rather than verify-

ing correct refinements. Their test automata construction encodes a temporal logic formula. Consequently, they use a different set of test constructs compared to our approach.

The approaches by Gerth et al. [GPVW96] and Tripakis et al. [Tri09] perform LTL model checking [BK08] on (timed) Büchi automata and encode the properties in automata as well. Again, they use a different construction because they encode a temporal logic formula instead of a refinement definition.

Li et al. [LBD+10] specify safety and liveness properties for timed automata as live sequence charts (LSC, [HM03]). They translate the LSC into an observer timed automaton that enters a special error location if the property is violated. Since they encode a LSC, they also use different test constructs compared to our approach." [HBDS15]

5.8 Summary

In this chapter, we extend the compositional verification approach of MECHATRONICUML by five additional refinement definitions and an integrated refinement check. Our refinement check may automatically select a suitable refinement definition based on the role RTSC, the port RTSC, and the properties that have been verified for the RTCP. Then, our refinement check generates a so-called test automaton that encodes the behavior of the role RTSC and the conditions of the applied refinement definition. Our approach extends the construction of test automata by Jensen et al. [JLS00] by additional test constructs that enable to check all of the six considered refinement definitions. In our evaluation, we showed that our approach was successfully applied to a RTCP of the RailCab system.

Our refinement check relieves developers of NMS from choosing a refinement definition manually. As a result, developers require less detailed expertise in refinements when refining roles of a RTCP to ports of a component. An additional advantage of our refinement check is that additional refinement definitions, if needed, only require minor extensions of the decision tree and the test automaton construction. The remaining algorithms do not need to be changed.

6 Simulating Self-Adaptive Mechatronic Systems in MATLAB/Simulink

Our component model as introduced in Chapter 3 supports to specify a software architecture for a self-adaptive mechatronic system. Therefore, it enables to specify and connect discrete components that contain event-discrete behavior specified by RTSCs and continuous components that contain feedback controllers. In addition, our concept for hierarchical reconfiguration as introduced in Chapter 4 enables to jointly reconfigure both kinds of components. Such joint reconfiguration is necessary, for example, for a RailCab that wants to join a convoy as a member. As described in Section 4.2, this reconfiguration requires (1) to instantiate a discrete component, (2) to replace a continuous component instance by another one, and (3) to connect these component instances. As a result, correctness of the RailCab's behavior with respect to this reconfiguration depends on the correct interaction of discrete and continuous components as well as the correct integration of fading functions, which are used for replacing continuous component instances, in a hierarchical reconfiguration. Thus, an error in the interaction or an erroneous fading function may lead to a crash when a RailCab enters a convoy.

Therefore, it is desirable to verify the correctness of the behavior by applying formal verification techniques for proving that such errors may not occur. Verifying the behavior of a RailCab for joining a convoy as a member, however, induces a so-called hybrid model checking problem [Hen96] because it includes event-discrete RTSCs as well as time-continuous feedback controllers and fading functions. At present, hybrid model checking approaches like PHAVer [Fre05] or SpaceEx [FLGD$^+$11] either use very simple models of time-continuous behavior or rely on several, manual overapproximation steps. Both kinds of approaches suffer from a loss of precision leading to potentially wrong verification results and may still only be applied to small models of academic nature [ERNF12]. Therefore, the correctness of the behavior of mechatronic systems is typically only assessed by testing, which cannot prove the absence of errors.

Our goal is to provide the best possible compromise between formal verification and testing approaches while avoiding the need for hybrid verification. In particular, we want to fully verify all of the discrete components for proving that they are free of errors. Then, testing is only necessary for checking the behavior of continuous components and their integration with the discrete components. As a basis, the 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 hybrid ports change and we do not allow, in particular, to include time-continuous variables in RTSCs. This enables to efficiently verify the discrete components based on MECHATRONICUML's compositional verification approach as outlined in Chapter 5. At the same time, we can use established

approaches based on model-in-the-loop (MIL) simulation [Plu06] for testing the continuous components of the mechatronic system using tools like MATLAB/Simulink [Matg] or Dymola [Das]/Modelica [Mod09]. MIL simulation and the corresponding tools are already successfully applied in industry [TDH11, KSHL12]. In contrast to MECHATRONICUML, the tools for MIL simulation of mechatronic systems do not support modifications of a simulation model during a simulation run. Consequently, these tools do not natively support runtime reconfiguration that we need, for example, for realizing the convoy mode of the RailCab system. In addition, they do not support communication between systems based on asynchronous messages.

In this chapter, we define an approach for performing a MIL simulation for a self-adaptive mechatronic system whose software architecture and behavior have been specified using MECHATRONICUML. In our approach, we consider both, the reflective operator and the controller level of the OCM. As our main contribution, we define how a model specified in MECHATRONICUML may be represented in a simulation environment that provides no built-in support for message-based communication and runtime reconfiguration. As a simulation tool, we use MATLAB/Simulink because it is widely used in industry and well supported by code generators like TargetLink [dSP] or ASCET [ETA] that enable to generate production code for the final system.

Our approach solves three particular challenges. First, the reconfiguration controller of a structured component operates on a model@runtime that is shared between manager, executor, and runtime risk manager. Since MATLAB/Simulink does not have a model@runtime, we define an explicit encoding of the model@runtime in MATLAB/Simulink. Second, since MATLAB/Simulink does not allow to modify the simulation model during a simulation run, we enumerate and encode all configurations beforehand in the MATLAB/Simulink model such that we may switch between these encoded configurations at runtime. Third, MATLAB/Simulink only supports communication via synchronous signals that may only be received as long as they are sent. Therefore, we provide helper blocks for MATLAB/Simulink that realize message-based communication and respect the QoS assumptions of MECHATRONICUML. The helper blocks only use build-in features of MATLAB/Simulink. In contrast to related works like Mosilab [ZJS08], Sol [Zim07], or the block library by Kovácsházy et al. [KSP03], this retains the ability to use the code generators for generating production code for the system out of the resulting MATLAB/Simulink model.

Our translation needs to preserve the semantics of the (verified) MECHATRONICUML model. Proving correct preservation of semantics is currently not possible because the operational semantics of Simulink and Stateflow are intellectual property of The MathWorks and solely defined by the implementation of the simulator and code generators [TSCC05]. Although approaches for defining a formal semantics exist for both, Simulink [TSCC05, BC12] and Stateflow [Ham05, HR07, Wha10], these are incomplete with respect to the features that we require and cannot guarantee to be correct because they also have only been checked based on a few test cases [TSCC05, BC12]. Therefore, we only provide an informal description of how we preserve semantics. In addition, we have tested the semantics of the resulting Simulink and Stateflow models using our implementation.

In the following, we first introduce basic concepts of Simulink and Stateflow that are required for understanding the concepts of our translation (Section 6.1). Thereafter, Section 6.2 introduces our approach for MIL simulation in MATLAB/Simulink. As part of this section, we define an algorithm consisting of several steps for translating a MECHATRONICUML

model into a MATLAB/Simulink model. Sections 6.3 to 6.5 describe the steps of the algorithm in more detail. Then, we describe our implementation of the algorithm in Section 6.6. Section 6.7 discusses the limitations of our approach and our implementation. Section 6.8 presents the results of our case study. Finally, we discuss related works in Section 6.9 before summarizing the approach in Section 6.10.

The contents presented in this section have been published in [HPR+12] and [HRS13]. The translation of the reconfiguration specification has been contributed by the Bachelor's Thesis of Pines [Pin12] and the Master's Thesis of Volk [Vol13].

6.1 MATLAB/Simulink and Stateflow

MATLAB® is a tool environment for numeric computations and visualizations [ABRW09, Matd]. MATLAB is extended by a set of tool boxes that provide additional modeling and simulation capabilities. The most important tool box for modeling and simulating mechatronic systems is Simulink® [Matg]. Simulink, in turn, is extended by the Stateflow® [Math] tool box for specifying state-based behavior. Since all features of MATLAB that we require for performing MIL simulations of mechatronic systems are offered by Simulink and Stateflow, we restrict our descriptions in Sections 6.1.1 and 6.1.2 to these tool boxes. We refer to Angermann et al. [ABRW09] for a detailed introduction to MATLAB.

6.1.1 Simulink

The Simulink toolbox supports the model-based design and simulation of technical systems [Matg]. Since mechatronic systems are a special kind of technical systems, their development is supported as well. A Simulink model is a block diagram that consists of blocks and lines. Figure 6.1 shows a simple Simulink model that contains the most important blocks that we use in our approach. The model computes a function based on two values and plots the result if it is greater or equal to 0.

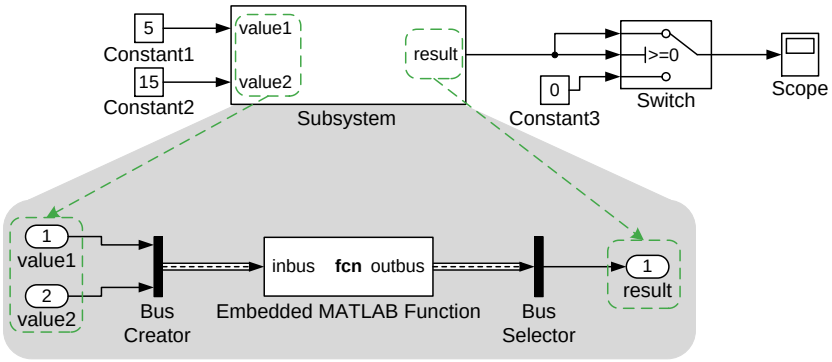


Figure 6.1: Simple Simulink Model

In the upper part of Figure 6.1, all blocks except Subsystem are *basic blocks* that are provided by the Simulink block library. The blocks Constant1, Constant2, and Constant3 are constant blocks that constantly emit the specified value. The switch block named Switch enables to select a data input (upper and lower input) based on a control input (middle input).

If the control input fulfills the switch criterion, $>= 0$ in the example, then the switch outputs the upper data input, otherwise the lower data input. The Scope block visualizes a 2D-plot of its input signal.

In Simulink, blocks are connected by lines that are visualized by arrows. Lines define how data flows between the blocks. They represent signals while "a signal is a time varying quantity that has values at all points in time" [Matf]. A signal has, among others, a data type and a dimension. The dimension defines whether the signal is a scalar or an (multidimensional) array. Signals are directed and can be forked. As an example, the signal from Subsystem to Switch in Figure 6.1 is forked into two signals. Forks are visualized by a small black dot on the line.

A Simulink model specifies a sample time that defines how often the values of the blocks in the model are recalculated. For a sample time of 1 ms, the values are recalculated every millisecond with respect to the simulation time.

In Figure 6.1, Subsystem is a *subsystem block* that can be used to group Simulink models hierarchically. Normal subsystems are virtual, i.e., they do not influence the semantics of the model. Subsystem has two inports named value1 and value2 and one outport named result. Data enters a subsystem via the inports and leaves the subsystem via the outports.

The lower part of Figure 6.1 shows the internals of the subsystem block Subsystem. The oval shaped blocks represent the inports and outports of the subsystem. The block in the middle is an embedded MATLAB function block that enables to specify user-defined scripts. An embedded MATLAB function may have multiple inputs and may produce multiple outputs.

In our example, the embedded MATLAB function operates on a bus signal that is created by the Bus Creator block. A *bus signal* is a composite signal that groups a set of named signals [Matb]. It is comparable to a structure in the programming language C [KR88]. Buses enable to group signals of different data types and dimensions. The Bus Selector enables to extract a signal from the bus. In the concrete syntax, buses are visualized as "a triple line with a dotted core" [Matf]. We will use bus signals in our approach for representing messages (cf. Section 6.3.3).

The upper half of Figure 6.2 shows a special kind of subsystem: the *enabled subsystem*. In addition to a normal subsystem, it has an enable port at the top. If the signal at the enable port is 0, then the enabled subsystem is off and not simulated. If the signal at the enable port is 1, then the enabled subsystem is on and simulated. We will exploit this behavior for emulating the creation and deletion of component instances in Section 6.5.4.

Inside the enabled subsystem, we specified a *chart block* named Chart as shown in the lower half of Figure 6.2. A chart block embeds a Stateflow chart (cf. Section 6.1.2). The Stateflow chart may interact with the Simulink model via inports and outports which is crucial for translating message-based communication of RTSCs to Stateflow as discussed in Section 6.3.3. In our example, the chart block receives a speed from the Simulink model and emits a distance via dist and an information whether the RailCab drives fast or not via fast. We introduce the contents of the chart block in more detail in the subsequent section.

6.1.2 Stateflow

The Stateflow toolbox extends Simulink towards "modeling and simulating combinatorial and sequential decision logic based on state machines and flow charts" [Math]. As a result, a Stateflow model consists of states and transitions. We will use Stateflow for mapping RTSCs

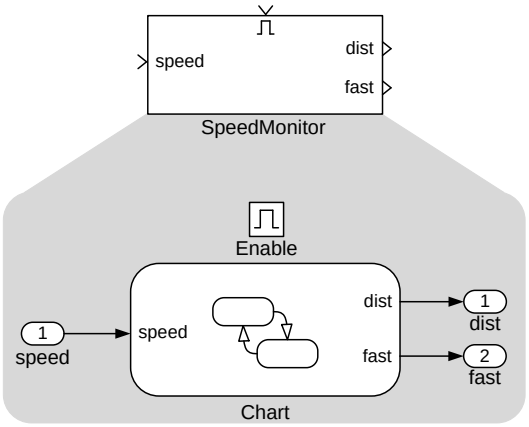


Figure 6.2: Enabled Subsystem

of MECHATRONICUML to MATLAB. Figure 6.3[1] shows a simple Stateflow chart that is embedded in the chart block of Figure 6.2. It implements a simple behavior for detecting whether a RailCab drives slow or fast and for updating the reference distance accordingly.

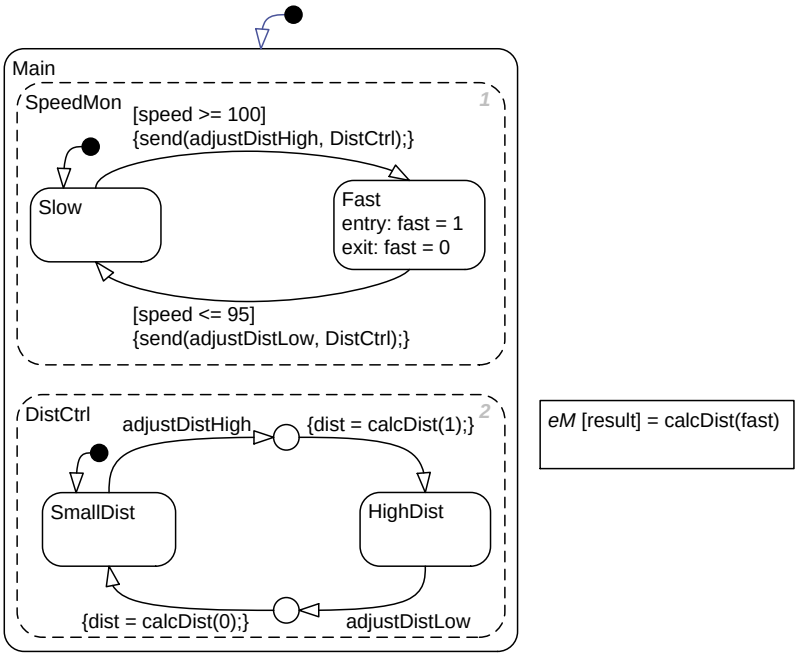


Figure 6.3: Simple Stateflow Chart

The chart contains a state Main. Main has a so-called default transition that marks it as the initial state. In addition, Main contains two embedded states SpeedMon and DistCtrl. These

[1] Please note that we use a black and white notation of Stateflow charts instead of the default colored notation. We do so to avoid confusion with the colored elements that appear in generation templates for Stateflow charts that we use in the remainder of this chapter.

states are *parallel states* as indicated by the dashed border line, whereas all other states are *exclusive states*. If Main is active, then SpeedMon and DistCtrl are both active. Inside SpeedMon, for example, only one of the two states Slow and Fast is active at any point in time. Parallel states specify an execution order in the upper right corner. If the chart is executed, then parallel states are executed according to increasing numbers.

In our example, the state SpeedMon monitors the value of the input speed and outputs whether the RailCab drives slows or fast. DistCtrl computes a distance value based on the RailCab's speed that serves as an output. Initially, the RailCab drives Slow and uses a small distance (SmallDist). If the input speed of the chart block becomes greater or equal to 100, the transition from Slow to Fast in SpeedMon becomes active due to the transition guard. Upon firing, the transition sends a signal event adjustDistHigh to the DistCtrl state. A signal event may either be sent to the whole chart or to a specific hierarchical state inside the chart. The signal event triggers the outgoing transition of SmallDist, i.e., the transition cannot fire until it receives the signal event. The receiving transition is then executed immediately after sending the signal. A sent signal needs to be consumed within the same execution step and will no longer be available in the next one. After the receiving transition has been executed completely, the execution resumes at the sending transition.

In each simulation step, a chart only executes one transition for each hierarchical state. The only exception to this rule is given by connective junctions. In a connective junction, the Stateflow chart may not rest, i.e., it needs to fire transitions until it reaches a state. As a consequence, there always needs to exist one enabled outgoing transition for a connective junction. As an example for a connective junction, consider the outgoing transition of SmallDist. After entering the connective junction, which is visualized by a circle in the concrete syntax, the chart immediately fires the outgoing transition to HighDist.

The transition to DistHigh calls a function calcDist and assigns its return value to the output dist of the chart block. The function is specified as an embedded MATLAB function inside the chart. As in Simulink, these functions may have an arbitrary number of input and output parameters. In Stateflow, functions are called according to call-by-value and may not modify any variables of the chart. As a result, if the function needs to change a value, it must be returned as an output parameter and explicitly assigned.

The state Fast in SpeedMon specifies an entry and an exit action. The entry action is executed when the state becomes active, the exit action is executed when the state is left. In our example, both, entry and exit action, assign a value to the output fast of the chart block.

6.2 MIL Simulation of MechatronicUML Models in Simulink and Stateflow

In the following, we describe our concept for testing the correct integration of discrete and continuous components in a self-adaptive mechatronic system based on MIL simulation. Our concept requires sofware engineers and control engineers to collaboratively perform several steps that are summarized in the process shown in Figure 6.4. This process specifies Step S_5 of our overview process in Figure 1.3 on Page 8 in more detail. In the following, we refer to the software engineers and the control engineers simply as the developers if they work collaboratively on a process step.

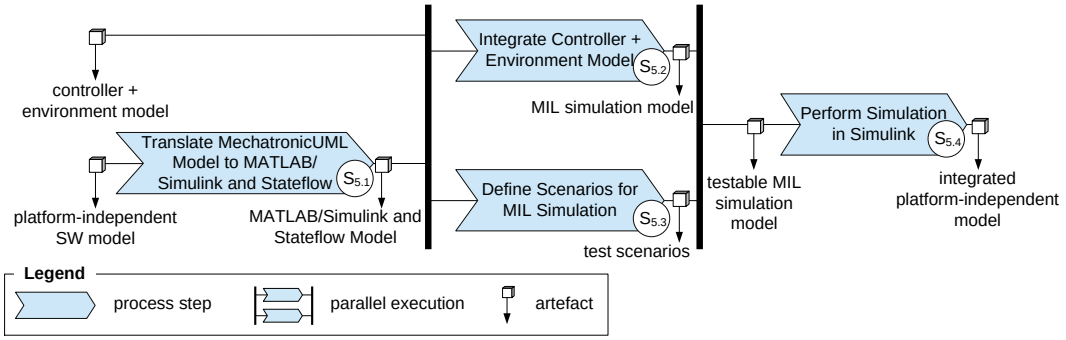


Figure 6.4: Process for Performing a MIL Simulation of a MECHATRONICUML Model in Simulink and Stateflow

The software engineer starts in Step $S_{5.1}$ by translating the verified MECHATRONICUML model into a MATLAB/Simulink and Stateflow model. The next two process steps may be performed in parallel. In Step $S_{5.2}$, the developers need to integrate the controller and environment models into the Simulink model that resulted from Step $S_{5.1}$. These models have been specified directly in Simulink by the control engineers. At the same time, the developers need to define scenarios for the MIL simulation in Step $S_{5.3}$. These scenarios are test cases that define a particular environmental situation and generate suitable stimuli for the MIL simulation model. In our RailCab example, we may, for example, define a scenario where two RailCabs start a convoy at a switch. In this case, the scenario defines where the RailCabs will start to drive on the track system and it configures the operation strategy such that the RailCabs will start a convoy. Based on the MIL simulation model and the test scenarios, the developers may perform the MIL simulation in Simulink in Step $S_{5.4}$.

In the remainder of this chapter, we will focus on Step $S_{5.1}$ and derive a concept for automatizing the translation from MECHATRONICUML models to MATLAB/Simulink and Stateflow models. Steps $S_{5.2}$ to $S_{5.4}$ need to be carried out manually by the developers. In particular, deriving a set of scenarios from requirements is beyond the scope of this thesis.

Figure 6.5 summarizes the algorithm for translating a MECHATRONICUML model into a MATLAB/Simulink model. The inputs are the (reconfigurable) components that are used in the MECHATRONICUML model and an initial CIC of the system. The latter defines how many component instances exist on the system level, e.g., how many RailCabs should be simulated. The output is a Simulink and Stateflow model that shows the same behavior as the MECHATRONICUML model.

The algorithm in Figure 6.5 consists of two phases. The first phase is defined by the expansion region and consists of Steps 1 to 4. These steps explicitly enumerate all possible configurations for each component that is used in the MECHATRONICUML model and encode these configurations. These steps need to be executed separately for each (reconfigurable) component. The second phase consists of Steps 5 to 7. These steps create the simulation model in Simulink and Stateflow and are executed once after the first phase has been finished.

In the first phase, we start in Step 1 by computing the possible configurations for each reconfigurable component based on the CSDs contained in its reconfiguration controller. If a component is not reconfigurable, it only has one possible configuration at runtime. There-

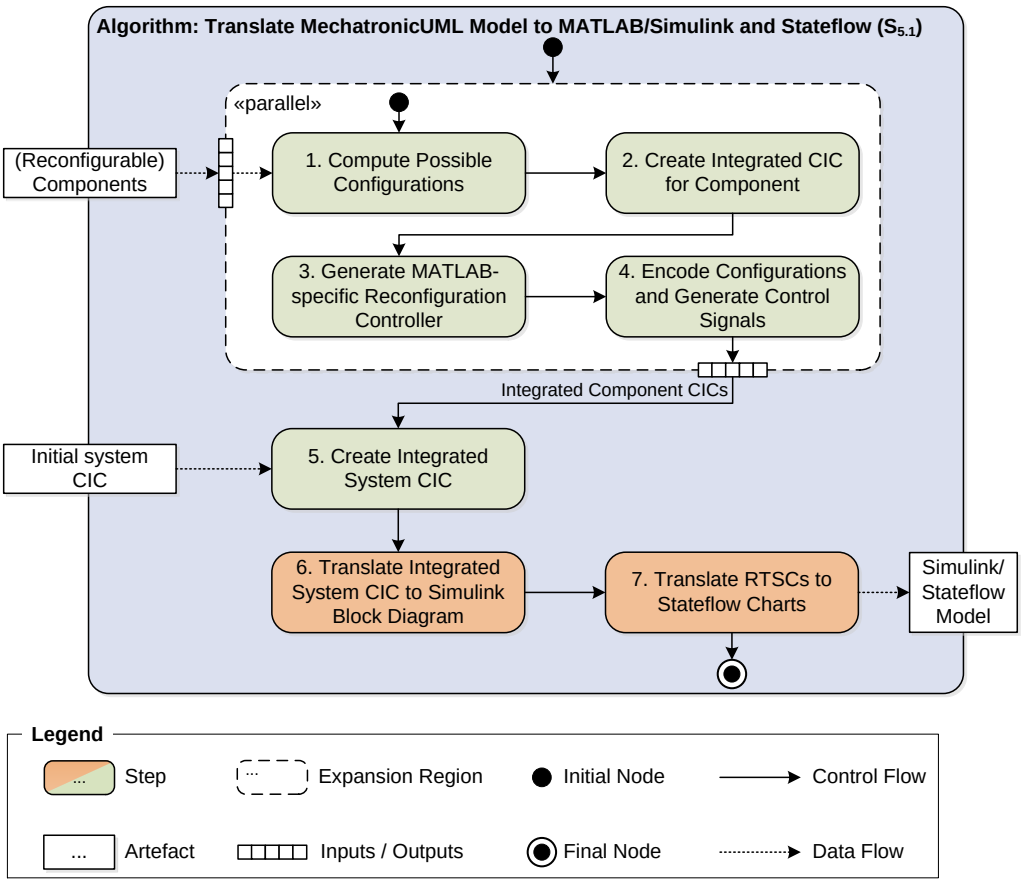


Figure 6.5: UML Activity Diagram Defining the Algorithm for Translating a MECHATRONICUML Model into a MATLAB/Simulink and Stateflow Model

after, we create a so-called integrated CIC for the component that contains the superposition of all possible configurations that were computed in Step 1. Based on this, we generate a MATLAB-specific reconfiguration controller in Step 3. The MATLAB-specific reconfiguration controller contains an additional component for encoding the model@runtime as given by the integrated CIC. The MATLAB-specific reconfiguration controller enables to execute reconfigurations using control signals that are computed in Step 4. The result of this phase is one integrated CIC for each component of the MECHATRONICUML model.

In the second phase, we create the Simulink and Stateflow models. Therefore, we use the integrated CICs in Step 5 for generating an integrated CIC for the overall system based on the initial system CIC. The result is a MECHATRONICUML model that (a) contains all possible configurations that the system may use at runtime and that (b) contains an explicit encoding of the model@runtime including control signals for switching between the encoded configurations. In Step 6, we translate the integrated system CIC to a Simulink block diagram. Thereby, we generate additional helper constructs for emulating message-based communication. In Step 7, we translate the RTSCs of the component instances in the integrated system CIC to Stateflow charts.

In Figure 6.5, we highlighted Steps 6 and 7 with different color because they may be used without prior execution of Steps 1 to 5 for translating a MECHATRONICUML model not employing runtime reconfiguration to MATLAB/Simulink. We address this use case in detail in our technical reports [HRB+13, HRB+14].

In the following, we start in Section 6.3 by describing the translation of a CIC to MATLAB/Simulink (Step 6). Thereafter, Section 6.4 describes the translation of RTSCs to Stateflow charts (Step 7). We describe these steps first because this translation determines how we need to encode configurations and which control signals we require for switching between configurations. Finally, Section 6.5 describes Steps 1 to 5 of our algorithm in detail and explains how the MATLAB-specific reconfiguration controller is integrated into the Simulink block diagram.

6.3 Translating Component Instance Configurations to Simulink Block Diagrams

This section defines the translation of a CIC into a Simulink block diagram as introduced in Section 6.1.1. Thus, it defines Step 6 of our algorithm shown in Figure 6.5. The input to this step is an integrated system CIC that contains one or more hierarchical component instances. The output is a Simulink block diagram that contains a set of subsystems. These subsystems reflect the structure of the CIC.

In the following, we start by illustrating how we translate atomic component instances (Section 6.3.1) and structured component instances (Section 6.3.2). Thereafter, we show in more detail how we encode message-based communication in Simulink using several kinds of helper blocks (Section 6.3.3). Finally, we discuss how the generated Simulink model considers the QoS assumptions (Section 6.3.4) that are specified in MECHATRONICUML (cf. Section 2.4.3).

6.3.1 Translating Atomic Component Instances

We create one enabled subsystem for each atomic component instance that appears in the CIC. Using enabled subsystems enables to emulate the creation and deletion of component instances by enabling and disabling the subsystem. Figure 6.6 shows the generation template for deriving the external interface of the enabled subsystem from the atomic component instance. In particular, the generation template defines how the port instances of a component instance are represented in Simulink. This step of the translation is identical for each type of atomic component instance.

Continuous and hybrid port instances are directly translated to inports[2] and outports of the subsystem because both receive or emit a signal value. The inports and outports in Simulink have the same names as the port instances in MECHATRONICUML. We strive at preserving the names of modeling constructs of MECHATRONICUML in Simulink because it enables the developers to relate simulation errors occurring in Step $S_{5.4}$ of our process in Figure 6.4 to the MECHATRONICUML model.

[2]Please note that we use the terms *in-port* and *out-port* for referring to ports of a MECHATRONICUML component, while *inport* and *outport* refer to ports of a Simulink subsystem.

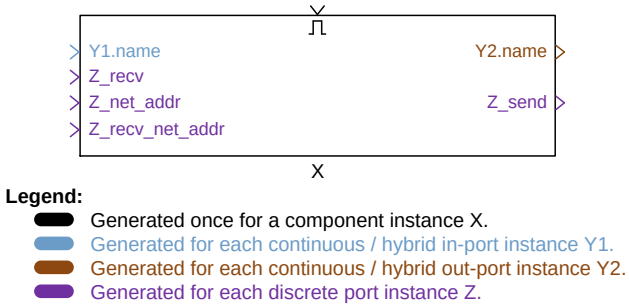


Figure 6.6: Generation Template for Creating a Subsystem for an Atomic Component Instance

Discrete port instances cannot be directly translated because they may send and receive messages which are not natively supported by Simulink. Therefore, we translate discrete port instances to *port structures* that consist of three inports and one outport [HPR+12, HRB+14]. Figure 6.6 shows the port structure that is generated for a discrete port instance Z. In the port structure, the inport Z_recv is used for receiving messages while the outport Z_send is used for sending messages. The inports Z_net_addr and Z_recv_net_addr define addresses for the message exchange that we explain in more detail in Section 6.3.3.

In the following, we explain how we generate the internals of the enabled subsystem in Figure 6.6. The internals of the enabled subsystem differ based on the type of atomic component instance being translated. We explain the internals of subsystems that result from discrete atomic component instances in Section 6.3.1.1. Thereafter, we describe the internal structure generated for continuous atomic component instances in Section 6.3.1.2 and for fading component instances in Section 6.3.1.3. We refer to Appendix A.8.1 for an example of an enabled subsystem that has been generated based on the template in Figure 6.6.

6.3.1.1 Discrete Atomic Component Instances

Discrete atomic component instances contain a behavior specification in terms of an RTSC including the message buffers for the port instances. Therefore, we create a block diagram that is embedded in the enabled subsystem that has been created for the atomic component instance. This block diagram contains, in particular, a chart block containing the Stateflow chart that implements the RTSC of the discrete atomic component instance. Figure 6.7 shows the generation template that is used for generating the block diagram that is embedded in a subsystem for a discrete atomic component instance. An example of a block diagram that results from applying the generation template to a discrete atomic component instance is presented in Appendix A.8.1.

The chart block is shown at the top of Figure 6.7. In addition, we generate one subsystem called *link layer* for each discrete port instance Z of the discrete atomic component instance [HPR+12]. The link layer subsystem has been developed as a part of this thesis. It handles the message-based communication via one port instance and implements the message buffer of this port instance. These are directly connected to the port structure generated for Z. We explain the link layer subsystems in more detail in Section 6.3.3.

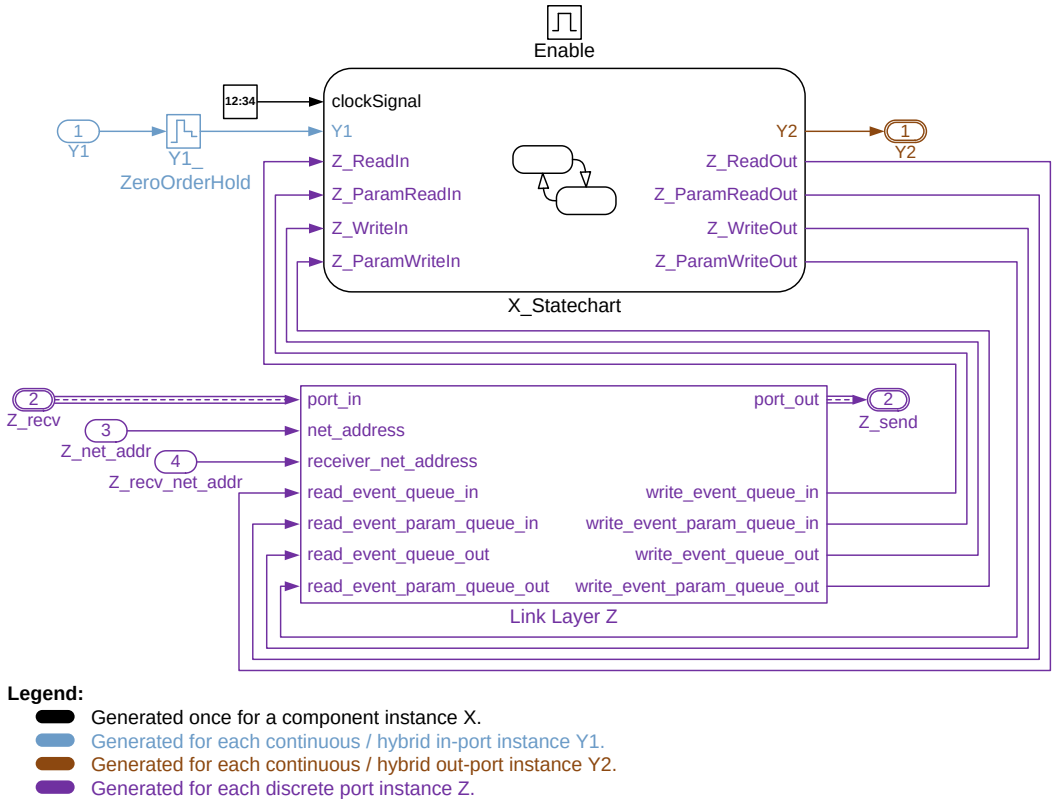


Figure 6.7: Generation Template for Creating the Internal Structure of a Subsystem for an Atomic Component Instance

Hybrid ports of a discrete atomic component instance are directly connected to the Stateflow chart by a line. As a result, they may be used as variables inside the Stateflow chart. For hybrid in-port instances, we additionally generate a *zero order hold* block in Simulink. This block reads its input in fixed, predefined time steps and propagates this value unmodified until a new input value is read. By setting the time steps to the sampling interval of the hybrid port instance, we retain the semantics of MECHATRONICUML's hybrid port instances as defined in Section 3.1.1.5.

Finally, the chart block always has an inport clockSignal that is connected to a *digital clock* block. The digital clock provides the global simulation time and is automatically updated. We exploit the time values provided by the digital clock block for implementing the clock concept of MECHATRONICUML in Stateflow as we explain in Section 6.4.3.

6.3.1.2 Continuous Atomic Component Instances

For continuous atomic component instances, the enabled subsystem remains empty in our translation. This is consistent to the MECHATRONICUML component model that only defines the interface of continuous components but not their behavior (cf. Section 3.1.2.2). The resulting enabled subsystem will contain the implementation of the feedback controller which will be added in Step $S_{5.2}$ of our process shown in Figure 6.4.

6.3.1.3 Fading Component Instances

For fading component instances, we generate a block diagram that is embedded in the enabled subsystem. This block diagram implements the behavior of the fading component as defined in Section 3.1.2.3. Its structure depends on the number of fading functions that are contained in the fading component and on the number of inputs of each fading function. Figure 6.8 shows the generation template for generating the block diagram for a fading component instance.

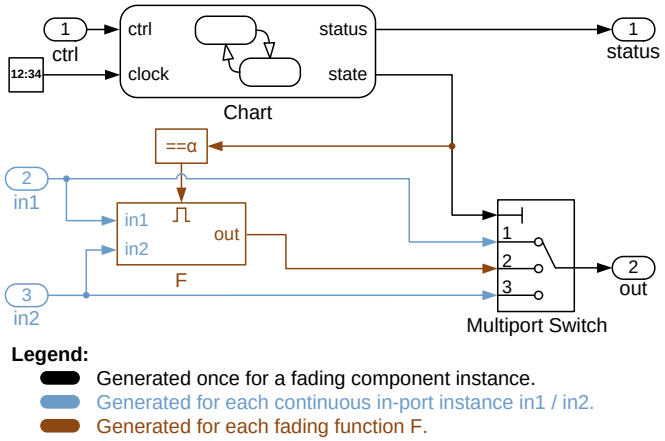


Figure 6.8: Generation Template for Internal Structure of a Subsystem for a Fading Component

The inports in1 and in2 represent the values that are provided by the continuous component instances that are connected to the fading component instance. In addition, we obtain one enabled subsystem for each fading function F that contains the implementation of the fading function. The contents for these subsystems will be added in Step $S_{5.2}$ of our process shown in Figure 6.4. The enabled subsystem F is enabled if the state of the chart block is equal to the integer constant α as defined by the compare block. The Multiport Switch at the right determines which of the input signals is propagated to the out outport. The Multiport Switch is also controlled by the state of the chart block.

In the following, we describe how the behavior of the fading component instance is realized by the block diagram and the Stateflow chart that is contained in the chart block. As an example, we use an instance of the fading component ConvoyFading shown in Figure 3.2c on Page 38.

Figure 6.9 shows the result of applying the generation template shown in Figure 6.8 to ConvoyFading. The block diagram contains two subsystems fadeToConvoy and fadeToStandalone that correspond to the eponymous fading functions of ConvoyFading.

Figure 6.10 shows the Stateflow chart that is embedded in the chart block of Figure 6.9. It receives the input signal ctrl and the current simulation time provided by the digital clock block. It outputs a signal status and a signal state. The former denotes whether the fading component is currently executing a fading function (1) or not (0). The latter is used for controlling the block diagram shown in Figure 6.9.

The execution of the Stateflow chart starts in the Static1 state. In this state, state is set to 1. As a result, both compare blocks evaluate to 0 and no fading function is executed. In addi-

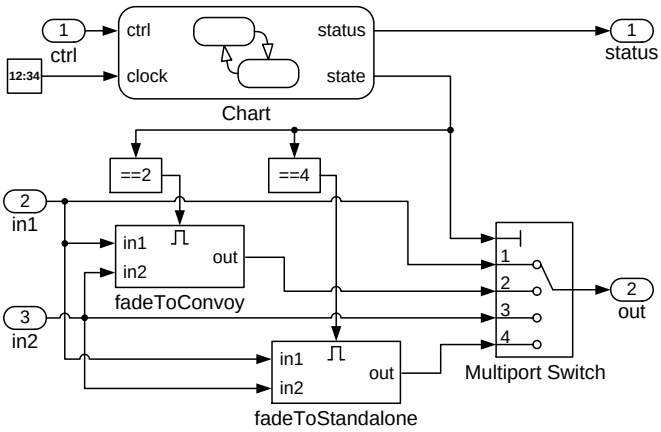


Figure 6.9: Example of a Simulink Model for a Fading Component [Vol13]

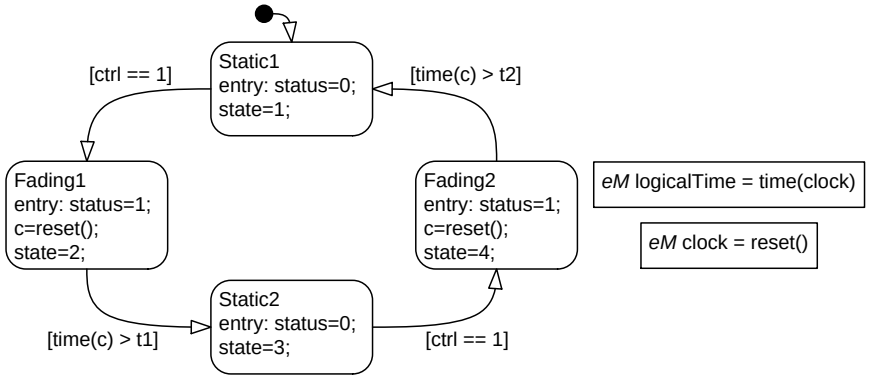


Figure 6.10: Example of a Stateflow Chart for a Fading Component [Vol13]

tion, state is forwarded as a control signal to the MultiportSwitch. There, a value i defines that the value of the i^{th} data input is forwarded. Thus, the fading component forwards the value received via in1. In state Fading1, state is set to 2. Consequently, the fading function fadeToConvoy is enabled and the MultiportSwitch forwards the output of the fading function. After the duration of the fading function t1 has elapsed, the Stateflow chart proceeds to state Static2. In this state, the value received via in2 is forwarded. The execution of the fadeToStandalone fading function in state Fading2 works analogously. As a consequence, the fading component either forwards one of its input values without modification or it executes a fading function which retains the semantics of fading components as defined in Section 3.1.2.3.

6.3.2 Translating Structured Component Instances

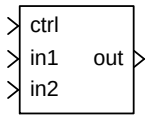
We create one enabled subsystem for each structured component instance that is contained in the CIC. The rules for creating the enabled subsystem and its external interface represented by the port instances are the same as for atomic component instances. As a result, we can apply the generation template shown in Figure 6.6 for structured component instances as well. However, the generation of the internal structure differs for structured component

instances. In particular, we need to translate its embedded CIC into a Simulink block diagram. The embedded CIC is translated in two steps. In the first step, we translate all embedded component instances by recursively applying the translation for atomic and structured component instances to them. In the second step, we translate all connector instances that connect the component instances in the embedded CIC. This requires, on the one hand, to translate connector instances between continuous and hybrid port instances as explained in Section 6.3.2.1. On the other hand, we need to translate all connector instances between discrete port instances as explained in Section 6.3.2.2. A complete example of a block diagram that results from translating a structured component instance is presented in Appendix A.8.2.

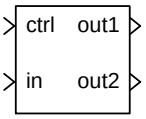
6.3.2.1 Continuous Connector Instances

Continuous connector instances propagate signal values and are, thus, equivalent to lines in Simulink. Therefore, we may directly translate continuous connector instances to lines in Simulink that connect the outport corresponding to the source port instance to the inport corresponding to the target port instance.

If we want to enable reconfiguration of continuous connector instances, we need to use two additional helper blocks named MultiSourceControl and MultiTargetControl [Vol13] shown in Figure 6.11. Both blocks are implemented using basic blocks of Simulink. We refer to Volk [Vol13] for a description of their implementation.



(a) MultiSourceControl Block



(b) MultiTargetControl Block

Figure 6.11: Helper Blocks that Enable Reconfiguration of Continuous Connector Instances in Simulink [Vol13]

The MultiSourceControl block in Figure 6.11a has one ctrl inport and one to many data inports whose names are prefixed by in. ctrl defines which of the data inports is propagated to out. If none of the data inports shall be propagated, it outputs 0. The MultiTargetControl block in Figure 6.11b has two inports ctrl and in. In addition, it has one to many data outports whose names are prefixed by out. For n outports, ctrl is an array of length n of type Boolean. If the n^{th} field of the array is true, then the value received via in is forwarded via the n^{th} data outport. This enables to specifically select an arbitrary number of receivers for the input value.

Figure 6.12 shows the generation template for using the MultiTargetControl block. We generate one MultiTargetControl block for each continuous or hybrid out-port Y of our MECHATRONICUML model that has an outgoing connector instance that is reconfigurable, i.e., there exist several in-ports Y1 that may be connected to Y at runtime. Then, we generate one outport at the MultiTargetControl including a line to every inport Y1. The generation template for MultiSourceControl blocks is defined analogously and, therefore, omitted. It is applied for each continuous or hybrid in-port Y of our MECHATRONICUML model that has an incoming connector instance that is reconfigurable.

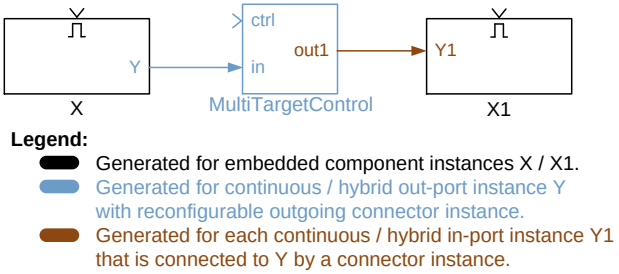


Figure 6.12: Generation Template for Translating Continuous Connector Instances

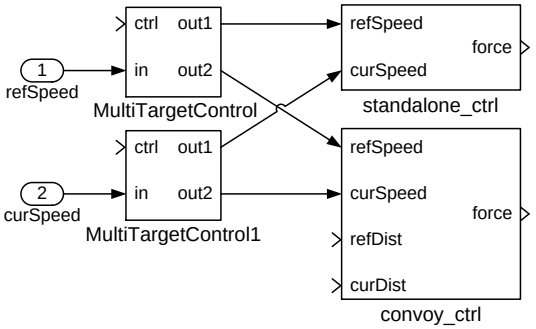


Figure 6.13: Example for Using a MultiTargetControl Block for Translating a Reconfigurable Delegation Connector

Figure 6.13 shows an example of using a MultiTargetControl block. The block diagram in Figure 6.13 results from translating the embedded CIC of an instance of VelocityController (cf. Figure 3.7) to Simulink. The inports refSpeed and curSpeed are either connected to their counterparts in standalone_ctrl or to their counterparts in convoy_ctrl or to both (cf. Section 3.3.2). Therefore, we connect both inports to MultiTargetControl while the out1 and out2 outports of the MultiTargetControl are connected to the embedded subsystems. The ctrl input is then a Boolean array of length 2. If the first array entry is true, the values are propagated to standalone_ctrl. If the second entry is true, the values are propagated to convoy_ctrl. If both entries are true, the values are propagated to both embedded subsystems.

6.3.2.2 Discrete Connector Instances

Discrete connector instances transport messages from the sending port instance to the receiving port instance. For the translation of a CIC into a Simulink subsystem, we need to distinguish between the assembly connector instances and the delegation connector instances that are used in the CIC. We present generation templates for their translation in the following.

Figure 6.14 shows the generation template for translating assembly connector instances based on a helper system called *communication switch*. The communication switch forwards messages from the sending port structure to the receiving port structure. We generate exactly one communication switch for each CIC. The communication switch is connected to a BusCreator and BusSelector. For discrete port instances of an embedded component instance, we generate one line from the outport of the corresponding port structure to the BusCreator

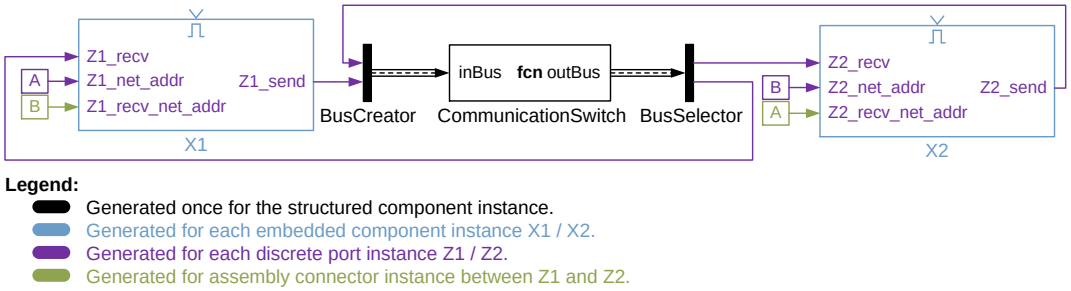


Figure 6.14: Generation Template for Translating Assembly Connector Instances

and one line from the BusSelector to the inport of the corresponding port structure. Then, we use the addresses of the port structures for translating the assembly connector instance. As an example, consider an assembly between Z1 and Z2 in Figure 6.14. The port structure Z1 has net_addr A, while the port structure Z2 has net_addr B. As a consequence, the recv_net_addr of Z1 is B while the recv_net_addr of Z2 is A. We explain the behavior of the communication switch in more detail in Section 6.3.3.3 and describe how the communication switch enables to reconfigure assembly connector instances in Section 6.5.

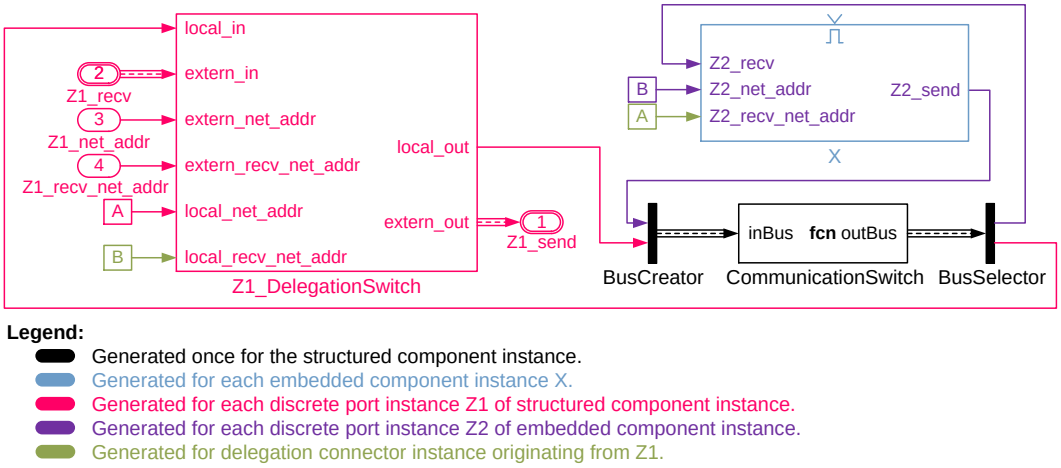


Figure 6.15: Generation Template for Translating Delegation Connector Instances

Figure 6.15 shows the generation template for translating delegation connector instances based on the communication switch and an additional helper system called *delegation switch*. The delegation switch is responsible for forwarding messages that have been received (or sent) by a port structure of the structured component subsystem to the receiving (or sending) port structure of an embedded subsystem. For each port instance Z1 of the structured component instance, we generate one port structure as introduced in Section 6.3.1 including a delegation switch. The inports and the outport of the port structure are connected to the delegation switch. The delegation switch, in turn, is connected to the communication switch in the same fashion as described above for an assembly connector instance. This enables to treat assembly connector instances and delegation connector instances identically in our

MATLAB-specific reconfiguration controller as we explain in Section 6.5. Then, we use the addresses of the port structures for translating the delegation connector instance. As an example, consider a delegation between Z1 and Z2 in Figure 6.15. The delegation switch has net_addr A, while the port structure Z2 has net_addr B. As a consequence, the recv_net_addr of the delegation switch is B while the recv_net_addr of Z2 is A. We explain the behavior of the delegation switch in more detail in Section 6.3.3.4.

6.3.3 Using Message-Based Communication

This section describes how we emulate message-based communication in Simulink. In particular, we explain the encoding of messages (Section 6.3.3.1) and the behavior of the helper blocks, namely of the link layer (Section 6.3.3.2), the communication switch (Section 6.3.3.3), and the delegation switch (Section 6.3.3.4). For a detailed description of the internal implementation of the helper blocks, we refer to our technical report [HRB+13] and the thesis of Volk [Vol13].

6.3.3.1 Encoding Messages

Since Simulink does not support asynchronous messages that are exchanged by discrete component instances in MECHATRONICUML, we need to define an encoding based on signals in Simulink. We encode messages as tuples consisting of six signals: a package ID, a message ID, a parameter value, a sender ID, a receiver ID, and a timestamp [HPR+12]. The message ID is a uniquely identifiable integer that represents a particular message type of the MECHATRONICUML model. We need to use an integer encoding because Simulink does not support strings. The parameter encodes a single parameter value. The sender ID is the address of the port structure sending the message, while the receiver ID denotes the address of the message receiver. The package ID is an incrementing integer ID that numbers the messages that are exchanged between two port structures. We utilize this field for detecting lost messages as we explain in Section 6.3.4. The timestamp refers to the point in time where the message was sent based on the simulation time. We use this value for simulating message delay and identifying messages that arrive too late. In Simulink, we implement the six-tuple for a message by means of a bus signal with six entries [HRB+14].

In MECHATRONICUML, a message may define more than one parameter. Since our encoding only considers one parameter for each message in Simulink, messages with more than one parameter need to be split into several messages [HRB+14]. As an example, consider the message update that is sent by the port instance refDistProvider of rg1 in Figure 3.10[3]. This message is split into two consecutive messages where the first one contains the value for the distance parameter and the second message contains the value for the speed parameter.

6.3.3.2 Link Layer

For each discrete port instance of an atomic component instance in MECHATRONICUML, we obtain a port structure and a link layer subsystem that is connected to the port structure as described in Section 6.3.1.1. The link layer serves as a middleware between the application-layer component behavior contained in the Stateflow chart and the network infrastructure.

[3] refDistProvider implements the role provider of the RTCP DistanceTransmission introduced in Section 2.4.1

Upon creation of the platform-specific model for the system, the link layer needs to be replaced by the middleware for the system that implements the interfaces to the networking hardware.

In the following, we describe the behavior of the link layer based on the generation template shown in Figure 6.7. If a message arrives via port_in at the link layer, the link layer reads the message from the bus signal and checks whether it is the intended receiver by comparing the receiver ID in the message with its own net_address. Then, the message is stored in the message buffer for incoming messages. The message buffer is implemented by two arrays; one for message IDs and one for parameter values. These arrays are then sent to the Stateflow chart using the signals write_event_queue_in and write_event_param_queue_in. Then, the Stateflow chart may consume messages from these signals as we explain in Section 6.4.2. The modified buffer is sent back to the link layer such that the link layer may keep track of the current buffer status. If the Stateflow chart sends a message, it adds the message to the z_WriteOut and z_ParamWriteOut queues that are defined analogously to the queues for received messages. Then, the link layer reads these queues and successively sends the messages via port_out.

The idea and concept for the link layer subsystems and the encoding of messages has been reused from Henke et al. [HTS+08b]. In their approach, each discrete component only used one such link layer subsystem that was shared by all of the port instances. As a result, the Stateflow chart needed to know the address of the receiving port structure. We reimplemented their concept such that we use one dedicated link layer for each discrete port instance of our MECHATRONICUML model, which corresponds to the semantics of MECHATRONICUML where each discrete port instance has its own message buffer. In addition, we extended the link layer such that it considers the QoS assumptions of MECHATRONICUML (cf. Section 6.3.4). In our approach, the Stateflow chart is independent of the net_address and receiver_new_address and, thus, of the concrete network infrastructure.

6.3.3.3 Communication Switch

The communication switch shown in the middle of Figure 6.14 is responsible for routing messages from the sender port structure to the receiver port structure. Thus, its serves as a virtual networking infrastructure [HPR+12]. Upon creation of the platform-specific model of the system, the communication switch is replaced by the networking hardware (or a simulation model of it).

The behavior of the communication switch is as follows: For each message in inBus, it reads the receiver ID and writes the message to the corresponding field in outbus that is connected to the receiving port structure. The communication switch learns automatically which field in outbus belongs to which receiver ID. First, the order of IDs in inBus and outBus are the same. That means, if a message is contained in inBus, the communication switch learns the ID of the port structure by reading the sender ID of the message. If a communication switch receives a message whose receiver ID is still unknown to it, it forwards the message to all unknown receivers. Then, the right receiver will answer with an acknowledgment and the communication switch may update its information. This behavior is inspired by the address resolution protocol (ARP, [Plu82]) that is used in Ethernet networks. It enables to use the same communication switch implementation for all subsystems that were created for structured component instances and the (integrated) system CIC.

6.3.3.4 Delegation Switch

The delegation switch shown in Figure 6.15 realizes a delegation connector instance between two discrete port instances by performing a network address translation (NAT, [SH99]). In our approach, the addresses of the ports are only unique within a subsystem that corresponds to a structured component instance, i.e., each structured component defines its own private subnet. This enables that we may change connector instances inside a structured component instance without exposing this change to another component instance thereby retaining component encapsulation. In particular, a component instance that is connected to a port instance of the structured component instance does not need to know about the change.

Therefore, the inports extern_net_addr and extern_recv_net_addr of the delegation switch receive the addresses that are used by the port structure for communicating with another subsystem *outside* its boundaries. The inports local_net_addr and local_recv_net_addr define the addresses for communicating *inside* the boundaries of the subsystem. If the delegation switch receives a message via extern_in, this message contains the external addresses of the port structure. Then, the delegation switch replaces these addresses with its local ones, i.e., the sender ID is set to local_net_addr while the receiver ID is set to local_recv_net_addr.

Consider the generation template shown in Figure 6.15 as an example. The port structure for Z2 in the embedded subsystem X has the address B, while the delegation switch has address A. Thus, for any message arriving at the delegation switch via extern_in, the sender ID is set to A while the receiver ID is set to B. Consequently, the port structure Z2 has the recv_net_addr A. Then, the message is sent via local_out to the communication switch that routes the messages to the receiver. Messages that are sent by embedded component instances are treated in the same fashion, however, the delegation switch replaces the local addresses with the external ones in this case.

6.3.4 Considering QoS Assumptions

In our translation, we support the QoS assumptions that are defined for MECHATRONICUML (cf. Section 2.4.3). We implemented all of these inside the link layer subsystem rather than the communication switch, as the communication switch is shared by all assembly connector instances and these may have different QoS assumptions. By implementing them in the link layer, we may separately configure QoS assumptions for each assembly.

The link layer implements FIFO buffers using the buffer size specified in the MECHATRONICUML model. Our message buffer implementation guarantees that received messages are never reordered. We exploit the package ID for this purpose. If the message buffer is full, we discard the incoming message, i.e., the message buffer does not change. If the link layer receives a message, it only inserts the message into the message buffer after the minimum propagation delay has passed. We compare the timestamp of the message with the current time for computing its current delay. If the message arrives after the maximum propagation delay, it is dropped and considered as lost. Then, this message loss needs to be handled by the component RTSC.

In addition, our implementation considers message loss with a given message loss percentage ϑ for simulating unreliable channels. Then, the link layer randomly drops an incoming message with probability ϑ. In addition, the link layer configures how a message loss is treated. First, we can ignore message loss similar to the user datagram protocol

(UDP, [Pos80]). Second, we can detect the lost messages and retransmit them as in the transmission control protocol (TCP, [IETF81]). If the link layer receives a message with package ID x, it sends an acknowledgment with x back to the sender. If the sender does not receive the acknowledgment within a timeout period, it sends the message again. The resending terminates if the message can no longer reach the receiver before the maximum propagation delay expires.

We refer to our technical report [HRB+14] for a more detailed description of the implementation of the QoS assumptions in the link layer.

6.4 Translating Real-Time Statecharts to Stateflow Charts

This section defines the translation of RTSCs to Stateflow charts and, thus, covers Step 7 of the algorithm shown in Figure 6.5. We reason that our translation preserves the semantics of RTSCs based on the informal description of the semantics of Stateflow provided in the online documentation [Mata].

In the course of this section, we illustrate most parts of the translation based on an example. In particular, we use an excerpt of the Stateflow chart shown in Figure 6.16 that is generated for the RTSC of the component instance rg1 of type RefGen (cf. Appendix A.5.1.5). The Stateflow chart shows the translation of the region refDistProvider that defines the behavior of the refDistProvider port instance[4]. We explain the figure in detail in the subsequent subsections and provide explicit generation templates for complex parts of the translation where no 1:1 correspondence between MECHATRONICUML model element and Stateflow model element exists. An extensive formalization of the transformation based on triple graph grammars (TGGs, [Sch95]) is given in our technical report [HRB+14].

In the following, we first introduce the basic concepts of the transformation (Section 6.4.1). Thereafter, we describe in more detail how message-based communication (Section 6.4.2), clocks (Section 6.4.3), urgency (Section 6.4.4), RTSCs of multi ports (Section 6.4.5), and synchronizations (Section 6.4.6) may be translated to Stateflow. In this section, we only provide a brief overview of the concepts of the translation. We refer to our technical reports [HRB+13, HRB+14] for a detailed description of the translation.

6.4.1 Basic Transformation Concepts

We reuse the basic parts of the transformation from RTSCs to Stateflow charts from a previous approach by Steinke [Ste07]. In particular, we translate states to states, transitions to transitions, and initial states to initial states. Furthermore, we encode parallel regions by parallel substates in Stateflow.

For the RTSC of the component RefGen, we obtain one state RefGen_Main that embeds five parallel states that correspond to the five embedded regions. Thus, we have one parallel state for each of the four port instances (refDistProvider, prev, next, and profileReceiver) and one for the internal behavior. In addition, the parallel state refDistProvider contains states Idle, SendUpdate, and AwaitAck that correspond to the eponymous states of the port RTSC. We omitted the internals of the remaining parallel states to improve readability of the figure.

[4] refDistProvider refines the subport behavior of the role provider shown in Figure 2.15 on Page 30

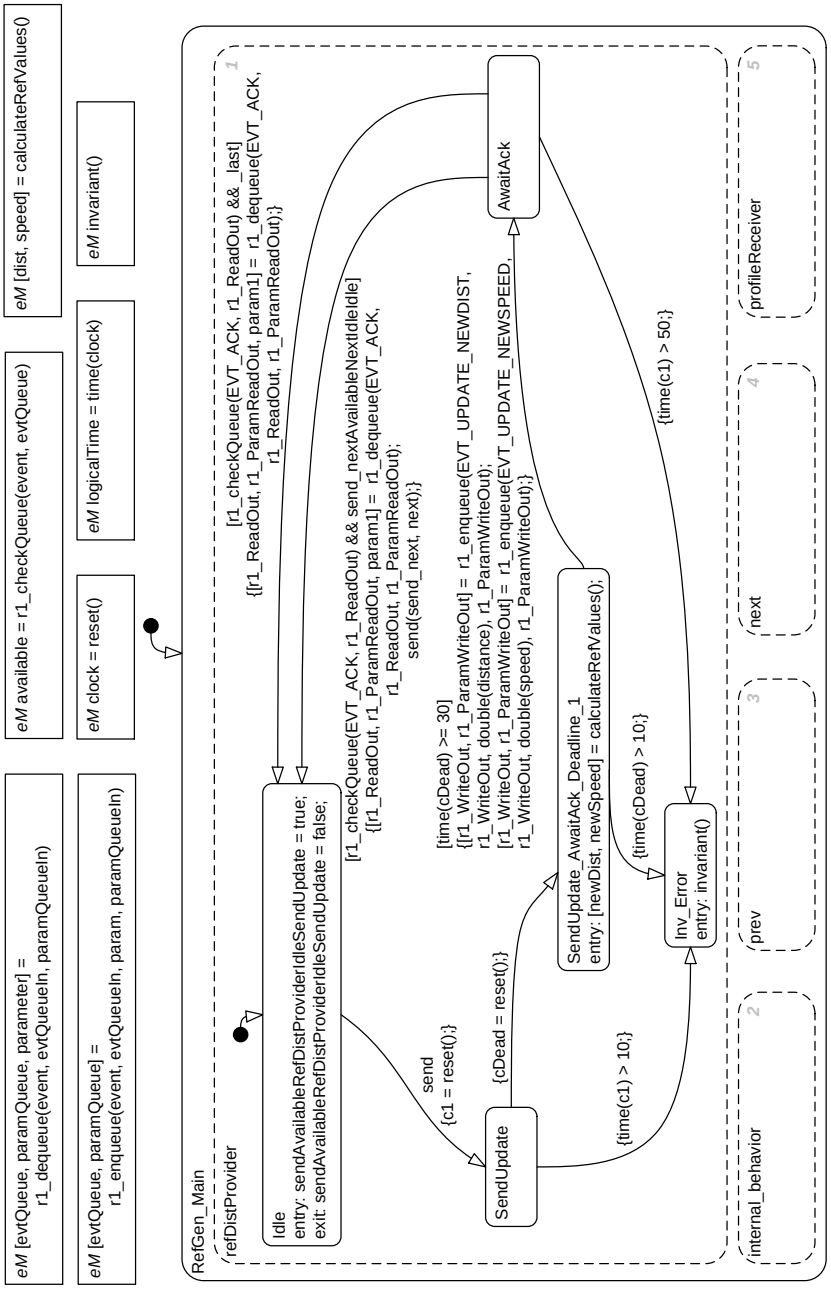


Figure 6.16: Stateflow Chart for the Subsystem rg1

For each transition in the RTSC, we obtain a corresponding transition in Stateflow as, e.g., for the transition from Idle to SendUpdate. An exception are transitions with deadlines such as from SendUpdate to AwaitAck as we explain in Section 6.4.3. In Stateflow, the transition specifies a guard in square brackets and an action in curly brackets comparable to MECHATRONICUML. We translate the guard conditions and transition actions that are specified using the action language of MECHATRONICUML to corresponding expressions in Stateflow.

Variables of RTSCs are translated to local data variables in Stateflow. We use the same name and data type as in the MECHATRONICUML model. Operations become embedded MATLAB functions of the same name. We translate the expressions contained in the body of the operation into an equivalent embedded MATLAB script. Entry- and exit-actions of states in MECHATRONICUML are translated to corresponding entry and exit behavior of Stateflow states.

6.4.2 Message-Based Communication

For realizing message-based communication in Stateflow, the Stateflow chart must operate on the message buffers that are provided by the link layer subsystems as described in Section 6.3.3. We reuse the approach by Tichy et al. [THB+10] that uses three embedded MATLAB functions named checkQueue, enqueue, and dequeue for operating on the message buffers as shown in Figure 6.17. However, we need to adjust the functions according to the changes in the link layers (cf. Section 6.3.3) such that the functions respect the FIFO property of our message buffers.

The function checkQueue checks whether a particular message is at the first position in our FIFO in-buffer. If so, it returns true, otherwise it returns false. The function dequeue removes the message that is located at the first position of the in-buffer and returns the parameter value contained in the message. Finally, enqueue adds a message including a parameter to the out-buffer. We generate these functions once for each port RTSC that is embedded in the component RTSC. The function names are prefixed with the name of the corresponding discrete port instance. In our example in Figure 6.16, we obtain the functions r1_checkQueue, r1_dequeue, and r1_enqueue for the port instance r1.

We use these three functions at the transitions of the Stateflow chart for processing messages that appear at the transitions of the port RTSC. As defined in our generation template in Figure 6.17, we use the function X_checkQueue in the transition guard for checking whether a received message is contained in the in-buffer. As a result, the transition is only enabled if the message is present, which corresponds to the semantics of RTSCs. The message is encoded by an integer constant EVT_Y1 where Y1 is the name of the message in MECHATRONICUML. If the transition fires, the transition action calls X_dequeue to consume the message. This call needs to be the first one in the action because the value of the message parameter needs to be available for the remainder of the transition action. The value of the message parameter is assigned to a variable param1. If the message had no parameter in MECHATRONICUML, the returned parameter value is 0 and the variable is not further used. Then, the message parameter may be used in the remaining action by accessing the corresponding variable. This is compliant to the semantics of MECHATRONICUML. If a transition of the port RTSC sends a message Y1, we invoke the X_enqueue function in the transition action. As parameters, we pass the integer constant EVT_Y2 and, optionally, the value of a message parameter. The calls of X_enqueue are placed behind the remaining statements of the transition action such that

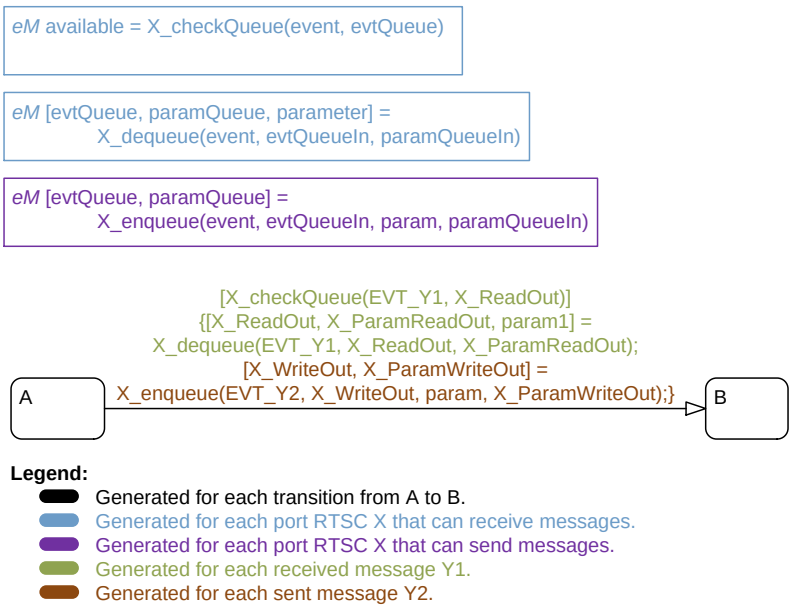


Figure 6.17: Generation Template for Translating Sent and Received Messages of Transitions to Stateflow

the message is sent after executing the transition action. This complies to the semantics of RTSCs. If the corresponding message in MECHATRONICUML has more than one parameter, then it is split into several messages as explained in Section 6.3.3.1. Then, we add one call to X_checkQueue, X_dequeue, and X_enqueue for each of the resulting messages.

In our example in Figure 6.16, consider the upper transition from AwaitAck to Idle. This transition needs to process the received message ack. Therefore, we invoke r1_checkQueue in the transition guard for checking whether the ack message, encoded by the integer constant EVT_ACK, is contained in the in-buffer. In the transition action, we invoke r1_dequeue for consuming the message. Since ack has no parameters, param1 is 0 in this case and is not further used in the transition action. In addition, consider the transition from SendUpdate_AwaitAck_Deadline_1 to AwaitAck. This transition needs to send the message update that has two parameters newDist and newSpeed. Thus, we invoke r1_enqueue twice in the transition action. The first invocation enqueues a message with the ID EVT_UPDATE_NEWDIST that contains the newDist parameter of update. As a value of the parameter, it passes the variable distance to the enqueue function. The second invocation enqueues a message with the parameter newSpeed in the same fashion.

The functions X_enqueue and X_dequeue always need to return the message buffers that they modified. These are then assigned by the transition action to the outports of the chart block due to the call-by-value semantics of Stateflow.

6.4.3 Clock Concept

We translate clocks of RTSCs to variables in Stateflow as proposed by Steinke [Ste07]. However, we decided to develop a new concept how these variables are used. In her approach, all clock variables are incremented by 1 each time the Stateflow chart is evaluated. As a

consequence, it is mandatory to execute the Stateflow chart with a fixed sample time of 1 ms and the chart needs to be aware of its own sample time. In contrast, our approach uses the clockSignal that is attached to a digital clock block in Simulink (cf. Figure A.95). This block provides the current simulation time in milliseconds (and may be connected to the system clock in a real system). Thereby, the RTSC becomes independent of its sample time. That, in turn, enables to change the sample time of the RTSC when creating a platform-specific model including, e.g., a task mapping and scheduling for the RTSC [BGS05, But05].

In our approach, clocks are reset by calling the function reset. It simply returns the current value of clockSignal that is assigned to the clock variable. Then, the current value of the clock is obtained by calling time which returns the difference between clock signal and the value of the clock variable. As in MECHATRONICUML, the result is the time that has passed since the last reset. Time guards are then translated to normal guards in Stateflow using the function time. Then, the transition may only fire if the time guard is fulfilled, which retains the semantics of RTSCs. The values of the time constraints in MECHATRONICUML are converted to ms.

Stateflow provides no concept that is comparable to invariants of states in MECHATRONICUML. However, a violation of an invariant indicates an error in the model. In accordance to Steinke [Ste07], we add an error location, called Inv_Error in Figure 6.16, to the chart that models a violation of an invariant. For each state that has an invariant, we create a transition from that state to Inv_Error as, e.g., for the state SendUpdate in Figure 6.16. The transition contains a guard that corresponds to the negated time constraint of the invariant, i.e., whenever the invariant is *not* fulfilled, the transition to the error location fires. This is compliant to the semantics of RTSCs.

In Stateflow, transitions fire in zero time as in timed automata. Therefore, we map transitions with deadlines to two transitions with an intermediate state as shown in Figure 6.18 as it has been defined for mapping RTSCs to timed automata [GB03, Ger13]. For each transition from A to B that has a deadline, we generate an intermediate state A_B_Deadline including transitions from A to A_B_Deadline and from A_B_Deadline to B. The transition from A to A_B_Deadline specifies the precondition of the original transition resets an additional clock called cDead. The transition from A_B_Deadline to B specifies the original transition's effect including a guard that specifies that cDead is greater or equal to the lower bound α of the deadline. Furthermore, we need to add a transition from A_B_Deadline to Inv_Error for resolving the invariant of A_B_Deadline that results from the transformation and described by Giese and Burmester [GB03]. In particular, this transition will fire if cDead is greater than the upper bound β of the deadline.

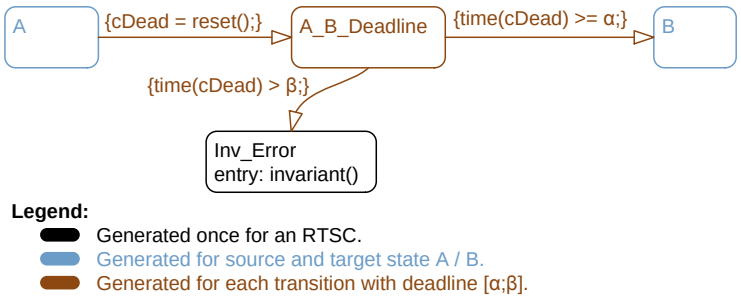


Figure 6.18: Generation Template for Translating Transitions with Deadline to Stateflow

In our example in Figure 6.16, the state SendUpdate_AwaitAck_Deadline_1 including the transitions from SendUpdate and to AwaitAck resulted from mapping a transition with deadline based on the generation template in Figure 6.18.

6.4.4 Urgency

In each simulation step, Stateflow checks the active states in a chart for enabled outgoing transitions. If a transition is enabled, it is immediately fired. This corresponds to the semantics of urgent transitions in RTSCs (cf. Section 2.4.2). Non-urgent transitions are not supported by Stateflow and can, thus, not be preserved by our translation. Due to Stateflow's semantics, we restrict the time intervals where non-urgent transitions may fire. Thus, our translation preserves the behavior according to a timed ready simulation (cf. Section 5.2). As a consequence, all verified ATCTL properties still hold for the Stateflow chart and we retain all urgent transitions. Properties that contain existential quantifiers need to be rechecked in Stateflow using test cases that need to be defined in Step $S_{5.3}$ of our process in Figure 6.4.

If a state is urgent in the RTSC, then the corresponding Stateflow chart may not rest in this state and wait for the next simulation step. Therefore, we translate urgent states to connective junctions in Stateflow. This retains the semantics of MECHATRONICUML because no time passes in Stateflow until the connective junction is left.

6.4.5 Real-Time Statecharts of Multi Port Instances

In MECHATRONICUML, a multi port instance contains a set of subport instances where each subport instance executes the behavior defined by the subport RTSC (cf. Section 2.4). As an example, consider the multi port instance of type coordinator shown in Figure 6.19 that is implemented by the component instance cm of type ConvoyManagement (cf. Figure 3.5). It contains two subport instances and, thus, the RTSC for this multi port instance executes two copies of the subrole RTSC.

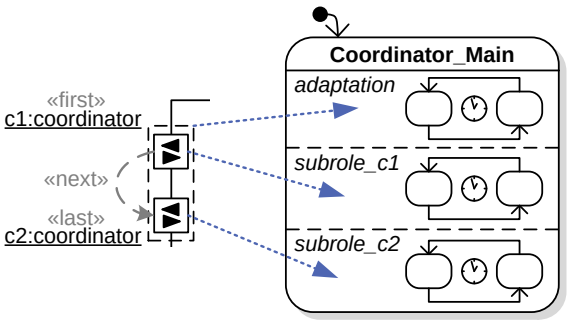


Figure 6.19: Multi Port Instance with Resulting RTSC

As a consequence, the Stateflow chart needs to contain corresponding subport charts for all subport instances. That means, we need to replicate the subport RTSC for each subport instance. This replication is performed before translating the RTSC to Stateflow.

6.4.6 Synchronizations

We translate synchronizations used at transitions of a RTSC to signal events in Stateflow as proposed by Steinke [Ste07]. However, we significantly extended the transformation because the concept by Steinke does not retain the semantics of MECHATRONICUML. In particular, Steinke's concept did not prevent that the transition that initiates the synchronization fires even though the receiving transition cannot fire. We introduce our new concept for translating plain synchronizations in Section 6.4.6.1. In addition, we extend the concept such that it supports selector expressions of synchronizations in Section 6.4.6.2.

6.4.6.1 Plain Synchronizations

For each synchronization channel, we create a signal event with the channel's name in Stateflow. Synchronizations that may appear at the transitions of an RTSC are translated to Stateflow based on the generation template shown in Figure 6.20. Basically, the receiving transition waits for the signal event sync while the initiating transition receives a send operation that sends the signal event sync to RegB. Sending the signal event needs to be the very last statement in the transition action. This is because upon sending, Stateflow immediately switches to the receiving transition and completely executes it before finishing to execute the sending transition. If sending the signal event is the last statement, we guarantee that the transition action of the sending transition is completely executed prior to executing the transition action of the receiving transition. This preserves the semantics of synchronizations in RTSCs.

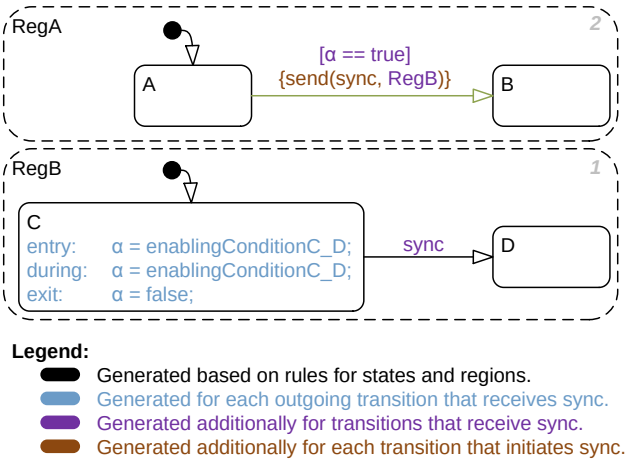


Figure 6.20: Generation Template for Translating Transitions with Plain Synchronizations to Stateflow

In our example in Figure 6.16, the transition from Idle to SendUpdate receives the signal event send that corresponds to the synchronization send in Figure A.38 on Page 233. The lower transition from AwaitAck to Idle sends the signal event send_next to the parallel state next.

In contrast to RTSCs, the transition sending a signal event does not block in Stateflow if there is no receiving transition that may fire. Therefore, we need to block the sending transition using an additional transition guard until a receiving transition becomes enabled.

For each transition that may receive a synchronization in the RTSC, we create one Boolean variable α in Stateflow that encodes whether the transition may fire as shown in Figure 6.20. Then, we add a transition guard to the sending transition that checks whether α is true. Thus, the sending transition is blocked until there exists a receiving transition that may fire. For plain synchronizations, the value of α corresponds to the enabling condition of the receiving transition, i.e., it is the conjunction of the following conditions: the source state is active, the transition guard of the transition is fulfilled, the time guard is fulfilled, and the trigger message is available in the message buffer. We assign the conjunction of these conditions to α in the entry action of the source state of the receiving transition and periodically update α in the during action. If the source state of the receiving transition is left, we set α to false in the exit action. If there exists more than one receiving transition, we need to replicate the transition from A to B for each region that contains a possible receiving transition using the α associated to this receiving transition in the transition guard. Please refer to our technical report [HRB+14] for a detailed discussion of this translation step.

In our example in Figure 6.16, the state Idle contains a variable α_1 = sendAvailableRefDistProviderIdleSendUpdate. Since the transition from Idle to SendUpdate in Figure A.38 only specifies the synchronization in its enabling condition, we only assign true to indicate that the outgoing transition to SendUpdate may fire. We may omit the during action in this case because the value of α_1 may never change. In addition, consider the lower transition from AwaitAck to Idle. This transition sends the signal event send_next to the hierarchical state next. Thus, it uses the variable α_2 = send_nextAvailableNextIdleIdle in its transition guard that defines whether the receiving transition inside the next state may fire.

