

**Specification and Verification of Real-Time Coordination
Protocols for Cyber-physical Systems**

by

Stefan Dziwok



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

SPECIFICATION AND VERIFICATION FOR REAL-TIME COORDINATION PROTOCOLS OF CYBER-PHYSICAL SYSTEMS

PhD Thesis

Submitted in partial fulfillment
of the requirements for the degree of
“Doktor der Naturwissenschaften” (Dr. rer. nat.)

by

STEFAN DZIWOK

Schulstraße 39, 33102 Paderborn

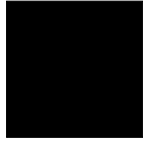
Supervised by

Prof. Dr. Wilhelm Schäfer

and Prof. Dr. Matthias Tichy

Paderborn, September 11, 2017





ABSTRACT

Cyber-physical systems (CPSs) are the next generation of embedded systems that heavily interact with each other and their environment to fulfill advanced functionality. The coordination between them may be realized in software via asynchronous message communication. A software engineer that designs this coordination has to consider that a CPS must be safe and obeys hard real-time constraints because any (timing) error may lead to severe damage and even loss of human life. Consequently, the development of CPSs requires formal verification like model checking to guarantee the functional correctness of the software. Model checkers are not appropriate for specifying CPSs as they have no built-in support for domain-specific aspects like asynchronous communication. Instead, software engineers should use a domain-specific modeling language (DSL) that encapsulates domain concepts effectively. Though, DSLs typically do not enable model checking directly. Therefore, state of the art approaches typically transform the DSL model into the input model of an external model checker. In this model, the software engineer has to manually set the verification properties, execute the model checking, and examine the counterexamples of the model checker (the latter are used to identify the cause of the error). However, many software engineers have no deep experience in model checking, which makes this approach error-prone and time-consuming. The concept of domain-specific model checking (DSMC) by Visser et al. hides this complexity successfully by providing an additional DSL for the specification of verification properties, by transforming these properties into the input language of the model checker, by automating the model checking, and by transforming counterexamples back to the DSL level. However, DSMC has not been applied to the coordination design of CPSs yet.

In this thesis, we present a model-driven approach for formally specifying and verifying the message-based coordination of CPSs. In particular, we improve a DSL called Real-Time Coordination Protocols (RTCPs) that enables to specify the coordination behavior of CPSs via asynchronous message exchange while obeying real-time constraints. We additionally provide formal design patterns for RTCPs that capture existing modeling experience resulting in less modeling errors. Moreover, we contribute a second DSL for specifying verification properties, which considers the (domain-specific) aspects of RTCPs and the capabilities of the model checker. Finally, we provide a DSMC for RTCPs and their domain properties that fully hides the complexity of the model checker.

We evaluate all of our contributions based on realistic examples, e.g., a coordinated overtaking maneuver. As a highlight, we show—using case studies—that both DSLs encapsulate the domain concepts effectively and that our DSMC works correctly for 16 different RTCPs.



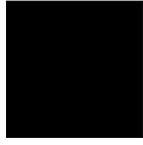
ZUSAMMENFASSUNG

Cyber-physische Systeme (CPSs) sind die nächste Generation von eingebetteten Systemen, die fortwährend miteinander und ihrer Umgebung interagieren, um anspruchsvolle Funktionen zu erfüllen. Die Koordination zwischen ihnen kann in Software durch asynchrone Nachrichtenkommunikation realisiert werden. Ein Softwareingenieur, der diese Koordination entwirft, muss berücksichtigen, dass ein CPS sicher sein und harten Echtzeitanforderungen gehorchen muss, weil jeder (Timing-) Fehler zu schweren Schäden und sogar zu menschlichen Verlusten führen kann. Folglich erfordert die Entwicklung von CPSs eine formale Verifikation wie z.B. Model Checking, um die funktionale Korrektheit der Software zu gewährleisten. Model Checker sind nicht geeignet für die Spezifikation von CPSs, da sie keine integrierte Unterstützung für domänenspezifische Aspekte wie asynchrone Kommunikation besitzen. Stattdessen sollten Softwareingenieure eine domänenspezifische Modellierungssprache (engl. domain-specific language, DSL) verwenden, da sie Domänenkonzepte effektiv kapseln kann. Ein Problem ist jedoch, dass DSLs in der Regel Model Checking nicht direkt ermöglichen. Typischerweise transformiert der Stand der Technik daher das DSL-Modell in das Eingabemodell eines externen Model Checkers. In diesem muss der Softwareingenieur manuell die Verifikationseigenschaften festlegen, das Model Checking durchführen und die Gegenbeispiele des Model Checkers untersuchen (letztere dienen zur Identifikation der Fehlerursache). Allerdings haben viele Softwareingenieure keine tiefergehende Erfahrung mit Model Checkern und deren Sprachen, was diesen Ansatz fehleranfällig und zeitaufwändig macht. Das Konzept des domänenspezifischen Model Checkings (DSMC) von Visser et al. verbirgt diese Komplexität erfolgreich, indem es eine zusätzliche DSL zur Spezifikation der Verifikationseigenschaften einführt, diese Eigenschaften ebenfalls in die Eingabesprache des Model Checkers transformiert, das Model Checking automatisiert und Gegenbeispiele zurück auf die DSL-Ebene transformiert. Bisher wurde das Konzept von DSMC jedoch nicht auf den Entwurf der Koordination von CPSs angewendet.

In dieser Arbeit präsentieren wir einen modellgetriebenen Ansatz zur formalen Spezifikation und Verifikation der nachrichtenbasierten Koordination von CPSs. Insbesondere verbessern wir eine bestehende DSL namens Real-Time Coordination Protocols (RTCPs), die es ermöglicht das Koordinationsverhalten von CPSs mittels asynchronem Nachrichtenaustausch unter Beachtung von Echtzeitanforderungen zu spezifizieren. Zusätzlich definieren wir formale Entwurfsmuster für RTCPs, die die existierende Modellierungserfahrung festhalten, wodurch die Anzahl der Modellierungsfehler gesenkt werden soll. Darüber hinaus präsentieren wir eine zweite DSL zur Spezifizierung von Verifikationseigenschaften, die die (domänenspezifischen) Aspekte von RTCPs und die Fähigkeiten des Model

Checkers berücksichtigt. Letztlich stellen wir ein DSMC-Konzept für RTCPs und deren Domäneneigenschaften vor, das die Komplexität des Model Checkers vollständig verbirgt.

Wir evaluieren alle unsere Beiträge anhand realistischer Beispiele, z.B. anhand eines koordinierten Überholmanövers. Ein Highlight sind hierbei Fallstudien, welche zeigen, dass beide DSLs die Domänenkonzepte effektiv kapseln und dass unsere Realisierung des DSMCs für 16 verschiedene RTCPs korrekt arbeitet.



ACKNOWLEDGMENTS

This work would not have been possible without the tremendous support from several people and the great working atmosphere at the software engineering research group at Paderborn.

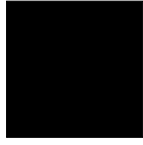
First of all, I especially thank my doctoral adviser Prof. Dr. Wilhelm Schäfer and Prof. Dr. Matthias Tichy. Wilhelm Schäfer incorporated me in the software engineering group and guided me throughout the years while Matthias Tichy significantly supported me in the final years of writing this thesis. I thank Prof. Dr. Matthias Tichy and Prof. Dr. Heike Wehrheim for writing their reports and Prof. Dr. Steffen Becker, Dr. Stefan Sauer as well as Dr. Ben Hermann for attending my oral exam.

Special thanks goes to two groups of colleagues: First, my office mates over the years, namely Christopher Brink, Tobias Eckardt, Lars Stockmann, Uwe Pohlmann, and Benedict Wohlers, for providing a comfortable working atmosphere. Second, the MECHATRONIC-UML developer group—especially Christopher Gerking, Dr. Christian Heinzemann, Uwe Pohlmann, and David Schubert—for the intensive discussions and the excellent collaboration while developing MECHATRONICUML and the MECHATRONICUML Tool Suite.

In addition, thanks go out to my further supervisors and colleagues at the software engineering group, the Heinz Nixdorf Institute, and Fraunhofer IEM for the scientific exchange and the enjoyable social activities that made my time at Paderborn a joyful and valuable experience. Among others, these are Dr. Matthias Becker, Prof. Dr. Eric Bodden, Dr. Christian Brenner, Andreas Dann, Jens Friebe, Johannes Geismann, Jörg Holtmann, Renate Löffler, Dr. Matthias Meyer, Goran Piskachev, Dr. Marie Christin Platenius, Dr. Claudia Priesterjahn, Dr. Jan Rieke, David Schmelter, Johannes Späth, Christian Stritzke, Julian Suck, Oliver Sudmann, Dietrich Travkin, and Dr. Markus von Detten. Moreover, Jutta Haupt and Jürgen “Sammy” Maniera were always very helpful when administrative or technical problems arose. I thank you for this a lot.

I am grateful to all students that supported me as this thesis and its implementation would not have been possible without them. In particular, I would like to thank Ingo Budde and Sebastian Thiele. Furthermore, I thank everyone who was proof-reading parts of this thesis or parts of my publications.

Most importantly, I could not have succeeded without the love, encouragement, and support of my family. I especially thank my mother Martina, her life companion Barbara, and my father Horst for providing me with the support to start a scientific career. But first and foremost, I thank my beloved wife Katrin for her trust and belief in me, her patience, and her extraordinary support despite all the things she had to endure during the last years.



CONTENTS

Titlepage	IV
Abstract	V
Zusammenfassung	VIII
Acknowledgments	IX
Table of Contents	XVI
1 Introduction	1
1.1 Running Example: Coordinated Overtaking	4
1.2 Problem Definition	6
1.2.1 Specification of Real-Time Coordination Protocols	6
1.2.2 Specification of Domain-Specific Verification Properties for CPS	7
1.2.3 Domain-Specific Model Checking (DSMC) of RTCP	8
1.2.4 Reusing Experience for the RTCP Design	9
1.3 Contribution	10
1.4 Thesis Structure	13
2 Foundations	17
2.1 Timed Model Checking	17
2.1.1 Timed Automata	17
2.1.2 Timed Computation Tree Logic (TCTL)	21
2.1.3 Model Checking Process	23
2.1.4 Uppaal Tooling	24
2.2 MechatronicUML	24
2.2.1 Compositional Verification Approach and DSL Overview	24
2.2.2 Real-Time Statecharts	27
2.2.3 MechatronicUML Tool Suite	31
3 Real-Time Coordination Protocols	33
3.1 Contributions	33
3.2 Stakeholder and Requirements	34

3.3	Structural Specification	36
3.3.1	Role	37
3.3.2	Role Connector	39
3.3.3	Classification of Real-Time Coordination Protocols	41
3.3.4	Incoming Message Buffer	42
3.3.5	Real-Time Coordination Protocol Instance	46
3.3.6	Unsupported Structures	47
3.4	Modeling Assumptions	48
3.4.1	Assumed Software Layers	48
3.4.2	Assumed Steps of a Message Transmission	49
3.4.3	Quality of Service (QoS) Assumptions	50
3.4.4	Discussion	52
3.5	Behavioral Specification	54
3.5.1	Single Role Behavior Specification	55
3.5.2	Multi Role Behavior Specification	57
3.6	Applying RTCPs into the Component Model	78
3.6.1	Incoming Message Buffers at Discrete Ports	79
3.6.2	Quality-of-Service Assumptions at the Component Model	79
3.6.3	One-To-Many Communication Schemata	80
3.6.4	Example	80
3.7	Implementation	84
3.7.1	User Interface	84
3.7.2	Plugin Structure	85
3.8	Evaluation	86
3.8.1	Case Study Context	87
3.8.2	Setting the Hypotheses	87
3.8.3	Preparing the Validation	87
3.8.4	Validating the Hypotheses	88
3.8.5	Threats to Validity	92
3.8.6	Analyzing the Results	94
3.9	Limitations	94
3.10	Related Work	95
3.10.1	Contracts and Protocols	95
3.10.2	Behavioral Connectors	97
3.10.3	Component Models	98
3.11	Summary	100
4	Domain-specific Verification Properties for RTCPs	103
4.1	Contributions	103
4.2	Stakeholder and Requirements	104
4.3	MTCTL - A Domain-Specific Verification Property Language for MechatronicUML	105
4.3.1	Simplified EBNF Grammar	107
4.3.2	Basic Semantics	109
4.3.3	Temporal Quantifiers	110
4.3.4	Referencing Elements	111

4.3.5	Predicates	113
4.3.6	Operations	116
4.3.7	Combining Temporal Properties	117
4.4	Default Verification Properties for RTCPs	120
4.5	Translating MTCTL into English Sentences	123
4.6	Implementation	129
4.6.1	User Interface	129
4.6.2	Plugin Structure	130
4.7	Evaluation	131
4.7.1	Case Study Context	132
4.7.2	Setting the Hypotheses	132
4.7.3	Preparing the Validation	132
4.7.4	Validating the Hypotheses	133
4.7.5	Threats to Validity	134
4.7.6	Analyzing the Results	135
4.8	Limitations	135
4.9	Related Work	136
4.9.1	TCTL-Based Languages	136
4.9.2	OCL-Based Languages	137
4.9.3	Generating the DSVPL	138
4.9.4	DSVPLs in MechatronicUML	139
4.9.5	Expressing Verification Properties in Natural English	139
4.9.6	Patterns for Verification Properties	140
4.10	Summary	141
5	Domain-specific Model Checking of RTCPs	143
5.1	Contributions	143
5.2	Stakeholder and Requirements	144
5.3	DMSC for CPS by Utilizing Model-To-Model Traceability	146
5.4	Model-To-Model Translation from RTCPs to Uppaal	148
5.4.1	Protocol to CIC Translation	150
5.4.2	MTCTL Set Quantifier Normalization	151
5.4.3	MTCTL Static Evaluation Normalization	153
5.4.4	MTCTL Split Properties Translation	155
5.4.5	Time Unit Normalization	156
5.4.6	Renaming Identifiers Normalization	156
5.4.7	RTSC Deadline Normalization	156
5.4.8	RTSC Composite Transition Normalization	158
5.4.9	RTSC State Do-Effect Normalization	161
5.4.10	RTSC Hierarchy Normalization	163
5.4.11	RTSC State Entry- / Exit-Effect Normalization	171
5.4.12	RTSC Urgency Normalization	171
5.4.13	MechatronicUML to Uppaal Migration	174
5.5	Uppaal-Level Activities	187
5.5.1	Layouting Timed Automata Templates	187
5.5.2	Model-to-Text-Transformation	188

5.5.3	Automating Uppaal	190
5.5.4	Text-to-Model-Transformation	192
5.6	Model-To-Model Back-Translation from Uppaal to RTCPs	194
5.6.1	UTCTL to MTCTL	195
5.6.2	Uppaal Trace to MechatronicUML Trace	196
5.7	Implementation	199
5.7.1	User Interface	199
5.7.2	Plugin Structure	203
5.8	Evaluation	204
5.8.1	Case Study Context	204
5.8.2	Setting the Hypotheses	205
5.8.3	Preparing the Validation	206
5.8.4	Validating the Hypotheses	209
5.8.5	Threats to Validity	217
5.8.6	Analyzing the Results	220
5.9	Limitations	221
5.10	Related Work	222
5.10.1	Translating DSLs for Communicating CPS to a Model Checking	222
5.10.2	DSMC	223
5.11	Summary	225
6	Formal Design Patterns for RTCPs	227
6.1	Contributions	227
6.2	Stakeholder and Requirements	228
6.3	Real-Time Coordination Patterns	229
6.3.1	Concept	230
6.3.2	Definition	231
6.3.3	Identified Patterns	232
6.3.4	Description Format of Real-Time Coordination Patterns	233
6.3.5	Example: The Pattern Acquire Assurance	235
6.4	Abstracting an RTCP to a Pattern	241
6.5	Selecting and Adapting a Pattern to an RTCP	245
6.6	Implementation	248
6.6.1	User Interface	249
6.6.2	Plugin Structure	249
6.7	Evaluation	250
6.7.1	Case Study Context	250
6.7.2	Setting the Hypotheses	251
6.7.3	Preparing the Validation	252
6.7.4	Validating the Hypotheses	253
6.7.5	Threats to Validity	254
6.7.6	Analyzing the Results	255
6.8	Limitations	255
6.9	Related Work	256
6.9.1	Informal Design Patterns	256
6.9.2	Formal Design Patterns	257

6.10 Summary	258
7 Conclusion	261
7.1 Summary	261
7.2 Future Work	263
A Catalog of One-To-Many Communication Schemata	267
A.1 Normalization Template for Multicast with Subrole Condition	267
A.2 Normalization Template for Multireceive with Subrole Condition	269
A.3 Unicast	271
A.3.1 Definition	271
A.3.2 Normalization Template	273
A.4 Singlereceive	275
A.4.1 Definition	275
A.4.2 Normalization Template	276
A.5 Iterate	278
A.5.1 Definition	278
A.5.2 Normalization Template	280
A.6 Loadbalancing	282
A.6.1 Definition	282
A.6.2 Normalization Templates	284
B Catalog of Real-Time Coordination Patterns	289
B.1 Request Information	289
B.2 Fail-operational Delegation	292
B.3 Fail-safe Delegation	295
B.4 Event-based Transmission	298
B.5 Periodic Transmission	301
B.6 Alternating Lock	304
B.7 Block Execution	306
B.8 Limit Observation	309
C Evaluation Details	313
C.1 Examples	313
C.1.1 Overtaking Cars (Extended)	313
C.1.2 RailCab	321
C.1.3 BeBot	328
C.2 MTCTL	334
C.3 DSMC	339
D Details of the RTCP Overtaking	341
D.1 Uppaal Representation	341
D.2 Exemplary SVG Counterexample Trace	352
E Implementation Details	357
E.1 RTCP	357
E.1.1 Package Core	358

E.1.2	Package ValueType	358
E.1.3	Package Behavior	359
E.1.4	Package Action Language	359
E.1.5	Packages Connector and MsgType	361
E.1.6	Packages Protocol and Constraint	362
E.1.7	Package Realtimestatechart	363
E.1.8	Package One-To-Many Communication Schemata	366
E.1.9	Package Component	367
E.1.10	Package Instance	369
E.2	MTCTL	371
E.2.1	Metamodel	371
E.2.2	Grammar	373
E.3	DSMC of RTCPs via Uppaal	377
E.3.1	DSMC Options Metamodel	378
E.3.2	MechatronicUML Verification Extension Metamodel	378
E.3.3	Uppaal Timed Automata Metamodel	379
E.3.4	UTCTL Metamodel	384
E.3.5	Uppaal Trace Grammar	384
E.3.6	Uppaal Trace Metamodel	386
E.3.7	MechatronicUML Trace Metamodel	387
E.4	Real-Time Coordination Patterns	390
E.4.1	Metamodel	390
E.4.2	Ontology	391
F	List of Abbreviations	393
G	Paper Contributions	395
	Bibliography	399
	Own Publications	399
	Supervised Theses	402
	Literature	404
	Norms and Specifications	419
	List of Figures	423
	List of Tables	429

INTRODUCTION

Cyber-physical systems (CPSs) [aca11] are the next generation of technical systems. Their novelty is that they no longer work in isolation. Instead, by forming a system of systems, they heavily interact with each other and their environment in order to fulfill advanced functionality [SW07]. For example, in the automotive industry, the interaction between several cars (called car-to-car) and between cars and infrastructure (called car-to-infrastructure) are emerging trends to improve the traffic flow and to avoid accidents [Car2Car]. However, the interaction is typically not permanent as CPSs may change their location as well as their internal objectives¹ at runtime. Therefore, CPSs shall remain independent actors that coordinate their behaviors towards their mutually defined objectives. The coordination between CPSs is typically realized in software via message-based communication because this enables an exchange of substantial information within a short amount of time and without the need of visual contact between them (which is an advantage compared to sensor-based communication).

A software engineer has to take into account that a CPS must be safe while defining the coordination between the CPSs, i.e., the “system does not, under defined conditions, lead to a state in which human life, health, property, or the environment is endangered” [ISO24765, p. 316]. This is especially relevant for the message-based communication as the absence of a message or an incorrect message order can lead to false conclusions about the other systems, and, therefore, to safety-critical situations. Thus, a software engineer must be able to analyze whether the software under development and especially the message exchange is safe w.r.t. these situations. Consequently, traditional testing is not sufficient as it only analyzes a subset of all possible execution traces. Therefore, safety-critical errors might not be revealed.

In addition, the software of a CPS runs under hard real-time constraints [aca11; Kop11] that must always be obeyed. Otherwise, safety-critical situations may not be avoided. Consequently, it is not only important that a CPS exchanges messages in a correct order, but at which point in time the exchange happens and how long the internal actions as well as the message delay take. Therefore, a software engineer must be able to define and verify real-time constraints (e.g., deadlines and assumed message delay) for the coordination.

Due to these stated properties, the software development of a CPS is highly complex. A methodology that aims to alleviate the complexity of the software while expressing domain concepts effectively is model-driven engineering (MDE) [Sch06]. According to MDE, a software engineer shall use a domain-specific language (DSL) [VDW12] to specify a model

¹“[O]bjectives are the required, desired and/or undesired properties of the system. They are pre-determined during system development, but their desired characteristics are defined only during operation.” [GRSS14]

of the software instead of implementing it directly by writing the target code. This shall ease the specification as it abstracts from technical and implementation-specific details by using domain-specific language constructs. Moreover, if the specified models have a defined semantics, it is possible to find (safety-critical) errors already at design time by verifying the models. In particular, formal verification techniques such as model checking [BK08] exist that—in contrast to traditional testing—automatically verify *all* possible execution traces whether a certain (real-time) property always holds (e.g., the absence of an error). Moreover, if a property is not fulfilled, model checkers are able to generate counterexamples that show the execution steps a behavioral specification may execute to violate the property. Therefore, counterexamples support the identification of the root cause, which is an essential information for the software engineer to correct his design model.

A study by Davis et al. [DCC+13] shows that the application of formal methods like model checking is hindered by several barriers in practice. The most important barriers are (1) that many software engineers have no (deep) experience in formal methods, (2) the user-interfaces of existing tools are too complex and error-prone, and (3) that many project managements do not encourage or (financially) enable the use of formal methods. In this thesis, we address the first and second barrier. In particular, our goal is that the software engineer for CPSs only has to use its DSL and may execute the model checking on its domain-specific models directly instead of using a model checker, which has its own languages for the design model, the properties, and the counterexamples. The reason for this is that DSLs provide—compared to the model checker’s input language—higher-level modeling constructs (e.g., hierarchical state machines) as well as domain-specific language concepts (e.g., asynchronous message exchange that require incoming message buffers). Thus, DSL models are more concise and better to understand for the software engineer compared to the models of the model checker.

Though, a DSL definition often underlies changes, i.e., its evolution is considered fast-paced [VBD+13, p. 31]. Therefore, a DSL typically does not provide its own model checking. The reason behind is that this would require high maintenance effort because developing a correct *and* efficient model checker is a very complex task, which requires tremendous skills especially in theoretical computer science. As a consequence, state of the art approaches like Dragomir et al. [DOP15] typically translate the DSL model to the input language of an external model checker that is developed independently. However, this does not hide the details of the model checking because the software engineer still has to define verification properties and inspect the counterexamples at the level of the model checker. Consequently, the software engineer currently has to learn the model checker’s input and output languages as well as the encoding of the domain concepts. Especially the latter one easily becomes a complex task when extensive concepts like asynchronous communication are encoded.

We solve these problems by applying the concept of domain-specific model checking (DSMC, [VDW12]), which hides the model checking level completely. DSMC has been successfully applied to the domains of software implementation (i.e., code), hardware design, and biological systems but not to the coordination design of CPSs. As a prerequisite, DSMC demands that the software engineer specifies not only the design model but also the verification properties at the DSL level using a so-called domain-specific verification property language (DSVPL) [VDW12]. As an advantage, a DSVPL may—in contrast to the verification property language of a model checker—provide domain-specific model elements that are necessary to express relevant domain-properties in a compact manner, e.g., concerning the state of an incoming message buffer. As another advantage, a DSVPL may be directly integrated into

the DSL, i.e., it enables references to the design model and, therefore, ensures consistency between the design model and the properties. If the design model and the DSVPL properties are specified, then the actual DSMC may happen. According to [VDW12], it consists of the following process steps:

1. The software engineer starts the DSMC. The inputs are primarily the design model and the properties stated in the DSVPL. Then, the following steps happen automatically:
 - a) A so-called forward translation translates the DSL models (the design and the properties) to the model checker's input languages.
 - b) The model checker verifies the properties.
 - c) A so-called back-translation translates the results (the truth value of each property) and counterexamples back to the DSL.
2. The software engineer may inspect the domain-specific counterexample.

Another hindrance for applying model checking in practice is that model checking a network of CPSs as a whole leads to the state-explosion-problem [CGP00]. This problem describes that the number of reachable runtime states is typically exponential in the size of the model. Consequently, model checking does not scale and is not applicable to verify complex models. This is especially relevant for concurrent systems like a network of CPSs as the number of reachable runtime states grows exponentially in both the number of concurrent processes and the number of states per process [CKNZ12]. Several approaches exist to tackle the state-explosion problem [BCC98]. One popular approach is the assume/guarantee principle [Pnu85]. By using this principle, a software engineer can verify parts of the system separately of each other. However, each part may depend on other parts of the system, e.g., due to message exchange. Thus, the software engineer has to additionally define assumptions for each separate verification of a part concerning the behavior of the other parts. These assumptions enable to deduce the correctness of the entire system as long as the other parts guarantee these assumptions (obviously, the guarantees need to be verified as well).

Related work already enable modeling and testing the software of CPSs, e.g., the standard MARTE [MARTE]. However, they all miss a scalable formal verification approach and do not enable DSMC. In contrast, various approaches exist that enable scalable formal verification [PT98; DLL+12; DOP15], e.g., by applying the assume/guarantee principle. However, these approaches either do not support DSLs but only low-level model checking languages like timed automata [BK08] or they do not enable DSMC. Therefore, they do not hide the model checking formalisms from the software engineer nor do they provide (sufficient) higher-level or domain-specific modeling constructs.

Having the possibility to formally verify the software of a CPS using model checking does not automatically result in meaningful and correct software models as the software engineer still has to design the model and has to choose the properties that shall be verified. One possibility to ease this task is to provide software design patterns [GHJV95] such that the software engineer can benefit from existent experience. Related work like Ramirez and Cheng [RC10] have defined formal design patterns that consider the need for model checking by providing formal models and a set of useful verification properties. However, formal design patterns for the asynchronous message exchange of CPSs do not exist yet.

A software development method that focuses on the real-time coordination design of CPSs—including scalable model checking—is MECHATRONICUML [*BDG+14a; *DPP+16]. It

is currently under development at Fraunhofer IEM and Paderborn University. MECHATRONICUML follows MDE by defining a component-based DSL for CPSs. Among others, MECHATRONICUML enables to specify verifiable coordination protocols that define a contract [HHG90] (resp. assumptions and guarantees) between interacting roles, whereby each role represents a component². We call these protocols *Real-Time Coordination Protocols* (RTCPs). RTCPs define the allowed message exchange between the components for coordinating their actions while considering real-time constraints. Therefore, RTCPs contribute to the safety of the network of CPSs as long as all components comply to them. RTCPs are tailored to the domain of CPSs as they provide domain-specific model elements, e.g., for defining real-time constraints and an asynchronous message exchange. A software engineer defines their behavior via extended state machines, which we call Real-Time Statecharts (RTSCs). We distinguish two forms of RTCPs: one-to-one and one-to-many. A one-to-one RTCP defines a message-based coordination between exactly two roles. Instead, a one-to-many RTCP defines a message-based coordination between one role on the one hand and multiple roles of the same type on the other hand.

The main purpose of RTCPs is to enable a scalable model checking of the design model based on the assume/guarantee principle. MECHATRONICUML's concept to realize the scalable model checking is called the compositional verification approach [GTB+03; *HBDS15]. It enables the software engineer to separately verify all components via model checking under the assumption that the message exchange with other components always respects the specified RTCPs. However, the software engineer has to execute two additional steps to guarantee this assumption: Among others, he³ has to prove that each RTCP is correct w.r.t. safety-critical verification properties by applying model checking, e.g., using the model checker UPPAAL [BLL+96].

1.1 Running Example: Coordinated Overtaking

In this section, we introduce the running example of this thesis, which we call *Coordinated Overtaking*. The example will help us to explain the upcoming contents of this section: the problems that we tackle (see Section 1.2) and the contributions that we provide within this thesis (see Section 1.3).

Our example is about autonomous cars that apply an overtaking maneuver. The overtaking shall be coordinated via an exchange of messages between the systems to increase the safety of the cars. We illustrate the example in Figure 1.1. Here, three cars—red, yellow, and blue—drive on a two-lane street. The red and yellow car drive on the same lane. In contrary, the blue car drives on the other lane in the opposite direction with a distance of one kilometer. Moreover, the red car drives faster than the yellow one and wants to overtake it before blue arrives to avoid a long waiting time.

Before the overtaking starts, the red car must be sure that the overtaking can be successfully completed. Otherwise, a collision is likely to happen and can cause severe damage and even loss of human life. Thus, red's decision whether it shall overtake before the blue car arrives is highly safety-critical. Therefore, the red car should gather the relevant information about

²A component represents either a CPS or a part of the CPS.

³Throughout the thesis, gender-specific terms may be used in order to ease the text flow. Whenever a gender-specific term is used, it should be understood as referring to both genders, unless explicitly stated.

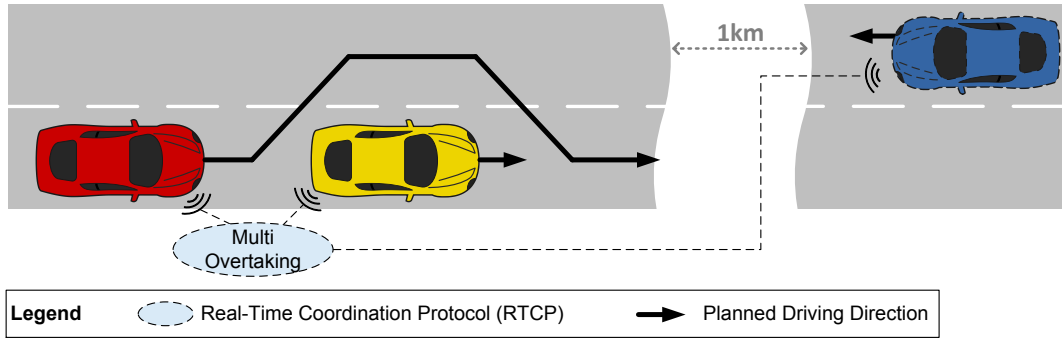


Figure 1.1: Example Scenario: Coordinated Overtaking

the other cars as well as relevant guarantees from them to come up with the right decision. For doing this, all systems are able to communicate with each other by exchanging messages. The maximum message delay between the cars is at most one second. Moreover, the message exchange is not reliable, i.e., messages may be lost during transmission.

The relevant information for the red car are the current distances to the other cars and their current speed. Based on these information, red can compute whether the overtaking is theoretically possible. However, the overtaking is still unsafe as the other cars could increase their speed during the overtaking. This would reduce the available time for the overtaking and, therefore, increase the risk for a crash. Consequently, red also requires a guarantee from the other cars that they do not accelerate while the overtaking happens to ensure a safe overtaking.

In our example, the software engineer defines the one-to-many RTCP `MultiOvertaking` to specify this coordination task. It specifies a car-to-car communication between up to three cars. The role `Overtaker` represents the red car that wants to execute an overtaking maneuver and defines a behavior that the `Overtaker` always ensures. The role `Overtakee` represents the cars that the role `Overtaker` may want to overtake or that drive on the other lane in the direction of the overtaker, i.e., the oncoming traffic. In our example, these are the yellow and the blue car.⁴ Concerning the RTCP's behavior, the software engineer must be able to define the request and reply for starting an overtaking, the actions of the cars during the overtaking, and the information by the overtaker that the maneuver is completed. As described before, one of many safety-critical situations may occur if the RTCP `MultiOvertaking` would allow a situation where red may overtake but the other cars can still accelerate as they assume that no overtaking happens. Therefore, a software engineer must be able to state the following requirement as a formal verification property: "The other cars (blue and yellow) cannot accelerate as long as the red car overtakes". Furthermore, the software engineer must be able to formally verify using DSMC whether the RTCP fulfills this property.

⁴In RTCP `MultiOvertaking`, we combine the set of cars that the overtaker is going to overtake and the cars that are oncoming traffic due to the contents of the protocol. Variants of this coordination scenario may require two one-to-many RTCPs: one between the overtaker and the cars that it wants to overtake and another one for the communication between the overtaker and oncoming traffic.

1.2 Problem Definition

In this section, we outline four problems that especially apply to the design and formal verification of MECHATRONICUML's RTCPs but also apply for verifiable coordination protocols for CPSs in general. For each problem, we argue why it leads to an error-prone development that increases the development time and—more importantly—may cause safety-critical faults in the resulting artifacts. All problems are currently not sufficiently solved by MECHATRONICUML and other related approaches.

1.2.1 Specification of Real-Time Coordination Protocols

RTCPs were initially defined by Giese et al. [GTB+03; BGHS04] and later on revised by Hirsch [Hir08]. Their definitions have three drawbacks concerning the syntax and semantics that lead to an error-prone development and ambiguous (and therefore unsafe) specifications. In the following, we explain these drawbacks in detail.

First, RTCPs have a wrong level of abstraction with respect to incoming message buffers: A software engineer must be able to define the buffers of an RTCP and properties like the buffer size. Otherwise, the model checking results are meaningless (e.g., the model checker cannot know for which buffer size it shall verify the behavior). For doing this, the software engineer currently has to learn complex modeling idioms to define the buffers, their behavior (e.g., adding a message to the buffer), and their properties within the roles RTSCs. This is an incorrect level of abstraction that leads to an error-prone modeling and to models that are hard to understand. For example, both roles of RTCP MultiOvertaking require an incoming message buffer. Therefore, the software engineer has to add additional states and transitions to both role RTSCs that encode—among others—the adding and removing of a message, the buffer size, and the behavior if the buffer is full and a new message arrives.

Second, the modeling language for one-to-many RTCPs misses model elements to define reoccurring one-to-many communication dependencies like a multicast (i.e., the message is sent to a specific set of receivers) directly. Instead, the software engineer has to learn complex modeling idioms to define such dependencies within the roles RTSCs. Again, this incorrect level of abstraction leads to an error-prone modeling and to models that are harder to understand. For example, the role Overtaker of RTCP MultiOvertaking shall send the overtaking request to all roles of type Overtakee, i.e., it shall send a multicast of message request. However, the software engineer cannot specify that a message shall be sent as a multicast in a declarative way but has to model it using additional states and transitions. In summary, the RTCP MultiOvertaking uses four one-to-many communication dependencies that require 16 additional states and 19 additional transitions even though the dependencies are based on reoccurring modeling idioms.

Third, the current specification of RTCPs as well as the semantics specification is ambiguous as the assumptions on the message transfer, i.e., the so-called quality of service (QoS) assumptions, are currently underspecified. As the QoS assumptions influence the model checking results, design errors may not be found and can ultimately result in unsafe systems. In particular, we have to distinguish between variable and fixed QoS assumptions. Variable QoS assumptions may vary between RTCPs. Therefore, the software engineer must be able to specify them explicitly. Fixed QoS assumptions do not vary between RTCPs. Therefore, the software engineer does not need to specify them but has to know that they exist and what

they mean. For example, a variable QoS assumption of RTCP MultiOvertaking is that the connector is unreliable (see Section 1.1). Therefore, the model checker needs to consider that each message being sent may be lost. An example for a fixed QoS assumption is that we assume that the middleware always enqueues each message at most once into the incoming buffer. If the software engineer does not know this, then he might select a middleware that does not delete duplicate messages. This would make the model checking result meaningless.

There already exist related approaches that enable to specify verifiable coordination protocols via asynchronous messages. Concerning our first drawback, approaches like MARTE [MARTE] exist that enable an explicit specification of message buffers. However, none of the approaches solves this drawback sufficiently. For example, if the buffer is full and a new message arrives, then the pre-defined buffers of MARTE always delete the oldest message in buffer. In contrary, we identified use cases where—for example—the incoming message shall be deleted. A software engineer can only specify this in MARTE if he defines this using an additional state machine. Concerning our second drawback, approaches like Hadj Kacem et al. [HHD09] already enable to specify one to-many communication dependencies on an abstract level. However, they do not cover all our identified one-to-many communication dependencies. For example, Hadj Kacem et al. only enable to specify a multicast but we also identified the need for a so-called multireceive (e.g., the overtaker expects an answer from all overtakes). Concerning our third drawback, approaches like Reo [ABRS04] enable to specify the assumed message delay. However, they provide no other QoS assumptions, e.g., whether messages may get lost during transmission. Furthermore, we identified no related work that states its implicit QoS assumptions for their verifiable coordination protocols.

Consequently, we have to adapt the syntax of RTCPs in order to express the relevant domain concepts on the correct level of abstraction while providing a complete semantics specification.

1.2.2 Specification of Domain-Specific Verification Properties for CPS

Currently, there exists no domain-specific verification property language (DSVPL) that a software engineer may use to define all relevant properties for an RTCP. Therefore, the software engineer has to state the properties with the provided property language of the model checker. This is in our case the property language of the model checker UPPAAL, which we call UTCTL. As a consequence, two drawbacks arise: (1) The software engineer has to learn the property language of the model checker (here: UTCTL) that typically does not enable to express domain-specific properties in a compact manner, e.g., that a message is in transit. Moreover, he also has to learn the language for specifying the design model (here: UPPAAL timed automata) as the property language refers to it. Typically, a software engineer is not familiar with the input language of a model checker. Therefore, its development task is getting more complex. (2) The software engineer has to understand the transformation from RTCPs to the model checker's design model (e.g., how a message in transit is encoded) to be able to state a property that has the same semantics as the informal requirement for the RTCP. Due to the huge differences of RTCPs and UPPAAL, understanding the transformation and specifying correct properties is a complex and error-prone task. Though, in our domain of safety-critical systems, an incorrect specification of a property must not happen. Otherwise, safety-critical errors may not be revealed while model checking. A DSVPL would avoid both drawbacks

and would enable a DSMC for RTCPs where the model checking is completely hidden to the software engineer.

Concerning our overtaking example, we present in the following a selection of informal verification properties that need to be expressible for RTCP MultiOvertaking in order to verify that the RTCP is meaningful and safe: (P1) “As long as the overtaker is overtaking, the overtakes will not accelerate any further.” (P2) “The overtaking must not take longer than 20 seconds.” (P3) “No message will ever be discarded in any buffer.” (P4) “All transitions may be fired.” (P5) “Each message may be in transit.” (P6) “There is no buffer that is always empty.” This list of verification properties shows that we require a DSVPL that has a high expressiveness. For example, the DSVPL has to support real-time properties like P2, safety properties (something bad may never happen) like P1–P3, liveness properties like P4–P6, and domain-specific concepts like P5–P6.

Related work already defines DSVPLs for their domain, e.g., dos Santos et al. [dSWP11] for the domain of railway tracks. However, a DSVPL for the domain of CPSs that supports real-time constraints as well as domain-specific concepts like message buffers does not exist yet. Moreover, Flake and Mueller [FM02] as well as Cengarle and Knapp [CK02] provide DSVPLs that are variants of OCL [OCL] to enable the specification of verification properties for real-time systems. However, their DSVPLs do not provide model elements to support domain-specific concepts. In addition, there already exist several DSVPLs in MECHATRONICUML, e.g., by Stallmann [Sta08]. However, Stallmann’s DSVPL does not need to focus on real-time requirements as well as on asynchronous communication concepts like messages and message buffers.

Consequently, we need a DSVPL for RTCPs that fits to the domain of CPSs as well as to the model checker in use.

1.2.3 Domain-Specific Model Checking (DSMC) of RTCP

Currently, there exists no complete domain-specific model checking (DSMC) for coordination protocols of CPSs like RTCPs of MECHATRONICUML. Therefore, the details of the model checking (the translations as well as the input and output languages of the model checker) are not hidden to the software engineer. This results in a time-consuming and error-prone task even if the software engineer has significant experience in model checking.

In a previous work of our research group, Giese, Burmester, and Hirsch [GB03; Hir04; BGHS04] already define a partial DSMC between RTCPs and UPPAAL. Concerning the forward-translation, they already transform several aspects, e.g., the transformation from hierarchical to flat state machines. However, important domain-concepts like asynchronous message exchange and message delay were not supported yet. Therefore, the software engineer currently has to adapt the resulting UPPAAL model manually if he wants to consider these aspects. Moreover, as there exists no DSVPL for RTCPs yet (see Section 1.2.2), the software engineer would have to translate the properties of the DSVPL to the property language of UPPAAL manually. Both tasks are complex and error-prone and do not hide the underlying model checker. Concerning the back-translation, Hirsch et al. already translate the verification result back, i.e., whether a property is fulfilled or not. However, counterexamples are not translated back. Therefore, the software engineer has to understand the counterexample at the level of UPPAAL in order to identify the root cause of the problem. Due to the huge

differences of UPPAAL and RTCPs, this is a complex and time-consuming task that an automatic back-translation would prevent.

Concerning our overtaking scenario, if a software engineer uses the existing model checking approach for RTCP Overtaking, then he currently has to apply the following steps: (1) He has to execute the forward translation that translates the RTCP Overtaking to a valid UPPAAL model. (2) Then, he has to adapt the UPPAAL model manually. For example, he has to change the model such that the maximum message delay is up to one second instead of zero and he has to exchange the synchronous communication between the cars by an asynchronous communication. (3) The software engineer has to manually define all verification properties in UTCTL. (4) If UPPAAL produces a counterexample, then the software engineer has to study it at the level of UPPAAL. If he succeeds in finding the root cause in the UPPAAL model, then the software engineer has to use this knowledge to identify the root cause in the RTCP Overtaking. In contrast to these complex steps, having a DSMC would result in 3 steps that are significantly easier: (1) The software engineer specifies the verification properties using a DSVPL. (2) Afterwards, he starts the DSMC that automatically translates the RTCP and the properties to UPPAAL. Then, the DSMC executes the model checking. If a counterexample exists, the DSMC automatically translates it back. (3) The software engineer can study the counterexample model, which references the model elements of RTCP Overtaking, at the level of MECHATRONICUML.

Several related approaches exist that enable DSMC (including back-translating counterexamples) for a specific DSL and a specific model checker [BBK+15; SAB10; ZCP13]. However, no approach exists that enables DSMC for the software development of CPSs where asynchronous communication needs to be addressed. In our research group, several translations from MECHATRONICUML to other formal verification tools exist, e.g., to the model checkers Phaver [Dor08] and Raven [Ste05]. However, they do not support the forward translation of asynchronous message exchange concepts nor of a DSVPL. In addition, they do not enable the back-translation of counterexamples.

Consequently, we need a complete DSMC approach for RTCPs that supports their complete syntax and adheres to the semantics definition.

1.2.4 Reusing Experience for the RTCP Design

Specifying an RTCP from scratch is a complex task and, therefore, time-consuming and error-prone. The reason for this complexity is that the software engineer has to ensure that the RTCP is free of faults despite aspects like hard real-time constraints, message delay, and the possibility of message loss. A software engineer is able to identify faults using DSMC (especially if counterexamples are provided). However, removing the faults may still be a non-trivial and time-consuming task as model checking does not directly provide the root cause of the fault nor does it suggest how to correct it. In addition, choosing the wrong verification properties may not reveal all errors. While developing and studying existing RTCPs for various CPSs, we identified that the message-based coordination is based on recurring use-cases. This applies for the coordination between CPSs, but also for the message-based coordination between components within one CPS. Therefore, the experience to design an RTCP for a new application such that it solves a given design problem while respecting safety-critical verification properties is hidden within the already existing application-specific RTCPs. A software engineer with low experience struggles when trying to grasp the intention

of an already existing RTCP due to its application-specific details. Moreover, such a software engineer is typically not able to extract this experience efficiently from existing RTCPs. As a consequence, he wastes a lot of time to select and adapt a fitting RTCP to its needs. Therefore, the software engineer needs support to reuse the experience of designing RTCPs (e.g., concerning the behavior definition and the verification properties). However, such a support is currently missing.

For example, a software engineer does not need to define the one-to-one RTCP Overtaking including its verification properties from scratch as a similar RTCP called ConvoyCoordination already exists [GTB+03]. The RTCP ConvoyCoordination enables to build (and break) a convoy of two autonomous railways systems in a safe manner. Among others, the RTCP defines that while in convoy mode, the front driving railway always informs the rear driving railway with a message before it brakes. On a first look, the RTCP ConvoyCoordination does not have much in common with an overtaking of cars. However, it encapsulates the experience to enable a role to acquire an assurance of another role (here: the braking message during convoy) despite message loss and message delay. RTCP Overtaking also has to acquire an assurance: if the overtakee accepts the overtaking, then it will not accelerate during the overtaking. Furthermore, RTCP ConvoyCoordination defines an important verification property concerning this assurance. This property is reusable as well for RTCP Overtaking. In summary, by extracting the specification experience from RTCP ConvoyCoordination, we can support the software engineer to model the RTCP Overtaking and define useful verification properties.

In related work, an approved approach for overcoming these problems is the usage of design patterns. They “describe a commonly recurring structure of communicating components that solve a general design problem within a particular context” [BMR+96]. Various design patterns exist that partially help to develop RTCPs, e.g., [GHJV95; BMR+96; Dou99; Dou02]. However, they do not focus on the design of asynchronous message exchange. Moreover, they do not take the aspect of model checking into account: First, they lack a formal description of their models. Second, they do not define relevant verification properties that the resulting model should fulfill. As a consequence, the work of the software engineer is still complex and error-prone and existing experience-based knowledge is not sufficiently reused. In contrast, other approaches like Ramirez and Cheng [RC10] provide formal design patterns including recommended verification properties. However, they do not focus on the domain of CPSs.

Consequently, we need an approach that enables to reuse the experience-based knowledge of existing RTCPs, e.g., a pattern-based approach for the domain of CPSs.

1.3 Contribution

The contribution of this thesis is a model-driven approach for specifying and verifying the real-time coordination of CPSs. In particular, this thesis provides four contributions:

- C1** A DSL with appropriate syntax and semantics for specifying verifiable coordination protocols of CPSs,
- C2** a DSVPL for specifying the verification properties for a verifiable coordination protocol of CPSs while respecting the capabilities of the selected model checker,
- C3** a model-driven DSMC approach for verifiable coordination protocols of CPSs, and

C4 formal design patterns for verifiable coordination protocols of CPSs that consider the need for formal verification.

Concerning the scope of this thesis, our contributions only focus on the MECHATRONICUML method. Therefore, we use RTCPs as a representative for verifiable coordination protocols of CPSs. Moreover, we do not contribute a formal semantics definition of RTCPs nor do we provide a formal proof that our DSMC always preserves the semantics while translating the models to the model checker and while translating the counterexample back to the DSL. Even though the domain of CPSs has to deal with continuous behavior and security concerns, both topics are not in the scope of this thesis. In addition, within this thesis, we do not focus on the usability of our concepts.

In the following, we explain our contributions in detail and define how we embed them within MECHATRONICUML’s development process. In Figure 1.2, we show an excerpt of this process that focuses on the design phase. This phase is split into two parts—the platform-independent and the platform-dependent design—to handle the complexity of the software development. Our contributions focus on the second step of the platform-independent design as it deals with designing (i.e., specifying and verifying) RTCPs. Concerning this step, we improve its development process, its modeling language, and its formal verification approach. The MECHATRONICUML Tool Suite [*DGB+14] contains all our implementations and serves for evaluating our concepts.

In particular, the new process of Step 2 (Design RTCPs), which provides solutions for all stated problems of Section 1.2, is as follows: At first, the software engineer may now optionally choose from a catalog of nine formal design patterns for RTCPs in the new **Step 2.1**. As a consequence, we facilitate the reuse of existing experience-based knowledge of an RTCP and, therefore, improve the efficiency of the software engineer. We call these design patterns Real-Time Coordination Patterns. Each design pattern provides an application-independent solution for a recurring coordination problem in the domain of CPSs. Moreover, each design pattern provides, aside from textual descriptions, formal models and formal verification properties that facilitate a formal verification. The definition of Real-Time Coordination Patterns, the pattern catalog, a process for selecting and adapting a pattern to an RTCP, and a process for abstracting an RTCP to a pattern are our contribution C4. As an example, we show in the lower left box 2.1 of Figure 1.2 the pattern Acquire Assurance. It enables to acquire an assurance that is necessary to execute a safety-critical task. As depicted, the pattern defines a formal model that is similar to a model of an RTCP, formal verification properties, and textual descriptions that shall facilitate the pattern selection. Concerning our overtaking scenario, the software engineer may select this pattern as the overtaker requires that the overtaken assures that it will not accelerate any further as long as the overtaker executes the overtaking.

Afterwards, in **Step 2.2**, the software engineer has to specify the RTCP based on the given requirements and—if selected—on the selected Real-Time Coordination Pattern. Concerning this step, we redefine the syntax and semantics of RTCPs to remove the current drawbacks. This is our contribution C1. In particular, we introduce two modeling elements: (1) incoming message buffers and (2) communication schemata for one-to-many RTCPs (short: one-to-many communication schemata) to express reoccurring coordination dependencies like a multicast explicitly. Both syntax changes make the specification of an RTCP more compact as the behavior does no longer need to be specified using additional states and transitions. Furthermore, we define a set of RTCP-independent and RTCP-dependent QoS assumptions

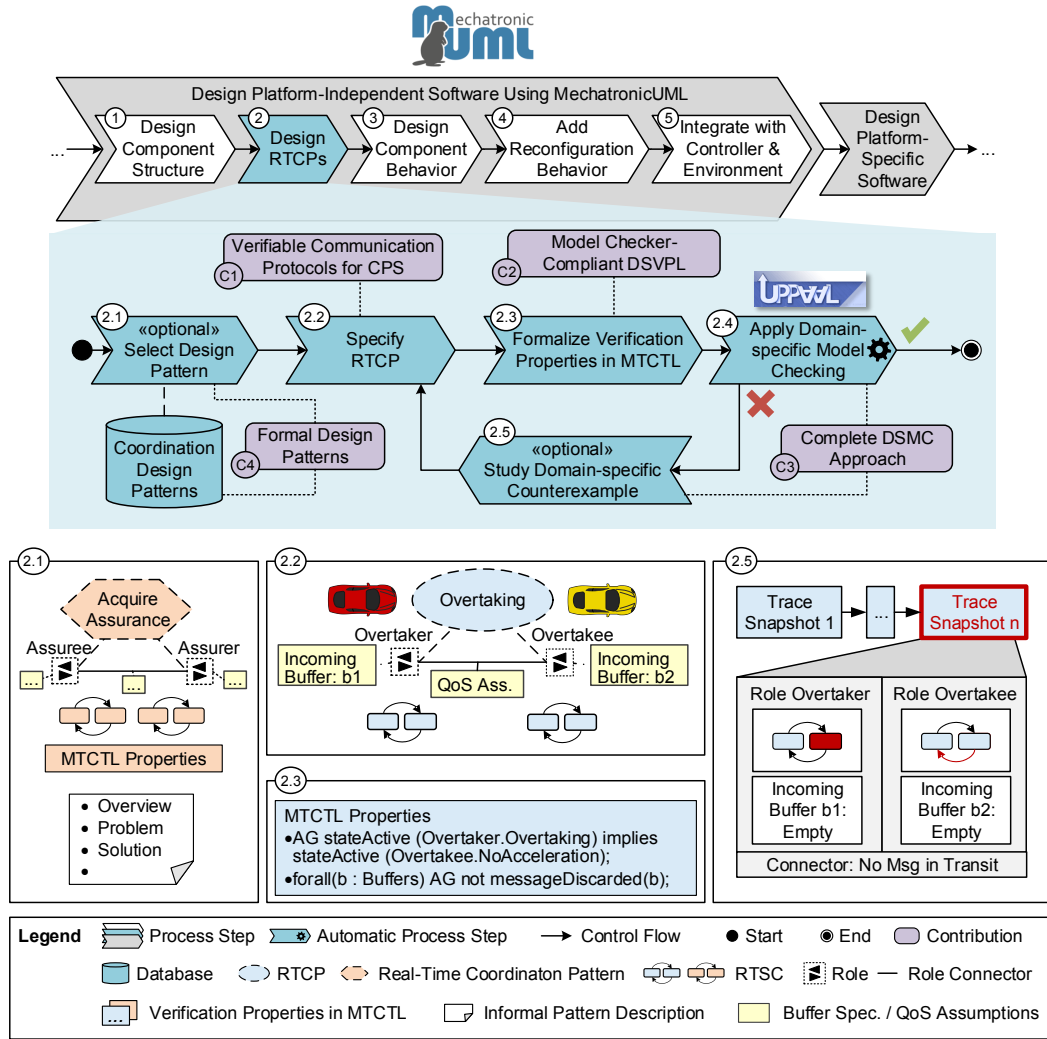


Figure 1.2: Enhanced Design Process for the Software Development of CPS

that are relevant for the formal verification. For example, as it is shown in box 2.2 of Figure 1.2, the software engineer may define a one-to-one RTCP Overtaking to support the scenario where only one car needs to be overtaken and no oncoming traffic exists. The RTCP is based on the Real-Time Coordination Pattern Acquire Assurance and defines the roles Overtaker and Overtakee. Both roles define an incoming message buffer. In addition, the role connector defines RTCP-dependent QoS assumptions, e.g., a maximum message delay of one second.

In **Step 2.3**, the software engineer has to specify formal verification properties for RTCPs via a new textual DSVPL called *MTCTL*—a domain-specific variant of TCTL [ACD93]. This DSVPL is our contribution C2. *MTCTL* is designed for verifying RTCPs via UPPAAL: On the one hand, it enables to reference all model elements of an RTCP that are relevant for the DSMC and provides keywords for domain-specific properties like the buffer size. On the other hand, it covers as much as possible from UTCTL’s language but does not allow to specify properties, which are not verifiable by UPPAAL. In the lower part of Figure 1.2 in box 2.3, we show two exemplary *MTCTL* properties for the RTCP Overtaking. The upper property

defines that it must always be the case that if the state *Overtaking* of role *Overtaker* is active, then the state *NoAcceleration* of role *Overtakee* must be active as well. The lower property defines that all incoming message buffers may never discard any message (i.e., it will never be the case that the buffer is already full but a new message arrives).

Then, in **Step 2.4**, the software engineer has to formally verify the RTCP via DSMC with the defined set of MTCTL properties using UPPAAL. Having such a DSMC, the software engineer does not need to have experience in UPPAAL, e.g., how he can encode domain concepts like asynchronous communication that UPPAAL does not support. If the model checking result is that a safety property fails, then the DSMC provides a domain-specific counterexample that the software engineer may study in **Step 2.5** to identify the root cause of the error. The DSMC of RTCPs using the model checker UPPAAL is our contribution C3. In particular, our translation from MECHATRONICUML to UPPAAL encodes all MECHATRONICUML-specific concepts (e.g., message buffer and QoS assumptions) of an RTCP into UPPAAL's input language and transforms MTCTL properties into UTCTL. Our back-translation produces a counterexample model at the level of MECHATRONICUML that references the model elements of the RTCPs. For doing this, the back-translation uses the traceability links that are automatically generated by the transformation engine while the forward translation is executing. Thus, the back-translation does not need to encode the concepts of the forward translation again. For example, the software engineer has to formally verify the RTCP *Overtaking* with all its MTCTL properties. We indicate a possible counterexample trace in box 2.5 at the lower right of Figure 1.2. As depicted, we divide the trace like UPPAAL into several system states that we call snapshots. A new snapshot occurs if a transition of a timed automaton fires or if time passes. Each snapshot provides several information to the software engineer, e.g., which states are currently active as well as the status of each buffer and the connector.

In this thesis, we evaluate several parts of our contributions using various case studies: Concerning our one-to-many communication schemata, we show that (1) they are sufficient to specify existing examples and (2) that they enable to model the behavior in a more compact manner than the original specification language. Concerning MTCTL, we show that (1) we are able to state a majority (ca. 90%) of realistic informal requirements as formal MTCTL properties and (2) that we preserve the semantics of the MTCTL properties when translating them to valid English sentences. Concerning our DSMC, we show for 16 RTCPs that (1) it correctly translates them to UPPAAL (including their MTCTL properties), (2) that it correctly translates results and (counterexample) traces produced by UPPAAL back to MECHATRONICUML, (3) that one minute is in most cases sufficient to execute the complete DSMC, and (4) that our DSL's encapsulate domain-specific aspects more effectively than UPPAAL. Concerning our formal design patterns, we show that (1) they have a high reusability rate within two different applications and (2) that the RTCPs that we adapted from a pattern are still similar to the pattern and do not change their intent.

1.4 Thesis Structure

The remainder of this thesis is structured as shown in Figure 1.3. In Chapter 2, we define the foundations for this thesis by introducing timed model checking and MECHATRONICUML. Afterwards, in Chapter 3, we improve the syntax and semantics of RTCPs: we

define the structural definition, the modeling assumptions, the behavior definition, and the application to the component model. Next, we introduce MTCTL in Chapter 4 for specifying domain-specific verification properties for RTCPs. Furthermore, we define so-called default verification properties for MTCTL that each RTCP should always fulfill and a translation to English sentences, which shall facilitate the understandability for domain experts that have no model checking experience. Then, we define our DSMC approach for RTCPs in Chapter 5 including the translation to UPPAAL, the automation of UPPAAL, and the back-translation to MECHATRONICUML. In Chapter 6, we propose formal design patterns for RTCPs called Real-Time Coordination Patterns. Moreover, we present processes to abstract an RTCP to a pattern and to adapt a pattern to an RTCP. Finally, in Chapter 7, we conclude the thesis with a summary and give an outlook for future work.

The chapters 3 to 6 present the main contributions of this thesis. All four of them start with two sections: the contributions of the chapter and the involved stakeholders and the requirements for each particular solution. Moreover, they all end with the same five sections: they explain the implementation, evaluate parts of the contribution, talk about limitations, discuss related work, and give a summary.

The appendices give additional information. Appendix A presents the remaining one-to-many communication schemata that we do not present in detail in Chapter 3. Then, Appendix B presents the catalog of Real-Time Coordination Patterns as we only present the pattern *Acquire Assurance* in Chapter 6. Next, Appendix C gives additional information concerning our evaluations, Appendix D shows details of our running example (the RTCP *Overtaking*), and Appendix E explains excerpts of our implementations like the defined metamodels and grammars. Appendix F defines all our abbreviations. Finally, I explain my contributions to each of my publications in Appendix G.