6.4.6.2 Synchronization with Selector

RTSCs support two types of selectors that we need to handle in our translation. These are *integer* and *port* while the latter only applies to multi port RTSCs (cf. Section 2.4). These impose an additional condition on whether two transitions may synchronize. The basic transformation of synchronizations with selector is identical to plain synchronizations, but we need to generate five additional constructs [HRB+14] when using selectors as shown in Figure 6.21. The additional constructs are generated as follows:

1. We need to generate a variable β that encodes the selector expression of the receiving transition.
2. We add an additional entry action to the source state of the receiving transition that assigns the value of the selector expression to β. As for α, we need to update β periodically in the during action of the source state.
3. We extend the guard condition of the sending transition by a conjunction of α with a comparison of β to the selector expression of the sending transition. As a result, the initiating transition may only fire if there is a receiving transition that is enabled and that has the same selector expression.
4. We add one additional variable γ for each synchronization channel with selector. Just before sending the signal event, the sending transition assigns the value of its own selector expression to γ.

5. We need to add a guard condition to each receiving transition that checks if β is equal to γ. This check is necessary to ensure that only the transition with the right selector expression may fire.

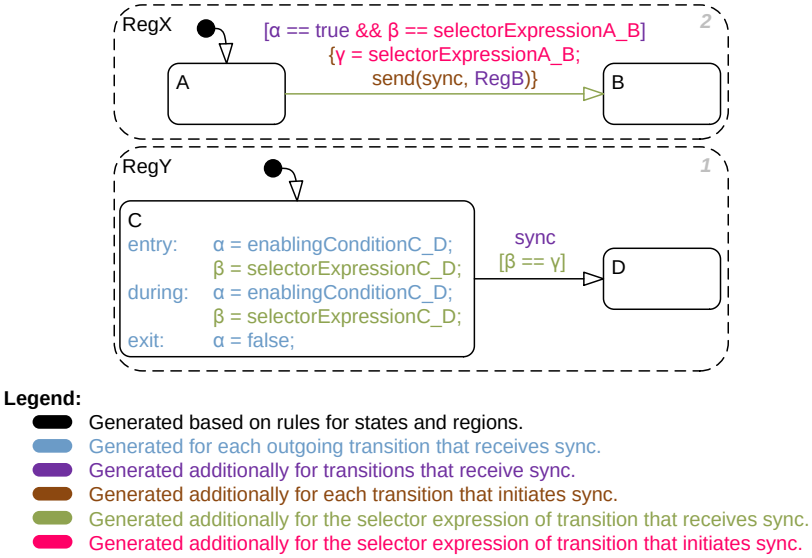


Figure 6.21: Generation Template for Translating Transitions with Synchronizations with Selectors to Stateflow

We illustrate these constructs in more detail using the example shown in Figure 6.22. The example shows a small excerpt of the Stateflow chart that is obtained by translating the component RTSC of Switch (cf. Figure 5.3c) that is shown in Figure A.40 to Stateflow. In particular, we consider the transitions from WaitForTrack to CheckRequest in the region left that receive the synchronization sectionFree and the transition from Notify to Idle in the region followingSection that initiates the synchronization sectionFree.

Since sectionFree uses a selector of type int, we need to apply the generation template shown in Figure 6.21 to the aforementioned transitions. As a result, we obtain two variables α_1 = syncAvailableLeftWaitForTrackCheckRequest1 and α_2 = syncAvailableLeftWaitForTrackCheckRequest2. α_1 is associated to the left transition from WaitForTrack to CheckRequest while α_2 is associated to the right transition from WaitForTrack to CheckRequest. Since both transitions only specify the synchronization in their enabling conditions, we set both variables to true in the entry action of WaitForTrack. Since the values of α_1 and α_2 cannot change, we omit the during action. In addition, we obtain two variables β_1 = selCondLeft_WaitForTrack_CheckRequest1 and β_2 = selCondLeft_WaitForTrack_CheckRequest2 (Construct 1). We assign the selector expressions of the corresponding transitions to β_1 and β_2 in the entry action of WaitForTrack (Construct 2). Thus, we assign 0 to β_1 and 1 to β_2. Again, we may omit the during action because β_1 and β_2 have constant values in this example.

The transition guard of the transition from Notify to Idle compares the values of β_1 and β_2 to the variable status that is used as a selector expression by this transition (Construct 3). Thus, the transition may only fire if the value of status equals either β_1 or β_2. In addition, we generate one variable γ = sendSelCond_sectionFree for the synchronization channel sectionFree.

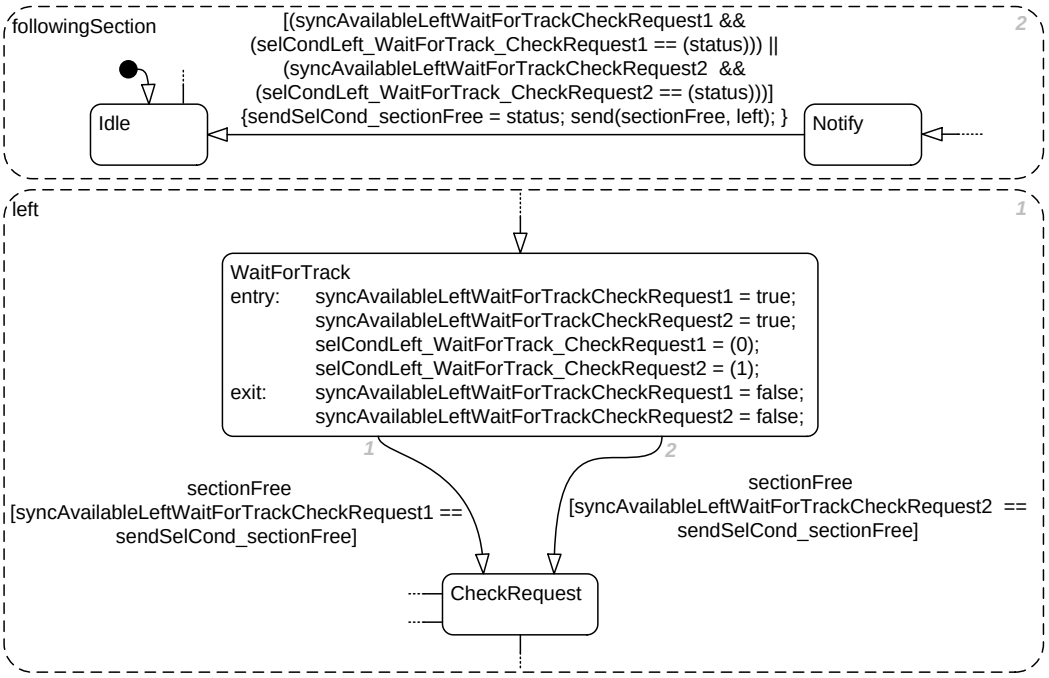


Figure 6.22: Example of Using Transitions with Synchronizations with Selectors in Stateflow

Then, we assign the value of status to γ in the transition action of the transition from Notify to Idle (Construct 4).

Finally, we add a guard condition to each transition from WaitForTrack to CheckRequest (Construct 5). Considering the left transition, this guard compares β_1 (= selCondLeft_WaitForTrack_CheckRequest1) and γ (= sendSelCond_sectionFree). If we omitted this guard condition, the transition from Notify to Idle could synchronize with either of the transitions from WaitForTrack to CheckRequest in Stateflow irrespective the value of the selector expression. By using the additional guard, we retain MECHATRONICUML's semantics of synchronization channels with selectors in Stateflow.

Please note that the transition from Notify to Idle needs to be replicated for each region that contains a transition that may receive sectionFree as described in Section 6.4.6.1. As a result, we obtain three transitions from Notify to Idle in Stateflow that send sectionFree to the hierarchical states resulting from the regions left, right, and bottom of the component RTSC, respectively.

Selectors of type port refer to the order of the multi port instance. As an example, consider the multi port instance of type coordinator shown in Figure 6.19. The order defines that c1 is the first subport instance while c2 is the last subport instance. In addition, c2 is the direct successor of c1. In the RTSC of the multi port, we may refer to the order using the keywords first, last, next, prev, and self as defined in Section 2.4. In Stateflow, we need to encode this order explicitly into the chart using integers. The reason is that Stateflow does not enable to define such order based on the resulting states. The resulting encoding is inspired by the representation of multi port instances as proposed by Hirsch [Hir08].

We generate one variable η for each subport RTSC that encodes the position of the subport. We start numbering the subport instances with 1. In our example, we obtain variables

η_1 = subport_c1_pos and η_2 = subport_c2_pos. The values of these variables encode the order, i.e., subport_c1_pos = 1 and subport_c2_pos = 2. Then, we replace each occurrence of self by η for each subport RTSC. In addition, we may replace next by $\eta + 1$ and prev by $\eta - 1$. Next, we replace first by 1 because the first subport instance always has position 1. Finally, we need to generate an additional variable numOfSubports that denotes the currently instantiated number of subport instances. Then, last may be replaced by numOfSubports. Thereby, we yield an integer encoding of the selector expressions and we may translate them to Stateflow using the rules for selectors of type integer as defined above.

6.5 Translating Reconfiguration Specifications to Simulink and Stateflow

This section describes how we translate the reconfiguration controller and the CSDs, which define reconfiguration behavior in MECHATRONICUML, to Simulink. Thus, this section covers Steps 1 to 5 of the algorithm shown in Figure 6.5. In the following, we introduce these steps in more detail in the order given by the algorithm. In addition, we describe how the MATLAB-specific reconfiguration controller may be represented in Simulink and how reconfiguration of port instances may be realized in Stateflow as part of Steps 6 and 7 of our algorithm.

6.5.1 Step 1: Compute Possible Configurations

In Step 1 of the algorithm, we compute the possible configurations for each (reconfigurable) component that is used in the MECHATRONICUML model. We compute these configurations by applying a reachability analysis [HSE10] using our framework (cf. Appendix C). The inputs for reachability analysis are (1) the initial configurations of the component as defined by its constructors (cf. Section 3.3.1) and (2) all CSDs defining the possible reconfigurations. The result of the reachability analysis is a reachability graph where each node corresponds to a possible configuration and where each transition corresponds to the application of a CSD.

Figure 6.23 shows an excerpt of the reachability graph for the structured component ConvoyCoordination shown in Figure 3.5 on Page 42. The states of the reachability graph are represented by rounded rectangles. In our example, we have two states config1 and config2 and three transitions. config1 is the initial state of the reachability graph. We mark the initial states analogously to RTSCs by using a filled black circle with a transition to the initial state.

Each state contains a CIC of ConvoyCoordination. config1 contains the initial configuration that is created by the constructor instantiate1Member shown in Figure A.55. We mark the initial state with the constructor that was used for creating it. config2 contains the configuration for two convoy members that results from applying the CSD addConvoyMemberAtPos to the configuration contained in config1. We label the transitions between configurations with the CSD that was applied. By repeatedly applying addConvoyMemberAtPos, we may generate additional configurations, each adding one additional member to the convoy. For the remainder of this section, we restrict ourselves to config1 and config2 for sake of simplicity.

Computing the possible configurations of a component requires that the number of configurations is finite. Otherwise, the reachability analysis will not terminate. In our example, ConvoyCoordination has an infinite number of configurations because RefGen and the coordi-

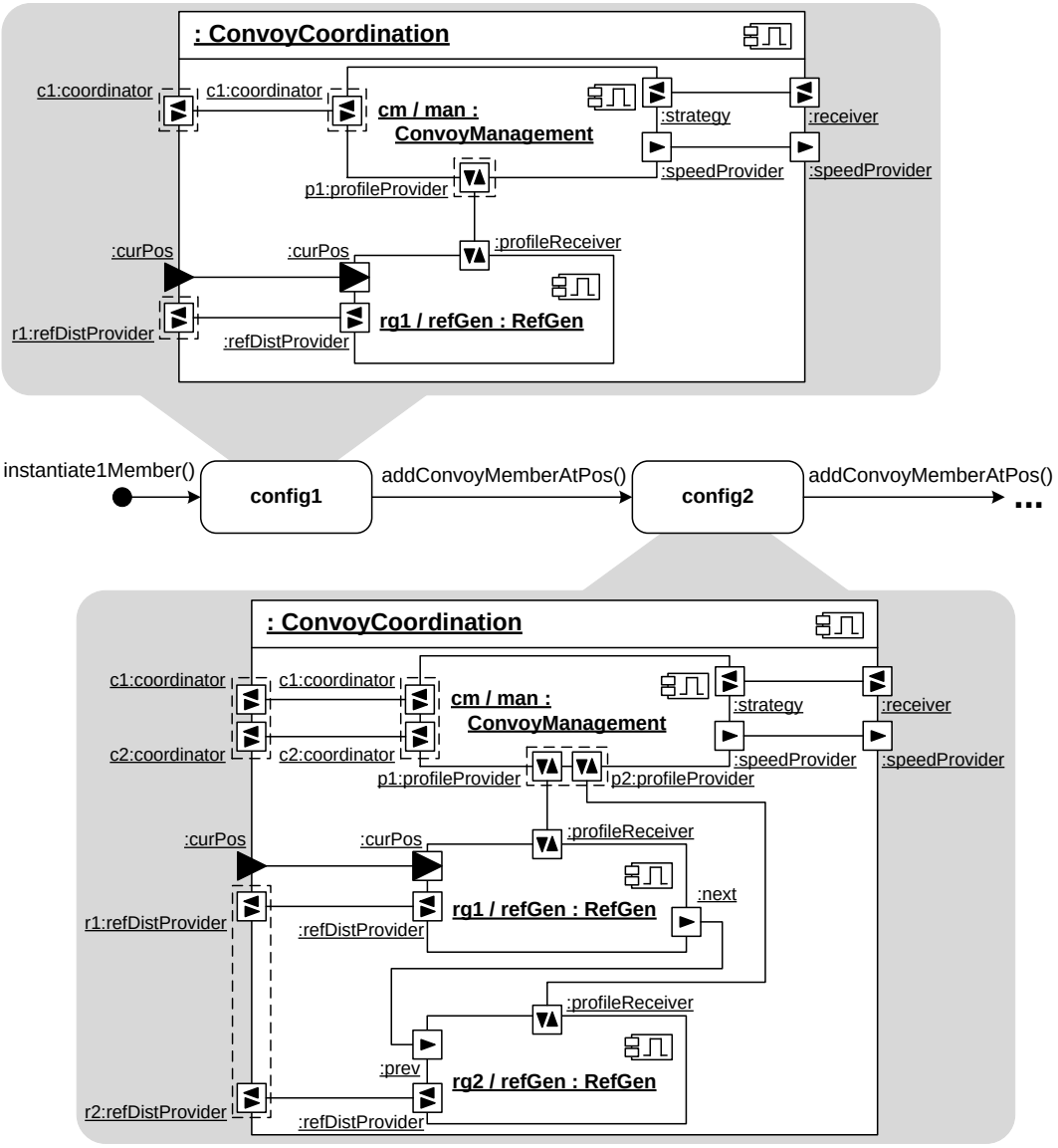


Figure 6.23: Excerpt of the Reachability Graph for the Component ConvoyCoordination

nator multi port may be instantiated arbitrarily often (cf. Figure 3.5). We have two options for ensuring that the reachability analysis terminates. First, we may restrict cardinalities and, second, we may use a depth limitation that prunes branches of the reachability graph if they reach the depth limit. In our example, it is sufficient to provide a finite upper bound either for the cardinality of the RefGen multi part or of coordinator multi port because both cardinalities imply each other. This is because the CSD shown in Figure 3.14 creates both in the same component story pattern.

6.5.2 Step 2: Create Integrated CIC for Component

The integrated CIC is the superposition of all configurations that appear in the states of its reachability graph. Thus, it contains the least set of port instances, embedded component instances, and connector instances that encodes all possibles configurations of instances of the component. The integrated CIC is the basis for generating the MATLAB-specific reconfiguration controller in Step 3 (cf. Section 6.5.3) and for creating the integrated system CIC in Step 5 (cf. Section 6.5.5).

Continuing our example from Figure 6.23, we compute the integrated CIC of ConvoyCoordination. Since the CSD addConvoyMember only adds component instances, port instances, and connector instances, the integrated CIC is equivalent to the configuration contained in config2.

6.5.3 Step 3: Generate the MATLAB-specific Reconfiguration Controller

The MECHATRONICUML reconfiguration controller as introduced in Section 4.1 operates on an implicitly defined model@runtime that is shared between manager, executor, and runtime risk manager. For the translation to Simulink, we need to encode the model@runtime manually by enumerating and encoding all configurations of a component instance based on the reachability graph and the integrated CIC. Therefore, we extend the reconfiguration controller by a *configuration store* that encodes the model@runtime and enables its modification. The configuration store is then integrated with the manager and executor. This is necessary to enable that the manager reads the current configuration and to enable that the executor may read and write the current configuration. The runtime risk manager is not yet considered in our simulation approach but needs to be connected to the configuration store as well for being able to read the current configuration. The resulting MATLAB-specific reconfiguration controller is shown in Figure 6.24.

Although we generate the MATLAB-specific reconfiguration controller on the level of MECHATRONICUML, we only generate it for the translation to Simulink. It restricts the capabilities of the reconfiguration controller as introduced in Chapter 4 to a fixed and finite number of configurations and also the possibility to switch between them. If the reachability graph of the component has been finite in the first place, we do not restrict the reconfiguration capabilities of the component. If the reachability graph is, in principle, infinite as for ConvoyCoordination (cf. Section 6.5.1), then restrict we restrict the reconfiguration capabilities but not significantly. This is because the reachability analysis that we use in Step 1 identifies configurations that are isomorphic [HSE10, Ren07], i.e., where the same embedded component instances, port instances, and connector instances are instantiated. Thus, we only

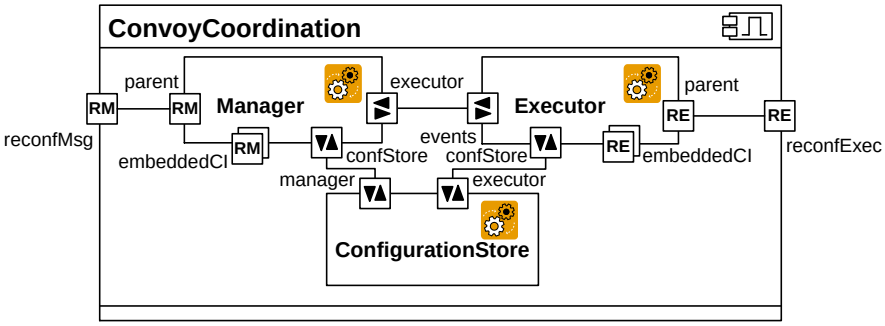


Figure 6.24: Integration of the ConfigurationStore in the MATLAB-specific Reconfiguration Controller (cf. [Vol13])

remove configurations where multi ports and multi parts have unbounded cardinalities, e.g., for interacting with an arbitrary number of other systems. In our RailCab example, we only restrict the maximum number of convoy members at runtime, which is a realistic restriction.

Since the configuration store explicitly encodes the model@runtime, manager and executor may no longer directly access the model@runtime but need to communicate with the configuration store. Therefore, we connect the configuration store to manager and executor using ports and assembly connectors as shown in Figure 6.24. In addition, we generate an RTSC for the configuration store that enables the interaction with the manager and the executor and that encodes the model@runtime. We introduce a generation template for this RTSC in Section 6.5.3.1. In addition, we need to adapt the generation templates of the manager and the executor (cf. Section 4.4) such that they may interact with the configuration store.

6.5.3.1 Behavior Specification of the Configuration Store

Figure 6.25 shows the generation template for generating the RTSC of the configuration store of a structured component based on the reachability graph. It consists of two regions named executor and manager. The former encodes the model@runtime and implements the communication with the executor while the latter implements the communication with the manager. Figure 6.26 shows the executor region that has been generated for the component ConvoyCoordination based on the reachability graph shown in Figure 6.23.

The RTSC in the executor region always contains one initial state named Initial. The remainder of the RTSC is generated based on the reachability graph. In essence, the RTSC encodes the reachability graph. For each state of the reachability graph, we generate one blue state named StateX that has a unique ID with a corresponding operation establishConfigX. The ID uniquely identifies the configuration that is contained in the corresponding state of the reachability graph. In its entry action, the ID is assigned to the variable config. In addition, the entry action calls the operation establishConfigX. This operation establishes the configuration X in Simulink using the control signals computed in Step 4 of our algorithm (cf. Section 6.5.4). In Figure 6.26, the states State_Config1 and State_Config2 have been generated based on the states config1 and config2 of the reachability graph in Figure 6.23. For each initial configuration of the component, we generate one green transition from Initial to the state corresponding to the particular initial configuration.

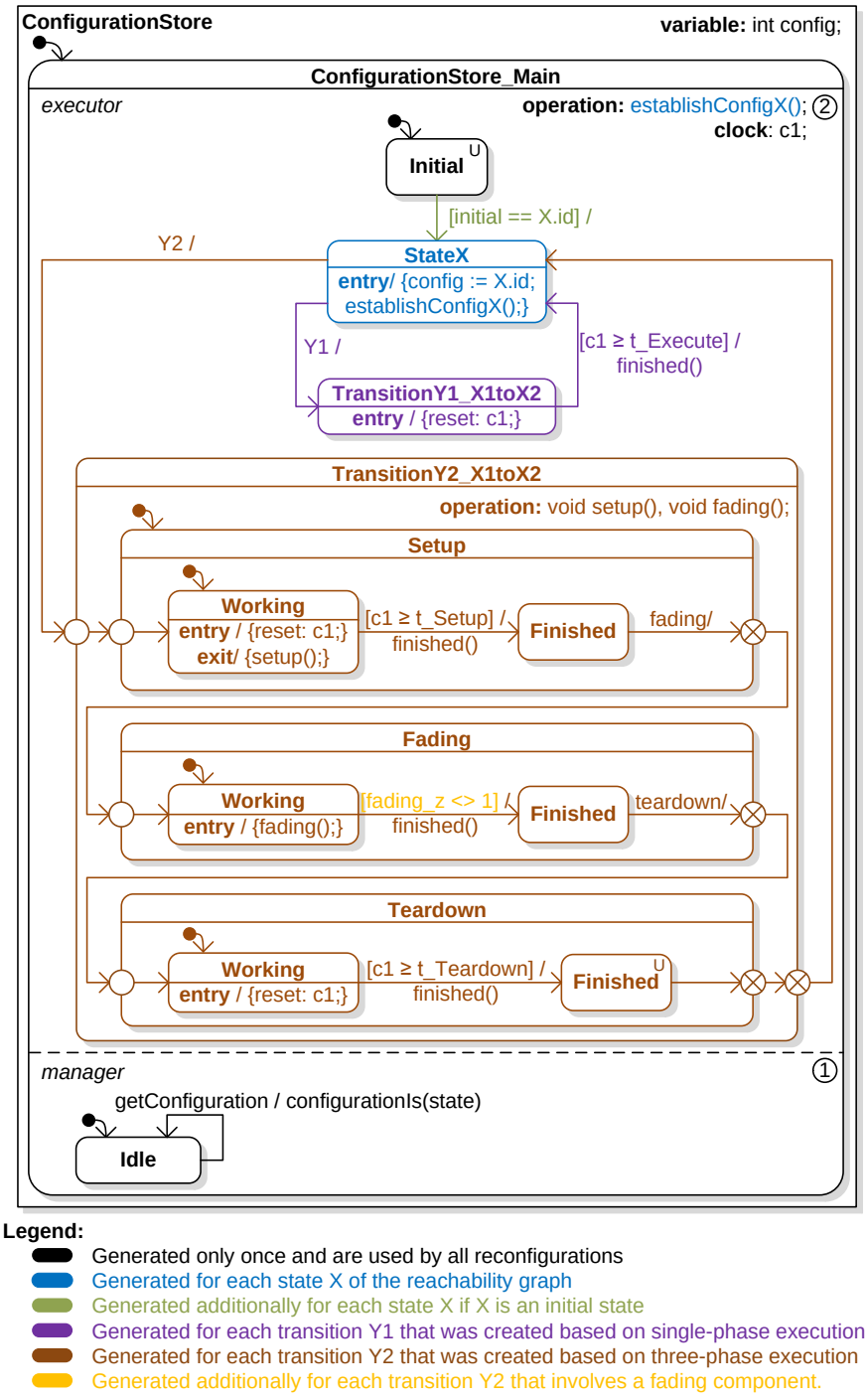


Figure 6.25: Generation Template for the Configuration Store RTSC (cf. [Vol13])

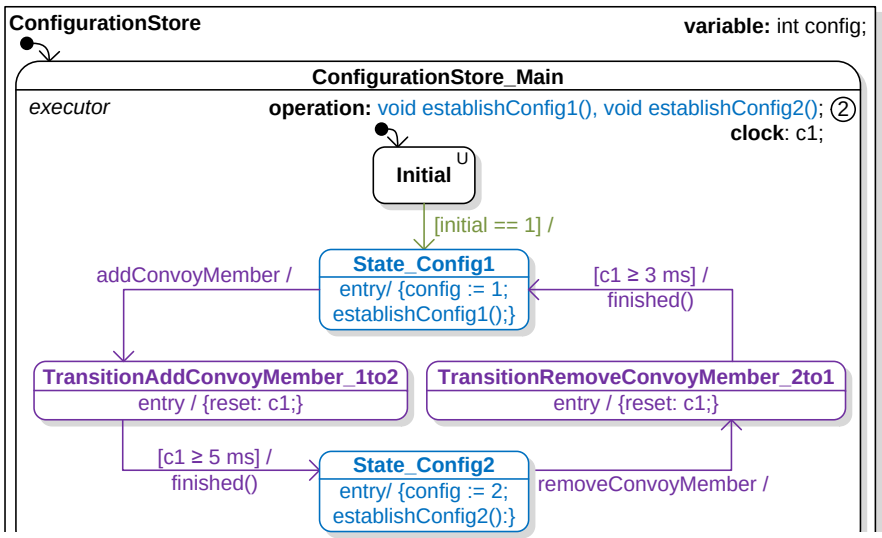


Figure 6.26: Example of the executor Region of the Configuration Store RTSC

Finally, we generate the purple and brown parts of the configuration store RTSC based on the transitions of the reachability graph. If the transition of the reachability graph corresponds to a reconfiguration that is executed according to single-phase execution (cf. Section 4.2.1), we generate a purple state including the adjacent transitions. If the reconfiguration is executed according to three-phase execution (cf. Section 4.2.2), we generate the hierarchical brown state including the adjacent transitions. In Figure 6.26, we only obtain purple states and transitions because the reconfigurations addConvoyMember and removeConvoyMember are executed based on single-phase execution. In both cases, the first transition leaving StateX is triggered by a reconfiguration message that is sent by the executor.

If the reconfiguration is executed based on single-phase execution, the reconfiguration message triggers a transition from StateX to state TransitionY1_X1toX2. In Figure 6.26, the reconfiguration message addConvoyMember triggers the transition from State_Config1 to TransitionAddConvoyMember_1to2. The state TransitionAddConvoyMember_1to2 is active as long as the reconfiguration is executed. Since Simulink and Stateflow do not consider that actions take time, we need to introduce this intermediate state. The entry action resets a clock c1, the outgoing transition is activated after the WCET of the CSD has passed. Then, the transition to the state representing the target configuration (State_Config2 in Figure 6.26) fires. Upon entering the target state, the result of the reconfiguration is established and, thus, becomes visible after the WCET has passed, which emulates the behavior of the real system.

If the reconfiguration is executed based on three-phase execution, the reconfiguration message triggers a transition from StateX to the hierarchical state TransitionY2_X1toX2. The hierarchical state contains three states that correspond to the three phases setup, fading, and teardown of the three-phase execution. In addition, it contains two operations setup and fading. setup establishes the intermediate configuration that results from executing the setup phase (cf. Section 4.2.2.1). fading controls the execution of the fading function using the fading component in Simulink (cf. Section 6.3.1.3). The RTSC waits in Working until the execution of the fading function has been finished. This is indicated by the value of fading_z

that is defined by a hybrid port that is connected to the fading component. The result of the teardown phase, which is the configuration resulting from executing the reconfiguration, is established when entering the state corresponding to this configuration.

The RTSC in the manager region is trivial. It only consists of one state Idle with a self-transition. The transition consumes a message getConfiguration and answers with a message configurationIs that contains the ID of the current configuration as a parameter.

6.5.3.2 Adaptated Behavior Specification of the Manager

The manager RTSC needs to access the model@runtime for evaluating the structural condition (cf. Section 4.3.2) that is specified by a component SDD. Since the model@runtime is now contained in the configuration store, the corresponding operation checkStructuralConditionForX for a reconfiguration X in the manager RTSC generation template can no longer be implemented by accessing the model@runtime. Instead, the manager needs to query the current configuration from the configuration store. Figure 6.27 illustrates how the manager RTSC (cf. Figure 4.15) needs to be adapted for translating it into a Stateflow chart.

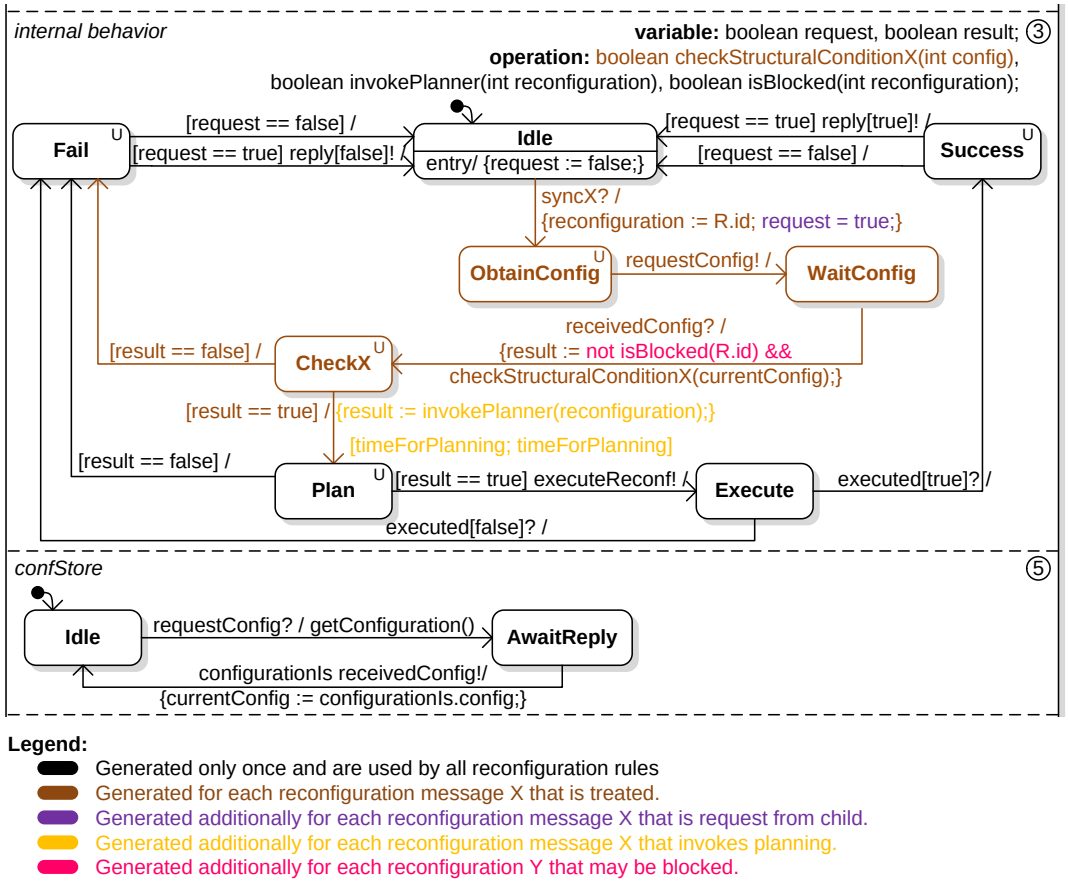


Figure 6.27: Adapted Generation Template for the Manager RTSC (cf. [Vol13])

First, we introduce a region confStore for communicating with the configuration store. The RTSC in this region sends the message getConfiguration when switching to AwaitReply and

receives the message configurationIs at the transition back to Idle. The received ID of the configuration is stored in the variable currentConfig.

Second, we need to replace the transition from Idle to CheckX in the internal behavior by three transitions with two intermediate states named ObtainConfig and WaitConfig. The transition from Idle to ObtainConfig receives the synchronization via syncX and sets the reconfiguration ID and the request flag as in the original template. The transition from the urgent state ObtainConfig to WaitConfig initiates a synchronization with the region confStore such that confStore requests the current configuration from the configuration store. After the configuration has been received, both regions synchronize via receivedConfig. Then, internal behavior checks the conditions for executing the reconfiguration at the transition from WaitConfig to CheckX. The operation checkStructuralConditionForX receives the ID of the current configuration as an integer parameter. The operation is then implemented using a switch case that decides whether the configuration with the given ID fulfills the structural condition. We may obtain the switch case by successively matching the component SDDs that specify the structural condition to all states of the reachability graph.

6.5.3.3 Adaptated Behavior Specification of the Executor

The executor RTSC, as introduced in Section 4.4.2, needs to modify the model@runtime for executing reconfigurations. In the MATLAB-specific reconfiguration controller, however, only the configuration store may switch between configurations and thereby modify the model@runtime. Therefore, we need to realize modifications of the model@runtime by a communication between executor and configuration store. As a consequence, we extend the executor RTSC generation template by an additional region confStore that implements the communication with the configuration store. In addition, we may simplify the internal behavior because it no longer needs to execute the reconfiguration. Figure 6.28 shows the adapted internal behavior region and the new confStore region of the executor RTSC generation template.

The internal behavior no longer contains the hierarchical state LocalExecuteY2 (cf. Figure 4.16). The corresponding behavior is now contained in the confStore region. Whenever the adaptation RTSC of embeddedCI synchronizes to trigger the execution of a reconfiguration, it now synchronizes directly with confStore. confStore then sends a corresponding message to the configuration store which performs the desired operation.

In addition, we replace the structure type AffectedComponents, which is used by the RTSC of the embeddedCI multi port (cf. Figure 4.17), by an array implementation. Thereby, we avoid the use of variable-size data structures in Stateflow. Figure 6.29 illustrates the generated arrays and their usage at runtime for the executor of ConvoyCoordination.

The executor of ConvoyCoordination has three subport instances in the embeddedCI multi port instance that handle the interaction with the three embedded component instances cm, rg1, and rg2 (cf. Figure 6.23). Therefore, we generate arrays with length 3 in the adaptation RTSC of embeddedCI that replace the variable ac of type AffectedComponents. We generate one array with the same length for each attribute of AffectedComponents (cf. Figure A.71 in Appendix A.6.4.1).

Based on the array implementation and the reachability graph, we can now generate an implementation for the operation computeAffectedChildrenForAddConvoyMemberAtPos of the adaptation RTSC. This operation computes the embedded component instances that need to recon-

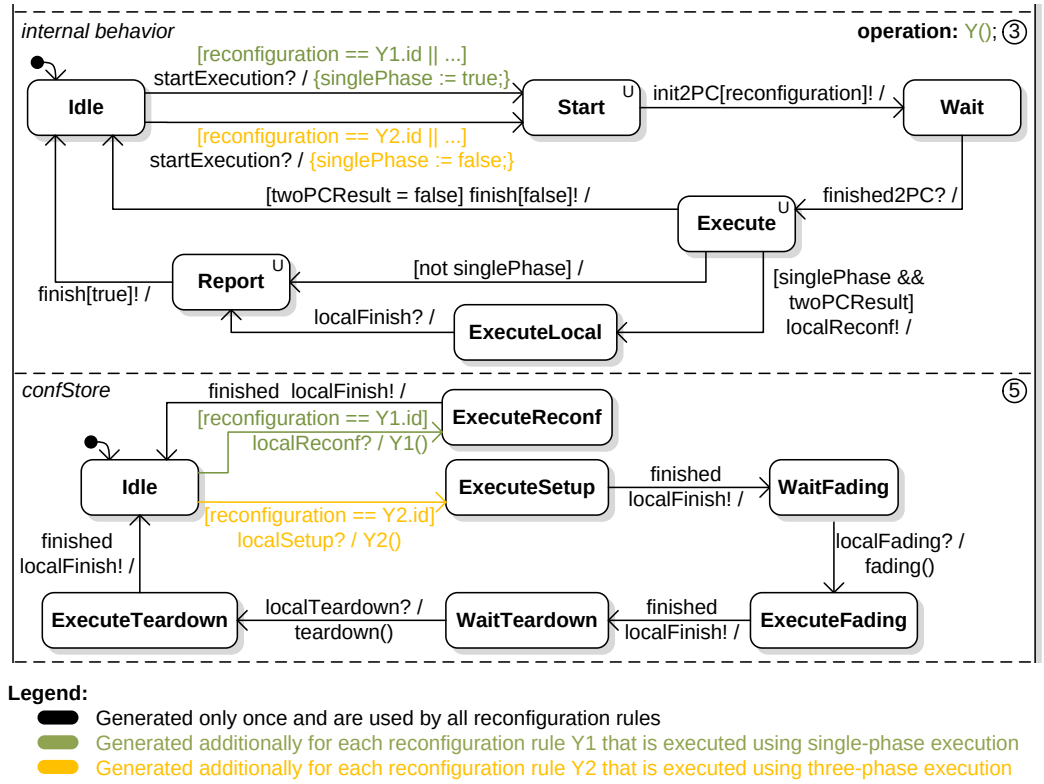


Figure 6.28: Adapted Generation Template for the Executor RTSC (cf. [Vol13])

figure for executing the CSD addConvoyMemberAtPos of ConvoyCoordination (cf. Figure 3.14). Based on the reachability graph in Figure 6.23 and the CSD, we can determine that only the reconfiguration createMemberPortsAfter needs to be triggered on cm. Therefore, the operation computeAffectedChildrenForAddConvoyMemberAtPos assigns 1 to the first entry of ac and 0 to the other two entries to indicate that only cm is affected by the reconfiguration. In addition, it assigns 2 to the first entry of message to indicate that the second message of the RE port interface specification of cm needs to be sent to cm (cf. Figure A.52). The remaining operations for accessing ac and its attributes can be translated by accessing and modifying the generated arrays.

6.5.4 Step 4: Encode Configurations and Generate Control Signals

In this step, we encode the configurations that are contained in the reachability graph into the functions of the configuration store RTSC. In particular, we generate implementations for the operations establishConfigX, setup, and fading that are contained in the executor region of the configuration store RTSC. These operations shall write the control signals to hybrid ports that are then connected to the control inports of our helper blocks when generating the Simulink model in Step 6 of our algorithm.

For the integrated CIC of a structured component, we create one control signal with an associated hybrid port at the configuration store

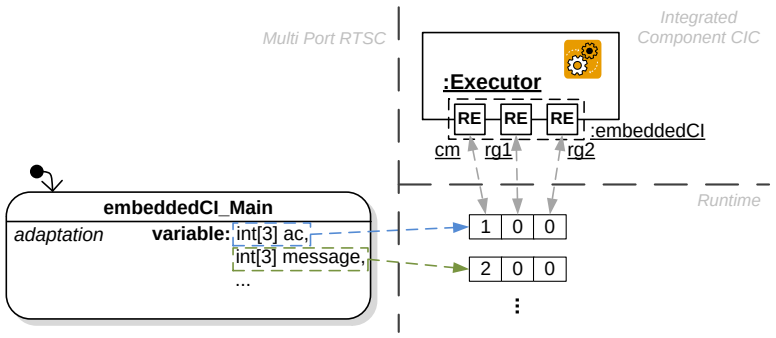


Figure 6.29: Array Implementation of the AffectedComponents Structure

- for each embedded component instance that is connected to the enable port of the corresponding enabled subsystem for activating and deactivating the instance in Simulink
- for each embedded fading component instance that sets the ctrl inport of the corresponding subsystem (cf. Section 6.3.1.3)
- for each discrete port instance of the structured component instance that defines the local_recv_net_addr of the corresponding delegation switch (cf. Section 6.3.3.4)
- for each MultiSourceControl and MultiTargetControl that defines the ctrl input of the blocks for rerouting the signal (cf. Section 6.3.2.1)
- for each discrete port instance of an embedded component instance that defines the receiver_net_addr of the corresponding port structure for controlling assembly connector instances and delegation connector instances (cf. Section 6.3.3.3)

In our example, we obtain a total of 22 control signals for the integrated CIC of ConvoyCoordination. In particular, we obtain control signals

- cm, rg1, and rg2 for the embedded component instances
- c1, c2, r1, r2, receiver, and speedProvider for the discrete port instances of ConvoyCoordination
- curPos for the MultiTargetControl block that handles the delegation of the continuous port instance curPos of ConvoyCoordination
- cm.c1, cm.c2, cm.p1, cm.p2, cm.strategy, cm.speedProvider, rg1.profileReceiver, rg1.next, rg1.r1, rg2.prev, rg2.profileReceiver, rg2.r2 for the discrete port instances of the embedded component instances where the name of the port instance is prefixed with the name of the component instance.

Figure 6.30 illustrates the hybrid ports that are generated for the control signals at the configuration store of ConvoyCoordination.

Based on the control signals, we now generate implementations of the establishConfigX operations in the configuration store RTSC. We illustrate the result for the operation establishConfig1 in Figure 6.26. The resulting implementation is shown in Listing 6.1. Applying the generated control signals to the Simulink model generated for the integrated CIC establishes the Simulink model shown in Figure A.97 that corresponds to config1 in the reachability graph in Figure 6.23.

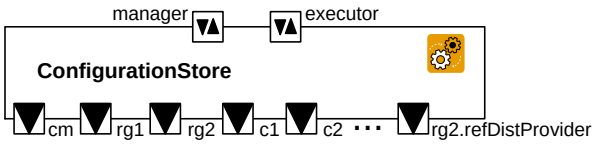


Figure 6.30: ConfigurationStore of the Component ConvoyCoordination with Hybrid Ports Generated for the Control Signals (cf. [Vol13])

Listing 6.1: "Implementation of the Operation establishConfig1"

```
cm   := 1;          speedProvider := 3;     rg1.profileReceiver := 4;
rg1  := 1;          curPos := 1;            rg1.refDistProvider := 8;
rg2  := 0;          cm.c1 := 7;             rg1.next := 0;
c1   := 1;          cm.c2 := 0;             rg2.prev := 0;
c2   := 0;          cm.p1 := 6;             rg2.profileReceiver := 0;
r1   := 5;          cm.p2 := 0;             rg2.refDistProvider := 0;
r2   := 0;          cm.strategy := 8;
receiver := 2;      cm.speedProvider := 9;
```

The implementations for the operations setup and fading are computed analogously. setup establishes the configuration after the setup phase as described in Section 4.2.2. fading triggers a state change in the Stateflow chart of the fading component (cf. Figure 6.10) to enable the corresponding fading function.

6.5.5 Step 5: Create Integrated System CIC

In this step, we replace the component instances that are contained in the initial system CIC by the integrated CICs of the components that have been computed in the previous steps. Component instances that are contained in an integrated CIC of a component are recursively replaced by their integrated CICs as well. The result is the integrated system CIC that encodes all configurations that the system may have during runtime. The integrated system CIC is then translated into a Simulink model as described in Section 6.3. The RTSCs that define the behavior of the component instances are translated to Stateflow charts as described in Section 6.4.

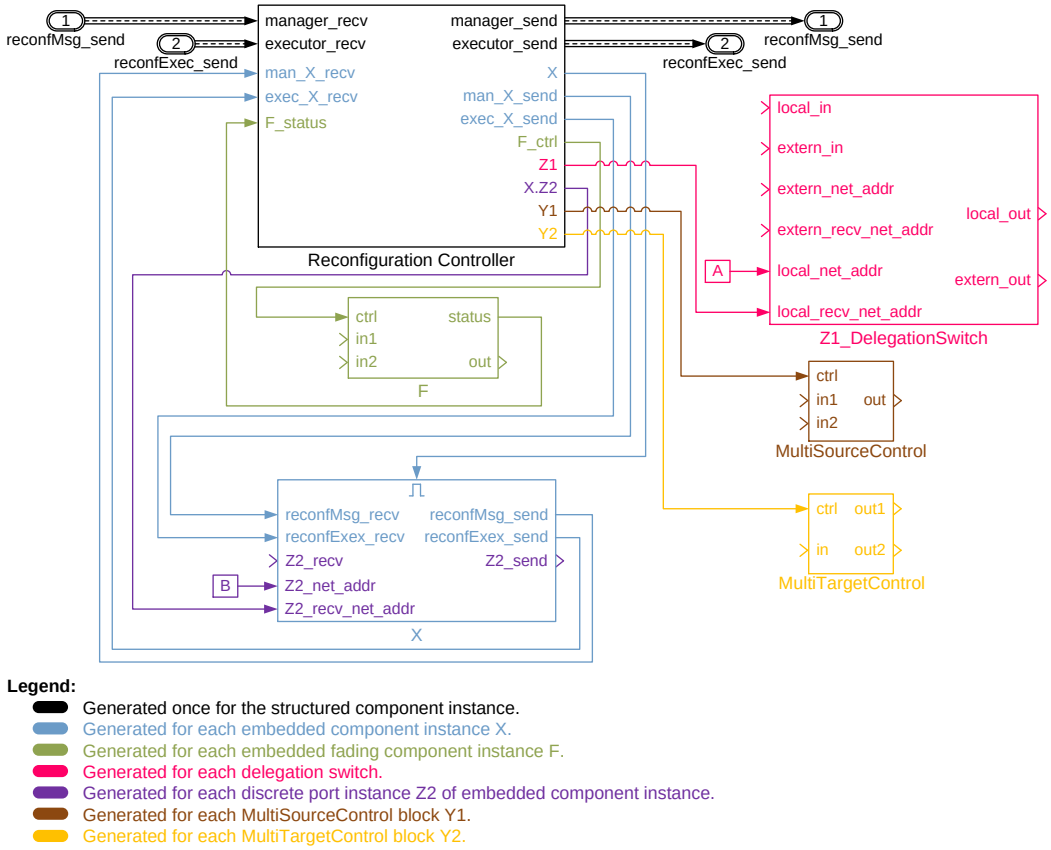
6.5.6 Integrate MATLAB-specific reconfiguration controller into the Simulink Block Diagram

In Step 6 of our algorithm in Figure 6.5, we translate the integrated system CIC into a Simulink block diagram. After generating the block diagram according to the rules presented in Section 6.3, we need to integrate the MATLAB-specific reconfiguration controller into the Simulink block diagram and connect its control signals.

Figure 6.31 shows the generation template for adding the MATLAB-specific reconfiguration controller including all of its control signals and their connections. An example of a resulting block diagram for an instance of ConvoyCoordination is presented in Appendix A.8.2.

The subsystem Reconfiguration Controller in Figure 6.31 contains the MATLAB-specific reconfiguration controller. The subsystem has different kinds of inports and outports. First, the four ports manager_recv, manager_send, executor_recv, and executor_send are generated for the

reconfMsg and reconfExec port instances of the structured component instance. Second, we obtain four ports man_X_recv, man_X_send, exec_X_recv, and exec_X_send for each embedded component instance X that correspond to a subport instance of the embeddedCI multi ports of manager and executor. Finally, we obtain one outport for each control signal that has been computed in Step 4 of our process (cf. Section 6.5.4), plus one additional inport for each fading component instance. In the following, we describe how these ports are connected to the remainder of the block diagram that has been generated according to the rules given in Section 6.3.



Legend:

- Generated once for the structured component instance.
- Generated for each embedded component instance X.
- Generated for each embedded fading component instance F.
- Generated for each delegation switch.
- Generated for each discrete port instance Z2 of embedded component instance.
- Generated for each MultiSourceControl block Y1.
- Generated for each MultiTargetControl block Y2.

Figure 6.31: Generation Template for Integrating the MATLAB-specific Reconfiguration Controller into a Block Diagram of a structure component instance

The ports man_X_recv, man_X_send, exec_X_recv, and exec_X_send are directly connected to the corresponding reconfMsg_recv, reconfMsg_send, reconfExec_recv, and reconfExec_send ports of the enabled subsystem X. We use a direct connection in this case because these assembly connector instances are immutable, i.e., as long as X is executed, the connection to the reconfiguration controller is active as well.

The ports for the control signals are connected as follows. The control signal X that has been generated for the embedded component instance X is connected to the enable port of the enabled subsystem X. By setting a 0 to the control signal, we stop simulating the subsystem and emulate the destruction of the component instance X. By setting a 1 to the control signal, we start simulating the subsystem and emulate the creation of the component instance X.

The control signal F_ctrl that has been generated for the embedded fading component instance F is connected to the ctrl inport of the corresponding subsystem F. In addition, the outport status of F is connected to the inport F_status of the subsystem Reconfiguration Controller. These control signals enable the interaction of the configuration store with the Stateflow chart shown in Figure 6.10 that is generated for a fading component instance.

The control signal Z1 is connected to the local_recv_net_addr inport of the delegation switch corresponding to the port Z1 of the structured component instance. The control signal then defines the net_addr of the receiving port structure. By changing the local_recv_net_addr via the control signal, we enable that the port instance is delegated to a different embedded component instance.

The control signal X.Z2 is connected to the recv_net_addr inport of the port structure corresponding to the discrete port instance Z2 of an embedded component instance X. This control signal defines the net_addr of the port structure that shall receive messages sent by Z2. Thus, we can redirect assembly connector instances by changing the recv_net_addr via the control signal.

The control signals Y1 and Y2 are used for emulating the reconfiguration of assemblies between continuous and hybrid port instances. Therefore, the control signals are connected to the ctrl inports of the corresponding MultiSourceControl and MultiTargetControl blocks. By modifying the control signal, we may change the sender or receiver of the signal, respectively.

The internal structure of the subsystem Reconfiguration Controller is a direct translation of the MATLAB-specific reconfiguration controller shown in Figure 6.24 (cf. [Vol13]). We refer to Appendix A.8.2 for a detailed description of the internals of the ReconfigurationController subsystem.

6.5.7 Realizing Port Reconfiguration in Stateflow Charts

In Step 7 of our algorithm in Figure 6.5, we translate the RTSCs of the component instances contained in the integrated system CIC to Stateflow charts as described in Section 6.4. If the component instance is reconfigurable, we also need to integrate additional constructs that enable to activate and deactivate parallel states in Stateflow for implementing the creation or destruction of (sub-)port instances. If a (sub-)port instance is activated in Simulink by a reconfiguration, then we also need to activate the corresponding parallel state that contains the behavior for this (sub-)port instance. If the (sub-)port instance is deactivated in Simulink, we need to deactivate the corresponding parallel state as well. Therefore, we generate one control variable for each (sub-)port chart in Stateflow. This control variable is true if the (sub-)port chart needs to be executed and false otherwise. In addition, we need to adapt the generation of Stateflow charts such that they use the control variable.

Figure 6.32 shows the general structure of a parallel state that may be activated and deactivated. The example is based on the chart for the subsystem rg1 in Figure 6.16. The parallel state r1 in Figure 6.32 corresponds to the parallel state r1 in Figure 6.16.

Inside r1 in Figure 6.32, we generate two states: Inactive and Active. The former indicates that the parallel state is currently inactive while the latter indicates that the parallel state is currently active. The state Active then contains the states and transitions that define the behavior of r1 (cf. Figure 6.16). The contents of Active are translated according to the rules defined in Section 6.4.

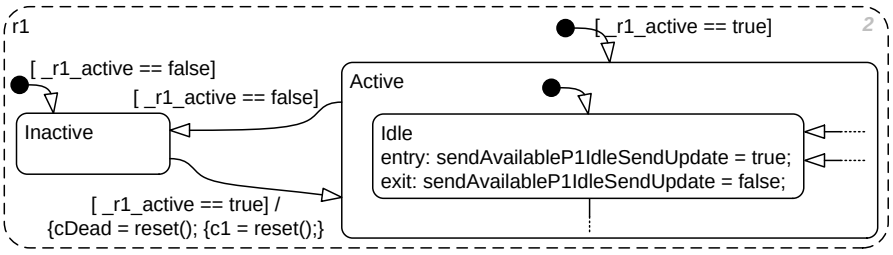


Figure 6.32: Reconfiguration in Stateflow

Both, Active and Inactive, have a default transition with a guard condition. The guard condition contains the control variable and defines whether the parallel state is initially active or not. During runtime, the parallel state may be activated and deactivated by modifying the control variable. After entering the state Active, the execution always starts at the initial state Idle. Furthermore, the transition from Inactive to Active resets all clock variables and set all variables that are owned by the parallel state to their initial values.

6.6 Implementation

We have prototypically implemented all steps of the algorithm shown in Figure 6.5. Our implementation is based on and integrated into version 0.5 of the MECHATRONICUML Tool Suite. Figure 6.33 shows the plugins that have been created as part of the implementation.

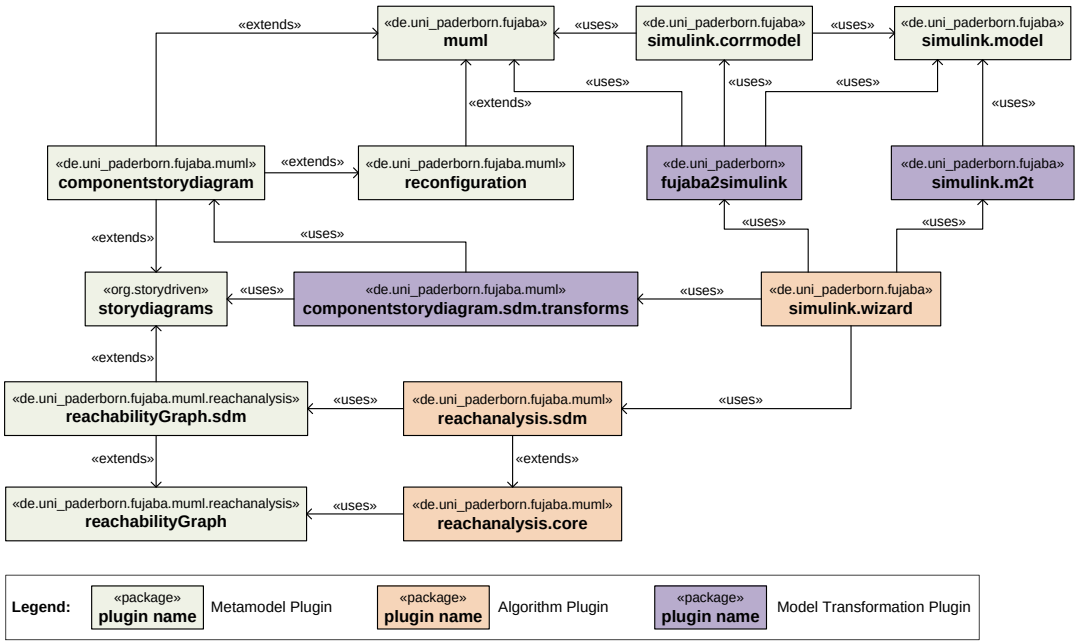


Figure 6.33: Plugins Implementing the Translation of MECHATRONICUML Models to MATLAB/Simulink Models

The plugin simulink.wizard implements the UI integration and controls the execution of the single steps of the algorithm shown in Figure 6.5. In the following, we describe which plugins implement which steps of the algorithm. The reachability analysis in Step 1 is implemented using our framework for reachability analysis (cf. Appendix C). The four plugins reachabilityGraph, reachabilityGraph.sdm, reachanalysis.core, and reachanalysis.sdm implement a reachability analysis based on story diagrams. Since we specify reconfiguration rules based on CSDs, we first translate the CSDs to story diagrams using the model transformation in plugin componentstorydiagram.sdm.transforms. The resulting story diagrams are then inserted into the reachability analysis. Step 2, i.e., computing the integrated CIC for a component, has been implemented based on the Eclipse project EMF Diff/Merge [Eclb] that enables to easily merge all configurations contained in the states of the reachability graph. Steps 3 to 5, namely generating the MATLAB-specific reconfiguration controller, encoding the configurations, and generating the integrated system CIC, have been implemented in Java as part of the simulink.wizard plugin.

Steps 6 and 7 have been implemented as a model-to-model transformation using triple graph grammars (TGG, [Sch95]). TGGs require a two domain metamodels and one correspondence metamodel. The domain metamodels define the source and target metamodels for the transformation while the correspondence metamodel associates elements of both metamodels that are equivalent with respect to the transformation. In our case, we use muml as the source metamodel. The plugin simulink.model contains a metamodel for Simulink and Stateflow models that we created based on EMF [SBPM08]. It serves as the target metamodel for the transformation. Finally, simulink.corrmodel contains the correspondence model while fujaba2simulink contains the TGG rules that define the transformation. We refer to our technical reports for a detailed description of the TGG rules [HRB+13, HRB+14]. After creating the Simulink and Stateflow models in EMF, we perform a layouting of both models. While layouting the Simulink model is only for usability reasons, layouting the Stateflow model is mandatory because layout defines the semantics in Stateflow. In particular, hierarchical states are defined by x-y-coordinates. We perform the layout using the tool Graphviz [Gra]. Finally, we generate a Simulink model file for the resulting Simulink and Stateflow models. We implemented this step as a model-to-text transformation using XPand [Ecla] that is contained in the plugin simulink.m2t.

6.7 Limitations

Our approach for the translation of MECHATRONICUML models to MATLAB/Simulink models underlies the following limitations:

1. Steps 1 to 5 of our algorithm shown in Figure 6.5 have only been defined and implemented for structured components. The reason is that we currently cannot define a reconfiguration controller for atomic components as discussed in Section 4.7.
2. Our approach does not support user-defined structure types. Currently, we only support the translation of the structure type AffectedComponents used in the executor RTSC (cf. Section 4.4.2).
3. We do not support transition actions that are specified by story diagrams except for those used in the executor RTSC. A prerequisite for translating user-defined story diagrams is a concept for translating structure types as mentioned above.

4. We do not support entry and exit points of hierarchical states with more than one region.
5. Do actions of states cannot be translated.
6. Complex transition actions including if-statements and loops are not supported.
7. Using multidimensional array data types for variables and using array data types for message parameters is not possible.

While limitations 4 to 7 are minor issues, limitations 1 to 3 require significant effort for being solved. Solving these limitations is beyond the scope of this thesis. In addition to the conceptual limitations, our implementation introduced in Section 6.6 does not yet cover all of the concepts presented in this chapter. In particular, our implementation does not support:

1. Activating and deactivating parallel states for (sub-)port RTSCs as described in Section 6.4.5
2. Translating the structure type AffectedComponents including the story diagrams that implement the operations of the executor RTSC as described in Section 6.5.3.3.
3. Urgent states as described in Section 6.4.4
4. Using more than one initial configuration for a component when computing possible configurations as described in Section 6.5.1.
5. Deriving an implementation for the operation checkStructuralConditionX as described in Section 6.5.3.2.

These tooling limitations prevent to automatically translate models of self-adaptive mechatronic systems such as the RailCab using the MECHATRONICUML Tool Suite. However, despite the limitations of our concepts and our implementation, our implementation is sufficient for translating a reasonable set of MECHATRONICUML models to MATLAB/Simulink and Stateflow as we show in our case study in Section 6.8.

6.8 Case Study

In this section, we evaluate our approach for enabling MIL simulation of mechatronic systems in MATLAB/Simulink. We evaluate our approach by conducting a case study based on the guidelines defined by Kitchenham et al. [KPP95]. In our case study, we evaluate the translation of MECHATRONICUML models for non-adaptive mechatronic systems to MATLAB/Simulink, i.e., models of systems that do not employ runtime reconfiguration. Thus, our case study considers Steps 6 and 7 of our algorithm shown in Figure 6.5. We perform our evaluation for three realistic examples of mechatronic systems but do not aim at generalizing this statement as part of this thesis.

We cannot yet conduct a case study for self-adaptive mechatronic systems such as the RailCab example presented in this thesis due to the limitations of our implementation. However, we have tested the effectiveness and feasibility of our approach by semi-automatically translating parts of the RailCab model from MECHATRONICUML to MATLAB/Simulink. These experiments have been successful, i.e., we could successfully emulate runtime reconfiguration for discrete and continuous component instances in Simulink. We refer to Pines [Pin12] and Volk [Vol13] for more information on the results of our experiments.

In the following, we describe the hypotheses and results of our case study for non-adaptive mechatronic systems.

6.8.1 Case Study Context

The objective of our case study is evaluating whether our translation of MECHATRONICUML models to MATLAB/Simulink and Stateflow models produces syntactically and semantically correct models that may be simulated in Simulink and Stateflow. We consider a model to be semantically correct if it shows the same behavior as the MECHATRONICUML model.

We conduct our case study based on models of three mechatronic systems that do not employ runtime reconfiguration. In the following, we give a brief description of the systems and denote the characteristics of the corresponding MECHATRONICUML models.

First, we consider the cooperating delta robots that are shown in Figure 6.34 [GTS14, PTD+14]. These robots are able to juggle a ball without utilizing a camera system. Instead, they sense the ball by sensors on the plate and compute a prediction when and where the ball will arrive at the other robot. This prediction is then sent to the other robot using a message and the other robot strikes based on the prediction and, in turn, computes a new prediction after hitting and thereby sensing the ball. The resulting MECHATRONICUML model of the discrete components is simple but relies on tight integration with the continuous components that contain the sensors for sensing the ball.

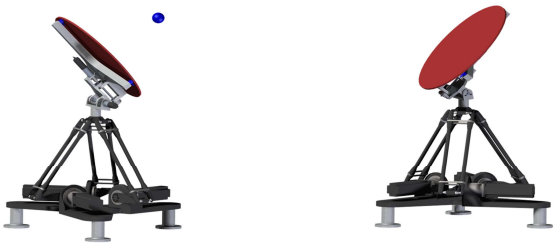


Figure 6.34: Cooperating Delta Robots (cf. [PTD+14])

Second, we consider a coordinated overtaking of cars shown in Figure 6.35 as introduced by Gerking [Ger13] and Pohlmann et al. [PHMG14]. There, the overtaking red car communicates with the overtaken yellow car such that the overtaking is safe, i.e., the overtaken car will not accelerate or decelerate if it is not safe. The MECHATRONICUML model contains hierarchical states and a complex timing specification.

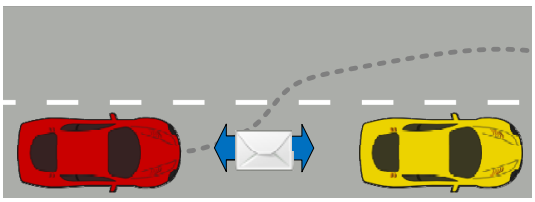


Figure 6.35: Coordinated Overtaking of Two Cars [PHMG14]

Finally, we consider the registration of RailCabs at track sections that we already used in our case study in Section 5.6. In particular, we simulate the scenario shown in Figure 6.36

where two RailCabs try to enter the same switch ts3. The MECHATRONICUML model extensively uses synchronizations and the coordination involves several component instances (cf. Section 5.1.2).

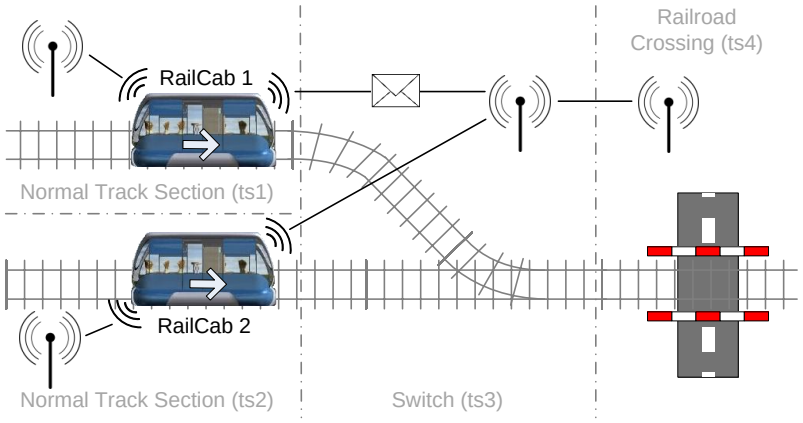


Figure 6.36: RailCabs Trying to Enter the Same Switch (cf. [HBDS15])

6.8.2 Setting the Hypothesis

The three example models that we use have been verified based on UPPAAL and our refinement check. Therefore, we consider them as correct with respect to their specifications.

For our case study, we define two evaluation hypotheses. Our *first evaluation hypothesis* H1 is that the generated MATLAB/Simulink and Stateflow models are syntactically correct. Our *second evaluation hypothesis* H2 is that the behavior of the generated MATLAB/Simulink and Stateflow models in a simulation complies to the behavior defined by the MECHATRONICUML model.

We evaluate our evaluation hypothesis based on MATLAB Release R2009b and our implementation described in Section 6.6. In particular, we evaluate H1 by compiling the generated model in MATLAB/Simulink. For evaluating H2, we first need to implement all continuous components in Simulink. Then, we simulate the models in Simulink and manually compare the behavior of the simulated model to the results of our verification procedures (cf. Section 4.5 and Chapter 5). For comparing the behavior, we plot values of variables using scope blocks (cf. Section 6.1.1) and analyze Stateflow charts using the Stateflow debugger [Matc].

6.8.3 Preparing the Input Models

In preparation of the case study, we obtained MECHATRONICUML models for the cooperating delta robots and the blind overtaking scenario. In addition, we use the RailCab models for entering a track section as described in Section 5.6.3. All of the models have been created using our implementation described in Section 3.6.

Each of the models contains the specification of RTCPs and components including their RTSCs. In addition, we created a CIC for each model that serves as an input for our translation.

6.8.4 Validating the Hypothesis

We start by translating the MECHATRONICUML model of the cooperating delta robots to MATLAB/Simulink. Then, we compile the resulting model and the compilation succeeds without errors. Thereafter, we integrate a Simulink model of the physical environment that includes, in particular, the movement of the ball and the sensing of the ball. Then, we simulate the resulting model. The simulation results show that the two robots can successfully exchange their predictions. The observed behavior in Simulink and Stateflow is compliant to the behavior specified in MECHATRONICUML.

Next, we translate the model for coordinated overtaking to MATLAB/Simulink. The compilation of the model succeeds without errors. Then, we integrate the generated model with a simple behavior of the overtaking car that signals that the overtaking has been finished. Thereafter, we simulate the resulting model. The simulation results show that the two cars can successfully coordinate during the overtaking. The observed behavior in Simulink and Stateflow is compliant to the behavior specified in MECHATRONICUML.

Finally, we translate the RailCab model for entering track sections to MATLAB/Simulink. Then, we compile the resulting model and the compilation succeeds without errors. In the step next, we implement a simple behavior for the continuous component Gates that represents the feedback controller of the gates of the railroad crossing (cf. Figure A.26 on Page 220). Thereafter, we simulate the resulting model. The simulation results show that the RailCabs may successfully register at the switch ts3. In particular, the simulation results show that only one RailCab at a time is allowed to enter the Switch. Thus, the observed behavior in Simulink and Stateflow is compliant to the behavior specified in MECHATRONICUML.

6.8.5 Analyzing the Results

The results of our case study show that our translation of MECHATRONICUML models into models of MATLAB/Simulink and Stateflow produces syntactically correct models. Thus, our first evaluation hypothesis H1 is fulfilled. In addition, the translation was fully automatized and did not require manual intervention. The simulation results show that the generated models behave as expected based on the verification results that have been obtained for the MECHATRONICUML models. Thus, our second evaluation hypothesis H2 is fulfilled as well and we conclude that our translation preserves the semantics of MECHATRONICUML.

In our case study, the most important *threats to validity* are as follows: (1) Our translation and its implementation do not yet support all modeling constructs of MECHATRONICUML as outlined in Section 6.7. Thus, models that use these modeling constructs will not be translated correctly. (2) We only tested the preservation of the semantics of MECHATRONICUML based on three examples. Although we consider all of these examples as realistic, other examples from other domains could be highly different. (3) We only checked manually that the generated Simulink and Stateflow models show the same behavior as the MECHATRONICUML models. Although we did not identify any deviations, we might have missed some minor deviation.

6.9 Related Work

This section discusses related works from four research areas. First, we review other approaches that enable reconfiguration of MATLAB/Simulink models (Section 6.9.1). Second, we compare our approach to other tools targeting the development of mechatronic systems (Section 6.9.2). Third, we discuss approaches enabling reconfiguration in AUTOSAR version 3.x (Section 6.9.3). Fourth, we relate our approach to approaches for hybrid verification (Section 6.9.4) that try to replace MIL simulation by a formal verification of the system.

6.9.1 Reconfiguration in MATLAB/Simulink

Cancare [Can08] and Paiz et al. [PKP07] describe approaches for simulating reconfigurable FPGA-boards in Simulink. They only switch between implementation variants of the same block using switches but provide no means for message-based communication and adding/removing components from the simulation. Schulze et al. [SWB12] provide a concept for product line support in Simulink where a concrete variant is configured via control signals. In addition, they support to switch between different variants of a component at run-time. In contrast to our approach, they do not enable to reconfigure connectors or to remove components completely.

The Quanser Real-Time Control Software (QUARC, [Qua]) provides special blocks for switching between two Simulink models during runtime. They stop the simulation, transfer variables, and restart the simulation on the target model. In contrast to our approach, this approach does not permit to simulate the transient phase where the reconfiguration is executed. In particular, this is necessary for correct simulation of fading functions. Kovácsházy et al. [KSP03] provide a block library for simulating reconfigurable digital signal processors (DSPs). Both approaches use self-defined blocks, which hinders the use of production code generators like TargetLink [dSP] or ASCET [ETA].

6.9.2 Reconfiguration in other Simulation Environments

There exist several competitors of MATLAB/Simulink. These include Modelica [Mod09, Fri04] with the commercial simulator Dymola [Das], CAMeL-View [iXt], SCADE [Est], and ASCET [ETA], that support the development and simulation of feedback controllers. None of these approaches natively supports runtime reconfiguration. CAMeL-View supports message-based communication using concepts of MECHATRONICUML [THB+10, BGSH11], Modelica can be extended by a library implementing RTCPs of MECHATRONICUML [PDS+12, PDM+14].

For Modelica, two extensions named Mosilab [ZJS08] and Sol [Zim07] exist that support reconfiguration. Both approaches rely on their own simulators because their extensions are not supported by Dymola. In contrast to our approach, they cannot use existing production code generators. ter Beek et al. [tBGS13] provide an approach for simulating a reconfigurable e-Banking system, which has been specified using Reo [Arb04], in Dymola. Reconfigurations are specified using graph transformations as in our approach, but the approach for simulation seems to be limited to supporting one particular application example with only two configurations.

Burmester et al. [BGH+07] provide an approach for simulating reconfigurable systems in CAMeL-View. They require to generate C++-Code for the reconfigurable discrete software that needs to be integrated manually with the controller code. 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 in Step $S_{5.4}$ of our process in Figure 6.4.

Güdemann et al. [GAOR07] support simulation of self-adaptive robots in SCADE. They model reconfiguration by manually specifying flags to switch between different function implementations in each robot. In contrast, our approach automatically generates control signals and may enable and disable parts of the model if they are not needed.

6.9.3 Reconfiguration in AUTOSAR 3.x

Since version 4.0, AUTOSAR supports reconfiguration based on modes as discussed in Section 3.7.1. For AUTOSAR 3.x [AUT11], which does not support reconfiguration, several approaches for integrating reconfiguration have been developed.

Becker et al. [BGN+10] define an extension for AUTOSAR to support architectural reconfiguration which makes it closest to our approach. In their approach, a developer needs to specify all configurations of a system manually including an automaton defining how to switch between the configurations. This is very similar to our approach, but our approach may automatically derive this information from a declarative rule set introducing less effort for a developer. Based on the automaton and the configurations, they generate an AUTOSAR system containing all configurations including code for a so-called *StateManager* and a *RoutingComponent*. The StateManager controls the current configuration, while the RoutingComponent redirects signals based on the current configuration. In contrast to our approach, their approach does not allow for early validation using MIL simulations.

Berger and Tichy [BT12] extend the AUTOSAR watchdogs towards transactional reconfigurations with rollback support, but they do not consider simulation of the system. Zeller et al. [ZPW+11, ZP12] and Klobedanz et al. [KKMR11] provide reconfiguration of networked embedded systems by reallocating software components to new ECUs at runtime. Their approaches can be used for technically realizing reconfiguration but not for MIL simulations in Simulink.

Trumler et al. [THP+07] and Feng et al. [FCT08] propose middlewares for automotive systems supporting runtime reconfiguration by migrating tasks (Trumler et al.) or switching between different component implementations (Feng et al.). Their middlewares are supposed to replace the AUTOSAR Runtime Environment (RTE) but do not support MIL simulation.

6.9.4 Hybrid Verification

Hybrid verification tries to formally prove that safety and liveness properties hold for a mixed discrete-continuous system. Such systems are usually formalized by a variant of hybrid automata [Hen96] that consist of a set of discrete locations where each location embeds a set of equations that defines how the continuous variables of the system evolve. The verification of hybrid automata is undecidable in the general case [HKPV98]. Thus, hybrid verification techniques fall in two categories. Approaches in the first category restrict themselves to simpler variants of hybrid automata whose verification is decidable. Approaches in the sec-

ond category apply approximation techniques for retrieving a finite state space. For recent surveys on hybrid verification approaches, we refer to Zaki et al. [ZTB08] and Alur [Alu11].

Approaches in the first category are typically based on variants of linear hybrid automata [Hen96, DISS11a]. Examples include HyTech [HHWT97], PHAVer [Fre05], RED [Wan05], and approaches by Damm et al. [DISS11a, DISS11b, DDD+12]. The property that all of these approaches have in common is that they significantly restrict continuous dynamics such that most systems of practical relevance cannot be specified with them [HHMWT00]. In particular, they do not support ordinary differential equations (ODEs) and differential algebraic equations (DAEs) that are essential for describing many physical phenomena. Since MATLAB/Simulink supports both [XC13, ch. 5.4], our approach supports such systems.

Approaches in the second category apply over-approximations of the continuous dynamics. Most approaches are based on flow pipes [CK99] where the system states are represented by polyhedra. Examples include HyperTech [HHMWT00], CheckMate [SK00, SRKC00], d/dt [ADM01, ADM02], an approach by Alur et al. [ADI06], and SpaceEx [FLGD+11]. A new class of approaches encodes hybrid models using constraints and solves them with a SAT-solver as, e.g., approaches by Ishii et al. [IUH11] and Eggers et al. [ERNF12]. All of the mentioned approaches have in common that they may only verify small system models with up to 200 variables [FLGD+11]. However, realistic examples that may be simulated in Simulink use thousands of blocks [SP12] where each block defines at least one variable.

In addition, none of the approaches mentioned above supports runtime reconfiguration which is supported by our simulation-based approach.

6.10 Summary

In this chapter, we introduce an approach for MIL simulation of self-adaptive mechatronic systems in MATLAB/Simulink and Stateflow. Our approach provides a syntactic decoupling of discrete and continuous components that enables to efficiently verify the discrete part of the system's behavior based on the compositional verification approach of MECHATRONICUML. Thus, we only need to rely on MIL simulations for testing the correctness of (1) the feedback controllers contained in the continuous components, (2) the fading functions used for replacing continuous components, and (3) the correct interaction of discrete and continuous components. As our main contribution, we define an algorithm that translates a MECHATRONICUML model into a MATLAB/Simulink and Stateflow model. In our approach, we explicitly compute and encode all possible configurations of the self-adaptive mechatronic system. The resulting Simulink model then enables to switch between the encoded configurations for emulating runtime reconfiguration. This enables to emulate runtime reconfiguration without needing to structurally modify the simulation model, which is not supported by Simulink.

Although our contributions have been illustrated based on MATLAB/Simulink and Stateflow, our approach for emulating reconfigurations of the software architecture of a system may easily be transferred to other languages and tools for MIL simulation such as Dymola [Das]/Modelica [Mod09].

7 Conclusions

7.1 Summary

Introducing self-adaptation into mechatronic systems increases the complexity of developing the software for them. In particular, it introduces more sources for errors that may occur at runtime and hardens to predict the behavior of the system. However, self-adaptive behavior is the basis for self-healing [Sha02, Pri13] and self-optimization [GRS09, GRS14] that enable to improve safety, availability, and (resource) efficiency of the system. The contributions of this thesis enable software engineers of self-adaptive mechatronic systems to cope with the additional complexity such that they may safely unleash the full potential of self-adaptive behavior when developing the next generation of mechatronic systems. In the scope of this thesis, all of our contributions have been defined based on the MECHATRONICUML method, but our contributions may also be transferred to other model-driven approaches that provide support for developing platform-independent models of software for self-adaptive mechatronic systems. We have implemented all of our contributions as part of the MECHATRONICUML Tool Suite [DGB+14].

As our first contribution, we define a component model that enables to specify a software architecture for a self-adaptive mechatronic system. The component model explicitly includes the necessary variability in the definition of component types and provides CSDs, which enable the model-driven specification of runtime reconfigurations of the software architecture. In particular, CSDs improve comprehensibility of reconfiguration behavior by providing a visual representation based on the concrete syntax of components [Moo09, HB14]. As a key benefit of our component model compared to related approaches, we explicitly consider the integration of feedback controllers including their reconfiguration into the software architecture. In addition, our component model enables to establish RTCPs between AMS for dynamically building NMS and provides component SDDs that allow specifying architectural constraint based on components. We illustrate the effectiveness of our component model by creating a model of the RailCab system including the reconfiguration behavior for building convoys that is documented in detail in Appendix A.

As our second contribution, we define a formal execution semantics for reconfigurations in a hierarchical component model that is based on an adaption of the 2-phase-commit protocol [BHG87, ch. 7]. In our approach, we syntactically extend the components in our component model by a dedicated reconfiguration controller that executes the 2-phase-commit protocol. The reconfiguration controller enables to execute reconfigurations across different levels of hierarchy without violating component encapsulation. Our approach significantly reduces the complexity of specifying such hierarchical reconfigurations by providing a rather simple declarative specification based on tables that enables to automatically generate an implementation of the 2-phase-commit protocol. We extended the existing 2-phase-commit protocol such that it can execute reconfigurations in a self-adaptive mechatronic system including the exchange of feedback controllers according to ACI-T properties. The ACI-T properties are

atomicity, consistency, isolation, and correct timing. While our 2-phase-commit protocol specification guarantees atomicity and isolation offhand, we define a verification approach for guaranteeing consistency and a correct timing of reconfigurations. Thereby, we can ensure the correctness and, thus, the safety of the reconfigurations. We demonstrated the effectiveness of our approach by specifying a hierarchical reconfiguration behavior for our RailCab model. In addition, we generated the 2-phase-commit protocol implementation and verified the resulting models as described on our website [Hei13]. Recently, our approach for hierarchical reconfigurations has been integrated into the ProCom component model by Hang and Hansson [HH13].

As our third contribution, we enhance MECHATRONICUML's compositional verification approach [GTB+03] by a new refinement check. Our refinement check enables to verify that the ports of the components in our component model correctly refine the roles of the RTCPs that define the interaction of components. In particular, our approach enables to prove that all safety and liveness properties that have been verified for the RTCPs still hold for the ports of the components. Our refinement check is based on test automata. The test automaton encodes both, the behavior of the role and the conditions of the refinement definition that is to be checked. Our construction of the test automaton is parameterized such that it supports to verify correct refinements based on six different refinement definitions. Each refinement definition supports different kinds of constructs that may be used in RTSCs and different kinds of safety and liveness properties. Combined with an automatic selection of the refinement definition to be used, our approach enables for a fully automatic verification of refinements as part of the compositional verification approach. We evaluate our approach by conducting a case study based on the RailCab system. In particular, our case study shows the viability of the automatic selection and verification of different refinement definitions. In addition, we illustrate how the returned counterexamples enable to identify the root cause of a refinement violation.