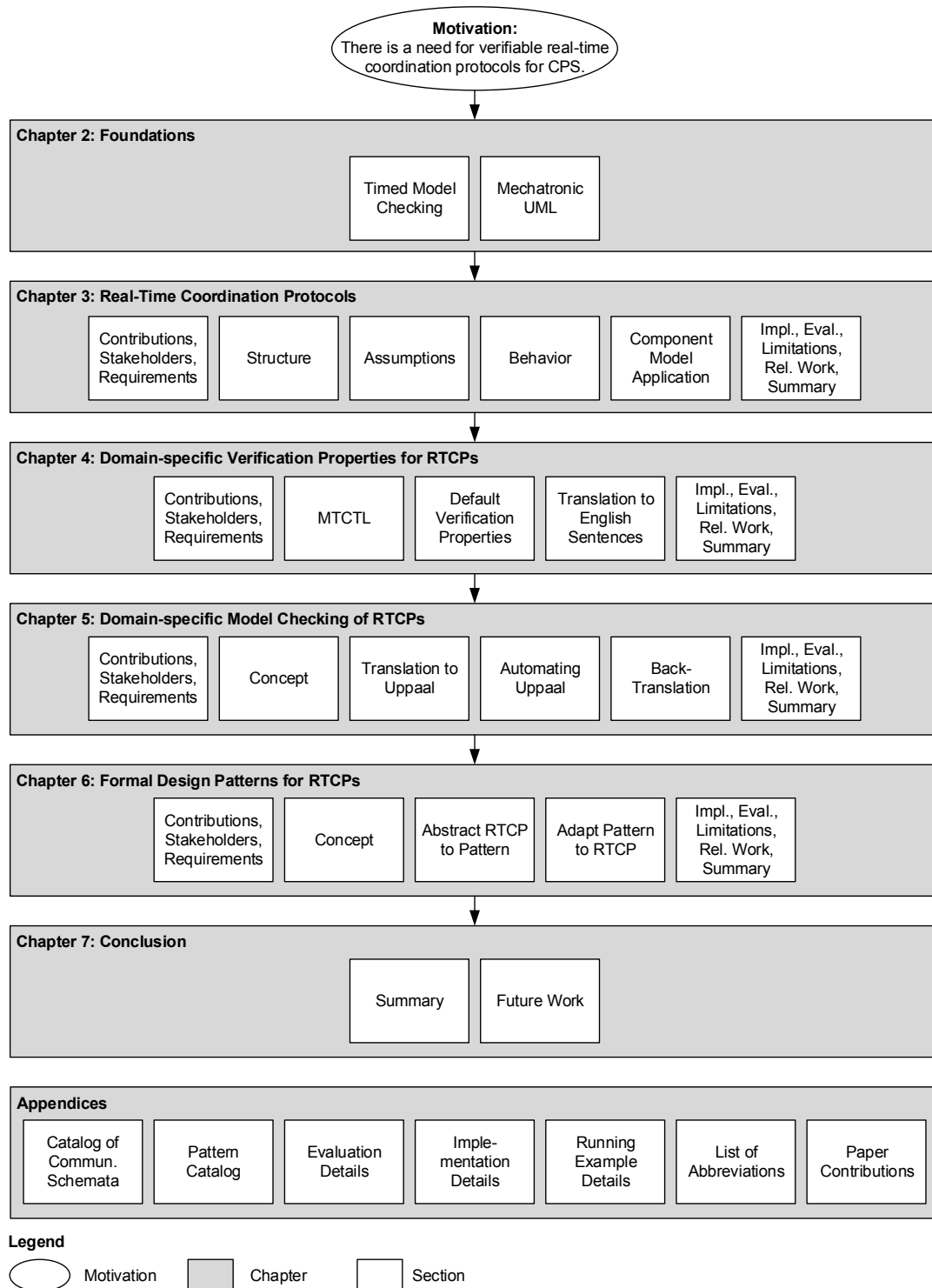


Figure 1.3: Thesis Structure

FOUNDATIONS

In this chapter, we present the foundations that are necessary to understand the concepts of this thesis. We expect that the reader of this thesis has at least basic knowledge of computer science and model-driven software development. Therefore, we only introduce topics that are beyond this knowledge. In particular, we first explain timed model checking in Section 2.1. Afterwards, we introduce the necessary concepts of MECHATRONICUML that are required for this thesis in Section 2.2.

2.1 Timed Model Checking

Model checking [BK08; CGP00] is a formal verification technique that enables an automatic mathematical proof, which verifies whether a given software model fulfills a given set of formal properties. Therefore, it enables to show the absence of errors in the software model concerning the stated properties while traditional testing only enables to show their presence. This is a clear advantage of model checking, especially when developing software for safety-critical systems. Timed model checking [ACD93] additionally supports software models that contain real-time constraints, e.g., invariants or a message delay. In this thesis, we use timed model checking for formally verifying coordination protocols of CPSs. In particular, we use the timed model checker UPPAAL [BLL+96].

In timed model checking, timed automata represent the software model under verification. We introduce them in Section 2.1.1. Formal verification properties are stated in the timed computation tree logic (TCTL), which we describe in Section 2.1.2. The model checking process can happen as soon as the timed automata and the verification properties are defined. We describe the basics of this process in Section 2.1.3. At last, we briefly describe the tooling that UPPAAL provides in Section 2.1.4.

2.1.1 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).” [Hei15] For the model checker UPPAAL [BLL+96], Bengtsson and Wang [BY04] define timed safety automata [HNSY94]. In this thesis, we use the updated definition of timed safety automata from Behrmann et

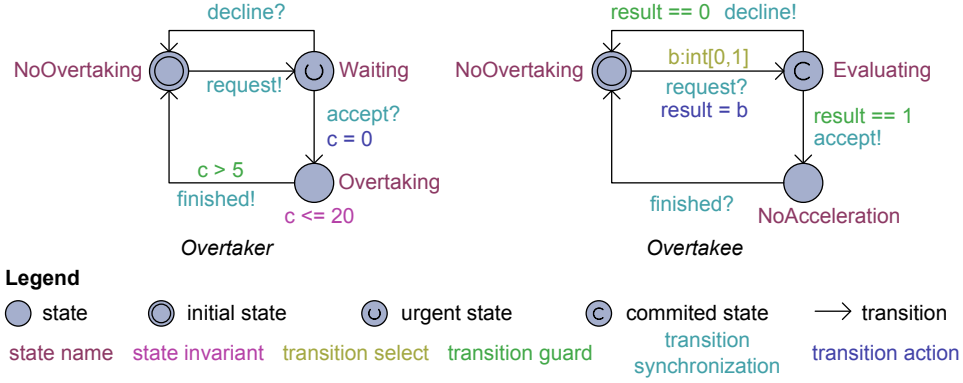


Figure 2.1: Network of Timed Automata Specifying a Coordinated Overtaking

al. [BDL06]. In the remainder of this thesis, we refer to timed safety automata as UPPAAL timed automata resp. timed automata.

A timed automaton consists of states and transitions¹, a set of integer variables, and a set of clocks. Timed automata are flat, i.e., they do not contain hierarchical states. A state has a name, may be the initial state of the automaton, and may define an invariant that must be true as long as the state is active. A transition may define a guard that must be fulfilled such that it may fire. Moreover, a transition may define an action that is executed when the transition fires. Guards, actions, and invariants may refer to the variables and clocks of the automaton. Specific time units, e.g., milliseconds, are not supported in UPPAAL.

A set of timed automata may be composed to a network of timed automata (NTA). In an NTA, timed automata may communicate with each other via shared variables and via channels that enable a synchronous firing of transitions of different automata. An example of an NTA is given in Figure 2.1. It realizes a simplified variant of our overtaking scenario that we introduced in Section 1.1. Within the example, the Overtaker may request the Overtakee to execute an overtaking. The Overtakee has to either accept or decline without any delay. If it accepts, then the Overtaker executes the overtaking within 5 to 20 time units and reports as soon as it finished.

Our NTA defines three states and four transitions per automaton. Moreover, the NTA defines four channels: request, accept, decline, and finished. For synchronizing two transitions, one transition must act as the sender (indicated by !) and the other as the receiver (indicated by ?). For example, the transition between the states NoOvertaking and Waiting of automaton Overtaker may synchronize via the channel request with the transition between the states NoOvertaking and Evaluating of automaton Overtakee. Furthermore, the automaton Overtaker defines a clock c . A clock may be used in three use cases: (1) resetting a clock to zero (e.g., we reset c when firing the transition from Waiting to Overtaking via the expression $c=0$), (2) expressing a time-based state invariant (e.g., the state Overtaking defines such an invariant via the expression $c \leq 20$), and (3) expressing a timed-dependent guard (e.g., we define such a guard via the expression $c > 5$ at the transition from Overtaking to NoOvertaking). In addition, the automaton Overtakee defines a variable $result$ of type int , which is assigned in the action

¹ In this thesis, we use the terms states and transitions instead of UPPAAL's terminology of locations and edges to improve the understandability of our DSMC between MECHATRONICUML and UPPAAL (see Chapter 5) as MECHATRONICUML uses the terms states and transitions as well.

of the transition that leads to state Evaluating. This variable is used within the guards of the transitions that have state Evaluating as its source.

A timed automaton enables to select a non-deterministic integer value of a given type at a transition. The selected value is only accessible at the transition but may be assigned to a variable within the action expression of the transition. For example, the automaton Overtakee uses this feature at the transition that leads to state Evaluating via the expression `b:int[0,1]`. When firing this transition, either the value 0 or 1 is assigned to the intermediate variable `b`. Then, the value of `b` is assigned to the variable `result` within the transition's action.

A state of an automaton may be defined as urgent or committed. If it is urgent, then time may not pass as long as the state is active. A committed state adopts the properties of an urgent state and enforces that one of its outgoing transitions fires next. Another transition may only fire, if its source is a committed state as well (multiple committed states may be active due to a synchronization of multiple transitions). In our example in Figure 2.1.1, the state Waiting is urgent and the state Evaluating is committed. Thus, the NTA assumes that the request may be answered without delay.

UPPAAL provides several language elements that extend the typical definition of timed automata. They are explained in [BDL06]. For example, it supports constant variables, bounded integer variables, urgent synchronization channels (“[d]elays must not occur if a synchronization transition on an urgent channel is enabled” [BDL06]), broadcast synchronization channels (more than two transitions can synchronize their firing), arrays, record types, custom types, template automata (a template automaton may be instantiated multiple times), template parameters, user functions (their behavior is specified using a C-like specification language), and global definitions (of variables, functions, etc.) that are accessible by all automata.

A snapshot² of an NTA is defined by its active states per automaton, its variable values, and its clock values (a clock may have a fixed value or define a range of values using comparisons, e.g., `c >= 5`). “Since clocks are real-valued and time increases continually, timed automata always have an infinite number of [snapshots]. Therefore, the semantics of an NTA is usually defined by means of symbolic [snapshots] 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.” [Hei15]

In Figure 2.2, we show an excerpt of the zone graph of the NTA given in Figure 2.1. In the initial snapshot of a zone graph (in our example `S1`), each automata has its initial state active, all variables start with their initial values, and all clocks are zero. A snapshot change occurs for three reasons:

Timed passes In this case, all clock values increase. However, no state and no variable may change and no state invariant may be violated. We label such a snapshot transition with δ . For example, the snapshot change from `S1` to `S3` defines a so-called delay transition. Afterwards, in snapshot `S3`, the clock `c` is unbounded as both initial states have no invariants and as the outgoing transitions do not use an urgent channel.

²In UPPAAL, snapshots are referred to as *states*. We use the term snapshot to avoid confusion with states of an RTSC.

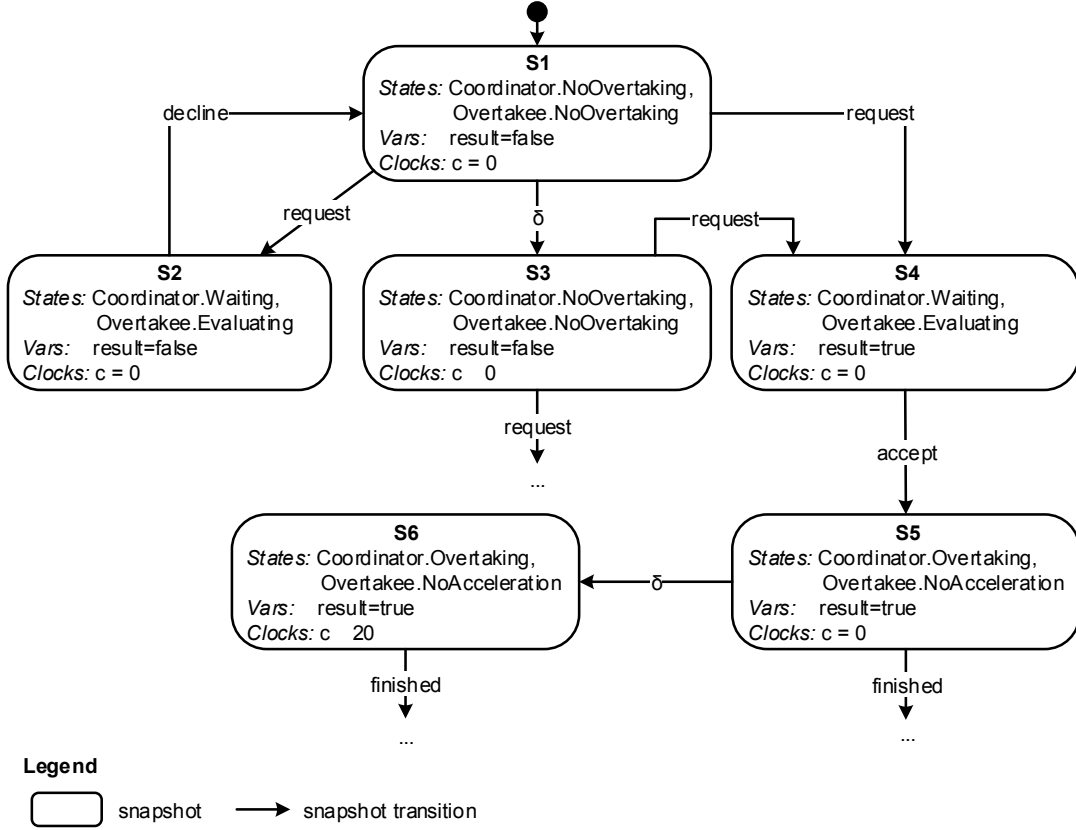


Figure 2.2: Zone Graph (Based on [Hei15])

A single transition fires This is only possible, if the transition does not define a synchronization channel. Moreover, the guard of the transition and the invariant of the target state must be fulfilled. Firing the transition may update the variable values and may reset the clocks. Though, firing a transition does not take time. Therefore, clocks may not progress. Our exemplary zone graph does not use such a snapshot change as all transitions use a channel.

Several transitions fire synchronously This is only possible if (1) the transitions define a synchronization channel with the same name, (2) the transitions belong to different automata, (3) the transitions fulfill their guards, (4) the invariants of the target states are fulfilled, and (5) one transition defines a sending channel (indicated by !) and the other transition/transitions a receiving channel (indicated by ?). Firing the transitions may update variable values and may reset clocks but clocks may not progress. The transition with the sending channel executes its actions before the transition(s) with the receiving channel. In the case of a “normal” channel, a transition with a sending channel must synchronize with exactly one transition that defines a receiving channel. In the case of broadcast channels, a transition with a sending channel synchronizes with all available transitions that define a receiving channel with the same name. In the zone graph, we label such transition snapshots with the name of the channel. An example for such a transition is the one between S1 and S4. Here, the two automata synchronize via the

channel request. In S4, both automata have a new active state, the variable `result` is changed to true and the clock `c` remains at zero.

A transition that fulfills its guards and that may synchronize with another transition (if a channel is assigned) does not have to fire as long as it is still possible that time passes. For example, our NTA of Figure 2.1.1 may always remain in its initial states. A software engineer may enforce state changes by using clock-based state invariants, by defining states as urgent or committed, or by assigning urgent channels to outgoing transitions.

2.1.2 Timed Computation Tree Logic (TCTL)

The timed computation tree logic (TCTL, [ACD93]) is a timed temporal logic that is especially designed for real-time systems. It enables to specify formal properties for a given real-time behavior model like an NTA (see Section 2.1.1). TCTL extends the computation tree logic (CTL, [CES86; HR04]), which enables to verify quantitative timing constraints of finite-state models, by the ability to specify real-time properties. Thus, CTL may only specify that a property will be true at some point in the future, but TCTL may also define a quantitative time bound, e.g., that the property must be fulfilled after 3 time units.

The UPPAAL model checker supports a subset of TCTL, which we call UTCTL (UPPAAL TCTL). In the following, we concentrate on UTCTL as we base our new DSVPL for RTCPs (see Chapter 4) and our DSMC for RTCPs (see Chapter 5) on this language.

UTCTL uses a textual syntax to specify verification properties. A property consists of one to two formula that refer to the elements of a snapshot (a so-called snapshot formula) and exactly one temporal quantifier³.

A snapshot formula is side-effect free and specified via one proposition, which may consist of several propositions that are connected using logical operators. Therefore, it results to true or false. A snapshot formula may reference elements of the NTA like variables, clocks, and states of a process (to test whether this state is active within the snapshot). Moreover, it may form propositions concerning the referenced elements using logical operators.

As an additional syntax element, snapshot formula may also use the predicate `deadlock`. This predicate is true if the snapshot has a deadlock, i.e., “if there are no outgoing action transitions neither from the [snapshot] itself or any of its delay successors.” [BDL06]

A temporal quantifier defines for which snapshots a proposition has to be fulfilled. It typically consists of one path quantifier and one temporal operator. The software engineer can specify via path quantifiers that a proposition must be fulfilled on all paths of the zone graph (A) or on at least one path (E). Concerning the temporal operators, the software engineer can specify that a proposition must be fulfilled for all snapshots ($\square$) of a path or that it is eventually fulfilled for at least one snapshot ($\diamond$) of a path.

UTCTL supports five kinds of verification properties that are divided into three classes: reachability, safety, and liveness (see Figure 2.3). A *reachability property* supports only one kind that states that something may happen, i.e., that a given proposition may be satisfied at any snapshot at any path (see Figure 2.3, top left). For example, a software engineer can specify a reachability property that expresses that the state `Overtaking` of process `Overtaker` may be reached. In UTCTL, we can express this via $E \diamond \text{Overtaker.Overtaking}$.

³UTCTL calls them path formula.

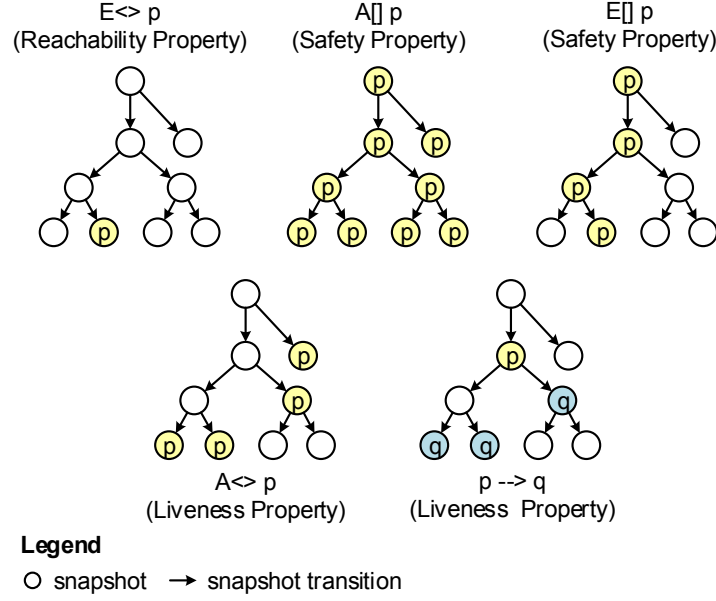


Figure 2.3: Temporal Quantifier Supported by UTCTL and Exemplary Illustrations. The Proposition p is Fulfilled in Yellow Snapshots, the Proposition q is Fulfilled in Blue Snapshots (Based on [BDL06])

UTCTL supports two kinds of *safety properties*. The first kind states that “[...] something will not happen” [Lam77] (see Figure 2.3, top middle). An exemplary property is that it will always be the case that if the state *Overtaking* of process *Overtaker* is active, then the state *NoAcceleration* of process *Overtakee* is active as well (in other words: it will not happen that *Overtaking* is active and *NoAcceleration* is inactive). We express this in UTCTL via $A[] \text{Overtaker.Overtaking} \text{ imply } \text{Overtakee.NoAcceleration}$. Another prominent example is that a deadlock will never happen. In UTCTL, we can express this via $A[] \text{not deadlock}$. The second kind states that something will possibly not happen (see Figure 2.3, top right). An exemplary property is that the process *Overtaker* may always remain in state *NoOvertaking* (in other words: we possibly never leave this state). In UTCTL, we can express this via $E[] \text{Overtaker.NoOvertaking}$.

In general, “[a] liveness property is one which states that something must happen” [Lam77]. UTCTL supports two kinds of these properties. The first kind defines that the proposition is eventually satisfied (see Figure 2.3, lower left). An exemplary property is that the clock c is eventually smaller than 5. In UTCTL, we can express this via $A \leftrightarrow c < 5$. The second kind is the so-called *leads to property*, which defines that whenever a proposition p is fulfilled, then eventually the proposition q will be fulfilled as well (see Figure 2.3, lower right). The concrete syntax for expressing this is $p \rightarrow q$. An example for the *leads to property* is that whenever the state *NoAcceleration* of process *Overtakee* is active, then eventually the state *Overtaking* of process *Overtaker* is active as well. In UTCTL, we can express this via $\text{Overtakee.NoAcceleration} \rightarrow \text{Overtaker.Overtaking}$. Noteworthy, the *leads to property* is semantically equivalent to $A[] (p \text{ imply } A \leftrightarrow q)$. In contrast to TCTL, nesting of nested quantifier

expressions are forbidden, i.e., $A[] (p \text{ imply } A<> q)$ is not expressible in UTCTL as it nests the temporal quantifier expression $A<> q$ within another temporal quantifier expression.

The temporal quantifier expression may not be embedded within a first-order logic expression. For example, the expression $\text{not}(A[] \text{ deadlock})$ is an invalid expression in UTCTL as the temporal quantifier expression $A[] \text{ deadlock}$ is embedded into a not-expression. In contrary, the expression $A[] \text{ not deadlock}$ is legal.

2.1.3 Model Checking Process

The process of model checking is fully automatic and is able to decide whether a property (here: an UTCTL property) is fulfilled or not. For doing this, the model checker generates a zone graph for the NTA and labels each snapshot with the propositions that holds for it. Typically, the model checking process depends on the property under verification. Concerning UPPAAL, the model checking process for each property is as follows (see Figure 2.3):

- If the property uses the temporal quantifier $E<>$, then the process has to identify one snapshot that fulfills the proposition.
- If the property uses the temporal quantifier $E[]$, then the process has to identify one path in the zone graph, where all its snapshots fulfill the proposition. Consequently, it may terminate if the root snapshot (or all snapshots of one hierarchy level of the zone graph) does not fulfill the proposition.
- If the property uses the temporal quantifier $A[]$, then the process has to decide whether all snapshots fulfill the proposition. Consequently, it may terminate as soon as one snapshot does not fulfill the proposition.
- If the property uses the temporal quantifier $A<>$, then the process has to decide whether all paths of the zone graph contain a snapshot that fulfills the proposition. Consequently, it may terminate if the root snapshot already fulfills the proposition.
- If the property uses the temporal quantifier $\text{leadsTo } (p \rightarrow q)$, then the process has to identify one snapshot in the zone graph that fulfills the proposition p and where all paths that succeed p contain a snapshot that fulfills the proposition q .

As stated above, the result of the model checking is the decision whether the property is fulfilled or not. For example, all six UTCTL properties that we stated before as examples are fulfilled in our overtaking NTA. Additionally, a model checker is able to return a trace of the zone graph that shows the fulfillment or the violation of a property. Concerning UPPAAL, if the path quantifier E is used, then UPPAAL is able to generate an exemplary trace that leads to the fulfillment of the property. Moreover, if the path quantifier A is used or the temporal quantifier leadsTo , then UPPAAL is able to show a trace that leads to a snapshot that violates the property. The latter trace variant is called a *counterexample* resp. a counterexample trace.

A software engineer may influence the model checking process by altering several options [BDL06, p. 48]. For example, the software engineer may change the size of the hash table that represents the state space (by default it is 27 MB). Moreover, the software engineer may choose between no state space reduction as well as a conservative and an aggressive one. Furthermore, he may select between two strategies to search the state space: breadth first and depth first. Concerning the traces, he may activate the generation of (counterexample) traces. In addition, he may select between concrete traces and symbolic traces. A concrete trace always states a concrete value for each clock in each snapshot and, therefore, uses delay

transitions between two snapshots that indicate how much time passes. In contrary, a symbolic trace shows clock ranges (e.g., $c > 5$) within its snapshots and, therefore, does not need to use delay transitions between the snapshots. Thus, a symbolic trace might have less snapshots than its corresponding concrete trace. Moreover, the software engineer may choose between three trace identification kinds: some trace, shortest trace, and fastest trace. If some trace is chosen, then UPPAAL may return the first trace that it identifies. Therefore, we assume that this should typically result in the shortest model checking time. If shortest trace is chosen, then UPPAAL will return the trace that has the smallest amount of snapshots. Lastly, if fastest trace is chosen, then UPPAAL will return the trace that has the smallest time delay.

2.1.4 Uppaal Tooling

The UPPAAL model checker is a command line tool, which gets textual inputs as an XML file that respects a given document type definition. The output is also textual but without a defined schema. In addition, the developers of UPPAAL also provide a development suite. This suite enables the software engineer to specify timed automata in graphical diagrams (the concrete syntax is the one that we use in Figure 2.1). Moreover, the suite also enables the user to specify the textual verification properties and to define the model checker configuration. Furthermore, it also supports an interactive simulator where the software engineer can inspect the behavior of the system (the instantiated NTA), e.g., to study the generated (counterexample) trace. Noteworthy, the implementation of UPPAAL is closed source, i.e., we have no access to the internals of it. Moreover, a complete formal semantics definition that covers the complete expressiveness of UPPAAL does not exist but only definitions that cover excerpts of UPPAAL's expressiveness.

2.2 MechatronicUML

MECHATRONICUML [GTB+03; BGHS04; *HBDS15; *BDG+14a; *DPP+16] is a software development method that focuses on the coordination design of CPSs. It is based on UML [UML] as it provides—among others—a component-based software architecture, extended state machines called Real-Time Statecharts (RTSCs), and collaboration diagrams called RTCs. However, MECHATRONICUML is no UML profile but defines its own metamodel. In this thesis, we integrate all our contributions into the MECHATRONICUML method. We focus on MECHATRONICUML's concepts to specify platform-independent software models.

In the following, we first explain MECHATRONICUML's compositional verification approach in Section 2.2.1. In between, we present relevant parts of MECHATRONICUML's DSL that are required for the compositional verification approach, e.g., the component model and RTCs. Afterwards, we introduce RTSCs in Section 2.2.2. Finally, we briefly introduce the MECHATRONICUML Tool Suite, which implements all concepts of MECHATRONICUML, in Section 2.2.3.

2.2.1 Compositional Verification Approach and DSL Overview

“A major aim of MECHATRONICUML is providing formal analyses for guaranteeing correctness of the specified model and thereby safety of the CPS. However, the software

of a CPS typically consists of a multitude of concurrently executed components. As a result, applying formal methods like model checking [BK08] becomes quickly impossible due to the state explosion problem [BCC98]. A second obstacle is that the correctness of software components may depend on the physical environment of the CPS. This applies, in particular, to the feedback controllers of the system that control the movement of (parts of) a CPS [...].” [*GSDH15] For example, the message-based communication between overtaking cars influences their acceleration and deceleration, which may be controlled by a speed controller. Then, the software engineer has to specify discrete and continuous behavior of a car component. When verifying such components, “we are facing a so-called hybrid model checking problem [Hen96] that is not solvable for realistic models using current techniques [ERNF12].” [*GSDH15]

“MECHATRONICUML strives for providing the best possible compromise between efficient [verifications] and formal correctness proofs for the software of a CPS. The resulting approach is twofold. First, we tackle the state-explosion problem by a compositional verification approach that we describe in more detail below. Second, we avoid hybrid verification by distinguishing between discrete software components and feedback controller components [called continuous components] in the MECHATRONICUML component model [*DPP+16]. For discrete software components, the [software engineer] specifies a state-based real-time behavior that is then formally verified based on the compositional verification approach. For [continuous components], whose correctness depends on the physical environment, the [software engineer] does not specify a behavior model in MECHATRONICUML but in a control engineering tool like MATLAB/Simulink or Dymola/Modelica. For verifying and validating the correct integration of discrete software components and [continuous components], the [software engineer] exports the verified discrete software components to the control engineering tool. Then, the [software engineer] can test the correctness of the [continuous components] and their integration with the discrete software components based on simulation [HRS13; PHMG14].” [*GSDH15]

“The core idea of MECHATRONICUML’s compositional verification approach [GTB+03; *HBDS15], which is illustrated in Figure 2.4, is a syntactic decomposition of the software architecture into discrete software components and application level [coordination] protocols that define the interaction of [these discrete software] components.” [*GSDH15] This decomposition significantly improves the scalability of the verification and enables the formal verification of the software of a CPS. “In MECHATRONICUML, we refer to these protocols as Real-Time Coordination Protocols (RTCPs). Each RTCP specifies the asynchronous message-based communication of two communicating partners, called roles, while the behavior of each role is defined by a state-based real-time behavior. For the coordinated overtaking, we have the [O]vertaker role for the overtaker and the [O]vertakee role for the overtaken. Then, each [discrete] port of a component (Car in our example) must implement one role of an RTCP⁴. As a result, the behavior of each [discrete] port is defined by a state-based real-time behavior as well. Then, the decomposition of the software architecture enables the verification of the discrete software components of a CPS in three steps [, which we explain in the following].” [*GSDH15]

⁴In our example, we only provide one RTCP. However, in a fully realistic example, more RTCPs are necessary, e.g., for specifying the communications within a car and between cars and infrastructure.

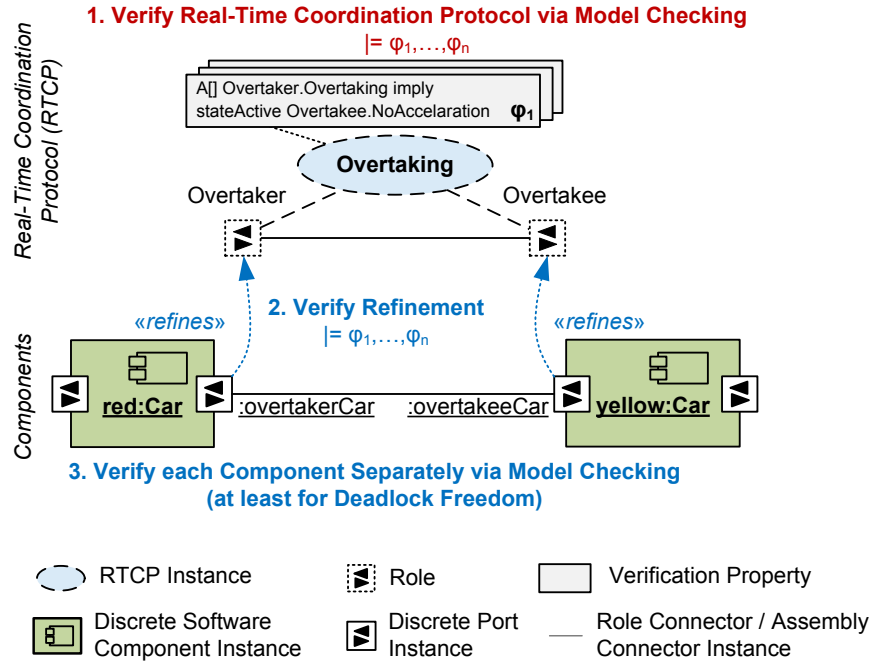


Figure 2.4: Overview of the Compositional Verification Approach (Based on [*GSDH15])

“In the first step, we verify all RTCPs with safety, liveness, and reachability properties using model checking [BK08]. Therefore, the roles of the RTCP have to be specified independently of the components’ ports that implement them. The RTCP contains all relevant information for the communication [...]” [*GSDH15] For example, they shall contain the exchanged messages, their message exchange behavior (including the message buffer specification), and non-functional properties of the connection (e.g., message delay). Based on this specification, we have to define the verification properties. However, there currently exists no domain-specific verification property language at the level of MECHATRONICUML. Therefore, we first have to translate the specification to UPPAAL. Then, we define the properties in UTCTL (see Section 2.1.2). “In our example, we require—amongst other things—that the overtakee may not accelerate while the overtaker is still overtaking.” In UTCTL, we express this via the property φ_1 , which we depict at the top of Figure 2.4. The first step is done as soon as UPPAAL returns that all UTCTL properties are fulfilled.

“In the second step, we need to ensure that the ports of the [discrete software] components correctly refine the roles of the RTCPs, i.e., the implementation of the port does not violate any of the properties that have been verified for the RTCP [in step one]. This [second] step is necessary because implementing the role’s behavior in a port typically requires adding data exchange with the other ports of the component, adding component specific functions, and accessing shared variables inside the component. This step is covered in detail in [*HBDS15].” For example, the software engineer could specify the following intra-component behavior: the red car may only send an overtaking request to the yellow car (via RTCP Overtaking) if there is currently no oncoming traffic (this information is provided via another RTCP). This is an additional dependency to the RTCP Overtaking and influences its execution behavior. Though, it still has to fulfill all its verification properties.

“In the third step, we need to verify for each [discrete software] component that it is free of deadlocks [Gie03]. Such deadlocks may result from an incorrect interaction of the different ports inside the component. We may verify additional safety, liveness, and reachability properties referring to this interaction if necessary.” [*GSDH15] Approaches for specifying intro-component dependencies, synthesizing the component behavior, and verifying its correctness have been introduced by Eckardt and Henkler [EH10] and enhanced by Goschin et al. [Gos14; *DGB14] and Kassner [Kas15].

In this thesis, we improve the specification of RTCPs as well as step one of the compositional verification approach.

2.2.2 Real-Time Statecharts

Real-Time Statecharts (RTSCs) are a combination of UML state machines [UML] and timed automata [BY04]. Compared to timed automata, they provide additional model elements to enable more concise models, which, in turn, shall help to reduce the number of modeling errors and raise the understandability. RTSCs were first defined by Giese et al. [GB03; BGHS04]. Since 2011, we made several adaptations to its syntax and semantics [*BDG+11; *BBB+12; *BBD+12; *BDG+14b; *DPP+16]. In the following, we informally introduce the syntax and semantics as defined in [*DPP+16]. In Appendix E.1.7, we show and explain their meta model as well as their OCL constraints. Heinzemann [Hei15] provides a formal semantics definition based on [*BDG+14b] in Appendix B of his PHD thesis. This formal semantics definition is still valid as [*DPP+16] does not define new syntax and semantics of RTSCs but only improves the understandability of the definitions.

Figure 2.5 shows in its upper part the RTCP SimpleOvertaking, which defines a simplified and slightly altered behavior of the RTCP Overtaking that we introduced in Section 1.1. Therefore, this RTCP encodes a coordination for an overtaking between two cars as well. However, compared to RTCP Overaking, we abstract from some requirements, e.g., we assume that messages will always arrive at the receiver. In the lower part of Figure 2.5, we present two examples for RTSCs. Each RTSC describes the behavior of a role of the RTCP SimpleOvertaking. In the following, we use both RTSCs to introduce the concepts of RTSCs.

Fundamentally, an RTSC describes the behavior of a behavioral element (e.g., a role), has a name and consists of states and transitions. For example, the right RTSC belongs to the role Overtakee, has the name Overtakee as well, and consists of three states (NoOvertaking, Evaluating, and NoAcceleration) and four transitions.

As in UML state machines but in contrast to timed automata, an RTSC may define compositional states. A compositional state contains one or more parallel regions whereby each region contains an embedded RTSC. For example, the RTSC Overtaker on the left of the figure specifies the composite state NoOvertaking. This state contains two regions: Region1 and Region2. RTSCs do not support transitions that connect states of different hierarchy levels. Therefore, the software engineer cannot directly connect the state Overtaker.NoOvertaking.Region2.Waiting (the state Waiting of Region2 contained in the composite state of NoOvertaking of RTSC Overtaker) with the state Overtaking. Instead, the software engineer has to use state connection points (i.e., entry points and exit points), which are provided by UML state machines as well. Due to the existence of composite states, RTSCs support so-called composite transitions. “A transition is composite if its source or target vertex [(a state or a state connection point)] is a composite state. [...] Such transitions are

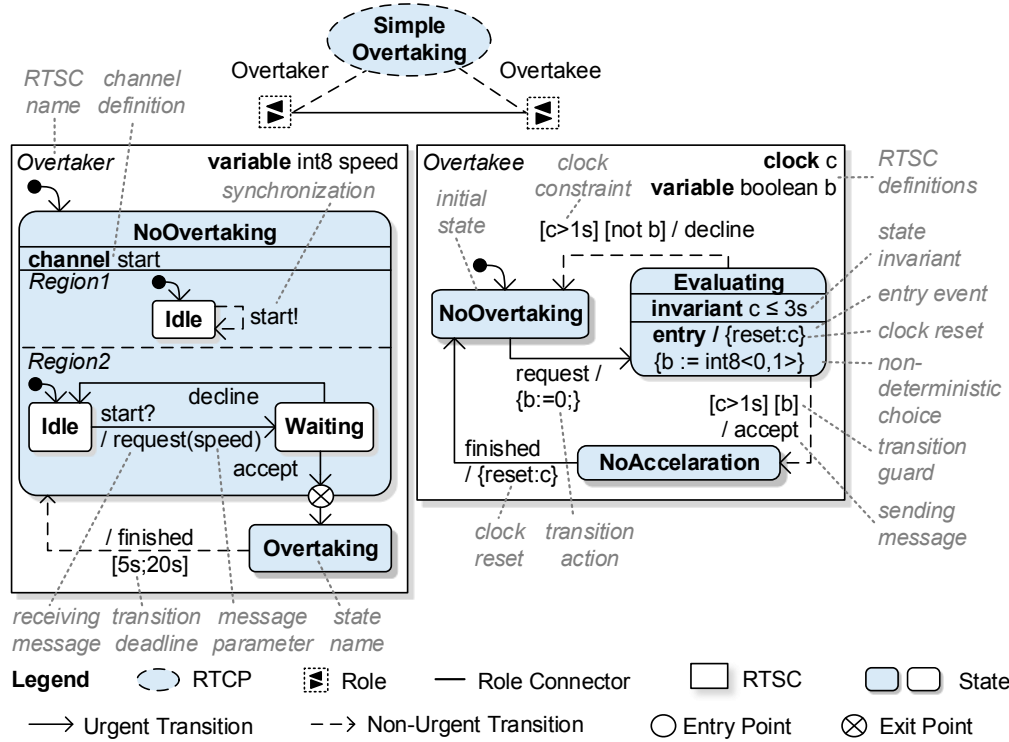


Figure 2.5: The RTCP SimpleOvertaking and its Role Behavior Defined Using RTSCs

abbreviations for compositions of numerous ordinary transitions” [*DPP+16]. For example, the transition from the state `Overtaker.Overtaking` to state `NoOvertaking` of RTSC `Overtaker` is composite.

Similar to timed automata, RTSCs can specify real-time constraints based on real-valued clocks to specify clock comparison expressions and clock resets that reset a clock to zero (see Section 2.1.1). For example, the RTSC `Overtakee` defines the clock `c`. Moreover, RTSCs may define variables and operations, e.g., the RTSC `Overtaker` defines the variable `speed`, which is of type `int8` (a signed integer of 8 bits), and the RTSC `Overtakee` defines the variable `b`, which is of type `boolean`. Clocks, variables, and operations may be used and/or changed within executable behaviors (e.g., assignments, arithmetic expressions) and boolean evaluations. For all these specifications, MECHATRONICUML defines a so-called action language, which is a textual DSL. For example, action language expressions may be used for defining guards and clock constraints at a transitions.

One state per RTSC must be the initial state (the one that is active as soon as the RTSC becomes active). For example, the state `NoOvertaking` is the initial state of RTSC `Overtakee`. A simple state may be set to final, i.e., it may never be left. In concrete syntax, we highlight such a state with a border drawn as a double line. In contrast to earlier definition by Giese et al. [GB03; BGHS04], an RTSC may not contain so-called history states.

Like UPPAAL, a state may define clock-based invariants that restrict the duration the state may be active. If more than one invariant is defined, then all must be true as long as the state is active. Unlike UPPAAL, variable-based state invariants and first-order logic expressions are not supported. Unlike UPPAAL as well, each time value expression in an RTSC (e.g., in a state

invariant) has to specify an SI unit [SI] of category time. For example, the state `Evaluating` defines an invariant via the expression $c \leq 3s$.

A state may define entry, do, and exit events like UML state machines. Though, we define their syntax and semantics differently.

- An entry event may define an action (via the action language) and a set of clock resets. It is executed before the state becomes active due to a firing transition. Thus, it is a short-hand notation for defining an action at each incoming transition. For example, the state `Overtakee.Evaluating` defines an entry event, which consists of the reset of clock `c` and an action, which defines an assignment via an action language expression.
- A do event may only define an action (via the action language). “[It is executed] periodically as long as the state is active. [...] Within each period, the action is executed once. The period is strict. We define the period by means of [an expression that results to an integer and an SI unit]. We assume that the worst case execution time of the effect is shorter than the period. Furthermore, the effect does not have to start at the beginning of the period, but has to end before the end of the period. [...] The event may not be triggered at all if the state is left before the execution of the first effect starts.” [*DPP+16]
- Like an entry event, an exit event may define an action (via the action language) and a set of clock resets. It is executed after the state gets deactivated due to a firing transition. Thus, it is a short-hand notation for defining an action at each outgoing transition.

Note that `MECHATRONICUML` does not enable to state a delay (e.g., a worst case execution time) to an action or a clock reset within an action. Instead, it assumes that their executing does not take time.

In addition, a composite state that consists of more than one region may define a set of synchronization channels, which are a concept taken over from `UPPAAL` timed automata. Therefore, these channels may be used to synchronize the firing of transitions of different RTSCs that (indirectly) belong to the same composite state.

A transition of an RTSC consists of three optional parts: a condition, an effect, and a deadline. If the condition is fulfilled, the transition is enabled, and may fire. Like `UPPAAL`, `MECHATRONICUML` assumes that evaluating whether a transition may be enabled does not take time. Firing a transition causes a state change in the RTSC. While firing, the transition’s effect is executed in one atomic step. The condition of a transition consists of five optional parts that all must be fulfilled at the same time (if they are specified) in order to enable the transition:

Guard The guard is a boolean expression (specified using the action language) that must result to true to enable the transition. It may reference variables and operations but no clocks. For example, the transition from `Overtakee.Evaluating` to `NoAcceleration` defines a guard by referencing the boolean variable `b`. Thus, `b` must be true to enable the transition.

Clock Constraints The software engineer may specify a set of clock comparison expressions (called clock constraints). Each clock constraint compares a clock of the RTSC with an expression that must result to an integer and a time unit. All clock constraints must be true at the same time such that the transition may be enabled. For example, the

transition from `Overtakee.Evaluating` to `NoOvertaking` defines a clock constraint via the expression $c > 1s$.

Synchronization Like UPPAAL, if a transition defines a synchronization by referring to a channel, then another transition of a parallel region must be able to fire a transition that specifies a synchronization using the same channel. Moreover, one synchronization must be defined as the sender (indicated via “!”) and one as the receiver (“?”). For example, the self-transition at state `Overtaker.NoOvertaking.Region1.Idle` and the transition from `Overtaker.NoOvertaking.Region2.Evaluating` to `NoOvertaking` both refer to the channel `start` and use complementary synchronization kinds. Thus, both need to be enabled at the same time. If this is the case, then they fire synchronously.

Trigger Message “A transition that specifies a trigger message may only be enabled if a message of a particular type has been received by the associated role [...]” [*DPP+16] of the RTCP. For example, the transition from `Overtakee.NoAcceleration` to `NoOvertaking` defines a trigger message by stating the message type `finished`.

Priority “The priority of a transition is represented by a fixed natural number greater than 0, whereas a larger number indicates a higher priority. If the enabling conditions described above apply to more than one transition of the same source state, we consider only the transition as enabled which has got the highest priority.” [*DPP+16] In our example, we refrain of using priorities as they are not necessary here.

In contrast to UPPAAL, MECHATRONICUML does not support urgent synchronization channels. Instead, it distinguishes between urgent and non-urgent transitions. “If an urgent transition fires, it does so immediately when it is enabled, i.e., no time passes between enabling and firing. In contrast, a non-urgent transition may postpone the firing even if it is enabled. Consequently, it may also become disabled again [...]. In MECHATRONICUML, non-urgent transitions enable the modeling of uncertain situations in which the firing of a transition is beyond the [software engineer’s] control. [...] In addition, urgent transitions have precedence over non-urgent transitions, i.e., as long as an urgent transition is enabled, no non-urgent transition may fire. If two transitions synchronize via a synchronization channel, then they only fire urgently if both transitions are urgent.” [*DPP+16] For example, the self-transition at state `Overtaker.NoOvertaking.Region1.Idle` is non-urgent and the transition from `Overtaker.NoOvertaking.Region2.Idle` to `Waiting` is urgent. Both transitions synchronize via the channel `start`. The synchronization is non-urgent as one of the two transitions is non-urgent as well. Vulgarakis et al. [VSC+09] and Bengtsson et al. [BGK+02] also use the term urgent and non-urgent transition while Behrmann et al. [BDL06, p. 34] use the term urgent edge. All these definitions and our definition as well have in common that an urgent transition may be expressed in UPPAAL via a transition that defines an urgent channel (see Section 5.4.13.2).

The effect of a transition consists of three optional parts that the software engineer may define and that are executed while firing the transition:

Action The action defines a set of expression that are specified via the action language. It may reference variables and operations but no clocks. For example, the transition from `Overtakee.NoOvertaking` to `Evaluating` defines an action, where the variable `b` is set to zero.

Sending a Message The software engineer may define that an instance of the specified message type shall be sent when firing the transition. Furthermore, if the message defines parameters, then the software engineer has to specify a value for each parameter. For example, the transition from `Overtaker.NoOvertaking.Region2.Idle` to `Waiting` defines that a message of type `request` shall be sent. Moreover, the value of the variable `speed` is used as the parameter value.

Clock Reset A transition may define a set of clock resets. For example, the transition from `Overtakee.NoAcceleration` to `NoOvertaking` defines to reset the clock `c`.

There are three additional parts of the transition effect: the execution of the target's state exit event, the execution of the source's state entry event, and the consumption of the trigger message. The first two are specified within a state. The latter one does not need to be specified explicitly as it is implicitly defined by specifying the message as the trigger message within the transition's condition. The execution order when firing a transition is as follows: (1) the consumption of the trigger message, (2) the action of the exit event of the source state, (3) the transition's action, (4) the action of the exit event of the target state, (5) the sending of a message, (6) the clock resets of the exit event of the source state, (7) the transition's clock resets, and (8) the clock resets of the exit event of the target state.

MECHATRONICUML does not enable to state a delay (e.g., a worst case execution time) to any part of the transition's effect. However, in contrast to timed automata, MECHATRONICUML enables to state that firing a transition (and, thus, executing its effects) of an RTSC may take time. The software engineer may specify how long the firing may take using deadlines. MECHATRONICUML distinguishes two variants: relative and absolute deadlines. "A relative deadline defines the interval where the firing of a transition needs to be finished relatively to the point in time when the firing started. The lower bound specifies the minimum time the transition may take to fire and the upper bound specifies the maximum time it may take." [*DPP+16] For example, the transition from `Overtaker.Overtaking` to `NoOvertaking` defines a deadline between 5 and 20 seconds. "An absolute deadline defines an interval for the values of a specific clock in which the firing of the associated transition needs to be finished. The lower bound defines the minimum clock value at which the firing must terminate. The upper bound defines the maximum clock value accordingly." [*DPP+16] We call a transition with a deadline a time-consuming transition. During the firing of a time-consuming transition, other transitions may fire. If no deadline is defined, then we define that the transition is a non-time-consuming transition, which fires in an atomic step (i.e., no other transition may fire inbetween).

2.2.3 MechatronicUML Tool Suite

The MECHATRONICUML Tool Suite [*DGB+14]⁵ is a graphical tool that aims at providing support for the MECHATRONICUML process [HSST13]. Among others, the MECHATRONICUML Tool Suite supports the process step to define the platform-independent model including the specification and verification of RTCs.

Technically, the MECHATRONICUML Tool Suite is an Eclipse-based tool that provides several graphical and textual editors. It is developed using various Eclipse-based features,

⁵www.mechatronicuml.org/download.html

e.g., EMF [EMF] and OCL [OCL] for metamodeling, GMF [GMF] for implementing the graphical editors, Xtext [Xtext] for defining textual editors and text-to-model transformations, Xtend2 [Xtend] for model to text transformation, and QVTo [QVT; QVTo] for model-to-model transformations. Our self-developed parts of the MECHATRONICUML Tool Suite are open source and distributed under the terms of the EPL-License.

REAL-TIME COORDINATION PROTOCOLS

In this chapter, we explain our new definition of Real-Time Coordination Protocols (RTCPs) for MECHATRONICUML. In particular, we start with the contributions of this chapter in Section 3.1. Then, we point out the stakeholders and requirements of RTCPs in Section 3.2. We explain the new structural specification of RTCPs in Section 3.3. Then, we define our modeling assumptions concerning the underlying software layers, the message exchange, and the QoS in Section 3.4. We explain the new behavioral specification of RTCPs in Section 3.5. Next, in Section 3.6, we define which steps must be executed in order to apply an RTCP into a MECHATRONICUML component model. Afterwards, we explain our implementation in Section 3.7. We describe our evaluation in Section 3.8 before stating the current limitations of our work in Section 3.9. Finally, we discuss related work in Section 3.10 and summarize this chapter in Section 3.11.

Contents of this chapter have been published in [*GSDH15] and [*DPP+16]. Moreover, parts of this chapter were contributed by the bachelor's thesis of Arthur Krieger [Kri13] and the master's thesis of Jana Bröggelwirth [Brö14] that I both supervised. In particular, Krieger made some initial contributions towards the explicit QoS assumptions of the message transfer (see Section 3.3.2) and the message buffer specification (see Section 3.3.4). Bröggelwirth proposed an initial concept for the specification of the multi role behavior (see Section 3.5.2).

3.1 Contributions

The main contributions of this chapter are as follows:

- We define a set of eight requirements that a modeling language of RTCP shall provide.
- We improve the initial definition of RTCPs of Giese et al. [GTB+03; BGHS04] and Hirsch [Hir08]. In particular, we extend the structural definition by incoming message buffers as explicit model elements. Therefore, a software engineer does not need to specify buffers using RTSCs but can define them in a compact and abstract way.
- We enable an alternative behavioral definition for one-to-many RTCPs by providing new keywords for specifying coordination aspects, e.g., for specifying a multicast or a load balancing. We call them one-to-many communication schemata. For each schema, we provide several parameters that influence the behavior. As a consequence, a software engineer can specify the same behavior with significantly less model elements compared to the original behavioral definition.
- We refine the existing modeling assumptions such that no ambiguities exist anymore for formally verifying an RTCP via model checking. In particular, we define the assumed

software layers, the assumed steps of a message transmission, and the QoS assumptions. Concerning the latter one, we also distinguish between RTCP-independent assumptions and RTCP-dependent assumptions that the software engineer has to specify.

- We make adjustments to the MECHATRONICUML component model specification that are necessary due to our changes to the RTCP specification.
- In an evaluation, we show that our alternative behavioral definition for one-to-many coordination protocols is sufficient to specify existing examples and that they enable to model the behavior in a more compact manner than the original specification language.

3.2 Stakeholder and Requirements

We distinguish between two groups of stakeholders. The first group—the *RTCP specifiers*—consists of software engineers that have to define the structural and behavior specification of an RTCP (including the RTCP-dependent QoS assumptions) based on the given requirements. Thus, we assume that this group has significant experience about the domain and model-driven software engineering. Moreover, we assume that this group has at least an advanced experience in model checking as we verify RTCPs via this technique. Though, they do not need to know how to specify domain-specific aspects like asynchronous message exchange on the model checker level.

The second group of the stakeholders—the *component specifiers*—consists of software engineers as well that have to define the component model (structure and behavior) based on the given requirements. Among others, they have to define which RTCPs shall be used. Moreover, they have to define the structure and the behavior of the component model in such a way that the specification of the RTCPs are always fulfilled. Thus, analogously to the RTCP specifiers, we assume that this group has significant experience about the domain and model-driven software engineering. Moreover, we assume that this group has at least an advanced experience in formal verification techniques (especially refinement checking [Hei15]) as they must be able to verify whether the component model specification adheres the RTCP specification.

In the following, we present requirements that verifiable coordination protocols of CPSs like RTCPs shall have at least. We order the requirements according by their priority, where the first is the most important one. The requirements and their priorities are based on our experience and a literature study by Bröggelwirth [Brö14]. Moreover, our requirements consciously do not focus on topics where we have no experience, e.g., security and dynamic reconfiguration. Therefore, we do not claim that this is a complete list.

Concerning a modeling language for verifiable coordination protocols of CPSs like RTCPs, we require that they shall ...

- R1** enable to specify the asynchronous message exchange (structure and behavior) between the roles,
- R2** enable to define real-time requirements for the message exchange,
- R3** enable to specify incoming message buffers using explicit model elements,
- R4** enable to specify protocol-dependent QoS assumptions,

- R5** provide all necessary protocol-independent QoS assumptions,
- R6** focus on functional concerns and not on their technical realization,
- R7** enable to specify parametrized messages, and
- R8** provide explicit one-to-many communication support (structure and behavior).

In the following, we will justify our requirements and state whether they were already supported in the former definition of RTCPs by Giese et al. [GTB+03; BGHS04] and Hirsch [Hir08]. Requirement R1 and R2 are essential for specifying a coordination protocol that shall define the exchange of asynchronous messages under hard real-time constraints. Giese et al. and Hirsch fulfill both requirements. In particular, they adopt the concept of timed automata by using continuous running clocks in order to specify the real-time requirements.

By definition, the receiver in an asynchronous communication does not need to consume an incoming message immediately. Therefore, it is necessary to store all incoming messages in a message buffer until the receiver may consume them. In the concept of Giese et al. and Hirsch, the RTCP specifier could specify message buffers as part of the role's RTSC. The advantage of this approach is that the RTCP specifier is not restricted concerning the behavior of the buffer. However, the disadvantage is that modeling the behavior of a reoccurring buffer kind like a FIFO (First In – First Out) buffer is tedious as well as error-prone. Moreover, the size of the RTSC grows significantly (especially when using several buffers). This makes it harder for the RTCP specifier to understand the RTSC. Therefore, requirement R3 defines that it must be possible to specify the incoming message buffers using explicit model elements.

The requirements R4 and R5 address the topic of QoS assumptions of the RTCP. In general, it is important that all QoS assumptions are defined that influence the model checking results of the RTCP. In particular, we distinguish between assumptions that are valid for all RTCPs and, therefore, do not need to be part of the modeling language, and assumptions that may differ between two RTCPs. For example, the message delay influences the runtime behavior of the RTCP and, thus, the results of the model checking but may differ between two RTCPs. While Giese et al. and Hirsch provide some RTCP-specific QoS assumptions, they miss out to define a complete list. Moreover, they do not specify any RTCP-independent assumptions.

Via requirement R6, we want to achieve that coordination protocols of CPSs may be applied on different platforms. Therefore, the model language should abstract from technical details that are platform-specific. For example, the functional concern of asynchronous message communication may be realized—among others—via shared memory or a non-blocking remote procedure call. Both realizations may be handled by the underlying platform. Therefore, a coordination protocol should abstract from them and only focus on the asynchronous message exchange itself. However, the modeling language cannot abstract from the platform completely. For example, the maximum message delay depends on the platform but is also a QoS assumption that the software engineer has to specify as the results of the model checking depend on it. In our opinion, Giese et al. and Hirsch fulfill this requirement already.

Requirement R7, which defines that it shall be possible that messages may contain parameters, is essential in order to improve the applicability of coordination protocols of CPSs to real-world problems. For example, in our overtaking scenario, the overtaker shall send a request including the proposed overtaking speed. If the language supports message

parameters, the RTCP specifier may define a message request that has the overtaking speed as a parameter (e.g., of type natural number). Without parameter support, this would require more than 300 message types, if we would like to cover each integer value from 0 to 300 km/h. Giese et al. and Hirsch do not fulfill this requirement.

Lastly, Requirement R8 focuses on the one-to-many coordination. As an example, our overtaking scenario in Section 1.1 illustrates that the overtaker is able to communicate with more than one overtaken. Here, the overtaker may only start the overtaking if both overtaken agree. Therefore, modeling a so-called one-to-many RTCP enables to define such safety-critical dependencies within the coordination behavior. As a consequence, we require that it should be possible to define RTCPs that fulfill such coordination structures and that enable to model the behavior in a compact manner. Giese et al. do not enable one-to-many RTCPs. However, the thesis of Hirsch [Hir08] extends their work by this concept. Though, the modeling effort for specifying the behavior of a one-to-many RTCP is significantly higher compared to the effort for the behavior of a one-to-one RTCP. Therefore, the goal of this thesis is to reduce the modeling effort such that is similar to the effort for modeling one-to-one RTCPs.

3.3 Structural Specification

The structural specification of an RTCP has two main aspects: the participants of the coordination—we call them roles—and the role connector that defines assumptions of the message exchange of the roles. An RTCP consists of exactly two roles and one role connector. A role that may receive messages must specify incoming message buffers that store the message until the role instance may consume it. For example, the RTCP MultiOvertaking, which we introduced in Section 1.1, defines the coordination aspects between the roles Overtaker and Overtaken. Both roles are connected by a role connector that, among others, defines that the maximum message delay is 1 s. As both roles may receive messages, they both specify one buffer that shall store all messages.

An RTCP specifier has to instantiate an RTCP for using it in a concrete coordination task. In particular, he has to instantiate the roles, their buffers, and the role connector. For example, the RTCP specifier, can use the RTCP MultiOvertaking to define a coordination between a single instance of role Overtaker and one to two instances of role Overtaken.

Figure 3.1 shows the complete structure of the RTCP MultiOvertaking in concrete graphical syntax. A dashed ellipse contains the name of the protocol. A dotted non-cascaded square represents a so-called single role that may only communicate with exactly one instance of the other role. In contrary, the dotted square is cascaded if the role is a so-called multi role that may communicate with several instances of the other role. A dashed line connects each role with the protocol ellipse. A label at the dashed line shows the name of the role. Each yellow box that is connected via a point-dash line to the role defines an incoming message buffer for this role. The solid line, which represents the role connector, connects the roles. Moreover, a yellow box that is connected via a point-dash line to the role connector defines the QoS assumptions of the asynchronous message exchange. The two triangles within both roles define that they coordinate bidirectionally with each other. Finally, a label at each role (under the role connector) defines the so-called role multiplicity range, which determines the number of connections a role may have.

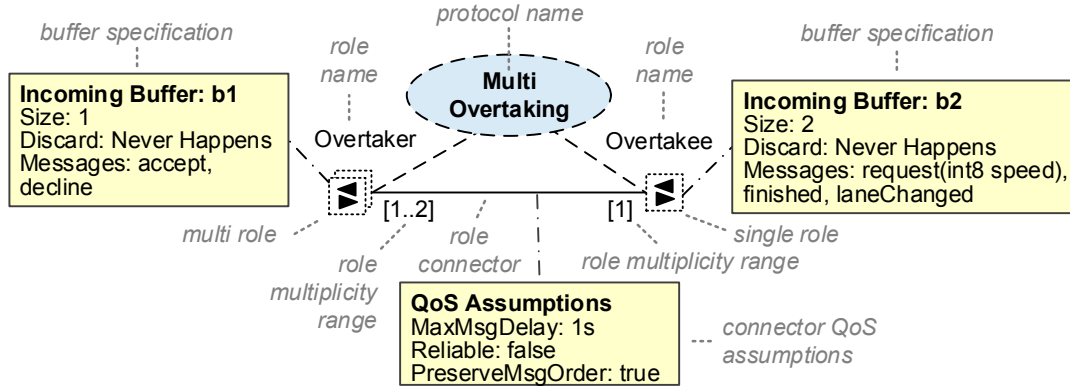


Figure 3.1: The RTCP MultiOvertaking (Based on [*DPP+16])

In the following, we define the structural concepts of roles and connectors in detail in Section 3.3.1 and 3.3.2. Afterwards, we present our classification of RTCPs that will facilitate the description of the remaining concepts. Then, we introduce our new concepts of incoming message buffers in Section 3.3.4 before we explain the instantiation of RTCPs in Section 3.3.5. Finally, we explicitly state in Section 3.3.6 the RTCP structures that are currently not supported.

3.3.1 Role

“A role is a directed, discrete interaction endpoint and represents the type of a coordination participant of an RTCP. Each role of an RTCP has a unique name within the RTCP. A role instance (see Section 3.3.5) represents a concrete coordination participant and is typed over a role.

Two role instances can exchange discrete, asynchronous messages (including message parameters). Furthermore, a role defines which messages its instances may send and receive by defining two unsorted sets of discrete, asynchronous [message] types. We call these sets *sender message types* and *receiver message types*. The sender message types define which types of messages the role may [send]; the receiver message types define which types of messages the role may receive. The sender message types of each role must be equal to the receiver message types of its collaborating role.

If a role may receive messages, then it has to specify incoming message buffers (see Section 3.3.4) that store the messages until the role instance may consume it.” [*DPP+16]

An RTCP always consist of exactly two roles. In principle, coordination scenarios exist that use more than two roles. In Section 3.3.6, we describe why we do not support such RTCPs.

3.3.1.1 Role Direction

“A role may (1) send, (2) receive, or (3) send and receive messages. We distinguish the direction of a role based on these three kinds. A role that may only send messages to the other role is an *out-role*. A role that may only receive messages from the other role is an *in-role*. A role that may send and receive messages is an *in/out-role*. The direction of a role can be derived from its message type specification. We illustrate this in Figure 3.2. If a role

references only sender message types, it is an out-role; if it references only receiver message types, it is an in-role; if it references both, it is an in/out-role.” [*DPP+16]

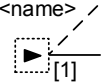
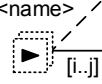
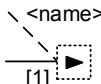
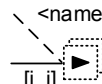
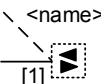
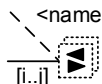
	single role	multi role	sender message types	receiver message types	incoming message buffers
out-role			✓	✗	✗
in-role			✗	✓	✓
in/out-role			✓	✓	✓

Figure 3.2: Concrete Syntax of Roles and their Messages Type Specification [*DPP+16]

“Moreover, the figure also illustrates that the direction of the role defines [whether] the role has to specify an incoming message buffer: An out-role must not specify these buffers, but an in-role and an in/out-role have to specify at least one.” [*DPP+16]

“As depicted in Figure 3.2, the concrete syntax of the out-role only has a filled isosceles triangle within its square (the so-called out-triangle) whose top points to the connector of the protocol. The in-role has a triangle (the so-called in-triangle) whose top points away from the connector of the protocol. The in/out-role is visualized with two triangles. The top of the upper one points to the left and the top of the lower one points to the right.” [*DPP+16]

3.3.1.2 Role Multiplicity Range

Each role has a multiplicity range that limits the number of role instances that an instance of this role may communicate to. In particular, the multiplicity range defines a lower and an upper bound. The lower bound and the upper bound are natural numbers greater than zero. Moreover, the lower bound must be equal or lower than the upper bound. “Note that we describe the lower and upper bound of the role [multiplicity range] by the Min-Max-Notation [Abr74], which is contrary to the multiplicity notation of UML [UML]: The Min-Max-Notation defines for each entity of type x how many associations of type y (at minimum and at maximum) to entities of type z it may have. In contrast to this, the multiplicity notation of the UML defines how many entities of type x may be (at minimum and at maximum) associated over associations of type y to one entity of type z .

The role [multiplicity range] may be variable or fixed. If it is variable, then the lower and the upper bound are different; if it is fixed, then lower and upper bound are equal. If the role [multiplicity range] is fixed and has a value of 1, then this role coordinates with exactly one role instance of a different type. We call such a role a *single role*. If the role [multiplicity

range] has an upper bound greater than 1, then this role can coordinate with more than one other role instance (then all of the other roles must have the same type). We call such a role a *multi role*. Note that the lower bound of the multi role may be greater than 1.

The subrole instances of a multi role are always ordered in a list. An [RTCP] specifier can use the order within the behavior specification of the multi role to define the order of message sequences. We explain this in detail in Section 3.5.” [*DPP+16]

“A square with a dashed, single borderline visualizes the single role; a square with a dashed, cascaded borderline visualizes the multi role (see Figure 3.2). The role-[multiplicity range] is depicted as a label that is located next to the role under the role connector. The label consists of square brackets and one number or two numbers separated by two dots within. If the [multiplicity range] is fixed, then only one number shows the value of the fixed [multiplicity range]; if the [multiplicity range] is variable, then two numbers are shown: the first number represents the lower bound while the second number represents the upper bound.” [*DPP+16]

3.3.2 Role Connector

“Each RTCP contains exactly one role connector that connects the two roles of the RTCP. It represents that instances of the two roles may exchange messages with each other, i.e., two role instances are connected via one role connector instance.

A role connector defines a set of QoS assumptions that the middleware of both roles (see Section 3.4.1) has to ensure. We distinguish between user-defined and fixed QoS assumptions. [User-defined QoS assumptions are RTCP-specific, while fixed QoS assumptions are RTCP-independent.] In the following, we list the user-defined QoS assumptions. Their detailed semantics as well as the definition of the fixed QoS assumptions are given in Section 3.4.3. Moreover, in Section 3.4.4, we discuss why we choose exactly these [user-defined] QoS assumptions. The user-defined QoS assumptions are:

Maximum Message Transmission Delay It is given by a time value, which consists of an expression and a time unit. The expression result must be static, i.e., it may not change during runtime. The delay defines the time between sending the message until enqueueing it into the incoming message buffer of the receiver (note that enqueueing may fail depending on the buffer properties). [We refine this definition in Section 3.4.3.]

Reliability The [RTCP specifier] may choose between the values *true* and *false*. *True* means that the connector is reliable, i.e., messages may be not lost during transmission. *False* means that the connector is unreliable.

Message Order Preservation The [RTCP specifier] may choose between the values *true* and *false*. *true* means that messages that are sent to the same incoming message buffer (see Section 3.3.4) of the other role (i.e., the receiver) have to preserve their sending order when they are enqueued into the buffer; *false* means that the order of the messages is not relevant at all. If the connector is reliable, then the [RTCP specifier] may not select [whether] the messages shall be preserved or not as we assume that the message order will always be preserved. Note that this property concerns all incoming message buffers of all roles.

” [*DPP+16]

Noteworthy, all three user-defined QoS assumptions are mandatory, i.e., an RTCP specifier must define for each of them a value.

To illustrate the dependencies between these three QoS assumptions, consider a role that sends message m1 first, then message m2, and finally message m3. Depending on the specification of the QoS assumptions, several results may be possible:

- If the connector is reliable and the message order shall be preserved, then only one result is possible: the incoming message buffer of the receiver will first enqueue m1, then m2, and finally m3.
- If the connector is unreliable and the message order does not need to be preserved, then 16 results are possible. We distinguish them in four cases: (1) all messages may be lost during transmission (1 possibility), (2) only one message is enqueued to the incoming message buffer of the receiver while the other two get lost (3 possibilities), (3) two messages will be enqueued in an arbitrary order while the third one gets lost (6 possibilities), (4) the three messages are enqueued in an arbitrary order (6 possibilities). In the cases (3) and (4), the messages may be enqueued in an arbitrary order as the messages may overtake each other during transmission. Overtaking is possible as we only state a maximum of the message transmission delay. Hence, the delay of a message may be smaller than the delay of another message, which leads to the overtaking.
- If the connector is unreliable and the message order shall be preserved, then 8 results are possible. We distinguish them in four cases: (1) all messages may be lost during transmission (1 possibility), (2) only one message is enqueued to the incoming message buffer of the receiver while the other two get lost (3 possibilities), (3) two messages will be enqueued while the third one gets lost: (3 possibilities), (4) the three messages are enqueued in the following order: m1, m2, m3 (1 possibility). In case (3), the possible orders are (a) m1, m2, (b) m1, m3, and (c) m2, m3. Therefore, if a message overtakes another message, it is not allowed that the receiver enqueues the overtaker first and the overtakee afterwards. However, it is allowed to enqueue the overtaker if the overtakee is not enqueued.

“In [MECHATRONICUML’s] action language (see Section 2.2), we provide the keyword `maxMsgDelay` that any time value expression of an RTSC may reference. For example, $(4+2 \times \text{maxMsgDelay})$ s is a valid time value for an invariant of a state (see Section 2.2.2).” [*DPP+16]

“We visualize the role connector by a solid black line between the two squares for the roles. Optionally, the QoS assumptions defined by the [RTCP specifier] is annotated as a textual label (a yellow box that has a solid black border) that is connected to the role connector via a point-dash line. We define the label of the QoS assumptions by the following EBNF grammar. The terminals (indicated by a #) refer to attributes of the metamodel.

```

<QoSLabel> ::= 'QoS_Assumptions\n' <delay> '\n' <reliability> '\n'
               <order>;
<delay> ::= 'MaxMsgDelay:_' #maxMessageDelay;
<reliability> ::= 'Reliable:_' #reliable;
<order> ::= 'PreserveMsgOrder:_' #preserveMsgOrder;

```

” [*DPP+16]

As an example, Figure 3.1 on page 37 shows the RTCP MultiOvertaking. The roles Overtaker and Overtakee are connected with the role connector. Moreover, we show the QoS

assumptions of the role connector. The protocol assumes that the maximum message delay is one second. Moreover, the protocol assumes that the connection is unreliable and the message order is preserved.

3.3.3 Classification of Real-Time Coordination Protocols

“We classify RTCPs on the direction of coordination (unidirectional or bidirectional) and on the form of coordination (one-to-one or one-to-many). Figure 3.3 shows the five kinds of an RTCP: a) unidirectional, one-to-one, b) unidirectional, one-to-many, c) unidirectional, one-to-many, d) bidirectional, one-to-one, and e) bidirectional, one-to-many.” [*DPP+16] In the following, we define the possible directions of coordination and the possible forms of coordination in detail.

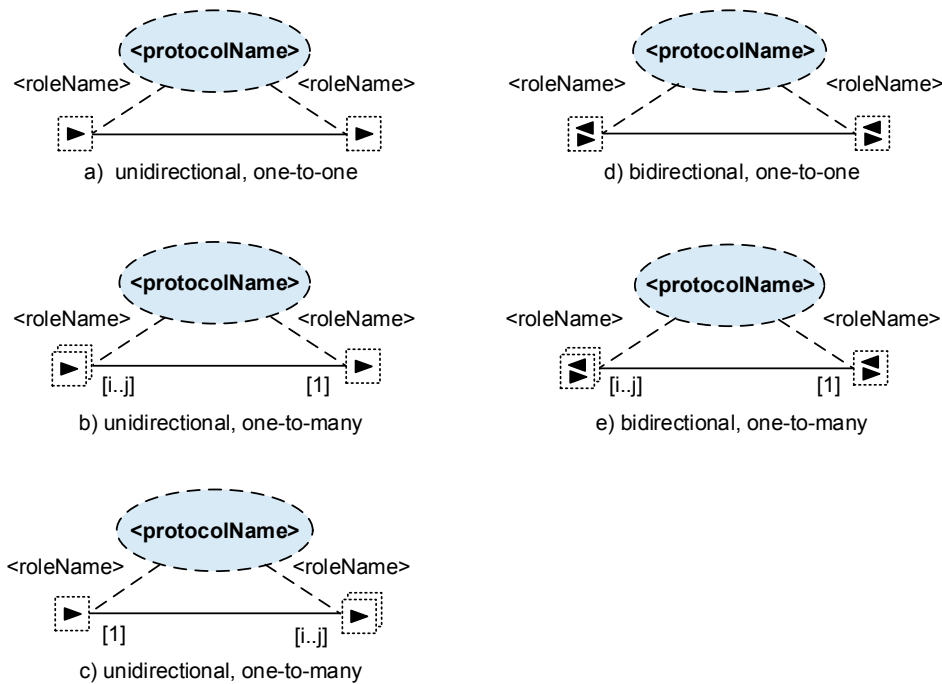


Figure 3.3: The Five Classes of an RTCP [*DPP+16]

3.3.3.1 Directions of Coordination

“An unidirectional coordination means that only one role can send messages and the other role can only receive messages. Therefore, this coordination consists of an out-role and an in-role. Figures 3.3 a), b) + c) show the three possibilities for this coordination.

A bidirectional coordination means that all roles can send and receive messages. Thus, all roles must be in-out-roles. Figures 3.3 d) + e) show the two possibilities for such a coordination.” [*DPP+16]

3.3.3.2 Forms of Coordination

“Two possible forms of coordination exist: one-to-one and one-to-many. These forms of coordination define with how many other instances one role instance can coordinate with and how many instances participate in this coordination in total.

A one-to-one RTCP consists of two single roles that are connected with a role connector. Therefore, only two role instances are participating [in] this form of coordination. Figures 3.3 a) + d) show the two possibilities for a one-to-one coordination.

A one-to-many RTCP consists of one single role and one multi role that are connected with a role connector. Therefore, only two or more role instances are participating [in] this form of coordination where one role instance may exchange messages with multiple other role instances. Note that we still call it a one-to-many RTCP even if the RTCP is unidirectional and the single role is the sender. Figures 3.3 b), c) + e) show the three possibilities for specifying a one-to-many form of coordination.” [*DPP+16]

3.3.4 Incoming Message Buffer

A role of an RTCP requires incoming message buffers as it has to be able to store the messages before it consumes them. The reason for this is that we define that the message exchange between roles is always asynchronously.

Giese et al. [GTB+03] define that the asynchronous messages are stored in “[message] queues given in form of additional automata”. Burmester et al. [BGHS04] refine this statement by defining that each message type that may be received is stored in a separate queue. Each queue has a maximum size that the software engineer may define. Moreover, they indirectly define that it may never be the case that the queue is full but a new message arrives as such a behavior leads in their state machine to an error state that may never be left.

By studying internal case studies and related work (e.g., [AUTOSAR] and [MARTE]), we identified that RTCPs require more expressiveness concerning the specification of incoming message buffers. For example, [AUTOSAR] supports the use case that a buffer has size 1 and shall only store the last incoming message. Thus, if the buffer already contains a message, then this message shall be discarded. The definition of Burmester et al. [BGHS04] does not support such a use case.

In the following, we present our new specification for incoming message buffers of roles of an RTCP that supports the relevant use cases that we identified based on our own experience and related work. In particular, our specification does not require additional state machines as we want to avoid this repetitive and error-prone task. Instead, we provide explicit model elements that represent incoming message buffers. In this section, we define their syntax in detail and briefly explain their semantics. A detailed semantics definition is given in Section 5.4.13, when we explain how we transform this specification to UPPAAL timed automata.

Our definition of incoming message buffers is as follows: “An instance of an in-role or an in/out-role always buffers each incoming messages first, before it may consume the message (by firing a transition within the role’s RTSC). For this buffering, we provide incoming message buffers as explicit [model] elements as we identified several properties for a buffer that depend on the RTCP under development. In contrary, we do not provide outgoing message buffers as explicit [model] elements as we did not identify properties that are RTCP-

specific. In particular, we do not assume [whether] outgoing message buffers exist at all (see Section 3.4).” [*DPP+16]

A role may have multiple incoming message buffers. In particular each in-role and each in/out-role has to define at least one buffer and at most n buffers, where n is the number of message types that this role may receive. Furthermore, each incoming message type that a role may receive must be assigned to exactly one incoming message buffer. Therefore, an incoming message buffer may store at least one and at most all message types that this role may receive.

“In addition, a message buffer has a name in order to reference a particular buffer in an MTCTL property [(see Chapter 4.3.4)]. Moreover, a message buffer is a strict queue with a fixed size that follows the FIFO (First In – First Out) principle. A message buffer enqueues a received message at the back of the queue and dequeues messages at the front of the queue. This enqueueing is [independent] of the type of the incoming message.” [*DPP+16]

In contrast to several related approaches (see Section 3.10), we enforce a fixed buffer size instead of an infinite one due to two reasons: First, we consider the buffer size as an important property that influences the behavior of the model (especially if the buffer is full and a new message arrives). Second, we need a fixed size in order to apply DSMC (our model checker UPPAAL cannot deal with an infinite size as this would result in infinite buffer states).

“If the [RTCP specifier] defines more than one incoming message buffer for a role, it is possible that more than one of these buffers contains at least one message at runtime. Then, the role’s RTSC may dequeue from any message buffer. Thus, there is no priority defined among the incoming message buffers of an *in-role*.

In MECHATRONICUML, an incoming message buffer cannot overflow, i.e., it may never be the case that the buffer is full but still enqueues a message. Instead, the [RTCP specifier] has to select one of three pre-defined strategies that avoid the buffer overflow:

Discard Oldest Message in Buffer Discards the oldest message in the buffer (i.e., the message at the head of the queue) and enqueues the new arriving message. Note that this is the only possibility in MECHATRONICUML to delete a message from the buffer without consuming it. A typical use case for applying this strategy is when each message represents a measurement value, e.g., the actual speed of a car, and the receiver is interested in the latest measurement(s). Then, messages that are already in the buffer contain old measurements and, therefore, are deleted such that a new message with a new measurement may be stored.

Discard Incoming Message Discards all incoming messages as long as the buffer is full. Therefore, the content of the buffer is not affected. A typical use case for applying this strategy is when each message represents a task that shall be executed by the receiver and an earlier task has a higher priority because the already received tasks are preserved and new incoming tasks are deleted.

Never Happens The situation that the buffer is full but a new message arrives must never happen as each message is important. If it still happens, then this is a fatal error that shall end the execution of the RTCP.” [*DPP+16] A typical use case for applying this strategy is when each message represents a task that the receiver must execute and the tasks depend on each other, e.g., each task could define a relative driving maneuver like “drive 50m straight” or “turn right”. The RTCP specifier can formally verify whether

this situation never happens by specifying a property in MTCTL (see Section 4.3; e.g., for a buffer `b1` at role `1` a valid property is `AG not messageDiscarded(role1.b1);`) and applying DSMC (see Chapter 5).

“In the following, we list the possible buffer properties that we consciously not support in MECHATRONICUML as we did not identify relevant use cases in the domain of CPSs:

Other Buffer Semantics We do not support that [an RTCP specifier] may select other strategies to sort the messages that a buffer contains. For example, we do not support the strategies LIFO (Last In – First Out), message type priorities (a message type with a higher priority is enqueued into the buffer before all message types with a lower priority), or user-defined sorting behaviors (e.g., using state machines).

Buffer Priorities We do not support that [an RTCP specifier] may assign each buffer of a role a certain priority such that a buffer with a higher priority must be empty in order to allow that [messages] from buffers with a lower priority may be consumed. Noteworthy, [an RTCP specifier] can already encode buffer priority indirectly within the role’s RTSC, e.g., by using transition priorities. Though, we did not encounter such examples yet.

Buffer Filtering We do not support that the [RTCP specifier] may filter (using a black- or whitelist) incoming messages (e.g., depending on their type or parameter values) before we try to enqueue them to the buffer. Again, the [RTCP specifier] can already encode a subsequent filter within the role’s RTSC, i.e., he can specify a consumption of specific message types without further processing. Though, we did not encounter such examples yet.

” [*DPP+16]

The concept of enabling multiple message buffers per role and enabling that a buffer may contain several message types increases the expressiveness for handling incoming messages without having the need for other buffer properties like the ones mentioned above. In particular, there exist two extreme cases, which we present in the following:

One Buffer for all Message Types If the RTCP specifier uses this option, then all messages have the same priority and shall be processed in incoming order. As a consequence, if a buffer may store the message types `t1` and `t2`, the buffer is already full, a message of type `t1` arrives at the buffer, and the option `Discard Oldest Message in Buffer` is selected, then it may be the case that a message of type `t2` may be discarded. As another consequence, the buffer size and the discardment strategy affects several message types. For example, this option is appropriate for both buffers of RTCP Overtaking (see Figure 3.1):

- Concerning buffer `b1` at role `Overtaker`, this option (combined with the option that the size is one) enables us to state that the buffer either contains the message `accept` or the message `decline`, and, therefore that only one of the two may be available for consumption. Moreover, in combination with the discardment strategy “never happens”, we are able to state that it shall never happen that one of the two messages are stored in the buffer and another message (independently of its type) arrives. Both statements are useful to restrict the behavior of the RTCP but cannot be made when each message type is stored in an extra buffer.

- Concerning buffer b2 at role Overtakee, we can enforce that the messages are processed in their incoming order—independently of their type. This is important as the messages request and finished may both be available at the same time while in state NoAcceleration (see Figure 3.6). The role’s RTSC assumes that if finished arrives before request, then only finished is available for consumption. Therefore, the behavior of the role would change if the message types would be stored in different buffers.

One Buffer per Message Type If the RTCP specifier uses this option, then the order of processing all messages of the same type is important, but it is not relevant whether messages of type t1 are processed before type t2. As a consequence, if a message of type t1 arrives at the buffer but the buffer is already full, and the option Discard Oldest Message in Buffer is selected, then it may never be the case that a message of type t2 may be discarded. As another consequence, the RTCP specifier can define a buffer size and a discardment strategy for each set of message types. For example, this option is appropriate in the following scenario: a role receives from the other role various measurements like the temperature, atmospheric pressure, and the wind direction. Each measurement is sent via an own message type. The receiver is only interested in the last update of each measurement. By using one message buffer per message type, the RTCP specifier can fulfill this requirement by setting the size of each buffer to 1 and setting the discardment strategy to “Discard Oldest Message in Buffer”. If the RTCP specifier would have to use one buffer only, then this requirement could not be specified without having complex filter strategies (that we currently not support). In this example, the filter must be able to specify that a message may only be enqueued within the buffer if a message of the same type is not already within the buffer.

Besides these two extreme cases, the RTCP specifier may also define variants in between. For example, consider the following variant of RTCP Overtaking: besides the default message exchange, the Overtaker may also be able to send a message emergencyBrake to the Overtakee that indicates that the Overtakee shall immediately execute an emergency brake. Consequently, this message has a higher priority than the other ones. The RTCP specifier can specify this by adding a second buffer b3 to role Overtakee that only stores messages of type emergencyBrake. If such a message is received, the role does not need to consume all other messages that arrived before but can directly access the emergencyBrake message.

“[The concrete syntax of a message buffer] is an optional textual label (a yellow box that has a solid black border) that is connected to the role via a point-dash line. The label consists of the name of the buffer, its size, its buffer overflow avoidance strategy, and the messages types the buffer may contain. We define the label of [an] incoming message buffer by the following EBNF [ISO14977] grammar. The terminals (indicated by a #) refer to attributes of the metamodel.

```

<bufferLabel> ::= <name> '\n' <size> '\n' <discard> '\n' <order> '\n'
                  <messages>;
<name>         ::= 'Incoming_Buffer:_' #name;
<size>         ::= 'Size:_' #bufferSize;
<discard>      ::= 'Discard:_' #bufferOverflowAvoidanceStrategy.literal;
<messages>    ::= 'Messages:_' <message> ['_',_ <message>];
<message>     ::= #messageType.name;

```

” [*DPP+16]

Figure 3.1 shows the incoming message buffers for the roles Overtaker and Overtakee. Both roles define exactly one buffer. The buffer *b1*, which belongs to role Overtaker, contains all message types that may be received. It has the size 1 as the Overtaker either receives an *accept* or an *decline*. Moreover, it may never happen that *b1* is full but a message arrives as we require that the message in the buffer is already consumed before a new message arrives. The buffer *b2*, which belongs to role Overtakee, contains all messages that may be received by its role. It is defined that it may never happen that *b2* is full but a new message arrives. We identified by applying our domain-specific model checking (see Chapter 5) that the size 2 is sufficient such that this never happens.

3.3.5 Real-Time Coordination Protocol Instance

“An instance of an RTCP consists of (1) a set of role instances that are typed over the defined roles and (2) a set of role connector instances that are typed over the role connector.

If the RTCP has the form one-to-one, then there exists only one possible RTCP instance, which consists of one instance per single role and one connector instance that connects the two role instances. If the RTCP has the form one-to-many, then [the] number of possible RTCP instances depends on the role [multiplicity range] of the multi role, which we call multi role [multiplicity range]. In fact, the number of possible RTCP instances is $[upperBound - lowerBound + 1]$, where *lowerBound* is the lower bound and *upperBound* is the upper bound of the multi role multiplicity range]. The RTCP specifier determines at design time the value of the multi role [multiplicity range], i.e., he defines a role multiplicity (*n*). A one-to-many RTCP instance consists of exactly one instance of the multi role and *n* instances of the single role, where $n \in \mathbb{N}$ and $lowerBound \leq n \leq upperBound$.

A multi role instance may not directly communicate with the multiple single role instances. Instead, the multi role instance contains *n* so-called subrole instances. Each subrole instance is connected to a different single role instance via a role connector instance. Consequently, [...] *n* does not only define the number of single role instances and the number of subrole instances but also the number of role connector instances.

Each single role instance contains its separate set of incoming message buffer instances that are typed over the specified incoming message buffers of this role. In particular, each buffer type of a single role is instantiated exactly once per single role instance. Note that this definition is independent of the RTCP’s form (one-to-one or one-to-many). Furthermore, a multi role instance does not contain incoming message buffer instances [directly]. [Instead], each subrole instance contains its separate set of incoming message buffer instances that are typed over the specified incoming message buffers of this multi role. In particular, each buffer type of a multi role is instantiated exactly once per subrole instance. Note that we do not explicitly define incoming message buffer instances in our metamodel as they do not have variable parts.

Concerning the QoS assumptions, we define that each role connector instance shares the same QoS assumptions that are already defined for the role connector type.

As an example, a valid instance of RTCP [MultiOvertaking] (see Figure 3.1) may specify a coordination between three role instances. One instance is typed over the role [Overtaker] and contains two subrole instances. Each of these two subrole instances is connected via a separate role connector instance to one of the two role instances, which are typed over

the role [Overtakee]. Therefore, this instantiation defines a one-to-two coordination. This instance of the RTCP [MultiOvertaking] is valid because the maximum role [multiplicity] of role [Overtaker] is two and the current role [multiplicity] of the role instance, which is typed over role [Overtakee] is two as well.” [*DPP+16]

“Figure 3.4 shows the aforementioned instance of RTCP [MultiOvertaking] in concrete graphical syntax. All concrete syntax elements of the example are annotated in gray. A dashed ellipse contains the protocol instance label which consists of the name of the protocol. The name is underlined and prefixed by a colon. Each instance of a single role has its own dotted square and its own dashed line, which connects it to the ellipse. A multi port instance contains a fixed number of subrole instances, which are framed by a dashed rectangle. Each subrole instance has its own dotted square. A multi port instance is connected to the ellipse via a dashed line. A label is placed next to the dashed line of each role instance (single and multi) and consists of the name of the role that is underlined and prefixed with a colon. Single role instances and subrole instances contain triangles to show the direction of the role instance. The solid line, which represents the role connector instance, connects a single role instance and a subrole instance. We refrain from showing the four incoming message buffer instances (one at each subrole instances and one at each single role instance) as they do not change compared to their definition at the RTCP. In addition, we do not show the QoS assumptions at the role connector instances as they do not change compared to their definition at the RTCP as well. We use the same graphical syntax elements if an RTCP instance has a one-to-one form of coordination.” [*DPP+16]

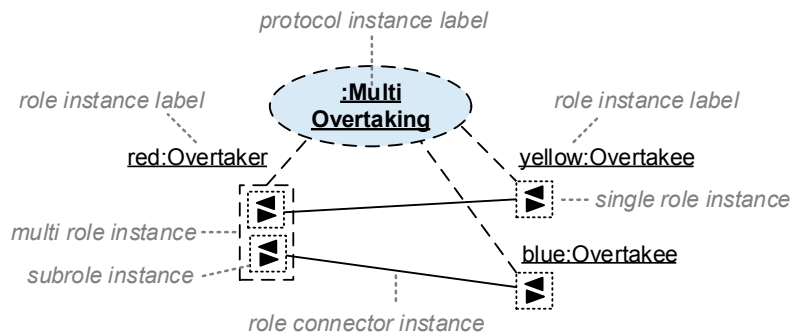


Figure 3.4: Instance of RTCP MultiOvertaking (Role Multiplicity = 2) [*DPP+16]

3.3.6 Unsupported Structures

We do not support many-to-many RTCPs (i.e., an RTCP, where each role may have multiple role instances) as our model checking approach (see Chapter 5) is already not able to complete the verification of some one-to-many RTCPs due to too high memory requirements. For example, as described in Section 5.8, our evaluation PC is not able to complete a verification of the RTCP MultiOvertaking for all selected verification properties when the role multiplicity is two. Therefore, an RTCP instance with only three role instances in total (one multi role instance with two subrole instances and two single role instances) and two role connector instances is sometimes already too big. The simplest many-to-many RTCP that is not a one-to-many RTCP has already four parallel subrole instances (two subrole instances per

multi role instances) and four role connector instances. Therefore, depending on the RTSCs, our current verification approach could already struggle with the simplest many-to-many RTCP. Consequently, unless the verification approach does not change, we assume that the vast majority of many-to-many RTCPs cannot be verified. Therefore, we cannot prove a safe coordination, which is a disqualification in the domain of CPSs.

We also do not support an RTCP that has more than two roles. Again, the reason for this decision is that we assume that our current verification approach would in most cases not be able to complete its task due to too high memory requirements for non-trivial RTCPs. For example, a rather simple RTCP with three roles that may communicate with each other would already require six subrole-instances (two per multi role instance) and three role connector instances. Again, depending on the RTSCs, our current verification approach could already struggle with this simple RTCP, which disqualifies this kind of RTCP structure as well.

In addition, we do not support an RTCP with only one role that may be instantiated several times and where each role instance may communicate with the other role instances. Again, we expect that we would have to deal with the same performance issues that we discussed before.

3.4 Modeling Assumptions

“According to the Model-driven Architecture (MDA) [MDA], the [platform-independent model (PIM)] shall not specify platform-specific properties to enable a reuse of these models at different platforms. Among others, RTCPs belong to the PIM and, therefore, do not specify platform-specific properties. However, RTCPs specify an asynchronous message exchange behavior. This behavior depends on the underlying platform, e.g., it influences the message delay and the reliability. Therefore, in order to enable a model checking of RTCPs, these platform-specific dependencies must be stated as assumptions. Otherwise, the verification results will be meaningless. In addition, these assumptions are the requirements for designing the platform-specific models as well as the code generation.

[I]n this section, we provide our modeling assumptions that we define for RTCPs such that we can specify a formal verification of RTCPs using model checking. At first, we define our assumed software layers in Section 3.4.1. Then, we define the steps that we assume if an asynchronous message exchange happens in Section 3.4.2. [We] state the fixed and variable QoS assumptions that an RTCP has in Section 3.4.3.” [*DPP+16] Finally, we discuss our assumptions in Section 3.4.4.

3.4.1 Assumed Software Layers

“In general, we assume that software specified in MECHATRONICUML is executed in a multi-layer architecture (see Figure 3.5). In particular, we assume that each system distinguishes between two layers: the application layer and the middleware whereby each system consists of exactly one application layer and at least one middleware.” [*DPP+16]

“The application layer executes all platform-independent models (PIM). For example, concerning RTCPs, the application layer executes the role behavior and the management of the incoming message buffers (adding and removing messages). The middleware handles all platform-specific details, e.g., the transmission of an asynchronous message. Hereby, we make no assumptions [whether] a system uses one middleware instance or several parallel middleware instances, e.g., one middleware instance per role behavior.

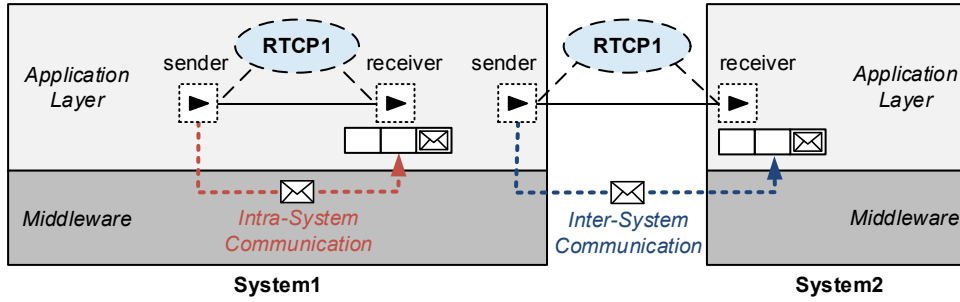


Figure 3.5: The Assumed Layer Model of an RTCP [*DPP+16]

Similar to AUTOSAR’s definition of the application layer and the virtual functional bus [AUTOSAR], the application layer of MECHATRONICUML may not execute an intra-system communication of an asynchronous message exchange (i.e., the sender and the receiver are allocated at the same system) by itself. Instead, the application layer has to hand over all asynchronous message exchanges to the middleware. As a consequence, the application layer cannot distinguish between intra- and inter-system communication (in an inter-system communication, the sender and the receiver are allocated at different [systems]). The advantage of this definition is that the same RTCP may be used for both communication styles (intra and inter) as long as its QoS assumptions are guaranteed. This raises the reusability of each RTCP and abstracts from platform-specific details. We illustrate this definition in Figure 3.5: The RTCP RTCP1 may be used for an asynchronous message exchange within [one] system as well as between two systems.” [*DPP+16]

3.4.2 Assumed Steps of a Message Transmission

“Based on the assumed layer model of Section 3.4.1, we assume that the transmission of an asynchronous message instance contains the following steps:

1. The transmission starts in the sender’s application layer. In particular, the sender’s RTSC fires a transition that defines that a message of a specific type shall be sent. Then, an instance of this message type is created and the parameter values (if the message defines parameters) are assigned to it.
2. The message instance is delivered to the sender’s middleware.
3. The sender’s middleware sends the message to the receiver’s middleware. As defined before, this may be the same middleware in the case of an intra-system communication.
4. The receiver’s middleware receives the message (if the message did not get lost or other problems occurred).
5. The receiver’s middleware eventually enqueues the message (and its parameters) in the assigned incoming message buffer of the receiver role (the enqueueing may fail, e.g., if the buffer is full and the buffer strategy is to discard incoming messages).
6. At the application layer, the receiver’s RTSC consumes the message from the buffer [as soon as it is available at the buffer’s front position] by firing a transition that specifies the type of this message as its trigger message.

” [*DPP+16]

3.4.3 Quality of Service (QoS) Assumptions

“MECHATRONICUML assumes that the middleware (see Section 3.4.1) handles the details of the asynchronous message exchange. Therefore, we present in the following the QoS assumptions that the middleware must always guarantee:

1. We assume that the message transmission delay defines the time from sending the message from the application level until the enqueueing of the message within the receiver’s incoming message buffer (note that the enqueueing may fail if the buffer is already full and the strategy “Discard Incoming Message” is selected). Therefore, this delay is not just the transmission via the physical medium but also the transport between the underlying network layers. Concerning our assumed steps of a message transmission (see Section 3.4.2), the message transmission delay comprises the steps 1 to 5.
2. We make no assumptions [whether] the sender’s middleware delays the sending of the message or [whether] the receiver’s middleware delays the enqueueing of a message into an incoming message buffer.
3. We assume that the sender’s middleware is able to determine [whether] a message must be delivered to another system (and to which system in particular) or [whether] the receiver is on the same system. Therefore, the application layer does not need to know about the deployment of the software. Thus, a PIM model element like an RTCP may be used for intra- as well as inter-system communication.
4. We make no assumption [whether] a message has to be transmitted over several hops to reach the receiver’s middleware.
5. We assume that the sender’s middleware cannot forbid the sending of the message. Thus, the transition of the sender may fire without considering the state of its middleware.
6. The [software engineer] does not need to specify outgoing message buffers in the PIM model as we assume that the message is directly handed over to the middleware. In particular, we do not make assumptions [whether] outgoing message buffers exist at all. Even if the sender’s middleware uses outgoing message buffers, the [software engineer] does not need to specify its properties explicitly due to our other assumptions. For example, via assumption 8.a), we indirectly assume that if an outgoing message is used, then its size is sufficient to store all outgoing messages. Therefore, the [software engineer] does not have to assume a certain size.
7. At the application layer, an unidirectional RTCP may only send messages in one direction. However, we do not make assumptions how the middleware realizes such a communication. Thus, it may be possible that the middleware uses a bidirectional communication as it sends acknowledgments for each received message. Nevertheless, from the perspective of the application layer, the communication is considered as unidirectional.
8. If the [software engineer] assumes that the connector is reliable, ...
 - a) ...then we assume that all messages that are sent from the sender’s application layer will be enqueueing into the receiver’s message buffers (note that enqueueing itself may fail if the buffer is already full and the strategy “Discard Incoming Message” is selected). As a consequence, we assume that all underlying layers

have sufficient buffer sizes to support the theoretically maximum number of concurrent messages in transit¹.

- b) ... then we assume that the sending message order is the same as the receiving message order. Therefore, as long as no message discarding happens due to a full buffer, the sending message order is the same as the order of message within the buffer. As a consequence, if messages may overtake each other within transmission, we assume that the receiver's middleware will still enqueue the messages in the correct order. For example, if the sender role sends message m1 first and then message m2 and m2 overtakes m1, then the receiver's middleware will not enqueue m2 directly. Instead, it will wait until m1 arrives. Then, it will enqueue m1 first and m2 afterwards.
- c) ... then we assume that the message will not arrive at the receiver's middleware after the user-defined maximum message delay. In particular, we assume that the receiver's middleware guarantees that each message is enqueued within its incoming message buffer before the maximum message delay expires (though the message may still be discarded if the buffer is already full and the strategy "Discard Incoming Message" is selected).

9. If the [software engineer] assumes that the connector is unreliable ...

- a) ... then we assume that the message may get lost during the transmission steps 2 to 5 (see Section 3.4.2).
- b) ... and a message arrives after its maximum transmission delay, then we assume that the message will be discarded and, therefore, not stored within the incoming message buffer of the receiving role.
- c) ... and the [software engineer] specifies that the message order shall be preserved when enqueueing a message into the incoming message buffer, then we assume that the receiver's middleware considers the message order before enqueueing the messages into the incoming message buffer. For example, consider that a sender role sends message m1 first and then message m2. Both messages have the same buffer as a target. If m2 overtakes m1 and m2 arrives earlier at the receiver, then the receiver's middleware has to make sure that it may not happen that m2 is enqueued first and m1 afterwards. However, it would be allowed to either enqueue m1 or m2 or none of them.
- d) ... and the message order does not need to be preserved, then we assume that the receiver's middleware enqueues all received messages (including overtaken messages) in their buffers that arrive before the maximum delay.

10. We assume that the receiver's middleware always enqueues each message at most once. Therefore, we assume that the receiver's middleware detects duplicate messages, e.g., using duplicate message detection mechanisms [Kiz05].

¹Noteworthy, we require for our domain-specific model checking [(see Chapter 5)] that the [software engineer] defines an upper limit of the maximum number of concurrent messages in transit. This is due to limitations of the model checker UPPAAL. However, the upper limit is a parameter for the model checking itself and no parameter for the PIM model. The [software engineer] can change this parameter value within the MECHATRONICUML Tool Suite. Moreover, using our DSVPL MTCTL [(see Chapter 4)], the [software engineer] can define a verification property that states that the cache for the messages in transit may never overflow. All coordination protocols have to fulfill this verification property.

11. We assume that corrupted messages are never enqueued into the receiver's incoming message buffer. As a consequence, we assume that the receiver's middleware detects all corrupted messages, e.g., using cyclic redundancy checks (CRCs) [Kiz05].
12. On the level of MECHATRONICUML, we currently do not consider security issues like a man-in-the-middle attack [Kiz05] that can change the message's content. Moreover, we currently do not assume that the middleware handles such security issues.
13. We assume that the middleware must not change the incoming message buffer of a role (by trying to enqueue a new message into the buffer) between enabling and firing a transition of the receiver's RTSC.

” [*DPP+16]

3.4.4 Discussion

In this section, we discuss why we choose these QoS assumptions. In particular, we present and discuss the concepts by Hoffmann (see Section 3.4.4.1), by Hirsch (see Section 3.4.4.2), and by Tanenbaum (see Section 3.4.4.3). Afterwards, we explain in Section 3.4.4.4 that we chose our QoS assumptions only to enable a model checking of RTCs.

3.4.4.1 Concepts by Hoffmann

Hoffmann [Hof07] defines no implicit QoS assumptions for the asynchronous message exchange in MECHATRONICUML but several explicit QoS assumptions that the software engineer shall state: (1) network kind (Ethernet, Wireless LAN, FireWire, CAN), (2) overtaking of messages (true/false), (3) simultaneous sending of messages enabled (true/false), (4) collision of messages possible (true/false), (5) minimum and maximum message transmission delay in ms, (6) bandwidth (maximum number of messages in transit at the same time), and (7) failure types (crash, receive omission, send omission, timing failure, response failure, arbitrary failure). As we distinguish between an application layer and a middleware, most of his properties and failures are not necessary for us as they belong to the middleware rather than to the application layer. In particular, our distinction is as follows:

- We abstract from the specific network kind as this is a platform-specific detail.
- We abstract from the fact whether the messages may overtake each other. Instead, we are only interested whether the order in the incoming message buffer is preserved or not.
- We abstract from the collision of messages and from the failure types as we are not interested in the details of the transmission but only whether the connector is reliable and, therefore, whether message loss is possible or not.
- We do not provide the option to define a minimum message transmission delay as we did not find existing platforms that can guarantee this.
- We assume that the bandwidth is no limiting factor from the perspective of the application layer. This assumption is necessary as the bandwidth depends on the chosen platform.

3.4.4.2 Concepts by Hirsch

Hirsch [Hir08] also does not state implicit QoS assumptions concerning the asynchronous message exchange in MECHATRONICUML but states that the software engineer shall specify explicit QoS assumptions like the reliability and the minimum and maximum message transmission delay. As explained before, our concept does no longer provide the option for the minimum message transmission delay. Moreover, we additionally enable that the software engineer can state whether the message order shall be preserved or not as we identified use cases and platforms for both variants.

3.4.4.3 Concepts by Tanenbaum

Tanenbaum [Tan10] states that each layer is based on services that can be characterized by its *reliability*. He defines this term as follows: “[...] services are reliable in the sense that they never lose data” [Tan10, p. 36]. We also enable the software engineer to define the reliability and reuse Tanenbaum’s definition of this term. Furthermore, Tanenbaum defines four explicit QoS assumptions (parameters) that characterize the needs for each packet stream in the network layer (the network layer “[...] is concerned with getting packets from the source all the way to the destination” [Tan10, p. 355]). His explicit QoS assumptions are as follows: (1) *delay* defines the time a packet needs from source to destination, (2) *jitter* defines “[t]he variation (i.e., standard deviation) in the delay or packet arrival times” [Tan10, p. 406]), (3) *loss* defines the amount of messages that are lost during transmission, and (4) *bandwidth* defines the rate of the data transfer. Each of his four QoS assumptions may be categorized via the values low, medium, and high. A comparison with our explicit QoS assumptions is as follows:

- We combine the *delay* and the *jitter* as we are only interested in the maximum message transmission delay. When verifying RTCPs via DSMC (see Chapter 5), this information is sufficient as the model checking will evaluate all possible transmission delays between zero and the maximum.
- We abstract from the amount of messages that are lost during transmission (e.g., given in a percentage) as we are only interested in the fact whether messages may get lost or not. Again, the reason for this is that the model checking will verify for each message what happens if it gets lost or not. Therefore, a percentage or another range of values besides true or false (e.g., low, medium, and high) would be ignored by the model checker.
- As stated before, we assume that the bandwidth (the number of messages that may be in transit at the same time) is no limiting factor if the connector is reliable and that we do not assume a specific bandwidth if the connector is unreliable. Therefore, we do not need the bandwidth as an explicit QoS assumption of an RTCP.

3.4.4.4 Distinction

In this thesis, we only focus on the formal verification of RTCPs using (domain-specific) model checking. Therefore, alternative verification forms like the (hybrid) simulation of a CPS [Hei15] or a statistical model checking [DLL+11] are not in our scope. These verification forms could require more explicit and implicit QoS assumptions. Moreover, they could require

or benefit from a more detailed range of value concerning the explicit QoS assumptions. For example, Heinzemann [Hei15] defines that its simulation of a MECHATRONICUML model requires a message loss probability if the connector is unreliable and a statistical model checking could verify performance aspects of an RTCP if the message loss probability is given.

3.5 Behavioral Specification

The concurrent execution of the role instances of an RTCP defines the execution behavior of the RTCP. The behavior of a role instance is stateful and is subject to real-time restrictions (e. g., the maximum time the system dwells within a certain state). Role instances of the same role adhere to the same behavior definition. The RTCP specifier may specify the variable parts of the role's behavior by an RTSC (see Section 2.2.2).

Role instances may only communicate with each other by exchanging asynchronous messages. These messages are typed over the message types declared in the message type specification of the role. Within the role's RTSC, the RTCP specifier may define at which point in time a role may/shall send or receive a message. We restrict the asynchronous message exchange by defining that the sending and the receiving role instance are typed over different roles. Consequently, asynchronous messages may not be used for the communication within one role instance (e. g., to communicate between several regions of the role's RTSC). The reasons for this definition is that we want to avoid too complex models, i.e., that each region of a role instance requires an own buffer and that each communication between two regions requires a definition of the QoS assumptions. In addition, a model with large numbers of asynchronous message exchanges would significantly increase the state space, which could easily lead to a verification that cannot complete its task (see Section 3.3.6).

We define that a role's RTSC must be able to send and consume all messages that the role may send and receive, i.e., for each message m that a role may send / receive, there exists at least one transition in the RTSC that sends / consumes m . The purpose of this definition is to enforce a meaningful behavior definition. Moreover, we define that all role instances start at the exact same time, i.e., all clocks are zero when the RTCP instance starts running.

An RTSC of a role may define (1) variables to store information concerning the coordination (e. g., the current physical position of the system), (2) operations to encapsulate statements (e. g., to calculate the current position of the system), and (3) continuous running clocks (e. g., to detect a timeout). A role instance and, therefore, its RTSC can access its defined variables, operations, and clocks, i.e., it may read / write the variables, call the operations, and read / reset the clocks. However, an RTSC of a role instance cannot access the variables, operations, and clocks of another role instance. In addition, each RTSC may use the keyword `maxMsgDelay` (see Section 3.3.4) within all of its time value expressions, i.e., invariants, clock constraints as well as relative and absolute deadlines. By providing this keyword, we aim to raise the maintainability as well as the understandability of the RTSC because the RTCP specifier does not have to state the specific maximum message delay expression multiple times but only once.

An RTCP (and an RTSC of a role in particular) may have non-deterministic behavior as we identified realistic examples where the RTCP does not require a deterministic behavior. For example, in the RTCP Overtaking, the behavior of role Overtaker is non-deterministic,

e.g., it may send an overtaking request at any point in time as long as the overtaking is not active and no other request is pending. We realize this by using non-urgent transitions. As another example, the role *Overtakee* has a non-deterministic behavior as well, e.g., it non-deterministically decides whether it accepts or declines the overtaking request. We realize this by using the non-deterministic choice operator (see Section 2.2.2). Allowing non-deterministic behavior has two advantages: First, the RTCP specifier does not have to reveal internal algorithms that may be seen as intellectual property. Second, we increase the reusability of the RTCP as different components may refine this non-deterministic behavior differently.

We already defined the behavior of an asynchronous message communication (dispatching, transmitting, buffering, consuming) in Section 3.4.2. Among others, we define that the transitions of the RTSC's specify when to dispatch and when to consume which message. Additional behavior specifications for incoming message buffers and QoS assumptions, e.g., using RTSCs, are not required because all variable parts of their behavior may be specified using the given properties (see Section 3.3.2 and 3.3.4). Noteworthy, we specify the semantics of incoming message buffers and the QoS assumptions in detail in Section 5.4.13 by defining their transformation to UPPAAL timed automata.

The specification of RTSCs for single and multi roles have much in common but differ in some aspects. In the following, we begin with explaining the behavior specification of single roles in Section 3.5.1 and explain the RTSCs for the one-to-one RTCP Overtaking. Afterwards, we explain the features and constraints of the behavior specification of one-to-many RTCPs in Section 3.5.2 and use the behavior specification for role *Overtaker* of RTCP *MultiOvertaking* as our running example.

3.5.1 Single Role Behavior Specification

We define the behavior of a single role by an RTSC without further restrictions compared to the ones given at the beginning of Section 3.5 and the default restrictions of an RTSC. The latter ones are given briefly in Section 2.2.2 and in detail in the MECHATRONICUML specification for platform-independent modeling [*DPP+16] (e.g., an RTSC must have exactly one initial state).

Two examples for an RTSC of a single role are given in the one-to-one RTCP Overtaking in Figure 3.6. Note that the RTCP Overtaking is a variant of the RTCP *SimpleOvertaking*, which we used in Section 2.2.2 to introduce the RTSC formalism. In contrast to the RTCP *SimpleOvertaking*, the RTCP Overtaking additionally defines incoming message buffers and QoS assumptions. Moreover, it uses an additional message and alters the behavior definition (e.g., it even works for an unreliable connector). In the remainder of this thesis, we use the RTCP Overtaking as our running example. Therefore, we informally explain in the following the behavior of this RTCP in detail.

At first, both roles are in state *NoOvertaking.Idle* while their clocks start at zero. Due to its non-urgent transition, the *Overtaker* may at any point in time request to overtake the *Overtakee* by sending the message *request* including the planned overtaking speed as a parameter. While sending, the *Overtaker* resets the clock *c* when entering the state *WaitForAnswer* and awaits an answer.

As soon as the message *request* arrives in the buffer of role *Overtakee*, the role will immediately consume this message while firing the transition to state *EvaluatingRequest*.

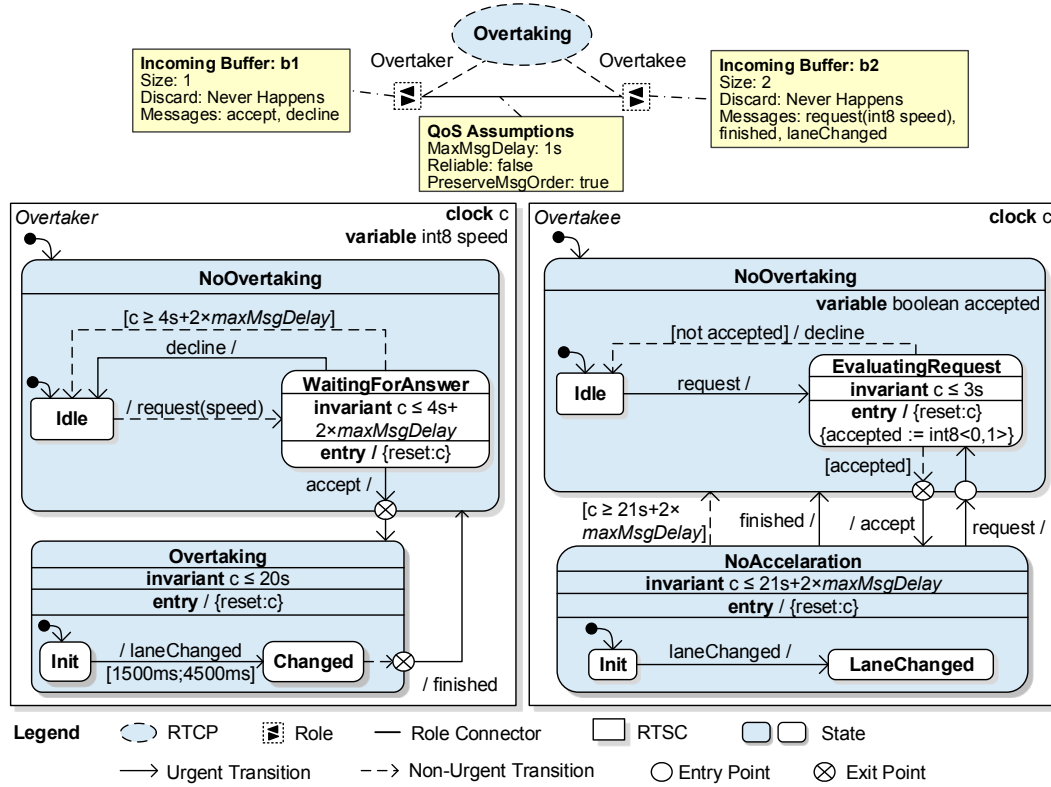


Figure 3.6: RTCP Overtaking (Structure and Behavior)

Hereby, the role's reaction is immediate (without further time delay) because the transition is urgent. Then, the Overtakee may at any point in time answer this request but not later than 4 seconds plus two times the maximum message delay after entering the state due to the given state invariant. Concerning its answer, the role may non-deterministically choose to accept or decline the request by either sending an accept or a decline message. In the RTCP, we specify the evaluation of this request via a non-deterministic choice expression. When the RTCP shall be applied to a component, this expression needs to be exchanged by the actual computation.

If the Overtakee does not accept the message (i.e., the variable `accepted` is false), then it sends the message `decline` and switches back to state `Idle`. If the Overtaker receives and consumes this message, it will switch back to state `Idle` as well and may start a new overtaking request, e.g., with an altered overtaking speed.

If the Overtakee accepts the request (i.e., the variable `accepted` is true), then it sends the message `accept` and switches to state `NoAcceleration.Init`. As soon as this state is active, the Overtakee assures to drive with a constant speed, i.e., accelerating is forbidden as this would increase the overtaking time and decelerating is forbidden as well such that the Overtaker can drive within the slipstream of the Overtakee to accelerate faster and to minimize the time on the other lane. As soon as the Overtaker receives the message `accept`, it immediately consumes this message, switches to `Overtaking.Init`, and starts the overtaking by accelerating to the planned speed and changing the lane. Then, the Overtaker has to announce that it changed the lane completely by sending a `laneChanged` message. This must happen within 1500 to 4500 milliseconds due to the deadline at transition `Overtaker.Overtaking.Init` to `LaneChanged`.

(we assume that changing the lane within this time is always possible for simplicity reasons). From this point on, the Overtakee may decelerate its speed again, but may still not accelerate until the message finished by the Overtaker announces that the overtaking has been carried out. This is still necessary to avoid a possible accident between the cars. Due to the invariant of state *Overtaking*, the Overtaker has to send the message finished at most 20 seconds after the overtaking started (again, we assume that this is always possible for simplicity reasons).

Noteworthy, the RTCP specifier has to add various model elements to both RTSCs as the QoS assumption of the RTCP defines that the connector is considered unreliable. In particular, the RTSC of role Overtaker requires the state invariant in state *NoOvertaking.WaitingForAnswer* and the transition from state *WaitingForAnswer* to *Idle*. Both are necessary in order to detect whether the request message of the Overtaker or the answer of the Overtakee (i.e., either the message accept or decline) were lost during transmission. If the Overtaker does not get an answer within six seconds ($4s + 2 \times \text{maxMsgDelay} = 4s + 2 \times 1s = 6s$), then it is no longer possible that an answer will arrive. As a consequence, the Overtaker may switch back to state *Idle* and may send a new message. Moreover, the RTSC of role Overtakee requires a transition from state *NoAcceleration* to state *NoOvertaking.EvaluatingRequest* that may fire as soon as the message request may be consumed from the buffer. This transition is necessary for the following scenario: The message *request* successfully arrives at the Overtakee. Then, the Overtakee accepts the request and switches to state *NoAcceleration.Init*. However, the message accept gets lost during transmission. Therefore, the Overtaker may (immediately) send a new request while the Overtakee is still in state *NoAcceleration.Init*. Via the additional transition, the Overtakee may consume the message request, switch back to state *NoOvertaking.EvaluatingRequest*, and may evaluate the request once again. Note, that it is necessary that the target of this new transition is state *NoOvertaking* and not state *NoAcceleration* as the proposed overtaking speed may be different within both requests and the environment may have already changed such that an overtaking is no longer appropriate and, therefore, acceptable by the Overtakee. In addition, it is possible that the message *finished*, which is sent by the Overtaker gets lost. In order to prevent that the Overtakee will stay in state *NoAcceleration*, we add a timeout to the state *NoAcceleration*. The RTCP specifier may realize this by adding an appropriate state invariant and an entry action that resets the clock *c*. Moreover, the RTCP specifier has to add an appropriate non-urgent transition from state *NoAcceleration* to state *NoOvertaking* that can only fire if the message *finished* may no longer arrive. Further modifications to the RTSCs are possible but not mandatory concerning the safety of the overtaking. For example, it may happen that the message *laneChanged* is lost during transmission. As a consequence, it may happen that the Overtakee stays in state *NoAcceleration.Init* for the complete overtaking.

3.5.2 Multi Role Behavior Specification

The behavior specification of a multi role has to describe the behavior of its multi role instance including all its subrole instances. For example, the Overtaker role of the one-to-one RTCP *Overtaking* only sends exactly one request message to the Overtakee and awaits exactly one accept message. Instead, the Overtaker role of the one-to-many RTCP *MultiOvertaking* shall send the message *request* via its subroles to all Overtakees and awaits an answer from all Overtakees whereby each subrole of the Overtaker receives one answer. As a consequence,

the specification is getting more complex as we have to define the behavior of the subroles and the dependencies between them as well.

Hirsch [Hir08] defines that all subrole instances shall have a similar behavior in order to reduce the effort to specify an RTSC for each possible subrole instance, e.g., if the upper limit of the multi role multiplicity range is ten, he wants to avoid that the RTCP specifier has to model ten additional RTSCs. Due to this definition, he proposes that all subrole behavior are based on one common template. In addition, Hirsch proposes that the RTCP specifier does not only has to model the subrole template but also an additional RTSC that synchronizes the behavior of the instantiated templates. As a consequence, this behavior definition is valid for an arbitrary number of subroles as all subroles share the same template. However, the number of required model elements (e.g., states and transitions) of a multi role compared to the behavior definition of a single role is still significantly higher. For example, our RTSC of the single role Overtaker of the one-to-one RTCP Overtaking requires six states and eight transitions (see Figure 3.6). In contrast to this, the RTSC of the multi role Overtaker of the one-to-many RTCP MultiOvertaking that applies the concept of Hirsch requires 23 states and 27 transitions (see Figure 3.17). Based on our experience, this significant difference increases the required modeling time enormously and makes the resulting RTSC hard to understand. In this thesis, our goal is to reduce the number of required model elements (especially the number of states and transitions) for modeling the behavior of a multi role. As a consequence, we expect that both the development time and the understandability are improved (though, we do not evaluate these expected effects).

To achieve this goal, we introduce a new concept that we call *one-to-many communication schema* in Section 3.5.2.1. Afterwards, we explain two exemplary one-to-many communication schemata: multicast (in Section 3.5.2.2) and multireceive (in Section 3.5.2.3). Next, we show in Section 3.5.2.4 how we apply these two schemata in order to model the behavior of the multi role Overtaker of RTCP MultiOvertaking in a compact manner. Then, we introduce four additional one-to-many communication schemata in Section 3.5.2.5 that we define for RTCPs before we present in Section 3.5.2.6 three candidates for one-to-many communication schemata that we identified during our related work study but that we assess as unsuitable for RTCPs.

In our opinion, an RTCP specifier will from now on model the behavior of a multi role using our one-to-many communication schemata and, therefore, without specifying an explicit subrole RTSC as the model is more compact, which makes it—in our opinion—faster to specify and easier to understand. However, our one-to-many communication schemata do not cover the complete expressiveness compared to the variant with an explicit subrole RTSC. As a consequence, we still enable to model the multi role behavior in this way. Compared to the definition of Hirsch [Hir08], we slightly altered the modeling language for specifying the multi roles in this way. We describe this in Section 3.5.2.7. Afterwards, we explain in Section 3.5.2.8 how we automatically normalize a multi role RTSC that uses one-to-many communication schemata to a multi role RTSC that uses an explicit subrole RTSC. Finally, we present in Section 3.5.2.9 the normalized RTSC of multi role MultiOvertaking.Overtaker as an example for our normalization.

3.5.2.1 One-To-Many Communication Schemata

By studying several existing multi role specifications [Hir08; Rie11; San11; Brö11; *BDG+14a] that apply the concepts of Hirsch [Hir08], we identified that an RTCP specifier typically has to realize six different one-to-many communication sequences. One exemplary sequence is that a multi role instance wants to send an instance of a message via all subrole instances to all instances of the single role. We realized that each communication sequence results in non-trivial RTSCs, where an RTCP specifier has to model the same states and transitions for each communication sequence again and again using a fixed schema. Therefore, our idea is to abstract from reoccurring modeling idioms by introducing new language keywords. In particular, we provide so-called one-to-many communication schemata that define the semantics of a certain communication sequence. As a consequence, the RTCP specifier only has to define for each message being sent and received by a multi role which one-to-many communication schema shall be applied instead of modeling additional states and transitions.

In particular, our concept is similar to the so-called routing algorithms [Tan10] that describe the delivery semantics in network topologies. Though, our one-to-many communication schemata focus on the communication at the application layer instead of the network layer, e.g., we abstract from IP addresses and the number of hops between the sender and the receiver. Nevertheless, two of our one-to-many communication schemata are based on two routing algorithms, e.g., our example given above that a multi role instance wants to send an instance of a message via all subrole instances to all instances of the single role is based on the routing algorithm *multicast*.

We define that a one-to-many communication schema has an explicit name. In particular, their names are: *multicast*, *multireceive*, *unicast*, *singlereceive*, *iterate*, and *loadbalancing*. According to the authors of several design patterns [GHJV95; BMR+96], having explicit names for a design solution is an advantage as it defines in one single word the design solution of a communication sequence. Therefore, as soon as the RTCP specifier learns the names and the semantics of the one-to-many communication schemata, he can understand the design of a given RTCP significantly faster compared to the concept of Hirsch because he does not need to understand which behavior a given set of states and transitions realizes. Moreover, if the name of the one-to-many communication schema becomes part of the design language of the RTCP specifiers, then it will facilitate the communication between the engineers as well.