As our fourth and final contribution, we provide an approach for MIL simulation of self-adaptive mechatronic systems. This approach enables to test the correct integration of discrete components and feedback controllers, which cannot be tackled by formal verification techniques for complex systems such as the RailCab. As our main contribution, we defined how message-based communication of discrete components and reconfiguration behavior of structured components may be realized in a tool for MIL simulation that has no built-in support for such behavior. We illustrated our contributions based on MATLAB/Simulink, which is a de facto standard tool in industry for developing and simulating feedback controllers. However, our contributions are not limited to MATLAB/Simulink but may also be used in related transformations [PHMG14] to other simulation tools that share the same restrictions with respect to reconfiguration such as Dymola/Modelica [Das]. Since the simulation model may not structurally change within these simulation tools while executing a simulation, we encode all possible configurations of the system into the simulation model. During a simulation, we may then switch between the different configurations and thereby simulate the reconfiguration behavior of the system. In our approach, we define how a MATLAB/Simulink and Stateflow model can be derived from a MECHATRONICUML model by an automatic model transformation. We evaluate our model transformation by a case study where we translate MECHATRONICUML models of three different mechatronic systems to MATLAB/Simulink. The results of our case study show that our model transformation pre-

serves the semantics of MECHATRONICUML and that the MECHATRONICUML models are much more concise compared to the resulting Simulink and Stateflow models.

In combination, our contributions reduce the complexity of specifying reconfiguration behavior for a hierarchical component model. Moreover, our integrated analyses enable software engineers to proof the correctness of the reconfiguration behavior and thereby re-establish the predictability of the system's behavior at runtime.

7.2 Future Work

The results of this thesis give rise to different possibilities for future works that we highlight in the following. As a basis, future works may enhance the contributions of this thesis by overcoming the limitations and possibly relaxing the assumptions that we described in the corresponding sections of the previous chapters. In addition, all of the contributions should be further evaluated in industrial projects and by using models from different domains such as automotive [FMB+09], avionics, or factory automation. In the following paragraphs, we discuss further directions for future works.

Requirements Engineering The input for the contributions of this thesis is the domain-spanning conceptual design of the self-adaptive mechatronic system that has been created by experts from all involved disciplines [GFDK09, GSG+09]. This specification uses a state-based technique for describing different configurations of the system as we illustrated in our paper [HSST13]. State changes in this approach typically translate to reconfigurations in MECHATRONICUML. Future works should investigate how this specification may be complemented by model-based requirements engineering techniques that focus particularly on software reconfiguration. Examples include adapt cases [LNGE11] and goal-based techniques like the approach by Cheng et al. [CSBW09]. Such approaches would improve the early consideration and traceability of reconfiguration-related requirements.

Cognitive Operator Our component model supports to define a software architecture including reconfiguration operations for the reflective operator of the OCM. In addition, it includes an interface to the controller level by using continuous components. Future works should provide a similar interface to the cognitive operator of the OCM (cf. Section 2.1.2). A starting point is given by using unsafe ports as proposed by Giese and Schäfer [GS13] that define interactions with non-real-time parts of the software. However, the interface to the cognitive operator needs to be integrated with our concept for transactional execution of reconfigurations such that the cognitive operator may trigger the execution of reconfigurations.

Monitoring Monitoring the environment and the operations of the mechatronic system itself are crucial for self-adaptive behavior. At present, we assume that all relevant monitoring data is gathered and accumulated by discrete atomic components in our component model. At present, MECHATRONICUML does not support the developer in specifying monitoring behavior. Therefore, future works should integrate monitoring of the system behavior [DGR04, WH07] into MECHATRONICUML, e.g., using a framework like Kieker [vHWH12]. In particular, this should also enable to specify additional monitoring in the reconfiguration controller of a structured component, e.g., for monitoring information

entering the structured component. An example is given by monitoring the current speed in the component VelocityController (cf. Figure 3.7 on Page 3.7) for deriving whether the RailCab drives slow or fast.

Uncertainty The decision about executing a reconfiguration is made based on monitoring data, which reflects information about the system itself and its physical environment, and based on communication with other systems in the environment. As a result, the effectiveness of the reconfigurations is determined by the quality of the knowledge about the environment. Often, this knowledge is incomplete or inconsistent, e.g., due to false assumptions, unpredictable phenomena in the environment, or even imprecise and inaccurate sensors [RJC12]. Therefore, future work should investigate whether the reconfiguration behavior in our approach may be improved by explicitly addressing uncertainty during the development, e.g., using RELAX [WSB+09] or ActiFORMS [IW14].

Quiescence The concept for quiescence of discrete atomic component instances that we outline in Section 4.2.3 needs to be further elaborated and evaluated. In particular, future works should investigate whether it is possible to perform part of the necessary runtime analysis already at design time, e.g., by identifying states that always fulfill a part of the imposed conditions for quiescence and by labeling these beforehand. In addition, it might be possible to automatize the creation of the condition for quiescence at least partially. An idea is introducing an ontology [GOS09] that may be used for relating monitored signals at port instances to properties of the physical system such as speed or distance to another system. Then, we may specify constraint patterns that automatically translate typically unsafe situations like high speed combined with a small distance into conditions for quiescence.

Learning Reconfigurations Our approach for transactional execution of reconfigurations only applies pre-programmed reconfigurations based on monitored situations. That means that the system will always react with the same reconfiguration to the same environmental situation. Future works may utilize the cognitive operator of the OCM for evaluating the effect of a particular reconfiguration in a specific situation. Then, we can provide several reconfiguration rules for a situation and the system can adjust the decision which rule to execute based on past decisions. It would also be possible that systems share their experiences to learn from each other. This, in turn, could provide a data set that is large enough to apply machine learning [Mit04] to further optimize reconfiguration decisions and predictions of the system. In our current approach, this would require an adaptation of the RTSCs for manager and executor at runtime as illustrated, for example, by Schäfer and Wehrheim [SW07]. In addition, this would enable to inject new reconfiguration rules or even completely new components including their reconfigurations into the system at runtime. In addition, that would require a modification of the allocation and to check at runtime whether this change does not compromise the consistency and timing properties of the 2-phase-commit protocol.

Security At present, our approach only addresses the safety of the system by applying formal verification and MIL simulation for guaranteeing that the system adheres to its specification. At runtime, however, security becomes an issue because self-adaptive mechatronic systems shall engange in NMS where they communicate via wireless communica-

tion links. These wireless communication links could be used by an intruder to perform, for example, a man-in-the-middle attack [Kiz05] that compromises the safe operation of the NMS. As a consequence, future works first need to integrate an authentication mechanism [DVOW92, BCK98] into the instantiation of RTCPs on system level to ensure that no unauthorized system enters an NMS. Second, future works need to integrate the use of encryption standards like the Advanced Encryption Standard (AES, [NIS01, DR02]) into RTCPs to enable secure communication. Such security measures may probably be generated into the system automatically when deriving the platform-specific model.

Executing Reconfigurations Our reconfiguration approach integrates flat switching for replacing feedback controllers [OMT+08]. In this approach, the decision whether a reconfiguration is possible may, in some cases, depend on the values of the new controller. These values cannot be obtained before executing the setup phase in our current approach but at this point no abort is allowed. Thus, it may happen that a reconfiguration that has been started cannot be finished. A solution would be to extend our approach towards a 3-phase-commit protocol [BHG87, SS83] consisting of a voting, pre-commit, and commit phase. Then, the execution of the setup phase would be part of the pre-commit phase. After executing the precommit phase, children are still able to abort the reconfiguration. Since no modification of the behavior took place in the setup phase, this will be possible and safe. In addition, it might be necessary to integrate roll-back behavior [ZCYM05, LLC10] or a controlled transition into a fail-safe behavior [dLdCGFR06] if an unexpected hardware failure occurs while executing the reconfiguration.

Refinement of Multi Roles At present, our refinement check is only applicable to single roles and single ports. Future works should extend this approach towards checking correct refinements for multi roles that include reconfigurations, i.e., the instantiation and removal of subrole instances. In [HH11a], the relaxed timed bisimulation has already been extended towards multi roles, but it requires a dedicated refinement check. Therefore, future works should extend our test automaton construction such that we may verify refinement of multi roles. Initial ideas towards such extended test automaton construction have already been presented by Brenner [Bre10] but require significant extensions of the approach. In this context, especially refinements of multi roles to ports of multi parts as for the multi part RefGen in Figure 3.5 are challenging and require additional concepts for constructing the test automaton.

Counterexample Analysis The counterexamples returned by our refinement check are tool-specific and refer to the generated test automaton. The test automaton, however, is not familiar to a developer and, therefore, interpreting the counterexample still requires a detailed knowledge of our test automaton construction. Future works should provide means for translating counterexamples back to the role and port RTSCs using, for example, the approaches by Gerking [Ger13] or Hegedüs et al. [HBRV10]. This back-translation of counterexamples may additionally provide an automatic root cause analysis of the refinement violation. The counterexample may be associated to the specific test construct described in Section 5.3.2 that lead to the error state which, in turn, can be associated to the root cause of the violation.

Synthesis of Component Behaviors The RTSC of a discrete atomic component is assembled from the port RTSCs. Typically, the port RTSCs of a component are not independent of each other. For example, they may need to exchange data or one port may only enter a particular state if one of the other ports is (or is not) in a specific state. In previous works, Eckardt and Henkler [EH10] as well as Goschin [Gos14] provided an automatic synthesis of component behaviors that resolves such dependencies automatically based on a formal dependency language [DGB14]. These approaches need to be extended towards supporting multi ports and runtime reconfiguration.

Model-Based Testing At present, our approach for MIL simulation of a self-adaptive mechatronic system only supports the developer in translating the MECHATRONICUML model into a MATLAB/Simulink model. Future works shall provide additional support for the remaining process steps for performing MIL simulations. In particular, Steps $S_{5.3}$ and $S_{5.4}$ of our process in Figure 6.4 need to be extended by a framework for model-based testing. This framework shall support the developers in deriving scenarios from the requirements in an (semi-) automatic fashion. In addition, it needs to support the automatic execution of the resulting test cases and the computation of metrics like test coverage [JFA+07, OHY11, T-V, Mate]. For Step $S_{5.3}$, an approach for automatically deriving test scenarios from a scenario-based requirements specification [Gre11] could significantly reduce the effort for testing and may positively influence test coverage.

Deployment The platform-independent models that may be created using the contributions of this thesis need to be deployed on a hardware platform [PMDB14] for being executed. The hardware platform need to provide enough resources for executing the reconfigurations and for executing the resulting CICs. This can be guaranteed by a using a deployment approach [TMD09, Dea07, MMMR12] that considers reconfigurations [Poh13].

Appendix A

Complete RailCab Example

This appendix introduces a complete example of a self-adaptive mechatronic system whose software has been specified using MECHATRONICUML. In particular, we continue the RailCab system [HTS+08a, HSD+15] that we already used in the main chapters of this thesis for illustrating our concepts. In the remainder of this chapter, we introduce the remaining parts of the MECHATRONICUML model of the RailCab system. We omit all models that have already been introduced in the main chapters and only provide references to those models in this appendix. All models presented in the following have been implemented in the MECHATRONICUML Tool Suite as far as possible under the given limitations of our tooling as discussed in the respective sections of our main chapters. The model is available on our website [HS15]. At present, the example is still limited in the number of use cases that it supports. In particular, we currently only enable to build and extend convoys with additional RailCabs. We do not yet support dissolving convoys and that RailCabs leave a convoy.

In the following, we start in Section A.1 by presenting the RTCPs that are used by the discrete components of the RailCab's software architecture. Thereafter, we introduce the behavior models and a simple environment model that can be used for instantiating RTCPs on system level in Section A.2. Section A.3 introduces one additional component that has not been included in Section 3.1, while Section A.4 introduces instances of these components for different convoy situations. In Section A.5, we present RTSCs for all discrete atomic components that we defined in our component model. Next, we describe the reconfiguration behavior of all structured components including a declarative, table-based specification of the reconfiguration behavior and the CSDs of the components. Section A.7 presents the component SDDs of our components. Finally, Section A.8 presents an excerpt of a generated Simulink model for the components RefGen and ConvoyCoordination.

A.1 RTCPs

This section introduces the RTCPs that specify the communication between the discrete components of the RailCab. In particular, we introduce the RTCPs ConvoyEntry (Section A.1.1), ConvoyCoordination (Section A.1.2), ProfileDistribution (Section A.1.3), SpeedTransmission (Section A.1.4), StartExecution (Section A.1.5), StrategyExchange (Section A.1.6), and NextSectionFree (Section A.1.7). The RTCP DistanceTransmission has been introduced in Section 2.4, while the RTCP EnterSection has been introduced in Section 5.1.

A.1.1 ConvoyEntry

The RTCP ConvoyEntry, whose declaration is shown in Figure A.1, provides a simple negotiation of a convoy coordinator. ConvoyEntry only has one role peer with a cardinality 2. Thus, two communication partners execute the same behavior for electing a coordinator. The RTCP defines a message buffer for one message and a message delay of 10 ms. In our RailCab example, ConvoyEntry is refined by the port peer of the component OperationStrategy as shown in Figure 3.6. The initial version of the RTCP has been derived from the real-time coordination pattern Master-Slave-Assignment [DBHT12] but significantly extended for the RailCab system.

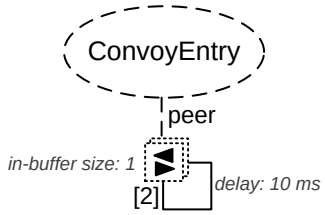


Figure A.1: Declaration of the RTCP ConvoyEntry

Figure A.2 shows the RTSC of the role peer. The behavior that is specified by the RTSC is as follows. Both peers start in the state NoAssignment. In the following, we will refer to them as "the one peer" and "the other peer" for explaining the interaction between the two peers. The RTSC uses two Boolean variables masterPossible and slavePossible that encode whether a peer can operate as a coordinator or as a member, respectively.

If masterPossible is true, then the one peer may nondeterministically switch to MasterProposed by sending a youSlave message to the other peer. If the other peer cannot be a member, it fires the self transition of NoAssignment and answers with cannotSlave. In this case, the one peer switches back to NoAssignment as well. If the other peer may still be a member (slavePossible is true), then the other peer switches to AcceptSlave after receiving the youSlave message and sends a confirm. Then, the other peer switches from AcceptSlave to StartingSlave. This transition is used for triggering the reconfiguration for becoming a member in a component that uses this RTCP. Therefore, it specifies a deadline of 50 ms. The one peer switches from MasterProposed to StartingMaster. This transition is used for triggering the reconfiguration for becoming a coordinator in a component that uses this RTCP.

The further behavior depends on whether the reconfigurations have been successful or not. In the RTSC, we model both results by using non-deterministic choice expressions in the entry actions of the states StartingSlave and StartingMaster. If the one peer has successfully executed the reconfiguration for becoming coordinator (member), then masterStarted (slaveStarted) is true. If masterStarted is true, then the one peer sends masterReady otherwise it sends cannotMaster while switching to WaitForSlaveFinish. In the same fashion, the other peer sends slaveReady if slaveStarted is true and cannotSlave, otherwise, while switching to WaitForMasterFinish.

If the other peer receives cannotMaster from the one peer, then it switches back to NoAssignment regardless of the value of slaveStarted and sets slavePossible to false. Thus, it cannot be member because the one peer cannot be the coordinator. If slaveStarted is true and

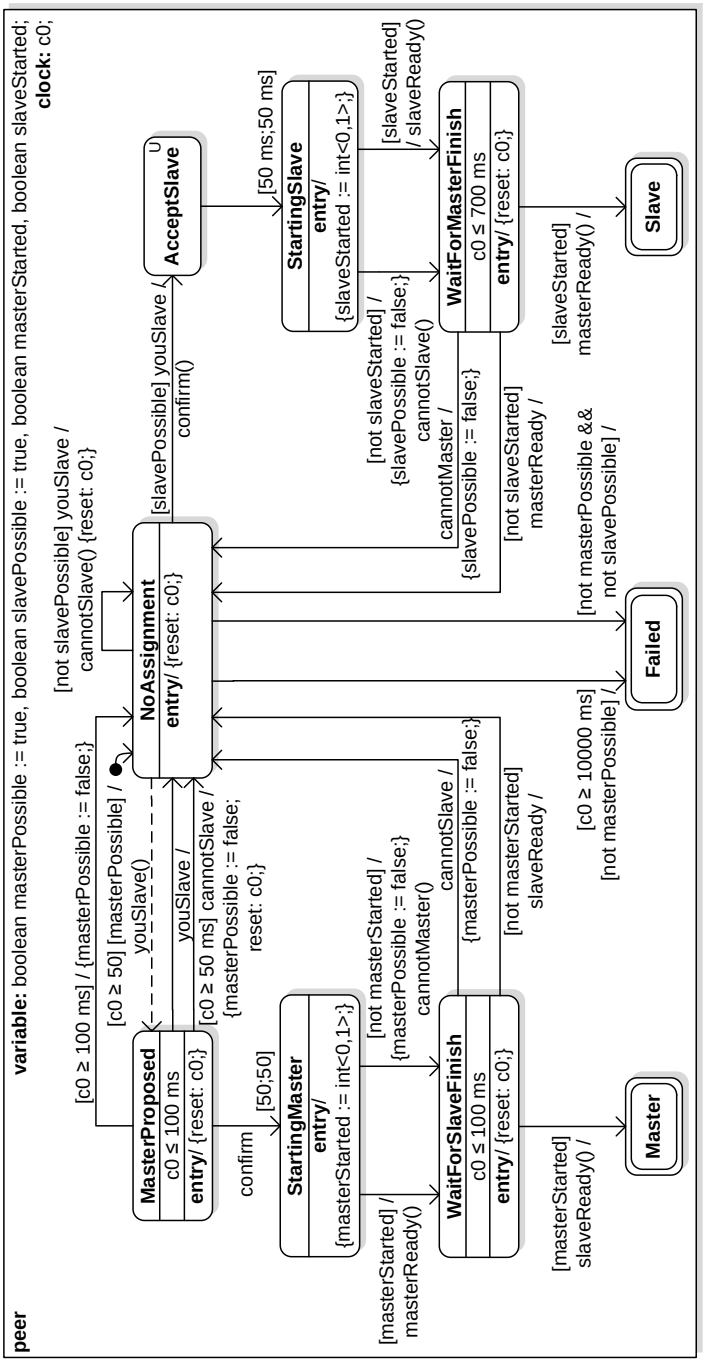


Figure A.2: RTSC of the Role peer of the RTCP ConvoyEntry

the other slave receives masterReady, it switches to Slave and the assignment is finished for the other peer. If slaveStarted is false and the other peer receives masterReady, then the other peer switches back to NoAssignment. The one peer reacts in the same way as the other peer based on the messages slaveReady and cannotSlave. If both, masterPossible and slavePossible are false, then RTSC switches to Failed. In Failed, the one peer may still receive youSlave messages from the other peer, which it answers with cannotSlave. In addition, the one peer will switch from NoAssigment to Failed if it has not received a message for 10.000 ms. These two transitions are necessary to prevent deadlocks in case that one or both peers start with one of the variables masterPossible or slavePossible being false at the start of execution.

We verified the RTCP using UPPAAL. We have verified the following properties:

- The RTCP is free from deadlocks.
- None of the message buffers may overflow.
- If one peer reaches the Master state, then the other peer will always eventually enter the Slave state.
- If one peer enters the Fail state, then the other peer will always eventually enter the Fail state as well.

A.1.2 ConvoyCoordination

The RTCP ConvoyCoordination, whose declaration is shown in Figure A.3, is responsible for managing the convoy. In particular, this RTCP finally decides whether a RailCab may join a convoy as a member and it defines the position where the RailCab may enter the convoy. Both decisions are made based on so-called motion profiles. A motion profile, in the following simply referred to as *profile*, is a certificate how a RailCab moves in a particular driving maneuver such as braking. For driving in a convoy, each RailCab needs to be equipped with one or many of such profiles in order to guarantee safe convoys [FHK+13, FHK+14].

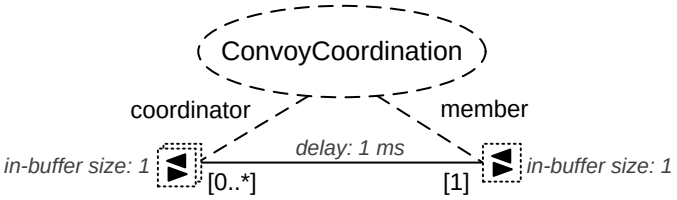


Figure A.3: Declaration of the RTCP ConvoyCoordination

The RTCP consists of two roles, namely coordinator and member. coordinator is a multi role such that a coordinator RailCab may coordinate a convoy with many members. If a new member wants to enter the convoy, it sends all of its profiles to the coordinator. Then, the coordinator checks whether an assignment of profiles to convoy members exists such that the convoy is safe in all driving maneuvers. If so, the new member may enter the convoy, otherwise it may not enter. Figure A.4 shows the RTSC that defines the behavior of the coordinator role while Figure A.5 shows the RTSC of the member role.

The behavior of the coordinator is slightly extended compared to our previous publications [FHK+13, FHK+14]. In particular, it enables that the profiles of RailCabs that already

drive as part of the convoy may be changed if a new RailCab wants to enter. We describe the behavior executed by coordinator and member in the following.

The coordinator starts by initializing its variables. In particular, it must create a new ProfileStore that stores all received profiles and that is assigned to variable allProfiles. Then, at an arbitrary point in time, a new member may appear and a corresponding subrole is created by the transition from Idle to HandleNewMember. The member fires the transition from Idle to Request and creates its profiles. In addition, it sends requestConvoyEntry to the coordinator. The subrole receives this message and synchronizes via newMemberPossible with the adaptation RTSC. Then, the adaptation RTSC checks whether it is possible and useful to add a new convoy member at the given point in time. If not, it synchronizes via entryFail and the subrole will decline the convoy entry. If the member may enter, the adaptation RTSC synchronizes via entrySuccess and the subrole approves the convoy entry.

After this, the member initiates sending its profiles using the message startProfileTransmission while entering the Wait state. The subrole acknowledges that it is readyForProfileTransmission and the transmission of the profiles starts. As long as the member has unsent profiles, it switches from Transmit to awaitAck and sends a profile to the subrole. The subrole stores the profile in allProfiles and acknowledges via profileReceived. After all profiles have been transmitted, the member sends endOfProfileTransmission, which causes the subrole to switch to ProfilesReceived. Using this transition, the subrole synchronizes with the adaptation RTSC via requestPosition in order to request an entry position for the new member.

The adaptation RTSC then invokes calculateProfiles. This function compares the profiles of all RailCabs with each other in order to obtain an assignment of profiles to RailCabs such that the convoy is safe [FHK$^+$13, FHK$^+$14]. If no such assignment could be found, newRailCabPosition is 0 and the adaptation RTSC synchronizes via entryFail with the subrole. Then, the subrole declines the convoy entry and switches to Fail. Similarly, the member switches from WaitForPosition to Declined and the convoy entry has failed. Finally, the adaptation RTSC deletes the subrole including its profiles and returns to Idle.

If calculateProfiles could obtain a profile assignment, the adaptation RTSC switches to UpdateRequired. If changed is false, then no profiles of the current convoy members have been changed. In this case, the adaptation switches to Finished and synchronizes via entrySuccess with the subrole. Then, the subrole sends the profile and the position to the member. The member acknowledges by sending startConvoy and enters the Convoy state. After receiving this message, the subrole also switches to Convoy and synchronizes via convoy with the adaptation RTSC, which finishes the convoy entry.

If calculateProfiles derived a profile assignment that requires to change the profiles of the existing convoy members, the adaptation RTSC switches to UpdateProfiles. Then, the adaptation RTSC iterates all subroles and synchronizes via sendNewProfile with them. In this case, the corresponding subrole switches from Convoy to NewProfile and sends the new profile to the corresponding member. The member processes the message at the self-transition at Convoy and confirms the update. The subrole of the new member is treated as before and the convoy setup finishes after all members have been informed about their new profiles.

A.1.3 ProfileDistribution

The RTCP ProfileDistribution, whose declaration is shown in Figure A.6, is responsible for propagating profiles and the data, which is necessary for using the profile, inside the co-

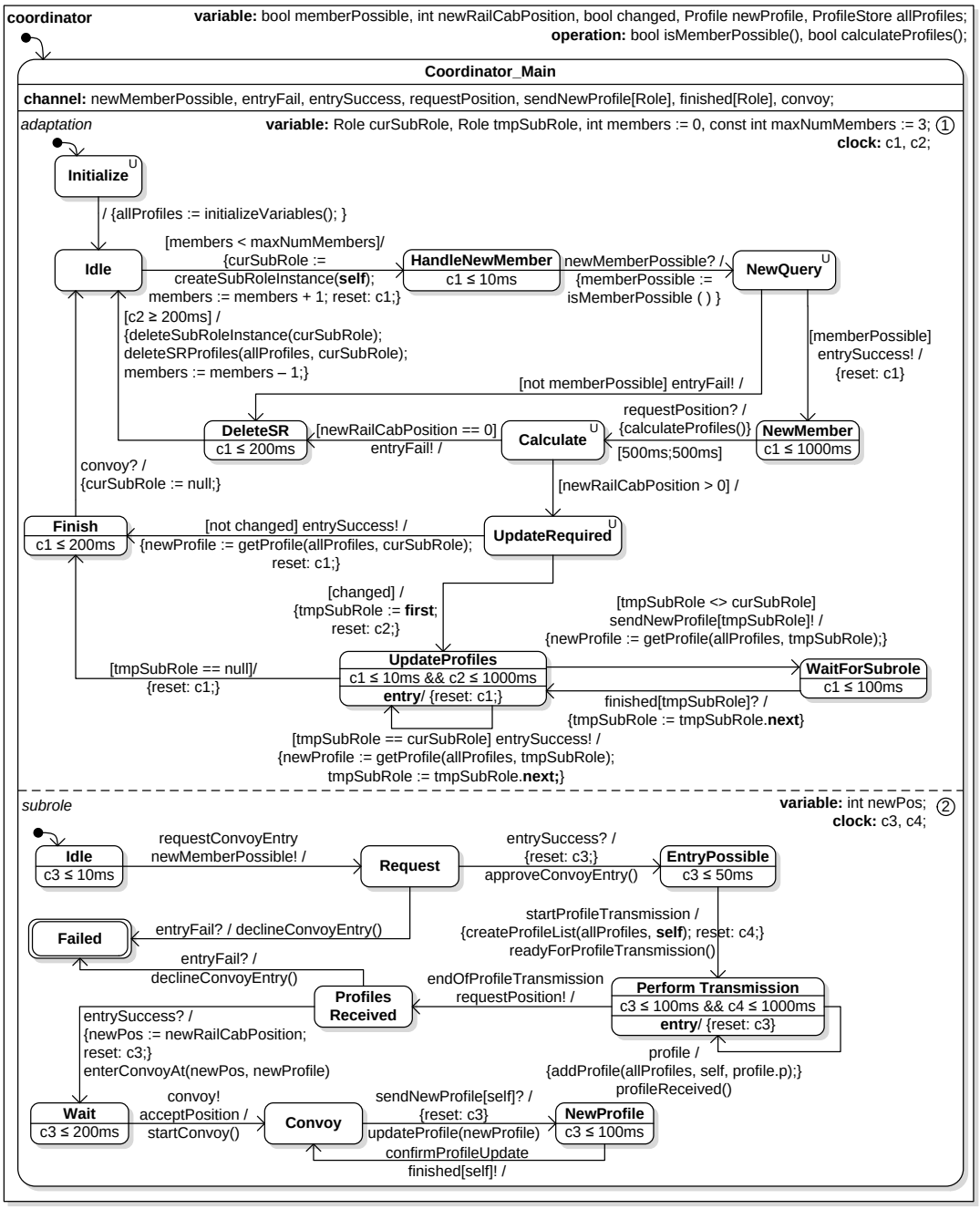


Figure A.4: RTSC of the Role coordinator of the RTCP ConvoyCoordination (cf. [FHK+14])

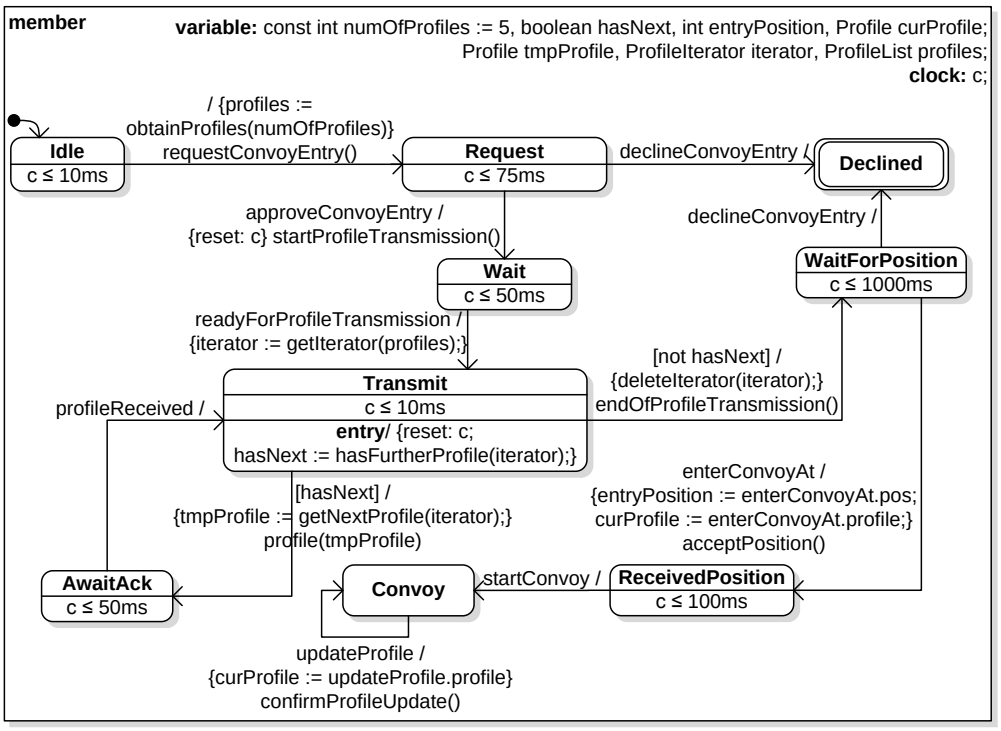


Figure A.5: RTSC of the Role member of the RTCP ConvoyCoordination

ordinator RailCab. This profile is used within the ConvoyCoordination component shown in Figure 3.5. The multi role profileProvider sends the profile information to many profileReceivers and receives information about the current maximum speeds for the profileReceivers. The latter information may be used for adjusting the convoy speed after a profile change.

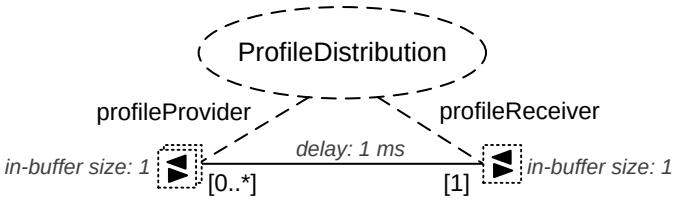


Figure A.6: Declaration of the RTCP ProfileDistribution

Figure A.7 shows the RTSC of the multi role profileProvider, while Figure A.8 shows the RTSC of the role profileReceiver. The execution of the profileProvider starts in the Idle state of the adaptation RTSC. At an arbitrary point of time, it may add a new subrole by firing the self-transition at the Idle state. Thus, it will be defined by the implementing component at which point in time a new instance is required.

Once per second, the adaptation RTSC switches from Idle to sendUpdate and synchronizes with the first subrole via startUpdate. This initiates and update process where new data and profile information are sent to the receivers. The synchronization causes the first subrole to switch from Idle to SendMsg. If a new profile is available, it sends a newProfile message to the

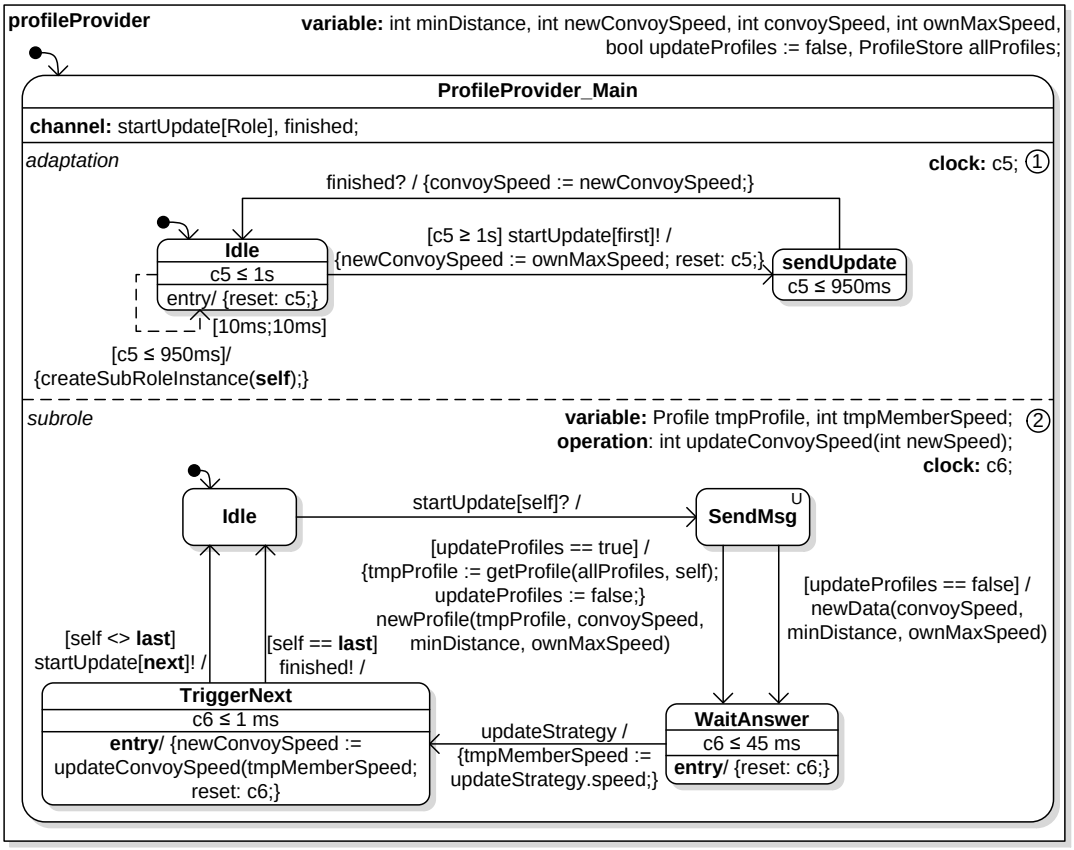


Figure A.7: RTSC of the Role profileProvider of the RTCP ProfileDistribution

profileReceiver. This message contains the profile and information that is necessary for using the profile such as the current reference speed of the convoy, the minimum distance to be kept, and the own potential maximum speed of the coordinator. If no new profile is available, then the subrole sends newData that contains the same information as newProfile except for the profile. Thereby, we acknowledge the fact that applying a new profile requires more complicated operations by the profileReceiver and that the profiles will change less frequently than the remaining information because the remaining information depends on the goals of the RailCabs and the current environmental conditions such as strong wind or slopes.

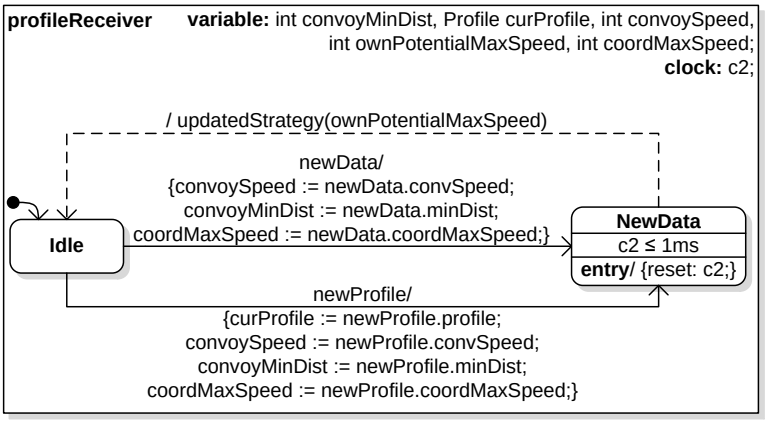


Figure A.8: RTSC of the Role profileReceiver of the RTCP ProfileDistribution

After receiving either newProfile or newData, the profileReceiver switches to NewData. Then, it sends updatedStrategy containing its new potential maximum speed back to the subrole of profileProvider. Then, the subrole switches to TriggerNext and updates the convoy speed. The corresponding operation updateConvoySpeed computes the minimum of all speeds provided by the profileReceivers and stores it in newConvoySpeed. Then, the subrole either triggers the next subrole or, if it is the last one, it synchronizes via finished with the adaptation RTSC. Finally, the adaptation RTSC returns to Idle and sets the convoySpeed to the newConvoySpeed. As a result, the new convoy speed will be applied as part of the next update.

A.1.4 SpeedTransmission

The RTCP SpeedTransmission, whose declaration is shown in Figure A.9, is used for periodically transmitting the current speed of the RailCab. It has been derived from the Real-Time Coordination Pattern PeriodicTransmission [DBHT12].

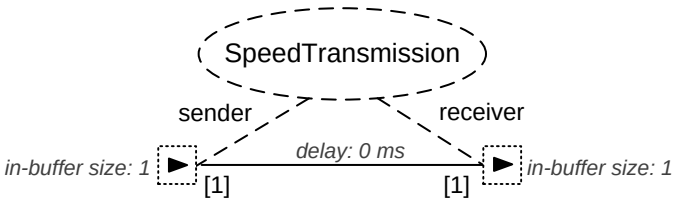


Figure A.9: Declaration of the RTCP SpeedTransmission

The behavior of the two roles sender and receiver, as given by the RTSCs in Figure A.10, is quite simple. Every 100 ms, the sender sends the current speed of the RailCab via newSpeed. The receiver waits in PeriodicReceiving for the new speed value. If newSpeed arrives, it fires the self-transition at PeriodicReceiving and stores the new speed value in the variable speed. If the new speed value does not arrive within 100 ms, then the receiver switches to Timeout. If eventually a new speed value arrives, the receiver switches back to PeriodicReceiving. The Timeout state may be used for handling a delayed update if necessary.

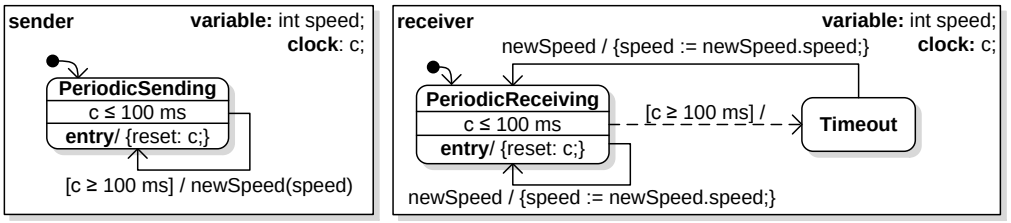


Figure A.10: RTSCs of the Roles sender and receiver of the RTCP SpeedTransmission

A.1.5 StartExecution

The RTCP StartExecution, whose declaration is shown in Figure A.11, enables the initiator to trigger the executor to execute some behavior on demand. We use this RTCP inside the ConvoyCoordination component (cf. Figure 3.5 on Page 42) such that one instance of RefGen may trigger the next one after it has finished its computation.

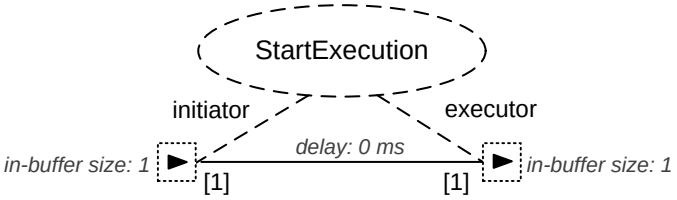


Figure A.11: Declaration of the RTCP StartExecution

The behavior of the two roles initiator and executor, as given by the RTSCs in Figure A.12, is quite simple. At an arbitrary point of time, the sender sends a startExecution message containing a newSpeed and a curPos parameter to the executor. The executor receives this message and may perform a computation using the parameter values.

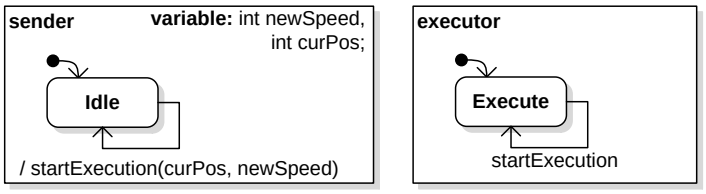


Figure A.12: RTSCs of the Roles initiator and executor of the RTCP StartExecution

The conditions when the sender is required to trigger the executor need to be defined by the component that uses this RTCP. In the same fashion, the operation to be executed by the executor needs to the defined by the component.

A.1.6 StrategyExchange

The RTCP StrategyExchange, whose declaration is shown in Figure A.13, is used for distributing information about the current operating strategy of the RailCab inside RailCabDriveControl. Since we are currently using a very simple operating strategy, the resulting behavior of the two roles sender and receiver, as given by the RTSCs in Figures A.14 and A.15, is rather simple.

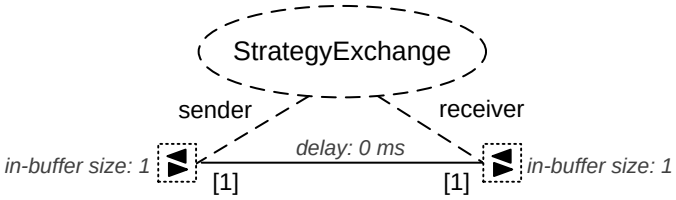


Figure A.13: Declaration of the RTCP StrategyExchange

At an arbitrary point in time, the sender sends a updateStrategy message to the receiver that contains information about the new strategy. At present, this message only contains the new maximum speed and minimum distance as parameters. Upon sending, it resets c1 and waits for 10 ms for an ackStrategy of the receiver. Upon receiving the updateStrategy message, the receiver stores the parameters into two variables and returns to WaitForUpdate after 5 ms thereby sending ackStrategy.

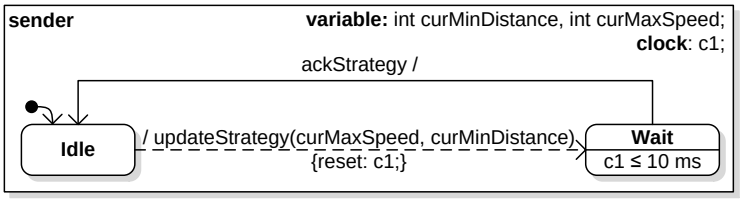


Figure A.14: RTSC of the Role sender of the RTCP StrategyExchange

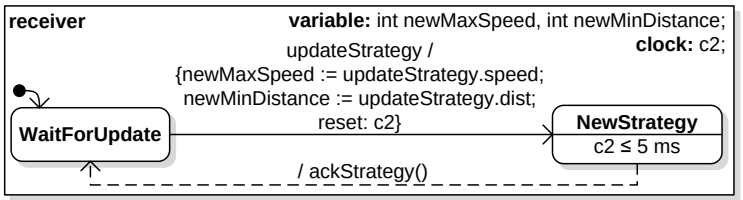


Figure A.15: RTSC of the Role receiver of the RTCP StrategyExchange

At present, the message exchange has been derived from the Real-Time Coordination Pattern Producer-Consumer [DBHT12]. If a more complicated operating strategy is applied, it might be necessary to extend this RTCP.

A.1.7 NextSectionFree

The RTCP NextSectionFree, whose declaration is shown in Figure A.16, has two single roles named tracksection and switch. The role switch is to be implemented by a switch while the role tracksection is to be implemented by the track section following the switch. Both roles have an in-buffer of size one. The transmission delay for a message is 3 ms.

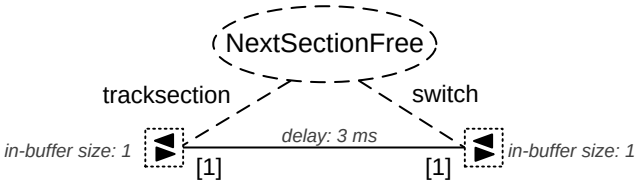


Figure A.16: Declaration of the RTCP NextSectionFree

Figure A.17 shows the RTSCs of both roles. The behavior implemented by the RTSCs is as follows: Initially, both RTSCs are in their Idle states. Then, switch sends a message requestSectionStatus to the tracksection at an arbitrary point in time thereby resetting its clock c2. Then, it waits for at most 500 ms in state WaitForSection for the answer of tracksection. tracksection receives the message requestSectionStatus at the urgent transition to Request and, thus, processes the message as soon as it arrives. Then, tracksection determines whether it is free, which modeled by the non-deterministic choice expression in the entry-action. After at least 400 ms, tracksection answers with sectionStatus where the current status is encoded as a Boolean parameter. After 550 ms, switch processes this message at the transition from WaitForSection to Idle. While firing the transition, switch assigns the parameter value of the message sectionStatus to its variable status.

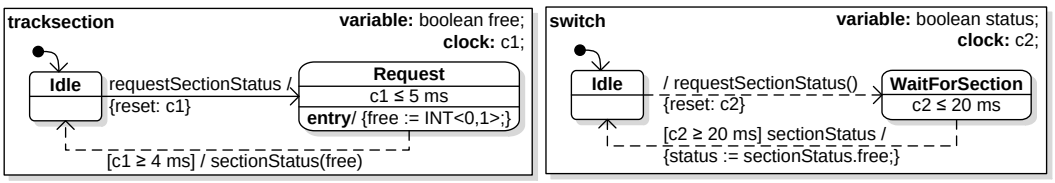


Figure A.17: RTSCs of the Roles tracksection and switch of the RTCP NextSectionFree

We verified the behavior of the RTCP in UPPAAL and showed four properties:

1. The behavior is free of deadlocks.
2. If tracksection and switch are in their idle states, then free and status have the same value.
3. If switch is in state WaitForSection, it will always eventually return to Idle.
4. There exists a execution where switch eventually enters WaitForSection.

A.2 Instantiating Real-Time Coordination Protocols on System Level

This section provides additional models for instantiating RTCPs on system level. In particular, we provide a simple discovery protocol that enables to store information about other systems in the environment in Section A.2.1. Thereafter, we introduce the RTSC of the protocolInst broadcast port that enables to instantiate the RTCP ProtocolInstantiation in Section A.2.2. Finally, Section A.2.3 presents the RTSCs of the roles requestor and requestee of the RTCP ProtocolInstantiation (cf. Section 3.4.2).

A.2.1 A Simple Discovery Protocol and Environment Model

In an open-world scenario [BDNG06], we need to gain knowledge about other systems in the environment for being able to collaborate with them. This is achieved by using a discovery protocol. A discovery protocol (periodically) broadcasts information about the system itself and listens to broadcast messages by other AMS. The information published via the broadcast port includes the networking address of the broadcast port and a short system identification.

The information about other AMS that is received by the broadcast port needs to be stored locally. We developed a simple *environment model* for storing this information. In the environment model, we distinguish between known types of systems and unknown types of systems. A known type of system is a type of system that the mechatronic system needs to interact with for realizing its functionality. In the RailCab system, other RailCabs and track side systems like track sections or switches are considered to be known systems. In contrast, an unknown type of system is a type of system that the mechatronic system usually does not interact with, but which it meets nevertheless. In the RailCab system, we may consider cars as unknown systems. In a close-world scenario, this model needs to be loaded from a local storage.

Figure A.18 shows a class diagram of an environment model for the RailCab system. It consists of an application independent part and an application specific part. The application independent part is the same for all AMS. It specifies the Enviroment which consists of an arbitrary number of ExternalSystems. For each ExternalSystem, we store its address and the timestamp indicating the last receipt of a message from the particular system. In addition, the application independent part contains a class UnknownSystem that is used for storing information about unknown types of systems. The application specific part contains a class RailCabKnownSystem that stores all the information about known systems. In addition, the enumeration RailCabKnownType contains one literal for each type of system that the RailCab knows. In this case, it knows other RailCabs and track sections.

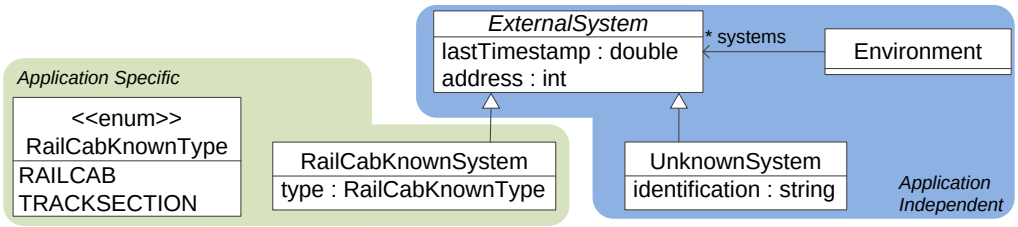


Figure A.18: Environment Model for the RailCab System

The environment models needs to be managed by the discovery protocol. If a message of a system that is not yet contained in the environment model is received, the discovery protocol needs to add an instance of RailCabKnownSystem or UnknownSystem to the environment model. If a message of a system that is already contained in the environment model is received, only the time stamp is updated. The timestamp may be used to clean the environment model from time to time. If no message of a particular system has been received for a longer period in time, it can be assumed that the corresponding system has moved out of reach and may no longer be contacted. The corresponding object is then removed from the environment model.

Most networking standards already include such discovery protocols. Examples include the Neighbor Discovery Protocol for IPv6 networks [NNSS07], the Bluetooth Service Discovery Protocol [Blu10], or ZigBee's device discovery protocol [Zig08]. They all fulfill the requirements stated above. In particular, any device sends broadcast messages including its own identification and listens to broadcast messages of other devices. Depending on the particular radio technology used for realizing communication between AMS, the discovery protocol for this technology should be used to fill the environment model given in Figure A.18.

In course of this thesis, we only consider platform independent models. These do not contain platform specific information like a concrete radio technology. However, we want to support simulation of AMS as early as possible (cf. Chapter 6) based on the platform independent models. In such simulations, we cannot rely on technology specific discovery protocols. Therefore, we provide a simple discovery protocol for platform independent models that is to be replaced by the technology specific protocol when creating the platform specific model.

The behavior of the broadcast port in our simple discovery protocol is given by the RTSC in Figure A.19. The RTSC contains two states: Idle and Update. Initially, the RTSC is in state Idle. Every 5 s, the RTSC fires the self-transition at the lower right of Idle. This transition sends a systemInformation message via the broadcast port. The systemInformation message contains the address of the mechatronic system, a short identification, and a timestamp. If the broadcast port receives such a message from another system, then the transition from Idle to Update fires. The transition consumes the message and stores the information on the other system in temporal variables. The entry action of Update updates the information on the system in the environment model. If the system has already been contained in the environment model, the method returns true. In this case, the RTSC returns to Idle using the upper transition not performing any further actions. If the system has not been contained in the environment model, the RTSC fires the lower transition from Update to Idle. This transition invokes the operation addSystem that adds a new system to the environment model based on the received information. If c1 becomes larger than $10min$, the RTSC fires the self-transition at the upper left of Idle. This transition invokes the clean operation that removes all systems from the environment model where no systemInformation message has been received for the past 10 min.

We specified the operations used in the RTSC of the discovery protocol formally by story diagrams. The operations updateEnvironment and clean are application independent in our specification. The operation addSystem is application dependent, because it needs to instantiate a <Sys>KnownSystem object with the corresponding enum literal depending on the short identification contained in the systemInformation message.

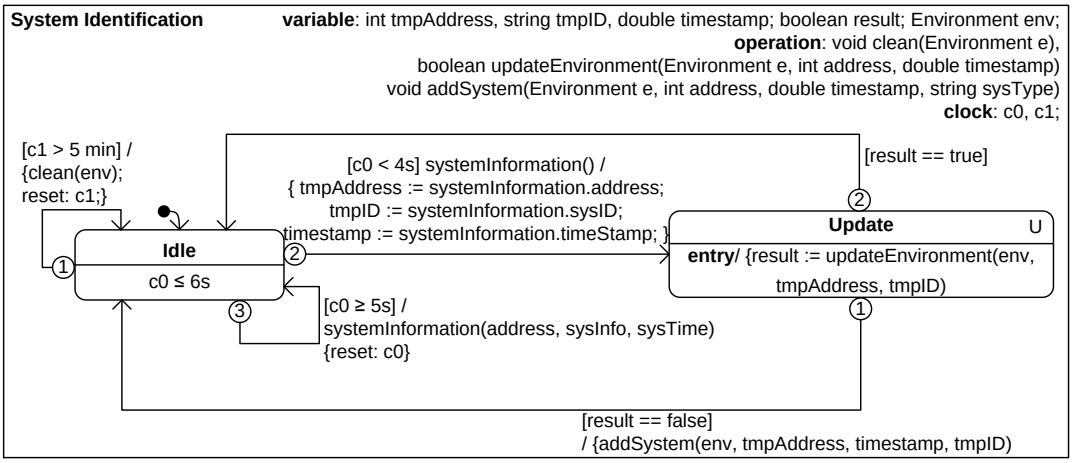


Figure A.19: RTSC for the SystemIdentification Protocol

The story diagram for the operation updateEnvironment is shown in Figure A.20. It takes the Environment object, the address which has been received, and the time stamp of the message as parameters. Then, the first story node tries to match an ExternalSystem in the environment with the given address. If such ExternalSystem can be found, the attribute lastTimestamp is set to the time given as a parameter. In this case, the matching was successful and the story diagram assigns true to the out parameter result. If no ExternalSystem with the given address could be found, the story diagram assigns false to the out parameter result.

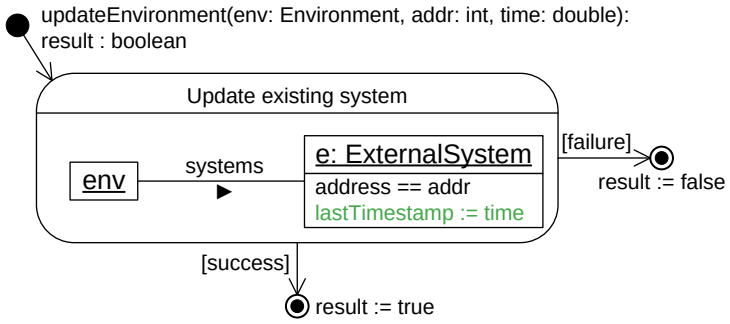


Figure A.20: Story Diagram Implementing the Operation updateEnvironment

The operation addSystem shown in Figure A.21 is application specific and needs to be generated using the enumeration RailCabKnownType. The story diagram contains one story node for each entry of the enumeration and an additional entry for UnknownSystems. The control flow specifies one decision node for each entry of the enumeration where one outgoing activity edge compares the id given as a parameter to the identification of the known system. In the example, the first decision node specifies the guard id == "RailCab". The else activity edge leads to the next decision node. The final else edge leads to the story node creating an UnknownSystem in the Environment.

The clean operation is formalized by the story diagram shown in Figure A.22. As parameters, it takes the Environment object and the current time. Then, the for-each activity node

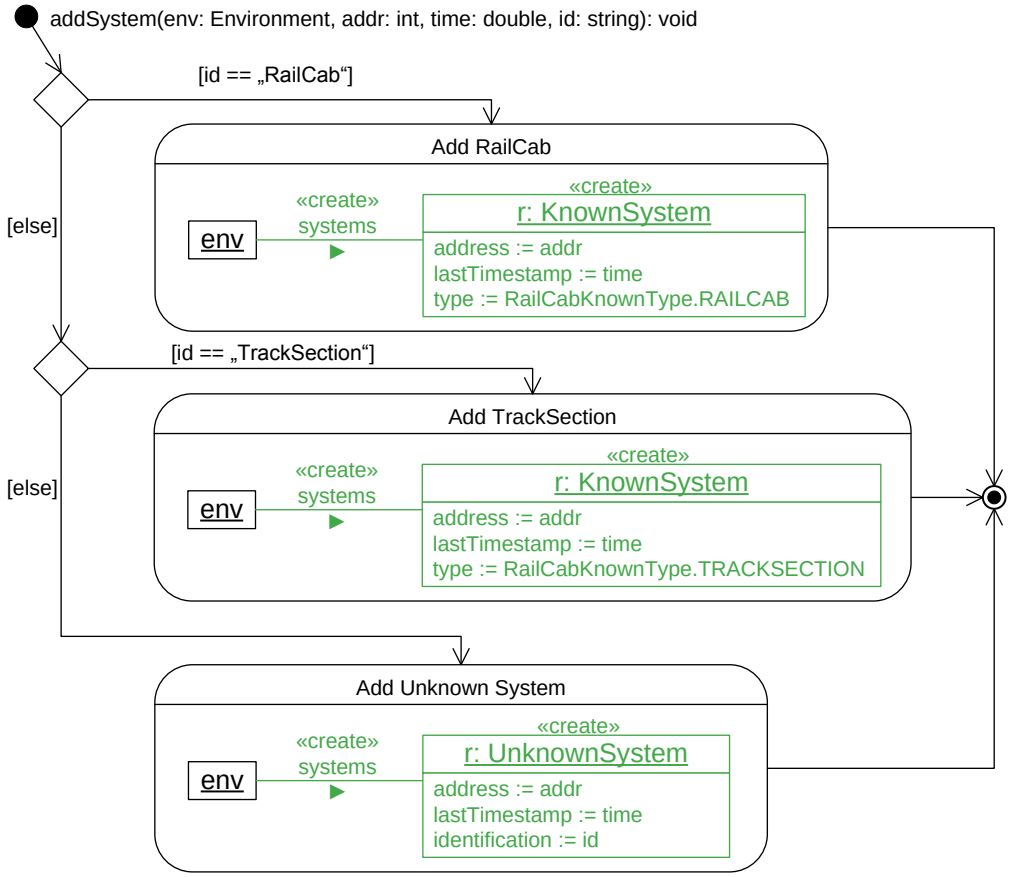


Figure A.21: Story Diagram Implementing the Operation addSystem

matches all ExternalSystems whose timestamp has not been updated during the last 10 minutes. These systems have not provided a systemInformation message for the last 10 minutes and are considered to be out of reach.

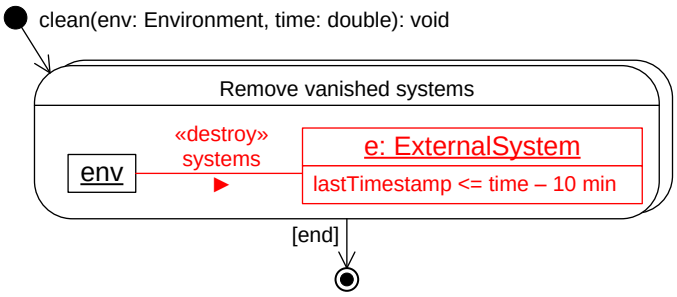


Figure A.22: Story Diagram Implementing the Operation clean

A.2.2 Instantiating the RTCP ProtocolInstantiation

Figure A.23 shows the RTSC of the broadcast port that is used for instantiating the RTCP ProtocolInstantiation. The RTSC implements the behavior described in Section 3.4.1. The RTSC has one state Broadcast with two regions named actor and reactor. The region actor contains the RTSC defining the behavior sys1 in Figure 3.16, i.e., of the system that initiates the instantiation. The region reactor contains the RTSC defining the behavior of sys2 in Figure 3.16, i.e., of the system that reacts to the instantiation request. The shared variable mutex and corresponding guards at the transitions in both regions ensure that at any point in time at most one of the two regions may execute.

In the following, we describe the behavior that is specified by the RTSC. Although we describe the interaction of the RTSCs in the two regions, of course these two regions never interact within the same broadcast port instance. The actor region of one broadcast port instance always communicates with a reactor region in another broadcast port instance.

Initially, both regions are in their Idle states. Then, the actor starts the interaction by sending a connectionRequest with the address of the intended communication partner and its own address as parameters. The address of the communication partner needs to be provided by the component implementing the broadcast port. We also assume that the component triggers the transition from Idle to Start in actor. This transition may only be fired if the variable mutex is false, i.e., the reactor is currently not engaged in an interaction. Upon firing, the transition sets mutex to true thereby indicating that it started an interaction.

The reactor receives the connectionRequest at the transition from Idle to CheckIDs. It stores the two addresses in the parameters in the variables tmpReceiverAddr and tmpSenderAddr. In CheckIDs, the reactor checks if it was the intended receiver of the message. If not or if mutex is true, the reactor returns to Idle without any further action. If the reactor is the intended receiver and if mutex is false, it switches to CheckRequest thereby setting mutex to true. If the maximum number of ports has been reached, reactor switches back to Idle and sends a connectionDenial. As for any message, it includes the address of the receiver as well as its own address as parameters. In addition, it sets mutex back to false because it stopped the interaction. If the maximum number of ports has not yet been reached, the reactor proceeds to ApproveRequest

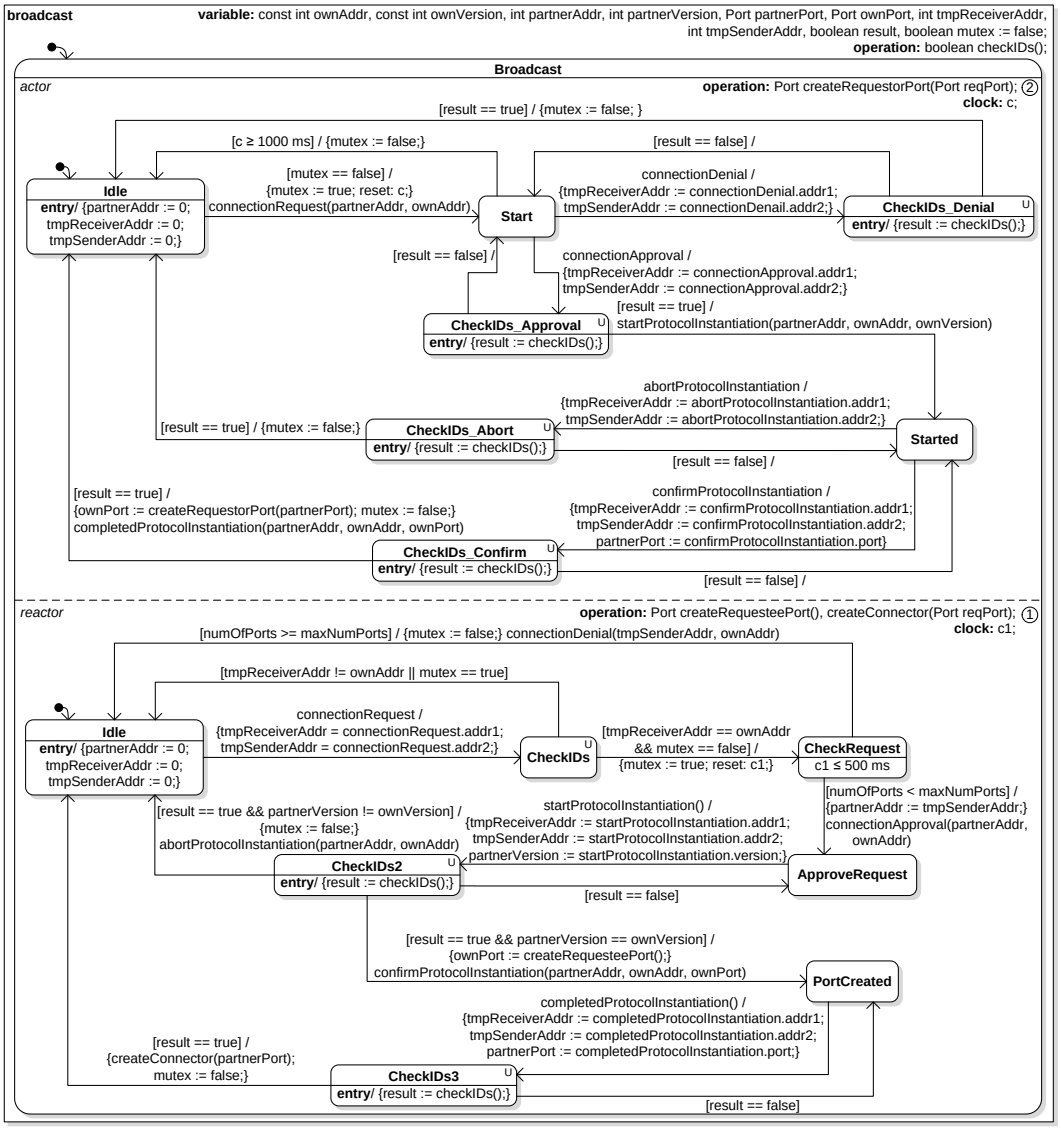


Figure A.23: RTSC Implementing the Broadcast Communication for Instantiating the RTCP ProtocolInstantiation

and sends a connectionApproval back to the actor. In addition, it stores the sender's address, which was temporarily stored in tmpSenderAddr, in the variable partnerAddr that denotes it communication partner for the remainder of the interaction.

If the actor receives a connectionDenial, it switches from Start to CheckIDs_Denial. Upon entering, it checks whether the message has been sent by the communication partner. If not, it returns to Start and waits for the message from the communication partner. If the message was from the communication partner, the actor switches back to Idle and sets mutex back to false thereby terminating the interaction. If actor receives a connectionApproval, it also checks the addresses. This time using the entry action in state CheckIDs_Approval. If the message has not been sent by the communication partner, then actor switches back to Start and waits for the message of the communication partner. Otherwise, actor proceeds to Started and sends a startProtocolInstantiation including its own protocol version to the communication partner.

If reactor receives the startProtocolInstantiation message, it switches to CheckIDs2. If the protocol version is not supported, reactor sends a abortProtocolInstantiation to the communication partner, switches back to Idle, and sets mutex back to false. Then, the interaction terminates. If the protocol version is supported by reactor, it proceeds to PortCreated. At this transition, the reactor creates a port instance implementing the requestee role of ProtocolInstantiation (cf. Section 3.4.2). This operation is a stub in the RTSC and needs to be replaced if the RTSC is integrated in a component. After the port instance has been created, the transition sends a confirmProtocolInstantiation message including the created port instance.

After receiving the confirmProtocolInstantiation messagen, the actor fires the transition from Started to CheckIDs_Confirm. If the message has been sent by the communication partner, the actor creates a port instance implementing the requestor role of ProtocolInstantiation including its virtual connector instance to the port instance partnerPort of the communication partner. After creating the port instance, actor sets mutex to false and sends a completedProtocolInstantiation message to the reactor. This message includes the newly created port instance. After sending this message, the actor returns to the Idle state and the interaction is finished.

The reactor waits in state PortCreated for the answer of the actor. After it has received the completedProtocolInstantiation message and confirmed that it has been sent by the communication partner, it creates its virtual connector instance to the port instance of the actor. Thereafter, it sets mutex back to false and returns to the Idle state. Now, the instantiation of ProtocolInstantiation is complete for both, actor and reactor.

A.2.3 The RTCP ProtocolInstantiation

Figures A.24 and A.25 show the RTSCs of the roles requestor and requestee of the RTCP ProtocolInstantiation introduced in Section 3.4.2. In particular, the RTSCs implement the behavior defined by the modal sequence diagram in Figure 3.18.

Initially, both RTSCs are in their Idle states. The requestor starts executing by firing the transition from Idle to Request where it choses a protocol and a role within the protocol that should be instantiated. On the protocol level, we use non-deterministic choice expressions. If the requestor role is refined to a port RTSC, then this transition needs to synchronize with the component behavior for defining the ID of the protocol to be instantiated. Thereafter, requestor sends a request with the protocol ID and the role ID to the requestee while it switches to SentRequest.

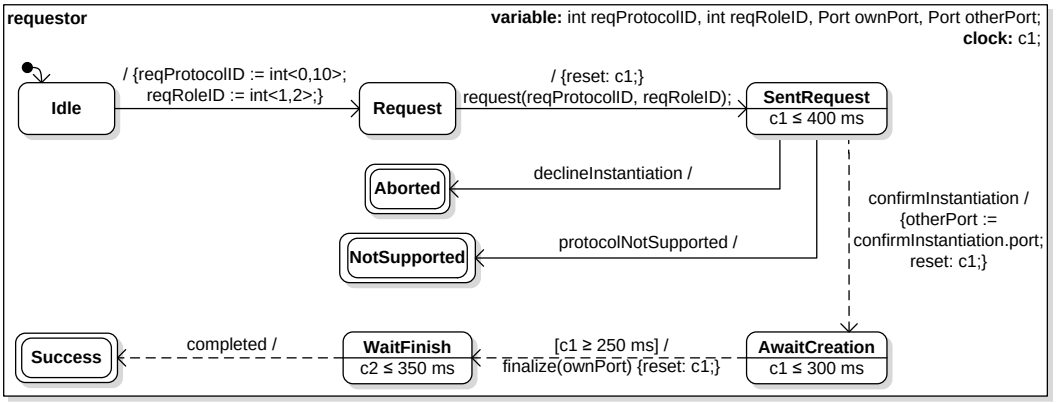


Figure A.24: RTSC Implementing the Role requestor of the RTCP ProtocolInstantiation

The requestee consumes the request at the transition from Idle to CheckRequest and resets c2. The entry action in state CheckRequest calls the operation isSupported with the requested protocol ID and role ID. Based on this information, the operation decides whether the requested protocol and role are supported by the requestee. This operation needs to be implemented for any port that refines the requestee role. This decision may take up to 50 ms as specified by the invariant. In the role RTSC, this operation is simply implemented by a non-deterministic choice. If the protocol or role are not supported, the requestee switches to Abort and sends protocolNotSupported to the requestor. In this case, the requestor switches from SentRequest to NotSupported. Since both RTSCs are now in final states, the interaction is terminated.

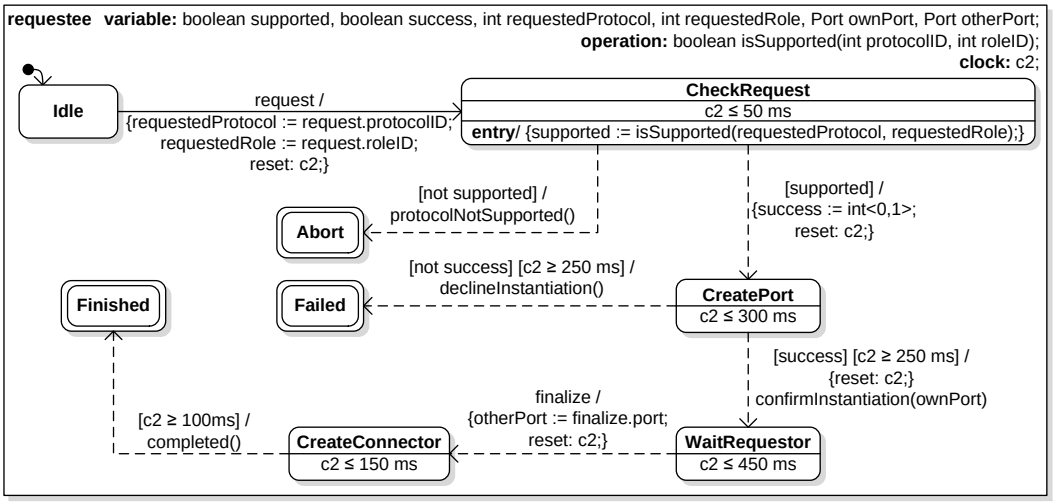


Figure A.25: RTSC Implementing the Role requestee of the RTCP ProtocolInstantiation

If the requested protocol and role are supported by the requestee, it fires the transition from CheckRequest to CreatePort. This transition shall initiate the creation of the corresponding port instance implementing the requested role of the requested protocol. Thus, this transition needs to be refined by a port RTSC and needs to be integrated with the reconfiguration

behavior of the component. In the role RTSC of requestee, the success of the reconfiguration is, again, realized by a non-deterministic choice. If the creation was not successful, the requestee switches to Failed and sends declineInstantiation. In this case, the requestor switches from SentRequest to Aborted. Since both RTSCs are now in final states, the interaction is terminated.

If the creation of the port instance has been successful, the requestee switches from CreatePort to WaitRequestor. Thereby, it sends a confirmInstantiation message to the requestor and resets c2. Then, it waits for 450 ms for the reply of the requestor. The requestor processes the confirmInstantiation message at the transition from SentRequest to AwaitCreation. At this transition, the requestor triggers the instantiation of the port instance implementing the other role of the requested protocol. In a port that refines this role, this transition needs to be refined such that it may trigger the actual reconfiguration. After the port has been created, the requestor switches to WaitFinish and sends a finalize message to the requestee.

The requestee receives the finalize message at the transition from WaitRequestor to CreateConnector. This state and transition need to be refined by a port such that they trigger the creation of the (virtual) connector instance to the port instance created by the requestor. After the connector instance has been created, the requestee switches to Finished and sends completed to the requestor. The requestor finally switches from WaitFinish to Success after receiving this message. Then, both RTSCs are in final states and it interaction terminates with success.

A.3 Components

We have introduced all but one component of our RailCab example in the main chapters of this thesis. In particular, we use three structured components for the RailCab itself. These are RailCabDriveControl shown in Figure 3.6, ConvoyCoordination shown in Figure 3.5, and VelocityController shown in Figure 3.7. We do not repeat the component definitions in this section. In addition, we use five discrete atomic components, five continuous atomic component, and one fading component for the RailCab. These are all contained in the three structured components and will not be presented in the section, again. Finally, we defined three components for the different kinds of track sections in Figure 5.3 on Page 116. Of these components, NormalTrackSection and Switch are atomic components whereas RailroadCrossing is a hybrid structured component. We introduce RailroadCrossing in more detail in Section A.3.1.

A.3.1 RailroadCrossing

The component RailroadCrossing is a hybrid structured component as shown in Figure A.26. It contains one component part infProcessing of type Crossing_InfProf and one component part gates of type Gates. The former is a discrete atomic component that implements the communication with the RailCabs via ports left and right and, if necessary, with a preceding switch via port precedingSwitch. The latter component part refers to a continuous atomic component that controls the gates of the crossing. Both component parts are connected such that infProcessing may advise gates to open or close the gates via the hybrid port gateAction. In addition, gates provides the current state of the gates (either open or closed) via status. Both continuous ports are Boolean-valued.

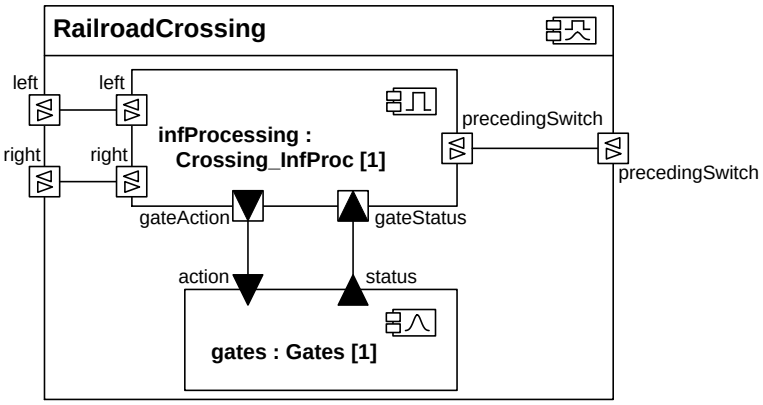


Figure A.26: Structured Component RailroadCrossing

A.4 Component Instances

In this section, we introduce additional component instances for the RailCab examples. We have already introduced the component instance standaloneRC in Figure 3.9 on Page 3.9, which we will not repeat in this section. In the following, we present component instances for a RailCab driving as a coordinator (Section A.4.1) and for a RailCab driving as a member (Section A.4.2).

A.4.1 RailCab Driving as a Coordinator

Figure A.27 shows an instance of RailCabDriveControl for a coordinator RailCab that coordinates a convoy with one member. As the main difference to the component instance standaloneRC shown in Figure 3.9, Coordinator has an instance cc of type ConvoyCoordination that is attached to an instance ps of the PositionSensor. Furthermore, Coordinator has instances of the coordinator and refDistProvider multi ports for communicating with the member.

In a coordinator RailCab, the instance os of OperationStrategy is no longer directly connected to dl of type DriveLogic. As a result, the operation strategy does no longer directly determine the reference speed of the RailCab. Instead, the reference speed determined by os is passed to cc which uses this value as a basis for defining a reference speed for the convoy. Then, cc sends the reference speed that it defined for the convoy to dl.

Figure A.28 shows the inner structure of the instance vc of type VelocityController that is embedded in Coordinator. vc1 executes an instance of StandaloneDrive that is connected to the instance f of type ConvoyFading.

Figure A.29 shows the inner structure of the component instance cc of type ConvoyCoordination that is embedded in Coordinator. It is used for computing reference data for a convoy with one member. Therefore, it contains an instance cm of type ConvoyManagement and, since the convoy has one member, one instance rg1 of type RefGen. Since rg1 is associated with the first convoy member, it receives the current position of the coordinator RailCab via curPos. It is connected to cm for receiving the current profile of the member.

Figure A.30 shows an instance cc2 of type ConvoyCoordination that is used for a coordinator RailCab with two members. Compared to cc shown in Figure A.29, it contains one additional

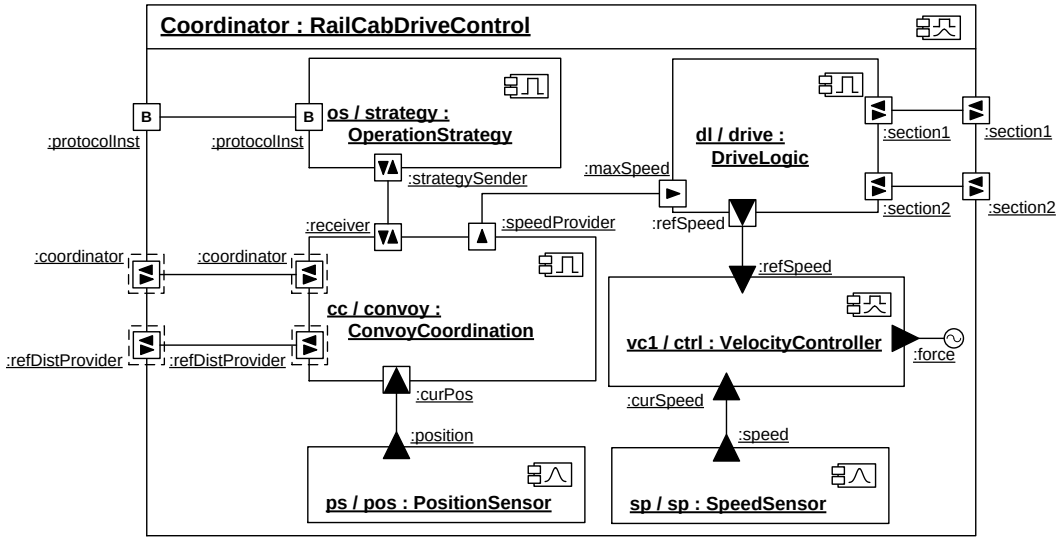


Figure A.27: Component Instance of Component RailCabDriveControl for a Coordinator RailCab

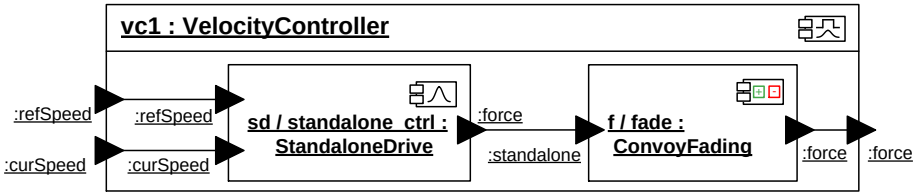


Figure A.28: Component Instance of Component VelocityController that is used by a Coordinator RailCab

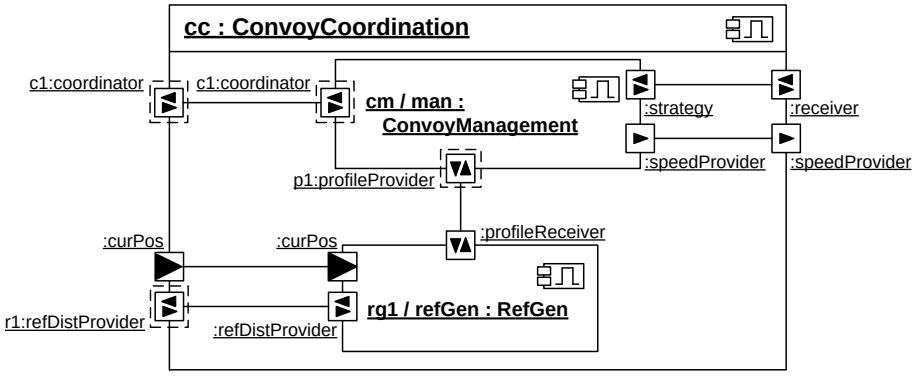


Figure A.29: Component Instance of Component ConvoyCoordination for a Convoy with 1 Member

instance rg2 of type RefGen including one additional subport instance for each multi port instance.