We distinguish our one-to-many communication schemata whether their message is part of the transition's condition (i.e., the message shall be received and consumed) or part of the transition's effect (i.e., the message shall be sent). If it is part of the condition, then the RTCP specifier can choose between two one-to-many communication schemata: *multireceive* and *singlereceive*. If it is part of the effect, then the RTCP specifier can choose between four one-to-many communication schemata: *multicast*, *unicast*, *iterate*, and *loadbalancing*.

By examining the existing multi role models, we realized that a subrole may need its own variables, clocks, and operations. For example, in the RTCP Overtaking, each subrole should be able to store within an own variable whether the overtaking request was accepted or not. Therefore, we introduce for variables, clocks, and operations specified in multi role RTSCs a new boolean property called *subrole-specific*, which indicates whether each subrole instance has its own instance of a variable/clock/operation. Hereby, the initialize expression of a

subrole-specific variable as well as the body of a subrole-specific operation may reference subrole-specific and non-subrole-specific variables and operations.

Concerning the namespace, if a variable/clock/operation is subrole-specific, then each subrole instance has exclusive access to its instance of this variable/clock/operation. However, if a variable/clock/operation is not subrole-specific, then only a single instance of this variable/clock/operation exists and the multi role instance including all subrole instances have access to this instance.

In order to access the subrole-specific variables, clocks, and operations, we introduce attributes for each one-to-many communication schema that the RTCP specifier may use. Only these attributes may reference or alter the subrole-specific variables, clocks, and operations. For example, for each one-to-many communication schema, we provide—among others—two subrole-specific attributes for specifying subrole-specific conditions and effects. Here, each subrole has to evaluate its subrole-specific condition and has to execute its subrole-specific effect. Consequently, a transition with one-to-many communication schemata may define subrole-independent and subrole-specific conditions and effects. Noteworthy, we also define attributes for one-to-many communication schemata that do not access a variable/clock/operation, but that alter the behavior of the schemata directly. Most of our attributes may be expressed using MECHATRONICUML’s action language. Thus, they allow a rich set of expressions.

Besides the attributes of a schema, we also allow that a subrole-specific variable/clock/operation may be referenced within the parameter bindings of a message that shall be sent. This is possible and useful, as the parameter binding is an expression that is evaluated within each subrole instance.

In summary, the RTCP specifier does not need to model the RTSC of the subrole at all but only has to define which one-to-many communication schema with which attributes shall be applied. This is a huge improvement concerning the number of model elements compared to the classic behavior specification of multi roles. For example, using one-to-many communication schemata, we only need seven states and eight transitions for modeling the multi role Overtaker. Thus, we only need one more state compared to the behavior specification of the single role Overtaker but 16 states and 19 transitions less than the classic specification.

Similar to the description of design patterns [GHJV95; BMR+96], we define each one-to-many communication schema in a consistent manner using a description template. Our template consists of the following aspects: First, we give a description of the one-to-many communication schema concerning its abstract syntax and semantics. Second, we give a concrete example in the domain of CPSs for applying this one-to-many communication schema. Third, we define the concrete syntax of the schema that we use within an RTSC diagram. Finally, we state alternative schemata and with which other schemata this schema may be combined. By having these descriptions, we are convinced that the RTCP specifier can understand the meaning of each one-to-many communication schema and can compare the one-to-many communication schemata with each other.

3.5.2.2 Example 1: Multicast

In the following, we explain the one-to-many communication schema multicast using our description template that we introduced before.

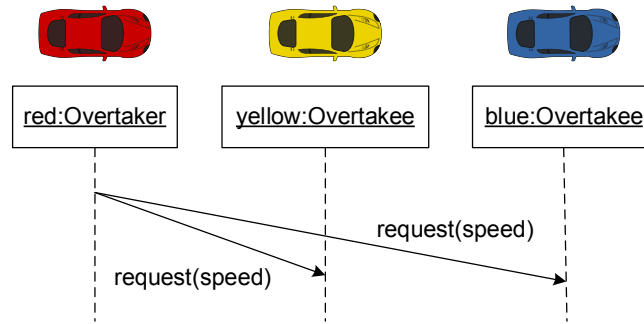


Figure 3.7: Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Multicast

Description The one-to-many communication schema *multicast* enables the multi role instance to send the same message to (a subset of) all instances of the single role. In particular, the parameters of each message may vary and the sending shall happen within the same time frame, i.e., no clock progresses. The subrole instances are always visited in ascending order.

Moreover, this one-to-many communication schema enables to define that not all subrole instances have to send this message but only these that fulfill a certain subrole condition. Defining such a condition may reduce network traffic (depending on the platform-specific model) and may reduce the workload on the receiver side. An RTCP specifier may define this via the attribute condition that contains textual expressions of MECHATRONICUML's action language (see Section 2.2.2). As an important fact, the transition that specifies a multicast with a subrole condition may only fire if at least one subrole instance fulfills the stated subrole condition. Otherwise, the transition is not enabled. If no subrole instance fulfills its subrole condition, then the RTSC has to wait a fixed time before it may retry to evaluate the subrole instances again, i.e., we implement a polling mechanism. The RTCP specifier has to define the time span between two tries using the attribute *retryAfter*. It must be defined if a condition is defined and has to be greater than zero.

In addition, this schema enables to define a subrole action that shall be executed by each subrole after sending the message. An RTCP specifier may define this via the attribute *action* that contains textual expressions of MECHATRONICUML's action language as well.

In summary, the attributes *condition* and *action* are optional attributes. The attribute *retryAfter* must be defined if and only if the attribute *condition* is defined.

Example In our overtaking scenario, the overtaking car may send a request to all cars in front of it if it may overtake. We depict this using a sequence diagram in Figure 3.7. Here, the overtaker sends instances of the same message with the same parameter value to all overtakes. We explain this example in detail in Section 3.5.2.4.

Concrete Syntax If the one-to-many communication schema multicast is applied, then we extend the transition label as follows. At first, we append the string '->multicast' to the name of the message. Then, if one of the three attributes is defined, we extend the

string by round parentheses. The attribute condition is given first with the prefix 'if:', the attribute `retryAfter` is given next with the prefix 'retryAfter:', and the attribute action is given last with the prefix 'do:'. If more than one attribute is used, we separate them with a pipe. As an example, we write `m1->multicast(if: <conditionExpr>|retryAfter: <retryAfterExpr>|do: <actionExpr>)` for a message `m1` that shall be sent via a multicast and that has the condition `<conditionExpr>`, the retry after delay `<retryAfterExpr>`, and the action `<actionExpr>`.

Alternative Schemata If the message is only relevant for one instance of the single role, then use either unicast or loadbalancing as this may reduce the network traffic (depending on the realization of the multicast). If the multi role instance shall not send all messages at the same time but with a delay after sending each message, then use `iterate`.

Combinability A typical combination is that the multi role sends a multicast first and awaits either an answer by all instances of the single role via a `multireceive` or by exactly one instance of a single role via a `singlereceive`.

3.5.2.3 Example 2: Multireceive

In the following, we explain the one-to-many communication schema `multireceive` using our description template that we introduced before.

Description The one-to-many communication schema *multireceive* enables the multi role instance to consume instances of the same message type from (a subset of) all instances of the single role. In particular, the consuming of the received messages shall happen within the same time frame, i.e., without further clock progresses. The subrole instances are always evaluated in ascending order. Optionally, this one-to-many communication schema enables to define that not all subrole instances have to consume a message but only these that fulfill a certain subrole condition. An RTCP specifier may define this via the attribute condition that contains textual expressions of MECHATRONICUML's action language (see Section 2.2.2).

Furthermore, the RTCP specifier may define a subrole-specific action that shall be executed by each subrole after receiving the message. For doing this, we define the attribute action that contains textual expressions of MECHATRONICUML's action language as well.

If the subrole shall consume a message of a certain message type and its subrole condition is fulfilled, then it should typically be the case that a message of this type is available at the head of the buffer. If such a message is not available, then we consider this as a failure. In order to handle such a failure, we enable the RTCP specifier to specify an action that differs from the one that is executed if the message is available. For doing this, the schema provides an additional attribute called `failureAction` that contains textual expressions of MECHATRONICUML's action language.

In summary, the attributes `condition`, `action`, and `failureAction` are optional attributes. The attribute `retryAfter` must be defined if and only if the attribute `condition` is defined.

Example In our overtaking scenario, the overtaker awaits an answer by all overtakees whether the overtaking request is accepted. This may be realized using the schema `multireceive`.

We depict this example using a sequence diagram in Figure 3.8. Here, both overtakeses send an instance of the same message type `answer` to the overtaker. We explain this example in detail in Section 3.5.2.4.

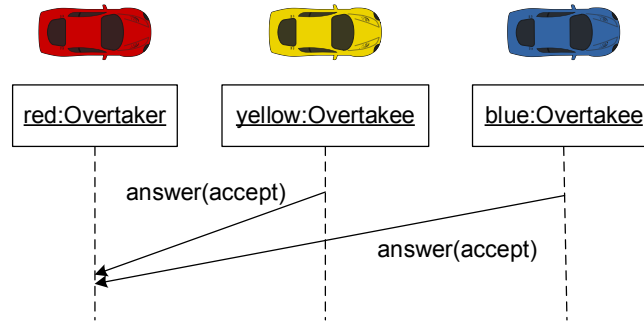


Figure 3.8: Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Multireceive

Concrete Syntax If the one-to-many communication schema multireceive is applied, then we extend the transition label as follows. At first, we append the string `'->multireceive'` to the name of the message. Then, if one of the three attributes is defined, we extend the string by round parentheses. The condition attribute is given first with the prefix `'if: '`, then the action attribute is given next with the prefix `'do: '`, and the failureAction attribute is given last with the prefix `'failureAction: '`. If more than one attribute is used, we separate them with a pipe. As an example, we write `m1->multireceive(if: <conditionExpr>|do: <actionExpr>|failureAction: <failureActionExpr>)` for a message `m1` that shall be sent via a multireceive and that has the condition `<conditionExpr>`, the action `<actionExpr>`, and the failureAction `<failureActionExpr>`.

Alternative Schemata If it is sufficient that only one instance of the single role answers, then the RTCP specifier should use the schema `unireceive`.

Combinability A typical combination is that the multi role sends a multicast first and awaits an answer by all instances of the single role via a multireceive.

3.5.2.4 Applying the One-To-Many Communication Schemata to the RTCP MultiOvertaking

We show the applicability of our concept by specifying the behavior of the one-to-many RTCP MultiOvertaking, which we depict in Figure 3.9. It is based on the behavior definition of the one-to-one RTCP Overtaking (see Figure 3.6). We highlight the differences to the behavior of RTCP Overtaking in red.

First of all, we change the upper bound role multiplicity range of role Overtaker from 1 to 2. Therefore, Overtaker is now a multi role. Then, we start to assign one-to-many communication schemata to each transition of the multi role RTSC that either sends or receives a message. At first, the message `request` shall be sent to all instances of role Overtakee. Therefore, we assign the one-to-many communication schema multicast to this message.

Then, we decide that the Overtakee shall wait for all answers of all instances of role Overtakee. Therefore, we want to use the one-to-many communication schema multireceive.

However, the schema requires that all received messages have the same type. Therefore, we adapt the messages that the roles may exchange by replacing the messages accept and decline with the new message answer and the parameter accept of type boolean. As a consequence, we adapt the role Overtakee at two transitions: we replace the sending of message decline by the message answer that binds the value false to the parameter accept and we replace the sending of message accept by the message answer that binds the value true to the parameter accept. Concerning the RTSC of the Overtaker, we adapt the states and transitions within the composite state NoOvertaking due to the replacement of the messages. In particular, we remove the two transitions from WaitingForAnswer to Idle, add an urgent state AnalyzeResult, add a new variable allAccept of type boolean, and connect this new state with two new

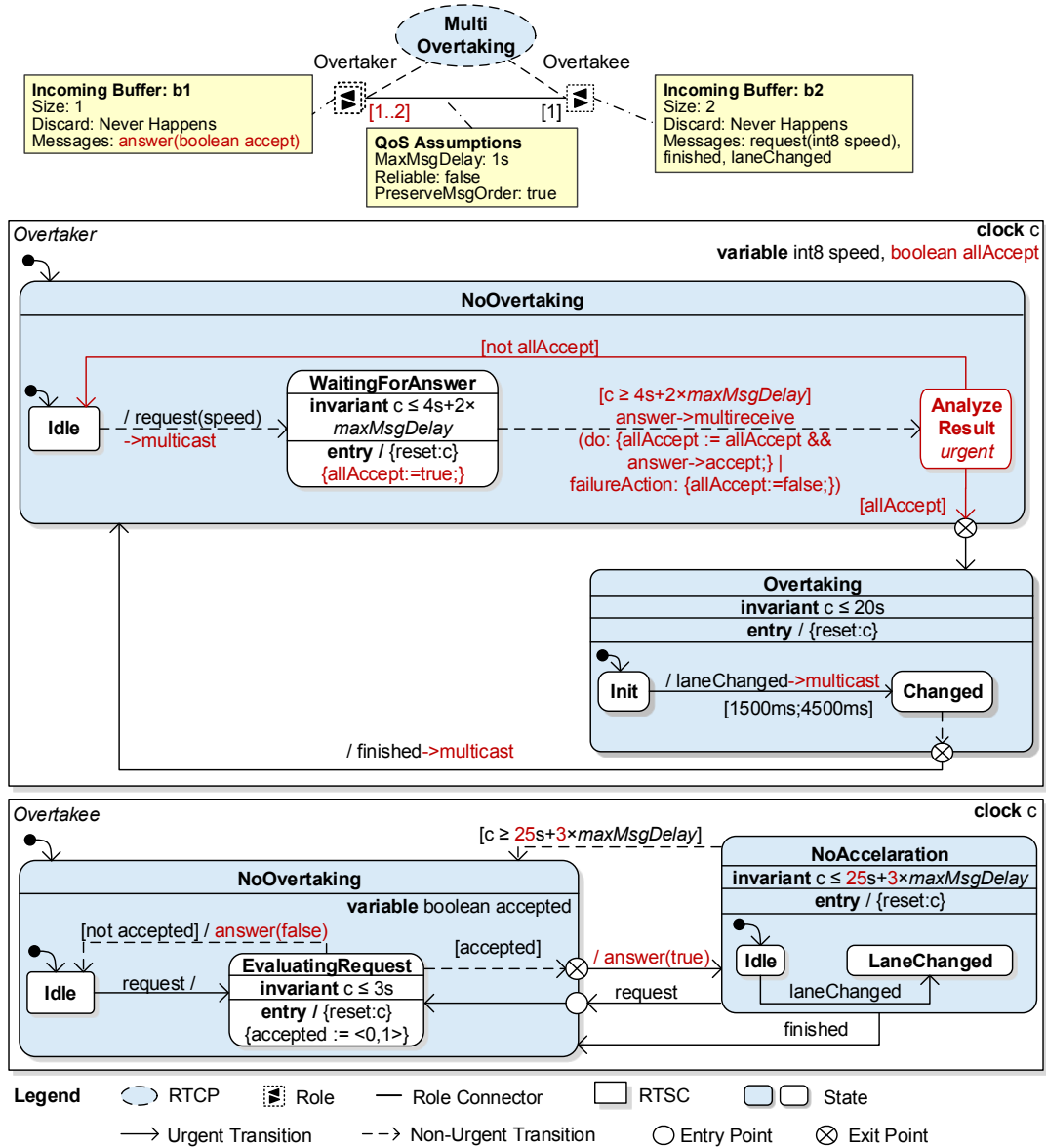


Figure 3.9: RTCP MultiOvertaking (Differences to RTCP Overtaking are Highlighted in Red)

transitions (to the state `Idle` and the state exit point) and the already existing transition that had the state exit point as target.

Subsequently, we add an entry action to state `WaitingForAnswer` that sets the variable `allAccept` to true. Then, we define that the transition to `AnalyzeResult` additionally defines a `multireceive` for the message `answer`. We add a subrole-specific action to this schema that sets the variable `allAccept` to false if any of the `Overtakees` returns that it does not accept the request. Moreover, we define a subrole-specific failure-action that sets the variable `allAccept` to false if any `Overtakee` does not answer in time.

Furthermore, we define that the transition from `WaitingForAnswer` to `AnalyzeResult` may not fire before all messages should be received via the additional clock constraint. As a first consequence, the transition becomes non-urgent as urgent transitions may not have a clock constraint. As a second consequence, we have to adjust the role `Overtakee`: both, the invariant of state `NoAcceleration` and the clock constraint at the transition from `NoAcceleration` to `NoOvertaking` have to be increased from $21s + 2 * \text{maxMsgDelay}$ to $25s + 3 * \text{maxMsgDelay}$. Otherwise, the state `Overtakee.NoAcceleration` may be left while the state `Overtaker.Overtaking` is still active. If the `Overtaker` is in state `AnalyzeResult` and receives a positive answer from all instances of role `Overtakee`, then we enforce the role to switch to state `Overtaking`, otherwise we enforce it to switch back to state `Idle`.

Lastly, we also define that sending the messages `laneChanged` and `finished` by the role `Overtaker` shall use the one-to-many communication schema `multicast` as well.

3.5.2.5 Additionally Identified One-To-Many Communication Schemata

In the following, we briefly present the four remaining one-to-many communication schemata that we also assess as suitable for RTCPs but that are not part of our overtaking example. We explain these four in detail in Appendix A.

Unicast sends an instance of a message type to exactly one instance of the single role. As a mandatory task, the RTCP specifier has to define a condition that a subrole instance must fulfill if it wants to send the message. To be deterministic, `unicast` iterates all subrole instances in ascending order. The first subrole instance that fulfills the condition sends the message. Furthermore, the RTCP specifier may define an action that the subrole instance shall execute after it sends the message. We explain this one-to-many communication schema in detail in Appendix A.3.

Singlereceive consumes exactly one instance of a message type that one of the subrole instances received even if multiple subrole instances have an instance of this message type in their buffer. The RTCP specifier may define a condition that the subrole instance has to fulfill in order to consume the message and an action that the subrole instance shall execute after it consumed the message. We explain this one-to-many communication schema in detail in Appendix A.4.

Iterate sends an instance of a message type with the same parameters to (a subset of) all instances of the single role. In particular, the sending does not need to happen within the same time frame, i.e., clocks may progress. Therefore, the RTCP specifier may define a delay between sending two messages. In addition, the RTCP specifier may define the following four properties: (1) a condition that a subrole instance has to fulfill

in order to send the message, (2) an action that each subrole instance shall execute after it sends its message, (3) a condition that forces the iteration to terminate even if not all subrole instances sent their message, and (4) whether the iteration shall iterate the subrole instances in ascending or descending order. We explain this one-to-many communication schema in detail in Appendix A.5.

Loadbalancing sends an instance of a message type to exactly one instance of the single role but considers the current workload of the single role instances. The schema provides two variants for balancing the load. In the first variant, the RTCP specifier has to define an upper limit of the working time of an instance of the single role. Then, the schema is able to calculate whether a specific single role instance is still working or not. In the second variant, the RTCP specifier has to define a response message that the single role instance will send after it completed its work. Therefore, the multi role instance is explicitly informed about the work load of the single roles. Moreover, the RTCP specifier may define a response action that is executed as soon as the subrole instance receives the response message. In both variants, the RTCP specifier may define a condition that a subrole instance has to fulfill in order to send the message and an action that the subrole instance shall execute after it sends its message. We explain this one-to-many communication schema in detail in Appendix A.6.

3.5.2.6 Unsuitable One-To-Many Communication Schemata

In the following, we present further one-to-many communication schemata that we identified during our related work study but that we assess as unsuitable for RTCPs. We briefly introduce these schemata and state why they are not suitable for us.

Broadcast In a *broadcast* at application-level [CDKB11], the sender (in our case the multi role) sends its message to all receivers (all single role instances) simultaneously. However, in contrast to the one-to-many communication schema *multicast*, the sender does not know its receivers or whether any receiver exists at all. According to our definition of one-to-many RTCPs, the multi role always knows its receivers. As a consequence, such a schema is not suitable for RTCPs.

Geocast The one-to-many communication schema *geocast* [GFZ13] delivers a message to a known set of receivers in a certain specific geographical location. Therefore, this one-to-many communication schema is a specific variant of a multicast. Currently, role instances of a one-to-many RTCP do not know the geographical location and have no language-specific keywords or operations. As a consequence, such a schema is not suitable for RTCPs.

Anycast The one-to-many communication schema *anycast* [BEH+97] enables that the sender (in our case the multi role instance) sends a message to a certain IP address that several receivers (single role instances) have in common. The network router decides which receiver is the topologically nearest and delivers it to this receiver. In RTCPs, we abstract from the IP address, network routers, and the network topology. As a consequence, such a schema is not suitable for RTCPs.

3.5.2.7 Multi Role Behavior Specification With Explicit Subrole RTSCs

As already stated in the beginning of Section 3.5.2, specifying the behavior of a multi role using an RTSC that uses one-to-many communication schemata slightly restricts the expressiveness of the multi role behavior specification compared to the classic specification by Hirsch [Hir08]. For example, when sending a multicast, we only enable to iterate all subrole instances in ascending order, e.g., it is not possible that the second subrole instance sends its message before the first subrole instance sends it. In our opinion, our restrictions concerning the expressiveness only affects rare use cases, though they are still restrictions. As a consequence, we still enable the RTCP specifier to specify the multi role behavior with an explicit subrole RTSC. Our concept is only similar to the concept of Hirsch but not identical. We present it in the following.

Similar to Hirsch, we define that “[t]he RTSC of the multi role [...] contains only one hierarchical state, which is the initial state. This state has no incoming or outgoing transitions. Moreover, this state contains exactly two regions, which are called coordinator and subrole.” [*DPP+16] As the name already suggests, the coordinator behavior coordinates the behavior of all subrole instances of the multi port, e.g., it defines when a multicast shall be executed. In contrary, the region subrole defines the behavior for the subrole instances, e.g., it defines when a subrole instance shall send a message. In particular, the subrole region is a template that we multiply n times as an additional orthogonal region to the coordinator region while instantiating the multi role, where n is defined by the role multiplicity. In particular, the complete RTSC of the region is multiplied, i.e., its states, transitions, variables, clocks, and operations.

Figure 3.10 shows an example for modeling and instantiating a multi role RTSC with an explicit subrole RTSC. On the upper left side, we show the RTCP MultiOvertaking including its roles Overtaker and Overtakee. We depict a simplified form of the multi role RTSC in the lower left (we show the complete version of this RTSC in Section 3.5.2.8). Here, the RTSC Overtaker has a single state called Overtaker_Main that is always active. This state contains the two regions coordinator and subrole that both define an RTSC. An instance of the RTCP MultiOvertaking that uses the role multiplicity of two is given in the upper right corner. Due to the role multiplicity, this instance has a multi role instance that contains two subrole instances. The corresponding instance of the RTSC is given in the lower right. Here, the subrole region is multiplied two times—one for each subrole instance.

In Figure 3.11, we give a more concrete example for a multi role RTSC that uses an explicit subrole region. The example defines an excerpt of the role Overtaker of RTCP MultiOvertaking (as stated before we show the complete version of this RTSC in Section 3.5.2.8). In particular, the given RTSC only covers the request for overtaking, which shall be realized by sending the message request to all instances of role Overtakee. The RTSC respects the constraints given above by defining only one top-level state called NoOvertaking_Main that is always active and that contains exactly two regions: coordinator and subrole. We will use this example to explain the remaining constraints and features when modeling the behavior with an explicit subrole region.

Typically, we use synchronization channels such that the coordinator and a subrole instance or that two subrole instances may synchronize their behavior without time delay (though, a variable-based communication works as well). For defining a synchronization with a specific subrole, we (Christian Heinzemann and myself) extended the expressiveness of

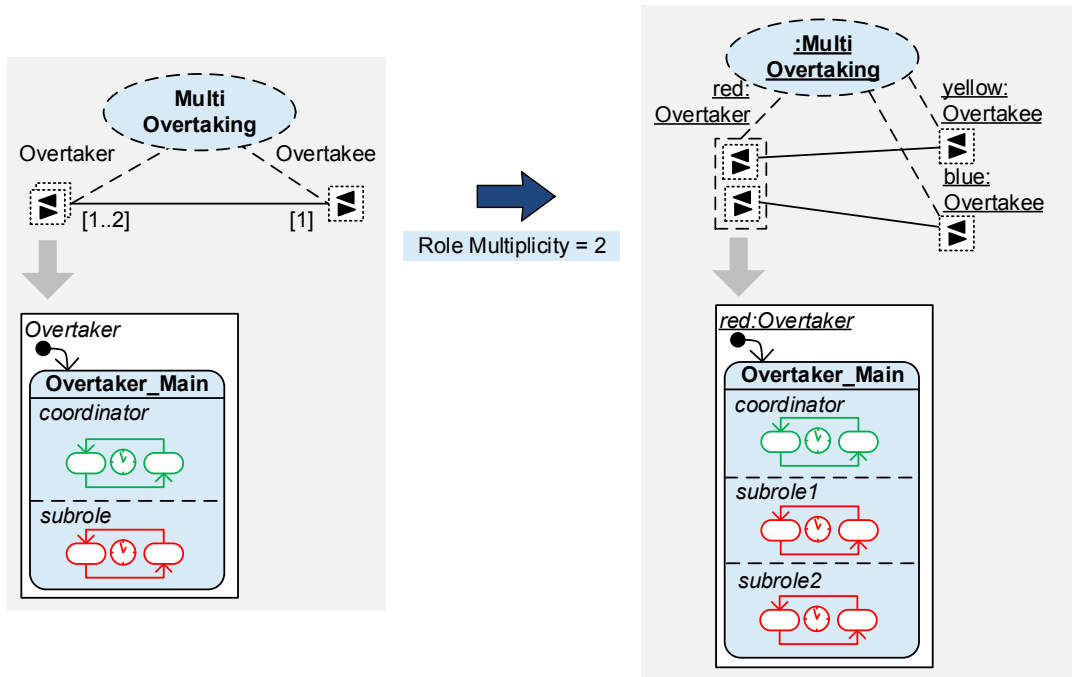


Figure 3.10: Example for Modeling and Instantiating a Multi Role RTSC With an Explicit Subrole RTSC

synchronization channels (see Section 2.2.2) in [*BDG+14b] by introducing selectors: “A synchronization with a selector extends a plain synchronization by an additional condition for the firing. Selectors are a generalization of UPPAAL’s synchronization channel arrays, which essentially are ordinary arrays containing synchronization channels. In the case of using a selector, the synchronization channel defines a type for the selector. Then, each synchronization using this synchronization channel needs to provide a selector expression that resolves to this type. [...] [S]ynchronizations with selectors may only synchronize if both, the sending and the receiving synchronization, provide the same value in their selector expression. If one or both selector expressions refer to null, then no synchronization is possible. We support the selector expressions of types Boolean and Integer. In case of multi roles, we additionally support to use selector expressions of type Role [...]” [*DPP+16]

We enable to refer to a subrole instance (e.g., within a selector expression) as follows: “The subrole instances of a multi role are always ordered in a list. As a result, there is one subrole instance, which is the first inside the multi role and one which is the last. For each subrole instance, we may also retrieve the next (or previous) one in the order. We exploit the order by defining additional keywords for referring to subrole instances within the specification of a multi role RTSC:

self This keyword may only be used inside the subrole RTSC. It refers to the subrole instance executing this RTSC. **self** will never refer to null.

first This keyword refers to the first subrole instance of the multi role. If no subrole is instantiated, the keyword refers to null.

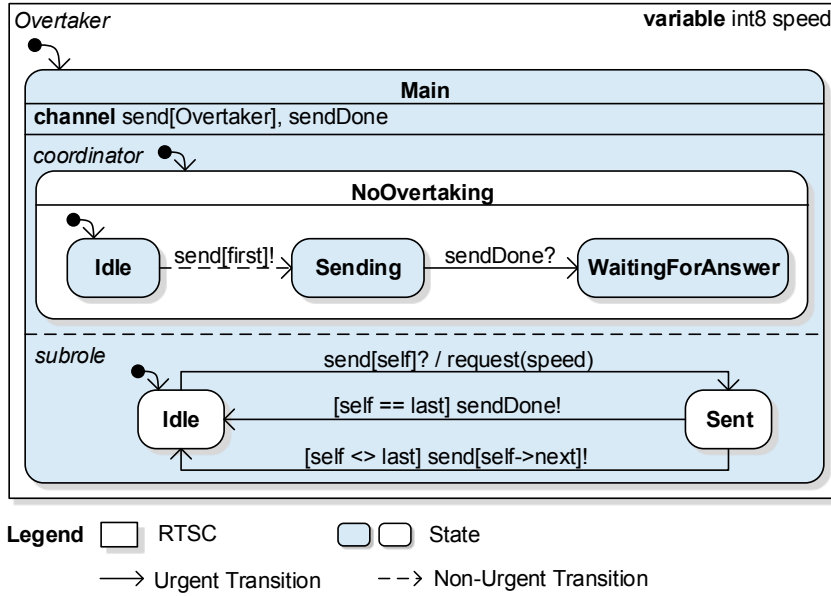


Figure 3.11: Simplified Version of RTSC Overtaker for RTCP Overtaking that Uses an Explicit RTSC for the Subrole

last This keyword refers to the last subrole instance of the multi role. If no subrole is instantiated, the keyword refers to null.

next This keyword must always be bound to a role, either specified by a variable or by another keyword. Examples are `role1.next` where `role1` is a variable of type `Role` or `self.next`. Then, the keyword refers to the subrole instance located after the subrole instance it is bound to. If the subrole instance that `next` is bound to is the last one in the multi role, `next` refers to null.

prev This keyword must always be bound to a role, either specified by a variable or by another keyword. Examples are `role1.prev` where `role1` is a variable of type `Role` or `self.prev`. Then, the keyword refers to the subrole instance located before the subrole instance it is bound to. If the subrole instance that `prev` is bound to is the first one in the multi role, `prev` refers to null.

” [*DPP+16]

As an example, the RTSC of the multi role *Overtaker* in Figure 3.11 defines the channel `send` that has the role *Overtaker* as its selector type. The transition from state `Main.coordinator.NoOvertaking.Idle` to `Sending` uses the keyword `first` as a selector expression for the channel `send`. As a result, the transition can only synchronize with the first subrole instance. As another example, the transition from `Main.subrole.Sent` to `Idle` in the subrole RTSC that synchronizes via channel `send` uses the keyword `next` bound to `self` in the selector expression to synchronize with the next subrole instance according to the ordered subrole instance list.

“Besides being used in a selector expression, the keywords may also be referenced in other expressions, e.g., in a guard `[expression]`, within a parameter binding when specifying

a message with parameters, or in the right side of an assignment. However, no value may be assigned to the keywords. For example, as shown in Figure 3.11, the keywords `self` and `last` have been used in the guard expression of the transitions from `Sent` to `Idle`. The guard `self<>last` defines that the transition may only fire if the subrole instance is not the last one, while the guard `self==last` defines that the transition may only fire if the subrole instance is the last one.” [*DPP+16]

If a specific subrole instance wants to synchronize with a transition of the coordinator and the subrole instance is the sender (indicated by “!”), then a selector is not necessary as the coordinator has exactly one instance. In our example, the subrole synchronizes via channel `sendDone` with the coordinator without using a selector. Though, if the subrole would also specify a transition with the synchronization expression `sendDone?`, then another subrole instance could be the second receiver of this synchronization (besides the coordinator). As no selector is used, this would result in a non-deterministic synchronization.

In contrast to our approach, Hirsch [Hir08] only enables to use integer-based selector expressions. Thus, each subrole instance requires a variable typed over a positive integer. This variable is denoted as k , where $k \in \mathbb{N}$ and $1 \leq k \leq n$ (n is the number of subrole instances). Therefore, if a coordinator wants to synchronize with a certain role, then it has to state the variable value of the subrole instance as its selector expression. We still support this modeling approach. Nevertheless, we are convinced that the usage of keywords that refer to a role is more appropriate as it avoids modeling idioms (specifying the variables k and n and defining their values) and as the RTSCs cannot influence the semantics of the keywords (variable assignments could influence the values of k and n).

3.5.2.8 Normalizing RTSCs with One-To-Many Communication Schemata

In this section, we explain how we enable an automatic normalization² of a multi role RTSC with one-to-many communication schemata that follows the definitions of Section 3.5.2.1 to a multi role RTSC with a contained subrole RTSC that follows our definition of Section 3.5.2.7.

By providing the normalization, we gain two advantages at once: First, we define the semantics of the one-to-many communication schemata by expressing them with the preexisting RTSC language (which uses an explicit subrole region in multi role RTSCs). Second, we enable the RTCP specifier to model multi roles using one-to-many communication schemata (we consider this as the easier specification variant) but also provide the possibility to modify the normalized RTSC in the case that our one-to-many communication schemata do not provide the required expressiveness.

We show our process of the normalization in Figure 3.12. In the following, we explain each process step in detail. Initially, in Step 1, we prepare our given RTSC such that the succeeding steps are easier to realize. In particular, we execute two normalizations that reduce some syntactical constructs but we do not normalize the one-to-many communication schemata yet. First, in Step 1.1, we split all transitions, where two one-to-many communication schemata are defined. This is the case if a transition defines that while firing the transition a message shall be consumed (via a specific one-to-many communication schema) and another message shall be sent (again via a specific one-to-many communication schema). We realize this step by replacing such a transition with two transitions and an urgent intermediate state (see

²In our context, we denote a transformation as a normalization if it reduces the number of syntactic features while it still preserves the semantics of the model.

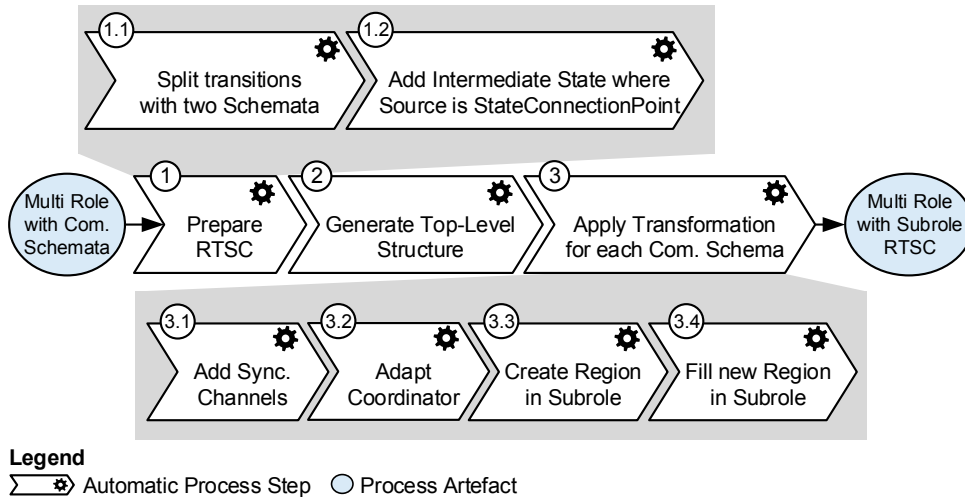


Figure 3.12: Process for the Normalization of RTSCs with One-To-Many Communication Schemata

Figure 3.13). We assign the conditions of the original transition to the transition that has the intermediate state as target while we assign the effects of the original transition to the transition that has the new intermediate state as a source. As a result, all further steps only have to deal with a single one-to-many communication schema per transition.

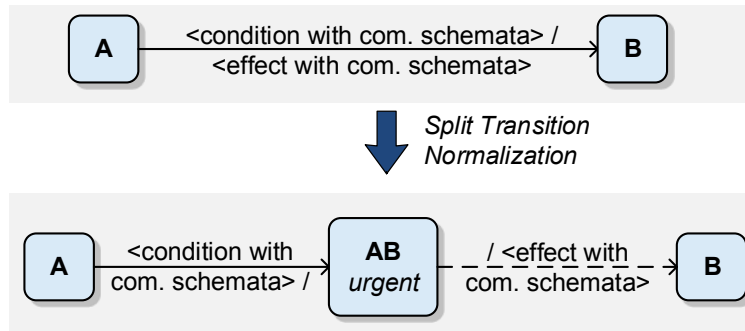


Figure 3.13: Split Transition Normalization

In Step 1.2, we replace all transitions that have a state connection point as source and that define to send a message by two transitions and an urgent intermediate state. This step is necessary as all our normalization templates of one-to-many communication schemata will add a synchronization to such a transition. However, MECHATRONICUML does not allow to define any condition (and thus also no synchronization) at transitions that have a state connection point as a source [*DPP+16, p. 32]. In particular, MECHATRONICUML defines two kinds of state connection points: entry and exit point. We show the normalization in case of an exit point in Figure 3.14(a) and in case of an entry point in Figure 3.14(b). As the figures depict, we assign the original effect definition to the transition that has the new intermediate state as a source.

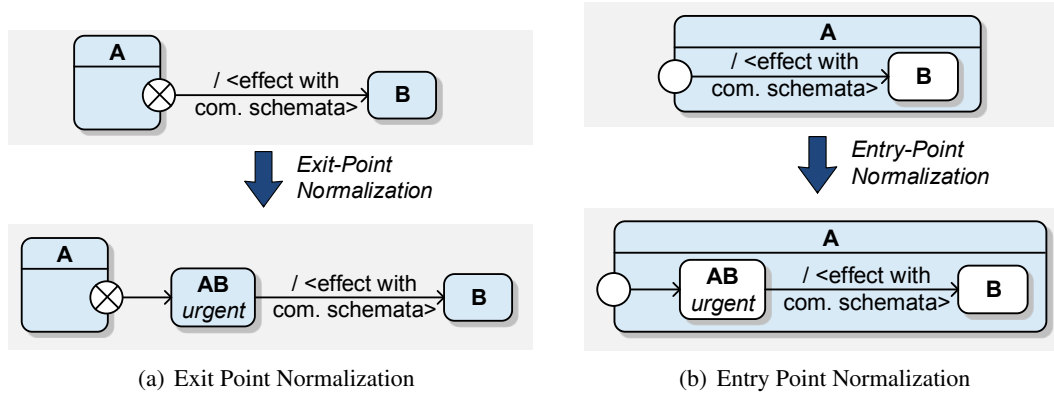


Figure 3.14: State Connection Point Normalization

Step 2 of our process generates the top-level structure of the new multi role RTSC according to the definition in Section 3.5.2.7. Therefore, we create a new top-level RTSC that defines the overall behavior for the multi role. Within this RTSC, we add the single main state that is always active. Then, we create two regions within this main state. We assign the original multi role RTSC, which still uses the one-to-many communication schemata, to the region of the coordinator. Moreover, we define that the second region is the one of the subrole. Afterwards, the structure of the overall RTSC has the same hierarchical structure as the multi role RTSC that we show in Figure 3.10 on page 68. Finally, we add a state called *Subrole_Main* to the region subrole. This state is initially active.

After Step 2, all preparations are done in order to normalize the one-to-many communication schemata in Step 3. For each transition of the original multi role RTSC that sends or receives a message (via a one-to-many communication schema), we apply the following three steps. At first, in Step 3.1, we add the necessary synchronization channels for this one-to-many communication schema in the main state of the root RTSC of the multi role. Next, in Step 3.2, we adapt the coordinator RTSC, e.g., by replacing the transition by intermediate states and connecting transitions. Then, in Step 3.3, we create a new region in the main state of the subrole region. Finally, in Step 3.4, we fill this newly created region with appropriate states, transitions, variables, and clocks.

The details of Step 3 differ between the one-to-many communication schemata as each schema has its own semantics. Therefore, we provide at least one transformation template for each one-to-many communication schema. In the following, we present a template for the one-to-many communication schema multicast and another template for multireceive as we need them for normalizing the one-to-many RTCP MultiOvertaking. Both templates may only be used if no subrole condition is specified (this is the case in our example). In Appendix A, we present for both schemata a template that shall be used if a subrole condition is specified. In addition, we present the templates of all other one-to-many communication schemata in this appendix as well.

Normalization Template for Multicast without Subrole Condition

The normalization for the one-to-many communication schema multicast is formally defined within our implementation using QVTo (see Section 3.7.2). In the following, we informally

describe the normalization of the one-to-many communication schema multicast that does not use subrole-specific conditions. We show its transformation template in Figure 3.15. The template supports a full specification of the transition except the attributes condition and `retryAfter`. Thus, it supports (1) the multicast attribute action (the actions a subrole instance shall execute after sending the message), (2) the subrole-independent conditions for enabling the transition, (3) the subrole-independent effects that shall be executed within the coordinator after the multicast is executed, and (4) transition deadlines.

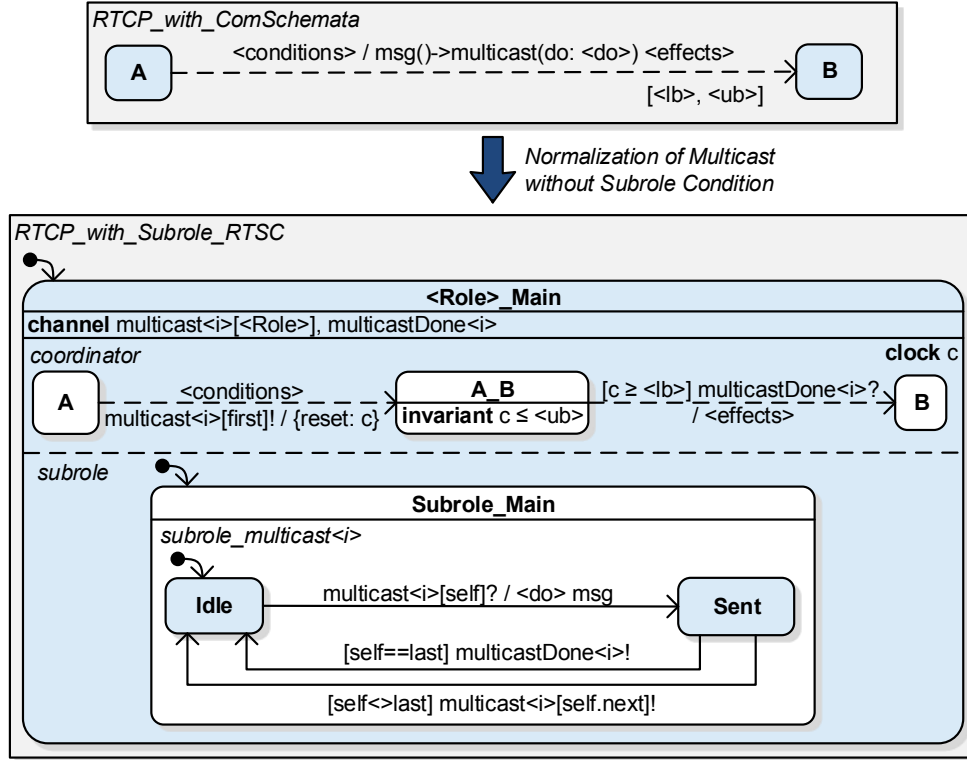


Figure 3.15: Normalization Template for Multicast without Subrole Condition

The general idea of the template is as follows: First, the coordinator has to fulfill all subrole-independent conditions. Then, the coordinator informs the first subrole instance that it shall send its message and execute its subrole-dependent behavior. Afterwards, the first subrole instance informs the next subrole instance that it shall send its message and execute its behavior. This pattern is repeated until the last subrole instance has sent its message and executed its behavior. Then, the last subrole instance informs the coordinator that the multicast has been sent. This has to happen before the transition deadline expires. Afterwards, the coordinator can execute the subrole-independent effects.

The template defines a deterministic behavior as the messages are sent in sequence rather than in parallel and as the order of messages conforms to the order of the subrole instances. The deterministic behavior benefits the required model checking time as only one execution path has to be analyzed rather than several ones.

In the following, we execute the process defined in Figure 3.12 to explain how our normalization uses this template. According to the template, we add in Step 3.1 two

synchronization channels `multicast<i>` and `multicastDone<i>`, where `<i>` is a unique number of all multicasts that are used within this multi role. Using this name schemata, we can support multiple multicasts within the same multi role. Moreover, the channel `multicast` is typed over the multi role. The channel `multicast<i>[<Role>]` is used by the coordinator to inform a specific subrole instance to send its message and between the subrole instances that the next subrole instance may send its message. The channel `multicastDone<i>` is used by the last subrole instance to inform the coordinator that the multicast has been executed.

We adapt the coordinator in Step 3.2 as follows: First, we have to replace the original transition by an intermediate state and two transitions. As long as the subrole instances execute the multicast, the coordinator will remain in this intermediate state. The subrole-independent conditions and the sending (“!”) of channel `multicast` are assigned to the first transition. Therefore, the multicast cannot start until both conditions are fulfilled. As soon as this is the case, the coordinator can inform the first subrole instance. Moreover, the first transition also adopts the urgency of the original transition, e.g., if the original transition is urgent, then the first one is urgent as well. We assign to the second transition the receiving (“?”) of channel `multicastDone` and the specified subrole-independent effects of the original transition. Therefore, the coordinator can be informed that the multicast has been executed and can execute the effects afterwards. If the original transition defines a deadline, then we execute five changes: (1) we add a clock to the RTSC of the coordinator, (2) reset this clock in the source state of the original transition (here: A), (3) define an invariant in the intermediate state, (4) add a clock constraint at the transition that leads to the target state of the original transition (here: B), and (5) define that this transition is non-urgent as MECHATRONICUML does not allow to define a clock constraint at an urgent transition. Note that we apply these five changes as well when we normalize deadlines when transforming an RTSC to UPPAAL timed automata in Section 5.4.7. If the original transition does not define a deadline, then we do not need to apply these five changes and the transition to B keeps its urgency.

In Step 3.3, we add a region with the name `subrole_multicast<i>` to the state `Subrole_Main`. Finally, in Step 3.4, we add two states `Idle` and `Sent` and three transitions to the region `subrole_multicast<i>`. Initially, the state `Idle` is active. The subrole instance is informed via channel `multicast<i>` when its turn starts. Then, it switches to state `Sent` while sending its message and executing its effect. Consequently, we execute the subrole-specific effect before the next subrole instance may send its message and before the coordinator executes its effect. Afterwards, the subrole instance either informs the next subrole instance that its turn starts or—if it is the last subrole instance—it informs the coordinator that the multicast has been executed.

Normalization Template for Multireceive Without Subrole Condition

The normalization for the one-to-many communication schema `multireceive` is formally defined within our implementation using QVTo (see Section 3.7.2). In the following, we informally describe the normalization of the one-to-many communication schema `multireceive` that does not use subrole-specific conditions. We show its transformation template in Figure 3.16. The template supports a full specification of the transition except the attributes `condition` and `retryAfter`. Thus, it supports (1) the `multireceive` attribute action (the actions a subrole instance shall execute after receiving the message), (2) the `multicast` attribute `failureAction` (the actions a subrole instance shall execute if receiving the message was not possible), (3) the subrole-independent conditions for enabling the transition, (4) the subrole-

independent effects that shall be executed within the coordinator after the multireceive is executed, and (5) transition deadlines.

The general idea of the template is as follows: First, the coordinator has to fulfill all subrole-independent conditions. Then, the coordinator informs the first subrole instance that it shall receive its message. If the subrole instance fulfills its subrole-specific condition, then it receives the message and executes its subrole-specific action. Though, if the subrole-specific condition is not fulfilled, then no message is received but the subrole-specific failure action is executed. Afterwards, the first subrole instance informs the next subrole instance that its turn starts. This pattern is repeated until the last subrole instance has its turn (thus, it either fulfills its conditions, receives the message, and executes its action or it only executes its failure action). Then, the last subrole instance informs the coordinator that the multireceive has been executed. This has to happen before the transition deadline expires. Afterwards, the coordinator executes the subrole-independent effects.

The template enables a deterministic behavior as the messages are consumed in sequence rather than in parallel and as the order of messages conforms to the order of the subrole instances. The deterministic behavior benefits the required model checking time as only one execution path has to be analyzed rather than several ones.

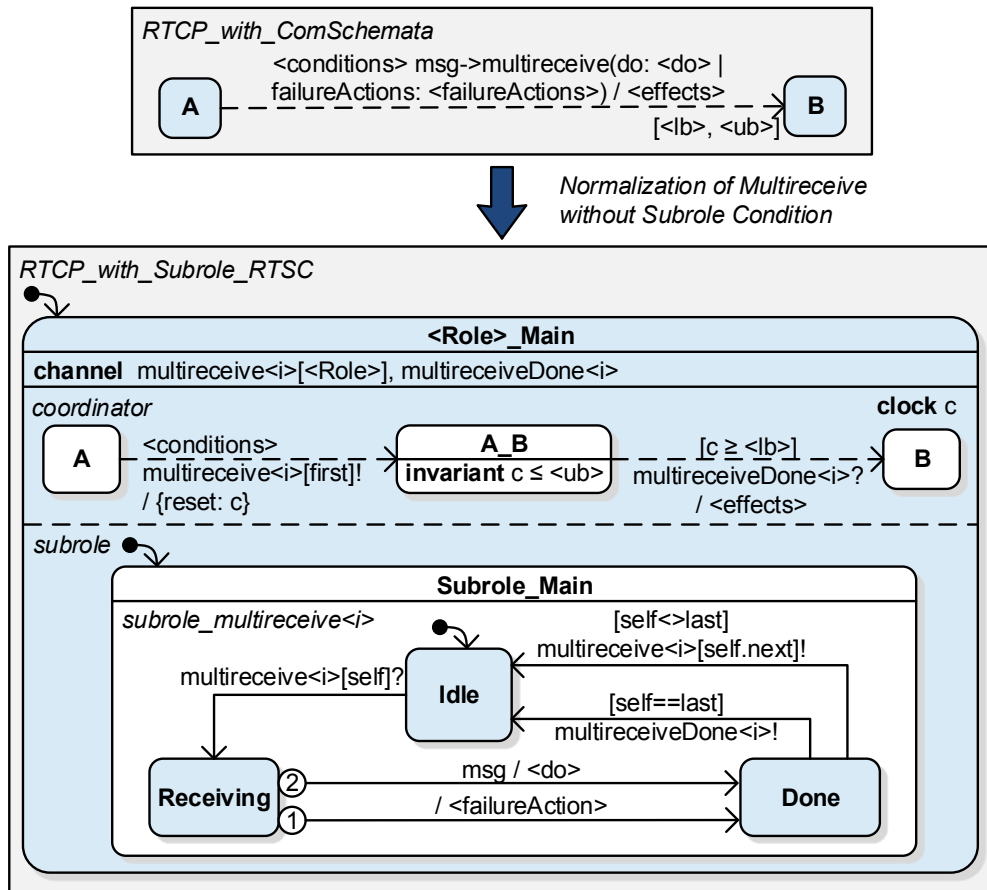


Figure 3.16: Normalization Template for MultiReceive without Subrole Condition

In the following, we execute the process defined in Figure 3.12 to explain how our normalization uses this template. According to the template, we add in Step 3.1 two synchronization channels `multireceive<i>` and `multireceiveDone<i>`, where `<i>` is a unique number of all multireceives that are used within this multi role. Using this name schema, we can support multiple multireceives within the same multi role. Moreover, the channel `multireceive` is typed over the multi role. The channel `multireceive<i>[<Role>]` is used by the coordinator to inform a specific subrole instance to receive its message and between the subrole instances that the next subrole instance may receive its message. The channel `multireceiveDone<i>` is used by the last subrole instance to inform the coordinator that the multireceive has been executed.

In Step 3.2., we have to replace the original transition by an intermediate state and two transitions. As long as the subrole instances execute the `multireceive`, the coordinator will remain in this intermediate state. The subrole-independent conditions and the sending (“!”) of channel `multireceive` are assigned to the first transition. Therefore, the multireceive cannot start until both conditions are fulfilled. As soon as this is the case, the coordinator can inform the first subrole instance. Moreover, this transition also adopts the urgency of the original transition, e.g., if the original transition is urgent, then the first one is urgent as well. We assign to the second transition the receiving (“?”) of channel `multireceiveDone` and the specified effects of the original transition. Therefore, the coordinator can be informed that the multireceive has been executed and can execute the effects afterwards. If the original transition defines a deadline, then we execute the same five changes as specified in the template of the schema `multicast` that does not use subrole-specific conditions.

In Step 3.3, we add a region with the name `subrole_multireceive<i>` to the state `Subrole_Main`. Finally, in Step 3.4, we add three states `Idle`, `Receiving`, and `Done` and six transitions to the region `subrole_multireceive<i>`. Initially, the state `Idle` is active. The subrole instance is informed via channel `multireceive<i>` when its turn starts. If the subrole instance fulfills its subrole condition, then it switches to the state `Receiving`, otherwise it switches to state `Done`. If the subrole instance is in state `Receiving`, then it tries to receive the message from the buffer. If a message is available, then it is consumed and the subrole action is executed. If no message is available, then the subrole-specific failure action is executed. In both cases, the subrole instance switches to state `Done`. Afterwards, the subrole instance either informs the next subrole instance that its turn starts or—if it is the last subrole instance—it informs the coordinator that the multireceive has been executed.

If the original transition defines no subrole condition, then we do not generate the transition from `Idle` to `Done` as the receiving of a message shall always happen. Furthermore, if the original transition defines no failure action, then we do not generate the transition from `Receiving` to `Done` that has priority 1 within the template.

Note that the transition priorities (see Section 2.2.2) that we define explicitly in this template are very important as the behavior would otherwise differ from the definition.

3.5.2.9 Normalized RTSC of Role Overtaker from RTCP MultiOvertaking

As an example for our normalization of one-to-many communication schemata, we show in Figure 3.17 the normalized RTSC of the multi role Overtaker of RTCP MultiOvertaking. We highlight the changes to the RTSC that uses one-to-many communication schemata (see Figure 3.9) in red. In the following, we execute the process, which we defined in Figure 3.12.

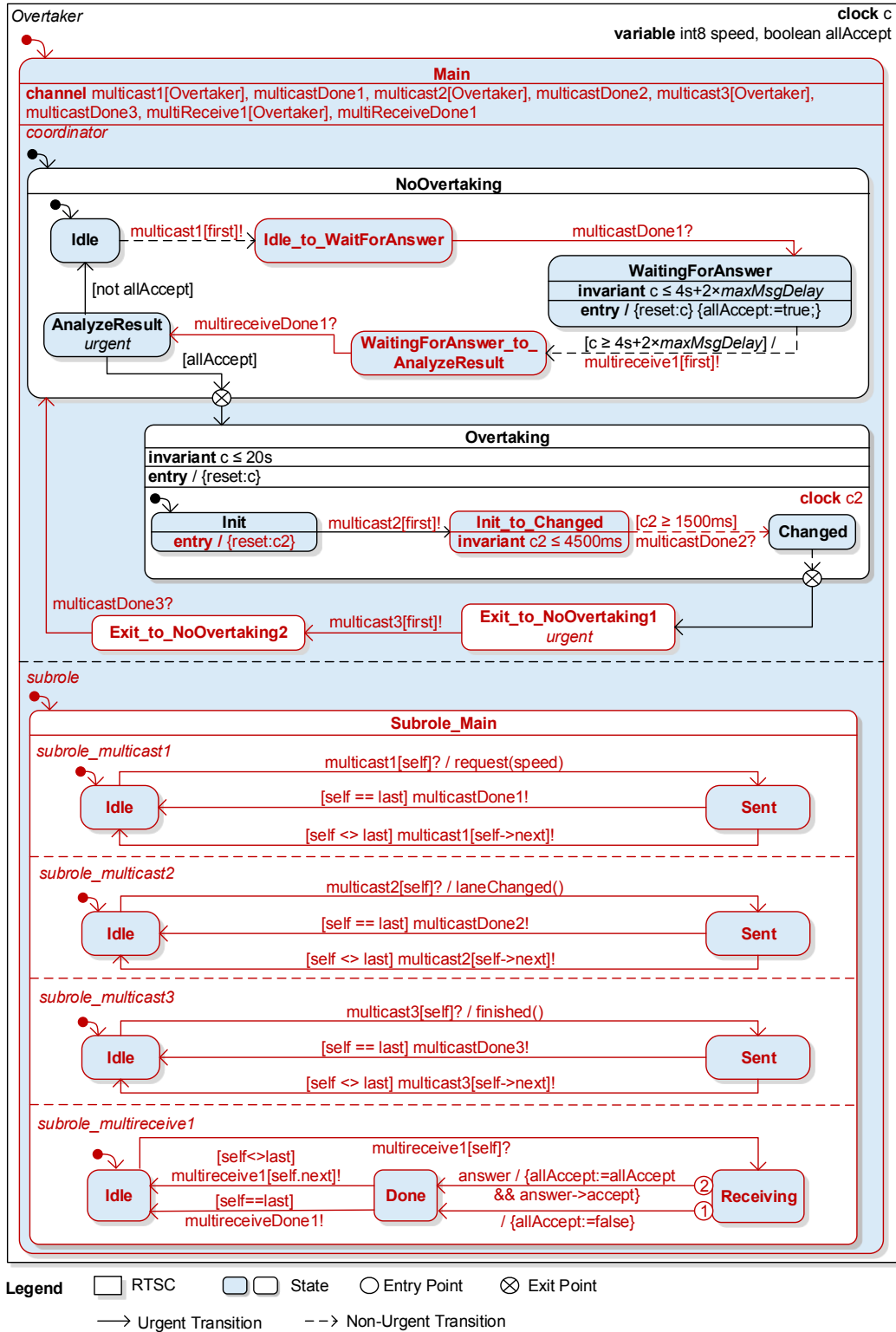


Figure 3.17: RTSC Overtaker without One-To-Many Communication Schemata (Changes are Marked Red)

In Step 1, our algorithm checks whether it has to apply one of the two preparation steps: The split transition normalization (Step 1.1) does not need to be applied as no transitions shall receive and send a message. However, the state connection point normalization must be applied once as the original RTSC shall send the message finished via a multicast and the transition has an exit point as source. Therefore, in Step 1.2, the state `Exit_to_NoOvertaking1` and two transitions replace the original transition.

In Step 2, the algorithm creates a new state `Main`, which is initially active, and creates two new regions within this state (coordinator and subrole). Afterwards, it moves the states and transitions of the original RTSC of the multi role into the region of the coordinator and adds the state `Subrole_Main`, which is initially active, into the region `subrole`.

In Step 3, the algorithm has to normalize three multicasts and one multireceive. All schemata do not define any subrole-specific condition. Therefore, the algorithm can apply the previously introduced templates (see Figures 3.15 and 3.16). According to the normalization templates, the algorithm adds eight synchronization channels to the state `Main` (two per schema). Afterwards, it adds four states to the coordinator (`Idle_to_WaitForAnswer`, `WaitingForAnswer_to_AnalyzeResult`, `Init_to_Changed`, and `Exit_to_NoOvertaking2`) and corresponding transitions. The transition that defines the multicast for the message `laneChanged` specifies a relative deadline. Consequently, the algorithm adds the clock `c2` in the region of state `Overtaking`, the clock reset in state `Init`, the invariant in state `Init_to_Changed`, and the clock constraint at the transition that has state `Changed` as target. In the state `Subrole_Main` in region `subrole`, the algorithm adds one region per schema. Then, it fills each region according to the template.

3.6 Applying RTCPs into the Component Model

As introduced in Section 2.2.1, the MECHATRONICUML component model defines—among others—discrete components, discrete ports, and assembly connectors. All three elements may be instantiated, i.e., discrete component instances may communicate with each other using asynchronous messages by containing discrete port instances that are connected via an assembly connector instance. For each message-based communication via discrete ports, MECHATRONICUML defines that one RTCP is applied [**DPP+16*]. In particular, it defines that each discrete port is assigned to a role of an RTCP. The discrete ports' behavior definition has to correctly refine the role behavior according to one of the six supported refinement definitions [**HBDS15*], i.e., the port behavior and, therefore, the complete component behavior, may not violate the behavior that the assigned role guarantees to its other role.

In this thesis, we improve the specification of RTCPs (see Section 3.3, 3.4, and 3.5). As described above, the component model depends on the RTCPs. Therefore, we present in the following the changes that we made to the component model specification. In particular, we define in Section 3.6.1 a specification for incoming message buffers at discrete ports. Then, in Section 3.6.2, we define how our QoS assumptions influence the component model. Afterwards, we briefly explain our support for one-to-many communication schemata for the behavior specification of multi ports. Finally, we present an example for a component model that incorporates our changes in Section 3.6.4.