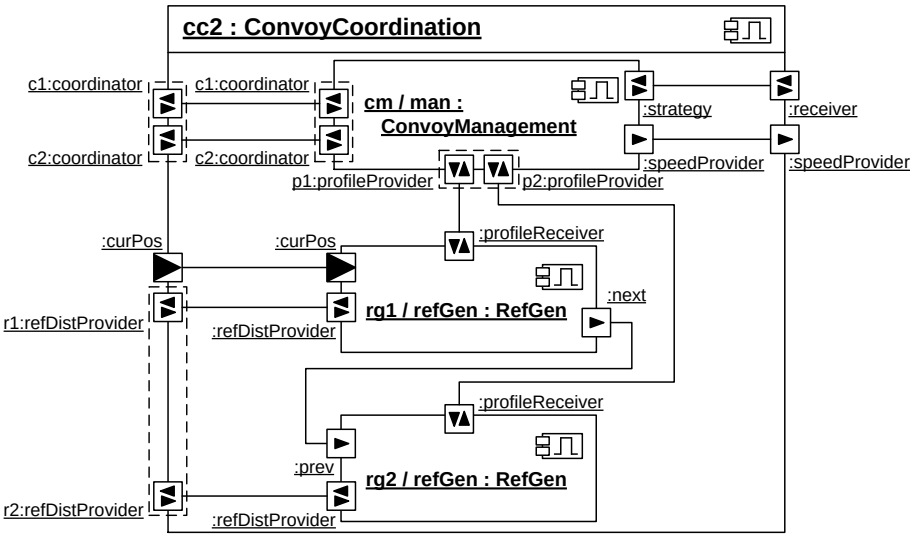


Figure A.30: Component Instance of Component ConvoyCoordination for a Convoy with 2 Members

In particular, Figure A.30 illustrates how the RefGen instances are arranged in a sequence. rg1 generates reference data for the first member that drives directly behind the coordinator. rg2 generates reference data for the member driving at the last position in the convoy. Then, rg1 and rg2 are connected via their next and prev port instances, thereby defining a sequence. Thus, the reference data that is calculated for a RailCab depends on the reference data of the RailCab driving in front of it.

A.4.2 RailCab Driving as a Member

Figure A.31 shows the CIC of RailCabDriveControl for a convoy member. The main difference to standaloneRC shown in Figure 3.9 is that the member has an instance of MemberControl for communicating with the coordinator. In addition, os is disconnected from the remaining components because a convoy member may not voluntarily decide upon a new speed. This information is solely provided by the coordinator and received by the member via the refDistReceiver port instance. The reference speed is then send to dl via the speedProvider port instance of mc.

In addition, the convoy member uses a different VelocityController vc2 that controls the speed of the RailCab based on speed and distance to the preceding RailCab. As a result, Member has an instance ds of type DistanceSensor to obtain the current distance that is provided to vc2.

Figure A.32 shows the corresponding instance vc2 of type VelocityController that is used by Member. vc2 embeds an instance cd of type ConvoyDrive that is connected to the instance f of type ConvoyFading.

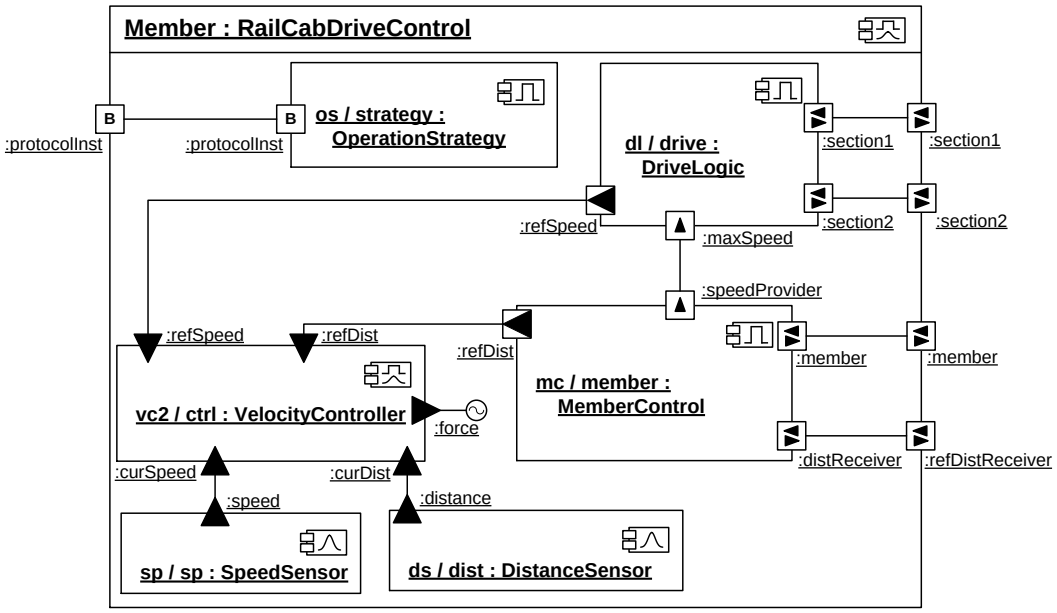


Figure A.31: Component Instance of Component RailCabDriveControl for a Member RailCab

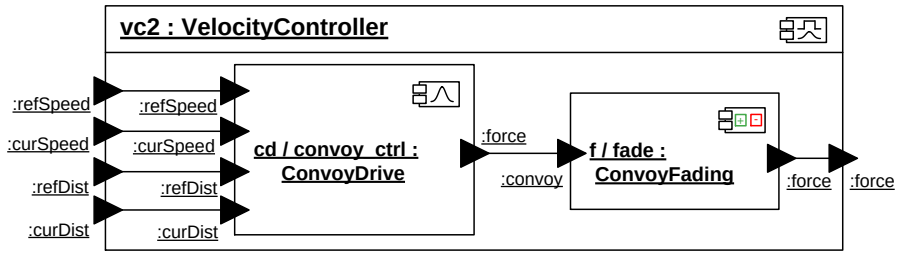


Figure A.32: Component Instance of Component VelocityController that is used by a Member RailCab

A.5 Component RTSCs

This section introduces the component RTSCs that we created for the discrete atomic components that we used in the RailCab example. The components for the RailCab itself have been introduced in Section 3.1. We present their RTSCs in Section A.5.1. The components for the different kinds of track sections have been introduced in Section 5.1.2. We present their RTSCs in Section A.5.2. For each of the component RTSCs, we focus on describing the internal behavior regions and the differences of the port RTSCs to their corresponding role RTSCs.

A.5.1 RTSCs of the RailCab Components

In this section, we present the component RTSCs of the discrete atomic components that are (recursively) contained in RailCabDriveControl. In particular, we present the component RTSCs of OperationStrategy (Section A.5.1.1), DriveLogic (Section A.5.1.2), MemberControl (Section A.5.1.3), ConvoyManagement (Section A.5.1.4), and RefGen (Section A.5.1.5).

A.5.1.1 OperationStrategy

The component RTSC of OperationStrategy is shown in Figures A.33 and A.34. OperationStrategy is embedded in RailCabDriveControl (cf. Figure 3.6 on Page 43) and is responsible for negotiating the convoy entry of further RailCabs and for setting the operation strategy. The RTSC has eight regions; five of which embed the port RTSCs of the five discrete ports of OperationStrategy, one embeds the RTSC of the broadcast port, and two embed the RTSCs of the RM and RE port.

The region reconfMsg contains the RTSC of the RM port. It has been derived manually from the parent region of the manager RTSC generation template shown in Figure 4.15 and satisfies the message exchange for the 2-phase-commit protocol. However, it does not yet incorporate a decision whether the reconfiguration shall be executed or not. The RTSC supports sending the three reconfiguration messages that appear in the RM port interface specification shown in Figure A.49. All of which are requests and, thus, the RTSC waits for a reply by the parent in AwaitReply. It synchronizes via started with the peer region and uses the selector expression of type Boolean to indicate the result of the request.

The region reconfExec contains the RTSC of the RE port. It has been specified manually and satisfies the message exchange for the 2-phase-commit protocol. The specified behavior, however, is very simple such that any request by the parent will be executed. The RTSC contains four states CheckApplyCoordinationStrategy, CheckApplyMemberStrategy, CheckDisableConvoyBuildUp, and CheckEnableConvoyBuildUp that are reached from Idle by receiving one of the messages that are offered by the RE port interface specification shown in Figure A.50. We manually defined a unique reconfID four each of the corresponding reconfiguration operations in accordance to the executor RTSC generation template (cf. Figure 4.16). In our example, the reconfExec RTSC checks the structural condition for executing the reconfigurations itself at the transitions from the Check* states to the Checked state. In particular, applyCoordinationStrategy (or applyMemberStrategy) may be executed if the RailCab is not in member mode (or in coordinator mode) as defined by the component SDD in Figure A.91 (or the component SDD in Figure A.90, respectively). Finally, the RTSC returns to Idle either by receiving abort

from the parent or by receiving execute from the parent. In the latter case, the reconfID is used to define the transition that is fired and that executes the corresponding CSD.

The region broadcast contains the RTSC of the broadcast port. It is almost unchanged compared to Figure A.23 and, therefore, we only show a small excerpt of the broadcast port RTSC in Figure A.33. The only change is that we inserted a state TriggerReq in the actor region of the Broadcast state. The transition from TriggerReq to Idle synchronizes with the RTSC of the requestor port such that the instantiation proceeds after the requestor port has been created by the broadcast RTSC using the CSD createRequestorPort shown in Figure A.64, which is called at the transition from CheckIDs_Confirm to TriggerReq.

The region requestor contains the port RTSC of the requestor port. The port RTSC has been adapted as follows compared to the role RTSC shown in Figure A.24. First, the transition from Idle to Request now waits for the synchronization startInstantiation initiated by the broadcast port. Second, we replaced the non-deterministic choice expressions by assigning the IDs of ConvoyEntry and its peer role to the variables reqProtocolID and reqRoleID because we only enable to instantiate the peer port of OperationStrategy via the RTCP ProtocolInstantiation. Finally, we moved calling the createPeerPort CSD shown in Figure A.66 to the entry action of AwaitCreation.

The region requestee contains the port RTSC of the requestee port. The RTSC is unchanged compared to the role RTSC shown in Figure A.25 except that we replaced the non-deterministic choice expression at the transition from CheckRequest to CreatePort by a call to the CSD createPeerPort shown in Figure A.66. Therefore, we omit the RTSC in Figure A.34.

The region peer contains the RTSC of the peer port. Compared to the role RTSC shown in Figure A.2, we added a new initial state Init. The transition from Init to Idle initializes the variables masterPossible, isMaster, and slavePossible by evaluating the component SDDs inMemberMode and inCoordinatorMode shown in Figures A.91 and A.90, respectively. These variables define whether the RailCab may become coordinator or member of the convoy. After an initial assignment has been proposed by switching to the MasterProposed and AcceptSlave states, the peer region synchronizes with the reconfMsg region via becomeCoord and becomeMember for executing the reconfigurations for becoming a coordinator or member of a convoy. The transitions from StartingMaster to WaitForSlaveFinish now wait for the answer of reconfMsg using the synchronization channel started. The same holds for the transitions from StartingSlave to WaitForMasterFinish. Thereby, we integrated the peer port with the reconfiguration behavior that executes the requested reconfigurations based on the 2-phase-commit protocol.

Finally, regions speedProvider and strategySender contain the port RTSCs of the ports speedProvider and strategySender. Both RTSCs are unchanged compared to their port RTSCs shown in Figures A.10 and A.14 except that they access the variables of the component RTSC instead of their local variables. Therefore, we omit the RTSCs in Figure A.34.

A.5.1.2 DriveLogic

The component RTSC of DriveLogic is shown in Figure A.35. DriveLogic is embedded in RailCabDriveControl (cf. Figure 3.6 on Page 43) and is responsible for setting the speed of the RailCab and for requesting permission to enter track sections. The component RTSC contains three regions that correspond to the three discrete ports of DriveLogic. Since DriveLogic is not reconfigurable, it has no RM port and no RE port.

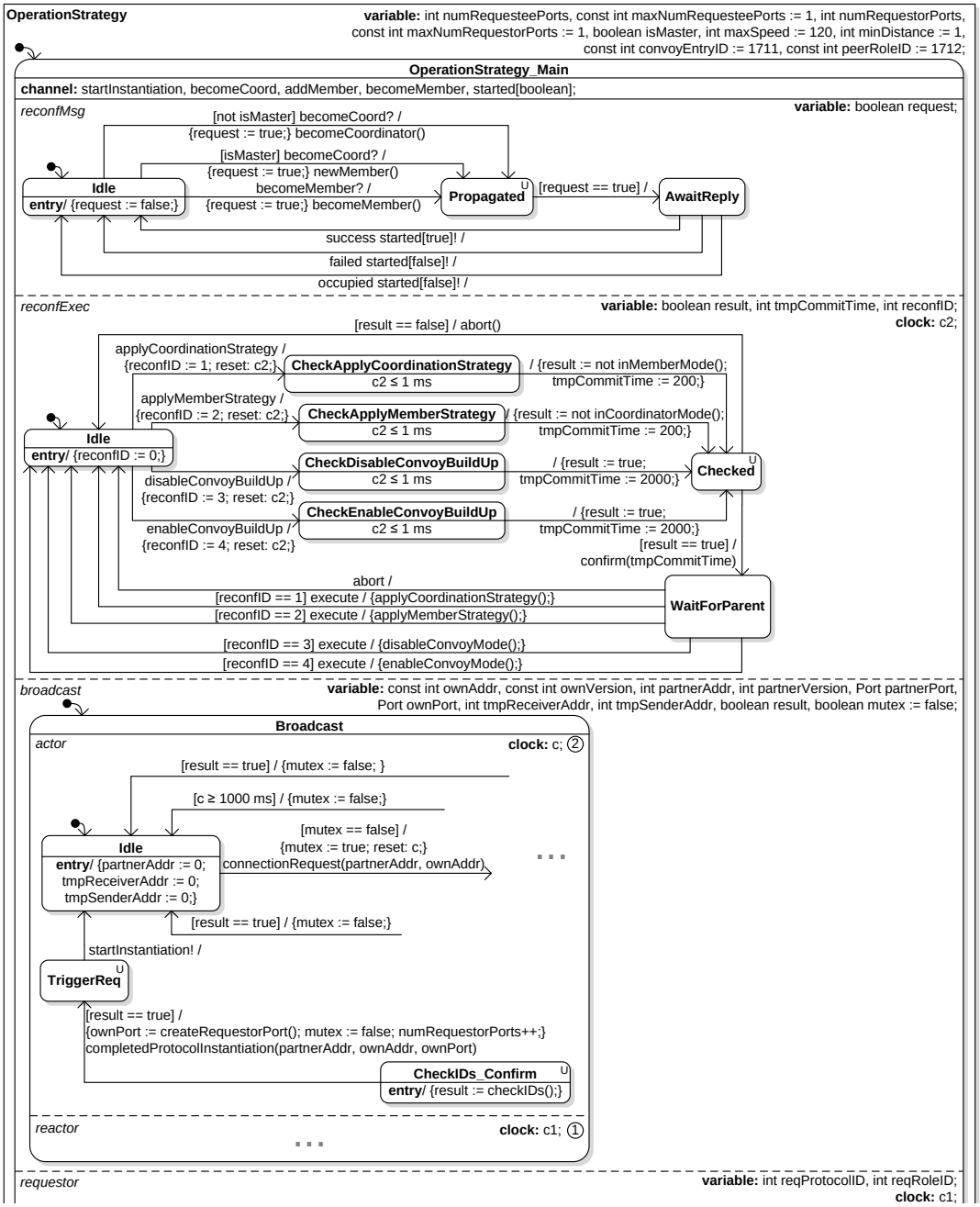


Figure A.33: RTSC of the Component OperationStrategy (Pt. 1)

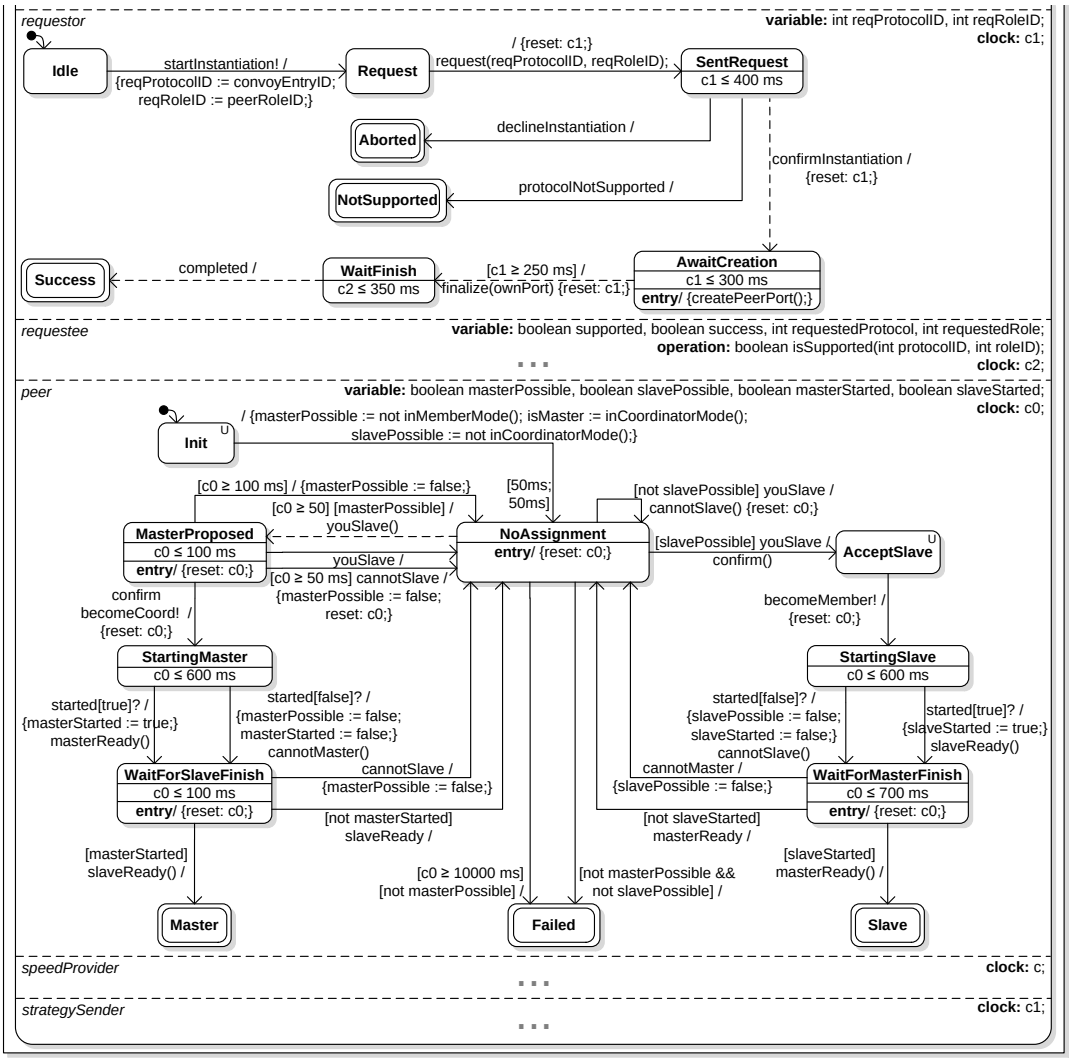


Figure A.34: RTSC of the Component OperationStrategy (Pt. 2)

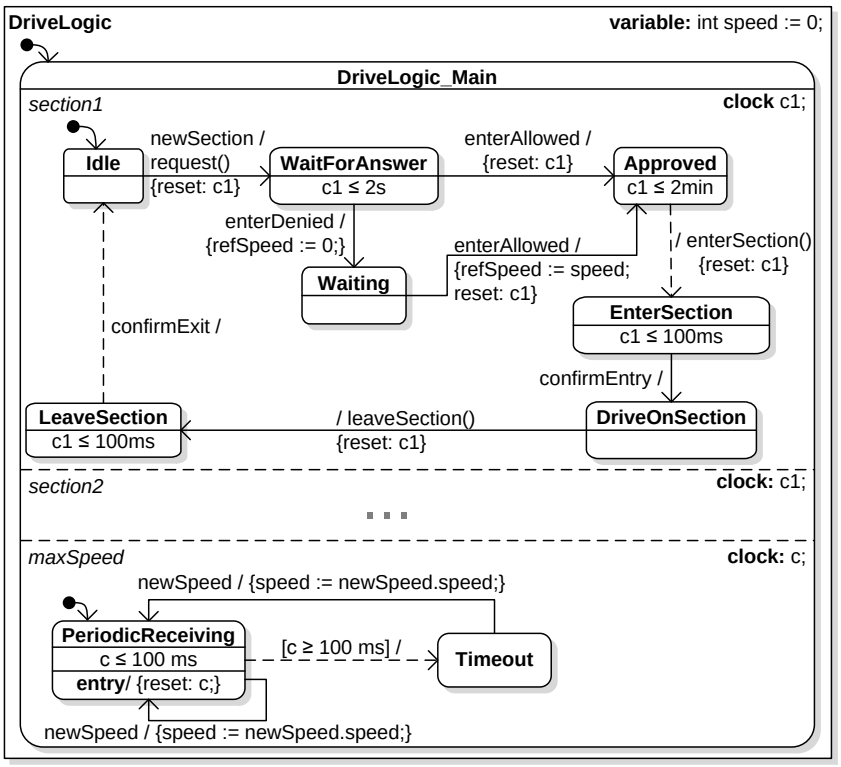


Figure A.35: RTSC of the Component DriveLogic

The region section1 contains the port RTSC of section1 that refines the railcab role of SectionEntry. Compared to the role RTSC shown in Figure 5.2 on Page 114, we only applied two changes. First, whenever the RailCab is denied to enter a track section (transition from WaitForAnswer to Waiting), we set the value of the hybrid port refSpeed to 0 such that the RailCab stops. If the RailCab eventually receives permission to enter the track section and switches from Waiting to Approved, we set refSpeed back to the current maximum speed of the RailCab.

The RTSC of section2 contained in region section2 is refined in the same way and, therefore, omitted in Figure A.35.

Finally, the region maxSpeed contains the port RTSC of the maxSpeed port. It is responsible for receiving the current maximum speed for the RailCab and for storing it in the variable speed.

A.5.1.3 MemberControl

The component RTSC of MemberControl is shown in Figure A.36. MemberControl is embedded in RailCabDriveControl (cf. Figure 3.6 on Page 43) and is responsible for executing the behavior that is necessary for driving in a convoy as a member. The component RTSC contains four regions; three of which correspond to the three discrete ports of MemberControl while the fourth one contains internal behavior. Since MemberControl is not reconfigurable, it has no RM port and no RE port.

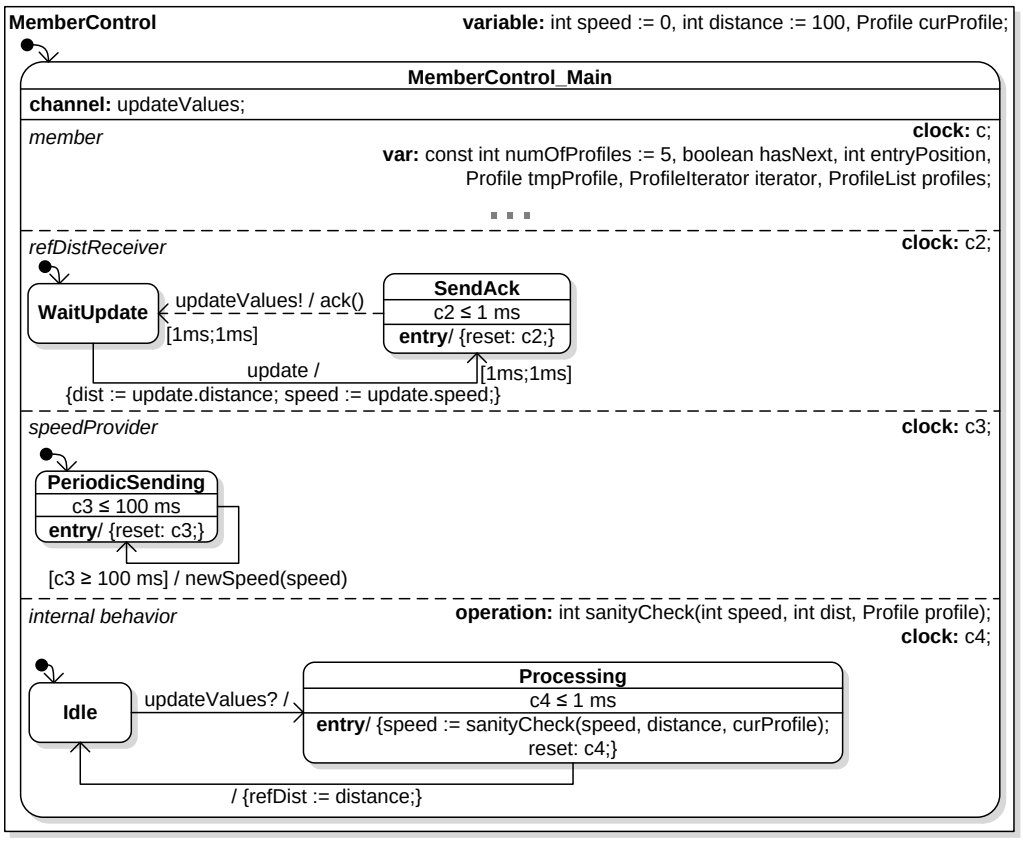


Figure A.36: RTSC of the Component MemberControl

The region member contains the RTSC of the member port that refines the role RTSC of the member role of ConvoyCoordination shown in Figure A.5. The RTSC is identical to the role RTSC and, therefore, is omitted from the figure.

The region refDistReceiver contains the RTSC of the refDistReceiver port that refines the role RTSC of the receiver role of DistanceTransmission shown in Figure 2.14. After the RTSC received an update from the coordinator, it stores the new reference speed and distance in the variables speed and distance of the component RTSC. Before sending the ack, it synchronizes with the internal behavior region via updateValues to indicate that new reference values are available.

The region speedProvider contains the RTSC of the speedProvider port that refines the role RTSC of the sender role of SpeedTransmission shown in Figure A.10. It periodically sends the value of the variable speed as a parameter of the newSpeed message.

Finally, the region internal behavior contains an internal behavior of DriveLogic. The RTSC waits in Idle until refDistReceiver initiates the synchronization via updateValues. Then, the RTSC switches to Processing and calls the operation sanityCheck in its entry action. sanityCheck checks the reference values that have been provided by the coordinator based on the current profile. The reference values are adjusted if necessary. In addition, the transition from Processing back to Idle sets the value of distance to the hybrid port refDist.

A.5.1.4 ConvoyManagement

The component RTSC of ConvoyManagement is shown in Figure A.37. ConvoyManagement is embedded in ConvoyCoordination (cf. Figure 3.5 on Page 42) and is responsible for managing the members of a convoy. The RTSC has six regions; four of which embed the port RTSCs of the four discrete ports of ConvoyManagement, two of which embed the RTSCs of the RM and RE port.

The region reconfMsg contains the RTSC of the RM port. It has been derived manually from the parent region of the manager RTSC generation template shown in Figure 4.15 and satisfies the message exchange for the 2-phase-commit protocol. However, it does not yet incorporate a decision whether the reconfiguration shall be executed or not. The RTSC supports sending the stopCoordination message that is defined in the RM port interface specification shown in Figure A.51. Sending this message is triggered by a synchronization via stopCoordination that is initiated by the RTSC in the coordinator region.

The region reconfExec contains the RTSC of the RE port. It has been specified manually and satisfies the message exchange for the 2-phase-commit protocol. The specified behavior, however, is very simple such that any request from the parent will be executed. The RTSC contains two states CheckCreateFirstMemberPorts and CheckCreateMemberPortsAfter that are reached from Idle by receiving one of the messages that are offered by the RE port interface specification shown in Figure A.52. We manually defined a unique reconfID for each of the corresponding reconfiguration operations in accordance to the executor RTSC generation template (cf. Figure 4.16). In our example, the reconfExec RTSC checks the structural condition for executing the reconfigurations itself at the transitions from the Check* states to the Checked state. In particular, executing both reconfigurations is possible if the current number of members is less than maxNumMembers indicating the maximum number of convoy members. Finally, the RTSC returns to Idle either by receiving abort from the parent or by receiving execute from the parent. In the latter case, the reconfID is used to define the transition that is fired and that executes the corresponding CSD. In addition, both of these transitions increase the number of members of the convoy.

The region speedProvider contains the RTSC of the speedProvider port that refines the RTSC of the role provider of SpeedTransmission shown in Figure A.10. It has not been modified and, therefore, we omit it is the figure.

The region coordinator contains the RTSC of the coordinator port that refines the RTSC of the role coordinator of ConvoyCoordination shown in Figure A.4. Most of the RTSC are unchanged with respect to the role RTSC and, therefore, we only show a small excerpt in Figure A.37. We applied the following changes. First, the transition from Idle to HandleNewMember does no longer check the condition for executing a reconfiguration and for executing the reconfiguration rule. Instead, it synchronizes via newMember with the reconfExec region, which now contains the behavior for executing the reconfiguration. Then, this transition will fire any time after a new subport has been created by reconfExec. As our second modification, we inserted the CheckCoordination state. After the entry of the new member has failed, the transitions from CheckCoordination to Idle check whether there still exists a member in the convoy (members is greater than 0). If not, then the RTSC synchronizes via stopCoordination with the reconfMsg region such that a request for stopping the coordination of a convoy is sent. Currently, this request is not further processed in our behavior model because we only consider building convoys yet. The remaining behavior of coordinator is unchanged except

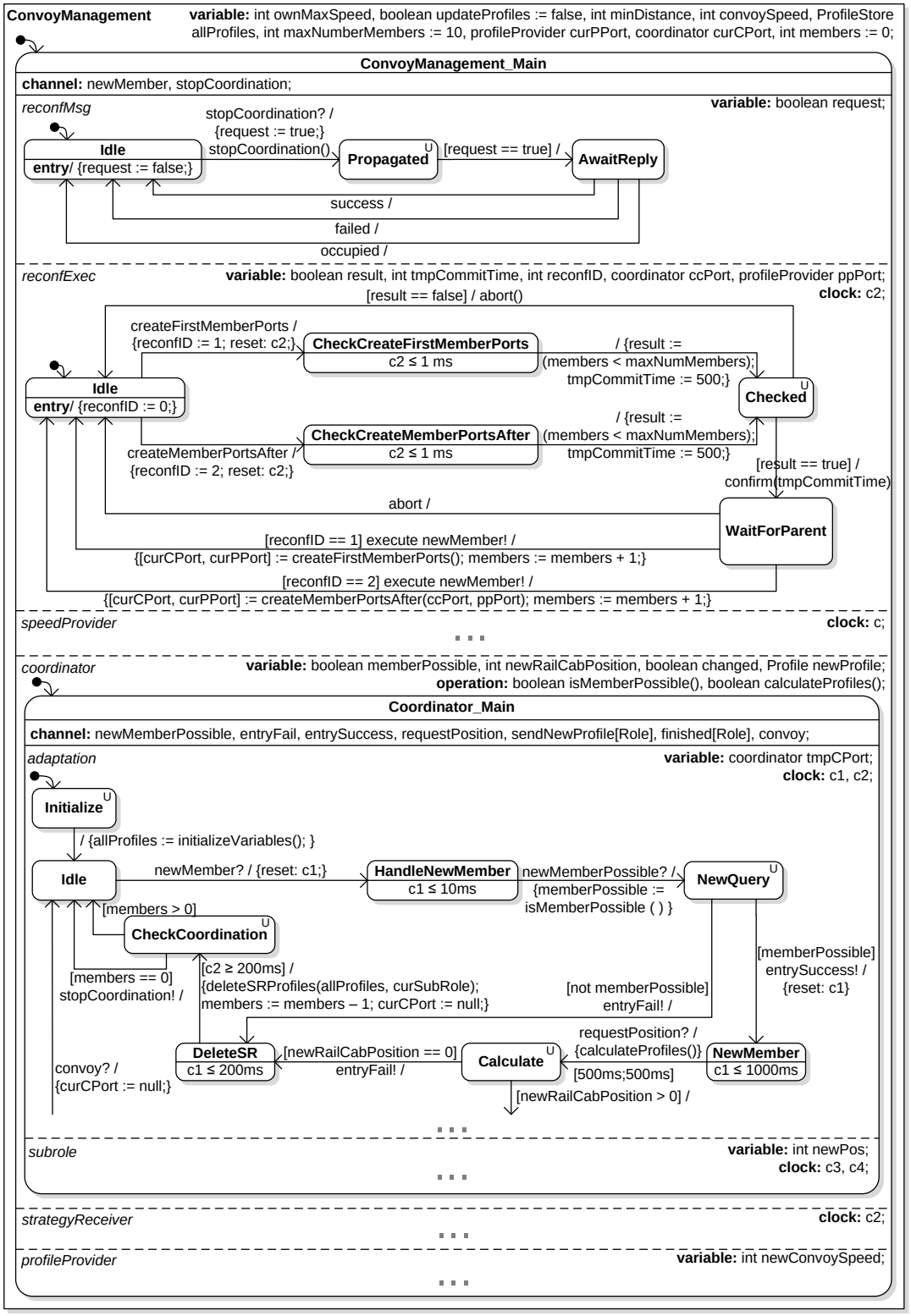


Figure A.37: RTSC of the Component ConvoyManagement

that we use variables that are typed by the coordinator port instead of variables typed by the role while iterating all subports of coordinator.

Finally, the regions strategyReceiver and profileProvider contain the RTSCs of the strategyReceiver and profileProvider ports. The RTSC of strategyReceiver is unchanged compared to the role RTSC shown in Figure A.15. The RTSC of profileProvider is unchanged compared to the role RTSC shown in Figure A.7 except that we removed the self-transition from the Idle state of the adaptation RTSC. The reason is that the reconfiguration executed by this transition is now contained in the reconfExec RTSC.

A.5.1.5 RefGen

The component RTSC of RefGen is shown in Figure A.38. ConvoyManagement is embedded in ConvoyCoordination (cf. Figure 3.5 on Page 42) and is responsible for computing reference values for the convoy members based on their profiles. The component RTSC contains five regions; four of which correspond to the four discrete ports of RefGen while the fifth one contains internal behavior. Since RefGen is not reconfigurable, it has no RM port and no RE port.

The region refDistProvider contains the RTSC of the refDistProvider port that refines the role provider of DistanceTransmission shown in Figure 2.15. Since refDistProvider is a single port, the port RTSC only refines the subrole behavior of the multi role provider. The creation of new RefGen instances, which corresponds to the creation of a new subrole instance in the role RTSC, is implemented by the parent component ConvoyCoordination. The synchronization behavior that triggers sending new reference values periodically is implemented by the internal behavior region as well as the next and prev port RTSCs.

The behavior is as follows. The internal behavior starts in the state InitUpdates. Every 500 ms, it fires the transition to UpdateVars and checks whether it is the first or the last RefGen instance in the sequence of RefGen instances inside ConvoyCoordination using the component SDDs isFirst and isLast given in Section A.7.5. If it is not the first instance, it returns to InitUpdates without action. If it is the first instance, it triggers the refDistProvider RTSC via the synchronization channel send. Then, the refDistProvider performs the update similar to the subrole RTSC of the provider role. However, it uses the curProfile and, if it is the first instance, the value of curPos for computing new reference values. Finally, it waits in AwaitAck for the ack of the member. If it is the last instance, it returns to Idle without further action after receiving the ack. If it is not the last instance, it triggers the next region via send_next. The RTSC in region next refines the role initiator of StartExecution shown in Figure A.12. Thus, it sends a startExecution message to the prev port of the next RefGen instance in the sequence. Upon receiving this message, the prev region synchronizes via send with the refDistProvider region and the behavior proceeds as described above.

Finally, the profileReceiver region contains the RTSC of the profileReceiver port that refines the role profileReceiver of ProfileDistribution shown in Figure A.8. It is responsible for receiving the profile that has been selected for the corresponding convoy member by ConvoyManagement. The RTSC is identical to the role RTSC except that all variables of the role became variables of the component RTSC. Therefore, the RTSC is omitted in Figure A.38.

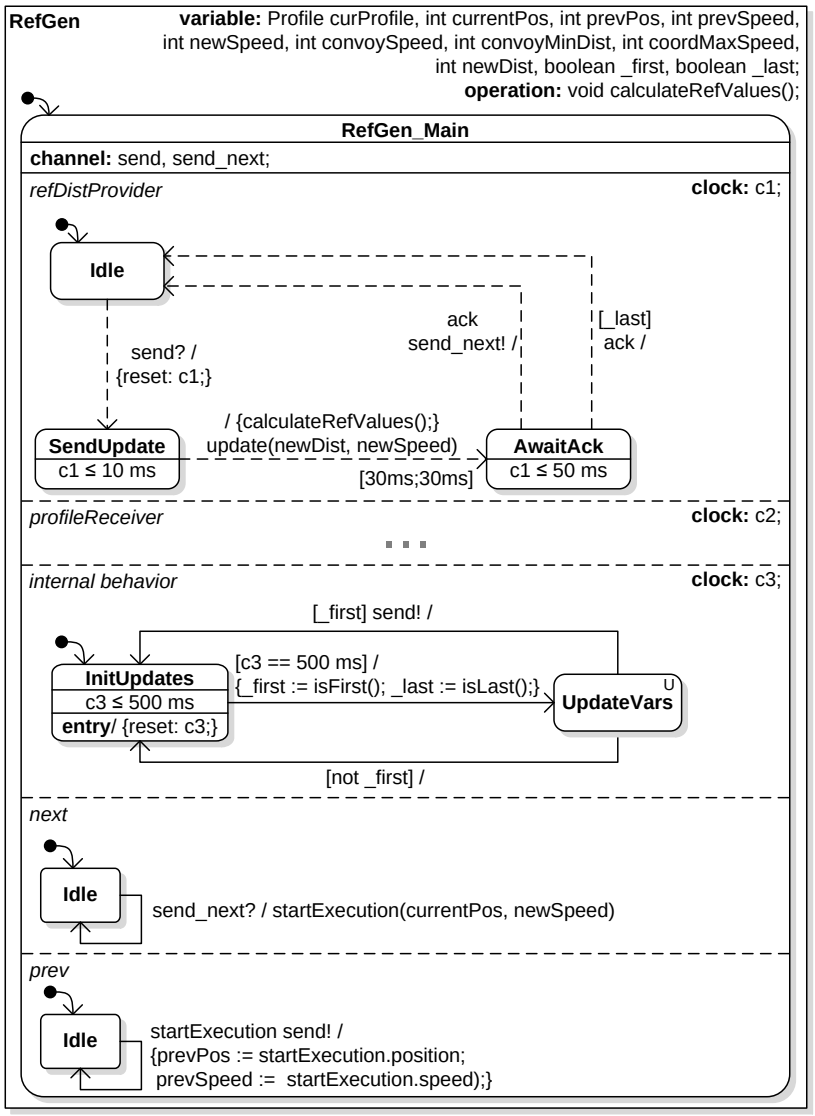


Figure A.38: RTSC of the Component RefGen

A.5.2 RTSCs of the Section Components

In this section, we present the component RTSCs of the components NormalTrackSection (Section A.5.2.1), Switch (Section A.5.2.2), and RailroadCrossing (Section A.5.2.3) that were used in Chapter 5 for illustrating our refinement check.

A.5.2.1 NormalTrackSection

Figure A.39 shows the RTSC of the component NormalTrackSection shown in Figure 5.3a on Page 116. The component RTSC defines one state NormalTrackSection_Main and one variable sectionFree that denotes whether the section is currently free or not.

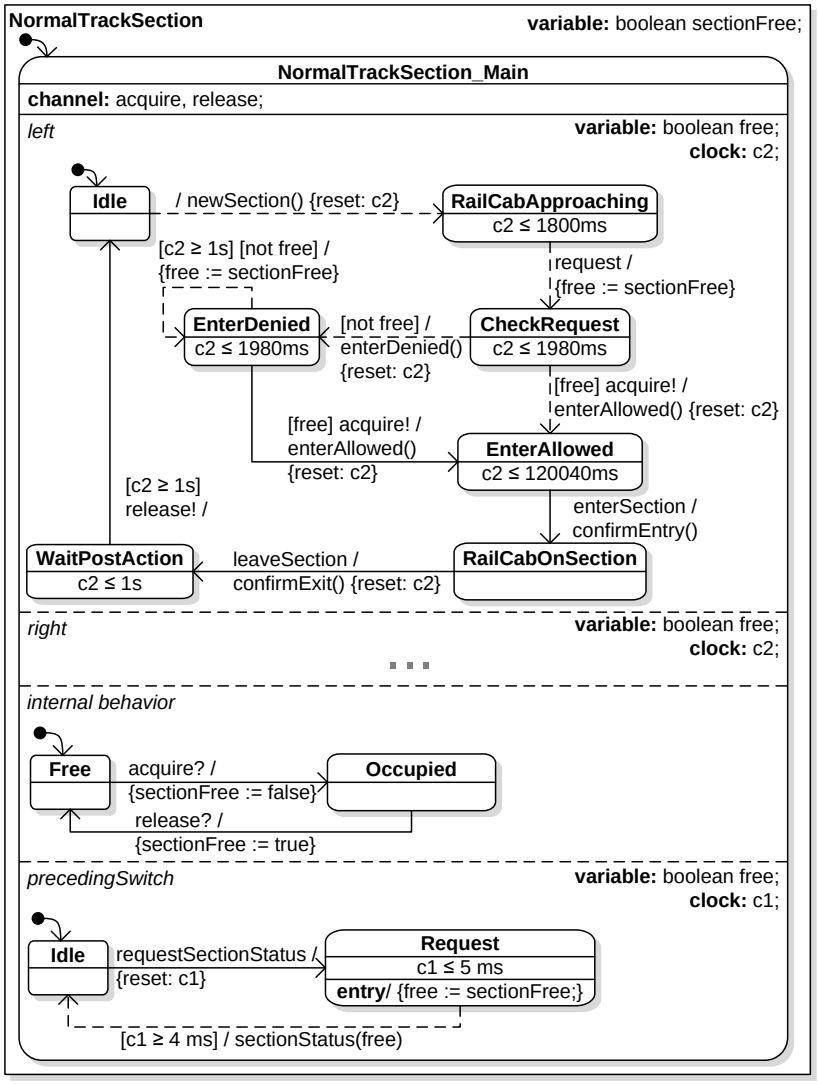


Figure A.39: RTSC of the Component NormalTrackSection

The state NormalTrackSection_Main has four regions. The regions left and right contain the port RTSCs of the ports left and right that both refine the role section of the RTCP EnterSection.

The RTSCs in both regions are identical to the RTSC shown in Figure 5.4 on Page 116. Therefore, we omitted the RTSC for right to improve readability of the figure.

The region internal behavior defines the internal behavior of the component. It consists of two states Free and Occupied that denote the current status of the track section. In our example, only the internal behavior may write the variable sectionFree. When the internal behavior switches from Free to Occupied, it sets sectionFree to false. The transition in the opposite direction sets the value of sectionFree back to true. The internal behavior synchronizes via the channels acquire and release with the RTSCs of left and right. In particular, left and right use acquire to block the section for the RailCab they are communicating with. Only if the synchronization via acquire succeeds, one of these RTSCs may send enterAllowed to a RailCab at the transition from CheckRequest to EnterAllowed. After the RailCab left the track section, left or right synchronizes via release to report that the track section is free again.

Finally, the region precedingSwitch implements the role tracksection of the RTCP NextSectionFree. The RTSC is almost identical to the RTSC shown in Figure A.17. The only difference is that the entry action of Request now assigns the value of the variable sectionFree to the local variable free.

A.5.2.2 Switch

Figure A.40 shows the RTSC of the component Switch shown in Figure 5.3c on Page 116. The component RTSC defines one state Switch_Main and one variable sectionFree that denotes whether the switch is currently free or not.

The state Switch_Main has five regions. The regions left, right, and bottom contain the port RTSCs of the ports left, right, and bottom that all refine the role section of the RTCP EnterSection. The RTSCs in these regions are identical to the RTSC shown in Figure 5.6 on Page 118. Therefore, we omitted the RTSCs for right and bottom to improve readability of the figure.

The region internal behavior defines the internal behavior of the component. It is identical to the internal behavior of the normal track section shown in Figure A.39 and interacts with the RTSCs in left, right, and bottom in exactly the same way.

The region followingSection implements the role switch of the RTCP NextSectionFree. The RTSC in followingSwitch refines the role RTSC by introducing one additional state Notify including a transition from Notify to Idle and a synchronization channel sectionFree for synchronizing with the regions left, right, and bottom. After one of the latter three regions received request, the transition from RailCabApproaching to WaitForTrack synchronizes with followingSection via nextSectionFree. Then, followingSection switches to WaitForSection and sends requestSectionStatus to the following track section. After the track section has answered with sectionStatus, followingSection switches to Notify. The transition from Notify to Idle then synchronizes with one of the regions left, right, or bottom. The status (free or not free) of the following track section is encoded in the selector expression of sectionFree.

A.5.2.3 RailroadCrossing

Figure A.41 shows the RTSC of the component Crossing_InfProc. The component RTSC defines one state Crossing_InfProc_Main and one variable sectionFree that denotes whether the railroad crossing is currently free or not.

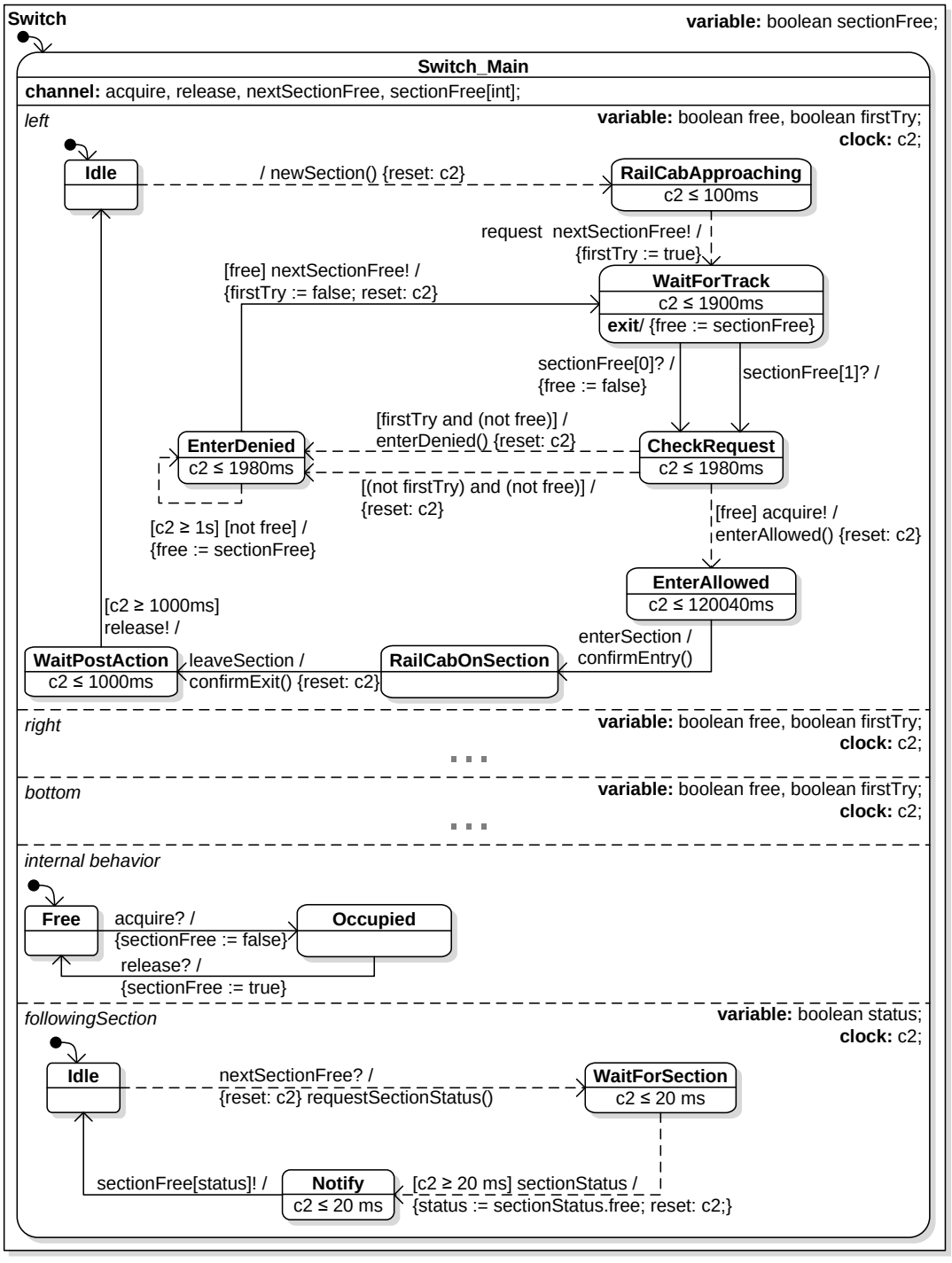


Figure A.40: RTSC of the Component Switch

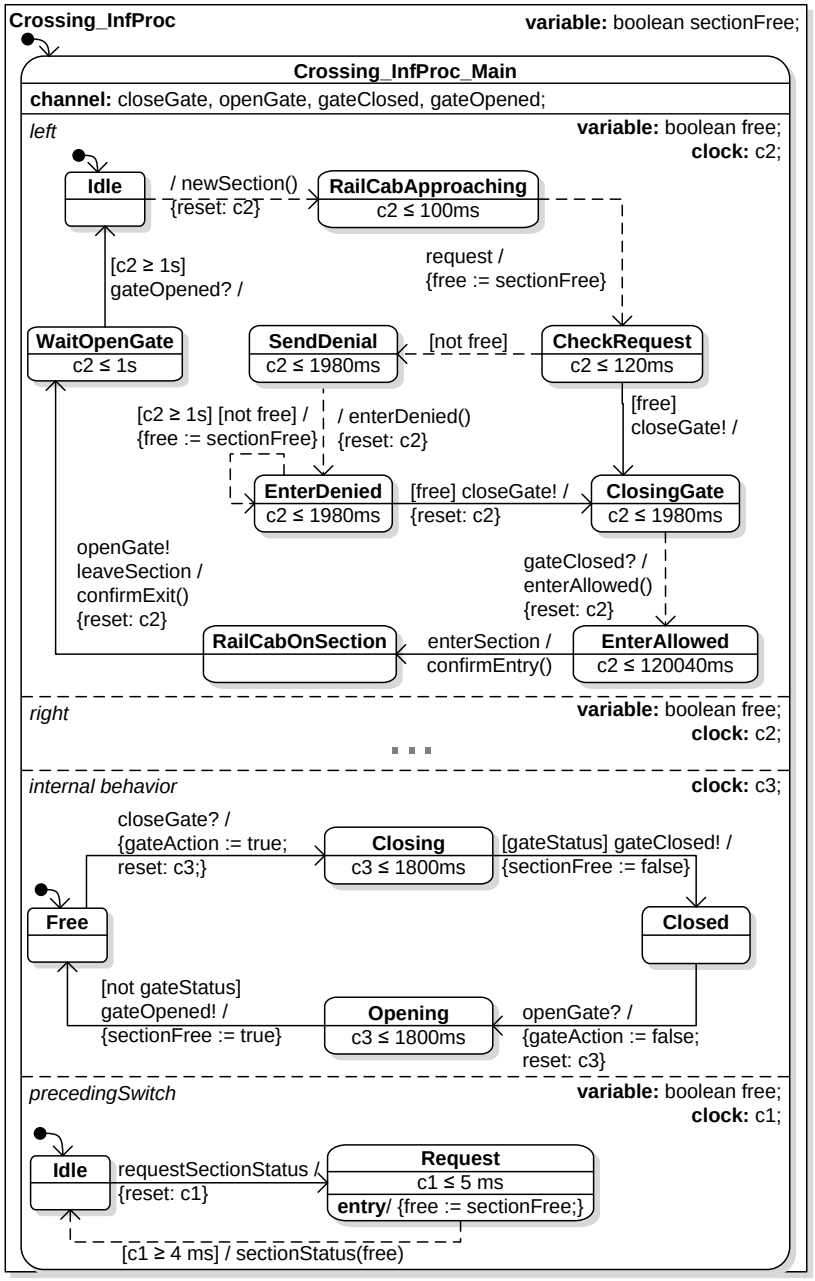


Figure A.41: RTSC of the Component Crossing_InfProc

The state Crossing_InfProc_Main has four regions. The regions left and right contain the port RTSCs of the ports left and right that both refine the role section of the RTCP EnterSection. The RTSCs in both regions are identical to the RTSC shown in Figure 5.16 on Page 137, i.e., they contain the correctly refined behavior for a railroad crossing. We omitted the RTSC for right to improve readability of the figure.

The region internal behavior defines the internal behavior of the component. It synchronizes with the regions left and right for closing the gates when a RailCab wants to enter the railroad crossing. Initially, the railroad crossing is in state Free that denotes that the railroad crossing is free. If a RailCab wants to enter, the transition from CheckRequest to ClosingGate synchronizes via closeGate with the transition Free to Closing in the internal behavior. In the transition action, the transition from Free to Closing sets the value of the hybrid port gateAction to true. This indicates that the gates need to be closed. As a result, the continuous component Gates closes the gates and sets the port status to true as soon as the gates are closed. Then, gateStatus becomes true in Crossing_InfProc and the transition from Closing to Closed fires. It synchronizes via gateClosed with left or right to indicate that the gate is now closed. After the RailCab left the railroad crossing, left or right synchronizes via openGate with the internal behavior to open the gates again. Then, the transition from Closed to Opening sets the hybrid port gateAction to false which causes Gates to open the gates. After the gates are open, gateStatus becomes false and the transition from Opening to Free may fire. This transition synchronizes with left or right to indicate that the section is free again.

Finally, the region precedingSwitch implements the role tracksection of the RTCP NextSectionFree. The RTSC is identical to the one in region precedingSwitch of NormalTrackSection_Main and works in exactly the same fashion.

A.6 Reconfiguration Behavior Specification of Components

This section introduces the remaining parts of the reconfiguration behavior specification of the three structured components RailCabDriveControl, ConvoyCoordination, and VelocityController. For the atomic components, we present the reconfiguration behavior only partially due to the current restrictions of our approach as denoted in Section 4.7. We start by describing the declarative, table-based reconfiguration specifications in Section A.6.1. Thereafter, we present the remaining CSDs used by the components in Section A.6.2. Finally, we present a manager and an executor RTSC that have been generated for the component RailCabDriveControl in Section A.6.3 and story diagram implementations of the operations used by the executor in Section A.6.4.

A.6.1 Declarative, Table-based Reconfiguration Specification

Section A.6.1.1 presents the declarative, table-based specification of ConvoyCoordination while Section A.6.1.2 presents the specification of VelocityController. The specification of RailCabDriveControl has already been given in Section 4.3. In addition, we present RM port and RE port interface specifications of OperationStrategy in Section A.6.1.3 and ConvoyManagement in Section A.6.1.4. Although we cannot yet define generation templates for deriving an executable reconfiguration behavior specification for atomic components, the interface speci-

fications of RM ports and RE ports need to be identical to structured components. Otherwise, we would violate component encapsulation.

A.6.1.1 ConvoyCoordination

Figure A.42 shows the manager specification of ConvoyCoordination. ConvoyCoordination may react to two messages. First, it may receive stopCoordination from ConvoyManagement. This message indicates that the RailCab shall no longer coordinate the convoy, e.g., after all members have left the convoy. This message is not further processed because we do not yet consider dissolving convoys. Second, ConvoyCoordination may receive addConvoyMemberAtPos from the parent component. This message will be treated by executing the reconfiguration rule addConvoyMemberAtPos (cf. Figure 3.14 on Page 54). This reconfiguration has no structural condition, is not safety relevant, and requires no planning.

	Message Type	Treat	Propagate to parent	Reconfiguration Rule	Structural Condition	Safety Relevant	Invoke Planner	Time For Planning
1	stopCoordination	No	No	---	true	No	No	---
2	addConvoyMember AtPos	Yes	No	addConvoyMemberAt Pos()	true	No	No	---

Figure A.42: Manager Specification of the ConvoyCoordination Component

Figure A.43 shows the executor specification of ConvoyCoordination. The executor only contains the reconfiguration rule addConvoyMemberAtPos that defines a WCET requirement of 50 ms.

ID	Reconfiguration Rule	WCET
1	addConvoyMemberAtPos(int pos) : (coordinator cp, refDistProvider rpp)	50 ms

Figure A.43: Executor Specification of the ConvoyCoordination Component

Since addConvoyMemberAtPos shall be received from the parent component, it is contained in the RE port specification of ConvoyCoordination shown in Figure A.44. The entry defines that ConvoyCoordination needs 50 ms for deriving a decision whether to execute the reconfiguration and that the execution takes another 100 ms.

Message Type	Description	Time for Decision	Time for Execution	Minimum Commit Time
addConvoyMember AtPos(int pos) : (coordinator cp, refDistProvider rpp)	The ConvoyCoordination will create and return port instances for communicating with a new convoy member.	50 ms	100 ms	200 ms

Figure A.44: RE Port Specification of the ConvoyCoordination Component

At present, ConvoyCoordination does not send reconfiguration messages to its parent and, thus, the RM port interface specification is empty.

A.6.1.2 VelocityController

Figure A.45 shows the RM port interface specification of VelocityController. It sends three messages to its parent. These are drivingAtHighSpeed, drivingAtNormalSpeed, and distanceSensorFailure. The former two are info messages that indicate that the RailCab drives at high speed or normal speed, respectively. The latter message indicates a hardware failure of the distance sensor and requests a self-healing operation from the parent.

Message Type	Type	Expected Response Time	Description
drivingAtHighSpeed	info	---	RailCab travels at high speed.
drivingAtNormalSpeed	info	---	RailCab travels at normal speed.
distanceSensorFailure	request	250 ms	Distance sensor is broken.

Figure A.45: RM Port Specification of the VelocityController Component

Figure A.46 shows the manager specification of VelocityController. The manager specification defines the three messages contained in the RM port specification are propagated to the parent. These are meant to be collected by a monitor of VelocityController as discussed in Section 7.2. In addition, the manager handles the message switchToConvoy that is received from the parent. This message will be treated and is connected to the eponymous reconfiguration rule. In addition, it defines a structural conditions that is specified by the component SDD inStandaloneCtrl shown in Figure A.87. Before switching to the convoy controller, VelocityController invokes a planner for 10 ms. Finally, the manager contains an entry for the message switchToStandalone that may be used by a member for leaving a convoy. The corresponding reconfiguration rule switchToStandalone will only be executed if the VelocityController is in convoy mode as expressed by the component SDD inConvoyCtrl shown in Figure A.88.

	Message Type	Treat	Propagate to parent	Reconfiguration Rule	Structural Condition	Safety Relevant	Invoke Planner	Time For Planning
1	switchToConvoy	Yes	No	switchToConvoy()	inStandaloneCtrl()	Yes	Yes	10 ms
2	switchToStandalone	Yes	No	switchToStandalone()	inConvoyCtrl()	No	No	---
3	distanceSensorFailure	No	Yes	---	true	No	No	---
4	drivingAtHighSpeed	No	Yes	---	true	No	No	---
5	drivingAtNormalSpeed	No	Yes	---	true	No	No	---

Figure A.46: Manager Specification of the VelocityController Component

Figure A.47 shows the executor specification of VelocityController. It contains the reconfiguration rules switchToConvoy (cf. Figure 3.12) and switchToStandalone. For both reconfiguration rules, it defines that the reconfiguration rules must be executable in 65 ms.

ID	Reconfiguration Rule	WCET
1	switchToConvoy()	65 ms
2	switchToStandalone()	65 ms

Figure A.47: Executor Specification of the VelocityController Component

Finally, Figure A.48 shows the RE port interface specification of VelocityController. It defines two entries, one for switchToConvoy and one for switchToStandalone. The entry for switch-

ToConvoy defines a time for decision of 20 ms. Since this reconfiguration includes a fading function for replacing a controller, it denotes separate times for execution of the setup, fading, and teardown phases. The entry for switchToStandalone is specified analogously.

Message Type	Description	Time for Decision	Time for Execution	Minimum Commit Time
switchToConvoy	The VelocityController operates as a convoy member and considers the distance to the preceding RailCab.	20 ms	Setup: 10 ms Fading: 50 ms Teardown: 5 ms	200 ms
switchToStandalone	The VelocityController operates as standalone or coordinator RailCab and will control speed solely based on a reference speed.	20 ms	Setup: 10 ms Fading: 50 ms Teardown: 5 ms	200 ms

Figure A.48: RE Port Specification of the VelocityController Component

A.6.1.3 OperationStrategy

Figure A.49 shows the RM port interface specification of OperationStrategy. It sends three messages to its parent. These are becomeCoordinator, newMember, and becomeMember. All of which are requests. The first message indicates that OperationStrategy negotiated that the RailCab shall become the coordinator of a convoy. The second message indicates that an additional member shall be added to the convoy. The third message indicates that OperationStrategy negotiated that the RailCab shall become member of a convoy.

Message Type	Type	Expected Response Time	Description
becomeCoordinator	request	500 ms	RailCab should start operating as a convoy coordinator.
newMember	request	500 ms	RailCab is coordinator and needs to add a new member to the convoy.
becomeMember	request	500 ms	RailCab should start operating as a convoy member.

Figure A.49: RM Port Specification of the OperationStrategy Component

Finally, Figure A.50 shows the RE port interface specification of OperationStrategy. It defines four entries that use four messages. These are applyCoordinationStrategy, applyMemberStrategy, disableConvoyBuildUp, and enableConvoyBuildUp. applyCoordinationStrategy causes OperationStrategy to instantiate the port instances that are required for driving as a coordinator. In the same way, applyMemberStrategy causes OperationStrategy to instantiate the port instances that are required for driving as a member. Since becoming a member requires three-phase execution, the entry specifies distinct times for execution for the setup, fading, and teardown phases. Finally, the latter two entries allow to disable and to enable that the RailCab may engage in convoys.

Message Type	Description	Time for Decision	Time for Execution	Minimum Commit Time
applyCoordination Strategy	The OperationStrategy will no longer send a reference speed but information on the current strategy.	5 ms	5 ms	200 ms
applyMemberStrategy	The OperationStrategy will no longer send a reference speed.	5 ms	Setup: 0 ms Fading: 0 ms Teardown: 2 ms	200 ms
disableConvoyBuildUp	The RailCab will not try to engage in convoys anymore.	5 ms	5 ms	2000 ms
enableConvoyBuildUp	The RailCab will try to join convoys if possible and useful.	5 ms	5 ms	2000 ms

Figure A.50: RE Port Specification of the OperationStrategy Component

A.6.1.4 ConvoyManagement

Figure A.51 shows the RM port interface specification of ConvoyManagement. It sends one message to its parent, namely stopCoordination. This message indicates that all member RailCabs have left the convoy and that the RailCab shall stop operating as a coordinator.

Message Type	Type	Expected Response Time	Description
stopCoordination	request	250 ms	RailCab stops being convoy coordinator.

Figure A.51: RM Port Specification of the ConvoyManagement Component

Finally, Figure A.52 shows the RE port interface specification of ConvoyManagement. It defines two entries that use two messages. These are createFirstMemberPorts and createMemberPortsAfter. Both messages are required for adding new members to the convoy. The first message, createFirstMemberPorts, is used for adding a member at the first position, i.e., directly behind the coordinator. The second message, createMemberPortsAfter, is used for adding a member after the one whose port instances are passed as parameters.

Message Type	Description	Time for Decision	Time for Execution	Minimum Commit Time
createFirstMemberPorts() : (coordinator cPort, profileProvider pPort)	Creates new port instances for dealing with an additional member driving at first position.	5 ms	5 ms	500 ms
createMemberPortsAfter (coordinator c, profileProvider p) : (coordinator cPort, profileProvider pPort)	Creates new port instances for dealing with an additional member driving behind the RailCab whose corresponding port instances are given as parameters.	5 ms	5 ms	500 ms

Figure A.52: RE Port Specification of the ConvoyManagement Component

A.6.2 Reconfiguration Rules

This section introduces the reconfiguration rules specified as CSDs that we use in our RailCab example for building convoys. In our example, we build convoys by first establishing a convoy of two RailCabs and then adding additional RailCabs one after the other later on. As a result, we need two reconfigurations. The first one reconfigures the CIC of a RailCab driving

alone (cf. Figure 3.9 on Page 47) to a CIC of a coordinator with one member (cf. Figure A.27 on Page 221). We explain the necessary CSDs and constructor CSDs in Section A.6.2.1. The second reconfiguration adds one additional member to a convoy. We explain this reconfiguration in Section A.6.2.2. The CSDs defining the reconfiguration for a RailCab to become a member of a convoy have already been introduced in Section 3.3 and will not be repeated in this section. In addition, we introduce the CSDs that are used by RailCabDriveControl for enabling and disabling the convoy mode in Section A.6.2.3. Finally, we present the CSDs that are used by the component OperationStrategy for handling the instantiation of RTCPs on system level in Section A.6.2.4.

A.6.2.1 Becoming Coordinator

Figure A.53 shows the CSD becomeCoordinator that reconfigures the component instance standaloneRC of Figure 3.9 such that it is equivalent to Coordinator in Figure A.27.

In the first story node, we match the embedded component instances of types OperationStrategy and DriveLogic. Then, we destroy the assembly connector instance between both component instances and invoke the reconfiguration applyCoordinationStrategy on the component instance matched by os. Since both component parts referenced by os and dl have cardinality [1] (cf. Figure 3.6), this story node is not expected to fail.

In the second story node, we create instances of ConvoyCoordination and PositionSensor. For instantiating ConvoyCoordination, we use a constructor instantiate1Member that initializes the component instance such that it is equivalent to cc shown in Figure 3.10. In addition, we create an assembly connector instance between speedProvider of c and maxSpeed of dl. Furthermore, we create the multi port instances coordinator and refDistProvider with one subport instance each and delegate them to the corresponding port instances of c. Since the VelocityController does not change if a RailCab becomes coordinator of a convoy, it is not used in the CSD.

Figure A.54 shows the CSD applyCoordinationStrategy that is invoked in the first story node of becomeCoordinator shown in Figure A.53. OperationStrategy is an atomic component and, therefore, the this variable has no embedded part variables. The CSD deletes the speedProvider port instance and creates a strategySender port instance.

Figure A.55 shows the constructor instantiate1Member of the component ConvoyCoordination that is used by the CSD becomeCoordinator shown in Figure A.53. Since the CSD is a constructor CSD, all variables of the component story pattern carry a «create» stereotype. The constructor creates instances of ConvoyManagement and RefGen. For RefGen, it recursively invokes the constructor CSD initWithCurPos shown in Figure A.56 for creating an instance with a curPos port instance. For ConvoyManagement, we use the default constructor. In addition, the constructor creates the necessary port instances for an instance of ConvoyCoordination including the delegations to cm and rg1. The resulting instance of ConvoyCoordination is equivalent to the component instance cc shown in Figure 3.10 on Page 49.

A.6.2.2 Adding Convoy Members

Having established a convoy with one member, we may add additional members by executing the CSD addConvoyMember shown in Figure A.57 on a component instance of type RailCabDriveControl. In the first story node, addConvoyMember triggers the reconfiguration add-

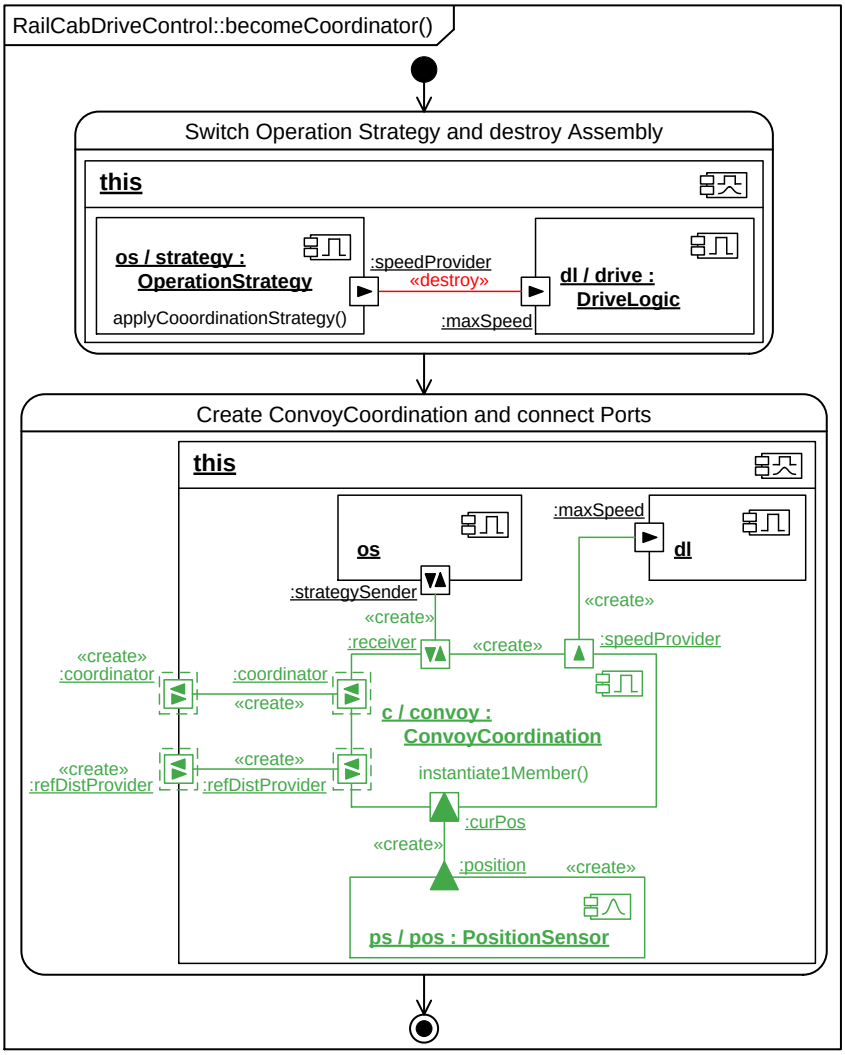


Figure A.53: CSD for Component RailCabDriveControl that Reconfigures the Component Instance to Serve as a Coordinator

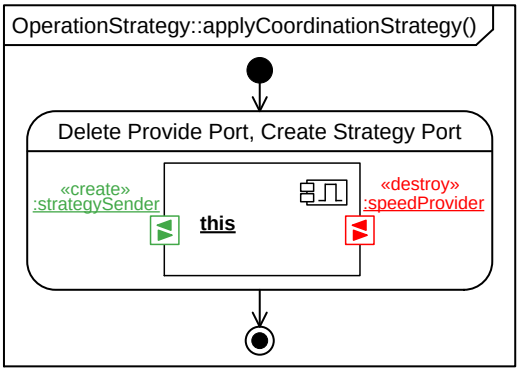


Figure A.54: CSD for Component OperationStrategy that Reconfigures the Ports for Being Coordinator

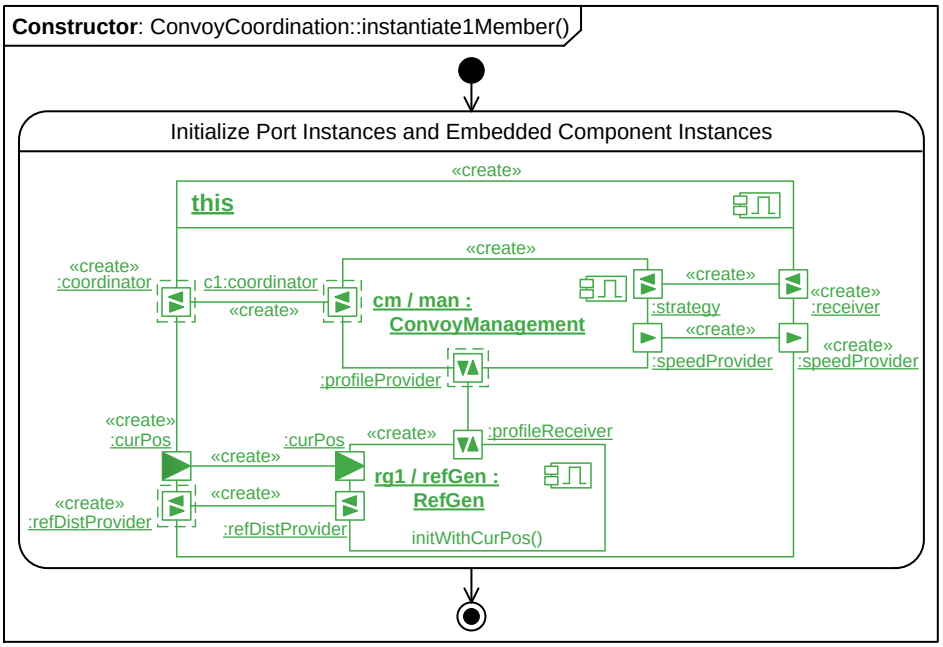


Figure A.55: Constructor CSD for Creating an Instance of ConvoyCoordination

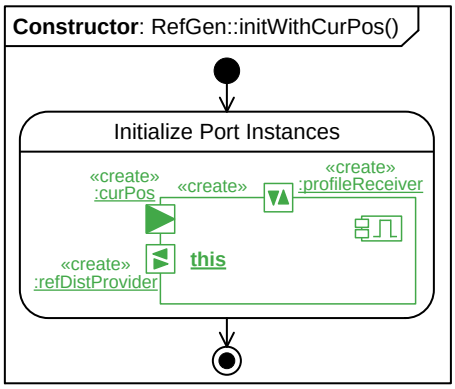


Figure A.56: Constructor CSD for Creating an Instance of RefGen

ConvoyMemberAtPos on the ConvoyCoordination instance. Then, ConvoyCoordinator reconfigures itself to include the new member at the given position pos. The necessary reconfiguration of ConvoyCoordination is defined by the CSD shown in Figure 3.14 on Page 54. The CSD addMemberAtPos returns the two port instances cp and rpp that it created. These are assigned to the variables cp and rpp.

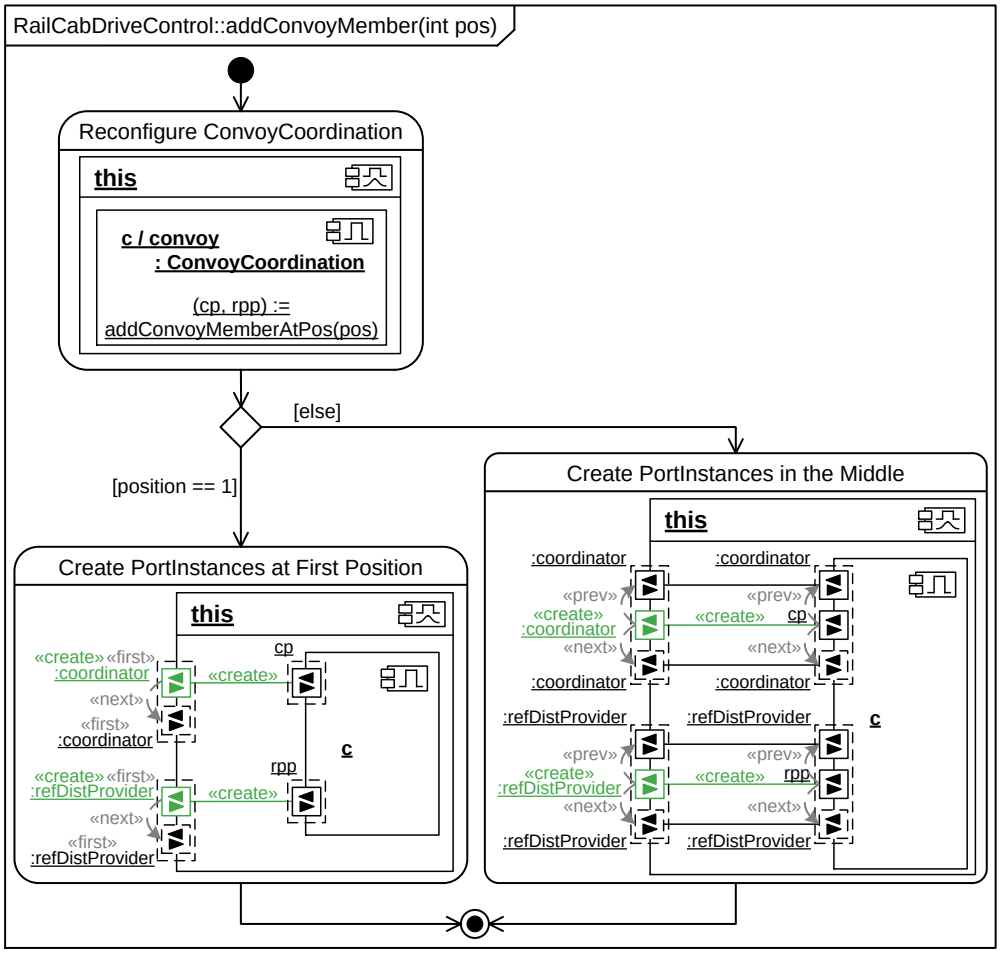


Figure A.57: CSD for Component RailCabDriveControl that Adds an Additional Convoy Member

The decision node after the first story node distinguishes two cases, namely, whether the new member is added at the first position or in the middle of the convoy. In the former case, the execution of the CSD proceeds with the story node at the bottom left. It matches the first subport instances of coordinator and refDistProvider of this, which are both optional. Then, it creates new subport instances for both multi port instances which both carry the «first» constraint. As a result, they will be inserted at the first position and the previously first subport instances are their direct successors. In addition, the component story pattern creates delegation connector instances to cp and rpp, respectively.

If the new member is not added at the first position, we proceed with the story node at the bottom right. This story node creates new subport instances for coordinator and refDistProvider

as well. It inserts these port instances at the same position where cp and rpp have been inserted in ConvoyCoordination. Again, cp and rpp are delegated to the newly created port instances of this.

The CSD addConvoyMemberAtPos in Figure 3.14 on Page 54 invokes two reconfigurations on ConvoyManagement: createFirstMemberPorts and createMemberPortsAfter that we will explain in more detail in the following.

The CSD createFirstMemberPorts of ConvoyManagement is shown in Figure A.58. It consists of three story nodes. The first story node in the upper left corner matches the first subport instances of coordinator and profileProvider. If these subport instances could be matched successfully, the story node at the bottom left creates new subport instances at the first position. The previously matched subport instances tmpC and tmpP become direct successor of the newly created subport instances. If the first story node could not be matched successfully, then no subport instances exist in coordinator and profileProvider. In this case, the story node in the upper right corner creates new subport instances and inserts them at the first position. In both case, the created subport instances newC and newP are assigned to the output parameters cPort and pPort, respectively, at the final node.

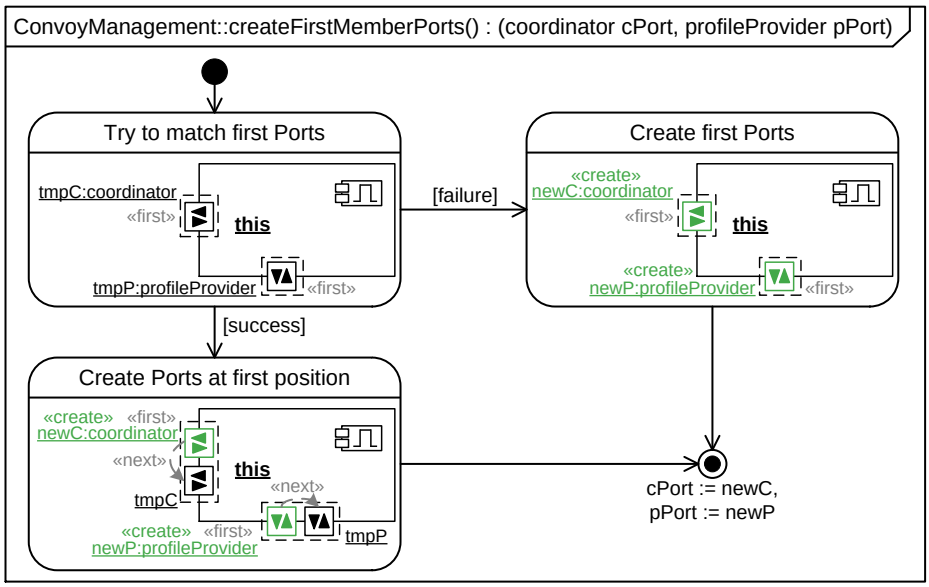


Figure A.58: CSD for Component ConvoyManagement that Adds Port Instances for an Additional Convoy Member at the Beginning of the Convoy

The CSD createMemberPortsAfter of ConvoyManagement is shown in Figure A.59. It takes two subport instances of the multi ports coordinator and profileProvider as its inputs. The subport instances that are created in the story node will be direct successors of these subport instances. If these subport instances already had direct successors, they would be matched by the optional variables. Then, the subport instances matched by the optional variables become direct successors of the newly created subport instances for maintaining a correct order of the subport instances.

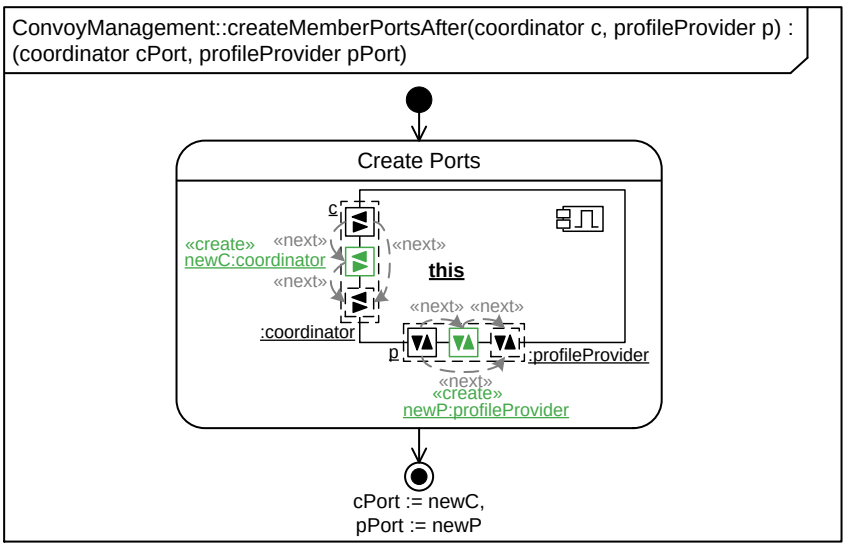


Figure A.59: CSD for Component ConvoyManagement that Adds Port Instances for an Additional Convoy Member in the Middle of the Convoy

A.6.2.3 Enabling and Disabling the Convoy Mode

In our example, we implement stopping the RailCabs from engaging in convoys and that they may start engaging in convoy, again.

The CSD disableConvoyMode is executed by RailCabDriveControl for stopping to engage in further convoy. Therefore, the CSD deletes the broadcast port that is used for establishing new connections to other RailCabs. Without the broadbast port, the RailCab will not be able to instantiate the requestor and requestee ports and, thus, it may not start collaborating with other RailCabs.

If the instance of RailCabDriveControl contains instances of the requestor and requestee ports, then these instances are deleted as well by the optional port variables. Along with the port instances, we delete all delegation connector instances. Finally, the second activity node of the CSD invokes the reconfiguration disableConvoyBuildUp on os.

The CSD of OperationStrategy that corresponds to disableConvoyBuildUp is shown in Figure A.61. This CSD operates in the same fashion as disableConvoyMode, i.e., it deletes the broadcast port instance and the possibly existing instances of requestor and requestee.

If the RailCab shall engage in convoys, again, it executes the CSD enableConvoyMode on RailCabDriveControl. The CSD is shown in Figure A.62. This CSD consists of two story nodes. The first story node invokes the reconfiguration enableConvoyBuildUp on os. The second story node creates a new instance of the broadcast port including a delegation connector instance to the broadcast port instance of os that has been created by enableConvoyBuildUp.

The CSD of OperationStrategy that corresponds to the reconfiguration enableConvoyBuildUp is shown in Figure A.63. Similar to enableConvoyMode, it creates a new instance of the broadcast port for the instance of OperationStrategy.

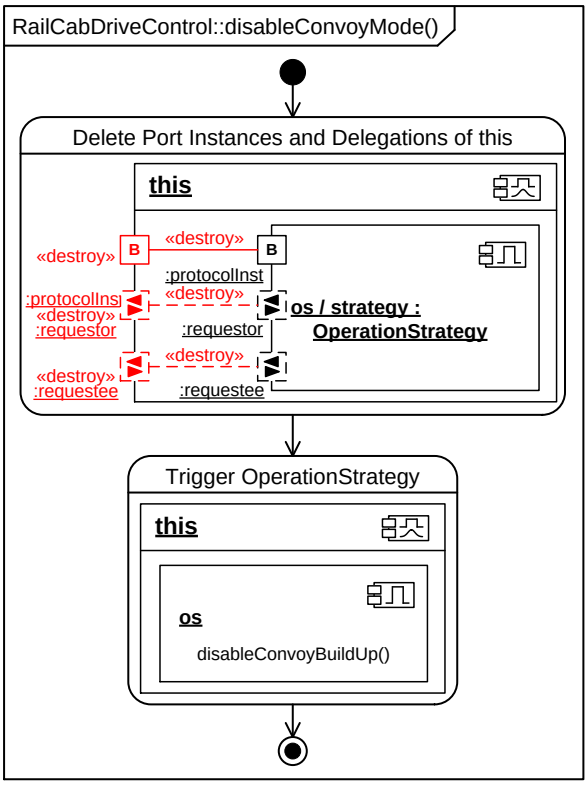


Figure A.60: CSD for Component RailCabDriveControl that Disables the Convoy Mode by Deleting the Necessary Port Instances for Convoy Build-up

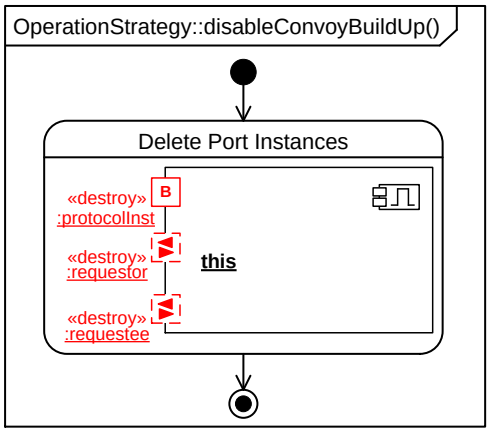


Figure A.61: CSD for Component OperationStrategy that Disables the Convoy Mode by Deleting the Port Instances that are Necessary for Convoy Build-up

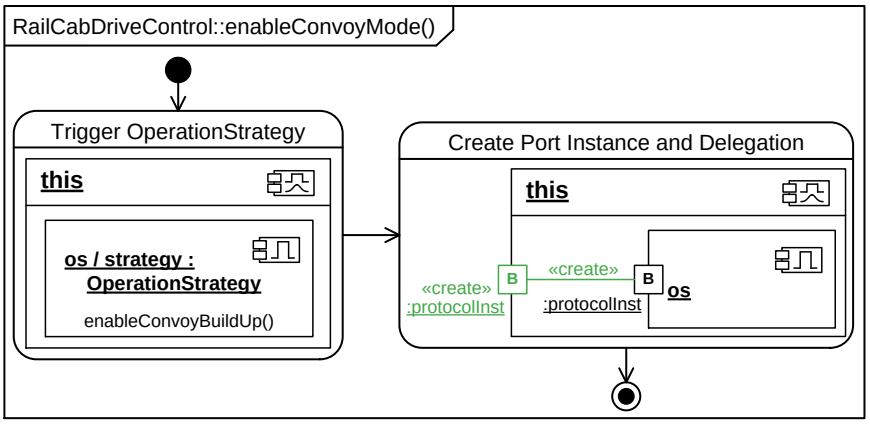


Figure A.62: CSD for Component RailCabDriveControl that Enables the Convoy Mode by Creating the Necessary Broadcast Port Instance

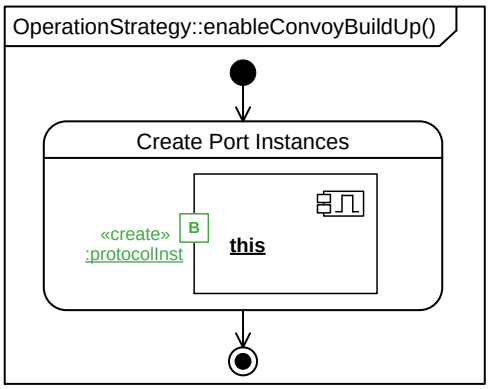


Figure A.63: CSD for Component OperationStrategy that Enables the Convoy Mode by Creating the Necessary Broadcast Port Instance

A.6.2.4 Handling Connection Setup in OperationStrategy

Our concept for instantiating RTCPs on system level as introduced in Section 3.4 involves several reconfigurations inside OperationStrategy. We introduce the CSDs that specify these reconfigurations in the following.

The RTSC of the broadcast port (cf. Section A.2.2) triggers the instantiation of an instance of requestor or requestee depending on whether it initiated the instantiation or not. The resulting CSDs that create these port instances are shown in Figures A.64 and A.65. Both CSDs contain only one story node that creates the port instance.

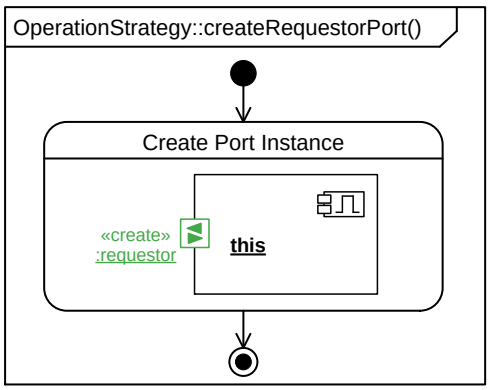


Figure A.64: CSD for Component OperationStrategy that Creates a requestor Port Instance

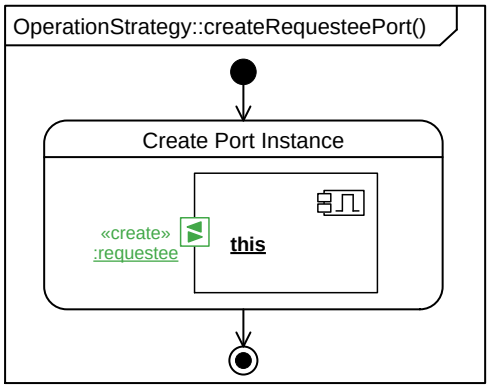


Figure A.65: CSD for Component OperationStrategy that Creates a requestee Port Instance

Thereafter, both RailCabs use the RTCP ProtocolInstantiation for instantiating the ConvoyEntry RTCP. As a consequence, OperationStrategy needs to instantiate the peer port, which is specified by the CSD in Figure A.66.

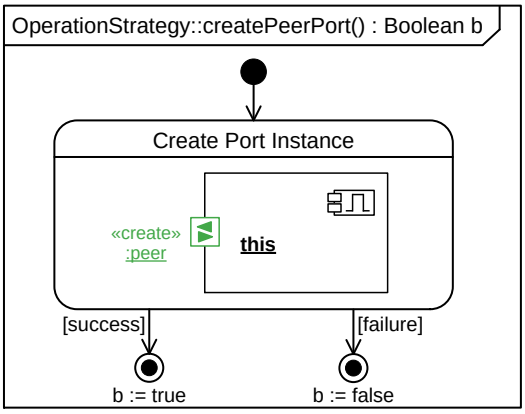


Figure A.66: CSD for Component OperationStrategy that Creates a peer Port Instance

A.6.3 Generated RTSCs for Manager and Executor of RailCabDriveControl

This section presents examples of a manager RTSC (Section A.6.3.1) and an executor RTSC (Section A.6.3.2) that have been derived for the component RailCabDriveControl based on the generation templates given in Section 4.4.

A.6.3.1 Manager RTSC

Figures A.67 and A.68 show the manager RTSC of the component RailCabDriveControl that has been derived based on the generation template shown in Figure 4.15 and the declarative, table-based specification introduced in Section 4.3. In the following, we briefly describe how the RTSC resulted from the template and we do not repeat the general behavior of the RTSC. We reused the color coding of the template in order to relate generated constructs in the manager RTSC to their corresponding template elements.

The parent region contains three transitions from Idle to Propagated that result from the three messages that are propagated to the parent as defined in the manager specification in Figure 4.12. These transitions send the messages distanceSensorFailure, drivingAtHighSpeed, and drivingAtNormalSpeed to the parent component. In addition, three corresponding synchronization channels syncDistanceSensorFailure, syncDrivingAtHighSpeed, and drivingAtNormalSpeed have been created for synchronizing the parent region with the subport that initially received this message.

The executor region contains two transitions from Idle to Request that result from the two messages that are received by the RE port of RailCabDriveControl (cf. Figure 4.14) and propagated by the executor. These are noConvoyMode and enableConvoyMode. Along with the transitions, we generated two synchronization channels syncNoConvoyMode and syncEnableConvoyMode for synchronizing the executor region and the internal behavior.

The internal behavior regions received five additional states; one for each entry of the manager specification in Figure 4.12 that is treated. These states are CheckBecomeCoordinator, CheckBecomeMember, CheckNewMember, CheckNoConvoyMode, and CheckEnableConvoyMode. The transitions from Idle to these states are triggered by synchronizations via the corresponding synchronization channels. In addition, these transitions check the structural condition, if

any, and whether blockable reconfigurations are indeed blocked. In the example, we replaced the checkStructuralConditionForX operations by the component SDDs that define the structural condition. For CheckBecomeCoordinator and CheckBecomeMember, the outgoing transitions to Plan invoke the planner and have a deadline that corresponds to the time for planning as specified in the manager specification in Figure 4.12.

Finally, the subport contains six additional states with adjacent transitions. These states and transitions have been created for messages that may be sent by the RM ports of embedded components. In our example, we need to handle messages that are defined in the RM port interface specifications of OperationStrategy (cf. Figure A.49) and VelocityController (cf. Figure A.45). For the four requests becomeCoordinator, newMember, becomeMember, and distanceSensorFailure, we additionally create an invariant and a transition back to Idle. The invariant is derived from the expected response time of the child, the time for planning, and by considering an additional overhead for the internal computation of the manager.

A.6.3.2 Executor RTSC

Figures A.69 and A.70 show the executor RTSC of the component RailCabDriveControl that has been derived based on the generation template shown in Figure A.69 and the declarative, table-based specification introduced in Section 4.3. In the following, we briefly describe how the RTSC resulted from the template and we do not repeat the general behavior of the RTSC. We reused the color coding of the template in order to relate generated constructs in the manager RTSC to their corresponding template elements. In addition, we omitted parts of the RTSCs that only consist of black states and, therefore, do not differ from the template.

The region parent contains two additional states CheckNoConvoyMode and CheckEnableConvoyMode that result from the two entries of the RE port interface specification in Figure 4.14. The parent region receives the corresponding message at the incoming transitions of these states and tries to synchronize with the events region at the transitions leading to CheckSelf. This results in two synchronization channels checkNoConvoyMode and checkEnableConvoyMode that are generated for the RE port interface entries. The invariants of the two states result from the time for decision given in the RE port interface entries minus the time that is necessary for checking the request by the manager. If the manager can no longer finish checking the request on time after waiting for a given amount of time, then parent directly switches to SendAbort.

The region events contains two additional transitions from Idle to Check that correspond to the two entries of the RE port interface specification. In particular, these transitions synchronize with the parent and forward the message that has been received by parent to the manager.

The region internal behavior contains additional constructs for executing the CSDs that are contained in the executor specification (cf. Figure 4.13). First, the transitions from Idle to Start are extended by a guard condition that enables to distinguish between reconfigurations that are executed based on single-phase execution and three-phase execution. In our example, only becomeMember with ID 3 (cf. Figure 4.13) needs to be executed based on three-phase execution and, thus, we only set singlePhase to false if becomeMember shall be executed. In addition, we receive four transition from Execute to Report; one for each CSD that is executed based on single-phase execution. Finally, we receive the hierarchical Lo-

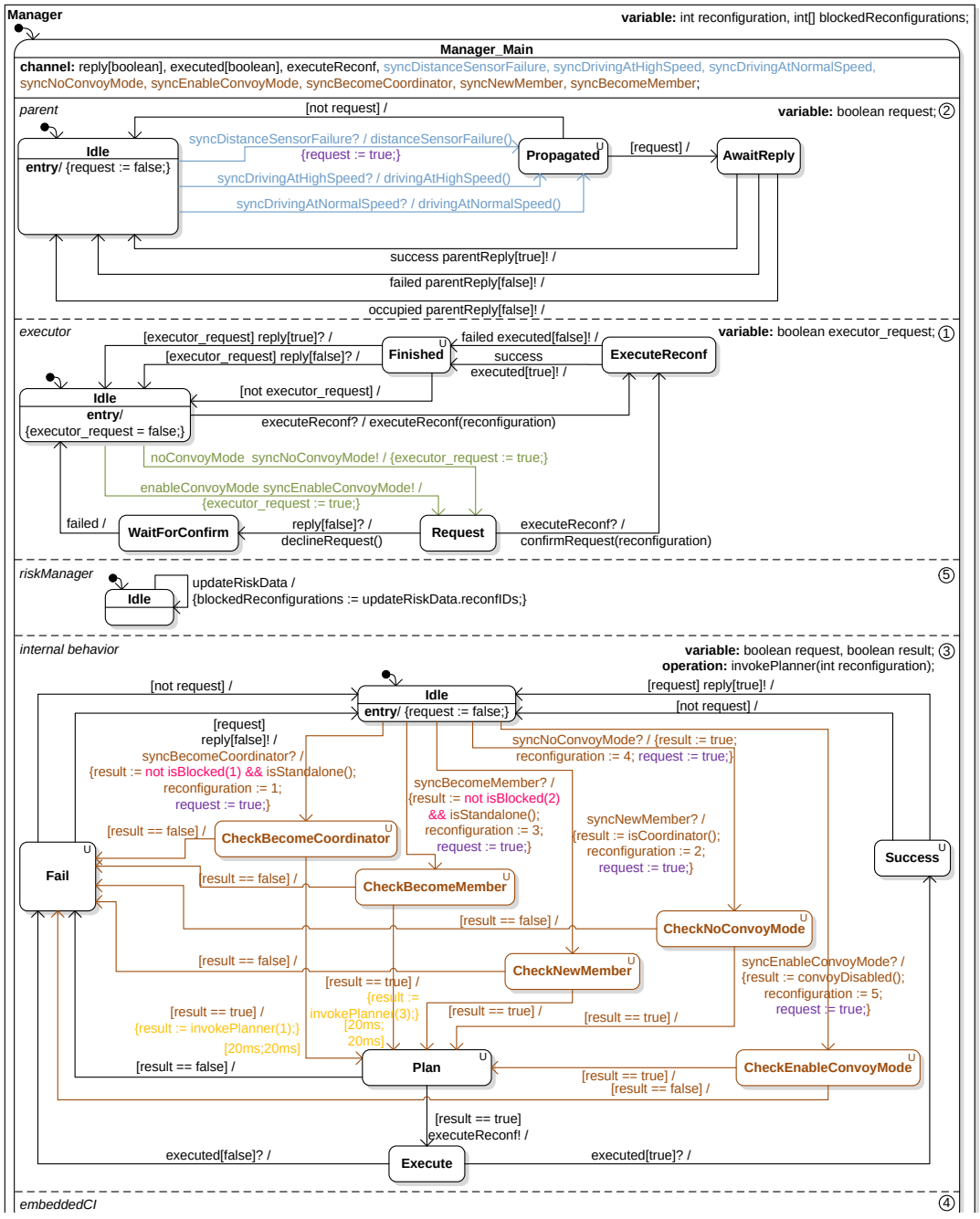
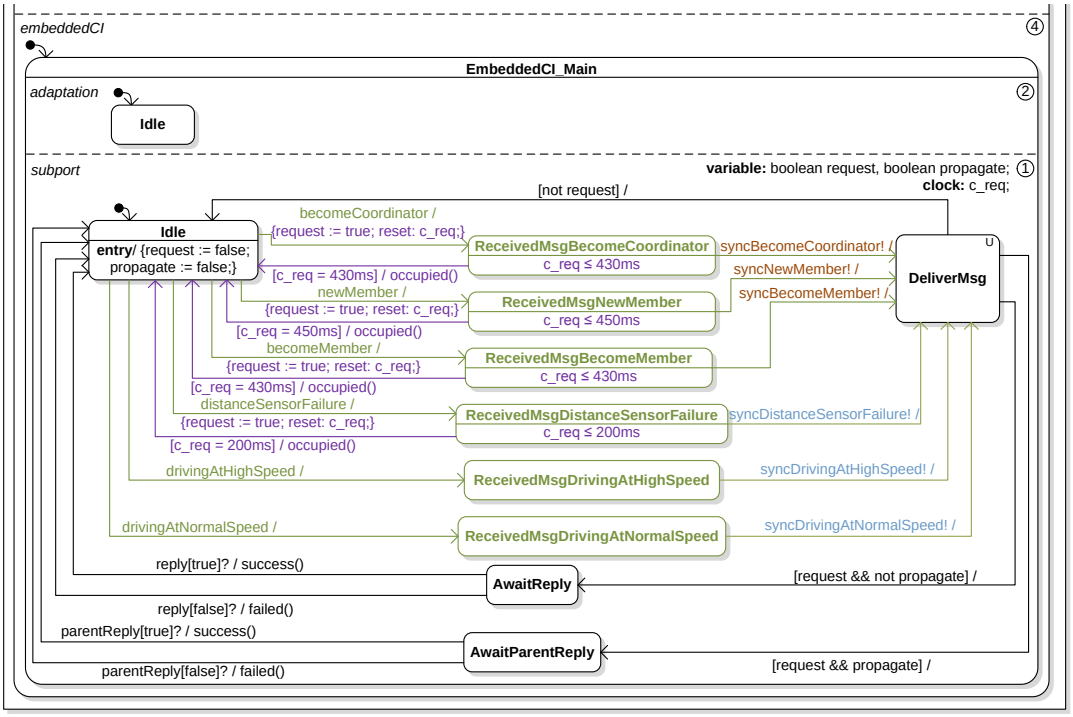


Figure A.67: Generated RTSC of the Manager of RailCabDriveControl (Pt. 1)



Legend:

- Generated only once and are used by all reconfiguration rules
- Generated for each reconfiguration message X that is propagated to the parent
- Generated for each reconfiguration message X that is treated.
- Generated additionally for each reconfiguration message X that is request from child.
- Generated for each reconfiguration message X that is received from child or executor.
- Generated additionally for each reconfiguration message X that invokes planning.
- Generated additionally for each reconfiguration Y that may be blocked.

Figure A.68: Generated RTSC of the Manager of RailCabDriveControl (Pt. 2)

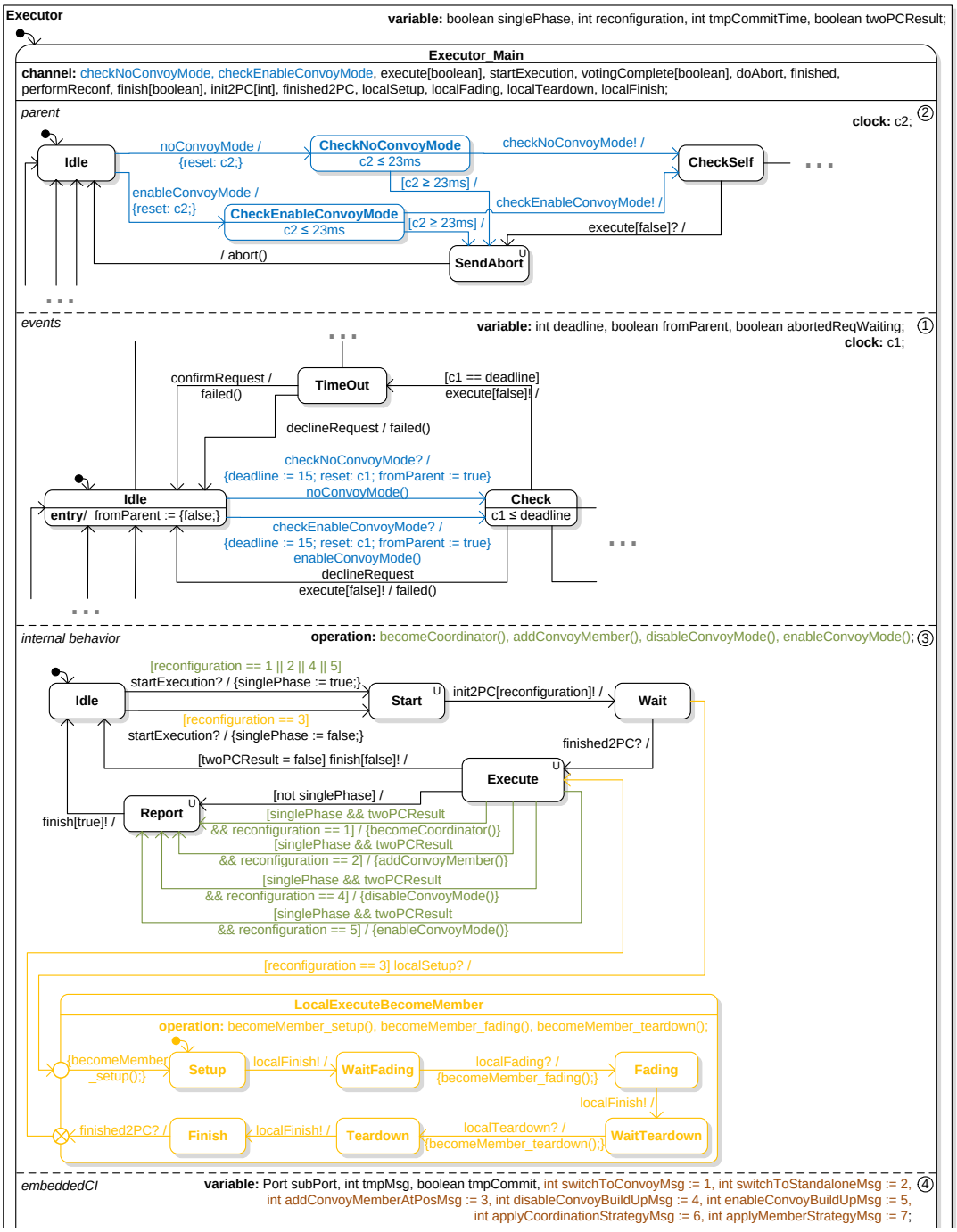


Figure A.69: Generated RTSC of the Executor of RailCabDriveControl (Pt. 1)

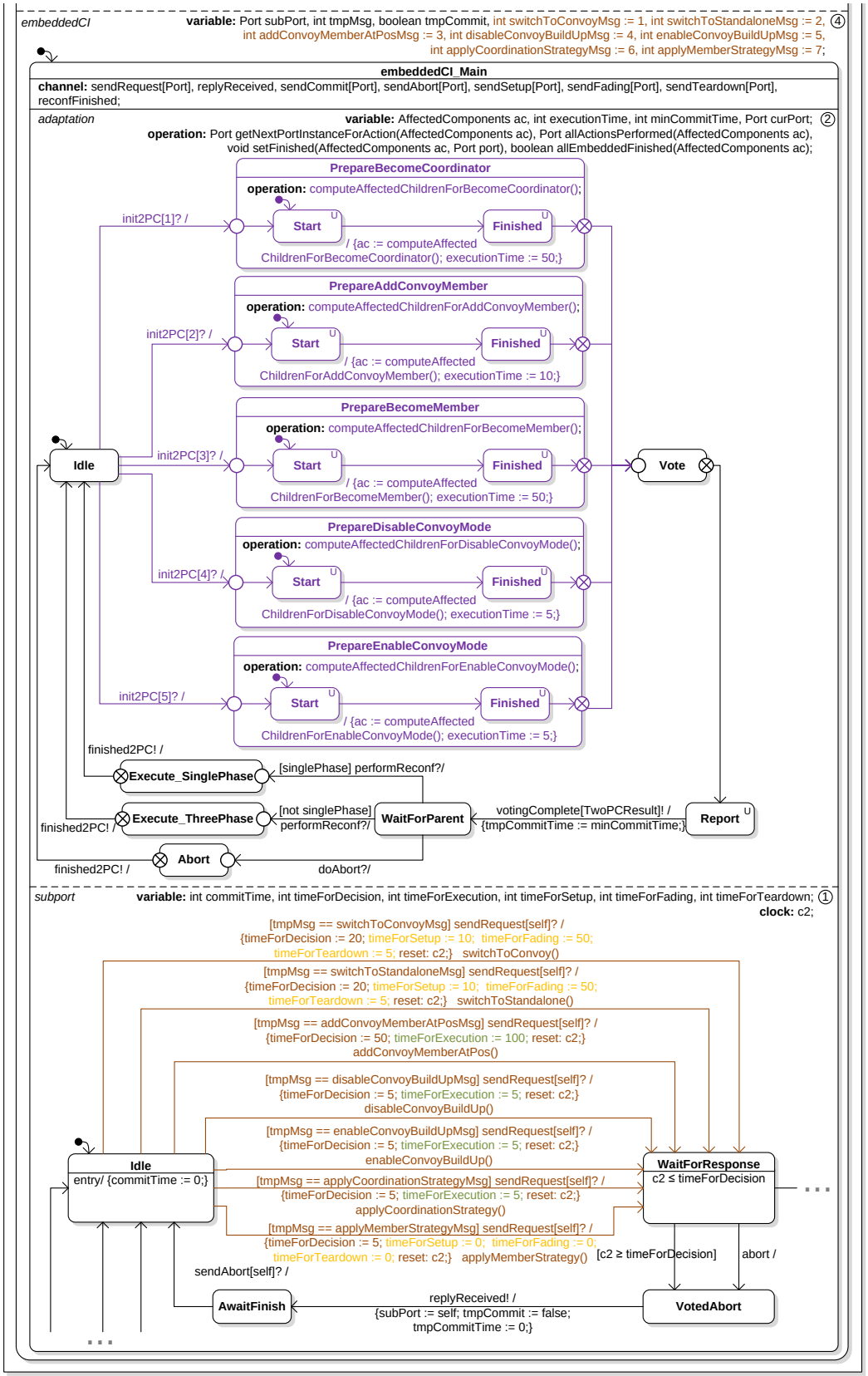


Figure A.70: Generated RTSC of the Executor of RailCabDriveControl (Pt. 2)

calExecuteBecomeMember state for the reconfiguration becomeMember that is executed based on three-phase execution.

The adaptation region of embeddedCI contains five additional hierarchical states; one for each entry of the executor specification (cf. Figure 4.13). In particular, we receive PrepareBecomeCoordinator, PrepareAddConvoyMember, PrepareBecomeMember, PrepareDisableConvoyMode, and PrepareEnableConvoyMode. Each of these is connected to Idle and the corresponding transition uses the ID of the reconfiguration in the executor specification as a selector expression for Idle. Inside each of the hierarchical states, we obtain an operation that is executed at the transition from Start to Finished that computes which children are affected by the reconfiguration. These operations need to be defined for each of the reconfigurations. We present the specification of computeAffectedChildrenForBecomeMember for the reconfiguration becomeMember in Section A.6.4.2.

Finally, the subport region of embeddedCI contains seven additional transitions. These transitions enable to sent messages to children that are defined in the children's RE port interface specifications. In our example, the executor of RailCabDriveControl may sent messages to OperationStrategy, to ConvoyCoordination, and to VelocityController. For each of the messages, we generate a constant in embeddedCI, e.g., switchToConvoyMsg for the message switchToConvoy, that defines an integer ID for this message. This ID is used by computeAffectedChildrenForBecomeMember for denoting that executing becomeMember requires sending switchToConvoy to the VelocityController. In addition, we store the time for execution that appears in the RE port interface specification entry of the child in corresponding variables upon firing one of the seven transitions.

A.6.4 Specification of the Executor Operations

This section presents an implementation of the operations that are contained in the executor RTSC. Section A.6.4.1 introduces an implementation of the AffectedComponents type. Section A.6.4.2 presents an example for the component-specific operation computeAffectedChildrenForY. Finally, Section A.6.4.3 introduces story diagrams that specify the behavior of all other operations that are used in the executor RTSC.

A.6.4.1 Structure Type AffectedComponents

Figure A.71 shows the structure type AffectedComponents. This type is used by the executor RTSC. The function computeAffectedChildrenForY introduced in the next section instantiates this type and creates one AffectedComponentEntry for each child that needs to perform a reconfiguration.

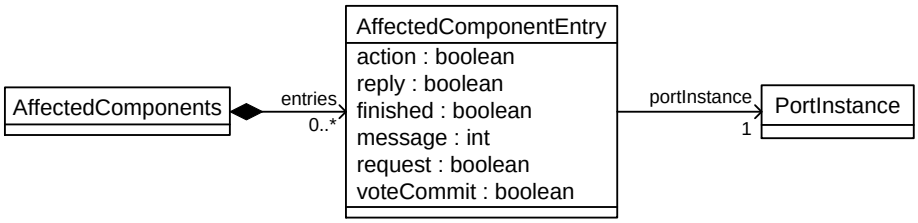


Figure A.71: Definition of the AffectedComponents Data Type

The AffectedComponentEntry defines an integer ID for the message that needs to be sent to the child for triggering the required reconfiguration. In addition, it refers via portInstance to the subport instance of the embeddedCI multi port instance of the executor that is connected to this child. Finally, the AffectedComponentEntry contains several Boolean attributes for keeping track of the progress of the interaction with the child.

A.6.4.2 Component-Specific Story Diagrams

The executor RTSC uses one component-specific operation computeAffectedChildrenForX for each reconfiguration X that appears in the executor specification. Each of these operations is implemented by a component-specific story diagram and is responsible for creating an instance of the AffectedComponents structure type introduced in Section A.6.4.1.

Figure A.72 shows a story diagram that implements the operation computeAffectedChildrenForBecomeMember for the CSD becomeMember shown in Figure 3.11 on Page 51. The story diagram consist of three story nodes and returns an instance of AffectedComponents.

The first story node simply creates the result object using the object variable ac.

The second story node is equivalent to the first story node of becomeMember except that the component story pattern has been translated into a normal story pattern and that all binding operators have been removed. Removing the binding operators ensures that the story pattern does not modify the model@runtime. Matching the story pattern to the model@runtime yields all component instances that will be matched by the component story pattern. In particular, we match the instances of OperationStrategy and VelocityController that trigger a child reconfiguration.

The third story node creates the AffectedComponentEntry objects for the two child invocations. In particular, we create e1 for OperationStrategy and e2 for VelocityController. For each entry, we need to add the ID of the message that needs to be sent to this particular child. For OperationStrategy, we need to send the message applyMemberStrategy, which has received ID 7 in the executor RTSC (cf. Figure A.70). Thus, we assign 7 to the attribute message. For e2, we obtain ID 1 because switchToConvoy has ID 1 in the executor RTSC. Starting from object variable os, which has been matched to the ComponentInstance of OperationStrategy in the second story node, we match the REPortInstance of this ComponentInstance. Then, we traverse the ConnectorInstance to the REPortInstance re1 that belongs to the executor. Finally, we add this REPortInstance to the AffectedComponentEntry e1. For vc, we proceed in the same way.

Since the second story node of becomeMember does not contain further invocations of child reconfigurations, we do not need to evaluate this story node. Thus, we may terminate computeAffectedChildrenForBecomeMember after the third story node and assign ac to the output parameter resultAC at the final node.

A.6.4.3 Component-Independent Story Diagrams

The executor RTSC uses ten operations whose behavior we specify using story diagrams. The story diagrams operate on the AffectedComponents structure and are the same for any executor RTSC independent of the component. We introduce all story diagrams briefly in the following.

Figure A.73 shows the story diagram getNextPortInstanceForRequest that returns a subport instance of embeddedCI that has not yet sent its message to the child. Based on the input

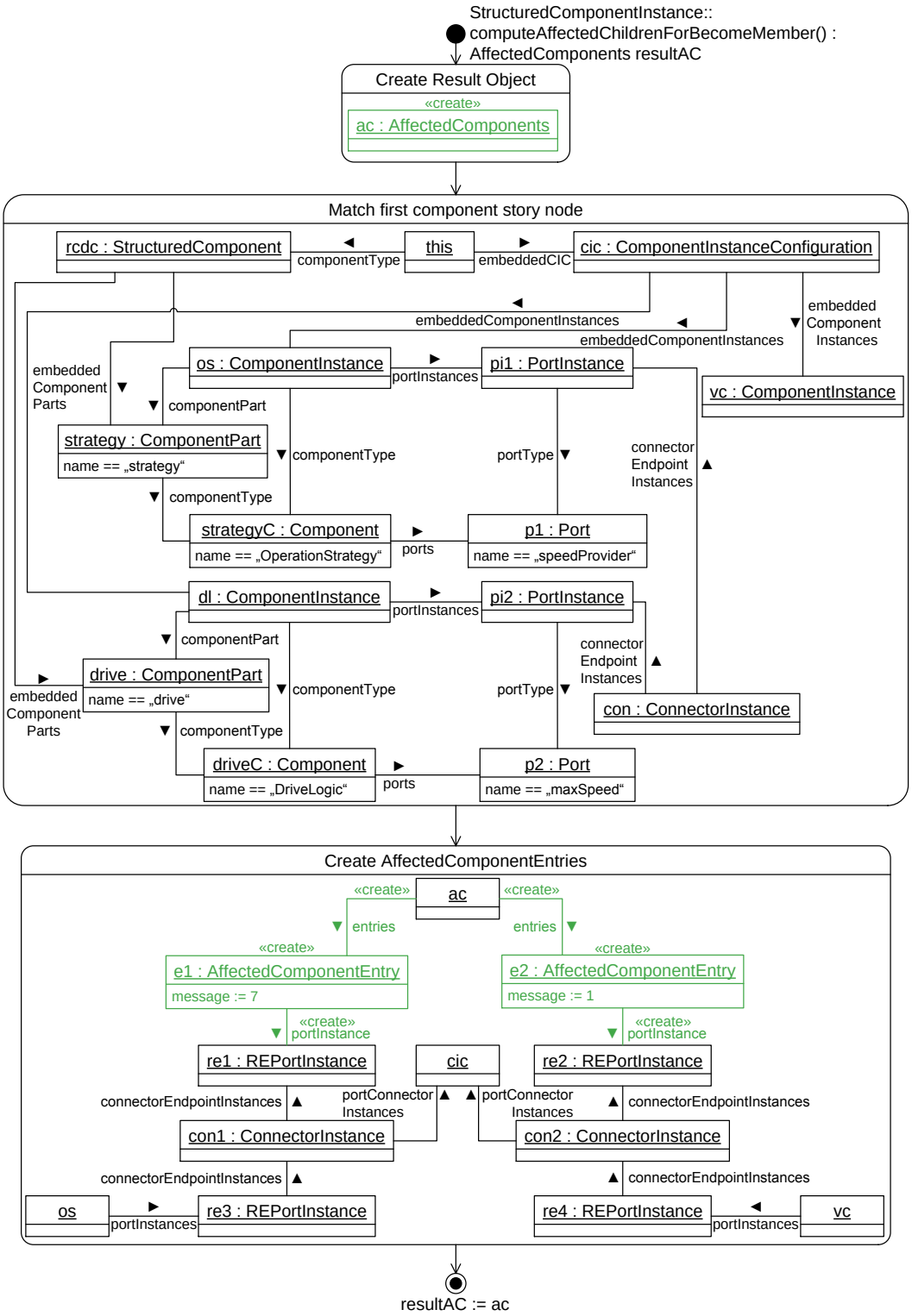


Figure A.72: Story Diagram Implementing the Operation computeAffectedChildrenForBecomeMember

parameter ac, it matches an AffectedComponentEntry where request is false. Then, it sets request to true and returns the portInstance that belongs to entry.

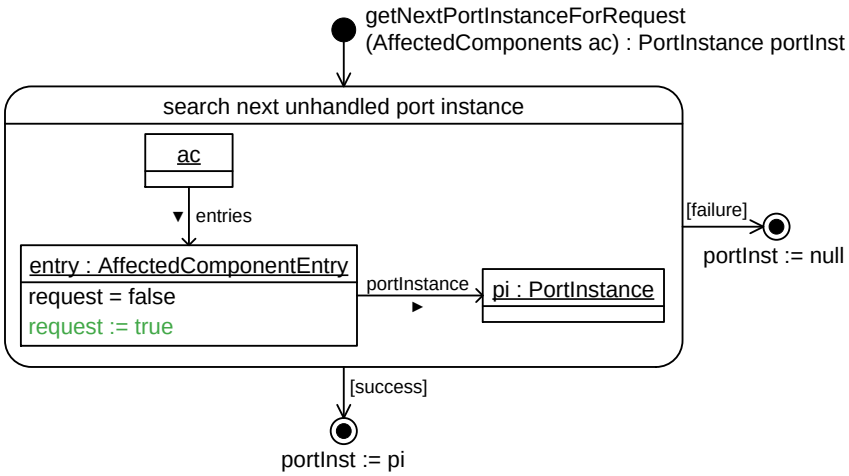


Figure A.73: Story Diagram Specifying the Behavior of getNextPortInstanceForRequest

Figure A.74 shows the story diagram getMessage that returns the message that needs to be sent to a child for triggering the required reconfiguration. Therefore, it matches the AffectedComponentEntry that belongs to portInst and returns the ID of the message that is stored in the entry.

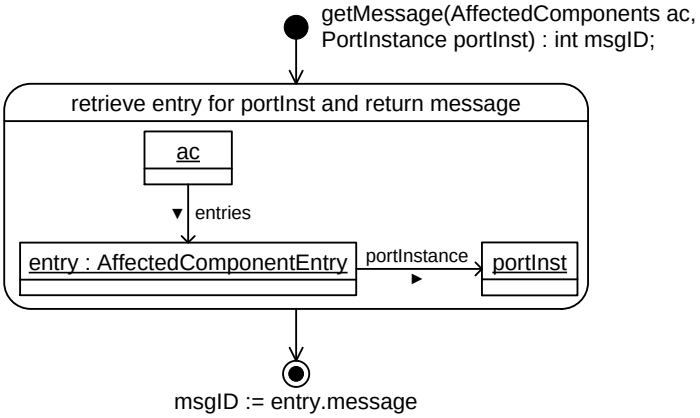


Figure A.74: Story Diagram Specifying the Behavior of getMessage

Figure A.75 shows the story diagram setReply that stores the voting result of a child in the corresponding AffectedComponentEntry. Therefore, the inputs are the AffectedComponents structure, the portInst that has provided its voting result, and the vote itself. Then, the story diagram simply matches the AffectedComponentEntry that belongs to portInst and assigns the vote to voteCommit. In addition, it sets reply to true to indicate that the voting result of the corresponding child has been received.

Figure A.76 shows the story diagram allRepliesReceived that checks whether all affected children have submitted their result of the voting phase. Therefore, the story diagram matches

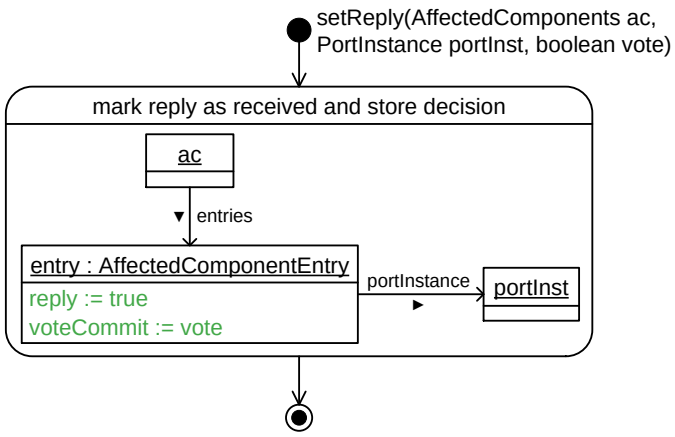


Figure A.75: Story Diagram Specifying the Behavior of setReply

an AffectedComponentEntry where reply is still false. If the matching succeeds, then it returns false because there still exists at least one child that has not yet submitted the voting result. Otherwise, it returns true.

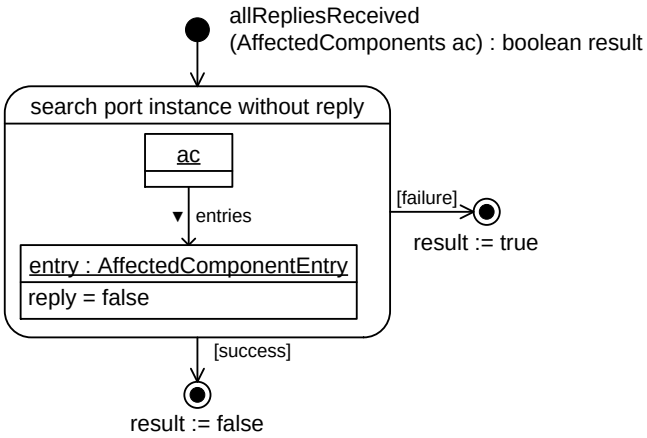


Figure A.76: Story Diagram Specifying the Behavior of allRepliesReceived

Figure A.77 shows the story diagram canCommit that decides whether the executor can commit the reconfiguration. Therefore, the story diagram matches an AffectedComponentEntry where voteCommit is false. If the matching succeeds, then at least one child aborted the reconfiguration. Then, the story diagram returns false to indicate that the reconfiguration cannot be committed. Otherwise, the story diagram returns true and the reconfiguration will be executed.

Figure A.78 shows the story diagram getNextPortInstanceForAction that returns a subport instance of embeddedCI that has not yet sent its execute or abort message to the child. Based on the input parameter ac, it matches an AffectedComponentEntry where action is false. Then, it sets action to true and returns the portInstance that belongs to entry.

Figure A.79 shows the story diagram allActionsPerformed that checks whether all affected children have received their execute or abort message. Therefore, the story diagram matches

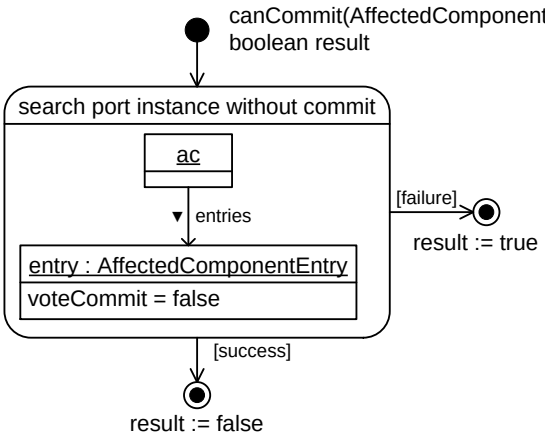


Figure A.77: Story Diagram Specifying the Behavior of canCommit

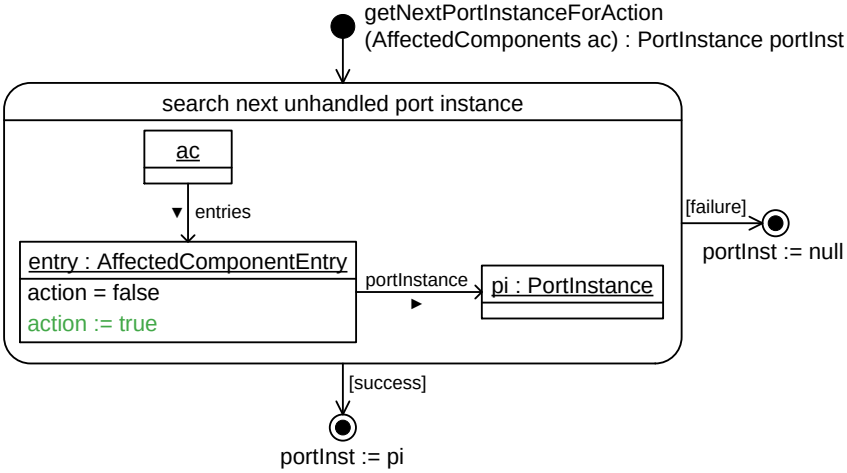


Figure A.78: Story Diagram Specifying the Behavior of getNextPortInstanceForAction

an AffectedComponentEntry where action is still false. If the matching succeeds, then it returns false because there still exists at least one child that has not yet received its message. Otherwise, it returns true.

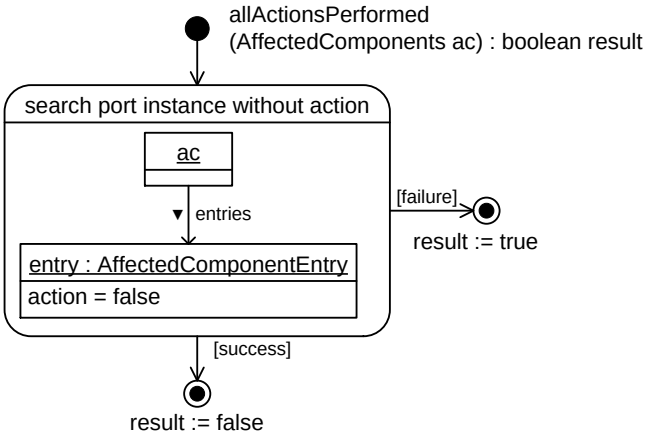


Figure A.79: Story Diagram Specifying the Behavior of allActionsPerformed

Figure A.80 shows the story diagram setFinished that marks that a child has finished its reconfiguration. Therefore, the story diagram matches the AffectedComponentEntry that belongs to the corresponding subport instance of embeddedCI. Then, it sets finished to true.

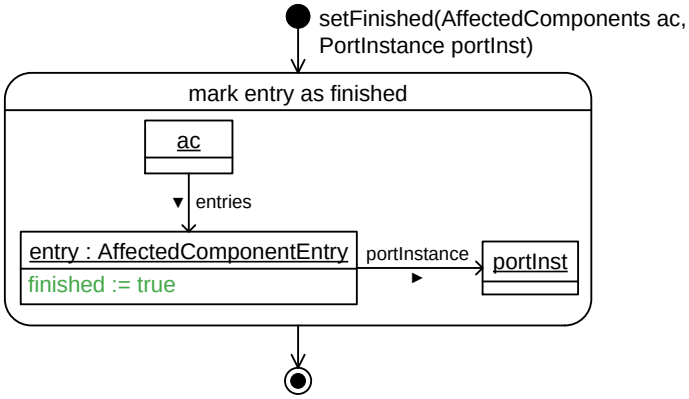


Figure A.80: Story Diagram Specifying the Behavior of setFinished

Figure A.81 shows the story diagram allEmbeddedFinished that checks whether all children have finished their reconfigurations. Therefore, the story diagram matches an AffectedComponentEntry where finished is still false. If the matching succeeds, then there exists at least one child that has not yet finished its reconfiguration. Then, the story diagram returns false. Otherwise, it returns true.

Figure A.82 shows the story diagram resetActionPerformed that resets the values of action and finished back to false for all AffectedComponentEntries. This function enables to reuse the attributes action and finished for all phases while executing a reconfiguration based on three-phase execution.

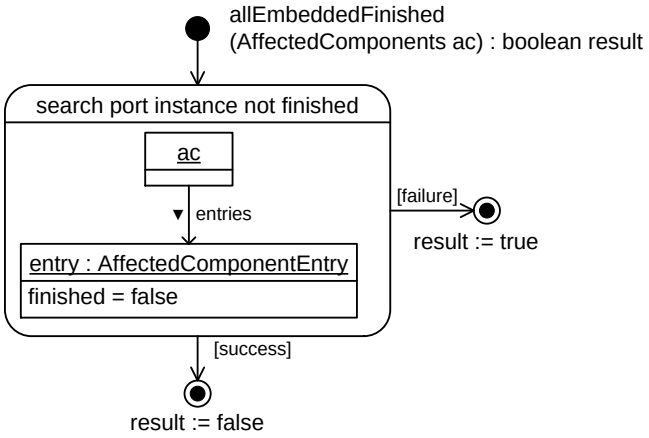


Figure A.81: Story Diagram Specifying the Behavior of allEmbeddedFinished

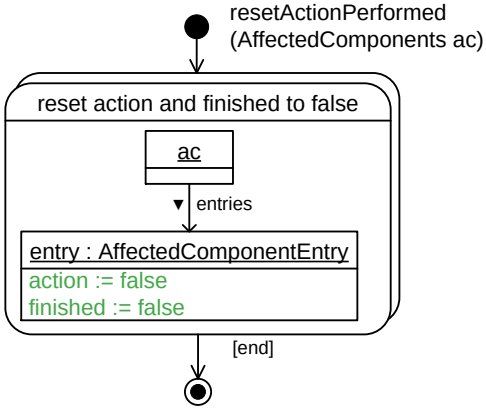


Figure A.82: Story Diagram Specifying the Behavior of resetActionPerformed

A.7 Component SDDs

This section presents the component SDDs that we use in our RailCab model for expressing component properties based on the current software architecture of the RailCab and for defining invariants. We introduce the component SDDs component-wise starting with RailCabDriveControl in Section A.7.1. Thereafter, we describe the component SDDs of ConvoyCoordination (Section A.7.2), VelocityController (Section A.7.3), OperationStrategy (Section A.7.4), and RefGen (Section A.7.5).

A.7.1 RailCabDriveControl

We already introduced three component SDDs for the component RailCabDriveControl, namely isCoordinator, isStandalone, and convoyOrder, in Section 3.5. We do not repeat these component SDDs in this section.

Figure A.83 shows the component SDD isMember. The component SDD formalizes the component property that defines that a RailCab operates as a member of a convoy. Therefore, the component story pattern in the first pattern node simply matches an instance of MemberControl. If this instance can be matched, the component SDD is fulfilled, otherwise it is not fulfilled.

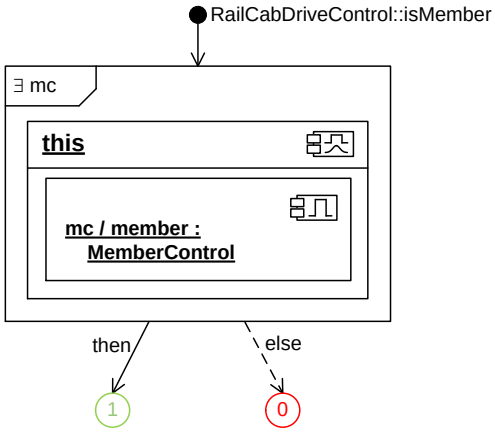


Figure A.83: Component SDD isMember for Component RailCabDriveControl that Specifies that an Instance of the Component Operates as a Convoy Member

Figure A.84 shows the component SDD convoyDisabled. This component SDD formalizes the property that the instance of RailCabDriveControl will not engage in convoys. Therefore, the component story pattern in the first story node matches an instance of OperationStrategy including the broadcast port instances of RailCabaDriveControl and os. If os including the broadcast port instances and the delegation connector instance could be matched, then the component SDD is not fulfilled. In this case, the instance of RailCabDriveControl may still engage in convoys via the broadcast port instance. Otherwise, the component SDD is fulfilled.

Figure A.85 shows the invariant component SDD validConvoyState. This component SDD specifies that a RailCab may not be coordinator and member of a convoy at the same time for being in a valid convoy state. The component story pattern expresses this fact by matching instances of both, MemberControl and ConvoyCoordination with the attached PositionSensor. If

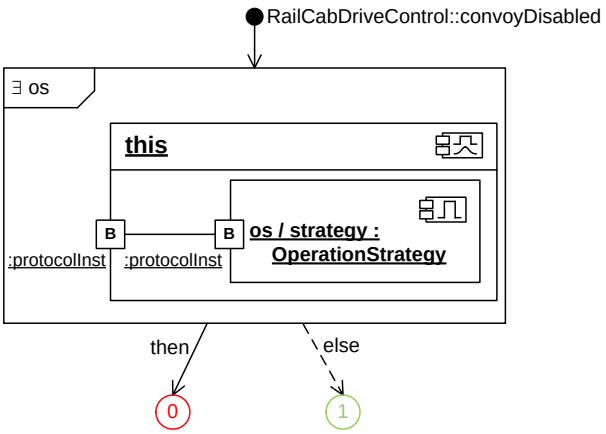


Figure A.84: Component SDD convoyDisabled for Component RailCabDriveControl that Specifies that an Instance of the Component will not Engage in Convoys

both instances may be matched for an instance of RailCabDriveControl, then the invariant is violated, otherwise it is fulfilled.

A.7.2 ConvoyCoordination

Figure A.86 shows the invariant component SDD convoyOrder of ConvoyCoordination. This component SDD defines an invariant that ensures that the subport instances of the refDistProvider and the RefGen instances have the same order.

The first pattern node matches the first subport instance of refDistProvider and the first RefGen instance in the sequence of RefGen instances. The first RefGen instance needs to have an instance of the curPos port that needs to be delegated to the curPos instance of ConvoyCoordination. We require that any instance of ConvoyCoordination contains at least one instance of RefGen. Therefore, the component SDD is not fulfilled if the first pattern node cannot be matched.

The second pattern node is a universal pattern node that matches all pairs of subsequent subport instances of refDistProvider. These are used in the third pattern node for checking the correct order of the RefGen instances. Therefore, the third pattern node matches two instances of RefGen using the variables rg1 and rg2. If rdp2 is delegated to rg1 and rdp3 is delegated to rg2, then rg1 and rg2 need to be connected by an assembly connector instance. The assembly connector instance needs to connect the next port instance of rg1 to the prev port instance of rg2. If the third pattern node may be matched for any pair of subsequent subport instances of refDistProvider, then the invariant holds, otherwise it is violated.

A.7.3 VelocityController

We use three component SDDs for the component VelocityController in our example. We introduce these in the following.

Figure A.87 shows the component SDD inStandaloneCtrl that formalizes the component property that the VelocityController executes the feedback controller for driving alone or as a

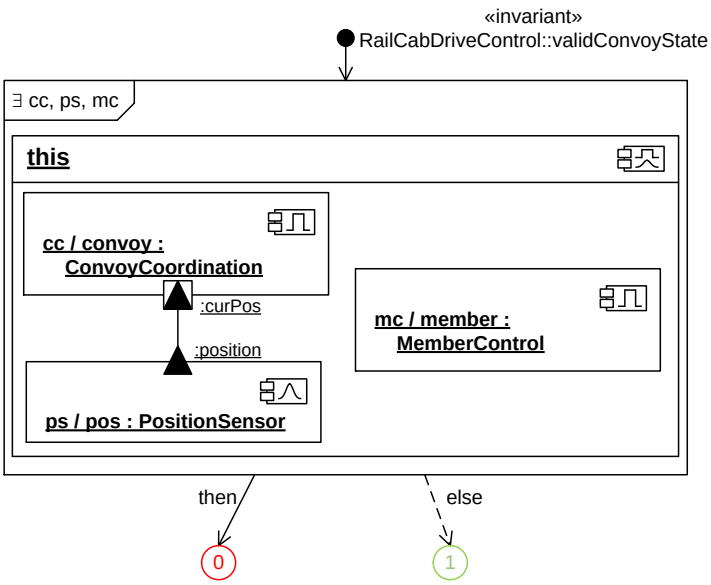


Figure A.85: Invariant Component SDD validConvoyState for Component RailCabDriveControl that Defines that a RailCab may not be Coordinator and Member at the Same Time

coordinator. Therefore, the first story node matches an instance of the StandaloneDrive component that contains the corresponding feedback controller. In addition, the component story pattern of the first story node matches an instance of the fading component of type ConvoyFading. Finally, it checks whether the instance of StandaloneDrive is connected to the fading component instance by an assembly connector instance such that the output of sd is forwarded by the fading component. If the component story pattern can be matched successfully, the component SDD is fulfilled.

Figure A.88 shows the component SDD inConvoyCtrl that formalizes the component property that the VelocityController executes the feedback controller for driving as a member. Therefore, the first story node matches an instance of the ConvoyDrive component that contains the corresponding feedback controller. In addition, the component story pattern in the first story node matches an instance of the fading component of type ConvoyFading. Finally, it checks whether the instance of ConvoyDrive is connected to the fading component instance by an assembly connector instance such that the output of cd is forwarded by the fading component. If the component story pattern can be matched successfully, the component SDD is fulfilled.

Figure A.89 shows the invariant component SDD validCtrl. This component SDD consists of three pattern nodes. The first pattern node is identical to the pattern node of inConvoyCtrl. Thus, it denotes that the VelocityController executes the feedback controller for driving as a member. If this pattern node can be matched successfully, the second pattern node at the lower left denotes that additionally an instance of StandaloneDrive is instantiated and connected to f. This situation may only occur while performing a reconfiguration but it may not occur before and after a reconfiguration. Therefore, we consider the invariant as violated if the second story can be matched as well. Otherwise, the invariant component SDD holds.

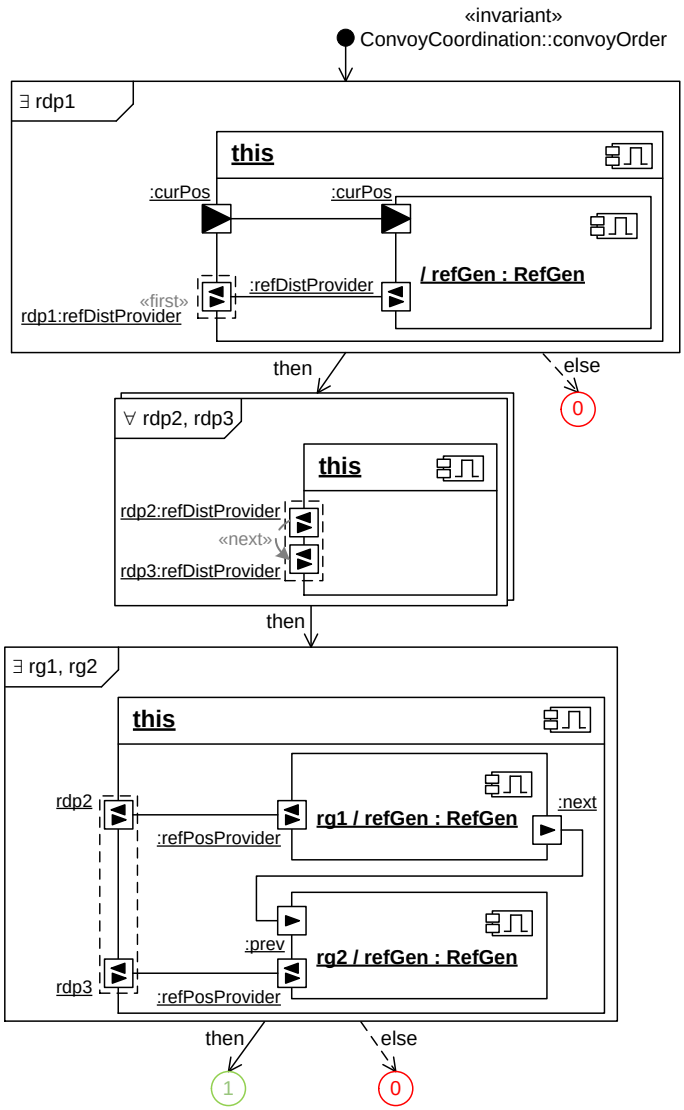


Figure A.86: Invariant Component SDD convoyOrder for Component ConvoyCoordination for Specifying a Correct Order of the RefGen Instances

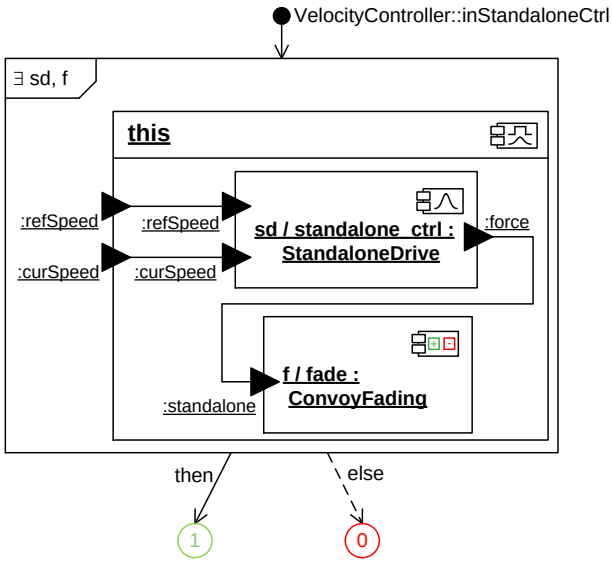


Figure A.87: Component SDD inStandaloneCtrl for Component VelocityController for Specifying that an Instance of the Component Executes the StandaloneDrive Controller

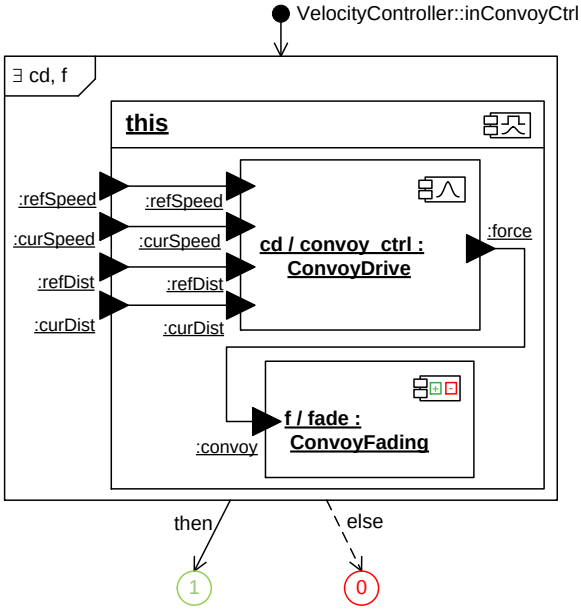


Figure A.88: Component SDD inConvoyCtrl for Component VelocityController for Specifying that an Instance of the Component Executes the ConvoyDrive Controller

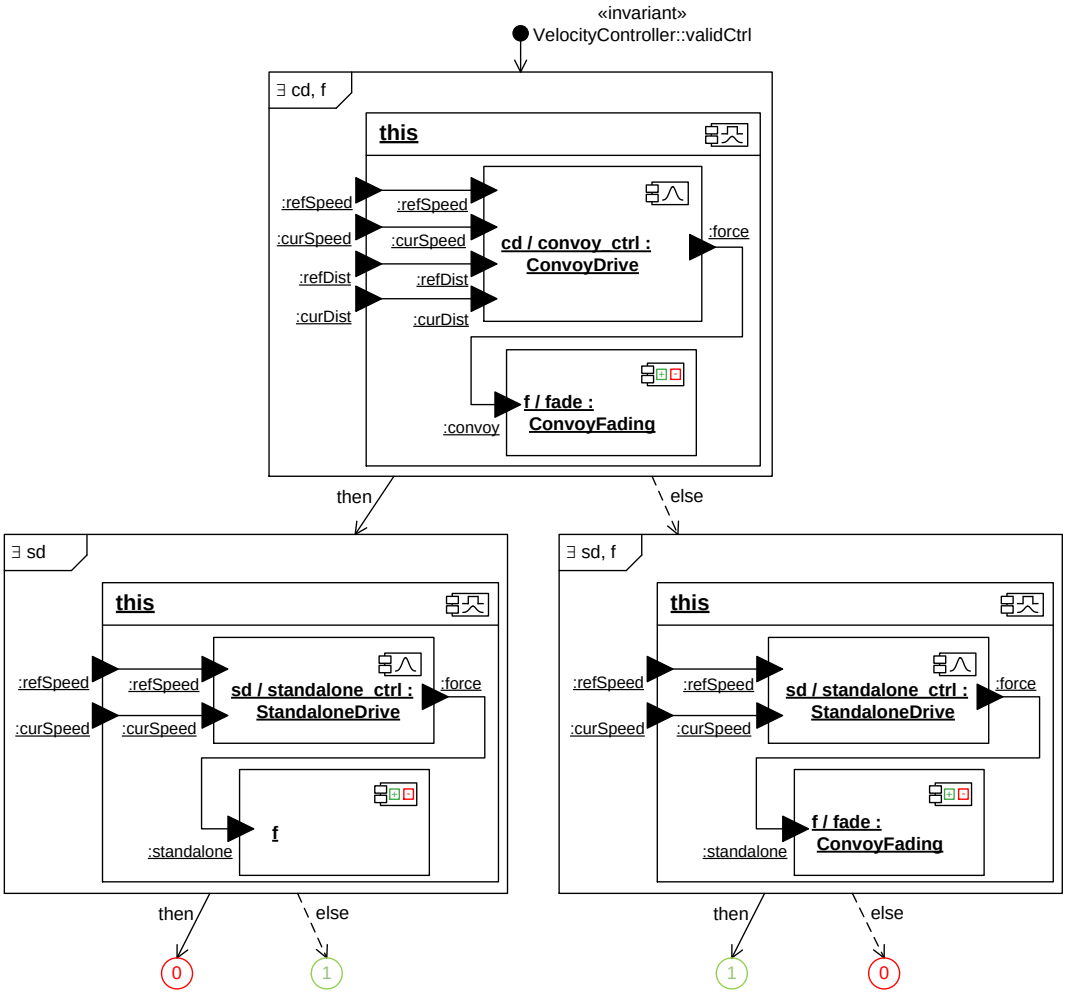


Figure A.89: Invariant Component SDD validCtrl for Component VelocityController for Specifying that an Instance of the Component does not Execute both Controllers at the Same Time

If the first pattern node cannot be matched, the third pattern node at the lower right is matched. It is identical to the pattern node of inStandaloneCtrl. If this pattern node cannot be matched, then the instance of VelocityController does not execute any feedback controller. This situation shall never occur and, thus, we consider the invariant component SDD as violated if the third pattern node cannot be matched. Otherwise, the invariant component SDD holds.

A.7.4 OperationStrategy

We use two component SDDs for the component OperationStrategy in our example that are used by the peer region of the component RTSC in Figure A.34. We introduce these in the following.

Figure A.90 shows the component SDD inCoordinatorMode that formalizes the component property that the instance of OperationStrategy is executed in a coordinator RailCab. In a coordinator RailCab, the instance of OperationStrategy needs to have instances of speedProvider and strategySender as shown in Figure A.27. Thus, the component story pattern in the pattern node matches these two port instances and the component SDD is fulfilled if the matching succeeds.

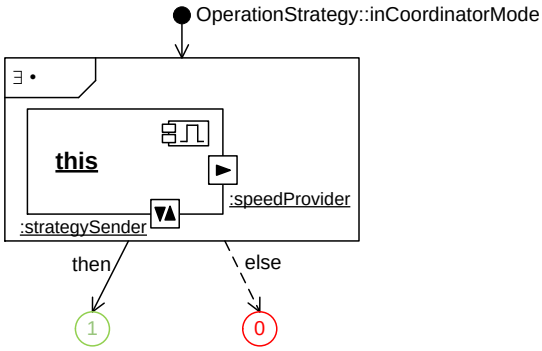


Figure A.90: Component SDD inCoordinatorMode for Component OperationStrategy for Specifying that an Instance of the Component Operates in a Coordinator RailCab

Figure A.91 shows the component SDD inMemberMode that formalizes the component property that the instance of OperationStrategy is executed in a member RailCab. In a member RailCab, the instance of OperationStrategy may not have an instance of speedProvider as shown in Figure A.31 because the reference speed is solely defined by the coordinator of the convoy. Thus, the component SDD matches this port instance and the component SDD is fulfilled if the matching fails.

A.7.5 RefGen

We use two component SDDs for the component RefGen in our example that are used by the component RTSC shown in Figure A.38. We introduce these in the following.

Figure A.92 shows the component SDD isFirst that formalizes the component property that the instance of RefGen is the first one in the sequence of RefGen instances. Since the first instance of RefGen has an instance of curPos as shown in Figure 3.10, the component story pattern matches this port instance. The component SDD is fulfilled, if the matching succeeds.

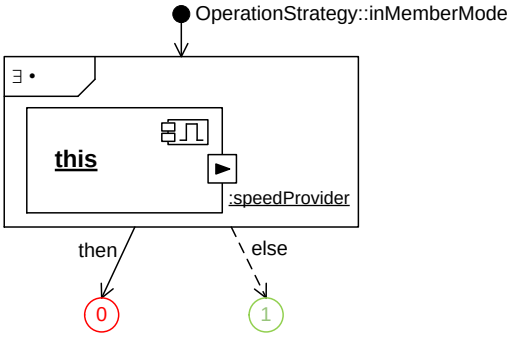


Figure A.91: Component SDD inMemberMode for Component OperationStrategy for Specifying that an Instance of the Component Operates in a Member RailCab

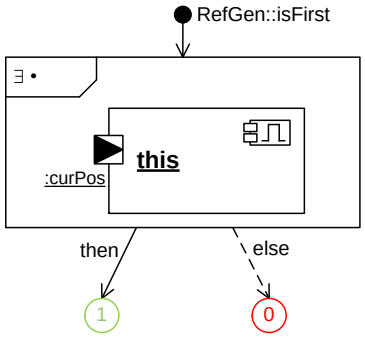


Figure A.92: Component SDD isFirst for Component RefGen for Specifying that an Instance of the Component is the First One in the Sequence of RefGen Instances

Figure A.92 shows the component SDD isLast that formalizes the component property that the instance of RefGen is the last one in the sequence of RefGen instances. Since the last instance of RefGen has no instance of the next port as shown in Figure 3.10, the component story pattern matches this port instance. The component SDD is fulfilled, if the matching fails.

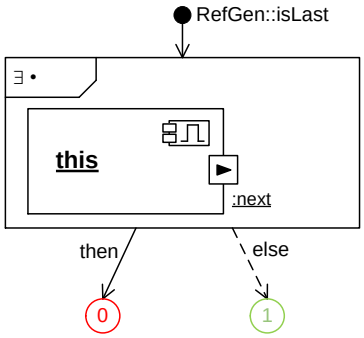


Figure A.93: Component SDD isLast for Component RefGen for Specifying that an Instance of the Component is the Last One in the Sequence of RefGen Instances

A.8 Excerpt of Generated MATLAB/Simulink Model

This section introduces examples of MATLAB/Simulink models that have been created based on our generation templates given in Sections 6.3 and 6.5.6. In Section A.8.1, we illustrate the result of translating an instance of the atomic component RefGen to Simulink. In Section A.8.2, we show the result of translating an instance of ConvoyCoordination (cf. Figure 3.10) to Simulink including the integration of the MATLAB-specific reconfiguration controller.

A.8.1 Simulink Model for Atomic Component Instance of Type RefGen

Figure A.94 shows the subsystem that has been generated for the discrete atomic component instance rg1 of type RefGen shown in Figure 3.10 on Page 49 using the generation template shown in Figure 6.6. The resulting subsystem has the same name as the component instance. The hybrid port instance curPos has been translated to an inport curPos of the subsystem rg1. The two discrete port instances refDistProvider and profileReceiver have been translated to port structures consisting of three inports and one outport.

Figure A.95 shows the internal structure of the subsystem rg1 in Figure A.94. The internal structure has been generated based on the generation template shown in Figure 6.7.

The resulting block diagram contains the chart block RefGen_Statechart and two link layer subsystems; one for refDistProvider and one for profileReceiver. The link layer subsystems are connected to the chart block using four signals that are used for transmitting the message buffers for received and sent messages from the link layer to the chart block and back again.

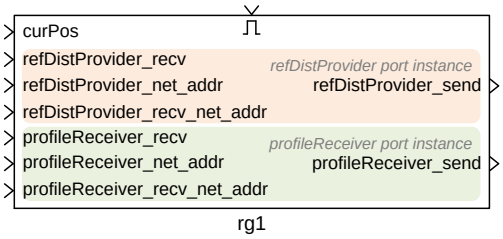


Figure A.94: Subsystem corresponding to Component Instance rg1 of Type RefGen

A.8.2 Simulink Model for Structured Component Instance of Type ConvoyCoordination

Figure A.96 shows the subsystem that has been generated for the structured component instance cc of type ConvoyCoordination shown in Figure 3.10 on Page 49 using the generation template shown in Figure 6.6. Again, the hybrid port instance curPos has been directly translated to an inport of the subsystem in Simulink. In addition, we obtain four port structures corresponding to the four discrete port instances c1, r1, receiver, and speedProvider of cc.

Figure A.97 shows the internal structure of the subsystem cc that results from translating the embedded CIC of the structured component instance cc. As a result of the first step of the translation, we obtain embedded subsystems cm and rg1 in Figure A.97 for the two eponymous embedded component instances. These subsystems have been created based on the template shown in Figure 6.6. Their internals are created by recursively applying the rule for translating atomic and structured component instances.

In Step 2 of our translation, we translate all connector instances between continuous and hybrid port instance by applying the generation template shown in Figure 6.12. As a result, we connect the inport curPos to the inport curPos of the embedded subsystem rg1 using a MultiSourceControl block.