3.6.1 Incoming Message Buffers at Discrete Ports

Like roles of an RTCP, discrete ports of a discrete software component receive asynchronous messages as well. Therefore, we can directly transfer our concepts of incoming message buffers for roles (see Section 3.3.4) to the definition of discrete ports. As a discrete port always has to refine a certain role, the component specifier does not have to specify the incoming message buffers explicitly as their specification may be derived from the role. However, depending on the chosen refinement definition, it may be required that the size of the incoming message buffers has to be increased. In particular, Heinzemann and Henkler [HH11] identified a refinement definition that they call *relaxed timed bisimulation*, which relaxes the strict requirements of a timed bisimulation. This new refinement definition “[...] enables to extend the time intervals for receiving message[s], but it still requires that the upper bounds for sent messages remain unchanged. This refinement is particularly useful for [CPSs]. If two [CPSs] coordinate on a specific task, it often does not matter when messages are received but only that the answer is on time.” [*HBDS15] For example, in our overtaking scenario of RTCP Overtaking (see Figure 3.6), it is not relevant for the Overtaker when the Overtakee consumes the message request as it only matters that the request is eventually consumed and that the answer is sent in time. “A consequence of delayed receiving of messages in the [discrete port] is that messages are taken out of the [incoming message buffer] later than in the [RTCP].” [*HBDS15] If the size of the incoming message buffer at the discrete port would remain identical to the size at the role, then messages may get discarded at the discrete port that would not be discarded at the role. This would violate the idea of timed bisimulation even if it is relaxed. “To prevent [such violations], we require the [discrete port] to have an [incoming message buffer] that is sufficiently larger than that of the [RTCP]. Otherwise, the relaxed timed bisimulation cannot be applied.” [*HBDS15] Consequently, we allow that a component specifier may define incoming message buffers for a discrete port explicitly. Then, the definition at the discrete port overwrites the definition of the role it refers to. In particular, we allow two changes: (1) The component specifier may increase the size of the buffers if he chooses the relaxed timed bisimulation as its refinement definition. (2) If the connector is unreliable (i.e., messages may get lost during transmission), then we also allow that the component specifier may change the buffer overflow avoidance strategy. This is possible, as the content of the incoming message buffer is transparent to the port RTSC. Therefore, the port RTSC cannot distinguish whether a message has been lost during transmission or due to a discardment from the buffer.

3.6.2 Quality-of-Service Assumptions at the Component Model

QoS assumptions for the component model are crucial for two aspects. At first, they are required in order to execute a meaningful system verification, e.g., via the transformation to Matlab/Simulink/Stateflow [Hei15]. Second, these assumptions are necessary for the platform-specific modeling (including the target code generation). Otherwise, a platform may be chosen that does not fulfill the assumptions that were considered during the verification of the RTCP. Like the role connector of an RTCP, an assembly connector that connects discrete ports of the component model transmits asynchronous messages. Therefore, we can directly transfer our concepts of explicit and implicit QoS assumptions for RTCPs (see Section 3.3.2 and 3.4) to the definition of such assembly connectors. We allow that a component specifier may define

explicit QoS assumptions for an assembly connector between discrete ports explicitly. Then, the definition at the assembly connector overwrites the definition of the role connector it refers to. In particular, we enable to make two changes concerning the explicit QoS assumptions at an assembly connector compared to the referenced role connector definition: (1) The component specifier may change the reliability of the connector from `false` to `true` and (2) if the connector is unreliable, then he may change the message order preservation from `false` to `true`. Both changes are always valid refinements as they represent a subset of the previous state space. Moreover, both changes are useful as they positively influence the simulation model as well as the selection of the platform: For example, if the reliability is changed to `true`, then the simulation does not need to simulate a loss of messages. This reduces the number of relevant test cases that the component specifier has to cover. All other changes to the explicit QoS assumptions (reducing the maximum message transmission delay, changing the reliability from `true` to `false`, and changing the message order preservation from `true` to `false`) might be valid refinements but also may lead to invalid refinement. Currently, the existing refinement checks of MECHATRONICUML [*HBDS15], does not cover changes to the explicit QoS assumptions. Consequently, we forbid such changes as long as the refinement check is not updated because an invalid refinement can lead to an unsafe system.

3.6.3 One-To-Many Communication Schemata

Before this thesis, the specification of a multi port RTSC and that of a multi role were identical. For example, both required an explicit subrole (see Section 3.5.2.7 and [*DPP+16]) for their behavior specification. Consequently, we can directly transfer our concepts of one-to-many communication schemata (see Section 3.5.2.1) to the behavior specification of discrete multi ports. Like in the multi role specification, the component specifier can choose between four one-to-many communication schemata (multicast, unicast, iterate, loadbalancing) for sending messages to multiple single role instances and between two one-to-many communication schemata (multireceive and singlereceive) for receiving messages from multiple single role instances.

3.6.4 Example

In the following, we give an example for a valid application of an RTCP to a component model. At first, in Section 3.6.4.1, we present the structural specification, which includes our conceptual changes, for our overtaking scenario with contraflow (the blue car) from Section 1.1. Then, we present an excerpt of a possible behavioral definition for this component model in Section 3.6.4.2.

3.6.4.1 Structural Specification

We show the atomic software components of the scenario in Figure 3.18. First of all, we define one common discrete software component `CarSW` for all three cars. It specifies four ports:

pOvertaker It specifies an optional discrete in/out multi port `pOvertaker`. It refines the multi role `Overtaker` of RTCP `MultiOvertaking`. The port does not need to define incoming message buffers explicitly but derives them from the role it refers to.

pOvertakee It specifies an optional discrete in/out single port pOvertakee. It refines the single role Overtakee of RTCP MultiOvertaking. Again, the port does not need to define incoming message buffers explicitly but derives them from the role it refers to.

distance It specifies an optional hybrid in-port Distance, which receives the distance to an obstacle (e.g. another car) in front as a value of type int8 each 30 milliseconds.

velocity It specifies an hybrid out-port Velocity, which sends the target velocity of the car as a value of type int8 each 30 milliseconds.

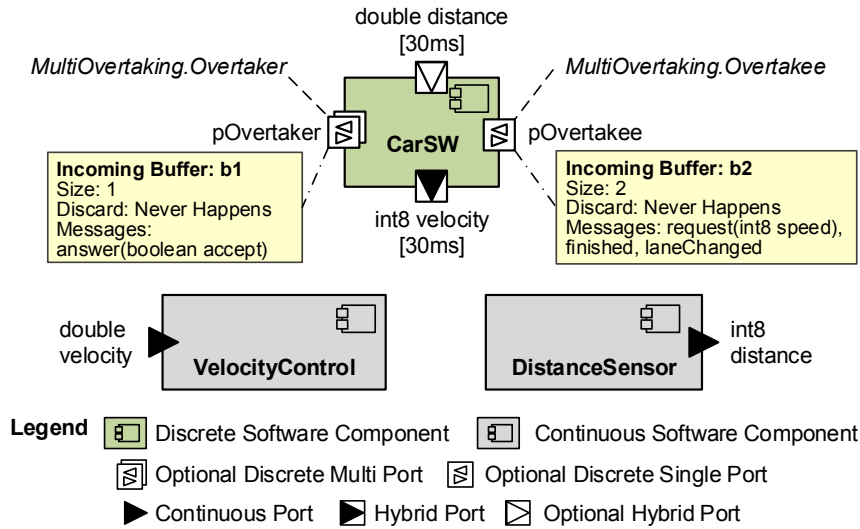


Figure 3.18: Atomic Software Components for the Overtaking Scenario

Concerning the continuous components, the task of VelocityControl is to control the velocity of the car. It defines a continuous in-port velocity that receives the target velocity the component shall achieve. In contrary, the task of the component DistanceSensor is to measure the distance to the car in front and to send the actual distance via its continuous out-port distance.

Figure 3.19 shows a valid component instance configuration (CIC) of these atomic components. The discrete software component instances red, yellow, and blue are all typed over CarSW. red instantiates the continuous ports of type distance and velocity and the multi port instance of type pOvertakee. The latter one is instantiated with a port multiplicity of two. In contrary, yellow and blue only instantiate the continuous port velocity and create a single port instance of type pOvertakee. red is connected via assembly connector instances to instances of the continuous components VelocityControl and DistanceSensor while yellow and blue are connected via an assembly connector instance to different continuous components instances of type VelocityControl. Moreover, red is connected via two assembly connector instances to yellow and blue. Therefore, these three instances may exchange messages with each other according to the RTCP MultiOvertaking. In the figure, we show the QoS assumptions of both assembly connector instances that are used to establish the RTCP instance. In our example, both QoS assumptions does not need to be refined. Therefore, we do not have to state them explicitly as they can be derived from the RTCP definition.

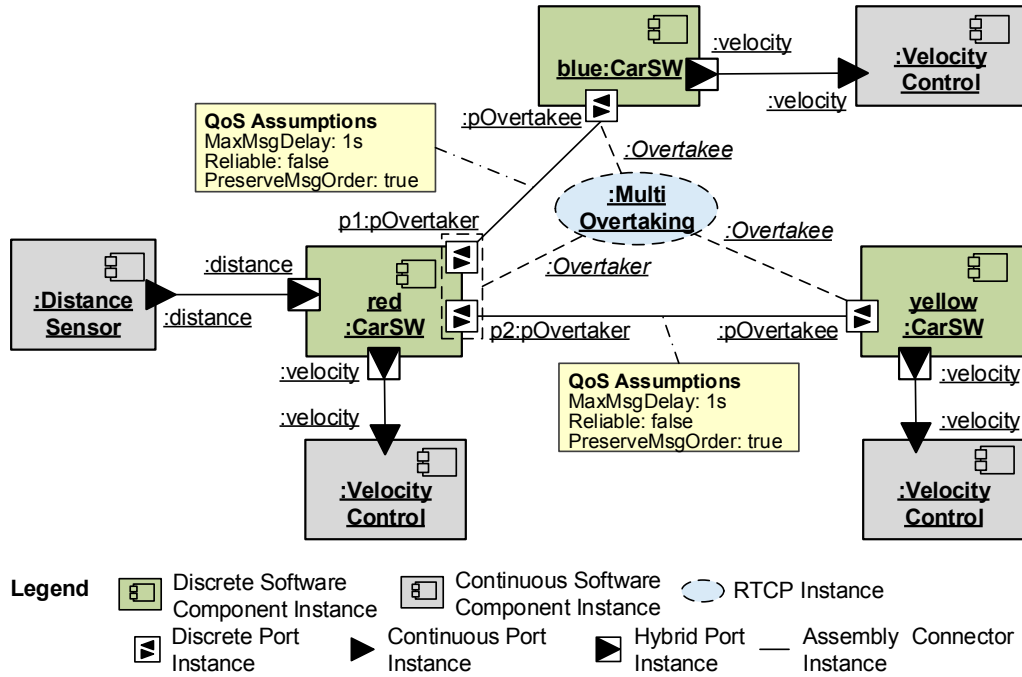


Figure 3.19: CIC of Two Cars that Apply RTCP Overtaking

3.6.4.2 Behavioral Specification

We show an excerpt of a possible behavioral definition for the discrete software component CarSW in Figure 3.20. As defined by MECHATRONICUML [*DPP+16, p. 107], the RTSC defines a single hierarchical state that is initially active. This state may not have transitions but has to consist of a region per discrete port. Therefore, the top state contains the regions pOvertaker and pOvertakee that define the behavior of the ports with the respective names. Moreover, the top state defines a region internalBehavior that defines internal component behavior—among others—the execution of the overtaking. We do not show the contents of this region in the figure as they are not necessary for our example. In the following, we first explain the refinements that the port RTSC in region pOvertaker applies compared to the RTSC of role MultiOvertaking.Overtaker (see Figure 3.9). Afterwards, we explain the refinements that the port RTSC in region pOvertakee applies compared to the RTSC of role MultiOvertaking.Overtakee (see Figure 3.9).

Concerning region pOvertaker, we replace the non-urgent transition from NoOvertaking.Idle to WaitingForAnswer by two urgent transitions and the intermediate state DefineSpeed. In the role, this transition could fire non-deterministically at any time. Now, the state Idle must be left as soon as the hybrid port distance receives a value below 50. Then, in the intermediate state, the planned overtaking speed is being calculated via the new operation defineOvertakingSpeed. After the execution of the operation, the message request shall be sent immediately via a multicast. As another change, if the port receives via a multireceive of message answer that all cars accept the request, then this region additionally synchronizes with the internal behavior via channel startOvertaking to trigger the start of the overtaking. The latter one is defined in the region internalBehavior. The region internalBehavior requires

access to the accepted overtaking speed. Therefore, we relocate the variable speed from the port RTSC to the component RTSC, which makes it available to all regions of the state CarSWCompMain. An additional change is that we add the channel changed to the main state of the component RTSC. We assign this channel as a condition to the transition from Overtaking.Init to Changed. Thus, this transition fires no longer non-deterministically but as soon as the internal behavior defines via this channel that the lane has been changed. Similarly,

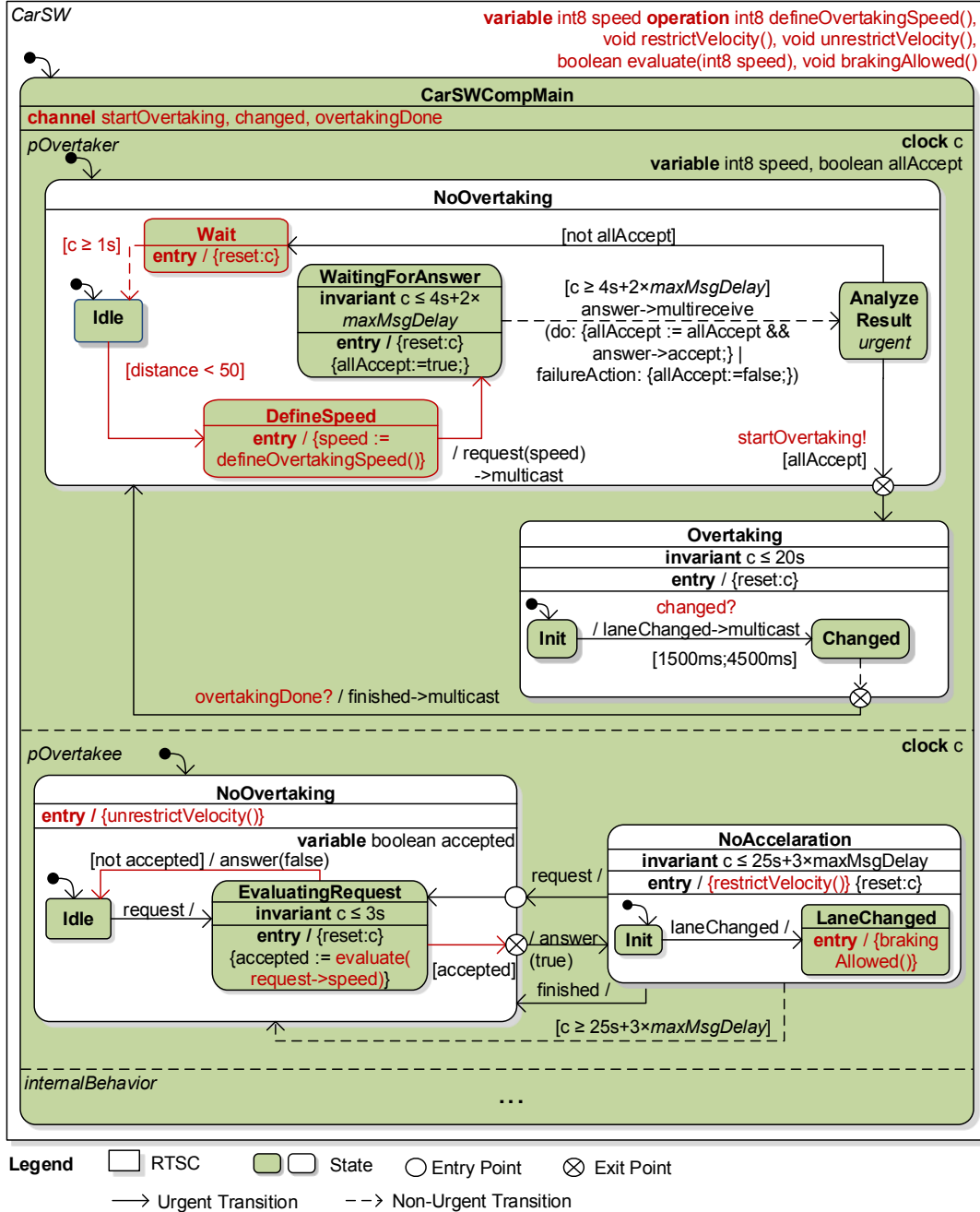


Figure 3.20: Component Behavior for RedSW and YellowSW (Refinements are Marked Red)

the state *Overtaking* is not left non-deterministically within 20 seconds anymore, but as soon as the internal behavior finished the overtaking (we still assume that overtaking is always done within 20 seconds). We realize this by changing the transition that has state *Overtaking* as source from non-urgent to urgent and by adding the condition that it shall synchronize its firing via the channel *overtakingDone*. Finally, we define that it is not allowed to send an overtaking request immediately after a previous request was unsuccessful but that the port has to wait at least one second. For doing this, we add the state *NoOvertaking.Wait* and a non-urgent transition with an appropriate clock constraint.

Concerning region *pOvertakee*, we define that whenever the port RTSC switches to state *NoOvertaking*, then the new operation *unrestrictVelocity* is executed, which frees the component behavior concerning a restricted accelerating or decelerating. Then, we replace the non-deterministic choice in state *NoOvertaking.EvaluatingRequest* by a call of the new operation *evaluate*, which gets the value of the parameter *speed* of message request as an input. Consequently, we can now change both transitions that leave the state *NoOvertaking.EvaluatingRequest* from non-urgent to urgent as they shall fire as soon as the operation is done. Next, the state *NoAcceleration* now additionally executes the new operation *restrictVelocity* whenever it is activated, which restricts the component behavior to accelerate or decelerate the car. Finally, the state *NoAcceleration.LaneChanged* now has to execute the new operation *brakingAllowed*, which allows the component behavior (e.g., the internal behavior) to decelerate the car.

3.7 Implementation

In this section, we present excerpts of our prototypical implementation of an RTCP editor and the wizard to transform a multi role RTSC that uses one-to-many communication schemata to a multi role RTSC with an explicit subrole. Our implementation is integrated into the MECHATRONICUML Tool Suite [*DGB+14].

In particular, we present the user interface in Section 3.7.1 and the plugin structure in Section 3.7.2. We show and explain our metamodel for RTCPs in Appendix E.1.

3.7.1 User Interface

As part of the MECHATRONICUML Tool Suite, our user interface provides various graphical editors to specify an RTCP. In particular, we provide editors for specifying the message types, the data types, the structure of an RTCP, and the role behavior (i.e., the RTSCs).

We show a screenshot of the user interface for specifying the structure of an RTCP in Figure 3.21, which depicts the one-to-one RTCP *Overtaking*. The editor follows our graphical syntax definition (compare Figure 3.6). The RTCP specifier may edit the structure of the RTCP within the diagram (e.g., using the palette) and using MECHATRONICUML's property view.

As another example, we show in Figure 3.22 a screenshot of the user interface for specifying an RTSC in order to define the role behavior of an RTCP. In particular, the screenshot shows the multi role behavior specification for the role *Overtaker* of RTCP *MultiOvertaking*. Again, our editor follows the graphical syntax that MECHATRONICUML defines and that we use within our illustrations (see Figure 3.9). Moreover, the RTCP specifier may edit the RTSC within the diagram (e.g., using the palette) and using our tool's property view as well. Furthermore, the

figure shows that the RTCP specifier can start the normalization³ of a multi role RTSC that uses one-to-many communication schemata by right-clicking in the diagram and selecting within the context menu the option MechatronicUML»Disassemble One-To-Many Communication Schemata.

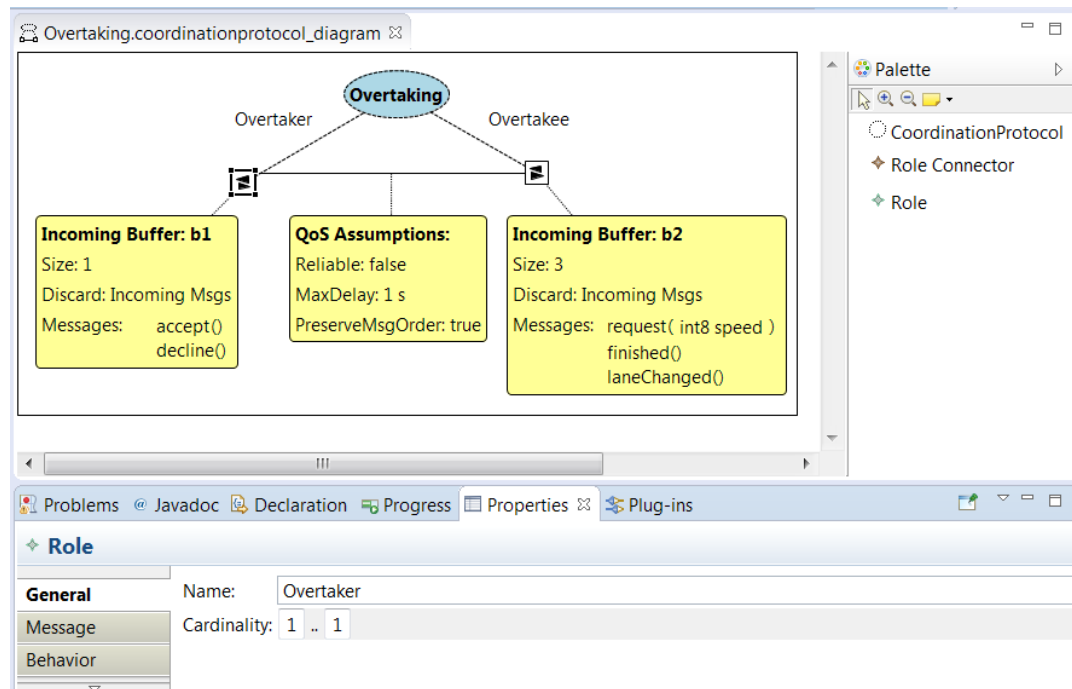


Figure 3.21: User Interface for Specifying the Structure of an RTCP

3.7.2 Plugin Structure

We show an excerpt of the plugins that we developed and adapted for our prototypical implementation in Figure 3.23. Note that we provide the documentation of our metamodel in Appendix E.1.

First of all, we define the EMF-based metamodel for RTCPs within the plugin `pim` and integrate it with the metamodel for the platform-independent modeling (pim) of MECHATRONICUML. In particular, we define all RTCP-specific elements in the package `protocol`. Moreover, we define all metamodel elements for specifying one-to-many communication schemata in the package `one_to_n_schemata`, which is contained in the package `realtimestatechart`.

We provide three plugins for the user interface. The plugin `coordinationprotocol.diagram` defines the GMF-based visual editor for specifying our RTCPs in their concrete syntax. Note that we store the commonalities with the visual editor for Real-Time Coordination Patterns (see Chapter 6) within the plugin `coordinationspecification.common`. Furthermore, the plugin `coordinationprotocol.diagram.custom` exists. It contains manual written code for the visual editor due to limitations of GMF (it does not enable all editor functionalities that we need).

³In the past, we referred to the normalization of one-to-many communication schemata as disassembling. Within this thesis, we only use the term normalization but our implementation still calls it disassembling.

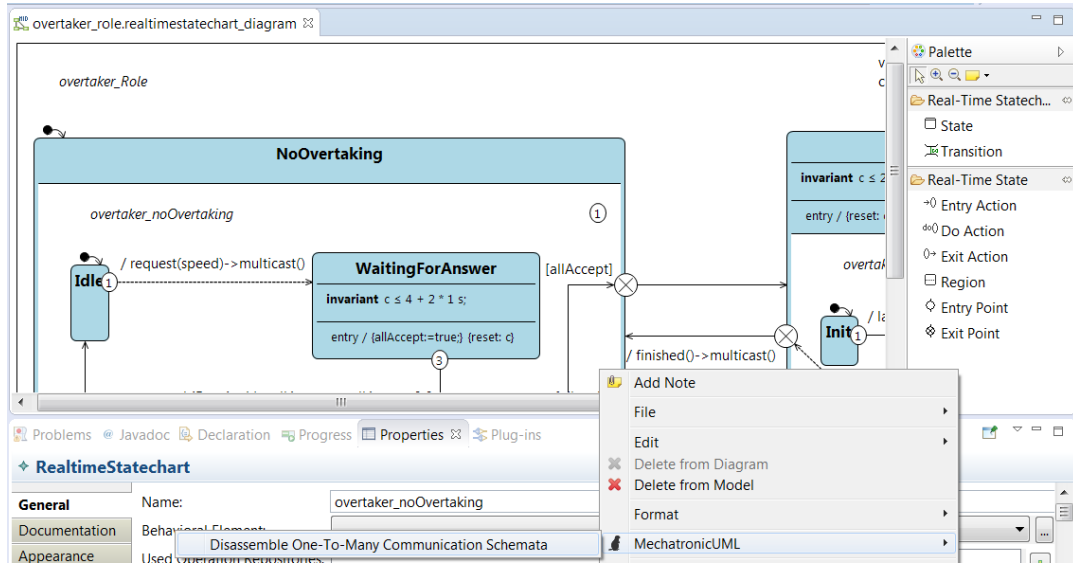


Figure 3.22: User Interface for Specifying the Behavior of a Role via RTSCs

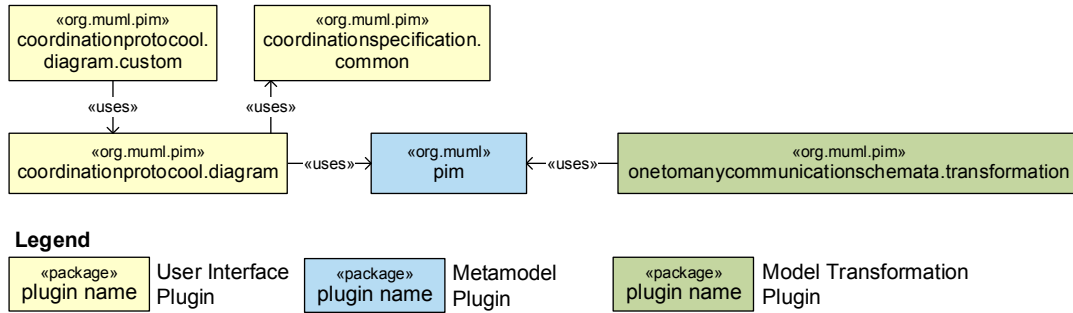


Figure 3.23: Plugins used for implementing RTCPs

At last, the plugin `onetomanycommunicationschemata.transformation` contains the transformation code written in QVTo to enable the normalization of one-to-many communication schemata of a multi role RTSC.

3.8 Evaluation

We conduct a case study based on the guidelines by Kitchenham et al. [KPP95] and Runeson et al. [RHRR12] for evaluating our concept of one-to-many communication schemata, which we introduce in Section 3.5.2.1. In particular, we execute the evaluation using realistic examples within the domain of CPSs but do not aim for generalizations. Noteworthy, all examples of this case study are defined by ourselves or colleagues and are not taken from real industry examples.

Our evaluation is structured as follows: First, we present the context of our case study in Section 3.8.1. Then, we set our hypotheses in Section 3.8.2 and prepare the validation in

Section 3.8.3. Afterwards, we validate the hypotheses in Section 3.8.4. Finally, we discuss the threats to validity in Section 3.8.5 and analyze our results in Section 3.8.6.

3.8.1 Case Study Context

The objective of this case study is to evaluate to what extent we fulfill the requirement R8 (“provide explicit one-to-many communication support”; see Section 3.2). In particular, we investigate whether our concept of one-to-many communication schemata is applicable to model realistic one-to-many RTCPs without requiring too much model elements.

3.8.2 Setting the Hypotheses

Our case study has two evaluation hypotheses:

- H1** The six one-to-many communication schemata that we identified are sufficient in order to model a variant of the given example with only small semantic derivations.
- H2** Our concept of using one-to-many communication schemata enables more compact multi role behavior definitions than using an explicit subrole RTSC.

For evaluating the hypotheses, we collect existing multi role behavior definitions that use an explicit subrole region. Afterwards, we model for each existing multi role a variant that uses one-to-many communication schemata. As we are experts of MECHATRONICUML and one-to-many communication schemata, we (one of my students and myself) can verify H1 by inspecting manually whether semantic deviations between the original and the variant exist. We expect that our six one-to-many communication schemata are sufficient in most cases. Concerning H2, we compare the following three metrics between the original and the variant: (1) the number of states, (2) the number of transitions, and (3) the number of characters that the concrete syntax uses. We expect that in most cases the variant requires less states, less transitions, and less characters than the original.

Noteworthy, we do not require that no semantic deviations exist as this is not the goal of our concept with one-to-many communication schemata. Instead, we only want to provide a concept that enables to model a broad range of one-to-many RTCPs.

3.8.3 Preparing the Validation

We selected four already existing one-to-many RTCPs. A brief overview of them is as follows:

PositionTransmission This RTCP is used for the coordination of autonomous robots called BeBots [HWR09; BeBot] that explore an unknown environment and have to avoid collisions with the environment and other BeBots. It defines two roles: a Sender and a Receiver and is used within the software of a single BeBot. In particular, the multi role Sender periodically sends position data to all instances of role Receiver. Initially, it has been defined by one of our students in [*BDG+11] before we defined our concepts. Later on, it has been refined by students, colleagues, and myself in [*BBD+12; *BBB+12; *BDG+14b; *DPP+16]. For our evaluation, we use the latest version. We show the RTCP in Appendix C.1.3.4 on page 332.

Distribution This RTCP is used in the aforementioned BeBot scenario as well. It defines two roles: Distributor and Client. The multi role distributor periodically awaits position data from each client and distributes all received position data to all clients. Like RTCP PositionTransmission, it has been defined initially by one of our students in [*BDG+11] before we defined our concepts. Later on, it has been refined by students, colleagues, and myself in [*BBD+12; *BBB+12; *BDG+14b; *DPP+16]. For our evaluation, we use the latest version. We show the RTCP in Appendix C.1.3.3 on page 330.

ProfileDistribution An example about autonomous railway shuttles, which we call RailCabs [HTS+08], uses this RTCP. It defines two roles: a ProfileProvider and a ProfileReceiver. The RTCP defines that the multi role ProfileProvider sends motion profiles to instances of the ProfileReceiver and receives data about the actual maximum speed of each ProfileReceiver. The RTSC has been defined by Heinzemann [Hei15]. Neither did Heinzemann know our work nor did we knew this RTCP before our evaluation. We show the RTCP in Appendix C.1.2.3 on page 324.

ConvoyCoordination This RTCP is used in the aforementioned RailCab scenario as well in order to establish a convoy of RailCabs. It defines two roles: Coordinator and Member. The multi role Coordinator is responsible for managing the convoy, i.e., it decides whether a Member may join the convoy and at which position. Furthermore, the Coordinator may decide that a change in the convoy results in updating the motion profiles of all members. The RTSC has been defined by Heinzemann [Hei15] as well. Again, neither did Heinzemann know our work nor did we knew this RTCP before our evaluation. We show the RTCP in Appendix C.1.2.4 on page 325.

The RTCPs ProfileDistribution and ConvoyCoordination both enable a runtime reconfiguration of the number of subrole instances and single instances by concepts of Heinzemann [Hei15]. However, in this thesis, we do not consider the possibility of runtime reconfiguration. As a consequence, we remove the model elements concerning the runtime reconfiguration in both RTCPs. Moreover, we had to fix errors in both RTSCs that would result in a deadlock or a non-fulfillment of the RTCP's requirements. We state our changes in detail in Appendix C.1.2.3 and C.1.2.4.

In order to count the number of characters of an RTSC, we select the following method: First, we store the RTSC diagram model as a PDF and open it within the tool Adobe Reader. Within this tool, we select the complete diagram, copy it, and paste it to Microsoft Word 2016. Then, we use Word's word count feature to retrieve the number of characters with spaces.

3.8.4 Validating the Hypotheses

In the following, we present the validation of our hypotheses. We start with role Sender of RTCP PositionTransmission in Section 3.8.4.1. Then, we continue with role Distributor or RTCP Distribution in Section 3.8.4.2 before we explain the role ProfileProvider of RTCP ProfileDistribution in Section 3.8.4.3. We finish with role Coordinator of RTCP ConvoyCoordination in Section 3.8.4.4.

3.8.4.1 Role Sender of RTCP PositionTransmission

We show the resulting multi role RTSC of role PositionTransmission.Sender in Figure 3.24. We are able to specify all message exchanges using the one-to-many communication schema multicast. In particular, we use a multicast to send the position data to receivers via the message position. Our variant using one-to-many communication schemata does not have semantic derivations. Concerning hypothesis H2, our variant needs 1 state, 1 transition, and 122 characters compared to 5 states, 5 transitions, and 322 characters in the original RTSC.

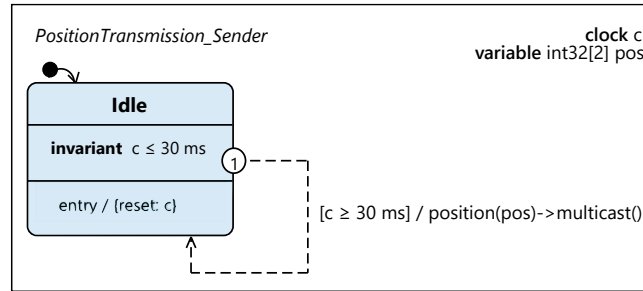


Figure 3.24: Variant of Role Sender of RTCP PositionTransmission that uses One-To-Many Communication Schemata

3.8.4.2 Role Distributor of RTCP Distribution

We show the resulting multi role RTSC of role Distribution.Distributor in Figure 3.25. We are able to specify all message exchanges using the one-to-many communication schemata multicast and multireceive. In particular, we use a multireceive to receive the position data of each client via message position and use a multicast to send all position data to all clients via the message positions. All effects that are executed within the subrole are now part of the attribute action of the multireceive. Our variant using one-to-many communication schemata does not have semantic derivations. Concerning hypothesis H2, our variant needs 2 states, 2 transitions, and 363 characters compared to 10 states, 10 transitions, and 667 characters in the original RTSC.

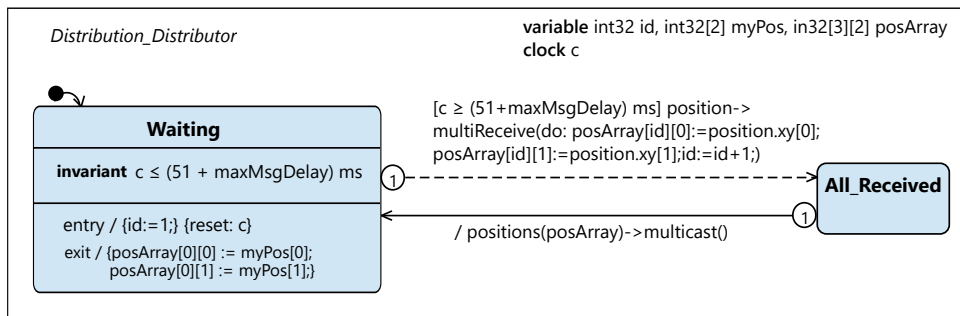


Figure 3.25: Variant of Role Distributor of RTCP Distribution that uses One-To-Many Communication Schemata

3.8.4.3 Role ProfileProvider of RTCP ProfileDistribution

We create two variants of the multi role ProfileDistribution.ProfileProvider. The first one, which we depict in Figure 3.26, shows a variant that nearly matches the original semantics by using the one-to-many communication schemata unicast and singlereceive. In particular, we use the unicast to send new data or a new profile to each receiver, while we use the singlereceive to get an update from each receiver concerning the strategy. We made two operations subrole-specific: `getProfile` and `updateConvoySpeed`. We use `getProfile` within the parameter binding of the message `newProfile` and `updateConvoySpeed` in the attribute action of the message `updateStrategy`. In addition, we require two additional variables and two additional subrole-specific variables: (1) the variable `receivers` represents the current number of `profileReceivers`, (2) the variable `updatedReceivers` represents the number of `profileReceivers` that already got an update, (3) the subrole-specific variable `msgSent` indicates whether the subrole already sent its message to the `profileReceivers`, and (4) the subrole-specific variable `waitForMsg` indicates whether a subrole already sent a message but did not receive a message of type `updateStrategy` yet. The both former variables of the explicit subrole region (`tmpProfile` and `tmpMemberSpeed`) are not required within our variant. Moreover, we change the subrole-specific clock to a normal clock as it is not necessary that this clock is subrole-specific. The reason for this is that subrole instances are executed one after the other and the clock is reset at the start of each subrole instance. The only semantic deviation is at the transition from `WaitAnswer` to `CheckForUpdate`: Using the attribute `retryAfter`, we define that the transition has to wait 1 ms before it may check again whether the message has already been received. In the original, the transition urgently fired as soon as the message was available. This semantic deviation does not effect the other role but still has effects on the timing behavior of this role. Our second variant, which we show in Figure 3.27, implements only a similar behavior compared to the original: When sending the new profile respectively the new data to the receiver, then we do not send one single message of type `newProfile` or `newData` to one receiver, wait up to 45 ms for the updated strategy by each receiver, and up to 1 ms additionally to compute the new convoy speed before sending the next single message to another receiver. Instead, we send a multicast to all receivers at once, wait 45 ms, and then consume all received messages via a `multireceive` within up to 20 ms (the RTCP is able to have 20 receivers and we wait up to 1 ms per receiver). Compared to the original, we achieve the same functionality with less model elements, e.g., we do not need the two additional variables and the two additional subrole-specific variables. Moreover, our second variant does not require changes to the other role of this RTCP. Concerning hypothesis H2, our first variant requires 3 states, 6 transitions, and 1,168 characters compared to 7 states, 9 transitions, and 937 characters in the original RTSC. The more compact variant (the second one) that matches only concerning the functionality requires 3 states, 5 transitions, and 732 characters.

3.8.4.4 Role Coordinator of RTCP ConvoyCoordination

We show the resulting multi role RTSC of role ConvoyCoordination.Coordinator in Figure 3.28. We specify all message exchanges using the one-to-many communication schemata unicast and singlereive. Moreover, we add the variable `updatedProfiles`, the subrole-specific variable `newMember`, and the subrole-specific operation `getProfile()`. Each schema that we apply either uses the variable `newMember` or the new operation within its condition or its action

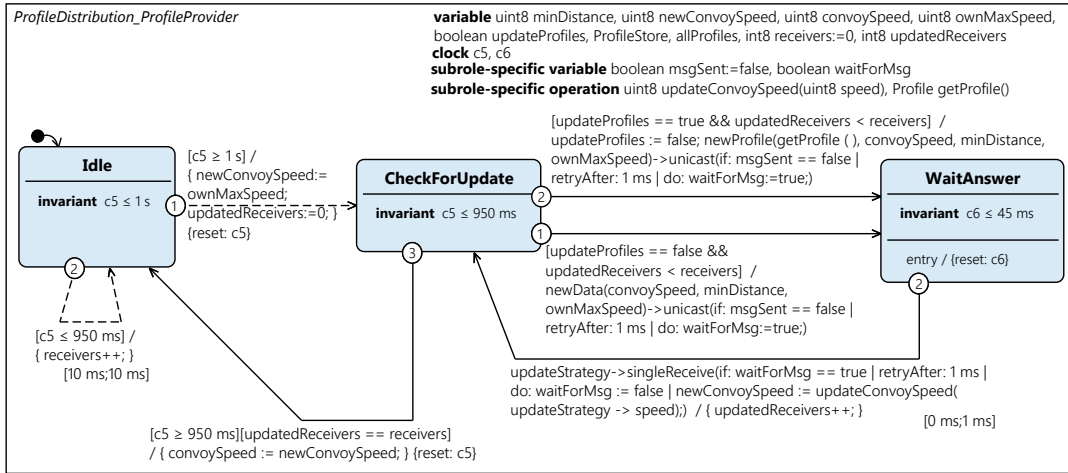


Figure 3.26: Variant 1 of Role ProfileProvider of RTCP ProfileDistribution that uses One-To-Many Communication Schemata

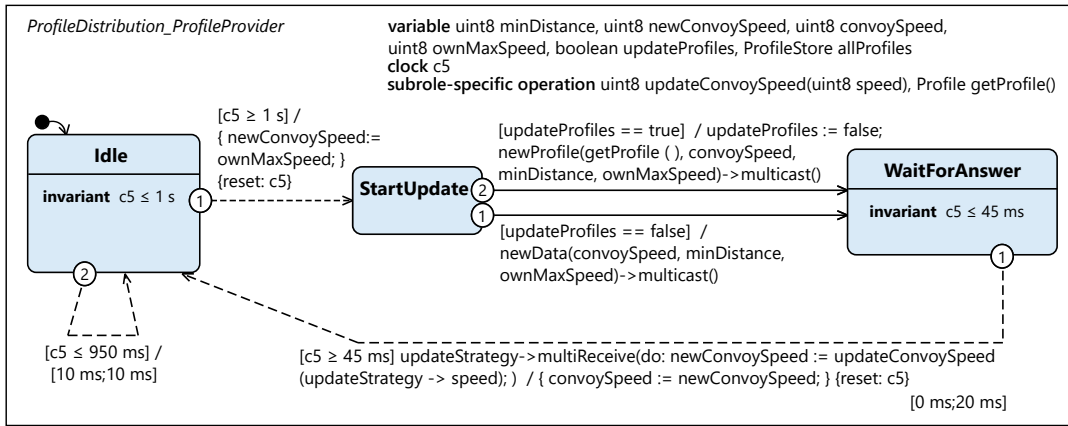


Figure 3.27: Variant 2 of Role ProfileProvider of RTCP ProfileDistribution that uses One-To-Many Communication Schemata

attribute. The original transition from adaptation.DeleteSR to Idle executes the operation deleteSRProfiles. Our variant does not execute this operation at this transition but at the two transitions that have the state DeleteSR as its target (we renamed this operation to deleteProfiles). This change is necessary due to two reasons: First, our concepts do not enable to store the currently active subrole as Heinzemann does it with the variable curSubRole. Therefore, we cannot call the operation with the current subrole as a parameter but have to call it within the subrole. Second, our concepts currently do not enable to call a subrole-specific action without sending and receiving a message. The transition from adaptation.DeleteSR to Idle does not send or receive a message. Therefore, we move the operation call to the two transitions before. Moreover, we use the attribute retryAfter at the schema singlereceive. Therefore, several transitions might have to wait when a message arrives. As a consequence, our variant does not exactly match the behavior of the original. Both semantic deviations do not effect the other role.

Concerning hypothesis H2, our variant needs 12 states, 16 transitions, and 2,283 characters compared to 19 states, 27 transitions, and 2,196 characters in the original RTSC.

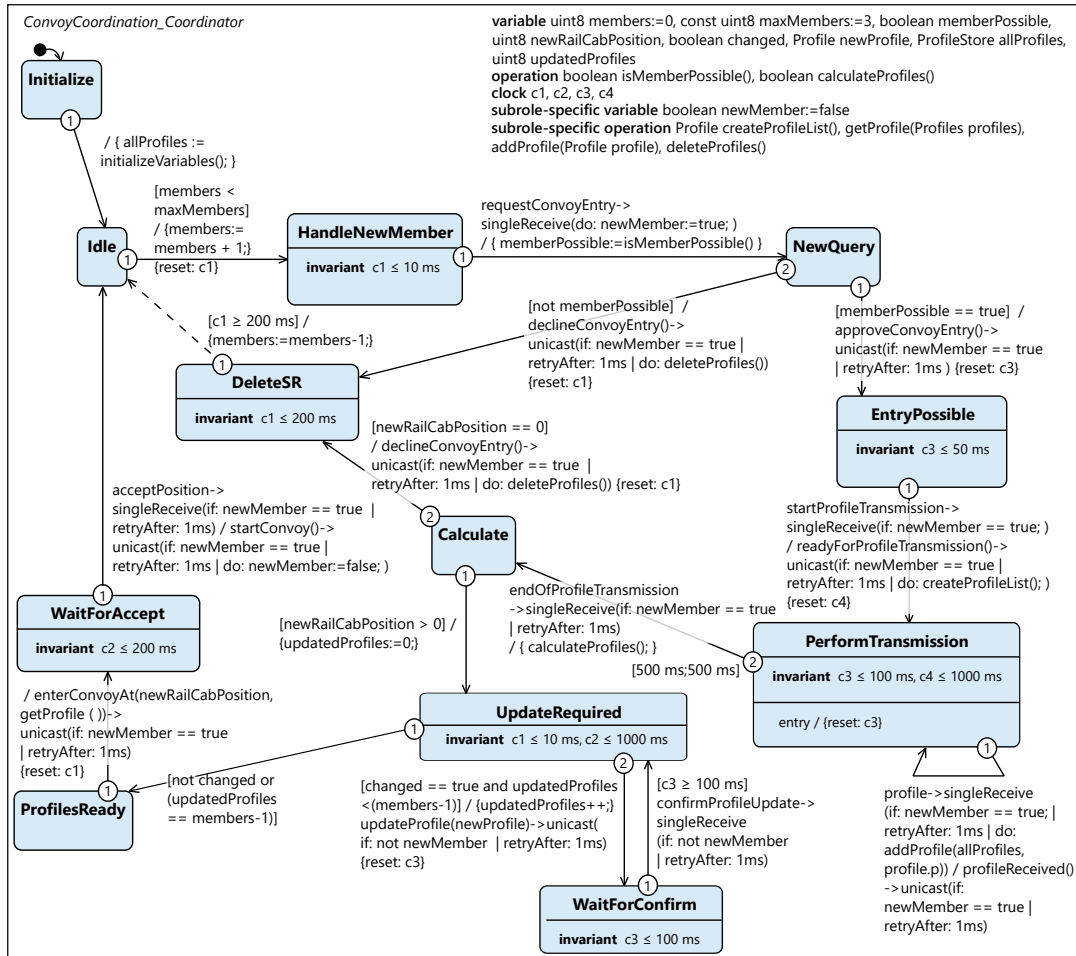


Figure 3.28: Variant of Role Coordinator of RTCP ConvoyCoordination that uses One-To-Many Communication Schemata

3.8.5 Threats to Validity

According to Runeson et al. [RHRR12], we classify our threats to validity into three parts: construct validity (is the case study designed correctly to answer the hypotheses?), external validity (what hinders us to generalize our findings?), and reliability (would the result be the same with another set of researches?). In the following, we discuss our threats based on this classification schema and report how we consider these threats in the design of our study. Noteworthy, Runeson et al. also define a fourth class called internal validity that we do not need as we do not examine causal relations.

3.8.5.1 Construct Validity

- We could have executed an alternative evaluation approach: In our case study, we defined that we collect existing RTCPs with explicit subroles and try to model them with one-to-many communication schemata. Instead, we could have defined requirements for various one-to-many RTCPs. Based on these requirements, we could have modeled two variants of the same RTCP: one variant using explicit subroles, the other variant using one-to-many communication schemata. Afterwards, we could have compared the variants.
- All four RTCPs that we use for our evaluations have been created without knowing our concept for one-to-many communication schemata. However, the RTCPs PositionTransmission and Distribution have influenced our concepts for one-to-many communication schemata. Nevertheless, the RTCPs ProfileDistribution and the variant of RTCP ConvoyCoordination by Heinzemann were not familiar to us before we executed this case study.
- Concerning hypothesis H1, I created the variants by myself. Therefore, our evaluation would have more significance if the specification of the variants would have been executed by multiple software engineers (preferable from different companies), who are experts in CPSs and who were not involved in designing the one-to-many communication schemata.
- Concerning hypothesis H2, we compare the compactness of two models by counting their number of states, their number of transitions, and their number of characters only. We could have been more precise by also counting the number of required channels, clocks, variables, operations, and regions. Moreover, we only compared the visual diagrams of the models. Instead, we also could have compared the number of elements in the tree model editor or the number of tags within the XML serialization of the model.

3.8.5.2 External Validity

- We only consider four examples that use an explicit subrole RTSC. These examples are defined by ourselves or colleagues and not taken from real industry examples. Even though we consider our examples as realistic, other realistic examples (from industry) could exist that we cannot express using our one-to-many communication schemata.

3.8.5.3 Reliability

- We might have made mistakes in our manual comparison of the semantics between the original RTSC and our variant during the validation of hypothesis H1. Thus, our conclusions may be wrong. We reduce the possibility of this threat via triangulation [RHRR12] as I did not compare the semantics only by myself. Instead, a student of our research group, who has one year experience with one-to-many communication schemata, compared the semantics as well.

3.8.6 Analyzing the Results

Our case study demonstrates that we are able to express existing multi role RTSC that use an explicit subrole RTSC using our six one-to-many communication schemata. For two RTCPs, we were able to model a variant that matches the semantics exactly. However, concerning the other two RTCPs, we only were able to define variants where the semantics are similar. Nevertheless, we consider our hypothesis H1 as fulfilled as we were still able to model variants that define the same message exchange behavior. As a consequence, we conclude that one-to-many communication schemata are a useful alternative when modeling multi roles (concerning the expressiveness).

Concerning hypothesis H2, our variants always require significantly less states and transitions. Moreover, the multi roles `PositionTransmission.Sender` and `Distribution.Distributor` also require less characters. However, the first variant of the multi role `ProfileDistribution.ProfileProvider` requires more characters (1,168) than the original (937). Only the second ones requires less (732). The multi role `ConvoyCoordination.Coordinator` requires slightly more characters (2,283) than the original (2,196). One consequence, of using one-to-many communication schemata is that transition labels require more characters and are, therefore, more complex. Nevertheless, we still consider hypothesis H2 as fulfilled due to the significantly less states and transitions. Therefore, we conclude that we can model multi roles in a more compact manner using one-to-many communication schemata.

In summary, it seems that our six one-to-many communication schemata enable to model one-to-many RTCPs on a more abstract level and in a more compact manner. Consequently, we expect that designing such RTCPs should be getting easier and more efficiently while the understandability of such RTCPs should increase as well.

3.9 Limitations

Our new specification of RTCPs underlies the following conceptual limitations:

- We define that all roles start at the exact same time. However, this might not always be realistic, e.g., two independent systems may not be able to start their roles at the same point in time.
- Neither do we support many-to-many RTCPs nor do we support RTCPs with more than two roles. We forbid both due to the state-space explosion problem when model checking such complex RTCPs (see Section 3.3.6).
- The explicit QoS assumption *maximum message transmission delay* only defines the delay between sending the message until storing it into the incoming message buffer. Currently, the RTCP specifier cannot define a maximum delay between storing the message in an incoming buffer until consuming it within the role's RTSC.
- The explicit QoS assumption *preserve message order* concerns all incoming message buffers of all roles. It may be possible that the software engineer wants to specify this for each incoming message buffer separately. Then, this assumption would be specified at each message buffer.
- We do not provide a concept to normalize a transition that uses a one-to-many communication schema *and* that defines a synchronization via a synchronization channel as one of its conditions. The reason is that our concept always adds a synchronization

to the transition and MECHATRONICUML does not allow to specify more than one synchronization at a transition.

- We do not enable to transform a multi role RTSC that uses an explicit subrole region into an RTSC that uses one-to-many communication schemata; neither do we provide a process for doing this.

We do not know about implementation-specific limitations.

3.10 Related Work

Our related work discussion focuses on approaches that provide a DSL for the domain of CPSs or related domains and enable that roles or components may exchange messages asynchronously. In particular, we are interested in approaches that provide explicit message buffers in their DSL, state fixed and variable QoS assumptions for the message transfer, and/or provide explicit model elements to specify one-to-many communication dependencies.

We distinguish related work in three different topics. First, in Section 3.10.1, we discuss related work that also defines a DSL for contracts and protocols concerning (asynchronous) message exchange. Second, in Section 3.10.2, we discuss approaches that focus on the assumed and guaranteed behavior of connectors but that do not focus on the behavior of each communicating role. Third, in Section 3.10.3, we discuss component models that focus on an asynchronous message exchange but that do not provide explicit concepts like protocols or contracts.

3.10.1 Contracts and Protocols

Dragomir et al. [DOP15; Dra14] provide an approach for a contract-based modeling and formal verification of timed safety requirements within SysML [SysML]. Their contracts are similar to RTCs as they define an asynchronous message exchange of roles whereby each role describes which behavior it assumes and guarantees. The behavior is modeled using state machines as well. Due to the asynchronicity, each role that may receive messages has a defined incoming FIFO message buffer. Moreover, their buffers have an unlimited size at the modeling level. However, in order to apply formal verification, the software engineer has to define an upper bound as an input parameter. Moreover, they make no statement which behavior is executed if the software engineer defines a buffer size such that it is possible that a message arrives but the buffer is already full. In contrary, we define that the maximum buffer size is part of the design model as this property does significantly influence the verification result and should be stored permanently because it is a relevant information in later development phases, e.g., the code generation. Moreover, we enable to choose between three discardment strategies (discard incoming, discard oldest message in buffer, discarding shall never happens) as we identified relevant use cases for all of them. As a another limitation, Dragomir et al. do not enable to specify QoS assumptions concerning the reliability, the message delay, or the ordering nor do they specify their fixed assumptions on the QoS. Contrary to us, they also do not focus on one-to-many communication.

Hadj Kacem et al. [HHD09] define coordination protocols for self-adaptive systems using Coloured Petri Nets. Like us, their protocols consist of interacting roles that exchange messages. The authors consider the need for buffering the messages. Therefore, they represent

input and output message buffers, which are ordered via the FIFO principle, by using places of the petri net. However, in contrast to us, they do not enable to store messages of different types within one buffer. Furthermore, it appears that their buffer size is infinite. Therefore, they do not need to consider the behavior of the buffer if a new message arrives but the buffer is already full. Concerning the properties of the connector, they assume that it is always reliable, i.e., message loss, message duplication, and message permutation is not possible. Moreover, the authors enable to specify the one-to-many dependencies unicast and multicast explicitly. For example, a transition with the label `AAxMsgMNG.mult(AA.all(),1'ReadyStReq)` defines that if the transition fires, then the message `ReadyStReq` is sent to all receivers of type `AA`. Nevertheless, their protocols are not directly applicable to the domain of CPSs in general as they do not consider the aspect of real-time requirements. Moreover, they consciously make several abstractions that are sound for their domain but not for the domain of CPSs, e.g., the infinite buffer size and their QoS assumptions of the reliable role connector. Concerning the one-to-many communication dependencies, they do not provide other one-to-many dependencies, e.g., like our one-to-many communication schemata `multireceive` or `loadbalancing`. Therefore, important use cases for a one-to-many communication are not considered as well.

The Foundation of Intelligent Physical Agents (FIPA) and the Object Management Group (OMG) propose an extension of the UML [UML] called `AGENTUML` [BMO01; OPB01] for designing and analyzing multi-agent systems. In particular, `AGENTUML` enables the software engineer to model so-called agent interaction protocols (AIPs) for multi-agent interaction, which are similar to UML sequence diagrams. Among others, these diagrams consist of agent roles, multi-threaded lifelines, guards (e.g., data and time constraints), and asynchronous messages. However, despite the asynchronicity, AIPs do not provide means to specify the properties of incoming message buffers. Moreover, the authors do not state whether the concept of message buffers is considered within their semantics definition (respectively their analysis) at all. Furthermore, the software engineer may choose whether a message transfer is delayed or not but cannot define concrete values, e.g., the maximum transmission delay. Other possibilities for stating QoS assumptions on the message transfer are not provided. In addition, `AGENTUML` enables to explicitly specify one-to-many communication dependencies within an AIP. In particular, the software engineer may split the lifeline into several ones via three different connector types: `AND`, `XOR`, and `OR`. The connector type `AND` enables to specify a multicast (i.e., the message is sent to all receivers), `OR` enables to specify a multicast with conditions (i.e., the message is sent to all receivers that fulfill the condition), and `XOR` a unicast (i.e., the message is sent to exactly one receiver). Our approach supports all three communication dependencies that `AGENTUML` offers. Moreover, we provide additional communication dependencies (e.g., `multireceive` and `loadbalancing`) that are relevant for our domain.

Asplund et al. [AMB+12] provide a formal approach to autonomous vehicle communication. In particular, they focus on the safety-critical coordination between independent vehicles while including message exchange, continuous time, and continuous changes of the environment. The number of vehicles and the number of messages are unbounded. For proving the correctness of a coordination scenario, they propose to use Satisfiability Modulo Theories (SMT) solvers. For doing this, they formalize an already existing distributed coordination protocol by designing an automata model and additionally stating constraints that the system has to fulfill. Then, they use the Z3 theorem prover [MB08] to verify safety properties. In

contrast to our new definition of RTCPs, the authors do not focus on asynchronous message exchange and QoS assumptions. Moreover, they do not have the goal to provide a DSL that eases the work of the software engineer. Therefore, their automata language does not provide explicit model elements for message buffers. Moreover, QoS assumptions like message delay or message ordering are not considered. The authors define that an infinite number of vehicles may communicate with each other via messages. They also give use cases where it is necessary that a vehicle receives a message from all other vehicles. However, they only provide capabilities to send and receive a single message and not—like us—to provide more sophisticated concepts, e.g., a multicast or a multireceive.

3.10.2 Behavioral Connectors

“[Lopes et al.] [LWF03] focus on the specification of connectors that enable a communication between components. They develop an approach of so-called higher-order architectural connectors that take other connectors as parameters. A connector consists of a set of roles and a so-called glue specification that contains another connector specification or a set of services, e.g., buffering, compression, fault-tolerance, security, and monitoring. The behavior specification of roles is an aspect that [Lopes et al.] do not focus on: they provide means to define role behaviors but do not enable to specify real-time constraints. The authors enable to specify the behavior of their services using their own textual DSL. For example, they enable to specify a FIFO-message buffer with a fixed size for asynchronous communication. The buffer prevents that a sender role can send a message if the buffer is already full (which contradicts the idea of asynchronous communication). In addition, the buffer prevents to read a message if the buffer is currently empty. [Lopes et al.] only focus on one-to-one communication. Moreover, they do not state [whether] a [software engineer] may be able to specify variables concerning QoS assumptions. Though, it appears that this should be possible.” [*DPP+16]

“The exogenous coordination language Reo [ABRS04; Arb06; MSKA14] enables the compositional construction of component connectors that coordinate the interactions among the concurrent roles that participate in this communication. Reo enables many-to-many communication. Furthermore, a [software engineer] may define the behavior of the connector, i.e., when and how each message is transmitted and from which sender to which receiver. This can be done via their DSL timed constraint automata. In particular, a [software engineer] is able to define various kinds of communication, e.g., asynchronous message exchange including FIFO-buffering. Concerning the buffering, the [software engineer] may choose between an unbounded buffer or a buffer with a fixed size, where the oldest message in the buffer is discarded if the buffer is full and a new message arrives. In addition, the [software engineer] may encode QoS assumptions like a message delay via timed constraint automata as well.” [*DPP+16] Reo enables to specify one-to-many communication schemata by encoding them as connectors. We did not find connectors that already specify the same behavior as our schemata iterate (see Appendix A.5) and our two variants of loadbalancing (see Appendix A.6). We assume that we could define them using Reo. However, specifying one-to-many communication schemata as connectors means that the same one-to-many communication schema has to be used for all message exchanges. In contrary, we enable to send each message using a different schema.

“In contrast to both approaches, we focus on the real-time behavior specification of the connector *and* the roles. Moreover, we do not provide means to the [software engineer] to

specify its connector behavior via complex rules or automata. Instead, we already provide a set of useful [model] elements, i.e., buffers and a set of QoS assumptions. Therefore, the [software engineer] may select from a set of useful properties (e.g., if the connector is reliable or unreliable) of the domain of [CPSs]. In our opinion, this facilitates the work of designing an RTCP within this domain as the set of useful properties is in our opinion limited. However, corner cases may exist where our pre-defined list of properties is too restrict.” [*DPP+16]

3.10.3 Component Models

Hošek et al. [HPB+10] surveyed component models for the domain of embedded real-time systems. Only few of them focus on the asynchronous message exchange. In the following, we discuss the most related component models in detail. In addition, we also discuss MARTE [MARTE] (Hošek et al. left it out as it is a modeling approach only without execution support).

The AUTomotive Open System ARchitecture (AUTOSAR) [AUTOSAR] is an international development partnership of several companies from the automotive domain. Its goal is to provide an open industry standard for the architectural design of automotive systems. AUTOSAR provides a component model to specify the (internal) structure of the system but does not provide means to model the internal component behavior. Furthermore, AUTOSAR has no comparable concept to RTCPs, i.e., they do not provide capabilities to explicitly define and verify the asynchronous message exchange of components including real-time constraints. Nevertheless, AUTOSAR defines that its components may interact via ports. In particular, a port can provide an interface that may specify operations and data elements. Moreover, AUTOSAR provides several kinds of interfaces that represent typical use cases in distributed systems, e.g., *client-server* and *sender-receiver*. All interface kinds enable a one-to-many communication. The client-server interface enables bidirectional communication: the server provides a set of operations that many clients may request. In contrary, the sender-receiver interface only enables unidirectional communication: the software engineer defines which data elements shall be sent from one sender to many receivers. Though, the sender is not aware whether receivers exist, i.e., it only broadcasts the data. Using RTCPs, we are able to support all interface kinds that AUTOSAR provides. Moreover, AUTOSAR provides two means to buffer incoming data elements: *queued*, which is a FIFO-buffer with a fixed buffer size, and *last-is-best*, which only buffers the last incoming data element. Like us, they do not provide means to buffer outgoing data elements. Moreover, AUTOSAR also defines the behavior of the message buffer if it is already full but a new message arrives: buffers of type *last-is-best* always discard the message that is already in the buffer, while buffers of type *queued* always discard the incoming message. In contrary, our definition does not distinguish different types of a buffer but the software engineer may decide on its own what the size of the buffer and its strategy for discarding a message is. Therefore, we support all buffer types of AUTOSAR but also enable to specify other variants as we identified use cases that make them necessary in our domain. In addition, AUTOSAR enables that the software engineer can specify filters that are applied before the data element is stored in the buffer. For example, he may specify that only the n -th message is accepted, where $n \in \mathbb{N}$. As another example, he may filter data elements based on their value, e.g., the value must be equal (greater than / lower than) to a pre-defined constant, the value must be within a pre-defined upper and lower limit, or the value must be different to the value that was the last time stored in the buffer. In principle, we could also

integrate such filtering mechanisms. However, we did not encounter use cases while modeling RTCPs that make them necessary. One reason for this is that AUTOSAR has no comparable concept to RTCPs: Via RTCPs, the software engineer always models the behavior of both roles. Therefore, instead of filtering a message, he can directly specify that the message shall not be sent at all. This avoids unnecessary message exchange. Until now, AUTOSAR does not enable the software engineer to specify QoS assumptions concerning the reliability, the message delay, or the ordering. Moreover, as AUTOSAR does not provide a behavior specification language, it also does not provide one-to-many communication schemata.

The UML profile for the Modeling and Analysis of Real-Time and Embedded systems (MARTE) [MARTE] enhances the UML for specifying and verifying real-time and embedded systems. MARTE defines a component model for specifying the structure of the system. Components may exchange messages (i.e., operations, signals, or primitive data) via their so-called InteractionPorts. There exist two kinds of these ports: FlowPorts and ClientServerPorts. ClientServerPorts may provide and/or require operations or signals while FlowPorts may transport primitive data. The software engineer may choose between two different semantics for FlowPorts: pull and push. If data arrives and push is chosen, then internal behavior is triggered and the data is directly processed (without further buffering). Instead, if data arrives and pull is chosen, then no internal behavior is triggered but the data is buffered via so-called DataPools. Therefore, MARTE also supports explicit model elements for incoming message buffers like our approach. However, in contrast to us, actions in MARTE can read from the buffer but do not necessarily have to remove the message from the buffer. Until now, we do not provide this option as we only identified use-cases where the software engineer always wants to read *and* remove a message from the buffer. Concerning the specification of DataPools, MARTE defines the following: The software engineer may define a fixed buffer size but may also define that the size is infinite. Moreover, the software engineer may choose between two pre-defined buffer semantics: FIFO and LIFO. In addition, the software engineer may also define its own buffer semantics using state machines. In addition, MARTE's FIFO and LIFO semantics defines that always the oldest message in the buffer will be removed if the buffer is full and a new message arrives. Though, the software engineer may also specify another discard strategy by encoding it into a user-defined semantics. Contrary, we only support a fixed buffer size as an infinite size abstracts in our opinion too much from real-world circumstances and hinders us to apply model checking with UPPAAL. Moreover, we only support the semantics FIFO as we did not identify use-cases where other semantics (fixed or user-defined) are useful. Furthermore, we provide three different buffer overflow avoidance strategies as we identified realistic use cases for each one of them. MARTE does not provide explicit model elements within their behavioral models for specifying a one-to-many communication. Like AUTOSAR, MARTE does not enable the software engineer to specify QoS assumptions of the message exchange nor does it provide explicit model elements to define one-to-many communication schemata.

The Architecture Analysis and Design Language (AADL) “provides formal modeling concepts for the description and analysis of application systems architecture in terms of distinct components and their interactions.” [AADL, p. 4] One of their goals is the specification and analysis of real-time embedded systems. For specifying the architecture, AADL provides a component-model. Among others, components may interact with each other by exchanging messages (data or events) via three port kinds: data ports, event ports, and event data ports. Data ports enable an exchange of state data but do not buffer more than one information.

Moreover, the software engineer may decide whether the connection between these ports is immediate or delayed. However, the software engineer may not define an explicit value of the transmission delay. Event ports enable an exchange of asynchronous events (e.g., by subprograms and threads) that are buffered according to FIFO semantics at the incoming port. Event data ports enable an exchange of messages that contain data. They are buffered at the receiving port according to FIFO semantics as well. Incoming buffers have a maximum size. Moreover, according to [CBF+10], ADDL provides a property for handling an imminent buffer overflow (i.e., a new message arrives but the buffer is already full). In particular, the software engineer may choose that the buffer shall discard the oldest message in the buffer, the incoming message, or that this situation may never happen. Thus, they provide the same choices as we do. In contrast to them, our approach enables a bidirectional communication but only supports one-to-one and one-to-many communication. Concerning their message buffer specification, they do not state whether a port may have multiple buffers and whether each buffer may contain messages of different types. In addition, we enable to model the behavior of each participating role including explicit model elements for the one-to-many communication. Moreover, besides the communication delay, a software engineer is not able to specify additional QoS assumptions.

The Progress Component Model (ProCom) [BCC+08; VSC+09] aims at the development of real-time embedded software systems for the domain of vehicular systems, automation, and telecommunication. Their components are able to exchange messages via so-called message ports. These ports are either of kind input or output, i.e., a port may either send or receive messages. Furthermore, message ports are not directly connected with each other but via a message channel. As a consequence, a software engineer can annotate information concerning the message exchange explicitly at this channel rather than on one of the several message ports. Message channels enable many-to-many communication as they may be connected to several input and output message ports. The buffering of messages must be defined by the software engineer in the internal behavior of the receiving subsystem. Concerning the internal behavior, ProCom enables to use timed automata extended by urgent and prioritized transitions and delay intervals for states. Like MARTE, ProCom does not enable the software engineer to specify QoS assumptions concerning the reliability, the message delay, or the ordering of the messages. In contrary, roles of an RTCP do not only enable unidirectional but also bidirectional message exchange. This enables to define non-trivial message exchange behaviors. Moreover, we provide explicit model elements to specify the incoming message buffers, the behavior of each role, and the QoS assumptions of the message transfer. As a consequence, we provide means to formally verify the structure and the behavior of the specified model as a whole. As a limitation, RTCPs currently support one-to-one and one-to-many communication only. Though, in contrary to ProCom, we provide explicit modeling concepts to specify one-to-many communication dependencies.

3.11 Summary

In the following, we summarize to what extent this thesis and the related approaches that we discussed in Section 3.10 fulfill the requirements that we stated in Section 3.2. As a reminder, the requirements were based on our own experience and are not proven, e.g., by a systematic literature review or requirements workshops.

	<i>this thesis</i>	<i>Dragomir et al.</i>	<i>Hadj Kacem et al.</i>	<i>AGENTUML</i>	<i>Asplund et al.</i>	<i>Lopes et al.</i>	<i>Reo</i>	<i>AUTOSAR</i>	<i>MARTE</i>	<i>AADL</i>	<i>ProCom</i>
R1	✓	✓	✓	✓	(✓)	✓	(✓)	(✓)	✓	✓	✓
R2	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓
R3	✓	(✓)	✓	✗	✗	✗	✓	✓	✓	✓	✗
R4	✓	✗	✗	(✓)	✗	(✓)	✓	✗	✗	(✓)	✗
R5	✓	✗	✓	✗	✗	(✓)	✗	✗	✗	✗	✗
R6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R7	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓
R8	✓	✗	✓	✓	✗	✗	(✓)	(✓)	✗	✗	(✓)

Table 3.1: Comparison of the Requirements Fulfillment for Verification Protocols of CPSs

We provide an overview of the requirements fulfillment in Table 3.1 (Legend: ✓ = fulfilled, (✓) = partially fulfilled, ✗ = not fulfilled). As the table shows, this thesis fulfills all requirements that we stated at the beginning of this chapter. Moreover, no other approach fulfills all requirements. A combination of the related approaches could theoretically fulfill all requirements but would—according to our experience—require huge conceptual efforts.

The most relevant requirement is R1, which states that the DSL should enable to specify the asynchronous message exchange between the roles. This thesis and most of the related approaches fulfill R1. Asplund et al., Reo, and AUTOSAR only fulfill it partially: Asplund et al. does not enable to exchange messages asynchronously while Reo and AUTOSAR only enable to specify the structure but not the behavior of the message exchange.

The requirements R2 (enable to define real-time requirements for the message exchange) and R3 (enable to specify incoming message buffers using explicit model elements) are fulfilled by this thesis and several other approaches. In particular, we enable to specify real-time requirements using RTSCs (see Section 2.2.2 and 3.5) and by assuming a maximum message delay (see Section 3.3.2). Moreover, we present our message buffer concept in Section 3.3.4. Even though some of the related work support to specify incoming message buffers, no related approach supports the complete buffer functionality that we require (i.e., explicit buffer size, buffer overflow avoidance strategies, multiple messages per buffer, multiple buffers per role). Dragomir et al. only fulfills R3 partially as they consider the need of asynchronous message buffers but they only support the specification of a buffer size for the purpose of formal verification. Moreover, they do not define what behavior shall be executed if the buffer is full but new messages arrive (or whether this may never happen).

The assessment of requirement R4 (enable to specify RTCP-dependent QoS assumptions) and requirement R5 (provide all necessary protocol-independent QoS assumptions) is subjective as it may depend on the platform which QoS assumptions are necessary. In our opinion, only this thesis and Reo completely fulfill R4. In particular, this thesis enables to specify the protocol-specific QoS assumptions of the domain of CPS (message delay, reliability, message order preservation; see Section 3.3.2). Reo requires that the software engineer specifies timed constraint automata, which is a time-consuming and error-prone

concept. Instead, we provide attributes where the software engineer only has to define their values. AGENTUML and AADL only enable to specify whether the communication is delayed or not (specifying a concrete delay is not possible). Lopes et al. only enable to define the capacity of the connector, which we assume is unreliable.

In our opinion, only this thesis and Hadj Kacem et al. fulfill the requirement R5. In our case, we present the assumptions in Section 3.4. Lopes et al. only partially fulfill R5 as they only state that they require for the sender “[...] that it does not produce another message before the previous one has been processed.” [LWF03]

Concerning requirement R6 (focus on functional concerns and not on their technical realization), this thesis and all approaches that we discussed fulfill the requirement as they all enable a platform-independent modeling of coordination protocols that does not focus on the technical realization. Furthermore, this thesis and most of the related approaches fulfill requirement R7 (enable to specify parametrized messages).

Finally, concerning requirement R8 (provide explicit one-to-many communication support), only this thesis, Hadj Kacem et al., and AGENTUML fulfill this requirement. Though, Hadj Kacem et al. and AGENTUML only support the schemata unicast and multicast. Reo only fulfills this requirement partially as it only enables to specify a schema for the complete connector. Instead, we can define a schema for each transition of a multi role that sends or receives a message. Furthermore, AUTOSAR and ProCom fulfill R8 only partially as well as they only enable to specify the structure of a one-to-many communication but not the behavior.

DOMAIN-SPECIFIC VERIFICATION PROPERTIES FOR RTCPS

In this chapter, we introduce a new domain-specific verification property language (DSVPL) called MECHATRONICUML TCTL (MTCTL) for specifying verification properties for a verifiable coordination protocol of a CPS while respecting the capabilities of the selected model checker. In particular, we start with the contributions of this chapter in Section 4.1. Then, we list and explain the stakeholders as well as the requirements for our DSVPL in Section 4.2. Afterwards, we introduce MTCTL in Section 4.3. Subsequently, we present and explain a set of application-independent verification properties for MTCTL in Section 4.4 and provide a translation from MTCTL properties into English sentences in Section 4.5. Next, we give insights into our implementation in Section 4.6. We evaluate MTCTL in Section 4.7 before we state the limitations of our approach in Section 4.8. Then, we discuss related work in Section 4.9. Finally, we summarize this chapter in Section 4.10.

The initial set of stakeholders and requirements for MTCTL (Section 4.2) were defined in the seminar thesis of Rebekka Wohlrab [SDP+15, p. 11–20]. Moreover, a first draft of MTCTL (Sections 4.3 to 4.6) has been contributed by the project group Cybertron [BCD+14]. Both, the seminar thesis of Rebekka Wohlrab and the project group Cybertron, were supervised by me.

4.1 Contributions

The main contributions of this chapter are as follows:

- We define a set of five requirements that a DSVPL for RTCPs shall provide.
- We define a new DSVPL called MTCTL for specifying all relevant verification properties of RTCPs that shall be verified using the model checker UPPAAL. In particular, this language enables to make statements about the whole RTCP and not only its RTSCs, e.g., it may specify properties concerning the message transfer and the message buffers.
- We define a set of seven application-independent verification properties (called default properties) that should always hold for all RTCPs.
- We provide a translation from MTCTL to English sentences. This shall enable all involved stakeholders that do not necessarily have model checking expertise, e.g., requirements engineers, to understand the MTCTL properties and, therefore, link them to other artifacts, e.g., to the requirements defined in earlier development phases.

- In an evaluation, we show that we are able to state a majority (ca. 90%) of realistic informal requirements as formal MTCTL properties and that we preserve the semantics of the MTCTL properties when translating them to valid English sentences.

4.2 Stakeholder and Requirements

In this section, we define the stakeholders for a DSVPL in general. Then, we formulate and justify requirements that a DSVPL and the development process have to fulfill for a given DSL and a given model checker. In our case, the DSL enables to specify RTCPs and the model checker in use is UPPAAL.

We distinguish between two groups of stakeholders. The first group—the *DSVPL specifiers*—consists of software engineers that specify the verification properties. They must be able to translate (informal) requirements as domain-specific verification properties and to come up with additional verification properties that are not explicitly stated as requirements. Moreover, they must be able to specify the properties correctly. Software engineers that specify these properties are typically either the developers or the testers. Thus, we assume that this group has significant experience about the domain and model-driven software engineering and at least basic experience in model checking (including verification property languages like TCTL [ACD93]). For example, they do not need to know how to specify domain-specific aspects like asynchronous message exchange on the model checker level.

The second group of the stakeholders—the *DSVPL readers*—are engineers that only need to understand the verification properties. Representatives of this group are systems engineers and requirements engineers. Among others, they must be able to validate whether the properties are useful and whether they can be linked to top-level requirements. Thus, we assume that this group has significant experience about the domain but it only needs basic experience in model-driven software engineering and model checking.

In the following, we present the requirements a DSVPL shall at least have. We order the requirements by their priority, where the first is the most important requirement. The requirements and their priorities are based on our experience. Therefore, we do not claim that this is a complete list. Our requirements are based on Wohlrab [SDP+15].

In our opinion, a DSVPL shall . . .

- R1** be able to express the relevant properties of the domain,
- R2** be compatible with the verification property language the underlying model checker accepts,
- R3** be directly integrated into the DSML, i.e., providing ways to reference the design model (e.g., states and messages), and
- R4** be understandable for stakeholders that are no experts in model checking.

In addition, we require that a software development method that uses a DSVPL shall ...

- R5** provide concepts to make experience-based specification knowledge accessible to inexperienced DSVPL specifiers.

In the following, we will justify our requirements. Concerning requirement R1, we require that the DSVPL must be able to express all relevant verification properties of the domain. In our case, the domain is the asynchronous message exchange behavior of CPSs. If R1 is not fulfilled, then the model checking is not hidden anymore because the DSVPL specifier has to add inexpressible verification properties via the model checker's input language. As we want to avoid this, R1 must hold. This means that all necessary kinds of verification properties should be supported. In our case, these kinds are safety, reachability, and liveness (see Section 2.1.2). In particular, the DSVPL must be able to express properties concerning the asynchronous message exchange, real-time constraints, and data variable values.

Concerning requirement R2, the DSVPL is—in the best case—as powerful as the property language of the model checker (we assume that the chosen model checker is appropriate for the formal verification of coordination protocols of CPSs). Thus, the DSVPL should not support to specify properties that cannot be checked via the underlying model checker. If this is fulfilled, then the DSVPL specifier can use the full expressiveness of the DSVPL without considering or knowing what the model checker accepts (this is important for making the model checking hidden).

We need requirement R3 to preserve the consistency between the DSVPL and the DSL model. Thus, changes in the DSL model shall automatically adapt the DSVPL properties. This reduces the possibility for incorrect properties and, therefore, eases the work of the DSVPL specifier. In particular, if a referenced design model element is renamed or moved to a new hierarchy level, then the MTCTL property shall adapt as well. Moreover, if a referenced design model element is being deleted, then the property should highlight this.

Via requirement R4, we ensure that stakeholders that are no experts in model checking do not misinterpret the DSVPL properties which could lead to the wrong assumption that a certain requirement is fulfilled. This is especially important for DSVPL readers, who do not necessarily have a software engineering background, e.g., the requirement engineers.

Having the possibility to apply model checking does not correlate with a correct or well-designed model. Instead, these aspects depend on the specified verification properties. Therefore, defining a list of the relevant verification properties for the system is a crucial task. Requirement R5 states that the software development method shall provide concepts to make experience-based knowledge accessible to inexperienced DSVPL specifiers. If this requirement is fulfilled, then the list of properties is more meaningful and, therefore, the quality of the software should rise (if the model checker's verification result is that these properties hold).

4.3 MTCTL - A Domain-Specific Verification Property Language for MechatronicUML

In this Section, we introduce the DSVPL MECHATRONICUML TCTL (MTCTL) for the DSL MECHATRONICUML. The goal of MTCTL is to fulfill all requirements that we stated in Section 4.2 such that an RTCP may be verified using the model checker UPPAAL via domain-specific model checking (DSMC, see Chapter 5).

The role of MTCTL in the context of model checking RTCPs using UPPAAL is illustrated in Figure 4.1. As depicted, a verifiable element of MECHATRONICUML (i.e., an RTCP, a component, or a component instance configuration) is the model under analysis and MTCTL

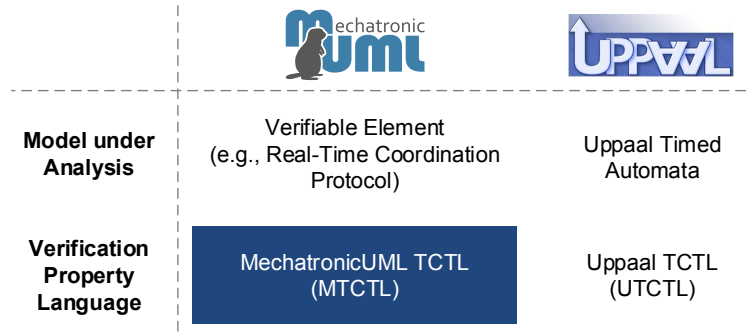


Figure 4.1: MTCTL: the Counterpart to UTCTL at the Level of MECHATRONICUML

is the verification property language that we define in the context of the verifiable elements. While a verifiable element of MECHATRONICUML is the counterpart of UPPAAL timed automata, MTCTL is the counterpart of UPPAAL TCTL (UTCTL).

MTCTL plays an important role for hiding the model checking details from the software engineer completely. Without this DSVPL, the software engineer has to use UPPAAL’s verification property language UTCTL, which references UPPAAL’s timed automata models. This would force the software engineer to learn UTCTL and UPPAAL’s timed automata models. Moreover, it would require that the software engineer understands the translation from RTCPs to timed automata in order to specify correct verification properties. In summary, having no DSVPL not only UTCTL would be exposed to the software engineer but also the timed automata and the translation from MECHATRONICUML to UPPAAL.

As indicated by its name, MTCTL is based on TCTL and especially UTCTL (see Section 2.1.2) but also uses first-order predicate logic. In particular, an MTCTL expression can be seen as a predicate logic expression with additional temporal quantifiers like AG (always globally). For example, concerning the RTCP Overtaking (see Figure 3.6), the requirement “If the overtaker is in state Overtaking, then the overtakee is in state NoAcceleration” may be expressed in MTCTL using the property **AG** (`stateActive(Overtaker.Overtaking) implies stateActive(Overtakee.NoAcceleration)`);. This property uses on the one hand the temporal quantifier **AG** that is known from TCTL and UTCTL. On the other hand, it uses an MTCTL-specific predicate `stateActive` that has a state of an RTSC as an argument and returns whether the state is currently active or not.

We use TCTL as a basis instead of structured English sentences (like [KC05]) or other existing languages like OCL [OCL] for three reasons: First, we realized that the same property stated in our TCTL-based language is in most cases significantly shorter (i.e., it has less characters) than a natural English sentence (see Appendix C.2 for a comparison). Thus, it is—in our opinion—faster to read/understand and faster to write for the MTCTL specifier. Second, TCTL is especially designed for model checking but alternative query languages (e.g., OCL) are not. Thus, it exactly provides the language features that we need. Third, as stated in Section 4.2, we assume that the MTCTL specifier has at least basic experience in model checking and, thus, is already familiar with languages like TCTL. Nevertheless, we do not state that our decision for a TCTL-based language is the best solution.

The behavior specification of an RTCP is not only modeled using RTSCs exclusively due to our introduction of incoming message buffers and QoS assumption (see Chapter 3).

Nevertheless, we still require to make statements about these behavior definitions, e.g., if a message is currently stored within a buffer. As a consequence, MTCTL is not only able to make statements about the elements of the RTSC (e.g., a state is active or a transition is firing) but concerning all relevant behavior definitions. We achieve this by defining additional predicates and operations that focus on the behavior outside of an RTSC. For example, concerning the incoming message buffers of an RTCP, MTCTL is able to express that it is possible that an instance of message type *m1* can possibly be stored in the buffer *b1* using the predicate `messageInBuffer`. The resulting property is `EF messageInBuffer(m1, b1)`.

Our goal is that MTCTL is able to express the relevant properties of the domain. However, it is currently unknown which verification properties are relevant and which are not. In fact, we think that this definition may vary between software engineers significantly. As a consequence, in this thesis, we rely on our experience with model checking and RTCPs to define the set of necessary model elements that enable to specify the relevant verification properties. Though, we will argue for all model elements why we consider them as necessary.

We designed MTCTL in such a way that it enables to express verification properties for all verifiable elements of MECHATRONICUML, i.e., RTCPs, atomic components, or component instance configurations. However, in this thesis, we focus on the specification of verification properties for RTCPs only.

Concerning the stakeholders, we refer to the DSVPL specifier that uses MTCTL as the *MTCTL specifier*. Moreover, we refer to the DSVPL reader that shall be able to read and understand MTCTL properties as the *MTCTL reader*.

In the following, we introduce MTCTL in detail. First, we give a simplified EBNF grammar in Section 4.3.1 to briefly introduce the major concepts of MTCTL. Afterwards, we define the basic semantics in Section 4.3.2. Then, we define the temporal quantifiers that MTCTL supports in Section 4.3.3. In Section 4.3.4, we explain which model elements of an RTCP may be referenced by MTCTL. Next, we introduce the supported predicates in Section 4.3.5 and the supported operations in Section 4.3.6. Finally, in Section 4.3.7, we explain how multiple temporal properties may be combined into one MTCTL property.

We do not describe our transformation from MTCTL to UTCTL within this chapter but in Chapter 5 as the transformation from MTCTL to UTCTL depends on the transformation from RTCPs to timed automata.

4.3.1 Simplified EBNF Grammar

We list in the following a simplified grammar for MTCTL in EBNF [ISO14977]. The complete grammar, which is mainly specified in Xtext [Xtext], is given in Appendix E.2. By stating and explaining this grammar, we give an overview of MTCTL and introduce concepts that we introduce in the following sections.

```

MTCTLProperty      = ( TemporalProperty | UnaryProperty |
                        BinaryProperty | SetProperty |
                        true | false ) ';' ;

TemporalProperty    = TempQuantifierProperty |
                        LeadsToProperty ;

TempQuantifierProperty = TemporalQuantifier Statement ;
TemporalQuantifier    = 'AG' | 'AF' | 'EG' | 'EF' ;

```

```
LeadsToProperty      = Statement 'leadsTo' Statement;

Statement            = Statement (BinaryOperator Statement)*;
Statement            = 'not ( ' Statement ')';
Statement            = (Comparison | Predicate | true | false);
Statement            = SetQuantifier+ Statement;

Comparison           = (IntExpression | IntOperation)
                      ComparisonOperator
                      (IntExpression | IntOperation);
ComparisonOperator   = '==' | '>' | '>=' | '<' | '<=' | '!=';

UnaryProperty        = 'not' MTCTLProperty;
BinaryProperty       = MTCTLProperty BinaryOperator
                      MTCTLProperty;
BinaryOperator       = 'implies' | 'and' | 'or';

SetProperty          = SetQuantifier+ (TemporalProperty | true |
                                         false );
SetQuantifier         = ( 'forall' | 'exists' ) VariableBinding;
VariableBinding       = '(' Identifier ':' SetType ')';
```

The informal description of the EBNF grammar is as follows: Similar to UTCTL (see Section 2.1.2), an MTCTL property may be a temporal property. A temporal property is either a temporal quantifier property or a so-called leads-to property. We choose both kinds of properties as they are both supported by UTCTL. Like in UTCTL, a temporal quantifier property defines an explicit temporal quantifier (**AG**, **AF**, **EG**, or **EF**) that is bound to a statement. In contrary, a leads-to property uses the temporal quantifier **leadsTo** to combine two statements with each other. Note that UTCTL uses a different concrete syntax: UTCTL uses $\rightarrow$ while we use the more readable form **leadsTo** as an arrow may be mistaken as an implication. We define MTCTL's temporal quantifiers in more detail in Section 4.3.3.

Like in UTCTL, a statement of MTCTL is side-effect free and results to true or false. It may consist of one or more statements that are connected via the binary operators **and**, **or**, or **implies**. Moreover, like in UTCTL, each statement may be negated via the unary operator **not**.

In UTCTL, a statement may only consist of comparison expressions, the deadlock predicate, or a check whether a state is active by stating the name of the state (see Section 2.1.1). MTCTL also supports the deadlock predicate and comparison expressions. In particular, a comparison compares integer values via the operators $=$, $>$, $\geq$, $<$, $\leq$, and $\neq$. These integer values are either expressed via an integer expression or via an operation that results to an integer. However, in contrary to UTCTL, MTCTL provides additional predicates such that a software engineer cannot only query whether a state is active but also other domain-specific queries, e.g., the predicate **messageInTransit** enables to query whether a specific message is currently in transit. We define the elements that an MTCTL statement may reference in Section 4.3.4. Moreover, we present the predicates of MTCTL statements in Section 4.3.5 and the operations that MTCTL supports in Section 4.3.6.

As stated in Section 2.1.2, a UTCTL property may only define one temporal quantifier. However, realistic properties exist, where more than one temporal quantifier is defined (e.g., TCTL enables to define more than one temporal quantifier). In contrast to UTCTL, MTCTL supports to concatenate several temporal properties via the binary operators **implies**, **and**, and

or as well as the unary operator **not**. Therefore, a software engineer still cannot nest a temporal quantifier within another temporal quantifier property but can state properties in MTCTL that are not expressible in UTCTL. Noteworthy, we are still able to verify such properties via UPPAAL as we split them into a set of temporal quantifier properties (see Section 5.4.4) that we verify in sequence and execute a succeeding static analysis afterwards. We specify the combination of temporal properties in detail in Section 4.3.7.

MTCTL enables to express several temporal properties that are connected via binary operators into a compact form by providing the quantifiers (**forall** and **exists**) for eight pre-defined sets (e.g., an integer interval or the set of all states). We call these quantifiers set quantifiers. Therefore, the software engineer can specify a property **forall**(s:States) **EF** stateActive(s); that defines that all states may be reached instead of specifying a temporal property for each specific state and concatenating them via **and**. We explain the combination of temporal properties using set quantifiers in detail in Section 4.3.7.2.

MTCTL also supports that set quantifiers may be used within a statement of a temporal property as this enables to express a temporal property in a compact manner. UTCTL also supports to use set quantifiers within a temporal property. However, only integer sets are supported but no other sets like the aforementioned set of all states. We describe this concept in detail in Section 4.3.7.2 as well.

An MTCTL property, a statement, and a set property may also define the values true or false. For example, the following properties are valid: **true**;, **EF true**;, and **forall**(s : States) **true**;. We require this for our static analysis of MTCTL properties (see Section 5.4.3 and 5.6.1).

Concerning the definition of statements, MTCTL consciously does not cover the full expressiveness of UTCTL. In particular, MTCTL does not provide some of UTCTL's operators as we did not find relevant use cases that require them yet. These operators are: increment, decrement, integer addition, integer subtraction, integer multiplication, integer division, modulo, left bitshift, right bitshift, minimum, maximum, bitwise and, bitwise xor, bitwise or, logical and, logical or, and if-then-else.

4.3.2 Basic Semantics

MTCTL's semantics definition depends on the semantics definitions of MECHATRONICUML. MECHATRONICUML is based on a timed transition system, which "[...] incorporate[s] time into an interleaving model of concurrency" [HMP92]. Therefore, "[t]he runtime state space of MECHATRONICUML can be described as an (infinite) tree. Each tree node references a system state (called *snapshot* here to avoid confusion with states in RTSCs). Each successor to a snapshot induces a child to a tree node. Notice that due to this definition, a single snapshot may be referenced in multiple tree nodes." [BCD+14] Moreover, the transitions between the snapshots may be enriched "[...] by quantitative lower-bound and upper-bound timing constraints" [HMP92].

"The semantics of MTCTL is defined by interpretations $I : \text{MTCTL} \rightarrow \{true, false\}$ that map a valid MTCTL property to a truth value. Every MTCTL property is defined in the context of [an RTCP]." [BCD+14] The interpretation of an MTCTL property typically depends on the RTCP, i.e., the same MTCTL property may return different results for different RTCPs. Furthermore, we define that an MTCTL property must be side effect free, i.e., it may only read but not modify the zone graph.

In order to interpret an MTCTL property, we often have to interpret the snapshots of the zone graph. Therefore, we denote I_s as the interpretation of a snapshot s of the zone graph. Let Φ and Ψ be statements, which are part of a temporal property (see Section 4.3.1). The interpretation of a statement may result to *true* and *false*. The interpretation I_s of MTCTL statements in snapshot s is defined recursively as follows:

- We interpret two statements that are connected via the operator **and** as true if and only if the interpretation of both statements is true: $I_s(\Phi \text{ and } \Psi) = \text{true}$ if and only if $I_s(\Phi) = \text{true}$ and $I_s(\Psi) = \text{true}$
- We interpret two statements that are connected via the operator **or** as true if and only if the interpretation of at least one statement is true: $I_s(\Phi \text{ or } \Psi) = \text{true}$ if and only if $I_s(\Phi) = \text{true}$ or $I_s(\Psi) = \text{true}$
- We interpret a statement that has the operator **not** assigned as true if and only if the interpretation of the statements is not true: $I_s(\text{not } \Phi) = \text{true}$ if and only if $(\text{not } I_s(\Phi)) = \text{true}$
- We interpret two statements Φ and Ψ that are connected via the operator **implies** as true if and only if the interpretation of Φ implies the interpretation of Ψ : $I_s(\Phi \text{ implies } \Psi) = \text{true}$ if and only if $(I_s(\Phi) \text{ implies } I_s(\Psi)) = \text{true}$
- The interpretations for statements that use comparisons are defined in the usual way.

For the upcoming formal definition, we denote P as the set of paths through the zone graph that start in the root of the zone graph. $p \in P$ is a path that is contained in P . p consists of a set of snapshots. $s \in p$ is a snapshot of p .

4.3.3 Temporal Quantifiers

MTCTL supports the same temporal quantifiers with the same semantics like UTCTL in order to be compliant with the UPPAAL model checker. In the following, we list these temporal quantifiers, define their interpretation, and give an example for each temporal quantifier:

- AG** We interpret the temporal quantifier always globally (**AG**) for a statement Φ as true if and only if Φ is true in all snapshots in the zone graph: $I(\text{AG } \Phi) = \text{true}$ if and only if $\forall p \in P, \forall s \in p: I_s(\Phi) = \text{true}$. For example, if the MTCTL property **AG** $i==5$; holds, then it invariantly holds that the variable i has the value 5.
- AF** We interpret the temporal quantifier always future (**AF**) of a statement Φ as true if and only if Φ is true in at least one snapshot on each path in the zone graph: $I(\text{AF } \Phi) = \text{true}$ if and only if $\forall p \in P, \exists s \in p: I_s(\Phi) = \text{true}$. For example, if the MTCTL property **AF** $i==5$; holds, then it eventually holds on all paths that the variable i has the value 5.
- EG** We interpret the temporal quantifier exists globally (**EG**) of a statement Φ as true if and only if Φ is true in each snapshot of at least one path in the zone graph: $I(\text{EG } \Phi) = \text{true}$ if and only if $\exists p \in P, \forall s \in p: I_s(\Phi) = \text{true}$. For example, if the MTCTL property **EG** $i==5$; holds, then it might always hold that the variable i has the value 5.
- EF** We interpret the temporal quantifier exists future (**EF**) of a statement Φ as true if and only if Φ is true in at least one snapshot in the zone graph: $I(\text{EF } \Phi) = \text{true}$ if and only if $\exists p \in P, \exists s \in p: I_s(\Phi) = \text{true}$. For example, if the MTCTL property **EF** $i==5$; holds, then it can possibly happen that the variable i has the value 5.

leadsTo We interpret the temporal quantifier **leadsTo** as follows: if and only if we define that a statement Φ leads to the statement Ψ , then this is semantically equivalent to “whenever Φ is true, then eventually Ψ will be fulfilled as well”. Formally, we denote P_s as the set of paths through the zone graph that start in the snapshot s . $I_{s'}$ is the interpretation of snapshot s' . $I(\Phi \text{ leadsTo } \Psi) = \text{true}$ if and only if $\forall p \in P, \forall s \in p, \forall p' \in P_s, \exists s' \in p': (I_s(\Phi) = \text{true}) \text{ implies } (I_{s'}(\Psi) = \text{true})$. For example, if the MTCTL property $i==5 \text{ leadsTo } j==3$; holds, then whenever variable i has the value 5, then eventually the variable j will have the value three as well.

4.3.4 Referencing Elements

MTCTL is able to reference all relevant model elements of an RTCP. In particular, within an MTCTL property, we do not only state the fully qualified name in the scope of the RTCP but also store the reference to the respective design model element. As a consequence, if the design model element changes (i.e., it is renamed or moved to another hierarchy level), then we can automatically update the fully qualified name in the property.

In the following, we list the model elements of an RTCP that we may reference attached with a concrete example from the RTCPs Overtaking and MultiOvertaking.

State We enable to reference a state of a role’s RTSC. For example, in RTCP Overtaking, the expression `Overtaker.Overtaking.Init` references the substate `Init` of the hierarchical state `Overtaking` that is contained in the role-RTSC `Overtaker`.

Statechart We enable to reference a statechart of a role’s RTSC. The statechart may be the root statechart of the role’s RTSC or an embedded RTSC. For example, in RTCP Overtaking, the expression `Overtaker.Overtaking.overtaking_rtsc` references the statechart `overtaking_rtsc` of the hierarchical state `Overtaking` that is contained in the role-RTSC `Overtaker`.

Clock We enable to reference a clock of a role’s RTSC. For example, in RTCP Overtaking, the expression `Overtakee.c` references the clock `c`, which is contained in the RTSC of role `Overtakee`.

Integer-Based Variable We enable to reference an integer-based variable (e.g., `uint8`, `int16`, `boolean`) defined within a role’s RTSC. For example, in RTCP Overtaking, the expression `Overtakee.speed` references the variable `speed`, which is contained in the RTSC of role `Overtakee`.

Transition We enable to reference a transition of a role’s RTSC. For example, in RTCP Overtaking, the expression `Overtaker.NoOvertaking.Init_to_Requested` references the transition that has its source in state `Overtaker.NoOvertaking.Init` and its target in state `Overtaker.NoOvertaking.Requested`.

Message Type We enable to reference a message type that is defined as a sender or receiver message in a role. For example, in RTCP Overtaking, the expression `OvertakingMsgs.accept` references the message type `accept`, which is contained in the message repository `OvertakingMsgs`.

Incoming Message Buffer We enable to reference an incoming message buffer of a role. For example, in RTCP Overtaking, the expression `Overtaker.b1` references the buffer `b1`, which is contained in the role `Overtaker`.

Role Instance We enable to reference an instance of a role. For example, in RTCP MultiOvertaking, the expression `Overtakee.NoAcceleration[0]` references the state `NoAcceleration`, which is contained in the RTSC of the single role instance with ID 0 of role `Overtakee`. If we reference a role that may only be instantiated once, then the ID of the instance may be omitted. For example, in RTCP MultiOvertaking, the expression `Overtaker.Overtaking` references the state `Overtaking` of the instantiated multi role `Overtaker` (see Figure 3.9 on page 64).

Subrole Instance We enable to reference a subrole instance of a multi role instance. For example, in RTCP MultiOvertaking_wSubroleRegion (a normalization of RTCP MultiOvertaking where the multi role contains a specific subrole region), the expression `Overtaker.Main.subrole.Subrole_Main.subrole_multicast1.Sent[0]` references the state `Sent`. This state is contained in the RTSC of the subrole instance with ID 0 of the multi role `Overtaker` (we depict its multi role RTSC in Figure 3.17 on page 77).

Formally, a reference x may be static or dynamic. Its value $V_s(x)$ for the snapshot s of the zone graph is as follows:

- If x is a message type, then we reference the static value of this element. In this case, x is a static reference as it is independent of the snapshot s .
- If x is a state, a statechart, a transition, an incoming message buffer, a role instance, or a subrole instance, then we reference the instantiated element. In this case, x is a static reference as it is independent of the snapshot s because instances of these elements may not change at runtime.
- If x is a clock or an integer-based variable, then we reference the value of the instance of this element for a given snapshot. In this case, x is a dynamic reference as it depends on the snapshot s because the value of these elements may change at runtime.

In the following, we justify our list of supported model elements: We support referencing states, clocks, and variables because UTCTL supports referencing exactly these three types as well. Therefore, MTCTL will not lose expressiveness compared to UTCTL. Concerning the variables, MTCTL currently supports all integer-based primitive types (e.g., `int8`) and the primitive type `boolean` but not the type `double` as UPPAAL does not support it either.

We support to reference transitions in order to evaluate whether a transition may fire. For such an evaluation, it is not sufficient to verify whether the target state of the transition is reachable as this state may have more than one incoming transition. Therefore, we have to explicitly reference the transition.

The concept of asynchronous parametrized messages is a domain-specific concept of MECHATRONICUML that is not available in UPPAAL and, therefore, is not supported in UTCTL. However, we can express it with other model elements in UTCTL. The MTCTL specifier requires it within MTCTL to make statements about the state of a certain message, e.g., whether the message is in transit.

Incoming message buffers are a domain-specific concept of MECHATRONICUML as well that is not available in UPPAAL and, therefore, is not supported in UTCTL. Again, we can

express it with other model elements in UTCTL. The MTCTL specifier requires it within MTCTL to make statements about the state of a message buffer, e.g., whether a certain buffer has to discard a message at some time.

As stated in Section 3.3.5, an RTCP may be instantiated. In particular, a one-to-one RTCP defines that each single role is instantiated once. In contrary, a one-to-many RTCP, which consists of one single and one multi role, defines that the single role may be instantiated multiple times (depending on the role multiplicity) and that the multi role is only instantiated once. For example, the one-to-many RTCP MultiOvertaking (see Figure 3.9) consists of the multi role Overtaker and the single role Overtakee. Therefore, if we instantiate the RTCP, then the Overtaker is instantiated once and the Overtakee may be instantiated one to two times. At runtime, instances of the same role may differ (e.g., each instance may have a different active state). Therefore, the MTCTL specifier needs to be able to reference certain role instances within MTCTL. An exemplary requirement for the RTCP MultiOvertaking is as follows: “It invariantly holds that if the instance of multi role Overtaker is in state Overtaking, then all instances of role Overtakee are in state NoAcceleration”. We can translate it into the following MTCTL property: **AG forall**(i:Instances<Overtaker>) **forall**(j:Instances<Overtakee>) (**stateActive**(Overtaker.Overtaking[i]) **implies stateActive**(Overtakee.NoAcceleration[j]));

A multi role instance may have multiple subrole instances (see Section 3.3.5). At runtime, subrole instances of the same multi role instance may differ (e.g., each subrole instance may have a different active state). Therefore, the MTCTL specifier needs to be able to reference certain subrole instances within MTCTL. An exemplary requirement for the RTCP MultiOvertaking_wSubroleRegion (a normalization of RTCP MultiOvertaking where the multi role contains a specific subrole region) is as follows: “One of the subrole instances of role Overtaker may reach the state Sent in region subrole_multicast1”. We can translate it into the following MTCTL property: **exists**(i:Subinstances<Overtaker>) (**EF stateActive**(Overtaker.Main.subrole.Subrole_Main.subrole_multicast1.Sent[i]));. Noteworthy, we currently do not enable that a specific subrole (e.g., the second subrole) may be referenced as we had no use case until now, where we would like to specify such a property. Therefore, a property like **EF stateActive**(Overtaker.Main.subrole.Subrole_Main.subrole_multicast1.Sent[0]) is not supported. It is only possible to make statements about all subrole instances (using the **forall** and the **exists** set quantifier; see Section 4.3.7.2).

4.3.5 Predicates

MTCTL provides a set of predicates that enable the MTCTL specifier to analyze snapshots of an RTCP concerning concepts that are—in most cases—specific to RTCPs, e.g., whether a message is in a buffer.

In particular, we distinguish dynamic and static predicates. The truth value of a dynamic predicate depends on a specific MECHATRONICUML snapshot. Contrary, the truth value of a static predicate is independent of a specific snapshot and may depend on a given RTCP of MECHATRONICUML. Furthermore, MTCTL also supports predicates that enable comparison expressions. In particular, we support six kinds of comparisons: equals, greater, greater or equal, less, less or equal, and not equal. Depending on the expressions that are compared, these predicates may be static or dynamic.

MTCTL defines predicates with zero to two arguments. Arguments may be specific values or variables respectively operations as a placeholder for a specific value.

In the following, we first introduce MTCTL's dynamic predicates in Section 4.3.5.1 before we introduce MTCTL's static predicates in Section 4.3.5.2.

4.3.5.1 Dynamic Predicates

In the following, we list the dynamic predicates that MTCTL supports (including their arguments) and define their interpretation (see [BCD+14]).

deadlock This predicate is true if and only if this snapshot has only outgoing snapshot transitions of kind delay (see Section 2.1.1). Formally, we define $I_s(\text{deadlock}) = \text{true}$ if and only if all snapshot transitions of all following snapshots of s are delay transitions.

connectorOverflow This predicate is true if and only if this snapshot or any snapshot before this snapshot should have added a message instance to a connector message buffer (see Section 5.4.13.3) but the buffer was already at full capacity. Formally, we define $I_s(\text{connectorOverflow}) = \text{true}$, if and only if the snapshot s' or the snapshot s ($\forall p \in P, \exists s, s' \in p, s'$ is any predecessor of s) should have added a message instance to the connector message buffer but the buffer was already at full capacity.

stateActive(state) This predicate is true if and only if the referenced state is currently active within this snapshot. In particular, a state is also active while any of its connection points are traversed during firing a transition. Formally, we define " $I_s(\text{stateActive}(\text{state})) = \text{true}$ if and only if [the state] $V_s(\text{state})$ is active in s " [BCD+14].

transitionFiring(transition) This predicate is true if and only if the referenced transition is currently firing within this snapshot. Noteworthy, this predicate is not only true for time-consuming transitions (i.e., transitions that define a deadline, see Section 2.2.2) but also for non-time-consuming transitions. Formally, we define " $I_s(\text{transitionFiring}(\text{transition})) = \text{true}$ if and only if [the transition] $V_s(\text{transition})$ is currently firing in s ." [BCD+14]

messageDiscarded(buffer) This predicate is true if and only if the referenced buffer has in this snapshot or any snapshot before discarded a message. Therefore, as soon as this predicate is true, it remains true in all upcoming snapshots. This definition is sufficient, as we are only interested whether the buffer ever discards a message and not how often the buffer discards a message. Formally, we define $I_s(\text{messageDiscarded}(\text{buffer})) = \text{true}$, if and only if an incoming buffer $V_s(\text{buffer})$ should have added a message in s or any snapshot before s but the buffer was already full.

messageInBuffer(messageType, buffer) This predicate is true if and only if an instance of the referenced message type is within this snapshot in an instance of the referenced incoming message buffer. Although an instance of a message type may only be stored in one of the role's buffers, the second parameter for specifying the buffer is still necessary, as both roles could be able to store the same message type. Formally, we define $I_s(\text{messageInBuffer}(\text{messageType}, \text{buffer})) = \text{true}$, if and only if in s , a message of type $V_s(\text{messageType})$ is currently stored in the incoming buffer $V_s(\text{buffer})$.

messageInTransit(messageType) This predicate is true if and only if an instance of the referenced message type is in this snapshot in transit (i.e., it was already sent but not yet enqueued within the incoming message buffer). If the connector is unreliable and, thus, may lose messages, the predicate is only true if the message is not lost yet. Noteworthy, the message may still be lost in a later snapshot. Formally, we define “ $I_s(\text{messageInTransit}(\text{messageType})) = \text{true}$, if and only if in s , a message of type $V_s(\text{messageType})$ is currently being transmitted to the [receiver], but has not arrived yet.” [BCD+14]

Our justification why we support these predicates is as follows. At first, the predicate **deadlock** is taken over from UPPAAL to search for deadlocks within an RTCP. We also support this predicate as well because an RTCP may also have deadlocks, which may end the protocol in an unsafe state. Therefore, a software engineer typically wants to check whether no deadlock exists.

In contrary, **connectorOverflow** is a predicate that is specific for MECHATRONICUML. For our model checking, we use a so-called connector buffer (see Section 5.4.13.3) that stores the message in transit. When verifying an RTCP, we assume that the middleware’s memory for storing messages as well as the physical connector are no limiting factor concerning the maximum messages in transit if the connector is reliable. Using this predicate, the software engineer can verify whether the internal connector buffer may overflow or not, and, therefore, whether the assumption is fulfilled. Without this predicate, the MTCTL specifier could not state such a property as we abstracted the behavior specification of message transmissions (this behavior is not modeled as an RTSC anymore).

The **stateActive** predicate indicates whether a certain state is active. In UPPAAL’s syntax, the software engineer only has to state the name of the state to express that it shall be active. In contrary, the MTCTL specifier has to use the **stateActive** predicate explicitly. We require this only for consistency reasons.

The predicate **transitionFiring** is specific for MECHATRONICUML. As a state of an RTSC may have several outgoing transitions, it is of relevant information which of the transition may fire. There exists no direct counterpart in UTCTL but only workarounds. For example, if the software engineer wants to verify whether a specific transition may fire, he has to add an intermediate state and verify whether this state is reachable.

messageDiscarded is a predicate that is specific for MECHATRONICUML as well. A message is discarded from a buffer if the buffer is already full but a new message arrives. In Section 3.3.4, we defined the buffer overflow avoidance strategy “Never Happens”, which states that it may never be the case that a message needs to be discarded. Using this predicate, the software engineer can verify whether this situation may happen. Without this predicate, the MTCTL specifier could not express such a property as we abstracted the overflow avoidance behavior such that it is not modeled as an RTSC anymore (see Chapter 3).

Using the predicates **messageInBuffer** and **messageInTransit**, the MTCTL specifier can define properties concerning the current whereabouts of a certain message. Again, these are predicates that are specific for MECHATRONICUML. We need them because we abstracted the behavior specification of message buffers and messages in transit such that they are not modeled as an RTSC anymore (see Chapter 3). Therefore, without these predicates, the MTCTL specifier could not express such properties. As a side note, we ensure via a static

semantics constraint that the MTCTL specifier may not select an arbitrary buffer but only buffers that may store the referenced message type.

4.3.5.2 Static Predicates

The supported static predicates and their interpretations are:

true This predicate is always true. Formally, we define $I_s(\text{true}) = \text{true}$.

false This predicate is always false. Formally, we define $I_s(\text{false}) = \text{false}$.

substateOf(state, superstate) This predicate is true if and only if the referenced state is a substate of the referenced superstate. Moreover, this predicate is non-reflexive, i.e., a state cannot be its own superstate. Therefore, if state and superstate reference the same state, then this predicate is false. Formally, we define “ $I_s(\text{substateOf}(\text{state}, \text{superstate})) = \text{true}$ if and only if $V_s(\text{state})$ is a (direct or indirect) substate of $V_s(\text{superstate})$ ” [BCD+14].

stateInStatechart(state, statechart) This predicate is true if and only if the referenced state is contained in the referenced statechart. Formally, we define “ $I_s(\text{stateInStatechart}(\text{state}, \text{statechart})) = \text{true}$ if and only if $V_s(\text{state})$ is (directly or indirectly) embedded in $V_s(\text{statechart})$ ” [BCD+14].

The predicates **true** and **false** have no arguments. Moreover, their interpretation is independent of the RTCP as they always result to true resp. false. We need these predicates during the DSMC when we statically evaluate the MTCTL properties (see Section 5.4.3 and 5.6.1).

The truth of the predicates **substateOf** and **stateInStatechart** depends on the RTCP but not on a certain snapshot. Typically, an MTCTL specifier uses them in combination with set quantifiers (see Section 4.3.7.2). For example, the MTCTL specifier can define that all substates of state **NoOvertaking**, which is defined in role **Overtaker**, are reachable: **forall**(s:States) (**substateOf**(s,Overtaker.NoOvertaking) **implies** **EF stateActive**(s));. As another example, the MTCTL specifier can define that there exists a state that is contained in the embedded statechart **noAccelerationRTSC** (this statechart is embedded in the state **Overtakee.NoAcceleration**) of role **Overtakee** and is reachable: **exists**(s:States) (**stateInStatechart**(s,Overtakee.noAccelerationRTSC) **and** **EF stateActive**(s));.

4.3.6 Operations

In MTCTL, an operation may be used within a statement, has one or more parameters, and a defined return type. Both, the parameter types as well as the return type, may be one of the referenceable model elements or an integer. In particular, we currently support the following three operations:

int bufferMessageCount(buffer) This operation returns an integer value that represents the number of messages within the given buffer in the current snapshot. Therefore, this operation depends on the snapshot under analysis. For example, **AG bufferMessageCount**(Overtakee.b2) < 5; defines that the buffer **b2** of role **Overtakee**

will never contain five or more messages. Using such a property, the MTCTL specifier can identify the maximum message buffer size. Noteworthy, it is also possible to compare the size of two different buffers. For example, `EF bufferMessageCount(Overtakee.b1) < bufferMessageCount(Overtakee.b2)`; defines that it may be the case that the size of the buffer b1 is smaller than the size of the buffer b2.

state sourceState(transition) This operation returns a source state of a given transition and does not depend on a particular snapshot but only on the RTCP under analysis. For example, `EF stateActive(sourceState(Overtaker.NoOvertaking.Idle_to_WaitingForAnswer))`; defines that the source state of the transition from state NoOvertaking.Idle to WaitingForAnswer of role Overtaker may be active.

state targetState(transition) This operation returns a target state of a given transition and does not depend on a particular snapshot but only on the RTCP under analysis. For example, `EF stateActive(targetState(Overtaker.NoOvertaking.Idle_to_WaitingForAnswer))`; defines that the target state of the transition from state NoOvertaking.Idle to WaitingForAnswer of role Overtaker may be active.

4.3.7 Combining Temporal Properties

UTCTL as well as the UPPAAL model checker have several limitations. Among others, an UTCTL property may only use at most one temporal quantifier. However, there exist requirements for RTCPs that may only be expressed using several temporal quantifiers. For example, a valid requirement is as follows: “Each state of the RTCP shall be reachable”. On the level of UPPAAL, the software engineer would have to define for each state one UTCTL property in the form `EF s`, where `s` is the name of the state. Then, the software engineer would have to verify each of these properties, and then check manually whether all UTCTL properties are fulfilled. As a negative side-effect, the information that all these separated properties must be fulfilled in order to fulfill the requirement is not part of the model under development anymore. This inconsistency can lead to the problem that a requirement is not correctly validated.

MTCTL also enforces via its grammar that each temporal property has only one temporal quantifier in order to be compliant with UPPAAL. However, it enables to combine a set of temporal properties using logical operators and so-called set quantifiers. Therefore, the requirement given above may be expressed using the single MTCTL property `forall(s:States) EF stateActive(s)`. This results in compact properties that should be easier to understand and that do not miss important parts of the requirement. Noteworthy, we obviously cannot verify such a property using UPPAAL directly. Instead, we extract all temporal properties of this property and verify them with UPPAAL one after the other. Therefore, UPPAAL returns one result (true or false) per temporal property. If we replace the results within the original MTCTL property, then we can simplify this using static analysis to a single truth value (true or false) as the original MTCTL property will only consist of `true`, `false`, and logical operators. We describe these steps in detail in Chapter 5.

In the following, we introduce the logical operators to combine temporal properties first. Then, we present the set quantifiers that MTCTL supports.

4.3.7.1 Logical Operators

MTCTL enables to combine properties (e.g., temporal properties) using logical operators to form a propositional calculus. Therefore, each property represents one proposition that may be true or false. We support four logical operators: **and**, **or**, **not**, and **implies**. For example, the MTCTL specifier can specify with one expression that three states s_1 , s_2 , and s_3 can possible become active via **(EF stateActive(s_1)) and (EF stateActive(s_2)) and (EF stateActive(s_3))**;). The order of the logical operator precedence is as follows: **not** has the highest precedence, **and** has medium precedence, **or** and **implies** both have the lowest precedence.

Formally, the interpretation I of an MTCTL property that combines multiple (temporal) properties is defined recursively as follows (Θ and Υ are properties of MTCTL):

- $I(\Theta \text{ and } \Upsilon) = \text{true}$ if and only if $I(\Theta) = \text{true}$ and $I(\Upsilon) = \text{true}$
- $I(\Theta \text{ or } \Upsilon) = \text{true}$ if and only if $I(\Theta) = \text{true}$ or $I(\Upsilon) = \text{true}$
- $I(\text{not } \Theta) = \text{true}$ if and only if $(\text{not } I(\Theta)) = \text{true}$
- $I(\Theta \text{ implies } \Upsilon) = \text{true}$ if and only if $(I(\Theta) \text{ implies } I(\Upsilon)) = \text{true}$

4.3.7.2 Set Quantifiers

MTCTL defines quantifiers for various sets of MECHATRONICUML model elements, which we call set quantifiers. In particular, we support the universal set quantifier (**forall**) and the existential set quantifier (**exists**). The MTCTL specifier can use these quantifiers to parametrize a property and, thus, combine several properties into one. For example, he can specify with one expression that all states can possible become active via **forall (s:States) EF stateActive(s)**;). If the RTCP consists of the three states s_1 , s_2 , and s_3 , then the property **(EF stateActive(s_1)) and (EF stateActive(s_2)) and (EF stateActive(s_3))**; is semantically equivalent but not as compact as the property stated before.

Formally, the interpretation I of an MTCTL property that combines (temporal) properties via set quantifiers is defined as follows (Θ is an MTCTL property, SET is one of our supported quantifier sets):

- $I(\text{forall}(x:\text{SET}) \Theta) = \text{true}$ if and only if $\forall \bar{x} \in \text{SET}: I(\Theta_{[x/\bar{x}]}) = \text{true}$
- $I(\text{exists}(x:\text{SET}) \Theta) = \text{true}$ if and only if $\exists \bar{x} \in \text{SET}: I(\Theta_{[x/\bar{x}]}) = \text{true}$

Both set quantifiers may be used for the following sets:

Integer intervals The set of integers in the specified bounds, e.g., $[0,10]$ defines the interval 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

States The set of instantiated states of all RTSCs of the RTCP. For example, **forall(s:States) EF stateActive(s)**; defines that all states of the RTCP shall be reachable.

Transitions The set of instantiated transitions of all RTSCs of the RTCP. For example, **forall(t:Transitions) EF transitionFiring(t)**; defines that all transitions of the RTCP shall be able to fire.

Clocks The set of instantiated clocks of all RTSCs of the RTCP. For example, **exists**(**c:Clocks**) **EF** **c**>100msecs; defines that a clock exists that will be greater than 100 milliseconds.

MessageTypes The set of all message types that may be sent or received via the RTCP. For example, **forall**(**m : MessageTypes**) **EF** **messageInTransit**(**m**); defines that at least one instance per message type of the RTCP can possibly be in transit.

Buffers The set of all instantiated incoming message buffers of the RTCP. For example, **forall**(**b : Buffers**) **EF** **bufferMessageCount**(**b**) >= 1; defines that there is no buffer that is always empty.

Instances<Element> The set of all instances of a role, port, or component of type 'Element'. This set is only needed if the RTCP under analysis is of kind one-to-many. Otherwise, both roles only have one instance. For example, our one-to-many RTCP MultiOvertaking consists of one instance of role Overtaker and up to two instances of role Overtakee. Therefore, the property **forall**(**i:Instances<Overtakee>**) **EF** **stateActive**(**Overtakee.NoAcceleration[i]**); defines that all instances of role Overtakee shall be able to reach the state NoAcceleration (not necessarily in the same snapshot).

Subinstances<Element> The set of all subrole instances (short: subinstances) of a multi role of type 'Element'. Likewise to the set **Instances**, this set is only useful for one-to-many RTCPs. In addition, this set may only be used if the RTCP does not use one-to-many communication schemata but distinguishes in the multi role between a coordinator and a subrole. For example, consider the RTCP MultiOvertaking_wSubroleRegion that does not use one-to-many communication schemata. Here, the role Overtaker is a multi role and may contain up to two subroles. Therefore, the property **forall**(**i:Subinstances<Overtaker>**) **EF** **stateActive**(**Overtaker.Main.subrole.Overtaking[i]**); defines that all subinstances of role Overtaker shall be able to reach the state Overtaking (not necessarily in the same snapshot).

In the examples given above, the set quantifiers are defined outside of the temporal property. Therefore, a normalized form that does not use the set quantifier would duplicate the temporal property, replace the parameters of the set quantifier with the concrete values, and concatenate the temporal properties using the **and** resp. the **or** operator. For example, **forall**(**m : MessageTypes**) **EF** **messageInTransit**(**m**); defines that each message types may be in transit. If the RTCP defines the messages **m1** and **m2**, then the normalized form is (**EF** **messageInTransit**(**m1**)) **and** (**EF** **messageInTransit**(**m2**));.

In addition, MTCTL enables to use set quantifiers to combine statements within a temporal property. A normalized form that does not use the set quantifier would duplicate the statement that is within the scope of the set quantifier, replace the parameters of the set quantifier with concrete values, and concatenate the statements using the **and** resp. the **or** operator. For example, **EF forall**(**m : MessageTypes**) **messageInTransit** (**m**); defines that it is possible that all message types are in transit at the same time. If the RTCP defines the messages **m1** and **m2**, then the normalized form is **EF** (**messageInTransit**(**m1**) **and** **messageInTransit**(**m2**));.