In Step 3 of our translation, we translate all connector instances between discrete port instances. In particular, we translate all assembly connector instances according to the generation template shown in Figure 6.14 and all delegation connector instances according to the generation template shown in Figure 6.15. As a result, we obtain the communication switch shown in the middle of Figure A.96. In addition, all of the discrete port instances of the embedded component instances are connected to the bus creator and bus selector blocks that belong to the communication switch. Although all of these lines transport messages and are, thus, bus signals, we visualize them as normal lines for reducing the visual complexity of the figure at least a little bit. Finally, we obtain four delegation switch subsystems; one for each of the four discrete port instances of cc. These are connected, on the one hand, to the inports and the outport of the corresponding port structures. On the other hand, the delegation switches are connected to the communication switch.

The assembly and delegation connectors are defined by the addresses of the port structures. As an example, consider the assembly between p1 of cm and profileReceiver of rg1 in Figure 3.10. The port structure for p1 has net_addr 4 in Figure A.97, while the port structure for profileReceiver has net_addr 6. Then we set the recv_net_addr of the port structure for p1 to 6 and the recv_net_addr of the port structure for profileReceiver to 4 for realizing the assembly

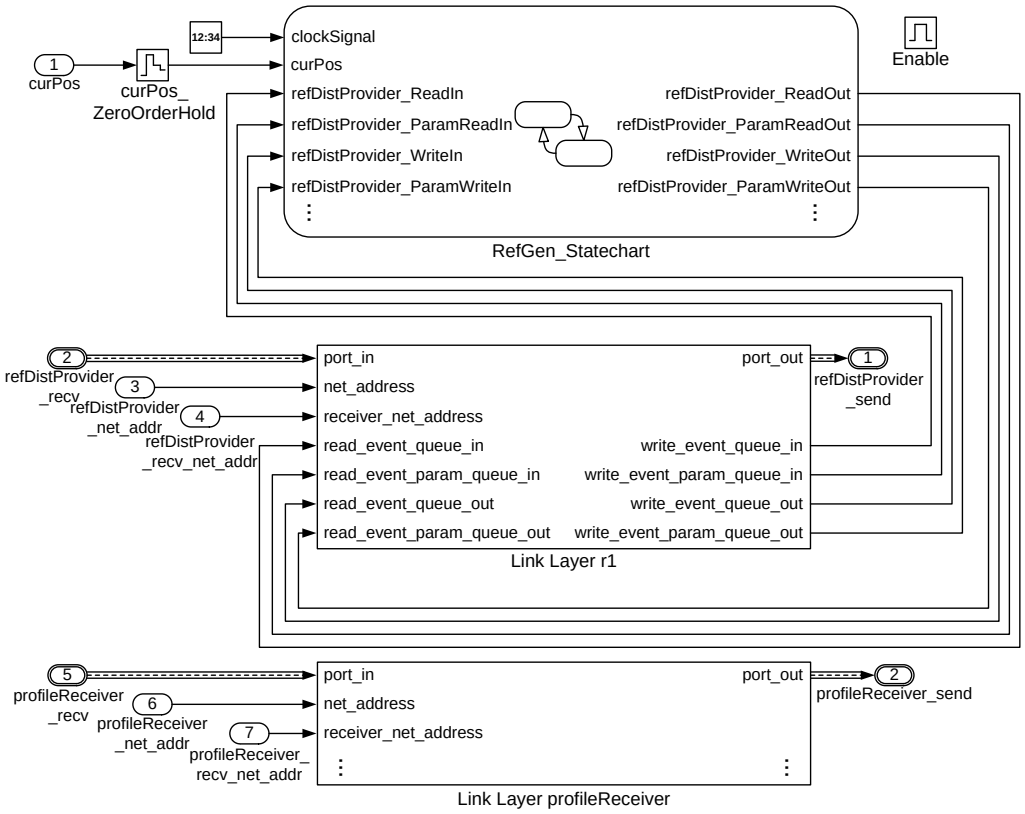


Figure A.95: Subsystem Corresponding to the Internal Structure of Atomic Component Instance rg1 of Type RefGen

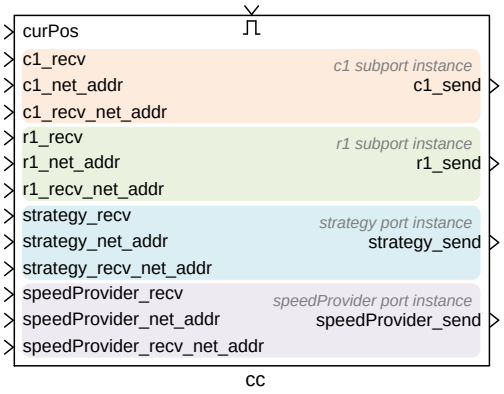


Figure A.96: Subsystem corresponding to Component Instance cc of Type ConvoyCoordination

connector instance in Simulink. Then, the communication switch ensures that all messages sent by either of the port structures arrives at the other port structure. Delegation connector instances are defined in the same way by using the local_net_addr of the delegation switch and the net_addr of the receiving port structure of the embbedded subsystem. As an example, consider the delegation from r1 of cc to refDistProvider of rg1. Then, the net_addr of the refDistProvider port structure, which is 5 in Figure A.97, is used as local_recv_net_addr of the delegation switch r1_DelegationSwitch and vice versa.

In Figure A.97, the assembly and delegation connectors have been encoded in a fixed, immutable way by using constant blocks for the recv_net_addrs. In order to obtain a reconfigurable subsystem, we need to apply Steps 1 to 5 of our translation as described in Section 6.5 and we need to integrate the MATLAB-specific reconfiguration controller into the Simulink model. Figure A.98 shows the Simulink model that results from adding the MATLAB-specific reconfiguration controller to the Simulink model shown in Figure A.97 using the generation template shown in Figure 6.31. For reducing the size of the figure, we omitted the communication switch including all connections from it and to it in the figure. In addition, we restrict ourselves to translating config1 contained in Figure 6.23 including the control signals as described in Section 6.5.4.

The subsystem Reconfiguration Controller in Figure A.98 contains the MATLAB-specific reconfiguration controller. The subsystem has inports and outports that correspond to the reconfMsg, reconfExec, and embeddedCI port instances of the reconfiguration controller (cf. Figure 6.24). We directly connect the embeddedCI inports and outports of the reconfiguration controller to their counterparts in cm and rg1. We use a direct connection in this case because these assembly connector instances are immutable, i.e., as long as cm is executed, the connection to the reconfiguration controller is active as well.

Furthermore, we obtain one outport at the reconfiguration controller for each control signal. The control signals cm and rg1 are connected to the enable ports of the subsystems cm and rg1. By setting a 0 to the control signal, we stop simulating them and emulate their destruction. By setting a 1 to the control signal, we start simulating them and emulate their creation. The control signals c1, rg1, receiver, and speedProvider, which correspond to the port instances of ConvoyCoordination, are connected to the local_recv_net_addr inports of the corresponding delegation switches. These control signals define the net_addr of the receiving port structure

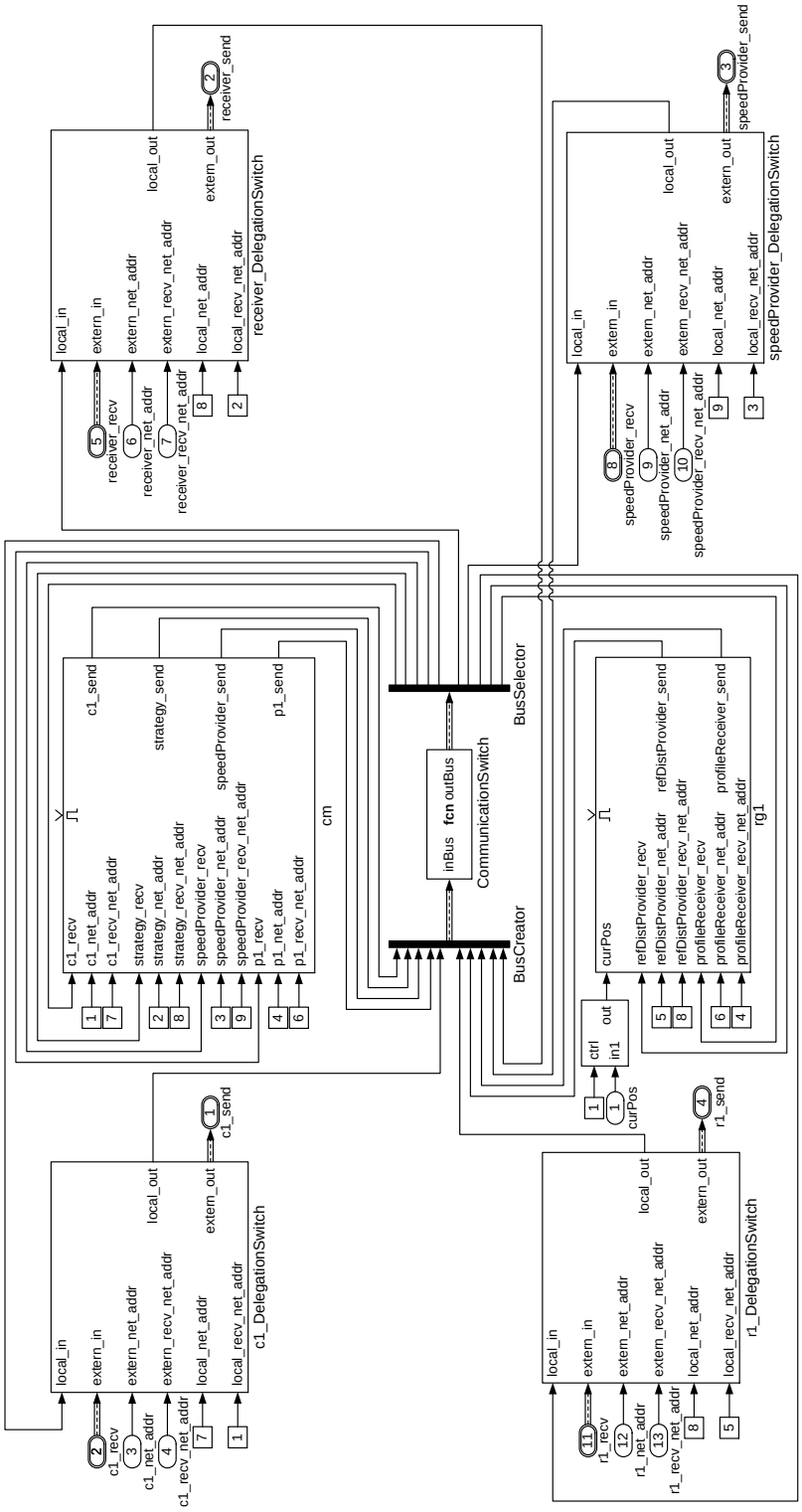


Figure A.97: Subsystem corresponding to the Embedded CIC of the Structured Component Instance cc of Type ConvoyCoordination

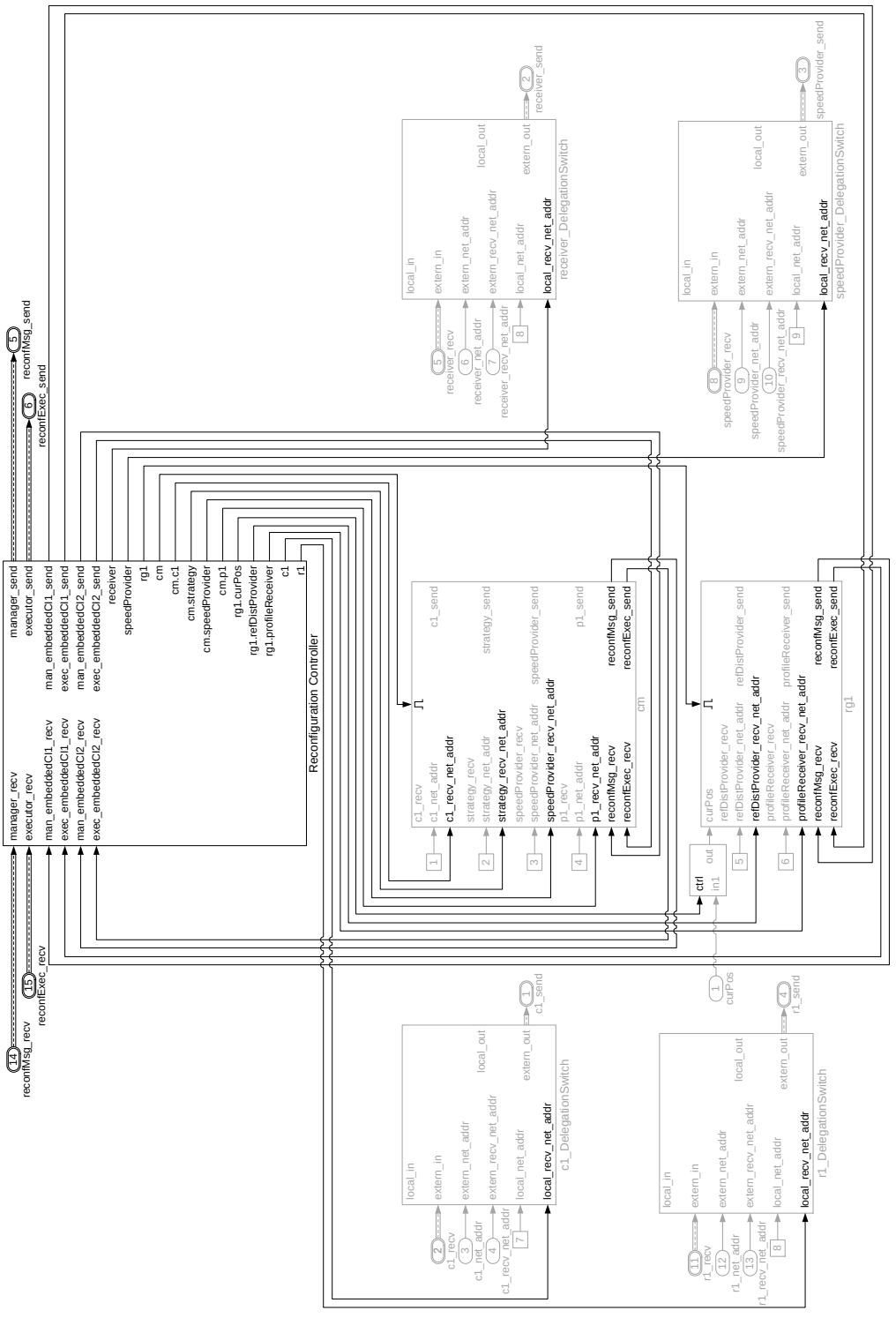


Figure A.98: Subsystem of Figure A.97 Including the Generated MATLAB-specific Reconfiguration Controller

for realizing the delegation connector instances. By changing the local_recv_net_addr via the control signal, we enable that the port instance is delegated to a different port instance of an embedded component instance. The control signal rg1.curPos is connected to the MultiSourceControl block of the curPos inport of rg1. By setting a 0 to the control signal, we stop delegating the inport curPos of cc to rg1. By setting a 1 to the control signal, we enable the delegation again. Finally, the control signals cm.c1, cm.p1, cm.strategy, cm.speedProvider, rg1.refDistProvider, and rg1.profileReceiver are connected to the recv_net_addr inports of the corresponding port structures of the subsystems cm and rg1. They define the net_addr of the receiving port structure. By changing the recv_net_addr, we can redirect assembly connector instances.

Figure A.99 shows the internal structure of the ReconfigurationController subsystem shown in Figure A.97. It has three embedded subsystems Manager, Executor, and ConfigurationStore that correspond to the three elements of the MATLAB-specific reconfiguration controller shown in Figure 6.24.

The Manager subsystem implements the manager of the MATLAB-specific reconfiguration controller. Therefore, we connect the inports manager_recv, man_embeddedCI1_recv, and man_embeddedCI2_recv as well as the outports manager_send, man_embeddedCI1_send, and man_embeddedCI2_send to this subsystem. The Executor subsystem implements the executor of the MATLAB-specific reconfiguration controller. Therefore, we connect the inports executor_recv, exec_embeddedCI1_recv, and exec_embeddedCI2_recv as well as the outports executor_send, exec_embeddedCI1_send, and exec_embeddedCI2_send to this subsystem. Finally, the Configuration Store subsystem implements the configuration store of the MATLAB-specific reconfiguration controller. Therefore, we connect all outports that correspond to control signals to the Configuration Store subsystem.

The internal connections resemble the three assemblies that are used by the MATLAB-specific reconfiguration controller. We use direct connections instead of a communication switch in this case because all three assemblies are immutable in MECHATRONICUML.

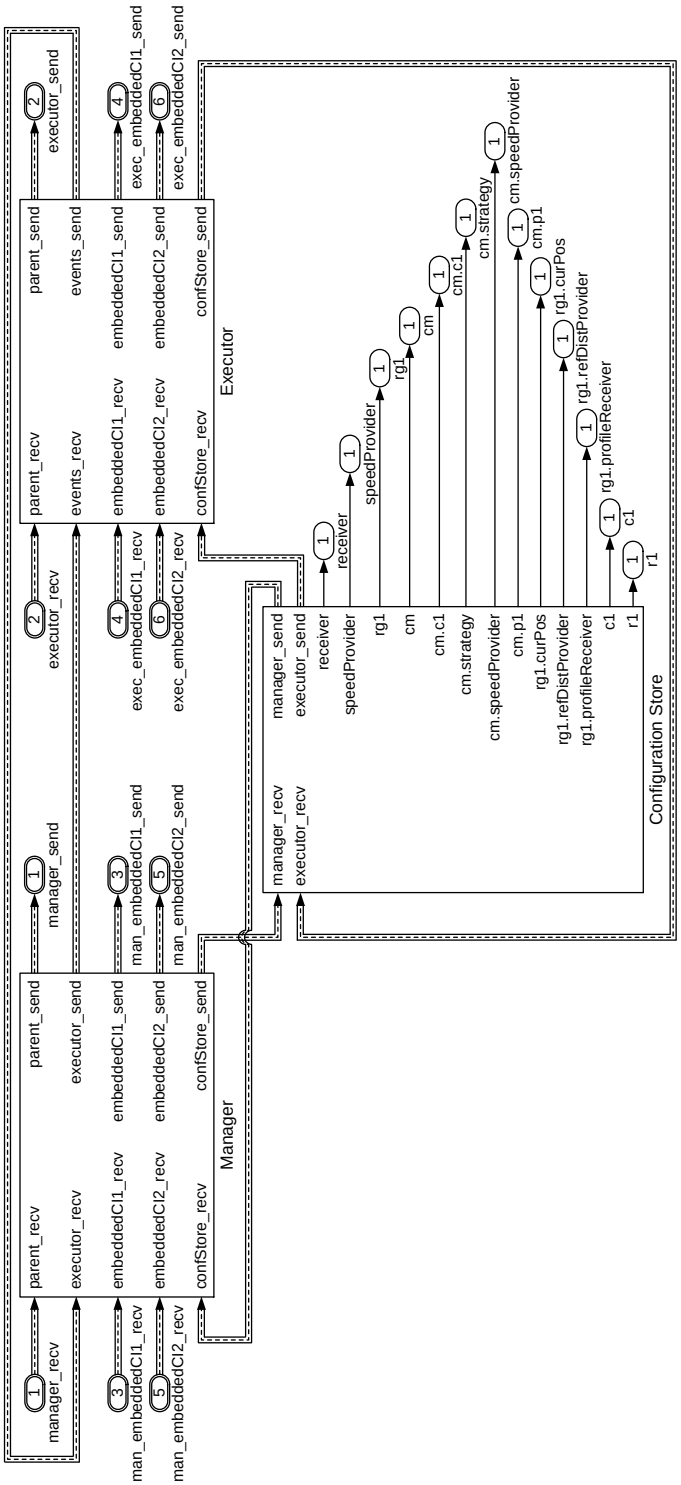


Figure A.99: Internal Structure of the ReconfigurationController Subsystem Generated for Component Instance cc of Type ConvoyCoordination

Appendix B

Formalization of the Real-time Statechart Semantics

This chapter introduces a formalization of the RTSC semantics that is used by our test automata-based refinement check in Chapter 5. The formal semantics of RTSCs is implemented by the reachability analysis for RTSCs described in Chapter C.

We formalize RTSCs based on networks of flat timed automata (cf. Section 2.2.1) that are formally defined by Bengtsson and Yi [BY04]. It is sufficient to consider networks of flat timed automata because all other features of hierarchical RTSCs can be mapped to this formalism. Hierarchical states may be flattened to a network of timed automata [DMY02, DMY03, Ger13]. Asynchronous communication using buffers may be mapped to additional timed automata representing the connector and buffer using shared integer variables for storing messages [KMR02, Ger13]. Deadlines as well as entry and exit actions may be resolved by intermediate states and transitions [GB03, DMY03]. Urgent transitions may be mapped to urgent channels using an additional automaton [DMY03].

Networks of timed automata as defined by Bengtsson and Yi, however, do not support time guards for urgent transitions. In addition, urgent transitions do not have precedence over non-urgent transitions in their approach. These two features are essential for the correctness of our test automaton construction in Section 5.3.2. Consequently, we need to provide a new definition of networks of timed automata that was informally introduced by Brenner [Bre10].

We start by defining the syntax of NTAs. First, we define clock constraints that are used as invariants and time guards.

Definition B.1 (Clock Constraint)
Let C be a set of real-valued clocks and V be a set of integer variables. A clock constraint *φ is a conjunctive formula of atomic clock constraints of the form $c_1 \sim x$ or $c_1 - c_2 \sim x$ for $c_1, c_2 \in C$, $\sim \in \{<, \leq, =, \geq, >\}$ and $x \in \mathbb{N} \cup V$. We use $\mathcal{B}(C)$ to denote the set of clock constraints. [BY04]*

Next, we define a simple expression language on integers which is the basis for defining variable updates and integer constraints that can be used as transition guards.

Definition B.2 (Integer Expression) *Let V be a set of integer variables. We define $Exp(V)$ the set of* integer expressions *over V. Each $exp \in Exp(V)$ is recursively defined by the rules:*

$$exp := x | v | (exp) | exp \sim exp$$

for $x \in \mathbb{Z}$, $v \in V$, and $\sim \in \{+, -, *, /\}$. (cf. [HR04, p. 260])

Definition B.3 (Integer Variable Constraint)
Let V be a set of integer variables. An integer variable constraint *ψ is a conjunctive formula of atomic integer variable constraints of the form $v \sim exp$ for $v \in V$, $\sim \in \{<, \leq, =, \geq, >\}$ and $exp \in Exp(V)$. We use $\mathcal{V}(V)$ to denote the set of integer variable constraints. [BGK+96]*

Using the above definitions, we can now define a single timed automaton with urgent transitions that may be used in a network of timed automata.

Definition B.4 (Timed Automaton with Urgent Transitions)
A timed automaton with urgent transitions *is a tuple $A = (L, l_0, C, V, \Sigma, E, U, I)$ where*

- *L is a finite set of locations*
- *$l_0 \in L$ is the initial location*
- *$C = C_L \cup C_G$ is a finite set of clocks where C_L is a set of local clocks and C_G is a set of global clocks*
- *$V = V_L \cup V_S$ is a set of integer variables where V_L is a set of local variables and V_S is a set of shared variables*
- *$\Sigma = (Ch \times \{?, !\}) \cup \{\tau\}$ is a finite set of events where Ch is a set of channels and τ is the empty event*
- *$E \subseteq L \times \mathcal{B}(C) \times \mathcal{V}(V) \times \Sigma \times 2^C \times U \times L$ is the set of transitions where $\varphi \in \mathcal{B}(C)$ is the time guard, $\psi \in \mathcal{V}(V)$ is the transition guard, $\lambda \in 2^C$ are the clock resets, U with $(v, exp) \in U$, $v \in V$, $exp \in Exp(V)$ is a set of assignments*
- *$U \subseteq E$ is the set of urgent transitions*
- *$I : L \to \mathcal{B}(C)$ assigns clock constraints to locations, the invariants.*

We shall write $l \xrightarrow{\varphi,\psi,\sigma,\lambda,u} l'$ when $(l, \varphi, \psi, \sigma, \lambda, u, l') \in E$. (cf. [BY04, BGK+96])

Next, we may define networks of timed automata with urgent transitions which concludes the definition of the syntax of NTAs.

Definition B.5 (Network of Timed Automata with Urgent Transitions)
A network of timed automata with urgent transitions *is a tuple $NTA = (\mathcal{A}, Ch, C_G, V_S)$ where*

- *$\mathcal{A}$ is a set of n timed automata with urgent transitions $A_1, \ldots, A_n$ with $n \in \mathbb{N}$ and $n \geq 1$*
- *Ch is a set of channels*
- *C_G is a set of global clocks*
- *V_S is a set of shared integer variables*

For all $A_i, A_j \in NTA$ with $i, j \in \{1, \ldots, n\}, i \neq j$: $L_i \cap L_j = \emptyset$, $C_i \cap C_j = C_G$, $V_i \cap V_j = V_S$ and $\Sigma_i = \Sigma_j = \Sigma$. (cf. [BY04])

All timed automata in the NTA share the same set of channels and, thus, the same set of events Σ. In addition, the NTA defines global clocks and shared integer variables that may be used by each automaton in the network. Moreover, we do not allow time guards at transitions using an urgent channel [BY04].

We continue with a definition of the operational semantics of an NTA. The semantics of an NTA is defined by a timed transition system [Alu99]. Since clocks of timed automata are real-valued, the timed transition system contains infinitely many states [BY04]. Therefore, we use the symbolic semantics based on *clock zones* that provides a finite timed transition system [Alu99, BY04]. We call that timed transition system the *zone graph* of the NTA.

The formalization of zone graphs requires a formalization of clock zones and federations which store the possible values of clocks. "A [clock] zone is the solution set of a clock constraint, that is the maximal set of clock assignments satisfying the constraint" [BY04].

Definition B.6 (Clock Zone, Federation) *Let C be a set of clocks and $\Phi \in 2^{\mathcal{B}(C)}$. A* clock zone *$z$ is a set of clock interpretations described by conjunction of clock constraints each of which puts a lower or upper bound on a clock or on the difference of two clocks, i.e., $z = \bigwedge_{\varphi \in \Phi} \varphi$. If C has k clocks, then z represents a convex set in the k-dimensional Euclidean space [Alu99]. A* federation *h is a disjunction of a set χ of convex clock zones, i.e., $h = \bigvee_{z \in \chi} z$ [DHLP06].*

In addition, we need integer variable value assignments to keep track of the values of integer variables in the NTA.

Definition B.7 (Integer Variable Value Assignment, Evaluation) *Let V be a set of integer variables. An* integer variable value assignment *$\nu : V \to \mathbb{Z}$ is an injective function that assigns a value out of $\mathbb{Z}$ to each variable in V. An* evaluation *is a function $\varpi : Exp(V) \times \nu \to \mathbb{Z}$ that evaluates an integer expression $\exp \in Exp(V)$ to an integer with respect to the integer variable value assignment ν.*

Using federations, we can define a symbolic state of an NTA.

Definition B.8 (Symbolic State of NTA) *Let NTA be a network of timed automata. A* symbolic state of *NTA is a tuple $s = (l, h, \nu)$ where l is a location vector that stores the active location for each automaton, h is a federation storing the possible clock interpretations, and ν is an integer variable assignment.*

In addition to these definitions, we need a function that returns the set of clock constraints that any urgent transition leaving a location enabled in a symbolic state uses as a time guard. At this point, we check whether the transitions are enabled except for their time guards. We need this function for specifying the delay operation because NTA may only delay until an urgent transition becomes enabled. In addition, we need this function for detecting time intervals where no urgent transition is enabled. In these time intervals, non-urgent transitions may fire. Please note that synchronizing transitions only fire urgently if both transitions are urgent.

Definition B.9 (Clock Constraints of Available Urgent Transitions) *Let NTA be a network of timed automata with urgent transitions with $A_i \in \mathcal{A} = (L_i, l_{0,i}, C_i, \Sigma, E_i, U_i, I_i)$. Let $s = (l, h, \nu)$ be a symbolic state of NTA. The set* clock constraints of available urgent

transitions *in s is defined by a function $\Xi : S \to \mathcal{B}(C)$ where $s \mapsto \Xi_\tau(s) \cup \Xi_\sigma(s)$ for $s \in S$ where*

- $\Xi_\tau : S \to \mathcal{B}(C)$ *where* $s \mapsto \{\varphi | \forall l_i \in l, \forall e_i \in U_i$ *with* $l_i \xrightarrow{\varphi,\psi,\tau,\lambda,u} l_i'$ *where* $\psi_i[v/\nu(v), exp/\varpi(exp,\nu)] \equiv true\}$ *for* $s \in S$
- $\Xi_\sigma : S \to \mathcal{B}(C)$ *where* $s \mapsto \{\varphi_j \wedge \varphi_m | \forall l_j \in l, \forall l_m \in l$ *with* $j \neq m, \forall e_j \in U_j, \forall e_m \in U_m$ *with* $l_j \xrightarrow{\varphi_j,\psi_j\sigma?,\lambda_j,u_j} l_j'$, $l_m \xrightarrow{\varphi_m,\psi_m,\sigma!,\lambda_m,u_m} l_m'$ *where* $\psi_j[v/\nu(v), exp/\varpi(exp,\nu)] \wedge \psi_m[v/\nu(v), exp/\varpi(exp,\nu)] \equiv true\}$ *for* $s \in S$

Given Definition B.9, we may now define the operational semantics of NTA based on a zone graph.

Definition B.10 (Zone Graph of NTA with Urgent Transitions)
Given an NTA with urgent transitions $NTA = (\mathcal{A}, Ch, V_G, C_G)$ with $A_i \in \mathcal{A} = (L_i, l_{0,i}, C_i, \Sigma, E_i, U_i, I_i)$, $i \in \mathbb{N}$. Its reachable state space is given by a zone graph *$Z = (S, s_0, T)$ where S is the set of symbolic states, s_0 is the initial symbolic state, and $T \subseteq S \times S$ is the set of transitions.*

For a symbolic state $s = (l, h, \nu)$, let l_i denote the i^{th} element of the location vector l representing the active location of A_i and $l[l_i'/l_i]$ the vector l with l_i being substituted with l_i'. In $s_0 = (l_{init}, h_{init}, \nu_{init})$, $l_{init,i} = l_{0,i}$ for all A_i, all clocks $c_{0j} \in h_{init}$ have value 0, and all integer variables $v_{0j} \in \nu_{init}$ are set to their initial values. Let $e_j = (l_j, \varphi_j, \psi_j, \sigma_j, \lambda_j, u_j, l_j') \in E_j$ and $e_m = (l_m, \varphi_m, \psi_m, \sigma_m, \lambda_m, u_m, l_m') \in E_m$ with $j \neq m$. $I(l) = \bigwedge_{l_i \in l} I(l_i)$ are the invariants of the active locations. $d_j = (v_j, exp_j)$ and $d_m = (v_m, exp_m)$ are the assignments of e_j and e_m. The transitions of the zone graph Z are defined by the rules:

1. $(l, h, \nu) \xrightarrow{\delta} (l, h', \nu)$ *with* $h' = relax(h^{\Uparrow} - ((\bigvee_{\varphi \in \Xi(l)} \varphi) - h^{\Downarrow})^{\Uparrow}) \wedge I(l)$
2. $(l, h, \nu) \xrightarrow{(j,\tau)} (l[l_j'/l_j], h', \nu')$ *if* $e_j = l_j \xrightarrow{\varphi_j,\psi_j,\tau,\lambda_j,u_j} l_j'$ *and* $\psi_j[v/\nu(v), exp/\varpi(exp,\nu)] \equiv true$ *where*
   - $h' = h_1$ *if* $e_j \in U_j$, $h' = h_2$ *otherwise where*
     - $h_1 = ((h \wedge \varphi_j)[\lambda_j \mapsto 0]) \wedge I(l[l_j'/l_j])$
     - $h_2 = ((h \wedge \varphi_j \wedge (\bigvee_{z \in \Xi(s)} \neg z))[\lambda_j \mapsto 0]) \wedge I(l[l_j'/l_j])$
   - $\nu' = [v_j \mapsto \varpi(exp_j, \nu)]$
3. $(l, h, \nu) \xrightarrow{((j,\sigma?),(m,\sigma!))} (l[l_j'/l_j][l_m'/l_m], h', \nu')$ *if* $e_j = l_j \xrightarrow{\varphi_j,\psi_j\sigma?,\lambda_j,u_j} l_j'$, $e_m = l_m \xrightarrow{\varphi_m,\psi_m,\sigma!,\lambda_m,u_m} l_m'$ *and* $\psi_j[v/\nu(v), exp/\varpi(exp,\nu)] \wedge \psi_m[v/\nu(v), exp/\varpi(exp,\nu)] \equiv true$ *where*
   - $h' = h_1$ *if* $e_j \in U_j \wedge e_m \in U_m$, $h' = h_2$ *otherwise where*
     - $h_1 = ((h \wedge \varphi_j \wedge \varphi_m)[\lambda_j \cup \lambda_m \mapsto 0]) \wedge I(l[l_j'/l_j][l_m'/l_m])$
     - $h_2 = ((h \wedge \varphi_j \wedge \varphi_m \wedge (\bigvee_{z \in \Xi(s)} \neg z))[\lambda_j \cup \lambda_m \mapsto 0]) \wedge I(l[l_j'/l_j][l_m'/l_m])$
   - $\nu' = [v_j \mapsto \varpi(exp_j, [v_m \mapsto \varpi(exp_m, \nu)])]$ *(cf. [BY04, BGK+96])*

In Definition B.10, Case 1 defines the new delay operation compared to the definition by Bengtsson and Yi [BY04]. Since urgent transitions do not allow the time to pass if they are enabled, time may only progress until an urgent transition gets enabled. The function Ξ returns the time guards of all urgent transitions (or pair of synchronizing urgent transitions) that are enabled, potentially except for their time guards. These clock constraints are combined into a single federation by disjuncting them. The operation $h^{\Downarrow}$ removes the lower bounds from the current federation h. By subtracting this federation from the disjunction of enabled urgent constraint, we remove all clock constraints that are *before* h, i.e., they may not be fulfilled by letting time progress. Then, we let time progress for the resulting federation. The resulting federation includes the earliest point in time where an urgent transition gets enabled. By subtracting this federation from the current federation $h^{\Uparrow}$, we obtain the time interval where no urgent transition is enabled. The $relax$ operation relaxes strict bounds ($<$ or $>$) to non-strict bounds ($\leq$ or $\geq$) and includes the single point in time where the urgent transition gets enabled into the federation. This construction only works if we restrict time guards at urgent transitions to non-strict clock constraints. Finally, we intersect against the invariants of the active locations.

Case 2 defines the conditions for firing a single transition. In contrast to Bengtsson and Yi [BY04], we need to give precedence to urgent transitions, i.e., as long as an urgent transition is enabled, no non-urgent transition is enabled. In general, a transition, either urgent or non-urgent, may only be fired if the time guard φ_j is true for the current federation h and if the transition guard ψ_j is fulfilled for the current integer variable value assignment ν. If the time guard is not fulfilled, the federation h' will be false. For urgent transitions, h' is defined by h_1. If the transition fires, the federation is updated by intersecting it with the time guard, applying the resets, and intersecting it with the invariants of the target locations. In addition, the integer variable value assignment is updated by applying the assignments of the transitions. For non-urgent transitions, we use h_2. For computing h_2, we first obtain the time guards of all urgent transitions using the function Ξ. By Definition B.1, these time guards are conjunctions of atomic clock constraints. Then, we negate each of these clock constraints for obtaining the time intervals where the urgent transition is *not* enabled. Then, we disjunct these negated clock constraints in a single federation. This federation combines all time intervals where no urgent transition is enabled. This federation is then conjuncted with the current federation h and the time guard φ_j of the transition e_j. As a result, e_j is restricted to time intervals where no urgent transition is enabled as intended.

Case 3 defines the conditions for firing two transitions that synchronize via a channel σ. Again, we need to distinguish between urgent and non-urgent transitions when computing the successor federation h'. Both cases, however, are identical to Case 2 but need to consider the time guards, resets, and integer variable value assignments of both transitions. The integer variable value assignments of the sending transition (e_m) are applied prior to the integer variable value assignments of the receiving transition (e_j).

Finally, we may define traces of an NTA that formalize counterexamples produced by a timed model checker.

Definition B.11 (Trace of NTA) *Let $Z = (S, s_0, T)$ be a zone graph of an NTA. A* trace ζ *is a path in Z such that $s_0 \rightarrow_T s_1 \rightarrow_T \ldots \rightarrow_T s_n$ where $\forall i, j \in \{0, \ldots, n-1\} : (s_i, s_{i+1} \in T) \wedge s_i = s_j \Rightarrow i = j$.*

A trace is a finite path in a zone graph that starts at the initial state and that does not contain any state twice.

Appendix C

A Framework for Reachability Analyses

As part of our implementation, we created a framework for conducting reachability analyses. A reachability analysis computes the state space of a given behavior model. Our framework consists of two core plugins that are independent of a concrete behavior model. They provide the basic state space traversal algorithm and a metamodel for storing the state space in a reachability graph (cf. Section C.1). Based on our framework, we implemented two reachability analysis algorithms; one for computing the state space of a story diagram specification (cf. Section C.2) and one for computing the state space of a set of RTSCs (cf. Section C.3) based on the formal semantics defined in Appendix B. The latter requires an implementation of clock zones (cf. Definition B.6) based on difference bound matrices (DBMs, [Dil90]) (cf. Section C.4).

Figure C.1 shows the plugins of the reachability analysis framework and the two reachability analyses that have been created based on the framework. We describe the figure in detail in the subsequent subsections.

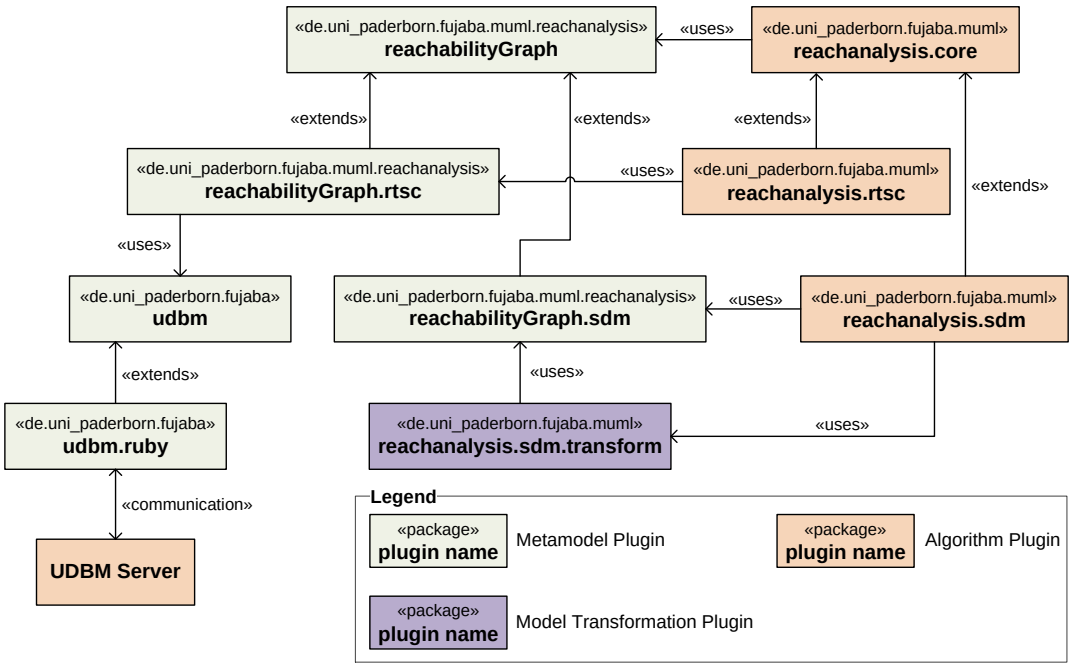


Figure C.1: Framework for Reachability Analyses

C.1 Reachability Analysis Framework

The reachability analysis framework consists of two plugins shown in Figure C.1. The plugin reachanalysis.core contains the main implementation of the algorithm that performs the state space traversal while reachabilityGraph contains the metamodel for storing the resulting reachability graph.

C.1.1 Metamodel

Figure C.2 shows a class diagram of the metamodel for storing the reachability graph. The class ReachabilityGraph represents the reachability graph that contains a set of ReachabilityGraphStates and ReachabilityGraphTransitions. Each ReachabilityGraphState represents one particular state of the state space of the behavior model. A ReachabilityGraphTransition connects one source ReachabilityGraphState to one target ReachabilityGraphState. In this case, the target ReachabilityGraphState has been derived from the source ReachabilityGraphState by a single execution step of the behavior model. Both classes are abstract. In particular, a concrete reachability analysis needs to define a concrete ReachabilityGraphState that contains the information of a state of the corresponding state space. The ActionTransition is a concrete subclass of the ReachabilityGraphTransition that may be used for an arbitrary execution step of the behavior model.

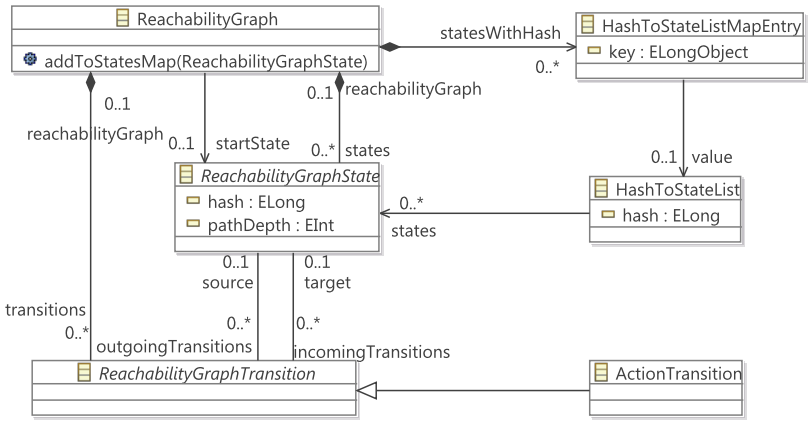


Figure C.2: Class Diagram of the Core Metamodel of the Reachability Analysis Framework

A ReachabilityGraphState has two attributes that are used by the reachability analysis algorithm. The pathDepth defines how many execution steps have been taken at least for reaching this ReachabilityGraphState from the startState of the ReachabilityGraph. This attribute is used for realizing a depth limitation in the state space traversal to ensure termination. The second attribute is a hash value of the ReachabilityGraphState. Each reachability graph state may receive a hash value that follows the general hashing constraint. In particular, it must hold for two ReachabilityGraphStates $obj1$ and $obj2$

$$obj1 \equiv obj2 \implies hash(obj1) = hash(obj2)$$

That means if two ReachabilityGraphStates are considered to be equivalent with respect to the behavior model, then these ReachabilityGraphStates must have identical hash values. This

information may be used to speed up the identification of equivalent states. Identifying equivalent states enables to reduce the computation and storage effort and may enable termination if the behavior model runs in a loop.

For managing the hash values, the ReachabilityGraph additionally contains a map where the hash value is the key and the value is a list of all ReachabilityGraphStates having this particular hash value. In EMF, this map is realized by the HashToStateListMapEntry and the HashToStateList.

C.1.2 Reachability Analysis Algorithm

Based on the metamodel as described above, the main reachability analysis algorithm is defined as shown in Algorithm 1. In essence, the algorithm is an adapted breadth first search (BFS, [Pea84, pp. 36-45]) that searches for a state satisfying a solution criterion. The solution criterion may be used, for example, to check for deadlocks or for identifying the error state in our refinement check (cf. Section 5.3). By using false as a solution criterion, we may compute the whole reachability graph of the behavior model. The behavior model to be explored needs to be set by a custom constructor of a concrete reachability analysis using the framework.

Algorithm 1 Core Algorithm for Computing the Reachability Graph

```
 1: function COMPUTEREACHABILITYGRAPH
 2:     reachabilityGraph := createReachabilityGraph()          ▷ Initialization Phase
 3:     initialize()
 4:     startState := createInitialState()
 5:     reachabilityGraph.startState := startState
 6:     computeHashValue(startState)
 7:     TODO.push(startState)
 8:
 9:     while TODO ≠ ∅ do                                       ▷ Expansion Phase
10:         curState := TODO.pop()
11:         if isPreSolution(curState) then
12:             return
13:         end if
14:         if curState.pathDepth < maxPathLength and not isDeadEnd(curState) then
15:             expand(curState)
16:         end if
17:         if isPostSolution(curState) then
18:             return
19:         end if
20:     end while
21: end function
```

The algorithm starts with an initialization phase where it first creates the ReachabilityGraph. The creation has been encapsulated into a function such that concrete reachability analyses may create more specific reachability graphs. Thereafter, the algorithm calls initialize in Line 3. This function enables concrete reachability analyses to initialize themselves, e.g., by

initializing further variables or performing a preprocessing. In the next step, the algorithm creates the initial state for the reachability graph. This function is abstract and needs to be implemented by a concrete reachability analysis. After creating the initial state, we set it as a start state for the reachability graph. Finally, we compute the hash value for the start state and add it to the TODO list. The TODO list contains all states that have not yet been expanded.

Then, the expansion phase starts, which is executed in a loop as long as there exists at least one state in the TODO list. The algorithm takes the state out of the TODO list and checks whether the state represents a solution. If so, the algorithm terminates. If not, the algorithm checks whether exploring the state will exceed the depth limitation and whether the state is a dead end. A dead end is a state that will not lead to a solution. If both is not the case, the algorithm expands the state and finally checks once again whether the state is a solution. We added an additional check for a solution after the expansion because we may only identify that a state represents a deadlock situation after expanding it. The code for identifying solutions is encapsulated in a separate class such that different strategies for identifying solutions may be used in combination with the same reachability analysis.

Algorithm 2 shows the function expand that is used for expanding a state. The expand function first calls computeSuccessors to obtain all states that may be reached from current state by a single execution step of the behavior model. Therefore, this function is abstract and needs to be implemented by a concrete reachability analysis. Thereafter, the function iterates all successors. First, it sets the path depth and, second, it invokes unifyStates in order to check whether the reachability graph already contains an equivalent state with respect to the behavior model.

Algorithm 2 The expand Function

```
1: function EXPAND(ReachabilityGraphState state)
2:     successors := computeSuccessors(state)
3:     for all s ∈ successors do
4:         s.pathDepth := s.pathDepth + 1
5:         unifyStates(s)
6:     end for
7: end function
```

The function unifyStates is given by Algorithm 3. First, unifyStates computes the hash value for the new state. Since the hash value depends on the behavior model, this function needs to be implemented by concrete reachability analyses. Then, unifyStates computes all candidates that might be equivalent to the new state in Line 4. Therefore, it utilizes the hash value and retrieves all states having the same hash. Then, unifyStates checks for each candidate whether it is equivalent to newState by calling isIsomorphic in Line 6. This function needs to be implemented by a concrete reachability analysis and returns true if and only if both states are equivalent with respect to the behavior model. If the new state is equivalent to an existing state, then we redirect the transition leading to the new state to the existing state (oldState).

If no equivalent existing state has been found, we execute Lines 13 to 14. These lines add the new state to the TODO list and to the reachability graph.

Algorithm 3 The unifyStates Function

```
 1: function UNIFYSTATES(ReachabilityGraphState newState)
 2:     computeHashValue(newState)
 3:     isoStateFound := false
 4:     candidates := reachabilityGraph.getStatesWithHash(newState.hash)
 5:     for all oldState ∈ candidates do
 6:         if isIsomorphic(oldState, newState) then
 7:             trans = newState.incomingTransition
 8:             redirectTransition(oldState, trans, newState)
 9:             isoStateFound := true
10:         end if
11:     end for
12:     if not isoStateFound then
13:         TODO.add(newState)
14:         reachabilityGraph.add(newState)
15:     end if
16: end function
```

C.2 Story Diagram Reachability Analysis

Based on our framework introduced in Section C.1, we implemented a reachability analysis on story diagrams. The reachability analysis and the accompanying metamodel extension are contained in the plugins reachanalysis.sdm, reachabilityGraph.sdm, and reachanalysis.sdm.transform in Figure C.1. Our reachability analysis computes all typed attributed graphs that may be derived from an initial graph based on a set of story diagrams. We use our reachability analysis for computing the possible configurations of a component instance, which is the basis for checking consistency in our transactional reconfiguration approach (cf. Section 4.5.1).

C.2.1 Metamodel Extension

Figure C.3 shows the metamodel extensions for the story diagram reachability analysis. The metamodel is built based on concepts presented by Zündorf [Zün09] and based on concepts presented in our previous publications [HSJZ10, HSE10]. The input to the reachability analysis is given by the GraphTransformationSystem that refers to all story diagrams to be used by the reachability analysis. Each story diagrams is modeled as an Activity [HRvD$^+$11]. In addition, the GraphTransformationSystem refers to classes of unchangeableNodes. Objects of these types will never be modified by the activities.

The SDMReachabilityGraph refers to the unchangeableNodes of the initial graph that is used for the reachability analysis. The initial graph is given as a set of EObjects including their references. As a utility reference, the SDMReachabilityGraph may contain all unchangeableNodes that are not contained elsewhere.

The states of the SDMReachabilityGraph are given by the StepGraph. Each StepGraph represents one graph that may be obtained based on the initial graph by applying a story diagram including the initial graph. The StepGraph refers to all changeable nodes using the change-

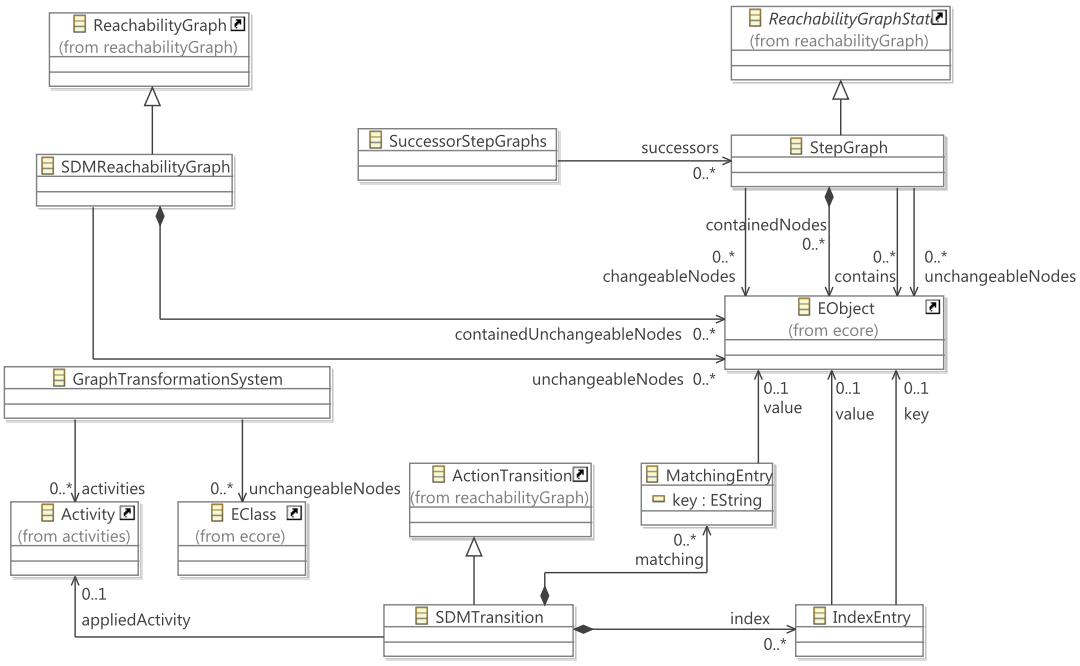


Figure C.3: Class Diagram of the Metamodel for Reachability Analysis on Story Diagrams

ableNodes reference. Optionally, it may contains these nodes via containedNodes if they are not contained elsewhere. In addition, it derives the unchangeableNodes from the SDMReachabilityGraph. Finally, contains is the union of changeableNodes and unchangeableNodes. SuccessorStepGraph is a helper class that is used for collecting all successors that have been computed via the computeSuccessors function in Line 2 of Algorithm 2.

The transitions of the SDMReachabilityGraph are given by the SDMTransition. The SDMTransition refers to the Activity that has been applied for deriving the target StepGraph. In addition, the SDMTransition contains two maps that may optionally be created. First, the MatchingEntry objects store the matching of the appliedActivity into the source StepGraph where the name of the object variable is used as the key. This enables to check which objects have been matched by which object variable for applying the story diagram. Second, the IndexEntry objects associate objects of the source StepGraph to the target StepGraph that are the same with respect to the underlying graph. In order to compute the whole reachability graph, deriving successors requires to copy the source StepGraph before applying the story diagram because applying the story diagram would destroy the source StepGraph otherwise. Then, the key refers to the object in the source StepGraph while value refers to the object in the target StepGraph that has been created as a copy. The use of the index has been reused from Zündorf [Zün09].

C.2.2 Functions of the Reachability Analysis

In addition to the metamodel extension, we also implemented the abstract functions of our reachability analysis framework. The algorithms for copying graphs and for computing hash values have been obtained from Zündorf [Zün09] but reimplemented using reflection in EMF [SBPM08]. We will not go into details on these functions but refer to the given

paper. In the following, we briefly describe our concepts for identifying unchangeable nodes (Section C.2.2.1), for computing successor graphs (Section C.2.2.2), and for computing isomorphisms between two graphs (Section C.2.2.3).

C.2.2.1 Unchangeable Node Detection

We consider a node of a graph as unchangeable if it will never be modified, either directly or indirectly, by applying a story diagram to the graph. By identifying nodes that will never be changed, we may remove these nodes from the single StepGraphs and store them only once for the whole SDMReachabilityGraph. This may reduce the size of the StepGraphs significantly. This, in turn, improves the performance of the reachability analysis because it requires fewer objects to be copied and fewer objects to be considered for computing hash values and isomorphisms. The computation of unchangeable nodes is performed as a part of the initialize function (cf. Algorithm 1).

We derive a conservative decision on which nodes are unchangeable by a static analysis of the story diagrams and the metamodels that are used as type graphs for the story diagrams. Our decision is conservative in a sense that it marks all nodes typed by the same class as changeable if one node type by this class might potentially be changed by a story diagram.

In order to define which nodes are unchangeable, we first define the conditions that make a node changeable. We consider eight conditions that make a node typed by a class A changeable. Conditions 1-4 provide conditions based on the story diagrams. Conditions 5-8 provide conditions based on the metamodels and explicitly consider inheritance and EMF's containment hierarchy.

1. There exists a story diagram that contains an object variable of type A with a binding operator «`create`» or «`destroy`» in one of its story patterns.
2. There exists a story diagram that contains an object variable of type A with an attribute assignment in one of its story patterns.
3. There exists a story diagram that contains a link variable typed by a reference originating from A that has binding operator «`create`» or «`destroy`».
4. There exists a story diagram that contains a link variable typed by a reference targeting A that has binding operator «`create`» or «`destroy`» and that has an opposite reference (making it bidirectional).
5. There exists a class B marked as changeable and nodes of type B (recursively) contain nodes of type A.
6. There exists a class B marked as changeable and nodes of type A (recursively) contain nodes of type B.
7. There exists a class B marked as changeable and A is (direct or indirect) subclass of B.
8. There exists a class B marked as changeable and A has a bidirectional reference to B.

In our analysis, we first analyze the story diagrams for Conditions 1-4, which gives an initial set of changeable nodes. Then, we expand this set by analyzing the metamodels used as type graphs for the story diagrams based on Conditions 5-8. In particular, Conditions 5-7 require the computation of closures on the containment and inheritance hierarchies and must be applied repeatedly.

The result is a set of classes where each node type by one of these classes is potentially changeable by the story diagrams. Then, a node x is unchangeable if x is contained in the initial graph and if x is typed by a class C that is not considered to be changeable by the above conditions.

C.2.2.2 Successor Computation

For computing the successors of a StepGraph, we need to apply all story diagrams to any possible matching that may be obtained for the StepGraph. This does not resemble the semantics of a regular story diagram, which is only applied to the first matching that can be obtained (not considering for-each nodes). At this point, we reused the idea by Zündorf [Zün09] also used in our previous publications [HSJZ10, HSE10] of enhancing the story diagrams such that they implement the necessary behavior for the reachability analysis. The benefit of this approach is that we may use any kind of story diagram interpreter [GHS09] or code generation [GSR05, GBD07, Zün09] as a black box.

The concept for enhancing the story diagrams by Zündorf [Zün09] has been automatized by a model transformation that is contained in the plugin reachanalysis.sdm.transform shown in Figure C.1. The transformation is illustrated abstractly in Figure C.4 for a story diagram sd1 shown on the left side of the figure. It contains an arbitrary number of story nodes starting with the story node A. This story diagram is transformed into the story diagram sd1_forEach shown on the right side of Figure C.4.

The model transformation works as follows. The first story node A becomes a for-each story node A_forEach that matches all possible matchings of sd1. This implies the restriction that the whole application condition for sd1 must be contained in A. A_forEach contains the LHS of the story pattern contained in A, i.e., all object variables without binding operator and all object variables with binding operator «destroy». However, the object variables having binding operator «destroy» in A have no binding operator in A_forEach in order to preserve step. In addition, A_forEach creates a new SDMTransition for each successful matching.

Then, we insert three additional activity nodes into the story diagram. The first one is a method call node that passes step and trans to the method copyState that creates an exact copy of step, called succ, and sets succ as the target of trans. The second one is a story node called Restore Matching. This node utilizes the index map (cf. Section C.2.1) to restore the matching that has been obtained based on step in A_forEach on the successor StepGraph succ. In addition, it adds succ to the set of SuccessorStepGraphs given by the object variable successors. Finally, the story node Execute A performs the rewrite step of the story pattern contained in A on succ using the restored matching. Thus, after executing the story pattern in Execute A, succ is isomorphic to the graph that would have resulted from applying the story node A of sd1 to step. After Execute A, the remaining activity nodes of sd1 are applied without modification to succ and the restored matching.

Finally, whenever sd1 enters a final node, we remove the final node and redirect the activity edge leading to the final node to A_forEach. Thus, after completely computing one successor, we backtrack to the for-each story node in order to search for another matching of A leading to another successor.

The model transformation that enhances the story diagrams is invoked as a part of the initialize function in Algorithm 1. In computeSuccessors, which is called in Line 2 of Algorithm 2, we invoke the story diagram interpreter [GHS09] using the enhanced story diagrams.

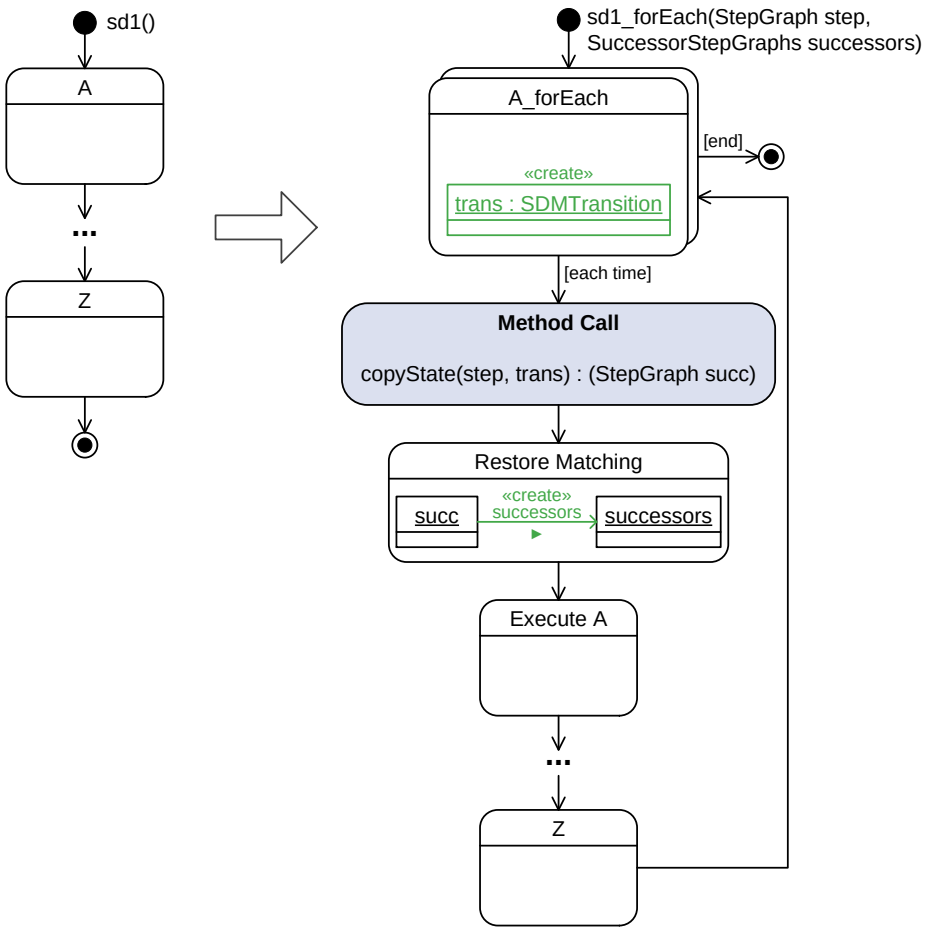


Figure C.4: Enhancing Story Diagrams for Computing Successors

After the interpreter finished executing a story diagram, all successors are contained in the SuccessorStepGraphs and passed to the unifyStates function in Algorithm 3.

C.2.2.3 Isomorphism Computation

The unifyStates function illustrated in Algorithm 3 requires an implementation of the isIsomorphic function that is invoked in Line 6. In our reachability analysis on story diagrams, this requires to decide whether two StepGraphs s_1 and s_2 are isomorphic. This, in turn, requires to compute a total bijective graph morphism iso that assigns each node of s_1 to exactly one node of s_2. In the general case, this computation is NP-hard but the consideration of typed attributed graphs allows us to reduce the computation effort significantly.

In the first step, we group the nodes of s_1 and s_2 based on their type such that we obtain one set for each type. Then, we check whether the number of elements in each set is the same for s_1 and s_2. If not, there cannot exist an isomorphism because there exists at least one node which cannot be mapped to a node of the same type by iso.

In the second step, we derive a partial mapping that contains all nodes where only one possible mapping exists. Therefore, we check all sets obtained in the first step whether they

contain only one element. After adding the corresponding pairs of nodes to the mapping, we evaluate their references whether they refer to a single node.

In the final step, we need to check all possible mappings for the remaining nodes. At this point, we may utilize the attribute values of the nodes because the attribute values of nodes that are mapped by iso need to be identical.

C.3 RTSC Reachability Analysis

Based on our framework introduced in Section C.1, we implemented a reachability analysis on RTSCs. The reachability analysis and the accompanying metamodel extension are contained in the plugins reachanalysis.rtsc and reachabilityGraph.rtsc in Figure C.1. Our reachability analysis computes a zone graph as defined in Definition B.10. We use the reachability analysis on RTSCs in our refinement check for deciding whether the error state is reachable.

C.3.1 Metamodel Extension

The input for the reachability analysis is an NTA as defined in Definition B.5. We encode the NTA based on the metamodel of RTSCs (cf. [BDG+14b, pp. 267ff]), i.e., all RTSCs that are contained in the NTA are contained in a single hierarchical state that defines all shared integer variables and synchronization channels of the NTA. Figure C.5 shows the metamodel for storing the resulting zone graph.

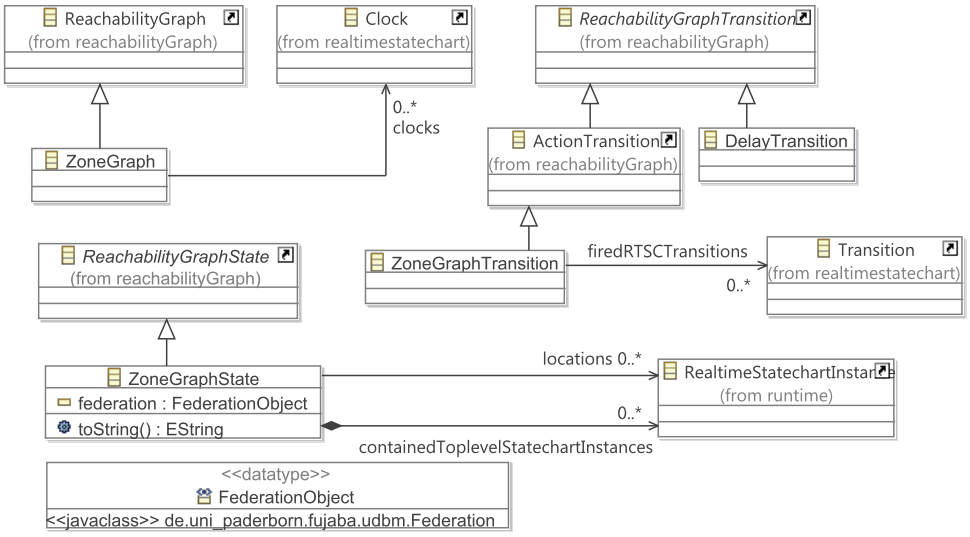


Figure C.5: Class Diagram of the Metamodel for Reachability Analysis on RTSCs

The class ZoneGraph represents the zone graph itself. The ZoneGraph refers to the Clocks of the RTSCs in order to associate them to the clocks of the UDBM library (cf. Section C.4). The ZoneGraphState inherits from ReachabilityGraphState and represents a single symbolic state of the NTA (cf. Definition B.8). Therefore, it contains a set of RealtimeStatechartInstances (cf. Section D.1.4) and refers to their active locations. The RealtimeStatechartInstance contains the

integer variable value assignments as defined in Definition B.7. In addition, the ZoneGraphState contains a federation that stores the values of the clocks of the RTSCs.

Finally, the metamodel defines two additional ReachabilityGraphTransitions, namely, the ZoneGraphTransition and the DelayTransition. A DelayTransition represents a δ transition of the zone graph (cf. Definition B.10). A ZoneGraphTransition represents both, τ transitions and transitions resulting from a synchronization. Therefore, the ZoneGraphTransition refers to the transitions of the RTSCs that were fired. In case of a τ transition, the firedRTSCTransitions reference refers to exactly one transition. In case of synchronizing transitions, the firedRTSCTransitions reference refers to exactly two transitions.

C.3.2 Functions of the Reachability Analysis

In addition to the metamodel changes, we also implemented the abstract functions of our reachability analysis framework. We implemented the computeSuccessors function called in Line 2 by the expand function shown in Algorithm 2. This function implements the three cases for successor transitions of Definition B.10. Furthermore, we implemented the isIsomorphic function for two ZoneGraphStates. Two ZoneGraphStates are equivalent if and only if the same states of the RealtimeStatechartInstances are active and if all variables of the RealtimeStatechartInstances have the same value for the integer variable value assignment and if the federations are equivalent. Two federations are equivalent if they allow for the same clock values for all clocks. The initialize function in Line 3 of Algorithm 1 converts all time units of the RTSCs to the smallest time unit in order to ease the computations of the clock values in the UDBM library (cf. Section C).

For the using the RTSC reachability analysis in our refinement check, we additionally implemented the isPreSolution and isDeadEnd functions used in Lines 11 and 14 of Algorithm 1. A ZoneGraphState is a solution if the error state of the test RTSC is active. A ZoneGraphState is a dead end if the neutral state is active.

C.4 UDBM Library

In our reachability analysis on RTSCs, we use DBMs [Dil90] for representing clock zones and federations (cf. Definition B.6). In particular, we integrated an existing DBM library of the UPPAAL model checker [Dav06] into our implementation. In Figure C.1, the integration of the library is given by the udbm plugin.

We chose a client-server architecture for integrating the DBM library [EH11]. The DBM library resides in a server component implemented in Ruby, called UDBM Server in Figure C.1, that interacts via sockets with the client that resides in the udbm.ruby plugin. The clock zones and federations are actually stored on the client side using the metamodel shown in Figure C.6. Whenever the client is requested to perform an operation on a federation, it encodes the federation and the requested operation and sends both to the server, which executes the operation. The resulting federation is sent back to the client and translated back into the metamodel. We refer to Eckardt and Heinzemann [EH11] for more information on the server implementation.

In the following, we give a brief description of the metamodel. Although the metamodel is visualized as a class diagram based on EMF, it is a plain Java library and independent

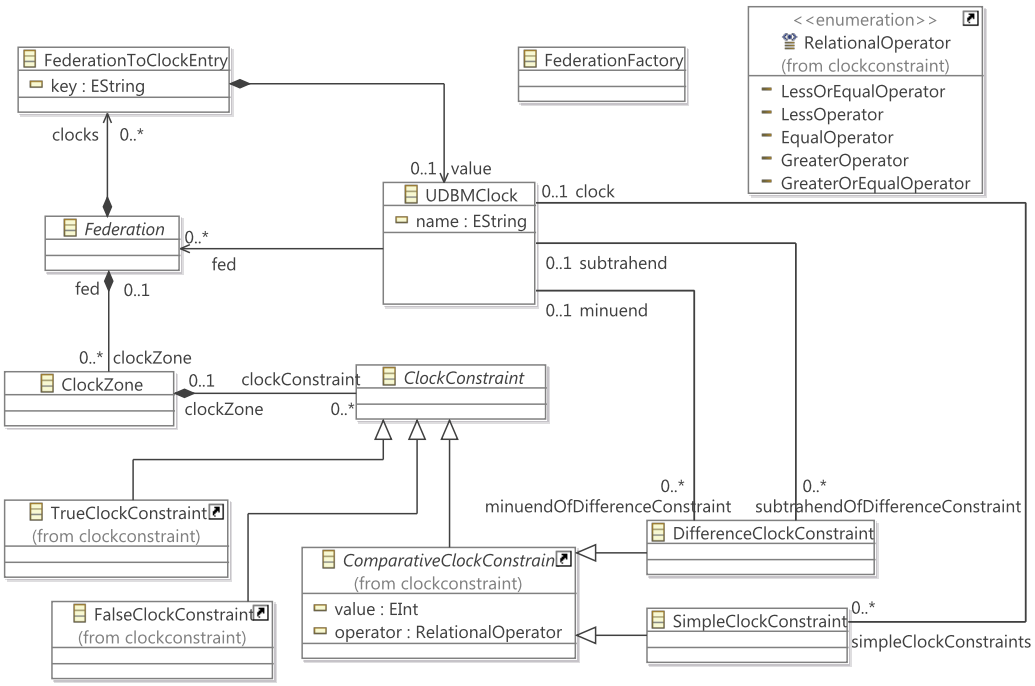


Figure C.6: Class Diagram of the Interface of the UDBM Library

of EMF. The basis of the metamodel is given by Federation that represents a federation of ClockZones that it contains. In addition, it contains a map of UDBMClocks that represent the clocks that are used in the clock zones. The clocks are identified by a unique key and may be shared among all federations.

A clock zone contains a set of ClockConstraints that represent the inequalities that put lower and upper bounds on each clock. We distinguish four kinds of clock constraints. These are the TrueClockConstraint, FalseClockConstraint, and two kinds of ComparativeClockConstraints, namely the SimpleClockConstraint and the DifferenceClockConstraint. The TrueClockConstraint represents a true value. It is only used for clock zones that do not restrict the values of the clocks at all. Analogously, the FalseClockConstraint represents a false value. It is used for empty clock zones that cannot be fulfilled by any assignment of values to the clocks. The ComparativeClockConstraints compare the values of clocks to an integer value using one of the RelationalOperators. A SimpleClockConstraint compares the value of a single clock to the value. A DifferenceClockConstraint put a condition on the difference of two clocks. It compares minuend − subtrahend to the value. Finally, the FederationFactory is used for creating new Federations and ClockZones.

Appendix D

Metamodels

In this chapter, we present the metamodels that we created as part of our implementation. Our metamodels provide a formal specification of the abstract syntax and the static semantics of our modeling languages. The abstract syntax has been specified using EMF [SBPM08]. The static semantics has been defined by OCL constraints [Gro12] that are contained in the EMF model. The operational semantics of RTSCs and CSDs, which are the two behavioral models of MECHATRONICUML, is defined based on timed automata and graph transformations as described in Chapter 3 and Appendix B.

In the following, we present our metamodels using class diagrams that visualize the abstract syntax. In particular, Section D.1 introduces the metamodel of the MECHATRONICUML component model for specifying components and components instances that do not employ runtime reconfiguration. Section D.2 introduces the metamodel for specifying reconfigurable components (cf. Chapter 3.6) including transactional execution of reconfigurations (cf. Section 4.6). Finally, we present the metamodels for MATLAB/Simulink and Stateflow that we created as part of our translation of MECHATRONICUML models to MATLAB/Simulink models (cf. Chapter 6.6). We refer to the MECHATRONICUML language specification [BDG+14b] for a listing of the OCL constraints that define the static semantics.

D.1 MechatronicUML Component Model

The metamodel of the MECHATRONICUML component model is separated into different packages. In this section, we present class diagrams for three of these packages. We first introduce several core classes in Section D.1.1 that provide a common basis for components and component instances. Thereafter, Sections D.1.2 and D.1.3 introduce the metamodels for specifying components and component instances that do not employ runtime reconfiguration.

D.1.1 Core

The classes shown in the class diagram in Figure D.1 are the super classes for all classes in the MECHATRONICUML metamodel. These classes have been developed as part of the new metamodel for story diagrams [HRvD+11]. We refer to them as core in the following. The classes shown in the class diagram in Figure D.2 form the basis for modeling components and component instances. They inherit from the core classes and we refer to them as component core in the following.

The root element of the core is the class ExtendableElement that, in combination with the class Extension, defines an extension mechanism. Each ExtendableElement may be extended by an arbitrary number of Extensions. An Extension enables to store additional information in

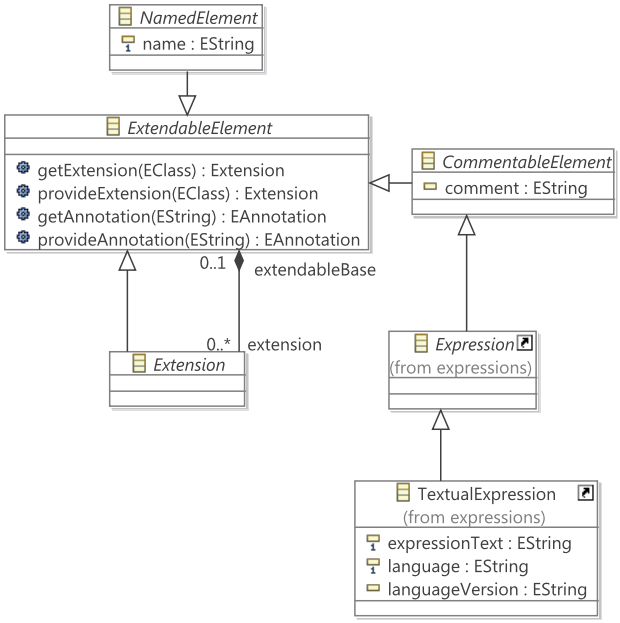


Figure D.1: Class Diagram of the Abstract Super Classes used by the MECHATRONICUML Metamodel

the metamodel without modifying it. The four operations of ExtendableElement have not been formalized based on OCL but implemented in Java. Although this contradicts with the aim of formally specifying the metamodel, we consider the extension mechanism to be useful. By using extensions, our metamodel is upward compatible, i.e., we may specify new modeling features and algorithms that require additional classes in the metamodel without needing to modify the existing metamodel and, thereby, breaking compatibility to older versions of our tooling.

In addition, the core package defines subclasses NamedElement and CommentableElement of ExtendableElement. These are used for all modeling elements of MECHATRONICUML that have a name or may be commented by the user. The class Expression is the root for the MECHATRONICUML action language that is used for specifying guards and actions in RTSCs (cf. Section 2.4.2). We refer to the MECHATRONICUML specification [BDG+14b] for a detailed overview of the action language metamodel. The TextualExpression enables to store expressions in an arbitrary textual language, e.g., Java or MATLAB Script, in a MECHATRONICUML model.

The component core shown in Figure D.2 defines ConnectorEndpoints and Connectors. ConnectorEndpoint is the super class for all metamodel elements that may be connected by Connectors such as roles or ports. Any ConnectorEndpoint has an arbitrary number of Connectors. In MECHATRONICUML, examples of connectors include assembly and delegation connectors. As a special type of ConnectorEndpoint, the component core defines DiscreteInteractionEndpoints. This type of ConnectorEndpoint has a Behavior (via super class BehavioralElement) and it sends or receives asynchronous messages. Therefore, it has a set of MessageBuffers where each MessageBuffer may store received messages of particular MessageTypes. The size of the message buffer is given as a NaturalNumber. In MECHATRONICUML, examples of DiscreteInteraction-

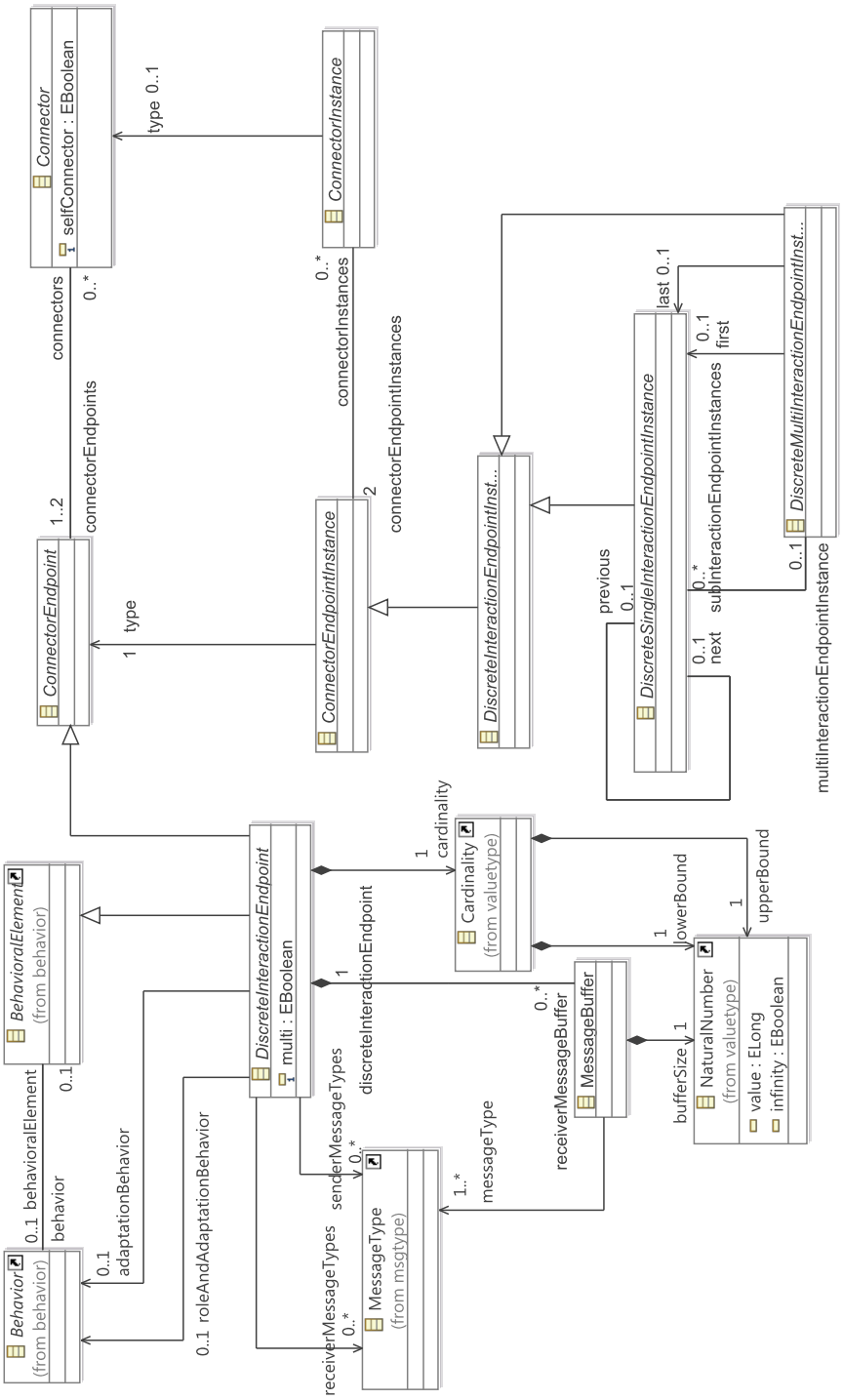


Figure D.2: Class Diagram of the Abstract Super Classes of Components and Component Instances

Endpoints are roles and discrete ports. Both have a Cardinality that defines a lower and upper bound based on a NaturalNumber.