Formally, the interpretation I_s of a set quantifier in combination with a statement is defined as follows (Φ is a statement, s is a snapshot):

- $I_s(\text{forall}(x:\text{SET}) \Phi) = \text{true}$ if and only if $\forall \bar{x} \in \text{SET}: I_s(\Phi_{[x/\bar{x}]}) = \text{true}$
- $I_s(\text{exists}(x:\text{SET}) \Phi) = \text{true}$ if and only if $\exists \bar{x} \in \text{SET}: I_s(\Phi_{[x/\bar{x}]}) = \text{true}$

Moreover, MTCTL enables to use more than one set quantifier for an expression. For example, **forall**($m : \text{MessageTypes}$) **exists**($b : \text{Buffers}$) **EF** **messageInBuffer**(m, b); defines that at least one instance per message type of the RTCP can possibly be buffered by the receiver (not necessarily in the same snapshot). Noteworthy, the order of the set quantifiers is semantically relevant—outside and inside of the temporal quantifier expressions. For example, if an RTCP defines the message types $m1$ and $m2$ and the buffers $b1$ and $b2$, then the MTCTL property **forall**($m : \text{MessageTypes}$) **exists**($b : \text{Buffers}$) **EF** **messageInBuffer**(m, b); is semantically equivalent to the property (**EF** (**messageInBuffer**($m1, b1$)) **or** **EF** (**messageInBuffer**($m1, b2$))) **and** (**EF** (**messageInBuffer**($m2, b1$)) **or** **EF** (**messageInBuffer**($m2, b2$))). However, for the same RTCP, the property **exists**($b : \text{Buffers}$) **forall**($m : \text{MessageTypes}$) **EF** **messageInBuffer**(m, b); is semantically equivalent to the property (**EF** (**messageInBuffer**($m1, b1$)) **and** **EF** (**messageInBuffer**($m2, b1$))) **or** (**EF** (**messageInBuffer**($m1, b2$)) **and** **EF** (**messageInBuffer**($m2, b2$))). Therefore, this second property has the following informal meaning: “There exists at least one buffer that can possible store each instance per message type of the RTCP (not necessarily in the same snapshot)”.

Noteworthy, all our sets that we support know their elements already at compile time. In addition, the number of elements that a set may refer to is finite. Consequently, a set is finite and may not change its elements at runtime. We use this fact when transforming an MTCTL property to UTCTL (see Section 5.4.2) as UTCTL only supports set quantifiers for integers and scalars but not for other model elements, e.g., states.

As a positive side-effect, MTCTL properties can become independent of the RTCP under analysis when using set quantifiers. For example, the expressions stated above that use the set quantifiers do not need to reference particular elements of the RTCP. This increases the reusability of verification properties. We discuss this topic in detail within the following section.

4.4 Default Verification Properties for RTCPs

The choice of verification properties for an RTCP is highly important. Otherwise, existent faults may remain undetected and may lead to safety-critical situations in the final system, which can—in the worst case—endanger human life. Existent design faults of the RTCP may remain undetected as well, which may lower quality aspects like understandability and maintainability. Therefore, we aim to support the MTCTL specifier for defining relevant properties concerning the runtime behavior of the RTCP.

Typically, the MTCTL specifier has to define verification properties that are specific for the RTCP under development and that depend on the (informal) requirements. For example, we require for the RTCP Overtaking that the overtakee does not accelerate during the overtaking. Therefore, the MTCTL specifier must be able to manually translate these RTCP-specific requirements into an MTCTL property. This is a task that needs experience and where we currently do not support the MTCTL specifier with appropriate properties.

However, we identified requirements that shall be assigned to each RTCP by default. By having these default requirements, we support to improve the safety (“something must

not happen”), the reachability (“something may happen”), and the design of the RTCP. In addition, we facilitate the task of the requirements engineer as he does not need to decide whether they are relevant for this RTCP. Moreover, we avoid the hazard that one of these requirements may be forgotten. One example for such a requirement is that it is never desired that a deadlock may occur in an RTCP. Another example is that all states of an RTCP shall be reachable (**forall**(s: States) **EF** stateActive(s);). The first example improves the safety while the second one improves the reachability (at least to some extent as the states only need to be reachable once) and the design (as it avoids unreachable states in the RTCP). Especially the validation concerning reachability is highly important. For example, as stated above, the RTCP Overtaking has to ensure that during the overtaking the overtaken does not accelerate. Therefore, the MTCTL specifier defines the safety property **AG** (stateActive(Overtaker.Overtaking) **implies** stateActive(Overtakee.noAcceleration));. This is a safety property that defines that something bad may not happen. However, this property is already fulfilled if the state Overtaker.Overtaking is not reachable, which is not intended for this RTCP. If the requirements engineer forgets to define a property that all states shall be reachable (or at least that the state Overtaking of role Overtaker shall be reachable), then something bad may not happen but something good neither. Due to our default requirement that all states should be reachable, we ensure that something good may happen (if the reachability requirement is fulfilled) and, therefore, provide a foundation for a meaningful verification of safety properties.

In total, we collected seven dynamic semantics requirements that all one-to-one have to fulfill. Moreover, five of these requirements have to be fulfilled by all one-to-many RTCPs and two of them have to be fulfilled in most cases (in some special cases it may be valid that they are only fulfilled to a great extent—we explain this later in detail). For each of these requirements, we provide one MTCTL property. We call these *default properties*.

Our default properties have two important characteristics: First, they do not need to reference specific model elements of an RTCP. Thus, the specification of the properties is independent of the concrete RTCP. In fact, it also does not matter if it is a one-to-one or a one-to-many RTCP. Second, the role multiplicity of a one-to-many RTCP is also not relevant for all our default properties. These characteristics are for five of the requirements possible due to the set quantifiers that we introduced in Section 4.3.7.2. This fact leads to two advantages for the MTCTL specifier: First, the MTCTL specifier does not need to specify these requirements for each RTCP. Second, if the RTCP changes, the default properties do not need to be changed as well. For example, the MTCTL property for our second requirement that we stated above is **forall**(s:States) **EF** stateActive(s);. Thus, if a new state is added to the RTCP, then the MTCTL property will cover this one as well.

In the following, we list our seven requirements, give the default property for each requirement, justify why each requirement is useful, state consequences if a default property is fulfilled, and define which actions the RTCP specifier shall execute if a property is not fulfilled.

A deadlock never occurs. The corresponding MTCTL property is **AG** not deadlock;. A deadlock shall never happen as it would stop the coordination suddenly in a possibly undefined state. This is especially not acceptable due to the safety-critical domain of CPSs. If this property is fulfilled, then it is verified that something bad (the deadlock)

does not happen. If the property is not fulfilled, then we consider the RTCP as unsafe. Therefore, the software engineer must adapt his RTCP.

The internal message connector does never overflow. The corresponding MTCTL property is **AG not connectorOverflow**;. For our model checking, we use a so-called connector buffer (see Section 5.4.13.3) that stores the messages in transit. When verifying an RTCP, we assume that the middleware's memory for storing messages as well as the physical connector are no limiting factors concerning the maximum messages in transit if the connector is reliable. This property shows whether our assumption is fulfilled. However, this property is also mandatory even if the connector is unreliable. Otherwise, we would not verify the complete state space of the RTCP as at least the case where all messages arrive at the receiver would not be considered. If this property is fulfilled, then it is verified that something bad (the assumption is not fulfilled) does not happen. If this property is not fulfilled, then we consider the RTCP as unsafe. Therefore, the software engineer must either adapt his RTCP or the verification parameter that defines the size of this connector.

All states of the RTCP (i.e., all states of both roles) shall be reachable. The corresponding MTCTL property is **forall(s: States) EF stateActive(s)**;. If this property is fulfilled, then it is verified that something good (reaching each state at least once) does happen. If this property is not fulfilled and the RTCP is of kind one-to-one, then the software engineer either defined unnecessary states that should be deleted or introduced errors in the RTSCs that prevent that one of the states may be active. In both cases, the software engineer should adjust his RTCP. If this property is not fulfilled and the RTCP is of kind one-to-many, then the RTCP may still be well defined. This is the case if all states are reachable except the few states of the subrole instances that define conditions that may only be fulfilled by specific subrole instance. For example, the RTCP MultiOvertaking_wSubroleRegion (see Figure 3.17 on page 77) defines a transition from Overtaker.subrole.Subrole_Main.subrole_multicast1.Sent to Idle with the guard [self == last]. If this transition would be split into two with an intermediate state, then this state may only be reached by the last subrole instance and not by the other subrole instances. Note that our DSMC approach does not only state whether the complete property is fulfilled but also lists the result for each single state (see Section 5.7.1). Therefore, the software engineer can inspect whether only these special states are not reachable for some of the subrole instances.

All transitions of the RTCP (i.e., all transitions of both roles) shall be able to fire. The corresponding MTCTL property is **forall(t: Transitions) EF transitionFiring(t)**;. If this property is fulfilled, then it is verified that something good (firing each transition at least once) does happen. If this property is not fulfilled and the RTCP is of kind one-to-one, then the software engineer either defined unnecessary transitions that should be deleted or introduced errors in the RTSCs that prevent that one of the transitions may be fired. In both cases, the software engineer should adjust his RTCP. If this property is not fulfilled and the RTCP is of kind one-to-many, then the RTCP may still be well defined. This is the case if all transitions are able to fire except the few transitions of the subrole instances that define conditions that may only be fulfilled by specific subrole instance. For example, in RTCP MultiOvertaking_wSubroleRegion (see Figure+3.17 on page

77), a transition from `Overtaker.subrole.Subrole_Main.subrole_multicast1.Sent` to `Idle` defines the guard `[self == last]`. Therefore, this transition may only be fired by the last subrole instance and not by the other subrole instances. Note that our DSMC approach does not only state whether the complete property is fulfilled but also lists the result for each single transition (see Section 5.7.1). Therefore, the software engineer can inspect whether only these special transitions may not be fired by some of the subrole instances.

At least one instance per message type of the RTCP can be in transit. The corresponding MTCTL property is `forall(m : MessageTypes) EF messageInTransit(m);`. If this property is fulfilled, then it is verified that something good (at least one instance of each message type can be in transit at least once and, therefore, can possibly be sent at least once) does happen. If this property is not fulfilled, then the software engineer either defined unnecessary message types that may be deleted as they are never sent by a transitions or introduced errors in the RTSCs that prevent that the message may be sent. In both cases, the software engineer should adjust his RTCP.

At least one instance per message type of the RTCP can arrive in any of the buffers. The corresponding MTCTL property is `forall(m : MessageTypes) exists(b:Buffers) EF messageInBuffer(m,b);`. Note that MTCTL statically checks for the predicate `messageInBuffer` whether the buffer `b` is able to store messages of type `m`. Therefore, we can conclude the following facts: If this property is fulfilled, then it is verified that something good (at least one instance of each message type can reach its buffer at least once) does happen. If this property is not fulfilled, then the software engineer either defined unnecessary message types that may be deleted, introduced errors in the RTSCs that prevent that the message may be sent, or—if a buffer can possibly store more than one message type—that the specified buffer `b` can only store other message types. In all three cases, the software engineer should adjust his RTCP. Note that this requirement implies that at least one instance per message type can possibly be in transit and, therefore, can possibly be sent.

Each incoming message buffer of the RTCP can contain at least one message. The corresponding MTCTL property is `forall(b : Buffers) EF bufferMessageCount(b) >= 1;`. If this property is fulfilled, then it is verified that something good (each buffer can contain at least one message at least once) does happen. If this property is not fulfilled, then the software engineer either defined unnecessary message buffers that may be deleted as they are never used (if this is the case the supported message types of the buffer may also be deleted) or the software engineer introduced errors in the RTSCs that prevent that any of the supported message types of this buffer may be sent. In both cases, the software engineer should adjust his RTCP. Note that this requirement implies that at least one instance of one of the supported message types of each buffer can possibly be sent and, therefore, can possibly be in transit.

4.5 Translating MTCTL into English Sentences

As we defined in Section 4.2, we distinguish between several stakeholders of a DSVPL. Among others, one group of stakeholders are the DSVPL readers that do not need to define verification

properties but only need to understand them. For example, the requirements engineer must only be able to link the (informal) requirements of the systems to the properties of the DSVPL, e.g., to validate whether all requirements are covered. Therefore, he does not need to define the property, only to understand it. However, MTCTL properties may be quite difficult to understand for engineers that have only basic experience in model checking (including formal languages like CTL-based languages) as it may be the case for the requirements engineer. The reasons for this is that CTL-like properties use many uncommon keywords where the semantics are not obvious, e.g., **AG** and **leadsTo**.

As a consequence, we define an automatic translation of MTCTL properties into valid English sentences. Like Dwyer et al. [DAC98] as well as Konrad and Cheng [KC05], we expect that they are easier to understand as CTL-like properties as they state the semantics explicitly and in natural language. For example, we automatically translate the MTCTL property **AG** (**stateActive**(Overtaker.Overtaking) **implies** **stateActive**(Overtakee.NoAcceleration)); into the following sentence: “It invariantly holds that if the state Overtaker.Overtaking is active, then the state Overtakee.NoAcceleration is active as well.”

In particular, we provide translations for two use cases: The first use case is that the stakeholder wants to know the meaning of a given MTCTL property independently of its verification result. Therefore, we directly translate the property into an English sentence that expresses the same content. Thus, our example given above that the requirements engineer wants to link requirements to the specified MTCTL properties belongs to this first use case.

The second use case is that the stakeholder wants to know the meaning of a verification result of a given MTCTL property. If the verification result is that the property is fulfilled, then our translation is identical to the translation of the first use case. However, if the property is not fulfilled, then we do not just give the translation of the property and say that it is not fulfilled. Instead, we state the opposite of the verified MTCTL property as an English sentence. For example, the opposite of the MTCTL property given above is “It can possibly happen that the state Overtaking.Overtaker.Overtaking is active and the state Overtaking.Overtakee.noAcceleration is not active.” In our opinion, this opposite form is not always obvious as MTCTL allows to form complex properties (e.g., by using the logical operators). Therefore, we support the software engineer to understand which property of the RTCP actually holds.

In the remainder of this section, we explain our approach, which consists of four steps, for translating MTCTL properties into English sentences for both use cases. In the **first step**, we check whether we are in use case 2 and whether the property is not fulfilled. If this is the case, we negate the complete property using the logical operator **not**. Otherwise, we do nothing within this step. For example, if our exemplary overtaking property is not fulfilled, then we adapt it to **not** (**AG** (**stateActive**(Overtaker.Overtaking) **implies** **stateActive**(Overtakee.noAcceleration)));.

The model representation of each MTCTL property is in the form of a tree, where each atomic MTCTL expression (i.e., temporal quantifier, set quantifier, predicates, logical operators, element references) represents a node. For example, we illustrate the tree representation of our property **not** (**AG** (**stateActive**(Overtaker.Overtaking) **implies** **stateActive**(Overtakee.noAcceleration))); in Figure 4.2. Therefore, in the **second step**, we traverse the model representation in a top down manner in order to assemble the English sentence. For doing this, we provide an English template for each kind of atomic MTCTL expression. We list all our templates in Table 4.1, Table 4.2, and Table 4.3. In particular, these tables show in their first column the original MTCTL expression, in their second column

the template for the direct translation to English, and in their third column the template for the negated variant of the translation. We require this third column if at least one atomic expression of the tree contains a **not** operator in order to negate the expression that is contained below this operator. Noteworthy, integer numbers and references to model elements are not listed in the tables as we can directly integrate the numbers to the sentence and as we call MECHATRONICUML's internal function to retrieve the fully qualified names of the model elements. Moreover, we translate all abbreviated time units into full words, e.g., we write milliseconds instead of msec.

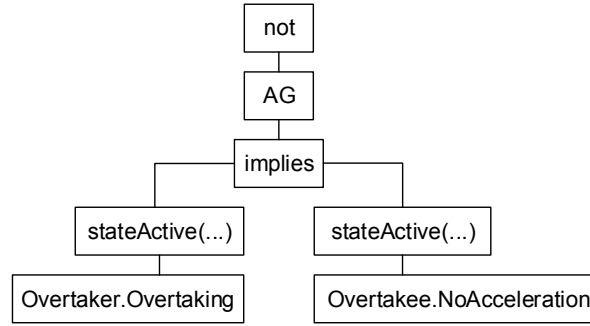


Figure 4.2: Tree Representation of the MTCTL Property **not** (**AG** (**stateActive**(Overtaker.Overtaking) **implies** **stateActive**(Overtakee.NoAcceleration)));

We depict the traversal for our exemplary property in Figure 4.3 on page 128. Its description is as follows: We always start the root expression with the English template. In our case, the root expression is **not** and the English template is `!expr`. This means that we do not have to add English words to our sentence and that we have to use the negated English template for the expression that this tree node contains. In our case, the contained expression defines the temporal quantifier **AG**. Its negated English template is “it can possibly happen that [expr]”. We always resolve a template from left to right. Therefore, we first add the words “it can possibly happen that ” to our sentence and then call the negated English template for the contained expression of the temporal quantifier. This expression is the logical operator **implies**. According to its negated English template, we first call the English template for the left expression. We resolve the left part completely (we traverse the complete subtree) before we continue with the other parts of this template. The left part of the **implies** expression references the predicate **stateActive**. Therefore, according to its English template, we first add the words “the state ” to our sentence and then call MECHATRONICUML's internal function for retrieving the qualified name of a state expression. As a result, we add “Overtaker.Overtaking ” to our sentence. Then, we go back to the **stateActive** predicate and add the words “is active ” to our sentence. Next, we go back to the logical operator **implies**. Now, we have completely resolved its left expression. Therefore, we add the word “and ” to our sentence. Afterwards, we call the negated English template of the right expression. The right part of the **implies** expression references the predicate **stateActive** as well. According to its negated English template, we add the words “the state ” to our sentence and retrieve the qualified name of the state expression. Therefore, we add “Overtakee.NoAcceleration ” to our sentence and return to the **stateActive** predicate. Here, we add the words “is not active ” to our sentence. Finally, we traverse the tree back to the root without adding anymore words. The resulting sentence is:

Expression	English Template	Negated English Template
AG [expr]	it invariantly holds that [expr]	it can possibly happen that [!expr]
EF [expr]	it can possibly happen that [expr]	it invariantly holds that [!expr]
AF [expr]	eventually it holds that [expr]	it might always hold that [!expr]
EG [expr]	it might always hold that [expr]	eventually it holds that [!expr]
[leftExpr] leadsTo [rightExpr]	whenever [leftExpr], then eventually [rightExpr] as well	it can possibly happen that when [leftExpr] it might always hold that [!rightExpr]
deadlock	a deadlock occurs	no deadlock occurs
connectorOverflow	a connector overflow occurs	no connector overflow occurs
stateInStatechart ([state],[statechart])	the state [state] is in statechart [statechart]	the state [state] is not in statechart [statechart]
transitionFiring ([transition])	the transition [transition] is firing	the transition [transition] is not firing
substateOf ([state],[statechart])	the state [state] is a substate of the statechart [statechart]	the state [state] is not a substate of the statechart [statechart]
stateActive ([state])	the state [state] is active	the state [state] is not active
messageDiscarded ([buffer])	a message in the buffer [buffer] gets discarded	no message in the buffer [buffer] gets discarded
messageInTransit ([message])	the message [message] is in transit	the message [message] is not in transit
messageInBuffer ([message],[buffer])	the message [message] is in the buffer [buffer]	the message [message] is not in the buffer [buffer]
[leftExpr] and [rightExpr]	[leftExpr] and [rightExpr]	[!leftExpr] or [!rightExpr]
[leftExpr] or [rightExpr]	[leftExpr] or [rightExpr]	[!leftExpr] and [!rightExpr]
not [expr]	[!expr]	[expr]
[leftExpr] implies [rightExpr]	if [expr.left] then [expr.right] as well	[expr.left] and [!expr.right]

Table 4.1: English Templates and Negated English Templates (Part 1)

Expression	English Template	Negated English Template
forall ([var]:States) [expr]	for all states [var] [expr]	there exists a state [var] so that [!expr]
exists ([var]:States) [expr]	there exists a state [var] so that [expr]	for all states [var] [!expr]
forall ([var]:Transitions) [expr]	for all transitions [var] [expr]	there exists a transition [var] so that [!expr]
exists ([var]:Transitions) [expr]	there exists a transition [var] so that [expr]	for all transitions [var] [!expr]
forall ([var]:Clocks) [expr]	for all clocks [var] [expr]	there exists a clock [var] so that [!expr]
exists ([var]:Clocks) [expr]	there exists a clock [var] so that [expr]	for all clocks [var] [!expr]
forall ([var]:MessageTypes) [expr]	for all messages [var] [expr]	there exists a message [var] so that [!expr]
exists ([var]:MessageTypes) [expr]	there exists a message [var] so that [expr]	for all messages [var] [!expr]
forall ([var]:Buffers) [expr]	for all buffers [var] [expr]	there exists a buffer [var] so that [!expr]
exists ([var]:Buffers) [expr]	there exists a buffer [var] so that [expr]	for all buffers [var] [!expr]
forall ([var]:[lowerBound],[upperBound]) [expr]	for all numbers [var] between [lowerBound] and [upperBound] [expr]	there exists a number [var] between [lowerBound] and [upperBound] [!expr]
exists ([var]:[lowerBound],[upperBound]) [expr]	there exists a number [var] between [lowerBound] and [upperBound] [expr]	for all numbers [var] between [lowerBound] and [upperBound] [!expr]
forall ([var]:Instances) [expr]	for all instances [var] [expr]	there exists an instance [var] so that [!expr]
exists ([var]:Instances) [expr]	there exists an instance [var] so that [expr]	for all instances [var] [!expr]
forall ([var]:Subinstances) [expr]	for all subinstances [var] [expr]	there exists a subinstance [var] so that [!expr]
exists ([var]:Subinstances) [expr]	there exists a subinstance [var] so that [expr]	for all subinstances [var] [!expr]

Table 4.2: English Templates and Negated English Templates (Part 2)

Expression	English Template	Negated English Template
<code>bufferMessageCount</code> ([buffer])	the number of messages in buffer [buffer]	<i>not necessary</i>
<code>sourceState</code> ([transition])	, which is the source of transition [transition],	<i>not necessary</i>
<code>targetState</code> ([transition])	, which is the target of transition [transition],	<i>not necessary</i>
[leftExpr] == [rightExpr]	[leftExpr] is [rightExpr]	[leftExpr] is not [rightExpr]
[leftExpr] != [rightExpr]	[leftExpr] is not [rightExpr]	[leftExpr] is [rightExpr]
[leftExpr] > [rightExpr]	[leftExpr] is more than [rightExpr]	[leftExpr] is at most [rightExpr]
[leftExpr] >= [rightExpr]	[leftExpr] is at least [rightExpr]	[leftExpr] is less than [rightExpr]
[leftExpr] < [rightExpr]	[leftExpr] is less than [rightExpr]	[leftExpr] is at least [rightExpr]
[leftExpr] <= [rightExpr]	[leftExpr] is at most [rightExpr]	[leftExpr] is more than [rightExpr]

Table 4.3: English Templates and Negated English Templates (Part 3)

“it can possibly happen that the state `Overtaking.Overtaker.Overtaking` is active and the state `Overtaking.Overtakee.noAcceleration` is not active”.

If we generate an English sentence for use case 2, we add in the **third step** the following prefix to the sentence: “the verification result is that ”. This is the case for our unfulfilled overtaking property. Therefore, we adapt it to “the verification result is that it can possibly happen that the state `Overtaking.Overtaker.Overtaking` is active and the state `Overtaking.Overtakee.noAcceleration` is not active”.

The **fourth step** is to change the first letter of the sentence to an upper case and make a period at the end of the sentence. As a result, we get a complete and valid English sentence. In our case, the resulting sentence is: “The verification result is that it can possibly happen that

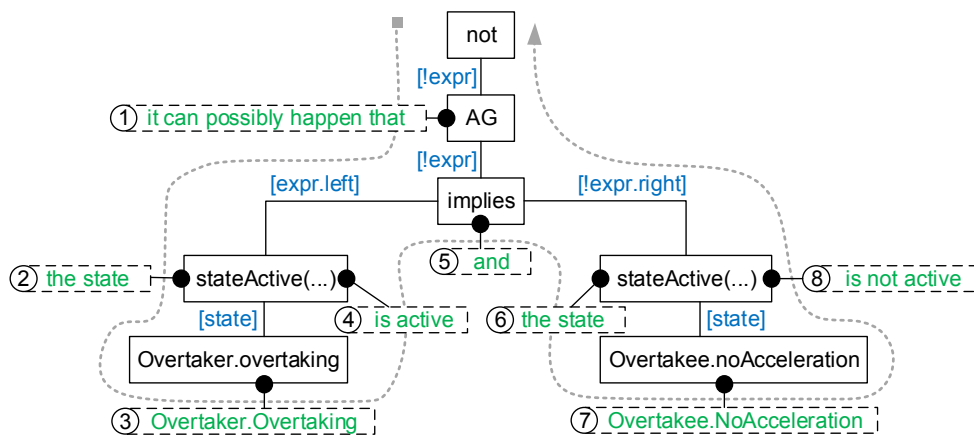


Figure 4.3: Traversing the Tree Representation for Producing the English Sentence

the state `Overtaking.Overtaker.Overtaking` is active and the state `Overtaking.Overtakee.noAcceleration` is not active.”.

A limitation of our approach is that we always translate a property to exactly one English sentence. Depending on the property, this can lead to long sentences that may be hard to understand. For example, consider the following property for the RTCP Overtaking: `(AG not messageDiscarded(Overtaker.b1)) and (AG not messageDiscarded(Overtakee.b2))`;. According to our concept, we translate it to the sentence “It invariantly holds that no message in the buffer `Overtaking.Overtaker.b1` gets discarded and it invariantly holds that no message in the buffer `Overtaking.Overtakee.b2` gets discarded.”. In our opinion, the sentence is still understandable, but the understandability might be increased by splitting the sentence into two: (1) “It invariantly holds that no message in the buffer `Overtaking.Overtaker.b1` gets discarded”. (2) “It invariantly holds that no message in the buffer `Overtaking.Overtakee.b2` gets discarded”. Splitting the sentence is possible as we have two independent temporal properties that are connected via a logical conjunction. This example only uses one logical conjunction but an MTCTL property may have arbitrary many conjunctions, which leads to even more complex sentences.

As another limitation, we only translate the property in a straightforward manner and only negate parts of the sentence when applicable but do not simplify the resulting sentence any further to increase the understandability even more. For example, we could simplify the property concerning the two buffers given above to the following sentence: “No message will ever be discarded in any buffer.”. This simplification is possible as RTCP Overtaking only consists of these two buffers. This simplified sentence is—in our opinion—a better variant compared to our direct translation (one long sentence) or the two sentences given above as it uses significantly less words and summarizes the content of the property on a more abstract level.

4.6 Implementation

In this section, we present excerpts of our prototypical implementation of an MTCTL editor and corresponding result dialogs. Our implementation is integrated into the MECHATRONIC-UML Tool Suite [*DGB+14].

In particular, we present the user interface in Section 4.6.1 and the plugin structure in Section 4.6.2. The complete grammar of MTCTL as well as its metamodel are presented in Appendix E.2.

4.6.1 User Interface

We provide a textual editor based on Xtext [Xtext] for specifying MTCTL properties as depicted in Figure 4.4. Our editor supports syntax coloring and a content assist that supports in completing a property. Moreover, when hovering with the mouse over a property, we provide additional information to the selected partial property as well as the complete property. Among others, we show the complete English sentence that represents the complete MTCTL property.

In addition, we enable a live validation of our properties and give reasonable error statements including proposals for solving the problem. For example, in the screenshot of Figure 4.5,

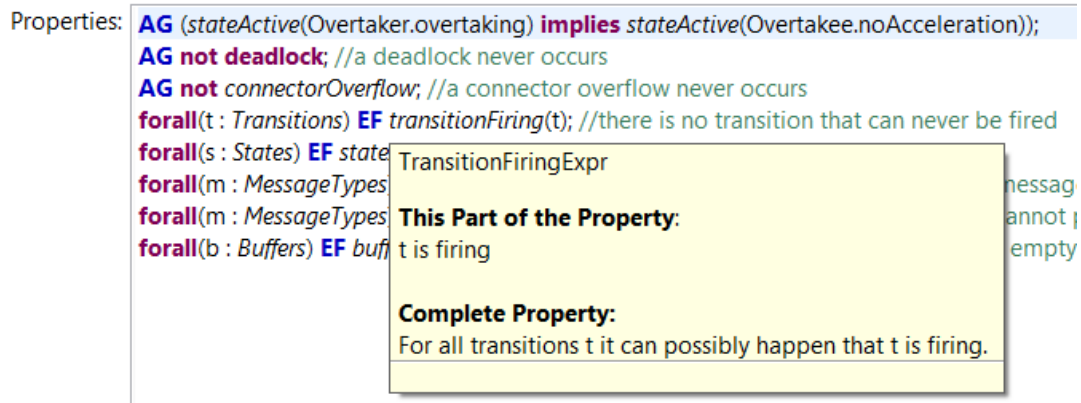


Figure 4.4: MTCTL Editor and MTCTL to English as Mouse-Over Event

the incorrect stated property is **not connectorOverflow**. The problem is that the expression **connectorOverflow** must be bound to a temporal quantifier like **AG**.

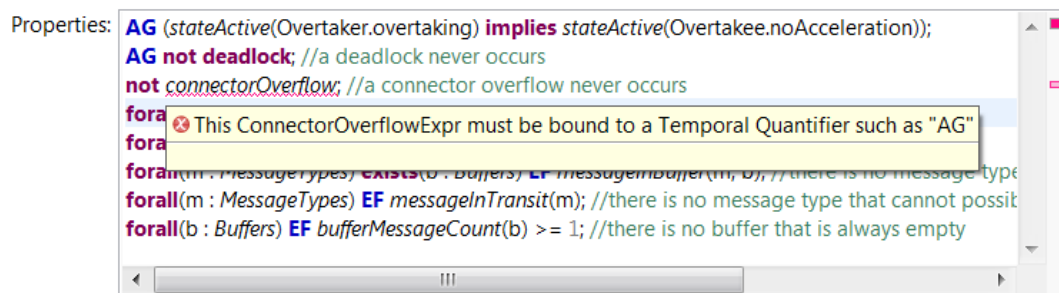


Figure 4.5: Live Syntax Check and Error Messages

As stated in Section 4.4, we provide default properties that are valid for all protocols. Figure 4.6 shows the user interface for adding default variables: It is available by right-clicking on the ellipse of an RTCP and then selecting “MechatronicUML » Add Default Verification Properties”.

4.6.2 Plugin Structure

We show an excerpt of the plugins that we developed and adapted for our prototypical implementation in Figure 4.7. Note that we provide the documentation of our grammar and our metamodel in Appendix E.2.

First of all, we define the grammar of MTCTL using Xtext in the plugin `mtctl.xtext`. Moreover, we define the metamodel of MTCTL in the plugin `mtctl`. Both plugins use the plugin `pim`, which provides the metamodel of the MECHATRONICUML platform independent model (`pim`), in order to reference all relevant parts of an RTCP. Thanks to Xtext, we do not have to define the parser, which translates textual inputs into a model (according to the given metamodel), the serializer, which transforms a model into the textual representation, and the MTCTL editor as Xtext generates all of them. The plugin `mtctl.english` contains the transformation from MTCTL properties to English sentences, which we realized using Xtext2.

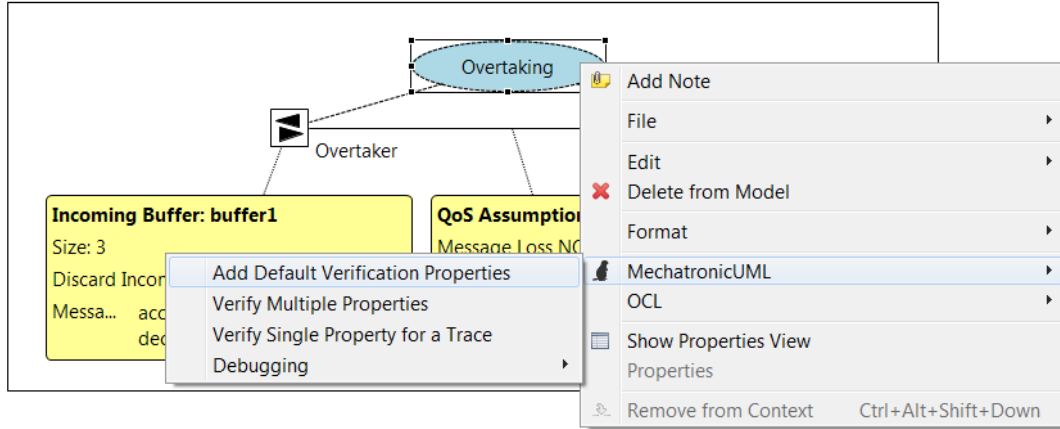


Figure 4.6: Adding Default Verification Properties

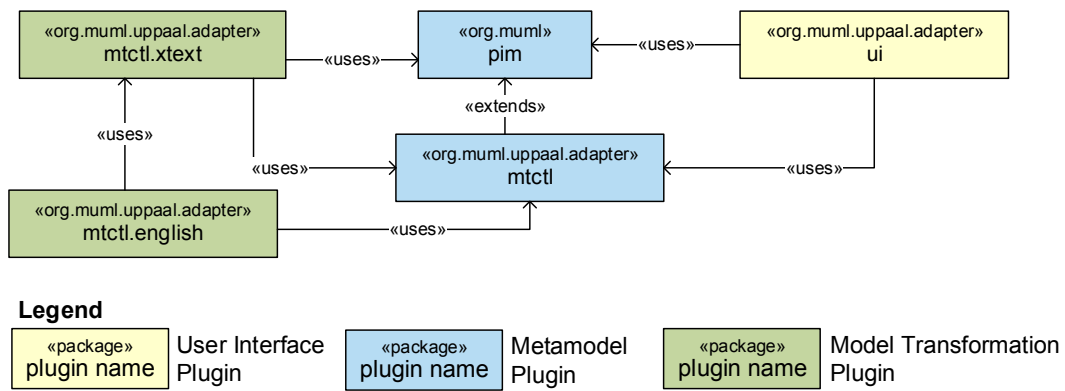


Figure 4.7: Plugins used for implementing MTCTL

Lastly, the plugin `ui` defines—among others—the context menu entry to generate the MTCTL default properties.

4.7 Evaluation

We conduct a case study based on the guidelines by Kitchenham et al. [KPP95] and Runeson et al. [RHRR12] for evaluating our approach. In particular, we evaluate MTCTL for realistic examples within the domain of CPSs but do not aim for generalizations. Noteworthy, all examples of this case study are defined by ourselves or colleagues and are not taken from real industry examples. We present the core points of our evaluation in this section and give more details in Appendix C.2.

Our evaluation is structured as follows: First, we present the context of our case study in Section 4.7.1. Then, we set our hypotheses in Section 4.7.2 and prepare the validation in Section 4.7.3. Afterwards, we validate the hypotheses in Section 4.7.4. Finally, we discuss the threats to validity in Section 4.7.5 and analyze our results in Section 4.7.6.

4.7.1 Case Study Context

The main objective of our case study is to evaluate whether we can express realistic verification properties for a given set of realistic RTCPs. Thus, we evaluate whether we fulfill the requirement R1 that we stated in Section 4.2. In addition, we want to evaluate whether our translation from MTCTL to English always produces valid sentences that preserve the semantics of the properties. This is a necessary prerequisite in order to fulfill requirement R4 (“be understandable for all stakeholders”).

We conduct our case study using the one-to-one RTCP Overtaking, the one-to-many RTCP MultiOvertaking, and a normalized form of RTCP MultiOvertaking that uses an explicit subrole region, which we call MultiOvertaking_wSubroleRegion. These RTCPs are good representatives as they define complex communication behavior and use many different model elements of an RTCP. Therefore, they enable to define a heterogeneous set of verification properties.

4.7.2 Setting the Hypotheses

Our case study has two evaluation hypotheses:

- H1** All given informal requirements can be stated as formal MTCTL properties without changing the RTCP.
- H2** All given MTCTL properties are translatable into valid English sentences that preserve the semantics of the properties.

For evaluating hypothesis H1, we manually state the informal requirements as formal MTCTL properties. As we (a colleague and I) are experts of MECHATRONICUML and MTCTL, we can evaluate whether an MTCTL property has the correct semantics. Then, we check whether the MTCTL parser accepts these properties. We expect that the parser accepts all of the properties. Concerning hypothesis H2, we automatically translate the MTCTL properties into English sentences. As we are familiar with the English language, we manually check whether the resulting English sentences are valid. Then, we manually check whether the semantics are preserved. We expect that we translate a valid English sentence for each property and that each sentence preserves the semantics of the formal property.

4.7.3 Preparing the Validation

At first, we define 28 informal requirements for both RTCPs in total and document them in Appendix C.2. In particular, the requirements were defined by me after I wrote the concepts of MTCTL. We consciously defined the informal requirements such that we cover a huge range of MTCTL’s expressiveness. Therefore, while formulating the requirements, we take care that we state various quantitative timing constraints, reference various elements of the RTCP (states, transitions, buffer, etc.), and define various requirements for a snapshot (to enforce the usage of various predicates and operations). For example, for the one-to-one RTCP Overtaking, the first informal requirements is “If the overtaker is in state Overtaking, then the overtakee is in state NoAcceleration.”.

We expect that most of the informal requirements can be formalized via MTCTL properties. Therefore, we will reuse the resulting set of MTCTL properties as input for the translation

into English sentences (to analyze H2). As we aim to have a set of MTCTL properties that cover a huge set of MTCTL's expressiveness, we also expect that we can cover a huge part of our translation to English sentences.

4.7.4 Validating the Hypotheses

We document our manual translations from informal requirements to MTCTL properties in Appendix C.2. Our result is that we are able to formalize 25 of the 28 requirements successfully. For example, we translate the first informal requirement into the property **AG** (`stateActive(Overtaker.Overtaking) implies stateActive(Overtakee.NoAcceleration)`). Moreover, all 25 MTCTL properties are accepted by the MTCTL parser.

We are not able to formalize the requirements 4 (“It is possible to initiate the overtaking more than once.”) and 5 (“If an overtaking request is declined, it is possible to request again and get an accept.”). Both require nested temporal quantifiers that MTCTL does not provide because UTCTL does not support it as well (see Section 2.1.2). For requirement 4, we would need to be able to express that it is possible to reach the state `Overtaker.Overtaking`, to leave this state afterwards, and to reach it again. Therefore, we would need 3 EF quantifiers where the second depends on the first and the third on the second quantifier. Requirement 5 is very similar as we would need to be able to express that it is possible that the `Overtaker` receives message `decline`, sends message `request` afterwards, and gets the message `accept` as a result. Again, we would need 3 EF quantifiers where the second depends on the first and the third on the second quantifier. However, a workaround to formalize both requirements exists: We could add variables within the RTCP that count how often a state is visited and how often a transition fires. Then, we could reference these variables within an MTCTL query. Nevertheless, we failed to express both requirements without changing the RTCP. A possible solution for this problem would be that we enrich MTCTL with additional keywords for reoccurring test cases (e.g., a state is activated more than once, a transition is fired more than once). Then, our DSMC (see Chapter 5) could integrate our workaround within the verification model. Consequently, our design model would remain unchanged.

Moreover, we are not able to formalize the requirement 22 (“The overtakee instance is at most 10 seconds in state `NoAcceleration.LaneChanged`.”) because we can only formulate real-time constraints in MTCTL when referencing an existing clock but the RTCP `Overtaking` defines no clock that is reset while entering the state `NoAcceleration.LaneChanged`. Noteworthy, UTCTL is also not able to express real-time constraints without referencing clocks. Again, a workaround would be to add an additional clock `c2` to the RTCP and reset it while firing the transition that leads to the state. Nevertheless, we failed again to express the property without changing the RTCP. A possible solution for this problem would be to enable clock constraints that use additional keywords that refer to events during the execution of the RTCP (e.g., a state gets activated, deactivated, a transition is fired, a message is sent).

We are able to formulate requirement 2 (“If the overtakee receives an overtaking request, it will either send an accept or decline message.”) via the MTCTL property `transitionFiring(Overtakee.NoOvertaking1.init1_to_received) leadsTo (transitionFiring(Overtakee.noOvertaking1.received_to_init1) or transitionFiring(Overtakee.noOvertaking1.received_to_exit1))`; Therefore, we can express the requirement using MTCTL. However, our expression is rather cumbersome as we cannot straightforwardly express that a message is being sent or received directly. Instead, we have

to use the predicate `transitionFiring` with the transition that sends this message. A possible solution for this minor problem would be to add two dynamic predicates `messageConsuming` and `messageSending` to MTCTL that both reference a message type. They would result to true when a message is being consumed or sent independently of the transition that consumes or sends the message (more than one transition may send and receive a specific message type).

Our 25 MTCTL properties reference all possible model elements and use all available temporal quantifiers, all dynamic and static predicates, all operations, all logical expressions (and, or, not, implies), all comparison operators, and both set quantifiers. Therefore, these MTCTL properties cover a huge range of MTCTL's expressiveness, where no important concepts of MTCTL are missing. Consequently, they are useful for our evaluation of H2 that addresses the correctness of our translation to English sentences.

We execute our MTCTL to English translation using the 25 MTCTL properties and document the outputs of this translation in Appendix C.2. As a result, our translation is able to produce for each MTCTL property a valid English sentence (concerning its grammar and style) that is semantically identical to the MTCTL property. For example, the first MTCTL property is translated into the following sentence: "It invariantly holds that if `Overtaking.Overtaker.Overtaking` is active, then `Overtaking.Overtakee.noAcceleration` is active as well.". The generated output has the same meaning as the MTCTL property and is a valid English sentence. Finally, we investigate whether the English translation is deterministic, i.e., whether it always produces the same sentences. We do this by executing the translation once more for each MTCTL property. The result is that the translation produces the identical English sentences. Therefore, we conclude that the translation is deterministic for this use case and these properties.

4.7.5 Threats to Validity

According to Runeson et al. [RHRR12], we classify our threats to validity into three parts: construct validity (is the case study designed correctly to answer the hypotheses?), external validity (what hinders us to generalize our findings?), and reliability (would the result be the same with another set of researches?). In the following, we discuss our threats based on this classification schema and report how we consider these threats in the design of our study. Noteworthy, Runeson et al. also define a fourth class called internal validity that we do not need as we do not examine causal relations.

4.7.5.1 Construct Validity

- The informal requirements were only defined by me, who designed MTCTL. Moreover, I alone translated them to MTCTL properties. Thus, our evaluation would have more significance if the requirements definition and the translation to MTCTL would have been executed by multiple software engineers (preferable from different companies), who are experts in CPSs and who were not involved in designing our language.

4.7.5.2 External Validity

- We only considered 29 informal properties for only two different RTCPs. Moreover, only five informal properties refer to a one-to-many RTCP. Even though we consider

our examples as realistic, other realistic requirements (of other RTCPs) could exist that we cannot translate into MTCTL at all (not even by changing the RTCP).

- It might be an exception that our automatic translation from MTCTL to English sentences works perfectly. Again, other examples could reveal problems with our automatic translation. In particular, our 26 MTCTL properties do not cover all 83 English phrases that we provide in Table 4.1, 4.2, and 4.3 (40 positive phrases, and 43 negated phrases). Instead, we only cover 34 phrases. Furthermore, we obviously do not cover all possible combinations of these phrases.

4.7.5.3 Reliability

- We might have made mistakes in our manual comparison of the semantics between the informal requirement and the MTCTL property as well as between the MTCTL property and the English sentence. Thus, our conclusions may be wrong. We reduce the possibility of this threat via triangulation [RHRR12] as I did not compare the semantics only by myself. Instead, a colleague of our research group, who has two years of experience with MTCTL, compared the semantics as well.
- Analogously, the validity of the English sentences (their grammar and style) were also checked by a colleague, who has a lot of experience concerning the English language.

4.7.6 Analyzing the Results

Our case study demonstrates that we are able to formalize the majority (ca. 90%) of the informal requirements directly without changing the RTCP. However, we were not able to formalize all requirements. Therefore, hypothesis H1 is only partially fulfilled. Nevertheless, the expressiveness of MTCTL seems to be good enough to be used productively. The three requirements that we cannot express would also not be expressible in UTCTL. We identified for each of the three requirements workarounds that require small changes to the RTCP. Though, these changes would pollute the RTCP with variables, clocks, and expressions that have no functional reason. In addition, some requirements cannot be formalized straightforwardly as fitting predicates are missing. We defined for each problem that we found a possible solution that requires a new revision of MTCTL.

We consider our second hypothesis as fulfilled because all 26 MTCTL properties were successfully translated into valid English sentences that preserve the semantics.

In summary, it seems that MTCTL is ready to be used as part of a DSMC of RTCPs (see Chapter 5). Moreover, it appears that we can ease the understandability of verification properties for MTCTL readers (e.g., requirements engineers), who only have basic experience in model checking and TCTL in particular, due to our correctly working translation to English sentences. Therefore, they should be able to assess the relevance of a property and to link MTCTL properties to their artifacts (e.g., top-level requirements).

4.8 Limitations

Our DSVPL MTCTL underlies the following conceptual limitations:

- Our evaluation showed that MTCTL would benefit from two new dynamic predicates: `messageConsuming` and `messageSending`.

- Our evaluation showed that MTCTL would benefit from the possibility to express real-time constraints without referencing an existing clock.
- Our concept requires that the MTCTL specifier has to select between the two set quantifier **forall** and **exists** when making statements about single and multi roles. However, a multi role always has exactly one instance. Therefore, the verification result does not depend on the selected set quantifier. Thus, this is a superfluous detail.
- Our concept does not enable to handle subrole-specific variables, clocks, and operations for a one-to-many RTCP that uses one-to-many communication schemata. Therefore, an MTCTL specifier first has to normalize the RTCP into an RTCP with an explicit subrole (see Section 3.5.2) before he can specify the properties.
- We only support variables that are primitive types. Hence, we do not support (multidimensional) arrays, enumerations, and structs, which are all supported by MECHATRONICUML [*DPP+16].
- Our translation to English sentences translates each expression on its own and does not simplify the returning sentence. We gave examples for this in Section 4.5.
- We can only generate MTCTL expressions into valid English sentences, not the other way around.

Moreover, we currently have the following implementation-specific limitations:

- If an MTCTL property references a design model element (e.g., a state) and the element is being deleted, then our implementation does not state this fact in a user-friendly way. Instead, Xtext is not able to parse the property as the identifier refers to null. A more user-friendly way would be if the MECHATRONICUML Tool Suite would check at front whether the deletion of this element affects the stated MTCTL properties. If this is the case, then the software engineer should be informed that deleting this element also causes the deletion of the properties that directly reference this element.
- Single-line and multi-line comments are accepted by the grammar but are not stored. Only comments that are in the same line as a property are stored.

4.9 Related Work

We distinguish related work in four different topics. First, we discuss related verification property languages that enable to specify real-time properties for (domain-specific) model-based languages such that the software engineer can apply model checking. In particular, we distinguish between TCTL-based languages (Section 4.9.1) and OCL-based languages (Section 4.9.2). Second, we discuss an approach that enables to generate a DSVPL based on a given DSL in Section 4.9.3. Third, we discuss existing DSVPLs in MECHATRONICUML (Section 4.9.4). Fourth, we discuss related approaches that facilitate the specification and understanding of verification properties in Section 4.9.5 and 4.9.6.

4.9.1 TCTL-Based Languages

Roubtsova et al. [RvKT+00] use a variant of TCTL [STA98] for specifying verification properties of models that are defined using a variant of UML that has been extended by real-time capabilities. They verify their models using their own model checker called Prototype Model Checker (PMC), which accepts their variant of TCTL. Thus, when translating their

UML model to PMC, they only have to update the references to the model elements within their TCTL properties such that they do not longer reference the elements of UML model but the elements of the PMC input model. Therefore, in contrast to MTCTL, the authors do not have to consider the capabilities of an underlying verification property language. Among others, they extend their variant of TCTL by stereotypes to simplify the specification, e.g., “to define the deadline of an operation or to specify an order in which states of the system are reached” [RvKT+00, p. 9] However, their variant of TCTL still only focuses on specifying properties by referencing elements of a state machine (a state, an event, ...). In contrary, MTCTL also has to consider that the software engineer needs to specify properties concerning domain aspects that are no longer specified using state machines only (e.g., MTCTL provides predicates for incoming message buffers and message in transit).

Suck et al. “reduce the model checking problem for [typed attributed graph transformation systems [EEPT06, p. 181] extended by real-valued clocks as in timed automata [ACD93]] to the well-studied problem of model checking for timed automata in order to benefit from the existing theory and tools.” [SHS11, p. 3] For doing this, they define a formalism called Timed Graph Transformation System (Timed GTS, [EHH+13]) and a verification property language called First-Order TCTL (FO-TCTL, [SHS11; Suc11]). FO-TCTL “[...] is an extension of TCTL by constructs of first-order logic” [SHS11, p. 2] and enables to reference objects of a Timed GTS. In particular, FO-TCTL defines variables and constant symbols “[...] to represent nodes of a graph whereas predicates describe certain properties of or relations between nodes respectively.” [SHS11, p. 13] Therefore, FO-TCTL is similar to MTCTL as it enriches TCTL using predicates in order to reference specific language aspects of Timed GTS (here: the nodes and the edges of the graph) that are not explicitly defined. However, in contrast to FO-TCTL, MTCTL focuses on the formal verification of a message-based communication including message buffers. Moreover, FO-TCTL only provides symbolic predicates that are specific for the model under analysis, i.e., they provide symbolic predicate for each possible node type, edge type, and node identifier. In contrary, MTCTL provides a fixed set of operations and predicates, which is independent of the model under analysis.

4.9.2 OCL-Based Languages

OCL [OCL] is a well-known constraint language that is independent of a specific formal verification tool. Several approaches already exist that enable to specify the verification properties using (a variant of) OCL and enable a translation into the input language of a model checker. In the following, we present the two approaches that enable to define real-time verification properties using a variant of OCL and enable a translation to a time-bounded variant of CTL [CE82] that uses clocks like TCTL.

Flake and Mueller present “[...] an OCL extension for the specification of real-time constraints in state-oriented UML diagrams” [FM02, p. 168]. They call this language Temporal OCL (TOCL). The semantics of TOCL are defined by a translation to the language Clocked CTL (CCTL, [RK97], which uses clocks like TCTL to express real-time constraints. Moreover, CCTL is the input language of a CCTL-based model checker [RK97]. Among others, TOCL enables to specify that certain state configurations shall be active for a given time interval. Moreover, they introduce a logical until operator to be able to express dependencies between consecutive configurations. In addition, they introduce new attributes and operations that are

compliant to the UML state machine metamodel, e.g., the operation `cfg->isActive()` : Boolean returns true if all states of a configuration `cfg` are active.

Cengarle and Knapp provide a language called OCL/RT [CK02], which “[...] extends OCL by a general notion of events accompanied by time stamps, satisfaction operators on expressions, and modal operators. This extension enables the specification of several common real-time paradigms for UML models. Moreover, OCL/RT provides means to clarify the semantics of OCL invariants, pre-/post-conditions, and action clauses” [CK02, p. 18]. In particular, OCL/RT is able to define timeouts and deadlines (for operations as well as for reactions to signals). Noteworthy, OCL/RT is not defined for a particular formal analysis tool like a model checker. However, Louati et al. [LBJ15] show that they can translate OCL/RT constraints into TCTL properties.

In contrast to MTCTL, both OCL-based languages target UML state machines and not a specific DSL like MECHATRONICUML. Therefore, they do not have to introduce domain-specific model elements like MTCTL’s predicates (e.g., `messageInBuffer`). For example, a software engineer cannot use their languages directly to specify that a certain message eventually arrives in a certain buffer. Thus, they violate requirement R1 (“be able to express the relevant properties of the domain”) and, therefore, have to be customized for a DSL like MECHATRONICUML.

Furthermore, both languages do not fit for specifying properties that shall be proved using a model checker like UPPAAL as they enable constraints that are not expressible using verification property languages like UTCTL (e.g., they support nested temporal quantifiers). Thus, they violate requirement R2 (“be compatible with the verification property language the underlying model checker accepts”). Consequently, the software engineer currently needs to know the differences between the OCL-based DSVPL and the model checker’s property language. Thus, the model checking would not be hidden.

In summary, MTCTL already fulfills these two important requirements but these two OCL-based languages would need additional changes to fulfill them as well.

4.9.3 Generating the DSVPL

Meyers et al. [MWVD13] define the ProMoBox approach, which—among others—enables to generate a DSVPL based on a given DSL. Their DSVPL is mostly generic and enables to specify temporal constraints including the possibility to use quantifiers and reference the (runtime) model elements of the DSL. Moreover, the authors enable that the properties stated in the generated DSVPL may be automatically translated to Promela [Hol90] and LTL [BK08] and may be directly verified by the model checker SPIN [Hol90; Hol97]. Deshayes et al. [DMMV14] execute a case study to evaluate ProMoBox. Among others, they successfully specify and generate a DSVPL for GISMO [RM15], a DSL for executable prototyping of gestural interaction.

Meyers et al. state that their approach requires that the behavior is expressed with state machines only. However, in our case, the behavior of the buffer specification and the message transfer is not modeled using state machines but on a higher abstraction level. As a consequence, MTCTL introduces domain-specific predicates and operations that enable to specify properties concerning the role connector and the message buffers. Moreover, the authors state that they do not provide concepts for real-time properties, which are a necessary for RTCP. As a consequence, the ProMoBox approach is (currently) not directly applicable to

generate a language that is as expressive as MTCTL or to provide an automatic translation to UPPAAL, which we introduce in Chapter 5.

4.9.4 DSVPLs in MechatronicUML

Stallmann [Sta08] defines the Culture and Community-based Modeling Approach (CurCuMA), which is a framework for multi-agent system design including formal verification. One focus of Stallmann is to verify the structure of the system that may change during runtime. For specifying constraints concerning the allowed changes of the system structure, Stallmann defines a visual constraint language called Timed Story Scenario Diagrams (TSSDs). His language may reference the structural elements (e.g., the agents) of the system, can define a temporal ordering of structural situations, and allows to define real-time constraints (e.g., how much time may pass before a new structural situation must be active). In contrary, MTCTL does not constraint structural situations of an RTCP as we do not focus on reconfigurable RTCPs in this work. Instead, MTCTL focuses on the behavior of an RTCP that has a fixed structure and provides explicit predicates for these aspects.

Brune [Bru15] defines a visual DSVPL at the level of MECHATRONICUML called MECHATRONICUML Property Charts (MPC), which is a domain-specific variant of FO-TCTL (see Section 4.9.1). Therefore, MPC may be translated to textual FO-TCTL properties. MPC is inspired by MTCTL. It provides (visual) predicates that are based on MTCTL's dynamic predicates `messageInTransit`, `stateActive`, and `transitionFiring`. However, it is—among others—still not possible to refer to the properties of a message buffer (e.g., how many messages are contained).

4.9.5 Expressing Verification Properties in Natural English

Several work exist that provide a (semi-) natural English grammar the software engineer can use to specify its verification properties. The goal is always to ease the understanding of the meaning of a property, especially for stakeholders like requirements engineers that do not necessarily know property languages for model checkers. For example, Flake et al. [FMR00] provide a natural English grammar and enable to translate resulting sentences to CCTL [RK97]. Konrad and Cheng [KC05] provide a more general natural English grammar than Flake et al. that enables sentences that may be translated to several verification property languages. In particular, they support the untimed verification property languages LTL, CTL, and GIL [RMM+96] as well as the timed verification property languages MTL [Koy90], TCTL, and RTGIL [MRK+97]. Jakšić et al. [JF13] provide a natural English grammar for the domain of web document verification and enable a translation to ALCCTL [Wei08] (a variant of CTL with more expressiveness) as well as to CTL directly.

MTCTL's language is based on TCTL rather than on natural English but supports the translation to English sentences. In our stakeholder description in Section 4.2, we defined that the MTCTL specifier has at least advanced experience in timed model checking including verification property languages like TCTL. Therefore, due to our experience, we think that the MTCTL specifier can write and grasp MTCTL properties faster if they are similar to TCTL rather than an English sentence as the MTCTL form is more compact (see the evaluation tables in Appendix C.2). However, a user study that confirms our assumption did not happen yet.

In contrast to related work, we enable to present the verification results optionally as English sentences. Moreover, in the case that the verification result is that a property does not hold, we are able to rephrase our English sentences. Thus, we do not just state that the property is not fulfilled but what actually holds. In our opinion, this helps to understand the verification result as the opposite of a stated property is not always obvious but it is necessary to know for understanding the counterexample.

4.9.6 Patterns for Verification Properties

In related work, several approaches exist that support the software engineer for specifying correct verification properties by providing sets of patterns. Such patterns were initially proposed by Dwyer et al. [DAC98; DAC99] for languages like LTL and CTL. Smith et al. [SACO02] provide “[...] templates that explicitly capture [...] details as options for commonly-occurring property patterns. These templates are represented using both ‘disciplined’ natural language and finite-state automata, allowing the specifier to easily move between these two representations.” [SACO02, p. 11] Moreover, Konrad and Cheng [KC05] propose patterns of real-time verification properties based on three commonly used real-time temporal logics (e.g., TCTL).

Most of the mentioned patterns are applicable for specifying several useful verification properties for RTCPs using MTCTL. However, the existing patterns do not focus on an asynchronous message exchange of CPSs including message buffers. Moreover, due to their nature, a pattern *may* be useful for the model under development but does not have to. Therefore, the MTCTL specifier has to manually select a pattern for a given requirement and adapt it manually to his needs. In this work, we identified a set of requirements and corresponding MTCTL properties concerning the state-based runtime behavior (including asynchronous message exchanges) that shall always hold for each RTCP (except some rare cases concerning one-to-many RTCPs). We call them default properties. In contrast to patterns, these properties are always useful and therefore selected for each RTCP by default. Moreover, our default properties do not need to be adapted for each RTCP due to the expressiveness of MTCTL. Therefore, we avoid errors by the MTCTL specifier that could occur during the selection and adaptation of these properties. Noteworthy, our default properties are not a replacement for patterns but are an additional help for the MTCTL specifier. In fact, we believe that property patterns that are especially designed for RTCPs could additionally facilitate the work of the MTCTL specifier.

Fornara and Colombetti [FC03] provide a method for defining so-called agent interaction protocols (AIPs) that enable to specify the message exchange of multiagent systems. Among others, they state that an AIP must satisfy a set of general, application-independent soundness conditions in order to define the AIP as meaningful. Concerning the dynamic semantics of the AIP, they define—among others—that all paths of the diagram must be executable, i.e., it should never be the case that a guard is always false. Furthermore, they also define dynamic semantics concerning their domain elements, e.g., they define that all commitments that a role may give have to be canceled, fulfilled, or violated in the final state of the AIP. Therefore, the authors follow the same goal (ensuring that the DSL model is meaningful) as we do with the same approach (stating application-independent conditions). However, we contribute conditions that not only focus on the behavioral model (e.g., if a state is reachable) but also on the message exchange (e.g., if a message may be in transfer or if a buffer may contain

messages). Moreover, in contrast to us, their conditions are only informal and not given with a formal DSVPL. In addition, they do not state how a software engineer can verify these conditions but we do by defining that the DSMC via UPPAAL (see Chapter 5) shall be used.

4.10 Summary

In the following, we summarize to what extent this thesis and the related approaches that we discussed in Section 4.9 fulfill the requirements that we stated in Section 4.2 for a DSVPL of coordination protocols of CPSs. As a reminder, the requirements were based on our own experience and are not proven, e.g., by a systematic literature review or requirements workshops.

	this thesis	Roubtsova et al.	Suck et al.	Flake et al.	OCL/RT	ProMoBox	Stallmann	Brune	Konrad et al.	Jakšić et al.	Smith et al.	Fomara et al.
R1	✓	(✓)	×	(✓)	(✓)	×	×	×	(✓)	×	(✓)	×
R2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R4	✓	×	×	✓	×	×	×	×	✓	✓	✓	×
R5	✓	×	×	×	×	×	×	×	✓	×	✓	✓

Table 4.4: Comparison of the Requirements Fulfillment for a DSVPL for CPSs

We provide an overview of the requirements fulfillment in Table 4.4 (Legend: ✓= fulfilled, (✓) = partially fulfilled, ×= not fulfilled). As the table shows, this thesis fulfills all requirements that we stated at the beginning of this chapter. Moreover, no other approach or a combination of approaches would fulfill all requirements.

The most important requirement is R1, which states that the DSVPL shall be able to express the relevant properties of the domain. In our case, the approaches shall be able to express the relevant properties for coordination protocols of CPSs. In our evaluation in Section 4.7, we show that this thesis fulfills the requirement. Though, it sometimes requires small changes to the RTCP. In contrary, no related approach that we studied completely fulfills this requirement: the majority of the approaches either does not focus on the specification of behavioral properties or does not consider important aspects of CPSs, e.g., real-time constraints. In contrary, the approaches of Roubtsova et al., Flake et al., OCL/RT, Konrad et al., and Smith et al. partially fulfill this requirement as they enable to specify constraints for this domain. However, they do not provide model elements that enable to make direct statements concerning domain aspects like an asynchronous message exchange (including message buffers). Moreover, some of them enable to express properties that are not expressible in UTCTL. Therefore, all related approaches would require additional changes to their language.