In addition, the component core defines ConnectorEndpointInstances and ConnectorInstances. They are the super classes for all instances of ConnectorEndpoints and Connectors. Similar to components, we use DiscreteInteractionEndpointInstance as a special type of DiscreteInteractionEndpoint. For instances, we additionally need to distinguish between DiscreteSingleInteractionEndpointInstances and DiscreteMultiInteractionEndpointInstances. The former is the super class for single port instance and subport instances. The latter is the super class for multi port instances. Consequently, a DiscreteMultiInteractionEndpointInstance refers to a set of subInteractionEndpointInstances, which are the subport for a discrete multi port instance. In addition, we use references first, last, next, and previous that are used for specifying the order of a multi role or multi port (cf. Figure 2.13 on Page 28).

D.1.2 Components

Figure D.3 shows a class diagram of the component package. The component package defines the classes for modeling Components that are not reconfigurable. In the metamodel, they are referred as StaticComponents. In addition, we have abstract subclasses of Component for AtomicComponents and StructuredComponents. Finally, StaticAtomicComponent and StaticStructuredComponent represent non-reconfigurable atomic and structured components. A StructuredComponent contains a set of ComponentParts while a ComponentPart is typed over a Component.

Any Component has a set of Ports. In our metamodel, we distinguish betweeen DiscretePorts and DirectedTypedPorts. DirectedTypedPort is the super class for HybridPorts and ContinuousPorts. Both have a kind that defines their direction (either in-port or out-port), a type (via super class TypedNamedElement), and they may be optional. Furthermore, out-ports may be initialized by an initializeExpression using the MECHATRONICUML action language that defines a sane initial value that is emitted after instantiated the port. HybridPorts additionally define a samplingInterval based on TimeValue that provides a value and a TimeUnit.

ComponentParts refer to the Ports of the componentType by PortParts. Any PortPart is typed by a Port. Additionally, StructuredComponents embed a set of PortConnectors for connecting Ports and PortParts. We distinguish DelegationConnectors that connect a Port with a PortPart and AssemblyConnectors that connect two PortParts.

D.1.3 Component Instances

Figure D.4 shows a class diagram of the instance package. The instance package defines the classes for specifying CICs. Thus, the root class for the package is ComponentInstanceConfiguration that contains a set of ComponentInstances. As on the type level, we distinguish between AtomicComponentInstances and StructuredComponentInstances. However, we do not need to distinguish between static components and reconfigurable components on the instance level because this is already defined by the component type of the instance.

The metamodel for ComponentInstances is structured recursively. Any StructuredComponentInstance contains a ComponentInstanceConfiguration, again, that defines its embedded component instances and their connections.

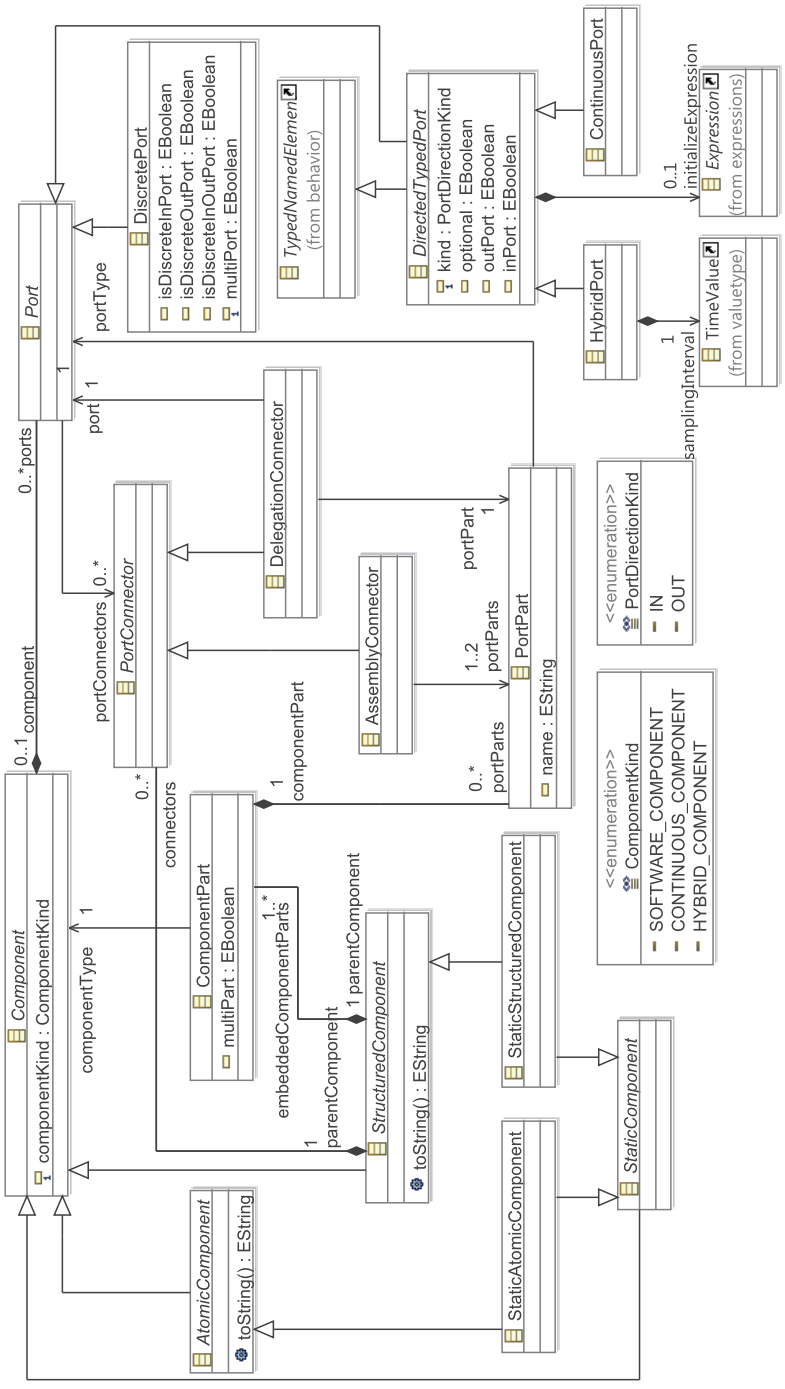


Figure D.3: Class Diagram of the Component Metamodel

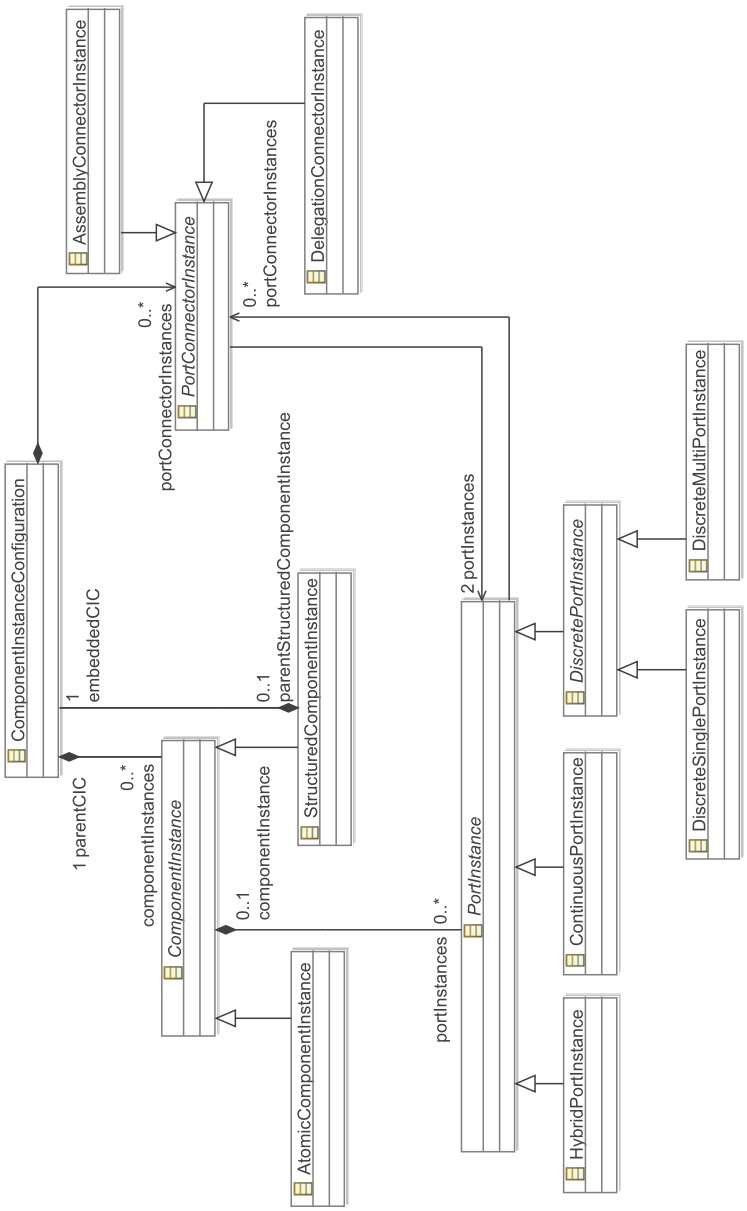


Figure D.4: Class Diagram of the Component Instance Metamodel

A ComponentInstance contains a set of PortInstances. As on the type level, we distinguish between HybridPortInstances, ContinuousPortInstances, and DiscretePortInstances. Furthermore, we distinguish DiscretePortInstances into DiscreteSinglePortInstances and DiscreteMultiPortInstances that inherit from DiscreteSingleInteractionEndpointInstance and DiscreteMultiInteractionEndpointInstance, respectively, of the component core.

In addition to the ComponentInstances, a ComponentInstanceConfiguration contains the set of PortConnectorInstances that connect the ComponentInstances. A PortConnectorInstance always connects two PortInstances, while a PortInstance may be attached to multiple PortConnectorInstances. A PortInstance of a StructuredComponentInstance typically has two PortConnectorInstances. One of these is a DelegationConnectorInstance that connects the PortInstance with a PortInstance of an embedded ComponentInstance. The other one is either a DelegationConnectorInstance to a PortInstance of the parentStructuredComponentInstance or it is an AssemblyConnectorInstance that connects it with another ComponentInstance in the same ComponentInstanceConfiguration.

D.1.4 Runtime Model

Figure D.5 shows a class diagram of the runtime package. The runtime package defines the classes for capturing a symbolic state of a MECHATRONICUML model. Thus, it forms the basis for a model@runtime of MECHATRONICUML. Therefore, the runtime package extends the instance package introduced in Section D.1.3 by additional runtime information such as variable values and the active states of the RTSCs.

The base classes of the runtime package are RuntimeBehavioralElement and RealtimeStatechartInstance. The RuntimeBehavioralElement is the super class for all metamodel elements that execute an RTSC at runtime. It refers to the RealtimeStatechartInstance that it executes. The RealtimeStatechartInstance has an active State that is located in the RTSC. In addition, it contains a set of VariableBindings that assign a concrete value to a Variable of the RTSC. Since RTSCs may contain hierarchical states, we enable that a RealtimeStatechartInstance contains subRealtimeStatechartInstances if a hierarchical state is currently active. The values of the clocks of the RealtimeStatechartInstance are not stored as part of the runtime metamodel but using the udbm library (cf. Section C.4). The clocks of the RealtimeStatechartInstance are associated to the clocks of the udbm library by a name mapping.

In accordance to the instance package, we distinguish between different types of RuntimeBehavioralElements. First, we distinguish between RuntimeComponentInstances and RuntimeDiscreteInteractionEndpointInstances. The latter are further distinguished into RuntimeDiscretePortInstances and RoleInstances. Both classes have subclasses for single and multi role/-port instances that inherit from DiscreteInteractionEndpointInstance and DiscretePortInstance, respectively.

In addition to the RealtimeStatechartInstance, a RuntimeBehavioralElement contains a RuntimeMessageBuffer with a bufferSize. The RuntimeMessageBuffer contains the RuntimeMessages that have been received from a communication partner and that need to be processed by the RealtimeStatechartInstance. A RuntimeMessage is typed by a MessageType and contains a set of RuntimeParameters that assign concrete values to the parameters of the MessageType.

Finally, the runtime package defines a RuntimeConnectorInstance with subclasses for RuntimeAssemblyConnectorInstances, which connect two RuntimeDiscretePortInstances, and RuntimeRoleConnectorInstance, which connect two RoleInstances. Since role connectors and as-

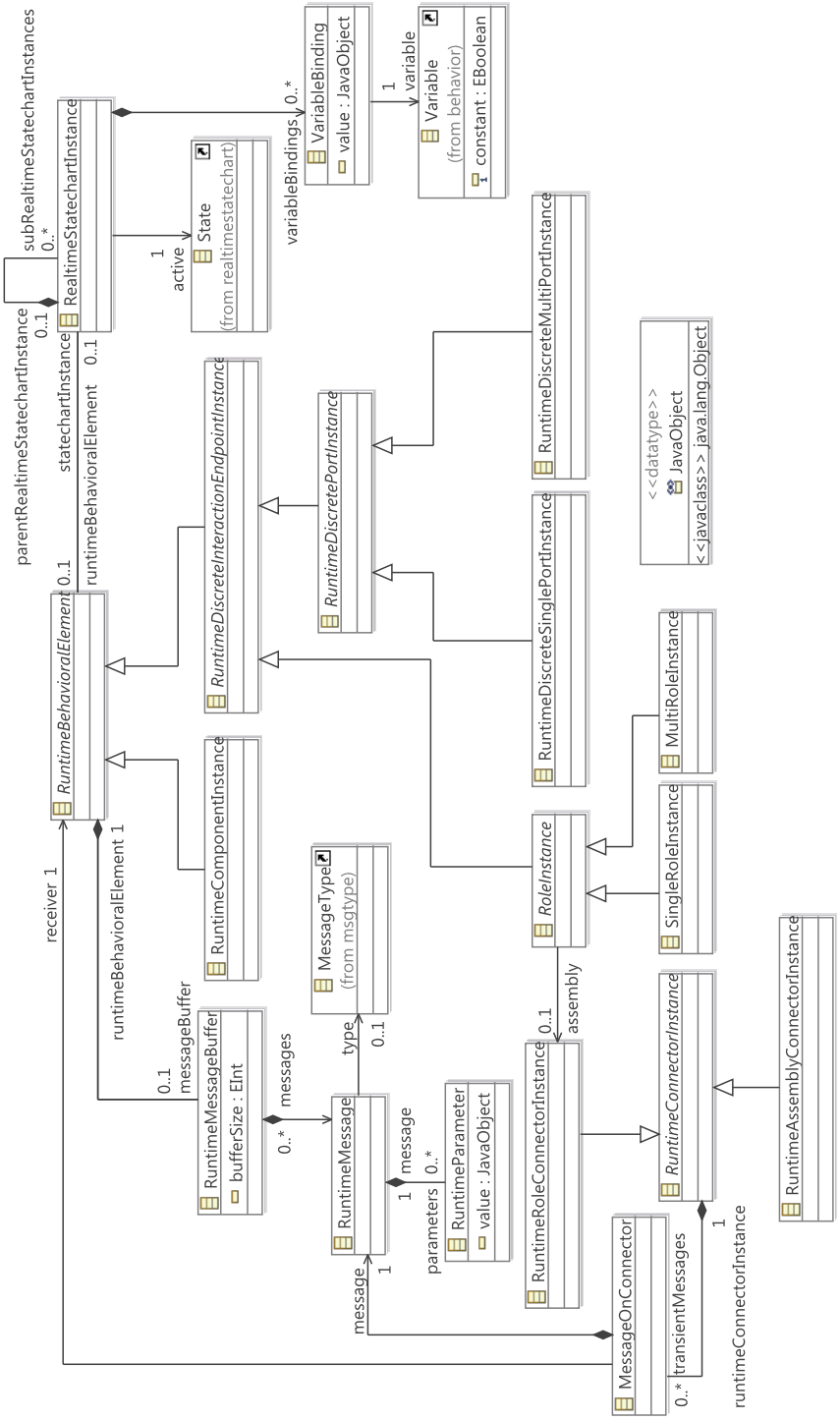


Figure D.5: Class Diagram of the Runtime Metamodel

sembly connectors have a delay, messages need time for being transmitted from the sender to the receiver. Therefore, each RuntimeConnectorInstance contains a set of transientMessages that are stored by the class MessageOnConnector. In addition to the RuntimeMessage, the MessageOnConnector defines the receiver of the RuntimeMessage, which is a RuntimeBehavioralElement in any case.

D.2 MechatronicUML Reconfiguration

This section introduces the metamodels for specifying reconfigurable components (Section D.2.1), component story patterns (Section D.2.2), component story diagrams (Section D.2.3), and component SDDs (Section D.2.4).

D.2.1 Reconfigurable Components

Figure D.6 shows a class diagram of the metamodel for reconfigurable components that is part of the reconfiguration package. The upper part of the figure shows the classes for the different component kinds. First, we have a new subclass of Component called ReconfigurableComponent. This is the super class for all kinds of reconfigurable components. Based on this, we define classes for ReconfigurableAtomicComponents and ReconfigurableStructuredComponents that inherit from the corresponding abstract super classes AtomicComponent and StructuredComponent of the component metamodel (cf. Figure D.3).

The FadingComponent on the right side of Figure D.6 is used for defining fading components. It is a special kind of AtomicComponent and defines a set of FadingFunctions. Each FadingFunction, in turn, defines a fading from one port (the fromPort) to another port (the toPort) of the FadingComponent.

In addition, the metamodel defines an abstract class ReconfigurationRule that serves as a super class for all kinds of reconfiguration rules that specify a modification of a ReconfigurableComponent. As part of this thesis, we only use CSDs (cf. Section D.2.3) as a subclass of ReconfigurationRule. Each reconfiguration rule has a Signature that defines it name as well as its input and output parameters.

Finally, we define special ports and connectors for ReconfigurableComponents. The class ReconfigurationPort serves as a super class for all kinds of ports that may only be used for ReconfigurableComponents. We use three subclasses. First, ReconfigurationMessagePort defines RM ports of ReconfigurableComponents and of the reconfiguration controller. Second, ReconfigurationExecutionPort defines RE ports in the same fashion. Third, InternalReconfigurationCommunicationPort is used for connecting manager and executor inside the reconfiguration controller. In addition, we define PortConnectors for connecting ReconfigurationPorts. A ReconfigurationPortAssemblyConnector connects two reconfiguration ports. We cannot use an AssemblyConnector of the component package because an AssemblyConnector connects two PortParts instead of two Ports. However, since the reconfiguration controller belongs to the ReconfigurableComponent instead of referring to another component, we do not use PortParts for referring to the ports of manager and executor. For the same reason, we need to define a ReconfigurationPortDelegationConnector for delegating the RM ports and RE ports of a ReconfigurableStructuredComponent to manager and executor, respectively.

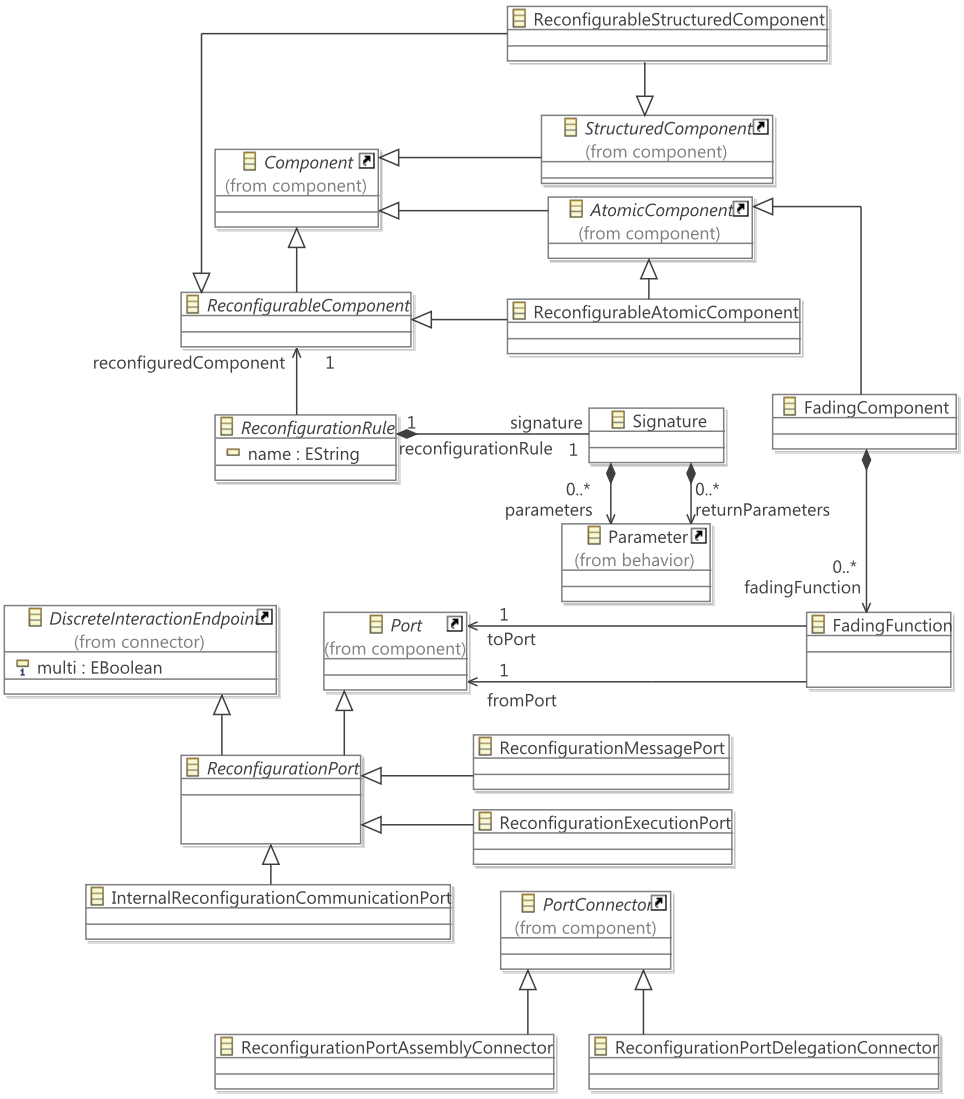


Figure D.6: Class Diagram of the Metamodel for Reconfigurable Components

Figure D.7 shows the metamodel for specifying the reconfiguration controller including the declarative, table-based specification of manager, executor, RM ports, and RE ports. Each ReconfigurableStructuredComponent contains at most one Controller. In future works, this reference may be used for integrating additional controllers into ReconfigurableStructuredComponents, for example, for performing monitoring. At present, our metamodel only supports ReconfigurationControllers and, in particular, the RuleBasedReconfigurationController that executes reconfigurations according the the 2-phase-commit protocol. The RuleBasedReconfigurationController contains the Manager and the Executor. Both are BehavioralElements, i.e., their behavior is defined by an RTSC.

The Manager contains a set of ManagerSpecificationEntry objects. Each ManagerSpecificationEntry defines one row of the table that defines the behavior of the Manager (cf. Section 4.3.2). The ManagerSpecificationEntry contains Boolean attributes treat, propagate, invokePlanner, and blockable for the four Boolean columns treat, progate to parent, invoke planner, and safety relevant. In addition, it refers to a ReconfigurationRule, a StructuralCondition, a MessageType, and a TimeValue that defines the timeForPlanning.

The interface of the ReconfigurationMessagePort is defined by ReconfigurationMessagePortInterfaceEntry. Each ReconfigurationMessagePortInterfaceEntry specifies one row of the table that defines the interface of the RM port (cf. Section 4.3.1). It inherits from ReconfigurationPortInterfaceEntry which defines the common features of the interface specification of RM ports and RE ports. These are a MessageType and a description. In addition, the ReconfigurationMessagePortInterfaceEntry specifies whether it defines an info message or a request and it contains a TimeValue for specifying the expectedResponseTime.

The Executor contains a set of ExecutorSpecificationEntry objects. Each ExecutorSpecificationEntry defines one row of the table that defines the behavior of the Executor (cf. Section 4.3.3). An ExecutorSpecificationEntry defines an id and refers to a ReconfigurationRule.

The interface of the ReconfigurationExecutionPort is defined by ReconfigurationExecutionPortInterfaceEntry that, in turn, inherits from ReconfigurationPortInterfaceEntry. Each ReconfigurationExecutionPortInterfaceEntry defines one row of the table that defines the interface of the RE port (cf. Section 4.3.4). In addition to the feature of its super class, ReconfigurationExecutionPortInterfaceEntry contains a TimeValue for defining the timeForPlanning and an ExecutionTimingSpecification. The ExecutionTimingSpecification defines the time that is necessary for the execution phase of the 2-phase-commit protocol. It has two subclasses ExecutionTimingSpecificationSinglePhase, which contains the timing specification for single-phase execution, and ExecutionTimingSpecificationThreePhase, which contains the timing specification for three-phase execution. Both classes contain TimeValues for defining the corresponding times for execution.

D.2.2 Component Story Patterns

The class diagram in Figure D.8 shows the metamodel for component story patterns. It transfers the metamodel by Tichy [Tic09] to the new MECHATRONICUML metamodel as introduced in Section D.1 and D.2.1 and reuses concepts from the new story diagram metamodel [HRvD+11] where possible.

A ComponentStoryPattern always contains exactly one ComponentVariable, which is the this-variable. The ComponentVariable contains a set of PortVariables, PartVariables, and ConnectorVariables. The common superclass for these three types of variables is ComponentStoryPattern-

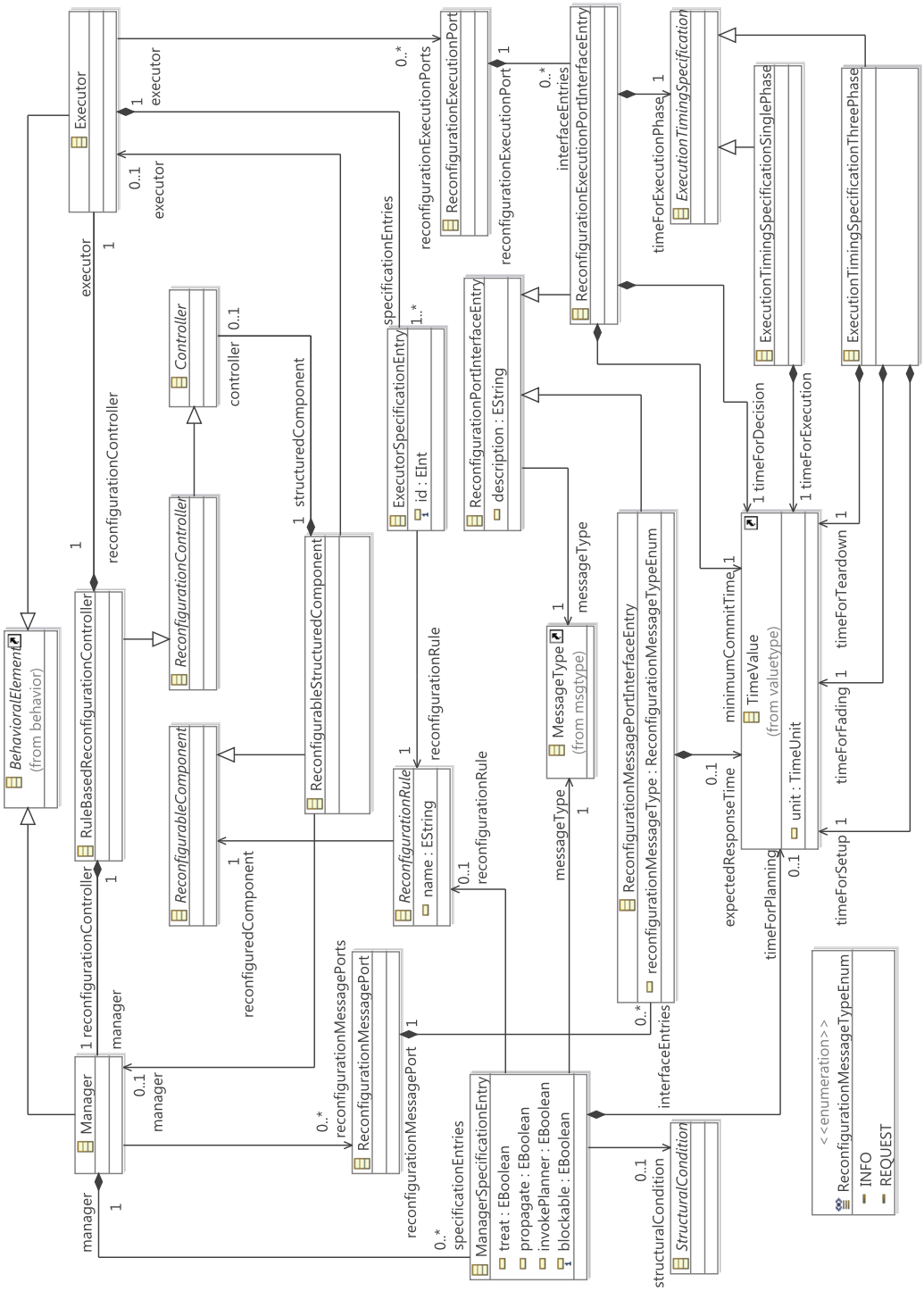


Figure D.7: Class Diagram of the Metamodel for Transactional Execution

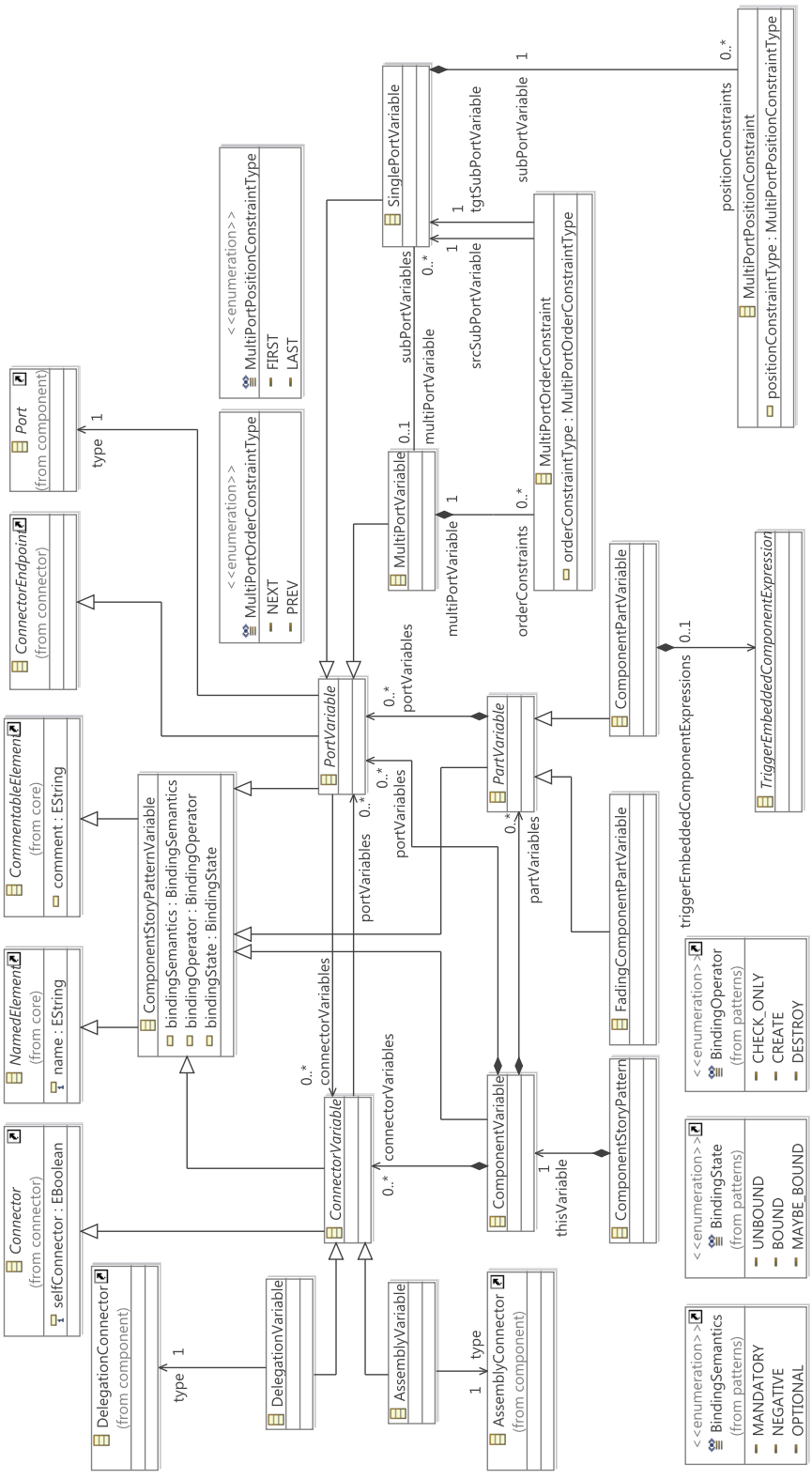


Figure D.8: Class Diagram of the Component Story Pattern Metamodel

Variable. It has three attributes for modifying the variable. First, the bindingSemantics defines whether the variable is mandatory for the matching or whether it is optional or negative. The bindingOperator defines whether the variable only matches or whether the matched object is created or destroyed. Finally, the bindingState defines whether it is a bound or unbound variable.

The PortVariables are typed by the ports of the component that is used as type of the this-variable. Similar to port instances, we distinguish between SinglePortVariables and MultiPortVariables, where each MultiPortVariable has a set of subPortVariables. Each SinglePortVariable contains a set of MultiPortPositionContraints that enable to define that a subPortInstance is the FIRST or LAST one in the MultiPortVariable. The MultiPortInstance contains a set of MultiPortOrderConstraints that specify that the tgtSubPortVariable is the successor (NEXT) or predecessor (PREV) of the srcSubPortVariable.

A PartVariable is typed over a component part of the component that is used as type of the this-variable. We distinguish two types of PartVariables: ComponentPartVariables and FadingComponentVariables. A ComponentPartVariable refers to a normal component part and contains a set of PortVariables that refer to the ports of the component part. In addition, it optionally specifies a TriggerEmbeddedComponentExpression that enables to trigger a reconfiguration of the component instance that is matched by the ComponentPartVariable. A FadingComponentVariable is typed by a component part that is, in turn, typed by a fading component. We use an additional class for FadingComponentVariables for integrating the particular syntactical constraints of fading components into the metamodel by means of OCL constraints.

ConnectorVariables connect the PortVariables. As on the type level, we distinguish between AssemblyVariables and DelegationVariables that both have a corresponding type.

D.2.3 Component Story Diagrams

The class diagram in Figure D.8 shows the metamodel for CSDs. It is based on the new metamodel for story diagrams [HRvD$^+$11] but introduces additional classes for integrating component story patterns.

The class ComponentStoryRule represents the CSD. It inherits from ReconfigurationRules and contains an Activity from the story diagram metamodel [HRvD$^+$11]. The Activity contains the ActivityNodes and ActivityEdges that constitute the control flow of the CSD. Each ActivityEdge connects two ActivityNodes and may specify a guardExpression for adding a Boolean condition to the ActivityEdge.

The CSD metamodel defines several ActivityNodes. These are StatementNode, JunctionNode, InitialNode, ActivityFinalNode, ComponentStoryNode, and ControllerExchangeNode. Only the latter two a specific for CSDs. The ComponentStoryNode contains a ComponentStoryPattern. The ControllerExchangeNode specifies the replacement of continuous component instances as described in Section 3.3.2.

Finally, the CSD metamodel defines a SendReconfigurationMessageExpression that refers to a MessageType and contains a set of ParameterBindings. It is contained in a ComponentPartVariable and enables to invoke a reconfiguration on an embedded component by sending a reconfiguration message to the component. The reconfigurationMessageType specified by the SendReconfigurationMessageExpression must be contained in the RE port interface specification of the component that is used as a type of the ComponentPartVariable.

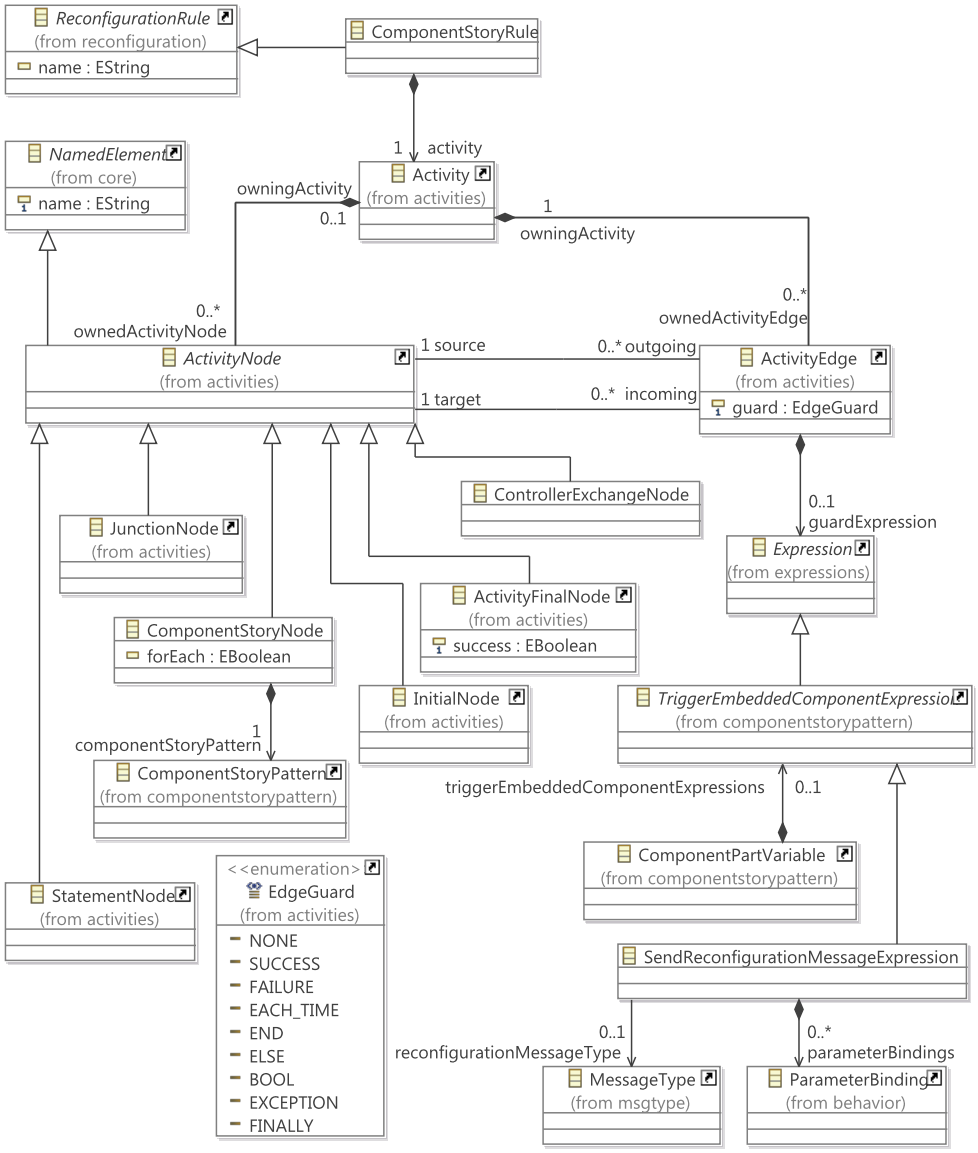


Figure D.9: Class Diagram of the Component Story Diagram Metamodel

D.2.4 Component Story Decision Diagrams

The class diagram in Figure D.10 shows the metamodel for component SDDs. The metamodel reuses concepts of the metamodel by Stallmann [Sta08] and integrates it with the metamodel for component story patterns (cf. Section D.2.2).

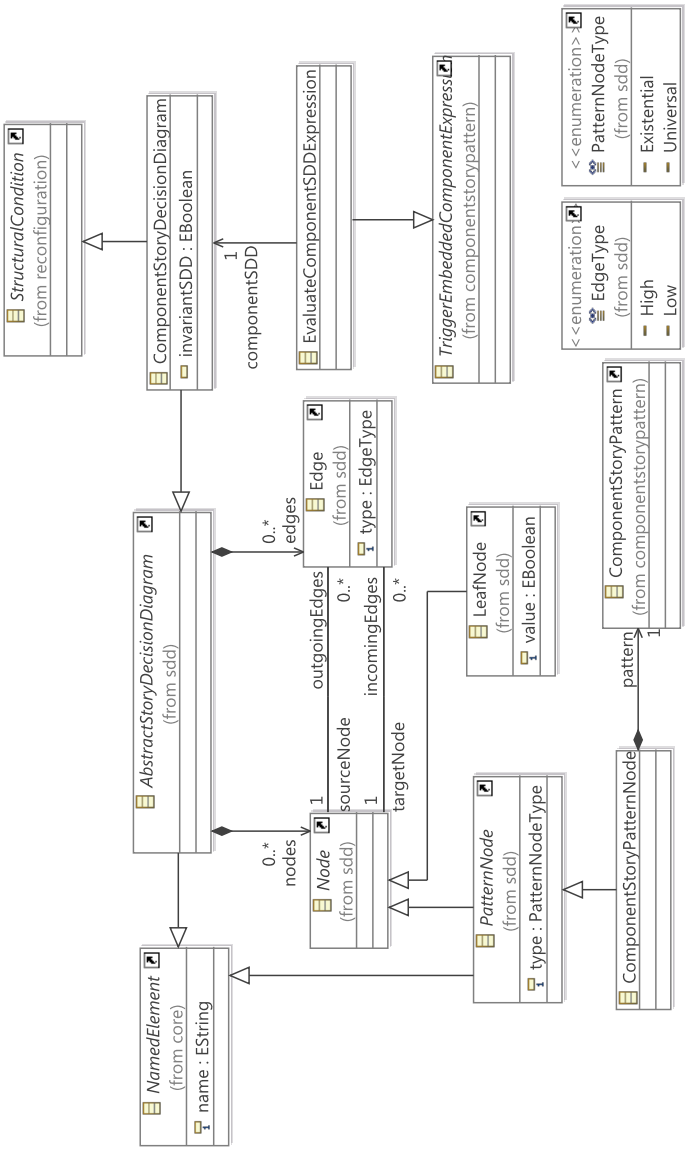


Figure D.10: Class Diagram of the Component Story Decision Diagram Metamodel

The ComponentStoryDecisionDiagram inherits from StructuralCondition such that it may be used in the ManagerSpecificationEntry (cf. Section D.2.1). In addition, it inherits from AbstractStoryDecisionDiagram that defines the Nodes and Edges that constitute the structure of an SDD. In addition to the LeafNodes of normal SDDs, the component SDD metamodel uses ComponentStoryPatternNodes that contain a ComponentStoryPattern.

Finally, the component SDD metamodel defines an EvaluateComponentSDDExpression that inherits from TriggerEmbeddedComponentExpression. It enables to refer to a componenSDD that is defined by a embedded component. It is contained in a ComponentPartVariable of the ComponentStoryPattern.

D.3 MATLAB/Simulink and Stateflow

This section introduces the metamodel for MATLAB/Simulink and Stateflow that we use as an intermediate model in our transformation of MECHATRONICUML models to MATLAB/Simulink. The metamodel reflects the model structure of Simulink and Stateflow, but is restricted to those language features of Simulink that we need for our transformation. In the following, we first present the metamodels for Simulink (Section D.3.1) and Stateflow (Section D.3.2). Thereafter, we introduce two utility metamodels for realizing message-based communication (Section D.3.3) and reconfiguration in Simulink (Section D.3.4). These utility metamodels do not reflect the Simulink model structure but ease the transformation and are translated to complex Simulink subsystems.

D.3.1 Simulink

The class diagram in Figure D.11 shows the core of the Simulink metamodel. The metamodel has been derived by reverse engineering the Simulink model structure, but includes several optimizations that ease the specification of a model transformation.

The root of a Simulink model is the SimulinkContainer that contains a set of SimulinkModels and SimulinkLibrary objects, both of which are SimulinkFiles. This enables to split a model over several files and libraries.

A Simulink model and everything that is contained therein is an Element. Besides the SimulinkContainer, the metamodel defines three types of Elements. These are Block, Line, and Bus. Each element has an id and a set of Parameters that define the properties of the Element. Parameters are specified as name/value pairs where the data type of the Parameter is given by an additional type attribute.

The basic building block of a Simulink model or library is the Block. SubSystems are special Blocks that contain further Blocks and thereby enable to structure the model hierarchically. Each SubSystem has a set of PortBlocks, which are either InPortBlocks where information enters the SubSystem or OutPortBlocks where information leaves the SubSystem. By adding an EnablePort to a SubSystem, the SubSystem becomes an enabled subsystem. In the same way, a SubSystem becomes a triggered subsystem by adding a TriggerPort. The TriggerPort also specifies whether it triggers the execution of the triggered subsystem when the input signal rises or falls or in either case by the TriggerEvent enum.

Two special kinds of Blocks are the LibraryReference and the ChartBlock. A LibraryReference enables to include a Block or even a complete Subsystem from a SimulinkLibrary. We utilize this feature for including an implementation of the link layer blocks (cf. Section 6.3.3.2) from a library.

The ChartBlock enables to include a Stateflow chart into a Simulink model. The ChartBlock provides the InPortBlocks and OutPortBlocks for connecting the ChartBlock with the remaining Simulink model. The ChartBlock refers to the Chart which contains the Stateflow chart. We

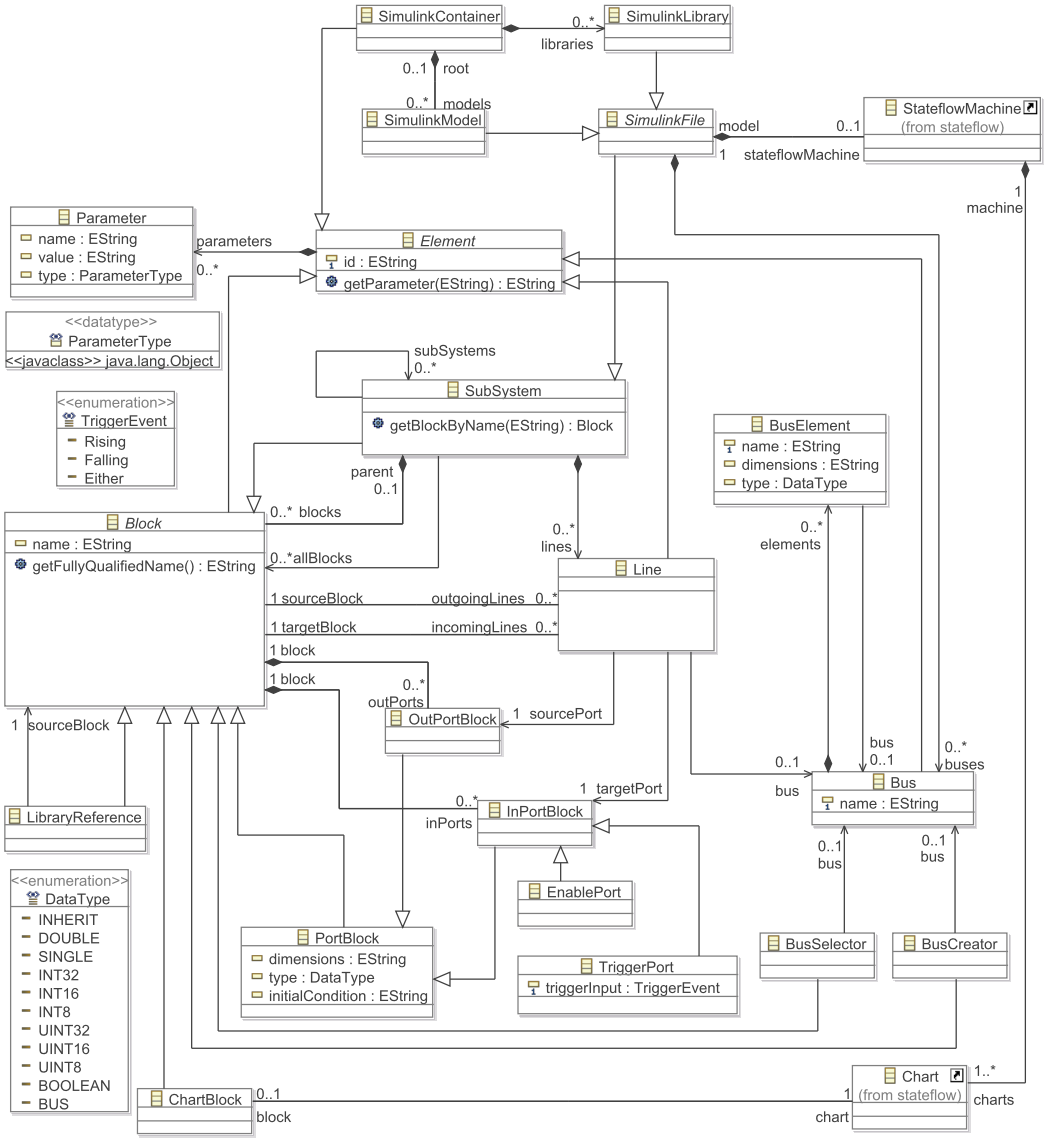


Figure D.11: Class Diagram of the Simulink Metamodel

introduce the Stateflow metamodel in detail in Section D.3.2. The Charts are contained in a StateflowMachine that is part of the SimulinkFile.

A SubSystem contains a set of Lines that connect the Blocks. Each Block may have several outgoingLines and incomingLines, but each Line connects exactly two Blocks. A Line refers to a Bus if it represents a bus signal.

A Bus has a name and contains a set of named BusElements, each having a DataType and a dimension. In addition, our metamodel contains two Blocks for handling Busses. These are BusSelector for retrieving a signal from the Bus and BusCreator for creating a Bus from a set of signals.

Figure D.12 shows additional types of Blocks. These are ZeroOrderHold, Constant, DigitalClock, UnitDelay, and EmbeddedMatlabFunction. All of these Blocks have the intended functionality as described in Chapter 6. The EmbeddedMatlabFunction specifies the code that defines its behavior as an attribute.

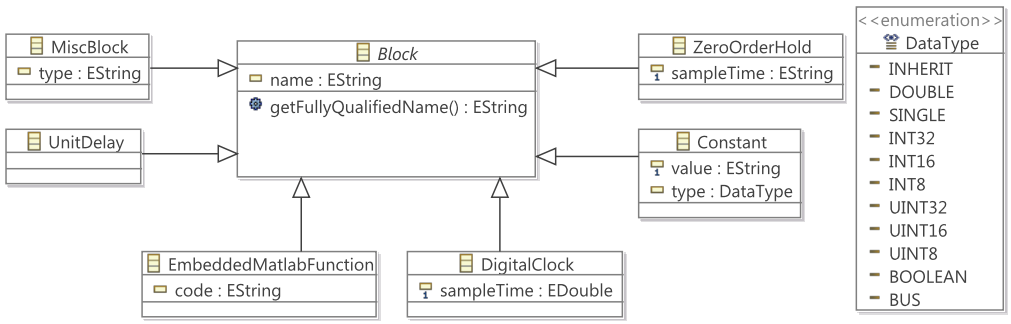


Figure D.12: Class Diagram of Additional Blocks of the Simulink Metamodel

A special case is given by the MiscBlock. The MiscBlock enables to represent any type of Block that may occur in a Simulink, but which is not explicitly represented in our metamodel.

D.3.2 Stateflow

The class diagram in Figure D.13 shows the Stateflow metamodel. The root of the metamodel is the StateflowElement. It is the super class for all elements of a Stateflow chart.

A Stateflow chart consists of Nodes and Transitions that connect the Nodes. Stateflow charts support three types of Nodes. These are States, Junctions, and History elements. States may again contain further Nodes, which enables to specify hierarchical state machines. The Chart itself is a special State. The subStateType defines whether a hierarchical State is an EXCLUSIVE or a PARALLEL State. In addition, States may be marked as initial and have a priority that defines the execution order for PARALLEL States.

A State contains a set of Data elements that enable to define local variables and constants for a State. A Data element has a name, a type, an initial value, and a size that defines whether the Data element is an array. A Chart additionally defines input and output Data elements. They have a 1:1 correspondence to the InPortBlocks and OutPortBlocks of the ChartBlock in Simulink. As a result, the signals that are received from and send to the Simulink model are used as regular variables in the Stateflow chart.

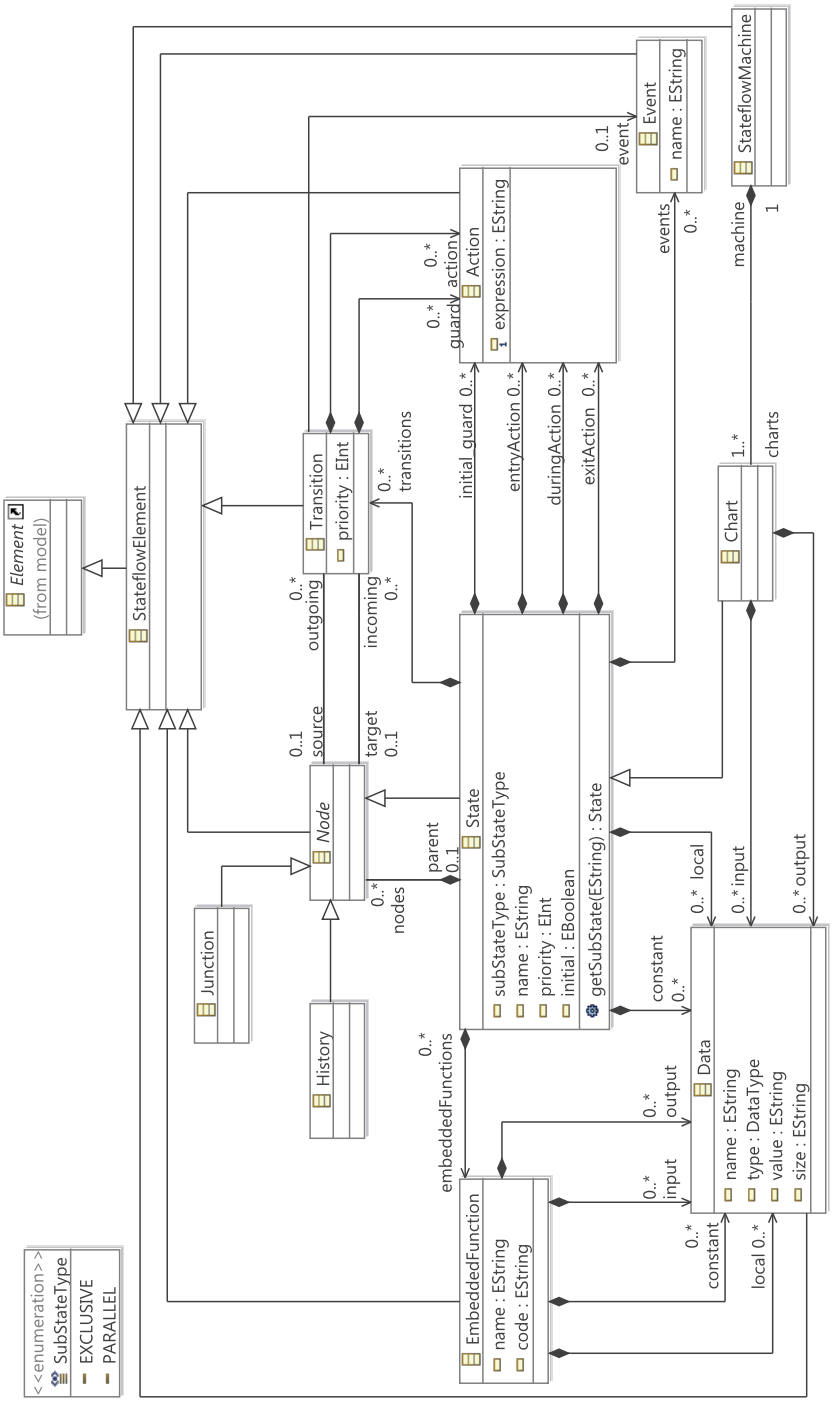


Figure D.13: Class Diagram of the Stateflow Metamodel

In addition, a State may define a set of EmbeddedFunctions. Each EmbeddedFunction has a name and a behavior specification that is given by the code. The input and output parameters of the EmbeddedFunction are specified by Data elements.

States and Transitions may contain a set of Actions. In a State, the Action is used for defining entryActions, exitActions, and duringActions. The initialGuard enables to select an initial State in case that the Stateflow chart contains more than one initial state, which is supported. For a Transition, an Action is used for defining the transition guard as well as the transition action.

Finally, a State defines a set of Events and a Transition may specify a received event. Sending an Event is an Action that is defined by a special expression String.

D.3.3 Message-Based Communication

The class diagram in Figure D.14 shows the metamodel for realizing message-based communication in Simulink. The metamodel only contains two classes that both inherit from Block. These are CommunicationSwitch and LinkLayer. Both are specific for our approach and have no direct correspondence to a block in Simulink, but are implemented by subsystems in Simulink.

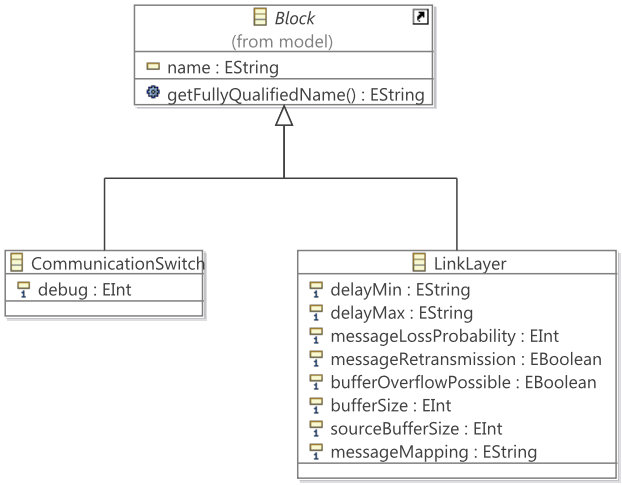


Figure D.14: Class Diagram of the Simulink Message Metamodel

The CommunicationSwitch represents a communication switch as defined in Section 6.3.3.3. The LinkLayer represents a link layer block as defined in Section 6.3.3.2. The LinkLayer specifies attributes for all QoS assumptions of MECHATRONICUML that we support in our transformation as described in Section 6.3.4.

The class diagram in Figure D.15 shows the metamodel for realizing message-based communication in Stateflow. In particular, message-based communication is realized by three BufferFunctions that are special EmbeddedFunctions. These functions realize the enqueue, dequeue, and checkQueue functions as defined in Section 6.4.2.

The subclasses of BufferFunction define two sets of buffer functions for realizing two variants of message buffers. The classes on the left side of Figure D.15 realize the Enqueue, Dequeue, and CheckQueue functions if there exists a separate message buffer for each message type. The classes on the right side of Figure D.15 realize the SharedEnqueue, SharedDequeue,

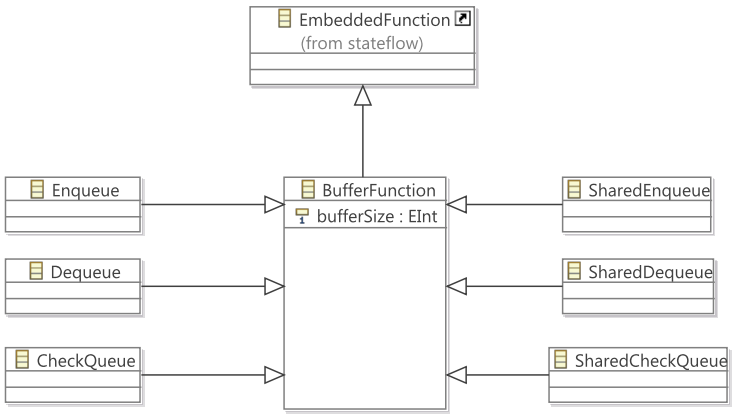


Figure D.15: Class Diagram of the Stateflow Buffer Metamodel

and SharedCheckQueue functions if all message types share the same buffer, i.e., there exists one message buffer that contains all message types.

D.3.4 Reconfiguration

The class diagram in Figure D.16 shows the metamodel that enables to emulate reconfiguration of continuous components in Simulink. The reconfiguration of discrete components is solely realized by features of Simulink already introduced in Section D.3.1.

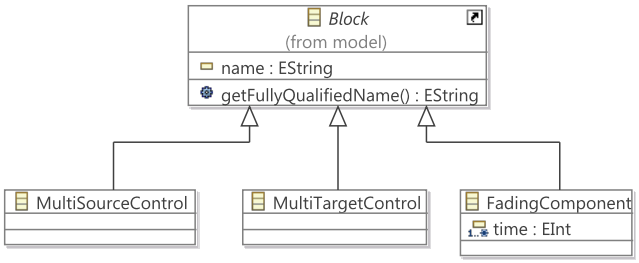


Figure D.16: Class Diagram of the Simulink Reconfiguration Metamodel

The metamodel defines three additional types of Blocks: MultiSourceControl, MultiTargetControl, and FadingComponent. MultiSourceControl and MultiTargetControl enable to reconfigure signals between continuous ports (cf. Section 6.3.2.1). The FadingComponent block represents a fading component. We provide a separate class for fading components because they have a fixed internal structure as defined in Section 6.3.1.3.

Own Publications

[ACE+08] Kahtan Alhawash, Toni Ceylan, Tobias Eckardt, Masud Fazal-Baqaie, Joel Greenyer, Christian Heinzemann, Stefan Henkler, Renate Ristov, Dietrich Travkin, and Coni Yalcin. The Fujaba Automotive Tool Suite. In Uwe Aßmann, Jendrik Johannes, and Albert Zündorf, editors, *Proceedings of the 6th International Fujaba Days 2008*, number TUD-FI08-09 in Technical Report, pages 36–39, Dresden, Germany, September 2008. Technische Universität Dresden.

[BBB+12] Steffen Becker, Christian Brenner, Christopher Brink, Stefan Dziwok, Christian Heinzemann, Renate Löffler, Uwe Pohlmann, Wilhelm Schäfer, Julian Suck, and Oliver Sudmann. The MechatronicUML design method – process, syntax, and semantics. Technical Report tr-ri-12-326, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, August 2012. Vers. 0.3.

[BBD+12] Steffen Becker, Christian Brenner, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann, Claudia Priesterjahn, Wilhelm Schäfer, Julian Suck, Oliver Sudmann, and Matthias Tichy. The MechatronicUML method – process, syntax, and semantics. Technical Report tr-ri-12-318, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, February 2012. Vers. 0.2.

[BDG+11] Steffen Becker, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann, Claudia Priesterjahn, Wilhelm Schäfer, Oliver Sudmann, and Matthias Tichy. MechatronicUML – syntax and semantics. Technical Report tr-ri-11-325, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, August 2011. Vers. 0.1.

[BDG+14a] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Wilhelm Schäfer, Matthias Meyer, and Uwe Pohlmann. The MechatronicUML method: Model-driven software engineering of self-adaptive mechatronic systems. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 614–615, New York, NY, USA, May 2014. ACM. ISBN:978-1-4503-2768-8. doi:10.1145/2591062.2591142.

[BDG+14b] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Sebastian Thiele, Wilhelm Schäfer, Matthias Meyer, Uwe Pohlmann, Claudia Priesterjahn, and Matthias Tichy. The MechatronicUML design method - process and language for platform-independent modeling. Technical Report

tr-ri-14-337, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, March 2014. Version 0.4.

[BHSH13] Christian Brenner, Christian Heinzemann, Wilhelm Schäfer, and Stefan Henkler. Automata-based refinement checking for real-time systems. In *Proceedings of Software Engineering 2013 – Fachtagung des GI-Fachbereichs Softwaretechnik*, volume P-213 of *Lecture Notes in Informatics (LNI)*, pages 99–112. Gesellschaft für Informatik e.V., March 2013. ISBN:978-3-88579-607-7.

[BvDHR11] Steffen Becker, Markus von Detten, Christian Heinzemann, and Jan Rieke. Structuring complex story diagrams by polymorphic calls. Technical Report tr-ri-11-323, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, March 2011.

[DBHT12] Stefan Dziwok, Kathrin Bröker, Christian Heinzemann, and Matthias Tichy. A catalog of real-time coordination patterns for advanced mechatronic systems. Technical Report tr-ri-12-319, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, February 2012.

[DGB+14] Stefan Dziwok, Christopher Gerking, Steffen Becker, Sebastian Thiele, Christian Heinzemann, and Uwe 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*, FSE 2014, pages 715–718, New York, NY, USA, November 2014. ACM. ISBN:978-1-4503-3056-5. doi:10.1145/2635868.2661665.

[DGH15] Stefan Dziwok, Christopher Gerking, and Christian Heinzemann. Domain-specific model checking of MechatronicUML models using Uppaal. Technical Report tr-ri-15-346, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2015.

[DHT12] Stefan Dziwok, Christian Heinzemann, and Matthias Tichy. Real-time coordination patterns for advanced mechatronic systems. In Marjan Sirjani, editor, *Coordination Models and Languages*, volume 7274 of *Lecture Notes in Computer Science*, pages 166–180. Springer Berlin Heidelberg, June 2012. ISBN:978-3-642-30828-4. doi:10.1007/978-3-642-30829-1_12.

[EH11] Tobias Eckardt and Christian Heinzemann. Providing timing computations for FUJABA. In Ulrich Norbisrath and Ruben Jubeh, editors, *Proceedings of the 8th International Fujaba Days*, pages 38–42. Kasseler Informatikschriften (KIS) 2012, 1, May 2011. URL: `https://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2012053041248`.

[EHH+13] Tobias Eckardt, Christian Heinzemann, Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, and Wilhelm Schäfer. Modeling and verifying dynamic communication structures based on graph transformations. *Computer Science - Research and Development*, 28(1):3–22, February 2013. ISSN:1865-2034. Published online July 2011. doi:10.1007/s00450-011-0184-y.

[FGH+14] Kathrin Flaßkamp, Stefan Groesbrink, Philip Hartmann, Christian Heinzemann, Bernd Kleinjohann, Lisa Kleinjohann, Martin Krüger, Sina Ober-Blöbaum, Claudia Priesterjahn, Christoph Rasche, Wilhelm Schäfer, Dominik Steenken, Ansgar Trachtler, Heike Wehrheim, and Steffen Ziegert. Development of the RailCab vehicle. In Jürgen Gausemeier, Wilhelm Schäfer, Franz-Josef Rammig, and Walter Sextro, editors, *Dependability of Self-Optimizing Mechatronic Systems*, pages 184–190. Springer, Heidelberg, Germany, January 2014.

[FHK+13] Kathrin Flaßkamp, Christian Heinzemann, Martin Krüger, Dominik Steenken, Sina Ober-Blöbaum, Wilhelm Schäfer, Ansgar Trächtler, and Heike Wehrheim. Sichere Konvoibildung mit Hilfe optimaler Bremsprofile. In Jürgen Gausemeier, Franz-Josef Rammig, Wilhelm Schäfer, and Ansgar Trächtler, editors, *9. Paderborner Workshop Entwurf mechatronischer Systeme*, volume 310 of *HNI-Verlagsschriftenreihe*, pages 177–190. Heinz Nixdorf Institut, Universität Paderborn, April 2013. ISBN:978-3-942647-29-8.

[FHK+14] Kathrin Flaßkamp, Christian Heinzemann, Martin Krüger, Sina Ober-Blöbaum, Wilhelm Schäfer, Dominik Steenken, Ansgar Trachtler, and Heike Wehrheim. Verification for interacting mechatronic systems with motion profiles. In Jürgen Gausemeier, Franz-Josef Rammig, Wilhelm Schäfer, and Walter Sextro, editors, *Dependability of Self-optimizing Mechatronic Systems*, chapter 3.2.10, pages 119–128. Springer-Verlag, Heidelberg, Germany, January 2014.

[GH14] Christopher Gerking and Christian Heinzemann. Solving the movie database case with QVTo. In Louis M. Rose, Christian Krause, and Tassilo Horn, editors, *Proceedings of the 7th Transformation Tool Contest part of the Software Technologies: Applications and Foundations (STAF 2014) federation of conferences*, pages 98–102. CEUR-WS.org Vol-1305, July 2014.

[HB13] Christian Heinzemann and Steffen Becker. Executing reconfigurations in hierarchical component architectures. In *Proceedings of the 16th international ACM Sigsoft symposium on Component based software engineering*, CBSE '13, pages 3–12, New York, NY, USA, June 2013. ACM. ISBN:978-1-4503-2122-8. doi:10.1145/2465449.2465452.

[HB14] Christian Heinzemann and Steffen Becker. Comparison of the mechatronicuml component models. Technical Report tr-ri-14-341, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, June 2014.

[HBD13] Christian Heinzemann, Christian Brenner, and Stefan Dziwok. Evaluation-models, 2013. URL: `https://trac.cs.upb.de/mechatronicuml/wiki/JournalCSRD2013`.

[HBDS15] Christian Heinzemann, Christian Brenner, Stefan Dziwok, and Wilhelm Schäfer. Automata-based refinement checking for real-time systems. *Computer Science - Research and Development*, 30(3-4):255–283, 2015.

ISSN:1865-2034. Published Online June 2014. doi:10.1007/s00450-014-0257-9.

[Hei09] Christian Heinzemann. Verifikation von Protokollverfeinerungen. Master's thesis, University of Paderborn, November 2009.

[Hei10] Christian Heinzemann. Verifikation von Protokollverfeinerungen. In *Informatiktage 2010 - Fachwissenschaftlicher Informatik-Kongress 19. und 20. März 2010, B-IT Bonn-Aachen International Center for Information Technology in Bonn*, volume S-9 of *Lecture Notes in Informatics (LNI)*, pages 57–60. Gesellschaft für Informatik, March 2010. ISBN:3-88579-443-1.

[Hei12] Christian Heinzemann. Anforderungen an eine globale Kommunikationsarchitektur für das RailCab System. RailCab Projektbericht tr-ri-12-321, Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn, April 2012.

[Hei13] Christian Heinzemann. Website accompanying executing reconfiguration in hierarchical component architectures, 2013. URL: `https://trac.cs.upb.de/mechatronicuml/wiki/PaperCBSE2013`.

[Hei14] Christian Heinzemann. Component story decision diagrams. Technical Report tr-ri-14-335, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, January 2014.

[HGH+09] Stefan Henkler, Joel Greenyer, Martin Hirsch, Wilhelm Schäfer, Kahtan Alhawash, Tobias Eckardt, Christian Heinzemann, Renate Löffler, Andreas Seibel, and Holger Giese. Synthesis of timed behavior from scenarios in the Fujaba Real-Time Tool Suite. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 615–618, Washington, DC, USA, May 2009. IEEE Computer Society. ISBN:978-1-4244-3453-4. doi:10.1109/ICSE.2009.5070569.

[HH11a] Christian Heinzemann and Stefan Henkler. Reusing dynamic communication protocols in self-adaptive embedded component architectures. In *Proceedings of the 14th International Symposium on Component Based Software Engineering*, CBSE '11, pages 109–118. ACM, June 2011. ISBN:978-1-4503-0723-9. doi:10.1145/2000229.2000246.

[HH11b] Christian Heinzemann and Stefan Henkler. Timed story driven modeling. Technical Report tr-ri-11-326, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2011.

[HHH10] Christian Heinzemann, Stefan Henkler, and Martin Hirsch. Refinement checking of self-adaptive embedded component architectures. Technical Report tr-ri-10-313, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, March 2010.

[HHZ09] Christian Heinzemann, Stefan Henkler, and Albert Zündorf. Specification and refinement checking of dynamic systems. In Pieter Van Gorp, editor, *Proceedings of the 7th International Fujaba Days*, pages 6–10, Eindhoven University of Technology, The Netherlands, November 2009.

[HP14] Christian Heinzemann and Claudia Priesterjahn. Convoy mode. In Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer, editors, *Design Methodology for Intelligent Technical Systems Systems*, chapter 2.1.7, pages 49–50. Springer, Heidelberg, Germany, January 2014.

[HPB12] Christian Heinzemann, Claudia Priesterjahn, and Steffen Becker. Towards modeling reconfiguration in hierarchical component architectures. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE'12, pages 23–28, New York, NY, USA, June 2012. ACM. ISBN:978-1-4503-1345-2. doi:10.1145/2304736.2304742.

[HPR+12] Christian Heinzemann, Uwe Pohlmann, Jan Rieke, Wilhelm Schäfer, Oliver Sudmann, and Matthias Tichy. Generating Simulink and Stateflow models from software specifications. In Dorian Marjanovic, Mario Storga, Neven Pavkovic, and Nenad Bojcetic, editors, *Proceedings of the 12th International Design Conference*, DESIGN 2012, pages 475–484. The Design Society, May 2012. ISBN:978-953-7738-17-4.

[HPSZ14] Christian Heinzemann, Claudia Priesterjahn, Dominik Steenken, and Steffen Ziegert. Software design. In Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer, editors, *Design Methodology for Intelligent Technical Systems*, chapter 5.2, pages 197–222. Springer, Heidelberg, Germany, January 2014.

[HRB+13] Christian Heinzemann, Jan Rieke, Jana Bröggelwirth, Andrey Pines, and Andreas Volk. Translating MechatronicUML models to MATLAB/Simulink and Stateflow. Technical Report tr-ri-13-330, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, May 2013. Version 0.3.

[HRB+14] Christian Heinzemann, Jan Rieke, Jana Bröggelwirth, Andrey Pines, and Andreas Volk. Translating MechatronicUML models to MATLAB/Simulink and Stateflow. Technical Report tr-ri-14-338, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, May 2014. Version 0.4.

[HRS13] Christian Heinzemann, Jan Rieke, and Wilhelm Schäfer. Simulating self-adaptive component-based systems using MATLAB/Simulink. In *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '13, pages 71–80. IEEE Computer Society, September 2013. doi:10.1109/SASO.2013.17.

[HRvD+11] Christian Heinzemann, Jan Rieke, Markus von Detten, Dietrich Travkin, and Marius Lauder. A new meta-model for story diagrams. In Ulrich Norbisrath and Ruben Jubeh, editors, *Proceedings of the 8th International Fujaba Days (Fujaba Days 2011), Tartu, Estonia, May 11-13, 2011*, number 2012, 1 in

Kasseler Informatikschriften (KIS), pages 1–5. University of Tartu, University of Kassel, May 2011.

[HS15] Christian Heinzemann and David Schubert. Downloading the RailCab convoy modeling example, 2015. URL: `https://trac.cs.upb.de/mechatronicuml/wiki/PaperSEAMS2015`.

[HSD+15] Christian Heinzemann, David Schubert, Stefan Dziwok, Uwe Pohlmann, Claudia Priesterjahn, Christian Brenner, and Wilhelm Schäfer. Railcab convoys: An exemplar for using self-adaptation in cyber-physical systems. Technical Report tr-ri-15-344, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, January 2015. doi:10.13140/2.1.4726.8163.

[HSE10] Christian Heinzemann, Julian Suck, and Tobias Eckardt. Reachability analysis on timed graph transformation systems. *Electronic Communications of the EASST*, 32, 2010. ISSN:1863-2122.

[HSJZ10] Christian Heinzemann, Julian Suck, Ruben Jubeh, and Albert Zündorf. Topology analysis of car platoons merge with FujabaRT & TimedStoryCharts - a case study. In Pieter Van Gorp, Steffen Mazanek, and Arend Rensink, editors, *Transformation Tool Contest*, Malaga, 2010.

[HSST13] Christian Heinzemann, Oliver Sudmann, Wilhelm Schäfer, and Matthias Tichy. A discipline-spanning development process for self-adaptive mechatronic systems. In *Proceedings of the 2013 International Conference on Software and System Process*, ICSSP 2013, pages 36–45, New York, NY, USA, May 2013. ACM. ISBN:978-1-4503-2062-7. doi:10.1145/2486046.2486055.

[PHST12] Claudia Priesterjahn, Christian Heinzemann, Wilhelm Schäfer, and Matthias Tichy. Runtime safety analysis for safe reconfiguration. In *Proceedings of the 3. Workshop „Self-X and Autonomous Control in Engineering Applications", 10. IEEE International Conference on Industrial Informatics*, INDIN'12, pages 1092 – 1097. IEEE Computer Society, July 2012. ISBN:978-1-4673-0312-5. doi:10.1109/INDIN.2012.6300900.

[SHS11] Julian Suck, Christian Heinzemann, and Wilhelm Schäfer. Formalizing model checking on timed graph transformation systems. Technical Report tr-ri-11-316, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, September 2011.

[vDHP+12a] Markus von Detten, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Julian Suck, Dietrich Travkin, and Stephan Hildebrandt. Story diagrams – syntax and semantics. Technical Report tr-ri-12-320, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, April 2012. Ver. 0.1.

[vDHP+12b] Markus von Detten, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Dietrich Travkin, and Stephan Hildebrandt. Story diagrams – syntax and semantics. Technical Report tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2012. Ver. 0.2.

[ZH13a] Steffen Ziegert and Christian Heinzemann. Durative graph transformation rules. Technical Report tr-ri-13-329, Heinz Nixdorf Institute, University of Paderborn, March 2013.

[ZH13b] Steffen Ziegert and Christian Heinzemann. Durative graph transformation rules for modelling real-time reconfiguration. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Proceedings of the 10th International Colloquium on Theoretical Aspects of Computing*, volume 8049 of *Lecture Notes in Computer Science*, pages 427–444. Springer Berlin Heidelberg, September 2013. ISBN:978-3-642-39717-2. doi:10.1007/978-3-642-39718-9_25.

Supervised Thesis

[AAB+11] Amir Shayan Ahmadian, Caner Aydogan, Denis Braun, Luis G. Bustamante, Christopher Gerking, Süleyman Issiz, Lukas Kopecki, and Paul Prescher. Developer documentation of the Project Group SafeBots I. Project group, University of Paderborn, September 2011.

[Bre10] Christian Brenner. Analyse von mechatronischen Systemen mittels Testautomaten. Master's thesis, University of Paderborn, August 2010.

[Brö11] Kathrin Bröker. Modellierung und Verifikation von Kommunikationsprotokollen für den Betrieb des RailCab Systems. Master's thesis, University of Paderborn, November 2011.

[Dre11] Christian Dreising. Reconfiguration of MechatronicUML component architectures. Bachelor's thesis, University of Paderborn, November 2011.

[Pin12] Andrey Pines. Transformation von rekonfigurierbaren MechatronicUML-Modellen nach MATLAB/Simulink. Bachelor's thesis, University of Paderborn, June 2012.

[Sch12] David Schubert. Integration von Modellierungssprachen für Rekonfiguration in MechatronicUML. Bachelor's thesis, University of Paderborn, June 2012.

[Sch15] David Schubert. Identification of safe states for reconfiguration in MechatronicUML. Master's thesis, University of Paderborn, July 2015.

[Suc11] Julian Suck. Model Checking zeitbehafteter Graphtransformationssysteme. Master's thesis, University of Paderborn, May 2011.

[Vol13] Andreas Volk. Hybride, rekonfigurierbare, hierarchische Komponentenstrukturen in MATLAB/Simulink. Master's thesis, University of Paderborn, December 2013.

Literature

[ABBL03] Luca Aceto, Patricia Bouyer, Augusto Burgueño, and Kim G. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 300(1-3):411–475, May 2003. ISSN:0304-3975. doi:10.1016/s0304-3975(02)00334-1.

[ABRW09] Anne Angermann, Michael Beuschel, Martin Rau, and Ulrich Wohlfarth. *MATLAB – Simulink – Stateflow, Grundlagen, Toolboxen, Beispiele*. Oldenbourg Verlag, München, 6 edition, 2009. ISBN:978-3-486-59546-8.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993. ISSN:0890-5401. doi:10.1006/inco.1993.1024.

[ÅCF+07] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655 – 667, May 2007. ISSN:0164-1212. doi:10.1016/j.jss.2006.08.016.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994. ISSN:0304-3975. doi:10.1016/0304-3975(94)90010-8.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382:21–37, 1998.

[ADI06] Rajeev Alur, Thao Dang, and Franjo Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 5:152–199, February 2006. ISSN:1539-9087. doi:10.1145/1132357.1132363.

[ADM01] Eugene Asarin, Thao Dang, and Oded Maler. d/dt: A tool for reachability analysis of continuous and hybrid systems. In *Proceedings of the 5th IFAC Symposium Nonlinear Control Systems*, NOLCOS, pages 3–34, July 2001. ISBN:978-0-08-043560-2.

[ADM02] Eugene Asarin, Thao Dang, and Oded Maler. The d/dt tool for verification of hybrid systems. In Ed Brinksma and Kim G. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 365–370. Springer Berlin Heidelberg, 2002. ISBN:978-3-540-43997-4. doi:10.1007/3-540-45657-0_30.

[ÅEM98] Karl Johan Åström, Hilding Elmqvist, and Sven Erik Mattson. Evolution of continuous-time modeling and simulation. In *Proceedings of the 12th European Simulation Multiconference on Simulation - Past, Present and Future*, pages 9–18. SCS Europe, 1998. ISBN:1-56555-148-6.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In Jacobus W. de Bakker, Cornelis Huizing, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer Berlin Heidelberg, June 1992. ISBN:978-3-540-55564-3. doi:10.1007/bfb0031988.

[AHJ+09] Matthieu Anne, Ruan He, Tahar Jarboui, Marc Lacoste, Olivier Lobry, Guirec Lorant, Maxime Louvel, Juan Navas, Vincent Olive, Juraj Polakovic, Marc Poulhiès, Jacques Pulou, Stéphane Seyvoz, Julien Tous, and Thomas Watteyne. Think: View-based support of non-functional properties in embedded systems. In *International Conference on Embedded Software and Systems*, ICESS '09, pages 147 –156. IEEE Computer Society, May 2009. ISBN:978-1-4244-4359-8. doi:10.1109/icess.2009.30.

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin Heidelberg, 1998. ISBN:978-3-540-64896-3. doi:10.1007/bfb0055622.

[Alu99] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer Berlin Heidelberg, July 1999. ISBN:978-3-540-66202-0. doi:10.1007/3-540-48683-6_3.

[Alu11] Rajeev Alur. Formal verification of hybrid systems. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 273–278, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0714-7. doi:10.1145/2038642.2038685.

[AM02] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, ASE 2002, pages 271–274, Los Alamitos, CA, USA, 2002. IEEE Computer Society. ISBN:0-7695-1736-6. doi:10.1109/ase.2002.1115028.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, May 2004. ISSN:1469-8072. doi:10.1017/s0960129504004153.

[ASTPH10] Rasmus Adler, Ina Schaefer, Mario Trapp, and Arnd Poetzsch-Heffter. Component-based modeling and verification of dynamic adaptation in

safety-critical embedded systems. *ACM Transactions on Embedded Computing Systems*, 10(2):20:1–20:39, December 2010. ISSN:1539-9087. doi:10.1145/1880050.1880056.

[AUT11] AUTOSAR. *AUTOSAR 3.2 - Technical Overview*, April 2011. Document Identification No. 067, Version 2.2.2. URL: `http://www.autosar.org/fileadmin/files/releases/3-2/main/auxiliary/AUTOSAR_TechnicalOverview.pdf` [cited March 14, 2015].

[AUT14a] AUTOSAR. *AUTOSAR 4.1 - Guide to Modemanagement*, March 2014. Document Identification No. 440, Version 2.2.0. URL: `http://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/system-services/auxiliary/AUTOSAR_EXP_ModemanagementGuide.pdf` [cited March 14, 2015].

[AUT14b] AUTOSAR. *AUTOSAR 4.1 - Specification of Timing Extensions*, March 2014. Document Identification No. 411, Version 2.1.1. URL: `http://www.autosar.org/fileadmin/files/releases/4-1/methodology-templates/templates/standard/AUTOSAR_TPS_TimingExtensions.pdf` [cited March 14, 2015].

[AUT14c] AUTOSAR. *AUTOSAR 4.1 - Virtual Functional Bus*, March 2014. Document Identification No. 056, Version 3.2.0. URL: `http://www.autosar.org/fileadmin/files/releases/4-1/main/auxiliary/AUTOSAR_EXP_VFB.pdf` [cited March 14, 2015].

[BB08] Gerd Baumann and Michael Brost. Testverfahren für Elektronik und Embedded Software in der Automobilentwicklung. In Holger Giese, Michaela Huhn, Ulrich A. Nickel, and Bernhard Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV, Schloss Dagstuhl, Germany, 7.-9. April 2008, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, number 2008-2 in Informatik-Bericht, pages 13–19. TU Braunschweig, Institut für Software Systems Engineering, April 2008.

[BBCP13] Jiří Barnat, Nikola Beneš, Ivana Cerná, and Zuzana Petruchová. DCCL: verification of component systems with ensembles. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, CBSE '13, pages 43–52, New York, NY, USA, June 2013. ACM. ISBN:978-1-4503-2122-8. doi:10.1145/2465449.2465453.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42(10):22 –27, October 2009. ISSN:0018-9162. doi:10.1109/mc.2009.326.

[BBG+06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *Proceedings of the 28th International Conference on*

Software Engineering, ICSE '06, pages 72–81, New York, NY, USA, May 2006. ACM. ISBN:1-59593-375-1. doi:10.1145/1134285.1134297.

[BC12] Olivier Bouissou and Alexandre Chapoutot. An operational semantics for Simulink's simulation engine. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 129–138, New York, NY, USA, 2012. ACM. ISBN:978-1-4503-1212-7. doi:10.1145/2248418.2248437.

[BCC98] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In Willem-Paul Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer Berlin Heidelberg, 1998. ISBN:978-3-540-65493-3. doi:10.1007/3-540-49213-5_4.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. In Marvin V. Zelkovwitz, editor, *Advances in Computers, Volume 58*, pages 117–148. Elsevier, 1st edition, 2003. ISBN:0-12-012158-1.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, WOSS '04, pages 28–33, New York, NY, USA, 2004. ACM. ISBN:1-58113-989-6. doi:10.1145/1075405.1075411.

[BCK98] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 419–428, New York, NY, USA, May 1998. ACM. ISBN:0-89791-962-9. doi:10.1145/276698.276854.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006. ISSN:1097-024X. doi:10.1002/spe.767.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer Berlin Heidelberg, September 2004. ISBN:978-3-540-23068-7. doi:10.1007/978-3-540-30080-9_7.

[BDL06a] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on* UPPAAL *4.0*. Department of Computer Science, Aalborg University, Denmark, November 2006.

[BDL+06b] Gerd Behrmann, Alexandre David, Kim G. Larsen, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, QEST 2006, pages 125–126, Los Alamitos, CA, USA, September 2006. IEEE Computer Society. ISBN:0-7695-2665-9. doi:10.1109/QEST.2006.59.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In Anders Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 298–302. Springer Berlin Heidelberg, 1998. ISBN:978-3-540-65003-4. doi:10.1007/bfb0055357.

[BDNG06] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36 –43, October 2006. ISSN:0018-9162. doi:10.1109/mc.2006.362.

[Bey01] Dirk Beyer. Efficient reachability analysis and refinement checking of timed automata using BDDs. In Tiziana Margaria and Thomas F. Melham, editors, *Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 86–91. Springer Berlin Heidelberg, 2001. ISBN:978-3-540-42541-0. doi:10.1007/3-540-44798-9_6.

[BFHP09] Etienne Borde, Peter H. Feiler, Grégory Haïk, and Laurent Pautet. Model driven code generation for critical and adaptative embedded systems. *SIGBED Review*, 6:10:1–10:5, October 2009. ISSN:1551-3688. doi:10.1145/1851340.1851352.

[BGH+07] Sven Burmester, Holger Giese, Stefan Henkler, Martin Hirsch, Matthias Tichy, Alfonso Gambuzza, Eckehard Münch, and Henner Vöcking. 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*, ICSE 2007, pages 801–804. IEEE Computer Society, May 2007. ISBN:0-7695-2828-7. doi:10.1109/ICSE.2007.88.

[BGH+13] Tomáš Bureš, Ilias Gerostathopoulos, Petr Hnětynka, Jaroslav Keznikl, Michal Kit, and František Plášil. DEECo: an ensemble-based component system. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, CBSE '13, pages 81–90, New York, NY, USA, June 2013. ACM. ISBN:978-1-4503-2122-8. doi:10.1145/2465449.2465462.

[BGK+96] Johan Bengtsson, David W. O. Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102

of *Lecture Notes in Computer Science*, pages 244–256. Springer Berlin Heidelberg, July 1996. ISBN:978-3-540-61474-6. doi:10.1007/3-540-61474-5_73.

[BGN+10] Basil Becker, Holger Giese, Stefan Neumann, Martin Schenck, and Arian Treffer. Model-based extension of AUTOSAR for architectural online reconfiguration. In Sudipto Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 83–97. Springer Berlin / Heidelberg, 2010. ISBN:978-3-642-12260-6. doi:10.1007/978-3-642-12261-3_9.

[BGO06] Sven Burmester, Holger Giese, and Oliver Oberschelp. Hybrid UML components for the design of complex self-optimizing mechatronic systems. In José Braz, Helder Araújo, Alves Vieira, and Bruno Encarnação, editors, *Informatics in Control, Automation and Robotics I*, pages 281–288. Springer Netherlands, March 2006. ISBN:978-1-4020-4136-5. doi:10.1007/1-4020-4543-3_34.

[BGP13] Fabienne Boyer, Olivier Gruber, and Damien Pous. Robust reconfigurations of component assemblies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 13–22, Piscataway, NJ, USA, May 2013. IEEE Computer Society. ISBN:978-1-4673-3076-3. doi:10.1109/ICSE.2013.6606547.

[BGS05] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In Alan Hartman and David Kreische, editors, *Proceedings of the European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA '05)*, volume 3748 of *Lecture Notes in Computer Science*, pages 25–40. Springer, Berlin/Heidelberg, November 2005. ISBN:978-3-540-30026-7. doi:10.1007/11581741_4.

[BGSH11] Christopher Brink, Joel Greenyer, Wilhelm Schäfer, and Martin Hahn. Simulation von hybridem Verhalten in CAMeL-View. In Jürgen Gausemeier, Franz-Josef Rammig, Wilhelm Schäfer, and Ansgar Trächtler, editors, *Wissenschaftsforum Intelligente Technische Systeme 2011*, volume 294 of *HNI-Verlagsschriftenreihe*. Heinz Nixdorf Institut, Universität Paderborn, May 2011. ISBN:978-3942647137.

[BGST05] Sven Burmester, Holger Giese, Andreas Seibel, and Matthias Tichy. Worst-case execution time optimization of story patterns for hard real-time systems. In *Proceedings of the 3rd International Fujaba Days 2005*, pages 71–78, September 2005.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987. ISBN:0-201-10715-5.

[BHR09] Boutheina Bennour, Ludovic Henrio, and Marcela Rivera. A reconfiguration framework for distributed components. In *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, SINTER '09, pages 49–56, New York, NY, USA, 2009. ACM. ISBN:978-1-60558-681-6. doi:10.1145/1596495.1596509.

[BIPM06] Bureau International des Poids et Mesures. *The International System of Units (SI)*, 8th edition, 2006.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, 2008. ISBN:978-0-262-02649-9.

[BK11] Björn Bartels and Moritz Kleine. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 158–167, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0575-4. doi:10.1145/1988008.1988030.

[BK13] Iva Bojic and Mario Kusek. Self-synchronization of nonidentical machines in machine-to-machine systems. In *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'13, pages 265–266. IEEE Computer Society, September 2013. doi:10.1109/saso.2013.39.

[BKPPT01] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A visualization of OCL using collaborations. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 257–271. Springer Berlin / Heidelberg, October 2001. ISBN:978-3-540-42667-7. doi:10.1007/3-540-45441-1_20.

[Blu10] Bluetooth SIG, Inc. *Bluetooth Specification Version 4.0*, June 2010. URL: `https://developer.bluetooth.org/TechnologyOverview/Pages/Core.aspx` [cited March 14, 2015].

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer New York, 2001. ISBN:978-1-4613-0091-5.

[BT12] Christian Berger and Matthias Tichy. Towards transactional self-adaption for AUTOSAR on the example of a collision detection system. In Ina Schaefer and Marcus Wille, editors, *Proceedings of the 10th Workshop Automotive Software Engineering, INFORMATIK 2012*, volume P-208 of *Lecture Notes in Informatics (LNI)*, pages 853–862. Gesellschaft für Informatik, September 2012. ISBN:978-3-88579-602-2.

[Bur06] Sven Burmester. *Model-Driven Engineering of Reconfigurable Mechatronic Systems*. PhD thesis, University of Paderborn, Warburger Str. 100, Paderborn, Germany, August 2006.

[But05] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, volume 23 of *Real-Time Systems Series*. Springer, 2 edition, 2005. ISBN:978-0-387-23137-2.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg, 2004. ISBN:978-3-540-22261-3. doi:10.1007/978-3-540-27755-2_3.

[CAC08] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–52, April 2008. ISSN:1049-331X. doi:10.1145/1348250.1348253.

[Can08] Fabio Cancare. Modeling methodologies for dynamic reconfigurable systems. Master's thesis, University of Illinois at Chicago, 2008.

[CdLG+09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2009. ISBN:978-3-642-02160-2. doi:10.1007/978-3-642-02161-9_1.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, April 1986. ISSN:0164-0925. doi:10.1145/5397.5399.

[CFJ+10] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. The EAST-ADL architecture description language for automotive embedded software. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 297–307. Springer Berlin Heidelberg, 2010. ISBN:978-3-642-16276-3. doi:10.1007/978-3-642-16277-0_11.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000. ISBN:978-0262032704.

[CGS09] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '09, pages 132 –141. IEEE Computer Society, May 2009. ISBN:978-1-4244-3724-5. doi:10.1109/seams.2009.5069082.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal – Model-driven software development*, 45(3):621–645, July 2006. ISSN:0018-8670. doi:10.1147/sj.453.0621.

[Cha06] Roderick 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*, SCS '05, pages 43–46, Darlinghurst, Australia, 2006. Australian Computer Society, Inc. ISBN:1-920-68237-6.

[CHP06] Jan Carlson, John Håkansson, and Paul Pettersson. SaveCCM: An analysable component model for real-time systems. *Electronic Notes in Theoretical Computer Science*, 160(0):127 – 140, August 2006. ISSN:1571-0661. doi:10.1016/j.entcs.2006.05.019.

[CHS01] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 635–643, Washington, DC, USA, April 2001. IEEE Computer Society. ISBN:0-7695-1077-9. doi:10.1109/ICDSC.2001.918994.

[CK99] Alongkrit Chutinan and Bruce H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In Frits Vaandrager and Jan van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin / Heidelberg, 1999. ISBN:978-3-540-65734-7. doi:10.1007/3-540-48983-5_10.

[CKNZ12] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg, 2012. ISBN:978-3-642-35745-9. doi:10.1007/978-3-642-35746-6_1.

[CSBW09] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 468–483. Springer Berlin Heidelberg, October 2009. ISBN:978-3-642-04424-3. doi:10.1007/978-3-642-04425-0_36.

[CSVC11] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593 –615, September 2011. ISSN:0098-5589. doi:10.1109/tse.2010.83.

[CY90] Peter Coad and Edward Yourdon. *Object Oriented Analysis*. Prentice-Hall, Englewood Cliffs, NJ, USA, 2nd edition edition, October 1990. ISBN:978-0136299813.

[dAH05] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In Manfred Broy, Johannes Gruenbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 83–104. Springer Netherlands, 2005. ISBN:978-1-4020-3530-2. doi:10.1007/1-4020-3532-2.

[dAHS02] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In Alberto Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2002. ISBN:978-3-540-44307-0. doi:10.1007/3-540-45828-x_9.

[Das] Dassault Systemes. *Dymola*. URL: `http://www.dymola.com` [cited March 14, 2015].

[Dav06] Alexandre David. *UPPAAL DBM Library Programmer's Reference*, October 2006. URL: `http://www.cs.aau.dk/~adavid/UDBM/manual-061023.pdf` [cited March 14, 2015].

[DDD+12] Werner Damm, Henning Dierks, Stefan Disch, Willem Hagemann, Florian Pigorsch, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Science of Computer Programming*, 77(10–11):1122 – 1150, September 2012. ISSN:0167-6423. doi:10.1016/j.scico.2011.07.006.

[Dea07] Alan Dearle. Software deployment, past, present and future. In *2007 Future of Software Engineering*, FOSE '07, pages 269–284, Washington, DC, USA, 2007. IEEE Computer Society. ISBN:0-7695-2829-5. doi:10.1109/fose.2007.20.

[DGB14] Stefan Dziwok, Sebastian Goschin, and Steffen Becker. Specifying intra-component dependencies for synthesizing component behaviors. In Federico Ciccozzi, Massimo Tivoli, and Jan Carlson, editors, *Proceedings of the 1st International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp 2014)*, pages 16–25. CEUR-WS.org Vol-1281, September 2014.

[DGR04] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, December 2004. ISSN:0098-5589. doi:10.1109/tse.2004.91.

[DHLP06] Alexandre David, John Håkansson, Kim Larsen, and Paul Pettersson. Model checking timed automata with priorities using DBM subtraction. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg, 2006. ISBN:978-3-540-45026-9. doi:10.1007/11867340_10.

[Dil90] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin / Heidelberg, February 1990. ISBN:3-540-52148-8. doi:10.1007/3-540-52148-8_17.

[DISS11a] Werner Damm, Carsten Ihlemann, and Viorica Sofronie-Stokkermans. Decidability and complexity for the verification of safety properties of reasonable linear hybrid automata. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 73–82, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0629-4. doi:10.1145/1967701.1967714.

[DISS11b] Werner Damm, Carsten Ihlemann, and Viorica Sofronie-Stokkermans. PTIME parametric verification of safety properties for reasonable linear hybrid automata. *Mathematics in Computer Science*, 5(4):469–497, December 2011. ISSN:1661-8270. doi:10.1007/s11786-011-0098-x.

[dLdCGFR06] Rogério de Lemos, Paulo Asterio de Castro Guerra, and Cecília Mary Fischer Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23(2):80–87, March 2006. ISSN:0740-7459. doi:10.1109/ms.2006.35.

[DLL+10] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: A complete specification theory for real-time systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, pages 91–100, New York, NY, USA, April 2010. ACM. ISBN:978-1-60558-955-8. doi:10.1145/1755952.1755967.

[DLLC09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annals of Telecommunications*, 64(1-2):45–63, February 2009. ISSN:0003-4347. doi:10.1007/s12243-008-0073-y.

[DMY02] Alexandre David, M. Oliver Möller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 208–241. Springer Berlin / Heidelberg, 2002. ISBN:978-3-540-43353-8. doi:10.1007/3-540-45923-5_15.

[DMY03] Alexandre David, M. Oliver Möller, and Wang Yi. Verification of UML statecharts with real-time extensions. Technical Report 2003-009, Department of Information Technology, Uppsala University, February 2003.

[DNFLP13] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer Berlin Heidelberg, October 2013. ISBN:978-3-642-35886-9. doi:10.1007/978-3-642-35887-6_2.

[DOLS13] Frederico Alvares De Oliveira, Thomas Ledoux, and Rémi Sharrock. A framework for the coordination of multiple autonomic managers in cloud environments. In *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'13, pages 179–188. IEEE Computer Society, September 2013. doi:10.1109/saso.2013.27.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Berlin Heidelberg, 2002. ISBN:978-3-642-07646-6. doi:10.1007/978-3-662-04722-4.

[dSP] dSPACE GmbH. *TargetLink Product Homepage*. URL: `https://www.dspace.com/de/gmb/home/products/sw/pcgs/targetli.cfm` [cited March 14, 2015].

[DVOW92] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992. ISSN:0925-1022. doi:10.1007/bf00124891.

[Ecla] The Eclipse Foundation. *Eclipse Model To Text (M2T)*. URL: `http://www.eclipse.org/modeling/m2t/` [cited March 14, 2015].

[Eclb] The Eclipse Foundation. *EMF Diff/Merge: a diff/merge component for models*. URL: `http://eclipse.org/diffmerge/` [cited March 14, 2015].

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006. ISBN:978-3540311874. doi:10.1007/3-540-31188-2.

[EGT+09] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidović, Gaurav Sukhatme, and Brad Petrus. Architecture-driven self-adaptation and self-management in robotics systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '09, pages 142 –151. IEEE Computer Society, May 2009. ISBN:978-1-4244-3724-5. doi:10.1109/seams.2009.5069083.

[EH10] Tobias Eckardt and Stefan Henkler. Component behavior synthesis for critical systems. In Holger Giese, editor, *Architecting Critical Systems*, volume

6150 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, June 2010. ISBN:978-3-642-13555-2. doi:10.1007/978-3-642-13556-9_4.

[ERNF12] Andreas Eggers, Nacim Ramdani, Nedialko S. Nedialkov, and Martin Fränzle. Improving the SAT modulo ODE approach to hybrid systems analysis by combining different enclosure methods. *Software & Systems Modeling*, pages 1–28, November 2012. ISSN:1619-1366. doi:10.1007/s10270-012-0295-3.

[Est] Esterel Technologies. *SCADE Suite - Product Homepage*. URL: `http://www.esterel-technologies.com/products/scade-suite/` [cited March 14, 2015].

[ETA] ETAS GmbH. *ASCET Software-Produkte*. URL: `http://www.etas.com/de/products/ascet_software_products.php` [cited March 14, 2015].

[EW92] Markus Endler and Jiawang Wei. Programming generic dynamic reconfigurations for distributed applications. In *International Workshop on Configurable Distributed Systems*, pages 68–79. IEEE, March 1992. ISBN:0-85296-544-3.

[FCT08] Lei Feng, DeJiu Chen, and Martin Törngren. Self configuration of dependent tasks for dynamically reconfigurable automotive embedded systems. In *47th IEEE Conference on Decision and Control*, CDC 2008, pages 3737–3742, December 2008. ISBN:978-1-4244-3123-6. doi:10.1109/cdc.2008.4739195.

[FFH05] Andrew Fish, Jean Flower, and John Howse. The semantics of augmented constraint diagrams. *Journal of Visual Languages & Computing*, 16(6):541 – 573, December 2005. ISSN:1045-926X. doi:10.1016/j.jvlc.2005.03.001.

[FHTW05] Andrew Fish, John Howse, Gabriele Taentzer, and Jessica Winkelmann. Two visualizations of OCL: a comparison. Technical Report VMG.05.1, University of Brighton, 2005.

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer Berlin / Heidelberg, 2011. ISBN:978-3-642-22109-5. doi:10.1007/978-3-642-22110-1_30.

[FMB+09] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. AUTOSAR - a worldwide standard is on the road. In *Proceedings of the 14th International VDI Congress Electronic Systems for Vehicles 2009*, 2009.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT '98), November 16-20, 1998, Paderborn, Germany*, volume 1764 of *Lecture Notes in Computer Science*, pages 296 – 309. Springer Berlin / Heidelberg, 2000. ISBN:3-540-67203-6. doi:10.1007/978-3-540-46464-8_21.

[Fre05] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Proceedings of the Hybrid Systems 8th International Workshop on Computation and Control (HSCC 2005)*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer Berlin Heidelberg, March 2005. ISBN:3-540-25108-1. doi:10.1007/978-3-540-31954-2_17.

[Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons, 1st edition, April 2004. ISBN:978-0471471639.

[GAOR07] Matthias Güdemann, Andreas Angerer, Frank Ortmeier, and Wolfgang Reif. Modeling of self-adaptive systems with SCADE. In *IEEE International Symposium on Circuits and Systems*, ISCAS 2007, pages 2922 –2925. IEEE Computer Society, May 2007. ISBN:1-4244-0920-9. doi:10.1109/iscas.2007.377861.

[GB03] Holger Giese and Sven Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Software Engineering Group, University of Paderborn, Paderborn, Germany, June 2003.

[GBD07] Leif Geiger, Thomas Buchmann, and Alexander Dotor. EMF code generation with Fujaba. In *Proceedings of the Fifth International Fujaba Days*, October 2007.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proceedings of the 12th ACM SIGSOFT Foundations of Software Engineering*, FSE 2004, pages 179–188, New York, NY, USA, November 2004. ACM. ISBN:1-58113-855-5. doi:10.1145/1029894.1029920.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004. ISSN:0018-9162. doi:10.1109/mc.2004.175.

[GCW+02] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. Components for embedded software: The

PECOS approach. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 19–26, New York, NY, USA, 2002. ACM. ISBN:1-58113-575-0. doi:10.1145/581630.581634.

[GCZ08] Heather J. Goldsby, Betty H. C. Cheng, and Ji Zhang. AMOEBA-RT: Runtime verification of adaptive software. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 212–224. Springer Berlin Heidelberg, 2008. ISBN:978-3-540-69069-6. doi:10.1007/978-3-540-69073-3_23.

[Gei13] Johannes Geismann. Codegenerierung für LEGO Mindstorms - Roboter. Bachelor's thesis, University of Paderborn, January 2013.

[Ger13] Christopher Gerking. Transparent Uppaal-based verifcation of MechatronicUML models. Master's thesis, University of Paderborn, May 2013.

[GFDK09] Jürgen Gausemeier, Ursula Frank, Jörg Donoth, and Sascha Kahl. Specification technique for the description of self-optimizing mechatronic systems. *Research in Engineering Design*, 20:201–223, 2009. ISSN:0934-9839. doi:10.1007/s00163-008-0058-x.

[GHS09] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. Improved flexibility and scalability by interpreting story diagrams. In Tiziana Margaria, Julia Padberg, and Gabriele Taentzer, editors, *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, volume 18. Electronic Communications of the EASST, 2009.

[Gie03] Holger Giese. A formal calculus for the compositional pattern-based design of correct real-time systems. Technical Report tr-ri-03-240, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Deutschland, July 2003.

[GJSH12] Mohammad Ghafari, Pooyan Jamshidi, Saeed Shahbazi, and Hassan Haghighi. An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE '12, pages 177–182, New York, NY, USA, June 2012. ACM. ISBN:978-1-4503-1345-2. doi:10.1145/2304736.2304765.

[GKP08] Jürgen Gausemeier, Sascha Kahl, and Sebastian Pook. From mechatronics to self-optimizing systems. In *Self-optimizing Mechatronic Systems: Design the Future, 7th International Heinz Nixdorf Symposium*, volume 223. HNI Verlagsschriftenreihe, Paderborn, February 2008.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of component-based systems*, pages 47–67.

Cambridge University Press, New York, NY, USA, March 2000. ISBN:0-521-77164-1.

[GOS09] Nicola Guarino, Daniel Oberle, and Steffen Staab. What is an ontology? In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 1–17. Springer Berlin Heidelberg, 2009. ISBN:978-3-540-70999-2. doi:10.1007/978-3-540-92673-3_0.

[Gos14] Sebastian Goschin. Synthesis of extended hierarchical real-time behavior. Master's thesis, University of Paderborn, February 2014.

[GPVW96] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd. ISBN:0-412-71620-8.

[Gra] Graphviz Open Source Team. *Graphviz - Graph Visualization Software*. URL: `http://www.graphviz.org/` [cited March 14, 2015].

[Gre11] Joel Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, University of Paderborn, October 2011.

[Gro01] Object Management Group. *OMG Unified Modeling Language Specification 1.4*, September 2001. Document formal/2001-09-67. URL: `http://www.omg.org/spec/UML/1.4/`.

[Gro05] Object Management Group. *Unified Modeling Language (UML) 2.0 Superstructure Specification*, July 2005. Document formal/2005-07-04. URL: `http://www.omg.org/spec/UML/2.0/`.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project – A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Addison-Wesley, 1st edition, March 2009. ISBN:978-0321534071.

[Gro10] Object Management Group. *OMG Systems Modeling Language (OMG SysML)*, June 2010. Document formal/10-06-02. URL: `http://www.sysml.org/specs/`.

[Gro11a] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification - Part 3: CORBA Component Model*, November 2011. Document formal/2011-11-03. URL: `http://www.omg.org/spec/CORBA/3.2/`.

[Gro11b] Object Management Group. *Query/View/Transformation (QVT) 1.1*, January 2011. Document formal/2011-01-01. URL: `http://www.omg.org/spec/QVT/1.1/`.

[Gro11c] Object Management Group. *Unified Modeling Language (UML) 2.4.1 Superstructure Specification*, August 2011. Document formal/2011-08-06.

[Gro12] Object Management Group. *Object Constraint Language (OCL) 2.3.1*, January 2012. Document formal/2012-01-01. URL: http://www.omg.org/spec/OCL/2.3.1/.

[Gro14] Object Management Group. *Model Driven Architecture (MDA) – MDA Guide rev. 2.0*, June 2014. Document – ormsc/14-06-01. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[GRS09] Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer, editors. *Selbstoptimierende Systeme des Maschinenbaus – Definitionen, Anwendungen, Konzepte*, volume 234. HNI-Verlagsschriftenreihe, Paderborn, 1st edition, January 2009. ISBN:978-3-939350-53-8.

[GRS14] Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer, editors. *Design Methodology for Intelligent Technical Systems*. Lecture Notes in Mechanical Engineering. Springer, 2014. ISBN:978-3-642-45434-9.

[GS04] David Garlan and Bradley Schmerl. Using architectural models at runtime: Research challenges. In Flavio Oquendo, Brian C. Warboys, and Ron Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 200–205. Springer Berlin Heidelberg, 2004. ISBN:978-3-540-22000-8. doi:10.1007/978-3-540-24769-2_15.

[GS13] Holger Giese and Wilhelm Schäfer. Model-driven development of safe self-optimizing mechatronic systems with MechatronicUML. In Javier Cámara, Rogério de Lemos, Carlo Ghezzi, and Antónia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 152–186. Springer Berlin Heidelberg, January 2013. ISBN:978-3-642-36248-4. doi:10.1007/978-3-642-36249-1_6.

[GSB+08] Heather J. Goldsby, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Danny Hughes. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS 2008., pages 36–45. IEEE Computer Society, March 2008. ISBN:0-7695-3141-5. doi:10.1109/ecbs.2008.22.

[GSG+09] Jürgen Gausemeier, Wilhelm Schäfer, Joel Greenyer, Sascha Kahl, Sebastian Pook, and Jan Rieke. Management of cross-domain model consistency during the development of advanced mechatronic systems. In Margareta Norell Bergendahl, Martin Grimheden, Larry Leifer, Philipp Skogstad, and Udo Lindemann, editors, *Proceedings of the 17th International Conference on Engineering Design*, volume 6, Design Methods and Tools of *ICED'09*, pages 1–12. Design Society, August 2009. ISBN:978-1-904670-10-0.

[GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and modelbased code generation for MDA-tools. In Holger Giese and Albert Zündorf, editors, *Proceedings of the Third International Fujaba Days 2005*, vol-

ume tr-ri-06-275 of *Technical Report*, pages 1–6. University of Paderborn, September 2005.

[GTB+03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan 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*, ESEC/FSE'03, pages 38–47, New York, NY, USA, September 2003. ACM. ISBN:1-58113-743-5. doi:10.1145/940071.940078.

[GTS14] Jürgen Gausemeier, Ansgar Trächtler, and Wilhelm Schäfer, editors. *Semantische Technologien im Entwurf mechatronischer Systeme: Effektiver Austausch von Lösungswissen in Branchenwertschöpfungsketten*. Carl Hanser Verlag, München, June 2014. ISBN:978-3446436305.

[GV14] Jürgen Gausemeier and Mareen Vaßholz. Design methodology for self-optimizing systems. In Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer, editors, *Design Methodology for Intelligent Technical Systems*, chapter 3.1, pages 66–69. Springer-Verlag, Heidelberg, Germany, January 2014. ISBN:978-3-642-45434-9.

[Ham05] Gregoire Hamon. A denotational semantics for Stateflow. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 164–172, New York, NY, USA, 2005. ACM. ISBN:1-59593-091-4. doi:10.1145/1086228.1086260.

[HB07] Petr Hnětynka and Tomáš Bureš. Advanced features of hierarchical component models. In Jaroslav Zendulka, editor, *Proceedings of the 10th International Conference on Information System Implementation and Modeling*, ISIM'07, pages 1–8. CEUR-WS.org Vol-252, April 2007.

[HBRV10] Ábel Hegedüs, Gábor Bergmann, István Rath, and Dániel Varró. Back-annotation of simulation traces with change-driven model transformations. In *8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM'10, pages 145–155. IEEE Computer Society, September 2010. ISBN:978-1-4244-8289-4. doi:10.1109/sefm.2010.28.

[HC01] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN:0-201-70485-4.

[HCH12] Yin Hang, Jan Carlson, and Hans Hansson. Towards mode switch handling in component-based multi-mode systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE'12, pages 183–188, New York, NY, USA, June 2012. ACM. ISBN:978-1-4503-1345-2. doi:10.1145/2304736.2304766.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 278–292, Los Alamitos, CA, USA, July 1996. IEEE Computer Society. ISBN:0-8186-7463-6. doi:10.1109/lics.1996.561342.

[Hen12] Stefan Henkler. *Ein komponentenbasierter, modellgetriebener Softwareentwicklungsansatz für vernetzte, mechatronische Systeme*. PhD thesis, University of Paderborn, Warburger Str. 100, Paderborn, Germany, June 2012.

[HESV91] Ann Hsu, Farokh Eskafi, Sonia Sachs, and Pravin Varaiya. Design of platoon maneuver protocols for IVHS. Technical Report UCB-ITS-PRR-91-6, UC Berkeley: California Partners for Advanced Transit and Highways (PATH), April 1991. URL: `http://escholarship.org/uc/item/89c6p0cn`.

[HH13] Yin Hang and Hans Hansson. Handling multiple mode switch scenarios in component-based multi-mode systems. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference*, volume 1 of *APSEC'13*, pages 404–413. IEEE Computer Society, December 2013. ISBN:978-1-4799-2143-0. doi:10.1109/apsec.2013.61.

[HHG08] Martin Hirsch, Stefan Henkler, and Holger Giese. Modeling collaborations with dynamic structural adaptation in Mechatronic UML. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 33–40, New York, NY, USA, May 2008. ACM. ISBN:978-1-60558-037-1. doi:10.1145/1370018.1370026.

[HHMWT00] Thomas Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond HyTech: Hybrid systems analysis using interval numerical methods. In Nancy Lynch and Bruce Krogh, editors, *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 130–144. Springer Berlin / Heidelberg, March 2000. ISBN:978-3-540-67259-3. doi:10.1007/3-540-46430-1_14.

[HHR09] Alexander Harhurin, Judith Hartmann, and Daniel Ratiu. Motivation and formal foundations of a comprehensive modeling theory for embedded systems. Technical Report TUM-I0924, Institut für Informatik, Technische Universität München, September 2009.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):110–122, 1997. ISSN:1433-2779. doi:10.1007/s100090050008.

[HIM98] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *Proceedings of the third international workshop on Software architecture*, ISAW '98, pages 69–72, New York, NY, USA, 1998. ACM. ISBN:1-58113-081-3. doi:10.1145/288408.288426.

[Hir08] Martin Hirsch. *Modell-basierte Verifikation von vernetzten mechatronischen Systemen*. PhD thesis, University of Paderborn, Warburger Str. 100, Paderborn, Germany, September 2008.

[HKMU06] Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastián Uchitel. Modes for software architectures. In Volker Gruhn and Flavio Oquendo, editors, *Software Architecture*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer Berlin Heidelberg, 2006. ISBN:978-3-540-69271-3. doi:10.1007/11966104_9.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94 – 124, 1998. ISSN:0022-0000. doi:10.1006/jcss.1998.1581.

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Secaucus, NJ, USA, 2003. ISBN:3540007873.

[HM08a] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, May 2008. ISSN:1619-1366. doi:10.1007/s10270-007-0054-z.

[HMTN+08] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus component model for resource constrained real-time systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*, SIES 2008, pages 177–183. IEEE Computer Society, June 2008. ISBN:978-1-4244-1994-4. doi:10.1109/SIES.2008.4577697.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994. ISSN:0890-5401. doi:10.1006/inco.1994.1045.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985. ISBN:978-0131532717.

[HOG04] Thorsten Hestermeyer, Oliver Oberschelp, and Holger Giese. Structured information processing for self-optimizing mechatronic systems. In Helder Araújo, Alves Vieira, José Braz, Bruno Encarnação, and Marina Carvalho, editors, *Proceedings of 1st International Conference on Informatics in Control, Automation and Robotics*, ICINCO 2004, pages 230–237. INSTICC Press, August 2004.

[HP06] Petr Hnětynka and František Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In Ian Gorton, George Heineman, Ivica Crnković, Heinz Schmidt, Judith Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume

4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer Berlin / Heidelberg, 2006. ISBN:978-3-540-35628-8. doi:10.1007/11783565_27.

[HPB+10] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka, and Michal Malohlava. Comparison of component frameworks for real-time embedded systems. In Lars Grunske, Ralf Reussner, and František Plášil, editors, *Component Based Software Engineering*, volume 6092 of *Lecture Notes in Compute Science*, pages 21–36. Springer, Berlin/Heidelberg, 2010. ISBN:978-3-642-13237-7. doi:10.1007/978-3-642-13238-4_2.

[HQCH13] Yin Hang, Hongwan Qin, Jan Carlson, and Hans Hansson. Mode switch handling for the ProCom component model. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, CBSE '13, pages 13–22, New York, NY, USA, June 2013. ACM. ISBN:978-1-4503-2122-8. doi:10.1145/2465449.2465451.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, UK, 2nd edition, 2004. ISBN:978-0-521-54310-1.

[HR07] Gregoire Hamon and John Rushby. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):447–456, July 2007. ISSN:1433-2779. doi:10.1007/s10009-007-0049-7.

[HTS+08a] Christian Henke, Matthias Tichy, Tobias Schneider, Joachim Böcker, and Wilhelm Schäfer. Organization and control of autonomous railway convoys. In *Proceedings of the 9th International Symposium on Advanced Vehicle Control*, AVEC'08, pages 318–323, October 2008.

[HTS+08b] Christian Henke, Matthias Tichy, Tobias Schneider, Joachim Böcker, and Wilhelm Schäfer. System architecture and risk management for autonomous railway convoys. In *Proceedings of the 2nd Annual IEEE International Systems Conference,*, pages 1–8. IEEE Computer Society, April 2008. ISBN:978-1-4244-2149-7. doi:10.1109/SYSTEMS.2008.4518986.

[HZ14] Christian Hölscher and Detmar Zimmer. Intelligent drive module (iDM). In Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer, editors, *Design Methodology for Intelligent Technical Systems*, chapter 2.1.2, pages 33–36. Springer, Heidelberg, Germany, January 2014.

[IBM06] IBM. An architectural blueprint for autonomic computing. Autonomic computing white paper, IBM, June 2006.

[IEC96] IEC. Graphical symbols for diagrams. *IEC 60617*, 1996.

[IEC10] IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems. *IEC 61508 Edition 2.0*, April 2010.

[IEE08] IEEE. IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages c1–269, July 2008. doi:10.1109/ieeestd.2008.4579760.

[IETF81] Internet Engineering Task Force (IETF). *RFC793 - Transmission Control Protocol*, September 1981. URL: `http://tools.ietf.org/html/rfc793`.

[ISO10] ISO. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1 –418, December 2010. doi:10.1109/ieeestd.2010.5733835.

[ISO11a] ISO. Road vehicles – functional safety. *ISO26262:2011*, November 2011.

[IUH11] Daisuke Ishii, Kazunori Ueda, and Hiroshi Hosobe. An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems. *International Journal on Software Tools for Technology Transfer*, 13(5):449–461, October 2011. ISSN:1433-2779. doi:10.1007/s10009-011-0193-y.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995. ISSN:0098-5589. doi:10.1109/32.385973.

[IW14] M. Usman Iftikhar and Danny Weyns. Assuring system goals under uncertainty with active formal models of self-adaptation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 604–605, New York, NY, USA, May 2014. ACM. ISBN:978-1-4503-2768-8. doi:10.1145/2591062.2591137.

[iXt] iXtronics GmbH. *CAMeL-View Product Homepage*. URL: `http://www.ixtronics.com/ix_hist/English/CAMeLView.htm` [cited March 14, 2015].

[JFA+07] A. Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust test generation and coverage for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *Hybrid Systems: Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 329–342. Springer Berlin Heidelberg, April 2007. ISBN:978-3-540-71492-7. doi:10.1007/978-3-540-71493-4_27.

[JLS00] Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou. Scaling up Uppaal – automatic verification of real-time systems using compositionality and abstraction. In Joseph Mathai, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 19–30. Springer Berlin Heidelberg, September 2000. ISBN:978-3-540-41055-3. doi:10.1007/3-540-45352-0_4.

[Ken97] Stuart Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 327–341, New York, NY, USA, 1997. ACM. ISBN:0-89791-908-4. doi:10.1145/263698.263756.

[KG07] Florian Klein and Holger Giese. Joint structural and temporal property specification using timed story scenario diagrams. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 185–199. Springer Berlin Heidelberg, March 2007. ISBN:978-3-540-71288-6. doi:10.1007/978-3-540-71289-3_16.

[Kil05] Christopher Kilian. *Modern Control Technology*. Delmar Cengage Learning, 3rd edition, 2005. ISBN:978-1401858063.

[Kiz05] Joseph Migga Kizza. *Computer Network Security*. Springer US, May 2005. ISBN:978-0-387-20473-4.

[KKH+08] Ji Eun Kim, Rahul Kapoor, Martin Herrmann, Jochen Haerdtlein, Franz Grzeschniok, and Peter Lutz. Software behavior description of real-time embedded systems in component based software development. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, ISORC'08, pages 307–311. IEEE Computer Society, May 2008. ISBN:978-0-7695-3132-8. doi:10.1109/isorc.2008.69.

[KKJ12] Slim Kallel, Mohamed Hadj Kacem, and Mohamed Jmaiel. Modeling and enforcing invariants of dynamic software architectures. *Software and Systems Modeling*, 11(1):127–149, February 2012. ISSN:1619-1366. doi:10.1007/s10270-010-0162-z.

[KKMR11] Kay Klobedanz, Andreas Koenig, Wolfgang Mueller, and Achim Rettberg. Self-reconfiguration for fault-tolerant FlexRay networks. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 207–216. IEEE Computer Society, March 2011. ISBN:978-1-4577-0303-4. doi:10.1109/isorcw.2011.38.

[KKTS09] Thomas Kuhn, Sören Kemmann, Mario Trapp, and Christian Schäfer. Multi-language development of embedded systems. In Matti Rossi, Jonathan Sprinkle, Jeff Gray, and Juha-Pekka Tolvanen, editors, *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, DSM '09, pages 21–27, Helsinki, Finland, October 2009. Helsinki School of Economics. ISBN:978-952-488-372-6.

[KM98] Jeff Kramer and Jeff Magee. Analysing dynamic change in software architectures: A case study. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, CDS '98, pages

91–100. IEEE Computer Society, May 1998. ISBN:0-8186-8451-8. doi:10.1109/CDS.1998.675762.

[KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking - timed UML state machines and collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer Berlin Heidelberg, September 2002. ISBN:978-3-540-44165-6. doi:10.1007/3-540-45739-9_23.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston / Dordrecht / London, 1st edition, 1997. ISBN:978-0792398947.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 1990. ISSN:0922-6443. doi:10.1007/bf01995674.

[KPP95] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995. ISSN:0740-7459. doi:10.1109/52.391832.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Upper Saddle River, NJ, USA, second edition, 1988. ISBN:0-13-110362-8.

[KR06] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In Antti Valmari, editor, *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer Berlin Heidelberg, April 2006. ISBN:978-3-540-33102-5. doi:10.1007/11691617_19.

[KRKH09] Ji Eun Kim, Oliver Rogalla, Simon Kramer, and Arne Hamann. Extracting, specifying and predicting software system properties in component based real-time embedded software development. In *31st International Conference on Software Engineering - Companion Volume*, pages 28–38. IEEE Computer Society, May 2009. ISBN:978-1-4244-3495-4. doi:10.1109/icse-companion.2009.5070961.

[KSA07] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. COMDES-II: A component-based framework for generative development of distributed real-time control systems. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 199–208. IEEE Computer Society, 2007. ISBN:0-7695-2975-5. doi:10.1109/rtcsa.2007.29.

[KSHL12] Karsten Krügel, Lars Stockmann, Dominik Holler, and Klaus Lamberg. Simulation-based development and testing environment for electric vehicles. In Clemens Gühmann, Jens Riese, and Thieß-Magnus Wolter, editors, *Simulation und Test für die Automobilelektronik IV*, pages 242–255. Expert Verlag, Renningen, May 2012. ISBN:978-3-8169-3121-8.

[KSP03] Tamás Kovácsházy, Gábor Samu, and Gábor Péceli. Simulink block library for fast prototyping of reconfigurable DSP systems. In *IEEE International Symposium on Intelligent Signal Processing*, pages 179–184. IEEE Computer Society, September 2003. ISBN:0-7803-7864-4. doi:10.1109/isp.2003.1275835.

[KT14] Jan Henning Keßler and Ansgar Trächtler. Active suspension module. In Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer, editors, *Design Methodology for Intelligent Technical Systems*, chapter 2.1.4, pages 39–42. Springer, Heidelberg, Germany, January 2014.

[KTW02] Christiane Kiesner, Gabriele Taentzer, and Jessica Winkelmann. VisualOCL: A visual notation of the Object Constraint Language. Technical Report No. 2002/23, Computer Science Department of the Technical University of Berlin, 2002.

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *International Journal on Software and Systems Modeling (SoSyM)*, 5(4):369 – 385, December 2006. ISSN:1619-1366. doi:10.1007/s10270-006-0017-9.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. ISSN:0098-5589. doi:10.1109/TSE.1977.229904.

[Lau06] Kung-Kiu Lau. Software component models. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 1081–1082, New York, NY, USA, 2006. ACM. ISBN:1-59593-375-1. doi:10.1145/1134285.1134516.

[LBD+10] Shuhao Li, Sandie Balaguer, Alexandre David, Kim G. Larsen, Brian Nielsen, and Saulius Pusinskas. Scenario-based verification of real-time systems using UPPAAL. *Formal Methods in System Design*, 37(2-3):200–264, 2010. ISSN:0925-9856. doi:10.1007/s10703-010-0103-z.

[LLC10] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in a reflective component model. In Lars Grunske, Ralf Reussner, and František Plášil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 74–92. Springer Berlin / Heidelberg, 2010. ISBN:978-3-642-13237-7. doi:10.1007/978-3-642-13238-4_5.

[LM98] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998. ISSN:0098-5589. doi:10.1109/32.708567.

[LNGE11] Markus Luckey, Benjaming Nagel, Christian Gerth, and Gregor Engels. Adapt Cases: Extending use cases for adaptive systems. In

Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, pages 30–39, New York, NY, USA, May 2011. ACM. ISBN:978-1-4503-0575-4. doi:10.1145/1988008.1988014.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In Horst Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer Berlin Heidelberg, August 1995. ISBN:978-3-540-60249-1. doi:10.1007/3-540-60249-6_41.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709 –724, October 2007. ISSN:0098-5589. doi:10.1109/tse.2007.70726.

[Maa05] Hugh Maaskant. A robust component model for consumer electronic products. In Peter Stok, editor, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research Book Series*, pages 167–192. Springer Netherlands, 2005. ISBN:978-1-4020-3454-1. doi:10.1007/1-4020-3454-7_7.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Notices*, 22(12):147–155, December 1987. ISSN:0362-1340. doi:10.1145/38807.38821.

[Mata] The MathWorks, Inc. *Chart Simulation Semantics*. URL: `http://www.mathworks.de/de/help/stateflow/chart-simulation-semantics.html` [cited March 14, 2015].

[Matb] The MathWorks, Inc. *Create Bus Signals*. URL: `http://www.mathworks.de/de/help/simulink/ug/creating-and-accessing-a-bus.html` [cited March 14, 2015].

[Matc] The MathWorks, Inc. *Debugging*. URL: `http://www.mathworks.de/de/help/stateflow/debugging.html` [cited March 14, 2015].

[Matd] The MathWorks, Inc. *MATLAB Homepage*. URL: `http://www.mathworks.com/products/matlab/` [cited March 14, 2015].

[Mate] The MathWorks, Inc. *Model Based Testing*. URL: `http://www.mathworks.de/discovery/model-based-testing.html` [cited March 14, 2015].

[Matf] The MathWorks, Inc. *Signal Basics*. URL: `http://www.mathworks.de/de/help/simulink/ug/signal-basics.html` [cited March 14, 2015].

[Matg] The MathWorks, Inc. *Simulink Product Homepage*. URL: `http://www.mathworks.com/products/simulink` [cited March 14, 2015].

[Math] The Mathworks, Inc. *Stateflow – Product Homepage*. URL: `http://www.mathworks.de/products/stateflow/` [cited March 14, 2015].

[MBG+11] Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 245–255, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0443-6. doi:10.1145/2025113.2025148.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, September 1955. ISSN:0005-8580. doi:10.1002/j.1538-7305.1955.tb03788.x.

[Mey09] Matthias Meyer. *Musterbasiertes Re-Engineering von Softwaresystemen*. PhD thesis, University of Paderborn, Warburger Str. 100, Paderborn, Germany, December 2009.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN:0387102353.

[Mit04] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 2004. ISBN:0-07-115467-1.

[MMMR12] Sam Malek, Nenad Medvidović, and Marija Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012. ISSN:0098-5589. doi:10.1109/tse.2011.3.

[Mod09] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling*. Modelica Association, May 2009. Version 3.1.

[Mon01] Robert T. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163R, Computer Science Department, School of Computer Science, Carnegie Mellon University, January 2001.

[Moo09] Daniel L. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756 –779, November 2009. ISSN:0098-5589. doi:10.1109/tse.2009.67.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–40, September 1992. ISSN:0890-5401. doi:10.1016/0890-5401(92)90008-4.

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56 – 64, July 2004. ISSN:0018-9162. doi:10.1109/mc.2004.48.

[MT00] Nenad Medvidović and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. ISSN:0098-5589. doi:10.1109/32.825767.

[NBP] Neue Bahntechnik Paderborn. *RailCab – Towards Innovative Railroad Traffic for Future Mobility*. URL: `http://www.railcab.de/index.php?id=2&L=1` [cited March 14, 2015].

[NIS01] National Institute of Standards. *Advanced Encryption Standard (AES)*, November 2001. FIPS PUB 197. URL: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[NNSS07] Thomas Narten, Erik Nordmark, William A. Simpson, and Hesham Soliman. *RFC4861 - Neighbor Discovery for IP version 6 (IPv6)*. Network Working Group, September 2007. URL: `http://tools.ietf.org/html/rfc4861`.

[OHY11] Jungsup Oh, Mark Harman, and Shin Yoo. Transition coverage testing for Simulink/Stateflow models using messy genetic algorithms. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1851–1858, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0557-0. doi:10.1145/2001576.2001825.

[OMT98] Peyman Oreizy, Nenad Medvidović, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 177–186. IEEE Computer Society, April 1998. ISBN:0-8186-8368-6. doi:10.1109/ICSE.1998.671114.

[OMT+08] Semir Osmic, Eckehard Münch, Ansgar Trächtler, Stefan Henkler, Wilhelm Schäfer, Holger Giese, and Martin Hirsch. Safe online-reconfiguration of self-optimizing mechatronic systems. In Jürgen Gausemeier, Franz J. Rammig, and Wilhelm Schäfer, editors, *Selbstoptimierende mechatronische Systeme: Die Zukunft gestalten. 7. Internationales Heinz Nixdorf Symposium für industrielle Informationstechnik*, pages 411–426, February 2008.

[Ora13] Oracle. *JSR 345: Enterprise JavaBeansTM, Version 3.2, EJB Core Contracts and Requirements*, April 2013. URL: `http://download.oracle.com/otn-pub/jcp/ejb-3_2-fr-eval-spec/ejb-3_2-core-fr-spec.pdf` [cited March 14, 2015].

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Krüger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society. ISBN:0-7695-2829-5. doi:10.1109/fose.2007.22.

[PDM+14] Uwe Pohlmann, Stefan Dziwok, Matthias Meyer, Matthias Tichy, and Sebastian Thiele. A Modelica coordination pattern library for cyber-physical systems. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, SIMUTools'14, pages 76–85, Brussels, Belgium, March 2014. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN:978-1-63190-007-5. doi:10.4108/icst.simutools.2014.254640.

[PDS+12] Uwe Pohlmann, Stefan Dziwok, Julian Suck, Boris Wolf, Chia Choon Loh, and Matthias Tichy. A Modelica library for real-time coordination modeling. In *Proceedings of the 9th International MODELICA Conference*, pages 365–374. DLR - Robotics and Mechatronics Center, Modelica Association, Linköping University Electronic Press, Linköpings universitet, September 2012. ISBN:978-91-7519-826-2. doi:10.3384/ecp12076365.

[Pea84] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. ISBN:0-201-05594-5.

[PHMG14] Uwe Pohlmann, Jörg Holtmann, Matthias Meyer, and Christopher Gerking. Generating Modelica models from software specifications for the simulation of cyber-physical systems. In *Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '14, pages 191–198. IEEE Computer Society, August 2014. doi:10.1109/SEAA.2014.18.

[PKH+11] Tomáš Pop, Jaroslav Keznikl, Petr Hošek, Michal Malohlava, Tomáš Bureš, and Petr Hnětynka. Introducing support for embedded and real-time devices into existing hierarchical component system: Lessons learned. In *9th International Conference on Software Engineering Research, Management and Applications*, SERA'11, pages 3 –11. IEEE Computer Society, August 2011. ISBN:978-1-4577-1028-5. doi:10.1109/sera.2011.14.

[PKP07] Carlos Paiz, Boris Keltelhoit, and Mario Porrmann. A design framework for FPGA-based dynamically reconfigurable digital controllers. In *IEEE International Symposium on Circuits and Systems*, ISCAS 2007, pages 3708–3711. IEEE Computer Society, May 2007. ISBN:1-4244-0920-9. doi:10.1109/iscas.2007.378648.

[PLM12] Valerio Panzica La Manna. Local dynamic update for component-based distributed systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE '12, pages 167–176, New York, NY, USA, 2012. ACM. ISBN:978-1-4503-1345-2. doi:10.1145/2304736.2304764.

[Plu82] David C. Plummer. *RFC826 - An Ethernet Address Resolution Protocol*. Network Working Group, November 1982. URL: `http://tools.ietf.org/html/rfc826`.

[Plu06] Andrew R. Plummer. Model-in-the-loop testing. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 220(3):183–199, May 2006. ISSN:0959-6518. doi:10.1243/09596518jsce207.

[PMDB14] Uwe Pohlmann, Matthias Meyer, Andreas Dann, and Christopher Brink. Viewpoints and views in hardware platform modeling for safe deployment. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '14, pages 23:23–23:30, New York, NY, USA, July 2014. ACM. ISBN:978-1-4503-2900-2. doi:10.1145/2631675.2631682.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, October 1977. doi:10.1109/SFCS.1977.32.

[Poh13] Uwe Pohlmann. Safe deployment for reconfigurable cyber-physical systems. In *Proceedings of the 18th International Doctoral Symposium on Components and Architecture*, WCOP '13, pages 31–36, New York, NY, USA, June 2013. ACM. ISBN:978-1-4503-2125-9. doi:10.1145/2465498.2465503.

[Pos80] John Postel. *User Datagram Protocol*. Internet Engineering Task Force (IETF), August 1980. URL: `http://tools.ietf.org/html/rfc768`.

[PPO+12] Tomáš Pop, František Plášil, Matej Outly, Michal Malohlava, and Tomáš Bureš. Property networks allowing oracle-based mode-change propagation in hierarchical components. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE'12, pages 93–102, New York, NY, USA, June 2012. ACM. ISBN:978-1-4503-1345-2. doi:10.1145/2304736.2304753.

[Pri13] Claudia Priesterjahn. *Analyzing Self-healing Operations in Mechatronic Systems*. PhD thesis, University of Paderborn, Warburger Str. 100, Paderborn, Germany, August 2013.

[PSR+12] Uwe Pohlmann, Wilhelm Schäfer, Hendrik Reddehase, Jens Röckemann, and Robert Wagner. Generating functional mockup units from software specifications. In *Proceedings of the 9th International MODELICA Conference*, pages 765–774. Linköping University Electronic Press, Linköpings universitet, September 2012. ISBN:978-91-7519-826-2. doi:10.3384/ecp12076765.

[PST13] Claudia Priesterjahn, Dominik Steenken, and Matthias Tichy. Timed hazard analysis of self-healing systems. In Javier Cámara, Rogério de Lemos, Carlo Ghezzi, and Antónia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 112–151. Springer Berlin Heidelberg, January 2013. ISBN:978-3-642-36248-4. doi:10.1007/978-3-642-36249-1_5.

[PTD+14] Uwe Pohlmann, Henning Trsek, Lars Dürkop, Stefan Dziwok, and Felix Oestersötebier. 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*, ETFA'14, pages 1–6. IEEE Computer Society, September 2014. doi:10.1109/ETFA.2014.7005358.

[PTH+10] Claudia Priesterjahn, Matthias Tichy, Stefan Henkler, Martin Hirsch, and Wilhelm Schäfer. Fujaba4Eclipse Real-Time Tool Suite. In Holger Giese, Gabor Karsai, Edward A. Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)*, volume 6100 of *Lecture Notes in Computer Science*, chapter 12, pages 309–315. Springer Berlin Heidelberg, 2010. ISBN:978-3-642-16276-3. doi:10.1007/978-3-642-16277-0_12.

[PV14] Marco Panunzio and Tullio Vardanega. A component-based process with separation of concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96:105 – 121, October 2014. ISSN:0164-1212. doi:10.1016/j.jss.2014.05.076.

[PWT+08] Marek Prochazka, Roger Ward, Petr Tuma, Petr Hnětynka, and Jiri Adamek. A component-oriented framework for spacecraft on-board software. In *Proceedings of DASIA 2008, DAta Systems In Aerospace, Palma de Mallorca, European Space Agency Report Nr. SP-665*, May 2008. ISBN:978-92-9221-229-2.

[Qua] Quanser Inc. *Quanser Real-Time Control Software (QUARC)*. URL: `http://www.quarcservice.com/ReleaseNotes/files/dynamic_reconfiguration.html` [cited March 14, 2015].

[RC08] Andres J. Ramirez and Betty H. C. Cheng. Verifying and analyzing adaptive logic through UML state models. In *1st International Conference on Software Testing, Verification, and Validation*, pages 529–532. IEEE Computer Society, April 2008. ISBN:978-0-7695-3127-4. doi:10.1109/icst.2008.67.

[RCC10] Tom Robinson, Eric Chan, and Erik Coelingh. Operating platoons on public motorways: An introduction to the SARTRE platooning programme. In *17th World Congress on Intelligent Transport Systems*, October 2010.

[Rei07] Peter Reineke. Model checking von Story-Diagrammen mittels GROOVE. Bachelor's thesis, University of Paderborn, July 2007.

[Ren06] Arend Rensink. Model checking quantified computation tree logic. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 110–125. Springer Berlin Heidelberg, August 2006. ISBN:978-3-540-37376-6. doi:10.1007/11817949_8.

[Ren07] Arend Rensink. Isomorphism checking in GROOVE. In Albert Zündorf and Dániel Varró, editors, *Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006)*, volume 1 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, September 2007.

[Ren08] Arend Rensink. Explicit state model checking for graph grammars. In Pierpaolo Degano, Rocco Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132. Springer Berlin Heidelberg, 2008. ISBN:978-3-540-68676-7. doi:10.1007/978-3-540-68679-8_8.

[RJC12] Andres J. Ramirez, Adam C. Jensen, and Betty H. C. Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *Proceedings of the 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'12, pages 99 –108. IEEE Computer Society, June 2012. ISBN:978-1-4673-1788-7. doi:10.1109/seams.2012.6224396.

[Roz97] Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. ISBN:9810228848.

[RS08a] Steve Reeves and David Streader. General refinement, part one: Interfaces, determinism and special refinement. *Electronic Notes in Theoretical Computer Science*, 214:277–307, June 2008. ISSN:1571-0661. doi:10.1016/j.entcs.2008.06.013.

[RS08b] Steve Reeves and David Streader. General refinement, part two: Flexible refinement. *Electronic Notes in Theoretical Computer Science*, 214:309–329, June 2008. ISSN:1571-0661. doi:10.1016/j.entcs.2008.06.014.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2nd edition, December 2008. ISBN:978-0321331885.

[Sch95] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin / Heidelberg, June 1995. ISBN:978-3-540-59071-2. doi:10.1007/3-540-59071-4_45.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006. ISSN:0018-9162. doi:10.1109/mc.2006.58.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002. ISBN:0-201-74572-0.

[SH99] Pyda Srisuresh and Matt Holdrege. *RFC2663 - IP Network Address Translator (NAT) Terminology and Considerations*. Network Working Group, August 1999. URL: `http://tools.ietf.org/html/rfc2663`.

[Sha02] Mary Shaw. "self-healing": softening precision to avoid brittleness: position paper for WOSS '02: workshop on self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 111–114, New York, NY, USA, 2002. ACM. ISBN:1-58113-609-9. doi:10.1145/582128.582152.

[SHS12] Lars Stockmann, Dominik Holler, and Dirk Spenneberg. Early simulation and testing of virtual ECUs for electric vehicles. In *International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium (EVS26)*, May 2012.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages - A Laboratory Based Approach*. Addison-Wesley, 1995. ISBN:0-201-65697-3.

[SK00] B. Izaias Silva and Bruce H. Krogh. Formal verification of hybrid systems using CheckMate: A case study. In *Proceedings of the 2000 American Control Conference*, volume 3, pages 1679 –1683. IEEE Computer Society, June 2000. ISBN:0-7803-5519-9. doi:10.1109/acc.2000.879487.

[SK06] Elisabeth A. Strunk and John C. Knight. Dependability through assured reconfiguration in embedded system software. *IEEE Transactions on Dependable and Secure Computing*, 3(3):172 –187, July-Sept. 2006. ISSN:1545-5971. doi:10.1109/tdsc.2006.33.

[SLT09] Francoise Simonot-Lion and Yvon Trinquet. Vehicle functional domains and their requirements. In Nicolas Navet and Francoise Simonot-Lion, editors, *Automotive Embedded Systems Handbook*, Industrial Information Technology Series, chapter 1. CRC Press, Boca Raton, FL, USA, 2009. ISBN:978-0-8493-8026-6.

[SP12] Ingo Stürmer and Hartmut Pohlheim. Model quality assessment in practice: How to measure and assess the quality of software models during the embedded software development process. In *Proceedings of the International Congress of Embedded Real Time Software and Systems*, ERTS'12, February 2012.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992. ISBN:0-13-978529-9.

[SRKC00] B. Izaias Silva, Keith Richeson, Bruce H. Krogh, and Alongkrit Chutinan. Modeling and verifying hybrid dynamic systems using CheckMate. In *Proceedings of the 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems*, ADPM 2000, pages 323–328. Shaker Verlag GmbH, September 2000. ISBN:978-3826578366.

[SS83] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, 1983. ISSN:0098-5589. doi:10.1109/tse.1983.236608.

[SSdR05] Maty Sylla, Frank Stomp, and Willem-Paul de Roever. Verifying parameterized refinement. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS 2005, pages 313 – 321. IEEE Computer Society, June 2005. ISBN:0-7695-2284-X. doi:10.1109/iceccs.2005.82.

[Sta08] Florian Stallmann. *A Model-Driven Approach to Multi-Agent System Design*. PhD thesis, Software Engineering Group, University of Paderborn, April 2008.

[Ste97] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, July 1997. ISSN:0001-5903. doi:10.1007/s002360050095.

[Ste07] Andrea Steinke. Integration Hybrider Rekonfigurationscharts mit Matlab/Simulink-Modellen. Diplomarbeit, University of Paderborn, September 2007.

[SV03] Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 92–95. Springer Berlin Heidelberg, October 2003. ISBN:978-3-540-20243-1. doi:10.1007/978-3-540-45221-8_8.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development – Technology, Engineering, Management*. John Wiley & Sons, Ltd., 1 edition, May 2006. ISBN:978-0470025703.

[SW07] Wilhelm Schäfer and Heike Wehrheim. The challenges of building advanced mechatronic systems. In *Future of Software Engineering*, FOSE '07, pages 72–84. IEEE Computer Society, May 2007. ISBN:0-7695-2829-5. doi:10.1109/FOSE.2007.28.

[SWB12] Michael Schulze, Jens Weiland, and Danilo Beuche. Automotive model-driven development and the challenge of variability. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 207–214, New York, NY, USA, 2012. ACM. ISBN:978-1-4503-1094-9. doi:10.1145/2362536.2362565.

[SWZ95] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234. Springer Berlin Heidelberg, September 1995. ISBN:978-3-540-60406-8. doi:10.1007/3-540-60406-5_17.

[T-V] T-VEC Technologies, Inc. *T-VEC Tester for Simulink and Stateflow*. URL: `http://www.t-vec.com/solutions/simulink.php` [cited March 14, 2015].

[tBGS13] Maurice H. ter Beek, Fabio Gadducci, and Francesco Santini. Validating reconfigurations of Reo circuits in an e-banking scenario. In *Proceedings of the 4th international ACM Sigsoft symposium on Architecting critical systems*, ISARCS '13, pages 39–48, New York, NY, USA, June 2013. ACM. ISBN:978-1-4503-2123-5. doi:10.1145/2465470.2465474.

[TDH11] Jacques Thomas, Christian Dziobek, and Bernd Hedenetz. Variability management in the AUTOSAR-based development of applications for in-vehicle systems. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 137–140, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0570-9. doi:10.1145/1944892.1944909.

[TFS10] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. A family of languages for architecture constraint specification. *Journal of Systems and Software*, 83(5):815 – 831, May 2010. ISSN:0164-1212. doi:10.1016/j.jss.2009.11.736.

[TGS06] Matthias Tichy, Holger Giese, and Andreas Seibel. Story diagrams in real-time software. In Holger Giese and Bernhard Westfechtel, editors, *Proceedings of the 4th International Fujaba Days*, volume tr-ri-06-275 of *Technical Report*, pages 15–22. University of Paderborn, September 2006.

[THB+10] Matthias Tichy, Martin Hirsch, Christopher Brink, Wilhelm Schäfer, Christopher Gerking, and Martin Hahn. Integration hybrider Modellierungstechniken in CAMeL-View. In *7. Paderborner Workshop Entwurf mechatronischer Systeme*, volume 272, pages 235–251, Paderborn, 2010. HNI-Verlagsschriftenreihe. ISBN:978-3-939350-91-0.

[THHO08] Matthias Tichy, Stefan Henkler, Jörg Holtmann, and Simon Oberthür. Component story diagrams: A transformation language for component structures in mechatronic systems. In *Postproceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, pages 27–39, 2008.

[THP+07] Wolfgang Trumler, Markus Helbig, Andreas Pietzowski, Benjamin Satzger, and Theo Ungerer. Self-configuration and self-healing in AUTOSAR. In *Proceedings of the 14th Asia Pacific Automotive Engineering Conference*, APAC-14, pages 25–36, Hollywood, California, USA, August 2007. SAE International. doi:10.4271/2007-01-3507.

[Tic09] Matthias Tichy. *Gefahrenanalyse selbstoptimierender Systeme*. Dissertation, University of Paderborn, Warburger Str. 100, Paderborn, Germany, May 2009.

[TMD09] Richard N. Taylor, Nenad Medvidović, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, February 2009. ISBN:978-0-470-16774-8.

[Tri09] Stavros Tripakis. Checking timed Büchi automata emptiness on simulation graphs. *ACM Transactions on Computational Logic (TOCL)*, 10(3):15:1–15:19, April 2009. ISSN:1529-3785. doi:10.1145/1507244.1507245.

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):779–818, November 2005. ISSN:1539-9087. doi:10.1145/1113830.1113834.

[TSDF11] Chouki Tibermacine, Salah Sadou, Christophe Dony, and Luc Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, CBSE '11, pages 31–40, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0723-9. doi:10.1145/2000229.2000235.

[TSL13] Mario Trapp, Daniel Schneider, and Peter Liggesmeyer. A safety roadmap to cyber-physical systems. In Jürgen Münch and Klaus Schmid, editors, *Perspectives on the Future of Software Engineering*, pages 81–94. Springer, Berlin/Heidelberg, 2013. ISBN:978-3-642-37394-7. doi:10.1007/978-3-642-37395-4_6.

[VDI04] VDI. *VDI 2206: Entwicklungsmethodik für mechatronische Systeme*. Verein Deutscher Ingenieure, 2004.

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007. ISSN:0098-5589. doi:10.1109/tse.2007.70733.

[Ven80] John Venn. On the diagrammatic and mechanical representation of propositions and reasonings. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, 10(58):1–18, 1880. doi:10.1080/14786448008626877.

[VG14] Thomas Vogel and Holger Giese. Model-driven engineering of self-adaptive software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):18:1–18:33, January 2014. ISSN:1556-4665. doi:10.1145/2555612.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 247–248, New York, NY, USA, 2012. ACM. ISBN:978-1-4503-1202-8. doi:10.1145/2188286.2188326.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE'01, pages 249–262. IEEE Computer Society, August 2001. ISBN:0-7695-1125-2. doi:10.1109/isre.2001.948567.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78 –85, March 2000. ISSN:0018-9162. doi:10.1109/2.825699.

[VSC+09] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the ProCom real-time component model. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, SEEA '09, pages 478–485, Los Alamitos, CA, USA, August 2009. IEEE Computer Society. ISBN:978-0-7695-3784-9. doi:10.1109/seaa.2009.53.

[VWMA11] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 202–207, New York, NY, USA, 2011. ACM. ISBN:978-1-4503-0575-4. doi:10.1145/1988008.1988037.

[W3C10] W3C. *XML Path Language (XPath) 2.0 (Second Edition)*, December 2010. URL: `http://www.w3.org/TR/2010/REC-xpath20-20101214/` [cited March 14, 2015].

[WA13] Danny Weyns and Jesper Andersson. On the challenges of self-adaptation in systems of systems. In *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*, SESoS '13, pages 47–51, New York, NY, USA, 2013. ACM. ISBN:978-1-4503-2048-1. doi:10.1145/2489850.2489860.

[Wan05] Farn Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. *IEEE Transactions on Software Engineering*, 31(1):38–51, January 2005. ISSN:0098-5589. doi:10.1109/tse.2005.13.

[WDR11] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. Timed automata for the development of real-time systems. Technical Report 2011-579, Queen's University, 2011.

[WH07] Conal Watterson and Donal Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1(5):172–179, October 2007. ISSN:1751-8806. doi:10.1049/iet-sen:20060076.

[Wha10] Michael W. Whalen. A parametric structural operational semantics for Stateflow, UML Statecharts, and Rhapsody. Technical Report 2010-1, University of Minnesota Software Engineering Center, August 2010.

[WL97] Carsten Weise and Dirk Lenzkes. Efficient scaling-invariant checking of timed bisimulation. In Rüdiger Reischuk and Michel Morvan, editors, *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS '97)*, volume 1200 of *Lecture Notes in Computer Science*, pages 177–188. Springer Berlin Heidelberg, February 1997. ISBN:978-3-540-62616-9. doi:10.1007/BFb0023458.

[WSB+09] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In *Proceedings of the 17th IEEE International Requirements Engineering Conference*, RE'09, pages 79–88. IEEE Computer Society, August 2009. ISBN:978-0-7695-3761-0. doi:10.1109/RE.2009.36.

[WSG+13] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. On patterns for decentralized control in self-adaptive systems. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer Berlin Heidelberg, 2013. ISBN:978-3-642-35812-8. doi:10.1007/978-3-642-35813-5_4.

[XC13] Dingyü Xue and Yang Quan Chen. *System Simulation Techniques with MATLAB and Simulink*. John Wiley & Sons, September 2013. ISBN:978-1118647929.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 371–380, New York, NY, USA, May 2006. ACM. ISBN:1-59593-375-1. doi:10.1145/1134285.1134337.

[ZCYM05] Ji Zhang, Betty H. C. Cheng, Zhenxiao Yang, and Philip K. McKinley. Enabling safe dynamic component-based software adaptation. In Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 194–211. Springer Berlin Heidelberg, 2005. ISBN:978-3-540-28968-5. doi:10.1007/11556169_9.

[ZGC09] Ji Zhang, Heather J. Goldsby, and Betty H. C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 161–172, New York, NY, USA, 2009. ACM. ISBN:978-1-60558-442-3. doi:10.1145/1509239.1509262.

[Zig08] ZigBee Standards Organization. *ZigBee Specification*, January 2008. Document 053474r17.

[Zim07] Dirk Zimmer. Enhancing Modelica towards variable structure systems. *Simulation News Europe*, 17(2):23–28, September 2007. ISSN:0929-2268.

[ZJS08] Günther Zauner, Florian Judex, and Peter Schwarz. Classical and statechart-based modeling of state events and of structural changes in the Modelica simulator Mosilab. *Simulation News Europe (SNE)*, 18(2):17–23, 2008. ISSN:0929-2268.

[ZP12] Marc Zeller and Christian Prehofer. Timing constraints for runtime adaptation in real-time, networked embedded systems. In *2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2012, pages 73–82. IEEE Computer Society, June 2012. ISBN:978-1-4673-1788-7. doi:10.1109/SEAMS.2012.6224393.

[ZPW+11] Marc Zeller, Christian Prehofer, Gereon Weiss, Dirk Eilers, and Rudi Knorr. Towards self-adaptation in real-time, networked systems: Efficient solving of system constraints for automotive embedded systems. In *Proceedings of the Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'11, pages 79–88. IEEE Computer Society, October 2011. ISBN:978-1-4577-1614-0. doi:10.1109/saso.2011.19.

[ZTB08] Mohamed H. Zaki, Sofiène Tahar, and Guy Bois. Formal verification of analog and mixed signal designs: A survey. *Microelectronics Journal*, 39(12):1395 – 1404, 2008. ISSN:0026-2692. doi:10.1016/j.mejo.2008.05.013.

[Zün95] Albert Zündorf. *PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung*. PhD thesis, RWTH Aachen, 1995.

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development*. Habilitation, Software Engineering Group, University of Paderborn, 2001.

[Zün09] Albert Zündorf. Model checking the leader election protocol with Fujaba. In Tihamer Levendovszky, Arend Rensink, and Pieter Van Gorp, editors, *Fifth International Workshop on Graph Based Tools*, GraBaTs, pages 1–11, July 2009.

[ZW14] Steffen Ziegert and Heike Wehrheim. Temporal plans for software architecture reconfiguration. *Computer Science - Research and Development*, pages 1–18, 2014. ISSN:1865-2034. Published Online July 2014. doi:10.1007/s00450-014-0259-7.

List of Abbreviations

ACI The Atomicity, Consistency, and Isolation properties of a database system. 8, 9, 71, 107, 108

ACI-T ACI properties combined with a correct Timing. 71–73, 99, 100, 108–110, 193

ADL Architecture Description Language 66, 68, 69

AMS Autonomous Mechatronic System 11–14, 26, 33–35, 37, 39, 56–58, 60, 70, 111, 193, 211, 212

ATCTL $\forall$-quantified Timed Computation Tree Logic, a variant of TCTL that only uses $\forall$-path quantifiers 122, 165

CIC Component Instance Configuration 47, 48, 50, 65, 66, 70, 77, 82, 100–102, 110, 111, 147–149, 153–155, 170, 172, 180, 182, 187, 198, 222, 242, 243, 275, 278, 304, 379

CSD Component Story Diagram 49–52, 54–56, 63, 65–67, 70, 72, 73, 77, 79, 80, 87, 88, 100–102, 104, 107, 110, 147, 170, 172, 175, 178, 184, 193, 199, 225, 230, 238, 242–253, 259, 301, 309, 314, 373, 374, 377, 378

CTL Computation Tree Logic 18, 19, 102, 120, 122

DBM Difference Bound Matrix 133, 289, 299

EMF Eclipse Modeling Framework 65, 184, 291, 294, 295, 299–301

GTS Graph Transformation System 19–21, 102

LHS Left Hand Side of a Graph Transformation Rule 21–23, 53, 55, 101, 296

LTL Linear-Time Temporal Logic 19, 67, 102, 120, 122, 138, 139

MFM Mechatronic Function Module 11

MIL Model-in-the-loop simulation 7, 9, 100, 142, 143, 146, 147, 185, 189–191

MSD Modal Sequence Diagram 57, 59

NAC Negative Application Condition of a Graph Transformation Rule 21–23

NMS Networked Mechatronic System 12, 33, 56, 109, 111, 139, 193, 196, 197

NTA Network of Timed Automata 15–19, 31, 103, 104, 124, 132, 283–285, 287, 298, 374

OCL Object Constraint Language 35, 49, 65, 68, 69, 301, 302, 314

OCM Operator-Controller-Module 12, 26, 34, 39–41, 56, 70, 75, 86, 113, 142, 195, 196

QoS Quality-of-Service 31, 32, 142, 149, 158–160, 321

RHS Right Hand Side of a Graph Transformation Rule 21–23, 53, 55, 101

RTCP Real-Time Coordination Protocol 26–28, 31, 32, 35–37, 39, 42–44, 46, 56–61, 65, 70, 111–115, 122, 132, 134, 135, 138, 139, 157, 187, 189, 193, 194, 197, 199–211, 215–218, 225, 234, 235, 238, 243, 251, 373, 374, 376

RTSC Real-Time Statechart 27–31, 33, 37–39, 57, 58, 60, 65, 72, 79, 82–84, 89, 91–100, 103, 104, 106, 110–120, 124, 125, 127, 128, 131–137, 139, 141, 144, 148–150, 159, 160, 162–170, 173–180, 182, 184, 185, 187, 194, 196–213, 215–219, 224–238, 251–259, 272, 283, 289, 298, 299, 301, 302, 307, 311, 373–379

SDD Story Decision Diagram 61–66, 68, 70, 72, 86, 87, 102, 176, 177, 193, 199, 224, 225, 232, 240, 253, 266–274, 309, 316, 374, 378, 379

TCTL Timed Computation Tree Logic 18, 19, 102, 115, 122

TGG Triple Graph Grammar 160, 184

WCET Worst-Case Execution Time 87, 104–106, 175

List of Figures