The requirements R2 (be compatible with the verification property language the underlying model checker accepts) and R3 (be directly integrated into the DSML) are fulfilled by all approaches. Though, most of them do not support a model checker that supports real-time constraints (e.g., UPPAAL or other model checker that support TCTL properties). Concerning

R2, we show in Chapter 5 that this thesis fulfills the requirement as we provide concepts to translate all language aspects of MTCTL (see Section 5.4) and as the evaluation of our DSMC (see Section 5.8) shows that the translation works correctly. Noteworthy, we do not violate R2 even though MTCTL is more expressible than UTCTL. The reason for is that we split MTCTL properties before we translate them into UTCTL properties.

The assessment of requirement R4 (be understandable for stakeholders that are no experts in model checking) is subjective. In our opinion, this thesis fulfills the requirement as it enables to translate MTCTL properties to valid English sentences. Therefore, the DSVPL reader does not have to understand the property directly. Flake et al, Konrad et al., Jakšić et al., and Smith et al. enable to specify the property in an English sentence (we refrained from doing this as the English sentence often becomes very long compared to the MTCTL property). Therefore, they also fulfill this requirement.

The assessment of requirement R5 (provide concepts to make experience-based knowledge accessible to inexperienced DSVPL specifiers) is subjective as well. In our opinion, this thesis as well as the approach of Konrad et al., Smith et al., and Fornara et al. fulfill the requirement. We fulfill it by providing default properties that shall always hold for all RTCs. Therefore, the DSVPL specifier does not have to come up with a list of useful properties. In contrary, the other approaches provide property pattern, i.e., they identified informal requirements that are often useful and provide templates that lead to correctly specified properties that capture the meaning of the requirement.

DOMAIN-SPECIFIC MODEL CHECKING OF RTCPs

In this chapter, we explain our fully automatic domain-specific model checking (DSMC) approach for MECHATRONICUML’s RTCPs using the model checker UPPAAL. The input of our DSMC is the RTCP under development (see Chapter 3) and a set of MTCTL properties (see Chapter 4); the output of the DSMC states the fulfillment of each property and may also provide a (counterexample) trace at the DSL level (here: MECHATRONICUML).

In particular, we start with the contributions of this chapter in Section 5.1. Then, we point out the stakeholders and requirements of our DSMC in Section 5.2. In Section 5.3, we explain our concept for complex DSMC scenarios by utilizing model-to-model (M2M) traceability. Based on our concept, we explain our DSMC between MECHATRONICUML and UPPAAL: First, we explain the forward translation from RTCPs to UPPAAL in Section 5.4. Second, we explain the necessary activities at the level of UPPAAL in Section 5.5. Third, we explain the back-translation of results and (counterexample) traces in Section 5.6. Afterwards, we explain our implementation in Section 5.7. We describe our evaluation in Section 5.8 before stating the current limitations of our work in Section 5.9. Then, we discuss related work in Section 5.10. Finally, we summarize this chapter in Section 5.11.

Contents of this chapter have been published in [*GSDH15] and [*DGH15]. Moreover, parts of this chapter were contributed by the master’s thesis of Christopher Gerking [Ger13] and the project group Cybertron [BCD+14]. Both, Christopher Gerking and the project group, were supervised by me. In particular, Gerking provides an initial version of our DSMC concept that we specify in Section 5.3. Moreover, he defines a subset of the transformation from MECHATRONICUML to UPPAAL (see Section 5.4), the automation of UPPAAL (see Section 5.5), and a subset of the transformation back to MECHATRONICUML (see Section 5.6). The project group Cybertron extends both the transformation to UPPAAL and the back-transformation to MECHATRONICUML, e.g., by defining a translation of MTCTL properties to UTCTL.

5.1 Contributions

The main contributions of this chapter are as follows:

- We define a set of nine requirements that a domain-specific model checking approach for coordination protocols of CPSs (e.g., RTCPs) shall provide.
- We provide a DSMC concept that is especially appropriate for large differences between the DSL and the model checker’s language as it enables to specify a multi-step

translation to the model checker but only requires a single step for the back-translation to the DSL.

- We define a DSMC for MECHATRONICUML's Real-Time Coordination Protocols (RTCPs) using the model checker UPPAAL. The inputs of the DSMC are the RTCP and MTCTL properties. The output is the verification result per property and—if available—a (counterexample) trace at the level of MECHATRONICUML. Our DSMC is fully automatic and completely hides the model checking from the software engineer.
- In an evaluation, we show for 16 RTCPs that the DSMC's translation to UPPAAL and the back-translation to MECHATRONICUML both work correctly and preserve the semantics. Moreover, we show that our complete DSMC requires for these RTCPs in most cases less than 1 minute and that our DSL's encapsulate domain-specific aspects more effectively than UPPAAL, which results in an efficient specification of coordination protocols and verification properties.

5.2 Stakeholder and Requirements

In this section, we define the stakeholders for a DSMC in general. Then, we formulate and justify the requirements to enable a DSMC for a given DSL and DSVPL as well as a given model checker. In our case, the DSL enables to specify RTCPs, the DSVPL is MTCTL, and the model checker in use is UPPAAL.

We distinguish between two groups of stakeholders. The first group—the *DSMC users*—consists of software engineers that apply the DSMC. Thus, they must be able to understand the DSL model and the verification properties that are specified via the DSVPL. Moreover, they should be able to understand the configurations for the model checking that we provide them. In addition, they need to be able to understand the counterexample of the DSMC. As a consequence, we assume that this group has significant experience about the domain and therefore about the DSL, and the DSVPL. However, they only need basic experience in model-driven software engineering (e.g., specifying model transformations) and model checking. For example, they do not need to know how to specify domain-specific aspects like asynchronous message exchange on the model checker level.

The second group—the *DSMC developers*—are software engineers that define and maintain the DSMC. Thus, we assume that they are experts in model-driven software engineering (especially model transformation) as well as of the DSL, the DSVPL, and the model checker in use. For example, they need to know how to specify domain-specific aspects on the model checker level.

In the following, we present requirements a DSMC shall have at least to hide the model checking details (i.e., the model checker's input language, its verification property language, its results output, and its counterexamples) to the DSMC user. We order the requirements according to their priority, where the first is the most important requirement. These requirements and their priorities are based on our experience. Therefore, we do not claim that this is a complete list.

In our opinion, a DSMC shall . . .

R1 provide concepts to translate the DSL into the model checker's input language,

- R2** provide concepts to translate the DSVPL properties into the model checkers' verification property language,
- R3** support to automate the model checker,
- R4** provide concepts to back-translate the result and the traces to the level of the DSL,
- R5** correctly translate the DSL and the DSVPL into the model checker's input language and its verification property language,
- R6** correctly translate the results and the traces (if available) back to the level of the DSL,
- R7** support the complete expressiveness of the DSL,
- R8** support the complete expressiveness of the DSVPL, and
- R9** enable to choose the relevant DSMC options by the DSMC user.

In the following, we will justify our requirements. Requirement R1, R2, R3, and R4 are necessary to successfully hide the verification models to the DSMC user. For example, a DSMC for coordination protocols of CPSs like MECHATRONICUML's RTCPs has to support concepts like real-time behavior and asynchronous message exchange. Otherwise, the DSMC user has to manually change the model checker's input models, which is an error-prone task.

Requirement R5 and R6 are essential for having trust in the DSMC. Otherwise, the DSMC user cannot be sure whether the DSMC's output is correct and, therefore, whether the system is safe w.r.t. this aspect.

If R7 or R8 is violated, the DSMC user either cannot apply model checking to all its models or cannot validate all verification properties. Both consequences can lead to unsafe systems. As a workaround, the DSMC user could try to adapt the inputs for the model checker after the translation. However, this is not only time-consuming but also error-prone because we stated the typical DSMC user does not necessarily has experience in model transformation or model checking.

Finally, as stated in requirement R9, the DSMC user should be able to set relevant DSMC options to adapt the translation, the model checking itself, or the back-translation of counterexamples. These options can influence several factors, e.g., the instantiation of the coordination protocol, the duration of the model checking, whether the model checker can terminate with a result, and the usefulness of the counterexample. For example, for a one-to-many RTCP, the DSMC user should be able to define the role multiplicity. As another example, if several counterexample traces exist, the DSMC user could be able to choose whether the trace that the model checker returns shall be as short as possible (i.e., the number of snapshots that the trace contains is as minimal as possible) or as fast as possible (i.e., the global time that passed is as minimal as possible). Consequently, if these options are not available, then the execution of the DSMC would be too restricted or even impossible. This would require that the DSMC user has to make changes to the model checker's input model or that he has to call the model checking manually. Therefore, the DSMC would not be hidden anymore.

5.3 DMSC for CPS by Utilizing Model-To-Model Traceability

“This section presents a DSMC approach that is suitable for the domain of [CPSs] as it enables complex scenarios with larger differences between DSL and model checker.” [*DGH15] We explain our approach independently of MECHATRONICUML and UPPAAL such that other DSMC developers may reuse our approach for their DSL and their model checker. We show the applicability of our approach by presenting our DSMC for MECHATRONICUML’s RTCPs using UPPAAL in the Sections 5.4, sec:dsmc:uppaal-level, and sec:dsmc:backward.

“We depict our approach in Figure 5.1. Similar to already existing DSMC approaches, e.g., [GdLMD09; BBK+15; SAB10], we apply model transformation techniques [SK03]. However, in contrast to the aforementioned DSMC approaches, we ease the task to overcome larger differences between the DSL and the input language of the model checker by allowing a multi-step translation (see [EABP12]) but still only require a single-step back-translation [(one step represents one model-to-model transformation rule)]. We achieve this by utilizing M2M traceability, i.e., we utilize traceability links that relate the models with each other and, therefore, indicate the semantic correspondence between the input/output elements of each forward translation step. In fact, transformation languages such as QVTo [QVT; QVTo] have a built-in traceability mechanism. Therefore, they do not require additional development effort in order to obtain these links while executing a M2M transformation.

Our approach starts in Activity 1 by applying the M2M translation for translating the *design model* given in terms of the DSL into the *verification model* that is based on the input language of the model checker. This translation may consist of multiple steps, e.g., our translation of RTCPs to the input language of the UPPAAL model checker consists of 13 steps. In each forward translation step, the aforementioned traceability links are automatically generated as an additional output.” [*DGH15] As stated before, we do not have to generate nor store the traceability links on our own as QVTo already provides these functionalities.

“Activities 2–4 use state-of-the-art concepts: since existing model checkers usually operate on a textual input language, Activity 2 includes a model-to-text transformation of the verification model. For example, UPPAAL requires a serialization into an XML-based file format. Afterwards, the actual model checking takes place in Activity 3. Counterexamples generated during model checking are usually given in terms of a textual format as well. Therefore, Activity 4 includes a text-to-model transformation. The resulting counterexample model is a sequence of snapshots. Each snapshot provides a runtime view of the verification model. Therefore, as depicted in Figure 5.1, the parsed counterexample model includes cross-references to the verification model. Among others, the DSMC for RTCPs uses these cross-references to indicate which states of the UPPAAL timed automata are active in each particular snapshot.” [*DGH15] We define all cross-references by executing a name-matching with the verification model, i.e., we take each model element that is given in the textual counterexample and search its correspondent model element in the verification model.

“The back-translation in Activity 5 receives the counterexample model and translates it back to the DSL level. During the back-translation, all cross-references to elements of the verification model need to be replaced by corresponding cross-references to design model elements. To resolve this semantic correspondence, the back-translation also receives the traceability links generated during Activity 1 as an input. Since related approaches address only single-step translations [GdLMD09; BBK+15; SAB10], they are restricted to direct traceability links between design model and verification model. In contrast, our approach

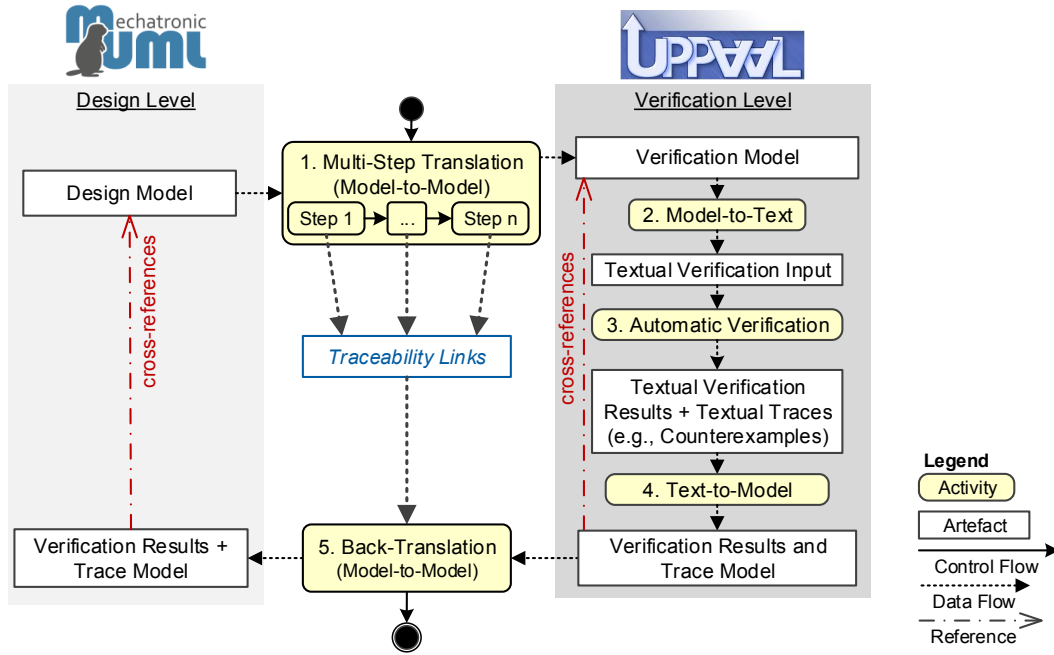


Figure 5.1: Utilizing M2M Traceability for DSMC (Based on [*DGH15])

considers paths of traceability links of arbitrary length, which traverse the intermediate models generated during the multi-step translations.

A particular benefit of our approach is that the back-translation does not need to depend on any conceptual or implementation details of the prior forward translation. It only depends on the [metamodels] of the DSL and the model checker. Therefore our approach is independent of the concrete number of forward translation steps, i.e., it enables an easy integration of additional steps without changing the back-translation.

Figure 5.2 illustrates the analysis of traceability paths in the context of a forward translation consisting of two steps. In this [simple] example, a state machine inside the design model (consisting of two states) is translated to a verification model with two parallel state machines, and an overall amount of five states. The traceability links (depicted in blue) connect corresponding elements inside the design model, the intermediate model, and the verification model. The model checker's counterexample model (depicted in the bottom right corner of Figure 5.2) uses cross-references to indicate which states of the two state machines inside the verification model are active during each snapshot. For example, Figure 5.2 highlights a single model checker snapshot in red, which marks two states as active (one per state machine). In order to translate such a snapshot back to the DSL, the back-translation analyzes the traceability links generated in Activity 1. For example, for the two states marked active, it resolves the traceability path in the opposite direction, until [it reaches the] states contained by the original design model. These states are semantically equivalent to the states marked active by the snapshot. In our example, both paths consist of two traceability links. However, we do not need two separate back-translations steps. Instead, we use a generic implementation of the back-translation that is independent of the number of traceability links that form the path. We realize it by resolving the traceability links incrementally until we reach an element of the

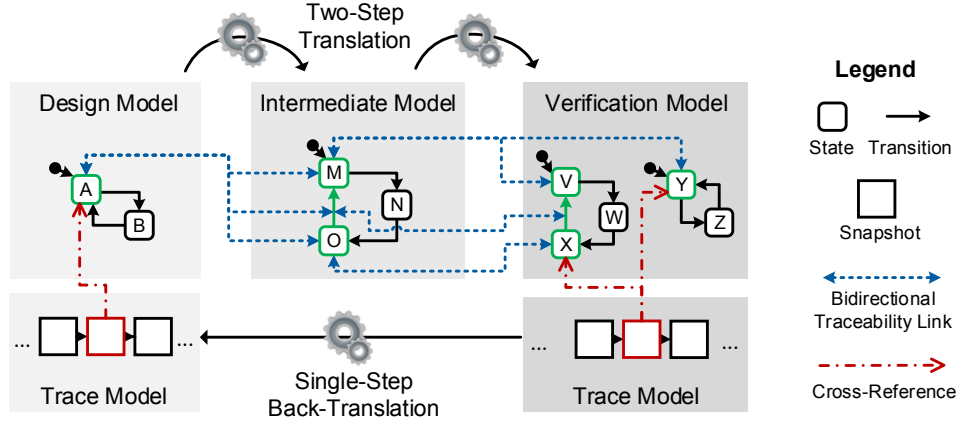


Figure 5.2: Resolving of Traceability Paths (Based on [*DGH15])

design model. Thus, one single back-translation step [(i.e., one model-to-model transformation rule)] is always sufficient to resolve traceability paths of arbitrary length.

“In our example, for the snapshot depicted in red, the traceability paths indicate that both active states inside the verification model evolved from one and the same state inside the design model. Thus, the back-translation generates a DSL snapshot that includes a cross-reference to mark this state active. The back-translation can therefore provide a DSL-specific runtime view of the original design model.” [*DGH15]

Noteworthy, we can traverse the traceability paths in both directions: (1) we can start with one element in the design model and gather all corresponding elements in the verification model and (2) we can start with one element in the verification model and gather the corresponding element in the design model. Depending on the back-translation task, one of the two strategies or a combination of both might be appropriate. For the example in Figure 5.2, the task is to identify which states of the design model are currently active. Strategy (1) would require that we gather all corresponding verification elements of the two states and check whether one of them is currently active. When applying strategy (2), we would only resolve the elements of the verification model that are currently active, which would result in the one corresponding state of the design model. Therefore, strategy (2) is in this case more efficient.

“Our approach is limited to [one-to-many] traceability links between two models. Thus, every step may translate an input element to more than one output element. In order to avoid ambiguous resolving results, it must not be the case that two or more input elements are translated to the same output element. However, the translation from a DSL to the input language of a model checker is usually a refinement, i.e., it increases the number of model elements (see Figure 5.2). Therefore, this aspect does not impose any practical restrictions.” [*DGH15]

5.4 Model-To-Model Translation from RTCPs to Uppaal

“According to our concept in Section 5.3, we develop in Activity 1 (see Figure 5.1) the model-to-model translation from our design model (i.e., RTCPs including the MTCTL properties) to our verification model (i.e., UPPAAL timed automata including UTCTL properties).

As a prerequisite, we need an EMF-based [metamodel] [EMF] for the UPPAAL model checker. In his PhD thesis, Greenyer [Gre11] defined a [metamodel] for UPPAAL TIGA, a variant of UPPAAL. Thus, we migrate this [metamodel] to UPPAAL and enrich it with additional classes, references, and attributes such that an UPPAAL model has less complex strings. Moreover, we additionally define OCL [OCL] constraints that can check if the static semantics of a model are correct.” [*DGH15] We present our metamodel for timed automata in Appendix E.3.3 and our metamodel for UTCTL properties in Appendix E.3.4.

“Next, we analyze the differences between RTCs and UPPAAL, separate them in different concerns, and define a translation step for each concern. As a result, our forward translation consists of 13 steps in total (see Figure 5.3). The first step transforms the RTC into a Component Instance Configuration (CIC[, see sections 2.2.1 and 3.6]) to unify the translation of protocols and components.” [*DGH15] “Then, we execute three MTCTL-specific translations steps.” [*DGH15] “Afterwards, we unify all time units [that the RTC uses] into one common time unit [as UPPAAL does not support time units at all].” [*DGH15] “Next, we rename all identifiers that are not compliant with UPPAAL. Then, six steps [mainly] focus on RTSC-specific translations. The last step migrates all remaining MECHATRONIC-UML-specific language constructs and has the UPPAAL model as result.” [*DGH15]

Concerning one-to-many RTCs, we assume that the multi role RTSC does not use one-to-many communication schemata but distinguishes between a coordinator region and an explicit subrole region, i.e., we assume that this RTSC is already normalized (see Section 3.5.2.7). This normalization may be executed automatically when using our concept from Section 3.5.2.8. Though, we currently do not automatically normalize the MTCTL properties as well.

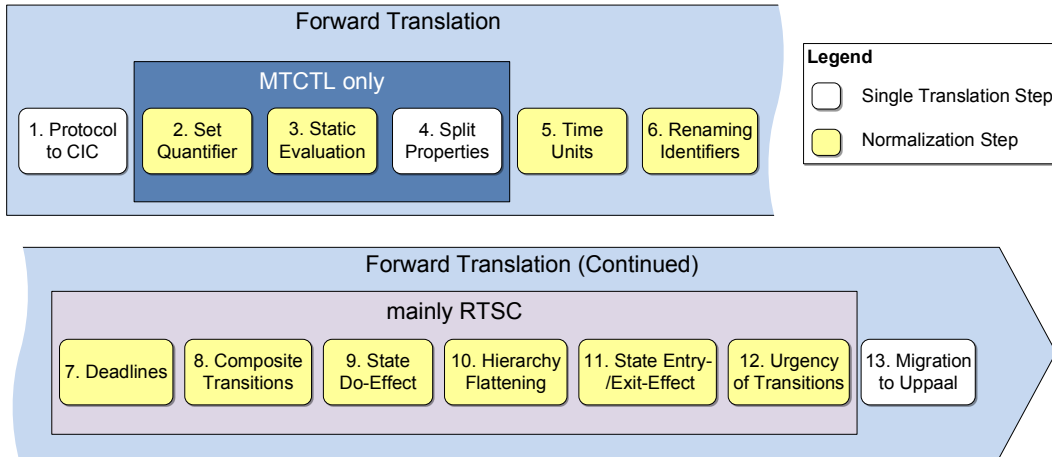


Figure 5.3: Concept for Forward Translation Chain (Based on [*DGH15])

“The first 12 steps are endogenous transformations, i.e., they remain at the level of MECHATRONICUML. Therefore, a valid MECHATRONICUML model has to be the result after each step. In particular, we check after each step [automatically whether] the result is valid w.r.t. the MECHATRONICUML [metamodel] by calling EMF’s validate function on our model [, which checks all OCL constraints as well]. The time effort for doing this is only minimal and is—in our opinion—worthwhile as it raises the trust in the correctness of the forward translation.

In addition, 10 of these 12 steps of our forward translation are normalizations, i.e., they remove [model] elements that MECHATRONICUML has compared to UPPAAL, e.g., [hierarchical states]. As a consequence, we ease the last step that migrates the MECHATRONICUML model to UPPAAL (including domain-specific constructs like the asynchronous message exchange).” [*DGH15]

The inputs of our DSMC are not only the RTCP and a set of MTCTL properties but also a set of DSMC options. Via these options, the DSMC user may configure the transformation as well as the UPPAAL model checking process. We support the following options: (1) The DSMC user may influence UPPAAL’s state space representation by altering the hash table size, which is by default 27 MB. This option can give a speedup especially for large systems. (2) The DSMC user may influence UPPAAL’s state space reduction by selecting one of the three strategies that UPPAAL supports (no reduction, conservative, aggressive). No reduction is beneficial if the DSMC user wants to decrease the model checking time. However, this strategy uses most memory. In contrary, the aggressive strategy is beneficial if the model checking shall use as little memory as possible. Though, this option has the slowest verification time. (3) The DSMC can select whether UPPAAL shall produce a (counterexample) trace (if it exists) or not. Furthermore, the DSMC user may select whether UPPAAL shall produce the shortest trace (i.e., with as little snapshots transitions as possible), the fastest trace (i.e., with as little as possible time progress), or any trace (this should typically result in the shortest model checking time). (4) “[T]he [DSMC user] has to define the size of the connector buffer, i.e., how much messages are at most in transit (we explain [the necessity for this option] in Section 5.4.13.3).” [*DGH15] (5) “[I]f the RTCP is of kind one-to-many, the [DSMC user] has to select the role multiplicity. This is necessary, as verifying a one-to-many RTCP for all possible multiplicities within one verification run would combine all instances within one state space. This can easily result in a state space explosion, which is unfeasible to verify. This is especially the case if the lower limit and the upper limit of the multi role’s multiplicity differ a lot. For example, a lower limit of 1 and an upper limit of 10 leads to ten different instances of the same RTCP. By choosing one multiplicity, we can avoid this problem.” [*DGH15]

In the following sections, we explain each step of our forward translation informally (we formalized all steps using QVTo [QVT; QVTo], see Section 4.6). Some of the steps are based on the concepts of Giese, Burmester, and Hirsch [GB03; Hir04; BGHS04] as well as of David et al. [DMY02]. We explicitly state if this is the case and compare our approach with their concepts.

Noteworthy, we currently do not support the whole expressiveness of RTCPs and RTSCs, e.g., we do not support RTCPs with final states. We list our limitation in Section 5.9.

5.4.1 Protocol to CIC Translation

“The task of the first translation step is to transform the RTCP into a component instance configuration (CIC). ‘A [CIC] is a design-time specification of a concrete instantiated (sub-) system under construction’ [*DPP+16].

The motivation for this is that we want to cover both use cases in MECHATRONICUML for [a] formal verification via UPPAAL (as defined by the compositional verification approach [; see Section 2.2.1]): verifying RTCPs and verifying discrete components¹. In particular, roles

¹Noteworthy, we only provide the fundamentals to formally verify a discrete component. In fact, their verification is not the focus of this thesis.

of an RTCP are proxies for ports of a component because an RTCP describes the coordination between two or more component instances. Therefore, if we transform an RTCP into a CIC, we can use all subsequent steps for translating not only RTCPs but also discrete components. This avoids maintaining similar translation steps.

Figure 5.4 illustrates our concept for translating a one-to-many RTCP into a CIC using a one-to-many variant of our Overtaking protocol named MultiOvertaking, where one car may want to overtake up to [two] cars. At first, we have to create component types. Therefore, for the multi role (here: Overtaker), we create one proxy component with one discrete multi port (here: OvertakerComp). For the single role (here: Overtakee), we create one proxy component with one discrete single port (here: OvertakeeComp). Afterwards, we assign to both ports their corresponding roles. As a side effect, each port gets access to the QoS assumptions of the role connector. Then, we copy the buffer specifications of each role to their respective ports. Afterwards, for each component, we create a component RTSC and contain the role RTSC within the respective component RTSC. Then, we can start to generate the CIC. First, we instantiate the component with the multi port. For a given role multiplicity m (in our case it is 2), we create m sub port instances that are contained in the discrete multi port instance. Second, we instantiate the component with the single role m times. Finally, we connect the sub port instances of the discrete multi port instance via an assembly instance with one of the single port instances. The protocol instance is then automatically derived as it is defined in the MECHATRONICUMLmetamodel.” [*DPP+16].

We define in Section 3.6.2 that an assembly connector instance can derive its QoS assumptions from the role connector of its corresponding RTCP. After this transformation, we retain the RTCP and additionally create the CIC. Therefore, we do not have to define QoS assumptions explicitly at the assembly connector instance.

“We handle the generation of a CIC for a one-to-one RTCP similar compared to generating a CIC for a one-to-many RTCP with multiplicity 1. The only difference is that we do not have to translate a multi role and a single role into a multi port and a single port, but two single roles into two single ports.” [*DGH15]

5.4.2 MTCTL Set Quantifier Normalization

“The task of this normalization is to remove the set quantifiers (**forall** and **exists**) from all MTCTL properties as UPPAAL only supports quantifiers for integers and scalars. In contrast, MTCTL also supports them for states, transitions, messages, and buffers.

We replace the set quantifier **forall** by duplicating and connecting the expressions that belong to the quantifier via conjunctions. Analogously, we replace **exists** by duplicating and connecting the expressions via disjunctions. In both cases, we substitute the variables bound by the quantifier with concrete values.” [*DGH15] This transformation is only possible as each set that MTCTL supports knows its elements at compile time and as each set always refers to a finite number of elements (see Section 4.3.7.2). As a consequence, the normalized MTCTL property is always finite as well.

“Notable, it is not important [whether] the set quantifier is defined outside or inside a temporal property. Moreover, the result is a set of valid MTCTL expression according to the MTCTL [metamodel].

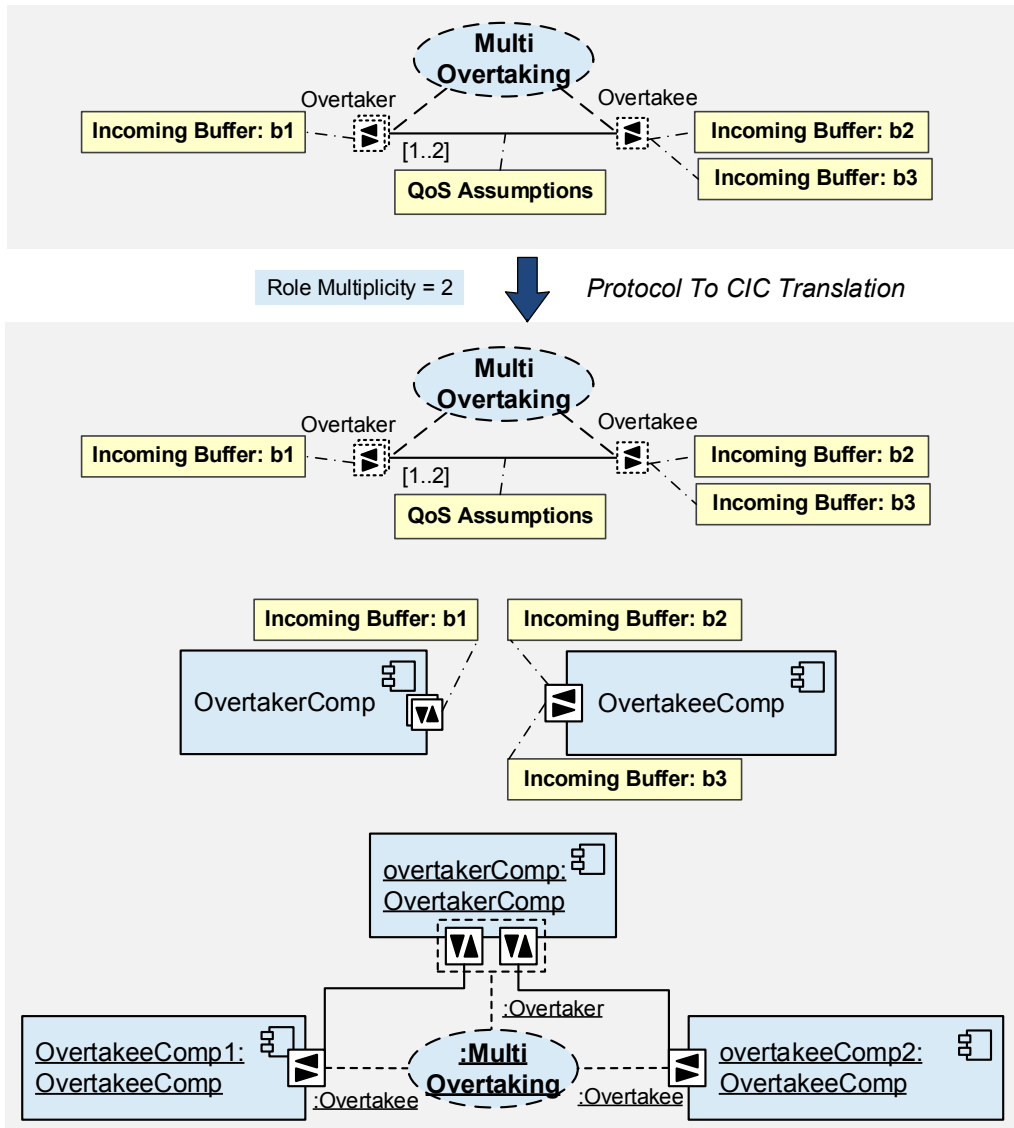


Figure 5.4: Generating a CIC for a One-To-Many RTCPs (Based on [*DGH15])

In the following, we present two examples for RTCP Overtaking. The first expression is `forall(m : MessageTypes) EF messageInTransit (m);`. It uses a forall set quantifier outside of a temporal property. We translate it to:

```
EF messageInTransit (accept) and EF messageInTransit (decline) and
EF messageInTransit (request) and EF messageInTransit (laneChanged) and
EF messageInTransit (finished);
```

The second expression is `EF exists(m : MessageTypes) messageInTransit (m);`. Thus, it uses an exists set quantifier inside a temporal expression. It is translated to:

```
EF (messageInTransit (accept) or messageInTransit (decline) or
    messageInTransit (request) or messageInTransit (laneChanged) or
    messageInTransit (finished));
```

” [*DGH15]

Obviously, this normalization is no longer valid as soon as MTCTL additionally supports sets, where the elements may change at runtime (e.g., the set of messages that arrived in the last 5 seconds) or where the number of elements may be infinite or unknown.

5.4.3 MTCTL Static Evaluation Normalization

“The task of this normalization is to simplify MTCTL expressions using a static evaluation. In particular, this normalization simplifies expressions that are independent of a particular snapshot. This includes static predicates (`substateOf`, `stateInStatechart`) and static operations (`sourceState`, `targetState`), which we normalize first. The simplification is now possible as the “MTCTL Set Quantifier Normalization” replaced all set quantifier variables by explicit model elements (see Section 5.4.2).

The advantage of this normalization is that the model checker will get easier properties to verify. This may lead to shortened model checking time. In the best case, we avoid the model checking of a property at all if we can simplify the complete property to `true` or `false`.

In the following, we present the predicates, operations, and expressions that we search within the MTCTL properties. If one of the rules apply, we simplify the property according to the rules stated below. Note that we do not enforce an order of the rules. We repeat this until no rule applies anymore. This task terminates as the MTCTL properties are finite and as we do not generate additional comparison expressions within this step but only simplify existing ones.

Static Predicates

- The expression contains the static predicate `substateOf(state, superstate)`: If the given state is a (direct or indirect) substate of the given superstate, then we simplify the expression to `true`. Otherwise, we simplify it to `false`.
- The expression contains the static predicate `stateInStatechart(state, statemachine)`: If the given state is (directly or indirectly) contained in the given state machine, then we simplify the expression to `true`. Otherwise, we simplify it to `false`.

Operations

- The expression contains the operation `sourceState(transition)`: We simplify this expression by replacing it with the source state of this transition.

- The expression contains the operation `targetState(transition)`: We simplify this expression by replacing it with the target state of this transition.

Comparison Expressions

- The comparison operator is `==` and the comparables are neither dynamic predicates nor dynamic operations: If the two comparables are identical, then we simplify the expression to `true`. Otherwise, we simplify it to `false`.
- The comparison operator is `!=` and the comparables are neither dynamic predicates nor dynamic operations: If the two comparables are identical, then we simplify the expression to `false`. Otherwise, we simplify it to `true`.
- The comparison operator is `<`, `<=`, `>`, or `>=` and the comparables are integers: We evaluate the integer comparison and simplify the expression to `true` or `false`.

Boolean Logic expressions

- If the expression is a “not expression” and it contains a truth value, then we simplify the expression according to the standard rules of boolean logics.
- If the expression is an “and expression” or an “or expression” and the left hand side and right hand side are truth values, then we simplify the expression according to the standard rules of boolean logics.
- If the expression is an “implies expression” and the left hand side or the right hand side is a truth value, then we simplify the expression according to the standard rules of boolean logics.

Note that we can simplify all occurrences of the static predicates `substateOf(state, superstate)` and `stateInStatechart(state, statemachine)` as all occurrences have now concrete parameter values due to the MTCTL Set Quantifier Normalization and as these predicates do not depend on a specific snapshot. After this normalization, no MTCTL property contains the predicate `substateOf` or the predicate `stateInStatechart`. Furthermore, the resulting MTCTL properties are as simplified as possible concerning their comparison and boolean logic expression.

As an example, consider the user-defined MTCTL property `forall (s:States) (stateInStatechart(s, Overtaker.NoOvertaking) implies EF stateActive(s))`. It informally states the following: For all states `s` in `Overtaking.NoOvertaking` it can possibly happen that `s` is active. In its original form, we cannot simplify the expression. However, after the MTCTL Set Quantifier Normalization, the expression is transformed into an expression that references all 12 states of both RTSCs explicitly:

```
(stateInStatechart(Overtaker.NoOvertaking.Idle, Overtaker.NoOvertaking) implies EF
stateActive(Overtaker.NoOvertaking.Idle))
and (stateInStatechart(Overtaker.NoOvertaking.WaitingForAnswer, Overtaker.
NoOvertaking) implies EF stateActive(Overtaker.NoOvertaking.WaitingForAnswer))
... // 9 more sub-expressions (one per state)
and (stateInStatechart(Overtakee.NoAcceleration.LaneChanged, Overtakee.
overtakee_Role) implies EF stateActive(Overtakee.NoAcceleration.LaneChanged));
```

Therefore, we can now simplify the static [predicate] `stateInStatechart`. The result is:

```
(true implies EF stateActive(Overtaker.NoOvertaking.Idle))
and (true implies EF stateActive(Overtaker.NoOvertaking.WaitingForAnswer))
... // 9 more sub-expressions (one per state)
and (false implies EF stateActive(Overtakee.NoAcceleration.LaneChanged));
```

Afterwards, we can simplify all “implies expressions”. The result is:

```
EF stateActive(Overtaker.NoOvertaking.Idle)
and EF stateActive(Overtaker.NoOvertaking.WaitingForAnswer)
and true and true and true and true and true
and true and true and true and true and true;
```

Finally, we can get rid of all remaining truth values within this conjunction. As a result, we get the following fully statically simplified expression: `EF stateActive(Overtaker.NoOvertaking.Idle) and EF stateActive(Overtaker.NoOvertaking.WaitingForAnswer);`.

5.4.4 MTCTL Split Properties Translation

“The task of this translation step is to split MTCTL properties that consist of more than one temporal quantifier into several properties that consist of at most one temporal quantifier. This is necessary because UPPAAL only enables to verify properties with exactly one temporal quantifier [... (see Section 2.1.2)].” [*DGH15] Moreover, as already stated in the foundations of this thesis, the temporal quantifier expression may not be embedded within a first-order logic expression. For example, the expression `not(A[] deadlock)` is an invalid expression in UTCTL as the temporal quantifier expression `A[] deadlock` is embedded into a not expression.

We realize this translation by collecting all (indirectly contained) temporal properties. Then, we add each of the temporal properties to a new list of MTCTL properties. By splitting up an MTCTL property, we remove the information which logical operator (and, or, implies) connected the temporal properties or if the temporal property was embedded into a not expression. Note that MTCTL properties cannot nest a temporal property into another temporal property (see Section 4.3.1). Therefore, we can simply extract the temporal properties from the original MTCTL property.

As an example, consider the following MTCTL property: `not (EF messageInTransit(accept)) or (AG (messageInTransit(decline) and stateActive(Overtaker.NoOvertaking.Init)))`; This property consists of two temporal properties. Therefore, we replace this property by two new MTCTL properties:

- `EF messageInTransit (accept);`
- `AG (messageInTransit(decline) and stateActive(Overtaker.NoOvertaking.Init));`

The result of this translation is a list of MTCTL properties that now fulfill the requirement of UTCTL that exactly one temporal quantifier is used and that the temporal property is not embedded into a first-order logic expression. Though, this new list of properties is not semantically equivalent to the list of properties before the transformation (this is the reason that this translation is no normalization) as the logical operators have been removed. Moreover, the number of MTCTL properties after this transformations is equal or greater than the amount before this transformation.

“Noteworthy, the traceability recording of our model transformation engine automatically “memorizes” which operators were removed [...]. During the back-translation, we

[...] retrieve this knowledge and, therefore, evaluate the original property (see Section 5.6.1).” [*DGH15]

5.4.5 Time Unit Normalization

“UPPAAL does not support time units nor floating numbers. Therefore, we currently accept RTCPs without floating numbers for the model checking only. However, the RTCP may use time units. Therefore, the task of this transformation is to normalize all time values² [such that they have] a common time unit without producing floating numbers [as UPPAAL only supports integers]. In particular, this affects all RTSCs, all MTCTL properties, and the message delay (lower and upper bound) of the connector QoS assumptions.” [*DGH15]

A division of two integers in UPPAAL is defined as an euclidean division that only results the quotient, e.g., in UPPAAL the result of $10 / 3$ is 3. Therefore, we cannot normalize a bigger time unit to a smaller one (e.g., we cannot normalize from minutes to seconds) as this would require an integer division that loses its remainder. As a consequence, our normalization iterates over all time value expressions and selects the minimal time unit [used within the RTCP]. Therefore, we only have to add factors with base 10 and a positive exponent and do not have to apply divisions.

“As an example, consider the RTCP Overtaking from Figure 3.6. The transition from `Overtaker.Overtaking.Init` to `Changed` uses the time unit [milliseconds] while all other clock-based expressions use the time unit seconds. Therefore, we have to convert all other time values to [milliseconds] as well, e.g., we transform the invariant in state `Overtaker.Overtaking` from $c \leq (10) \text{ s}$ to $c \leq (10 * 1000) \text{ ms}$.” [*DGH15]

5.4.6 Renaming Identifiers Normalization

“In MECHATRONICUML, identifiers have no restrictions concerning their name except that they must be of type `EString`. However, in UPPAAL, identifiers must fulfill the following EBNF expression: $[a-zA-Z\_][a-zA-Z0-9\_]*$. Thus, we replace all non-supported characters with their UTF8-Code (as hexadecimal value). Moreover, we add the prefix `[muml_]` before each identifier to avoid conflicts with reserved keywords in UPPAAL.

For example, if the software engineer gives a state the German name `KeinÜberholen`, we rename it to `[muml_Keindcberholen]`.” [*DGH15]

Note that it is possible that a software engineer defines two state names that are at first different but are identical after this normalization. For example, the state names `Kein Überholen` as well as `Keindcberholen` will be normalized to `m_Keindcberholen`. In UPPAAL, model elements must have unique names. We consider this case by adding an ascending index as a suffix to each model element. Though, this does not happen within this normalization but within the migration (see Section 5.4.13).

5.4.7 RTSC Deadline Normalization

“The task of this normalization is to remove absolute and relative deadlines from a transition. This is necessary as UPPAAL does not support deadlines respectively time consuming

²In MECHATRONICUML, a time value consists of an expression and a time unit.

transitions. Our concept is based on the work of Giese and Burmester [GB03] as we consume the time within an intermediate state.

We illustrate our concept in Figure 5.5. As depicted, we replace each transition in an RTSC that has a deadline (relative or absolute) annotated by an intermediate state and two transitions that connect the original states with the intermediate state. Moreover, the transition that has the new state as a source is non-urgent as it will receive clock constraints. In our example, we replace the transition from A to B by an intermediate state AB, an urgent transition from A to AB, and a non-urgent transition from AB to B.

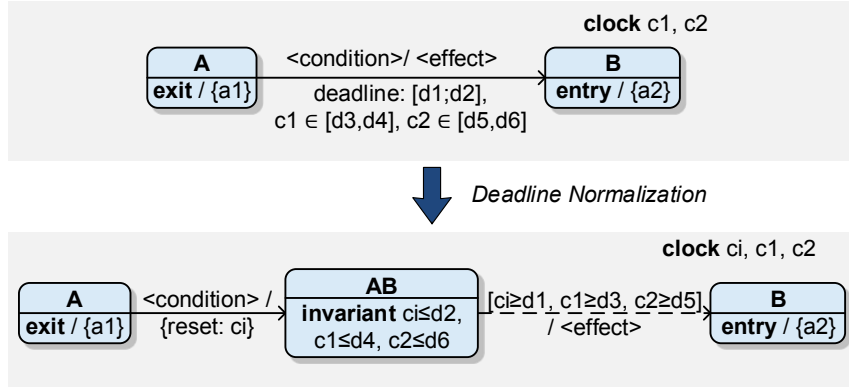


Figure 5.5: RTSC Deadline Normalization (Based on [*DGH15])

Next, we create one clock with a unique name (here: c_i) for all relative deadlines if the RTSC contains at least one relative deadline. This is possible because at most one transition may be active within an RTSC at the same time. We reset c_i while firing the transition that leads to AB. Moreover, we add an invariant to AB with d_1 as the upper limit. Therefore, the RTSC will rest within state AB at most d_1 time units. Afterwards, we add a clock constraint to the transition from AB to B with d_1 as the lower limit to enforce that the RTSC stays at least d_1 time units within AB.

Handling absolute deadlines is similar to relative deadlines. In particular, we add invariants to AB and clock constraints to the transition from AB to B in the same way as we do it for relative deadlines. However, we do not need to create a new clock as the clocks already exist. Moreover, we must not reset the clocks at the transition that leads to the intermediate state.

Next, we move the original transition condition to the transition that leads to the intermediate state. Therefore, we remain consistent that we only leave A if the condition is fulfilled. In contrast to Giese and Burmester [GB03], we do not move the original transition effect to the state AB but to the transition from AB to B. As a consequence, the effect will not become visible by other RTSCs before the firing is done.” [*DGH15] Moreover, if clock resets are defined in the effect of the original transition, then all referenced clocks will be zero after the transition fired. Thus, this will be consistent with clock resets of non-time-consuming transitions. Note that [*DPP+16] currently does not explicitly enforce that the effect may only become visible after the firing. Though, this requirement is planned to be added to future releases of the document.

“As another modification to [GB03], we do not move the exit-action of state A and the entry-action of state B into the intermediate state AB. The reason is that this is a superfluous task because UPPAAL is not able to execute actions within a state but only

at transitions. Moving entry- and exit-actions from a state to a transition is not done within this normalization but within a separate one: the entry-exit-event normalization, which we describe in Section 5.4.11.” [*DGH15]

5.4.7.1 Adapting MTCTL Properties

“The MTCTL predicate `transitionFiring` cannot directly be expressed via UTCTL because UPPAAL’s [transitions] may not be active and may not take time. However, UTCTL is still able to evaluate if a state is active. During [the RTSC Deadline Normalization], we [add] an intermediate state that is active while firing.” [*DGH15] Therefore, instead of referring to a time-consuming transition within an MTCTL property, we can now refer to the intermediate state. “Consequently, we replace all `transitionFiring` predicates that refer to a transition with a deadline by a `stateActive` predicate that refers to the newly introduced intermediate state. For our example in Figure 5.5, the user could have specified the MTCTL property `[EF transitionFiring(A_to_B);]`. After this normalization, the resulting property is `EF stateActive(AB);`” [*DGH15]

After this transformation, we still may have MTCTL properties that use the predicate `transitionFiring`. However, these predicates only refer to non-time-consuming transitions.

5.4.8 RTSC Composite Transition Normalization

“UPPAAL only supports flat timed automata but MECHATRONICUML supports hierarchical RTSCs. Therefore, a flattening of RTSCs must happen. The task of the composite transition normalization is to ease the forthcoming hierarchy flattening (see Section 5.4.10) by replacing all composite transitions with simple transitions.

Semantically, a composite transition is only an abbreviation of multiple simple transitions that have as a source and/or as a target a [substate] of the original transitions. In particular, a composite transition that originates from a composite state may start from all inner states of all regions. Thus, it may fire independently of the active [substate] (as long as its conditions are fulfilled). In contrary, a composite transition that leads to a composite state activates all its [substates] (distributed in several regions) that are marked as initial.

As a consequence, we can replace a composite transition by a graph of simple transitions that are connected via state connection points.” [*DGH15] In particular, if the source of the composite transition is a composite state, then we add transitions and state exit points to connect all inner states of the composite state with the original transitions. If the target of the composite transition is a composite state, then we add transitions and state entry points to connect all initial inner states of the composite state. “Noteworthy, the source and target state of a composite transition may contain composite states as well. Therefore, we have to consider this when deactivating the source state and activating the target state.” [*DGH15] Within our normalization, we only add non-time-consuming transitions and state connection points, which are non-time-consuming as well. Therefore, after our normalization, firing a transition cannot take more time than before. Moreover, even though we have a graph of transformations, the firing is still carried out as one atomic step as it is required by [*DPP+16, p. 32].

The upper part of Figure 5.6 shows an exemplary source model that we have to normalize. In particular, the source model consists of two composite states A1 and B1 that are connected

with a composite transition. “Both, A1 and B1, consist of two regions. In both states, the upper region consists of a composite state (that contains two simple states) and a simple state. Moreover, the lower region of A1 contains one state while the lower region of B1 consists of two states.” [*DGH15]

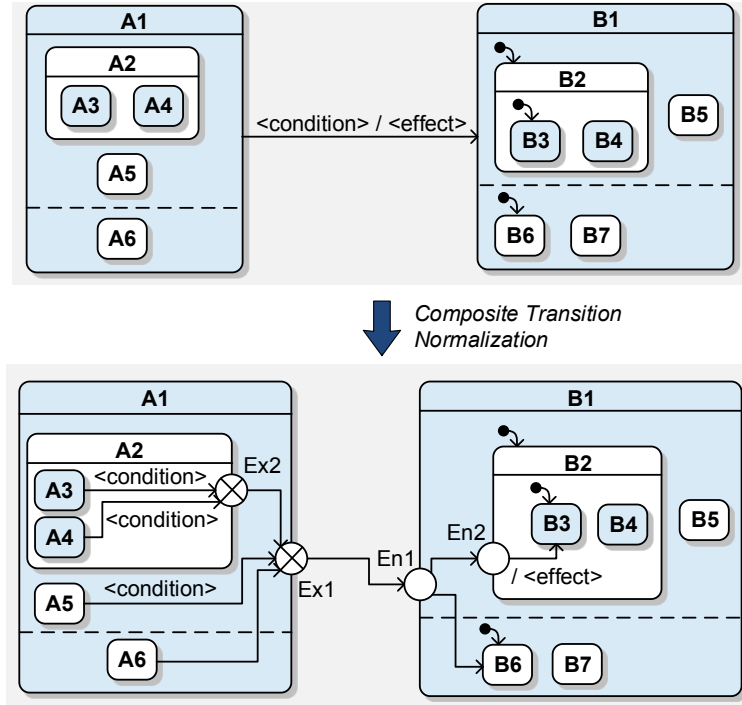


Figure 5.6: RTSC Composite Transition Normalization (Based on [*DGH15])

“Our approach transforms the source and the target of a composite transition independently from each other. Therefore, it is not relevant if the target is a composite or a simple state when transforming an outgoing composite transition (it is the same for the other way round). Because of this, [it is sufficient that] we illustrate our concept for this normalization [using one common example (Figure 5.6) that has a composite state as its source and its target].” [*DGH15]

In the following, we explain our concept for normalizing outgoing composite transitions in Section 5.4.8.1. Afterwards, we explain our concept for normalizing incoming composite transitions in Section 5.4.8.2.

5.4.8.1 Outgoing Composite Transitions

“Our transformation approach for each outgoing composite transition t is as follows: At first, we add an exit point ex to the source state $s1$ of t and change the source of t to ex . In our example, we add the exit point $Ex1$ to state $A1$ and define $Ex1$ as the source of the composite transition.” [*DGH15] Note that an exit point in MECHATRONICUML defines a join semantics: “leaving a state via an exit point is only possible, if all regions provide an enabled transition to the exit point” [*DPP+16, p. 45]

“Then, we add an exit point to all composite states that are (indirectly) contained by s_1 . Afterwards, for each (indirect) simple state s_2 in s_1 , we create a simple transition that has s_2 as its source and the newly created exit point of its composite state as target. Moreover, we create simple transitions that connect the state hierarchy levels within s_1 via the newly created exit points. As a result, we have a tree of simple transitions that have ex as its root and all (indirect) simple states within s_1 as its leaves. Therefore, the former composite transition can start independently of the active substates of s_1 . In our example, we have to add the exit point Ex_2 to the state A_2 . Then, we create five simple transitions: We connect the states A_3 and A_4 with Ex_2 , we connect the states A_5 and A_6 with Ex_1 , and we connect Ex_2 with Ex_1 .

As a final step, we have to move the conditions defined at t because MECHATRONICUML does not allow that conditions are defined at transitions that have a state connection point as its source [*DPP+16, p. 32]. Moreover, MECHATRONICUML defines that a composite state may only be left (1) if the currently active inner states are connected via a transition with this exit point and (2) if all conditions of these transitions are fulfilled. Requirement (1) is always fulfilled as we added a transition from each state to an exit point. Due to (2), it is sufficient that we move the condition of t to all newly created transitions of one region of the composite state s_1 (it is sufficient because the other regions may not be left until this region may be left as well). If this region contains composite states with several regions, it is, again sufficient to select one of the regions. For a deterministic transformation, we always select the first contained region if we have to select one of several regions. In our example, the composite state A_1 has two regions. Thus, we select the upper region as it is the first contained one. Herein, we move the condition of the composite transition to the transition from A_5 to Ex_1 and to the two transitions that have as source A_3 and A_4 and as target Ex_2 .” [*DGH15]

5.4.8.2 Incoming Composite Transitions

“Our transformation approach for each incoming composite transition t is as follows: At first, we add an entry point en to the target state s_1 of t and change the target of t to en . In our example, we add the entry point En_1 to state B_1 and define En_1 as the target of the composite transition.” [*DGH15] Note that an entry point in MECHATRONICUML defines a fork semantics: “a single enabled transition leading to an entry point is sufficient to enable all its outgoing transitions into orthogonal regions [*DPP+16, p. 45].

“Then, we add an entry point to all composite states that are (indirectly) contained by s_1 . Afterwards, for each (indirect) simple state s_2 in s_1 that is marked as initial state, we create a simple transition that has the newly created entry point of its composite state as source and s_2 as its target. Moreover, we create simple transitions that connect the state hierarchy levels within s_1 via the newly created entry points. As a result, we have a tree of simple transitions that have en as its root and all (indirect) simple states within s_1 that are marked as initial as its leaves. Therefore, the composite transition t can initialize all regions of s_1 . In our example, we have to add the entry point En_2 to the state B_2 . Then, we create three simple transitions: We connect En_2 with B_3 , En_1 with B_6 , and En_1 with En_2 .

As a final step, we have to move the effect defined at t because MECHATRONICUML does not allow that conditions are defined at transitions that have a state connection point as its target. Therefore, we have to move effect of t to one of the simple transitions of the transition tree that have a simple state as target. For a deterministic transformation, we always select the first contained region. In our example, the composite state B_1 has two regions. Thus,

we select the upper region as it is the first contained one. Herein, we move the effect of the composite transition to the transition from En2 to B3.” [*DGH15]

5.4.9 RTSC State Do-Effect Normalization

“While MECHATRONICUML supports state do effects within their states, UPPAAL does not support them. Therefore, we have to express them with basic [model] elements. This is the task of this normalization.

Semantically, a state do effect has an action that is executed periodically as long as the state is active. In particular, the action may be executed at any time within the period. Moreover, it is assumed that the WCET of the action is shorter than the period and that it ends before the period ends. In addition, MECHATRONICUML defines that a state must not be left as long as the action is executing.

Giese and Burmester [GB03] define this transformation aspect by adding a non-urgent self-transition to the state. However, their semantics definition differs from the current MECHATRONICUML specification [*DPP+16], e.g., their do action is always executed at the end of each period while we define that it has to be executed within the period without defining at which point in time during the period. Moreover, they do not provide an approach for hierarchical states, i.e., the RTSCs of their transformation to UPPAAL must not contain do actions at hierarchical states.” [*DGH15]

5.4.9.1 Simple States

“Our normalization concept depends on the fact [whether] the do effect is part of a simple or of a composite state. First, we explain the normalization for a simple state s : Instead of using a self-transition, we add a region r to s to define the behavior of the do effect (see Figure 5.7). In particular, we add three states to r : Wait, Execute, and Done. Then, we connect them via urgent and non-urgent transitions as depicted in Figure 5.7.

Afterwards, we move the do effect to the transition between Execute and Done and use the new clock c to define the timing behavior as depicted in Figure 5.7.” [*DGH15] As already stated in Section 2.2.2, MECHATRONICUML does not enable to state a delay (e.g., a worst case executing time) to the do effect. Instead, it only assumes that the worst case execution time of the effect is shorter than the period and that the effect does not have to start at the beginning of the period, but has to end before the end of the period. We fulfill all these assumptions by putting the do effect at the transition between Execute and Done: As the transitions from Wait to Execute is non-urgent, time may pass (but does not have to) before we execute the effect. Moreover, the effect is executed before the period ends (it may only end in state Done).

“Next, we translate each incoming transition t_{in} to s as follows: we add an entry point en to s , define en as the target of t_{in} , create a transition $t1$ from $[en]$ to Wait, and move the effect from t_{in} to $t1$. Moreover, we translate each outgoing transition t_{out} of s as follows: we add an exit point ex to s , define ep as the source of $[t_{out}]$, create a transition $[t1]$ from Wait to ex , create another transition $t2$ from Done to ex , and move the condition from $[t_{out}]$ to both transitions ($[t1]$ and $t2$). As a consequence, if s becomes active, Wait becomes active, too. Moreover, the state s may only be left if Wait or Done are active but not if Execute is active.

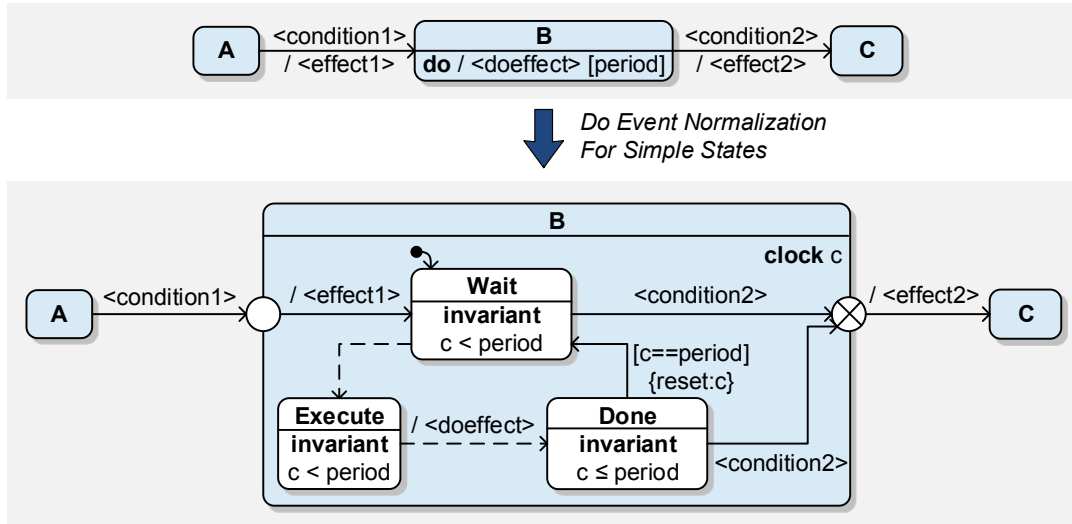


Figure 5.7: RTSC Do-Effect Normalization for Simple States (Based on [*DGH15])

The resulting informal behavior definition of this region is as follows: Initially, Wait is active and a new period begins. It is left before the period ends either by exiting the state *s* or by switching to Execute. If Execute is active, the do effect will eventually be executed (as no other outgoing transition exists). It is assumed that the execution of the effect is done when we switch to Done. Before the period ends, *s* may be left. Otherwise, the region switches to Wait and a new period begins.” [*DGH15]

5.4.9.2 Composite States

“Our concept for do effects at composite states is similar to the concept for simple states. Likewise, we add a region *r* to the state *s* that contains the do effect. Then, we add the same states, transitions, and the clock to *r*.

In contrary to the simple states, we do not have to add a new entry or exit point. The reason for this is that all incoming and outgoing transitions of *s* already have an entry point as target or an exit point as source due to the composite transition normalization. As a consequence, we have to add a transition from each already existing entry point to the state Wait to activate this new region via all possible incoming transitions. Moreover, we have to add transitions from Wait and Done for each already existing exit point to enable the deactivation of the composite state according to the rules of MECHATRONICUML. Compared to the simple state concept, moving effects and conditions is not necessary as all incoming transitions have no effects and all outgoing transitions have no conditions anymore due to the composite state normalization.

We illustrate our concept for composite states in Figure 5.8. The example shows a composite state B1 that contains a substate B2, an entry point En1, and an exit point [Ex1]. The normalization creates a region similar to the one for a simple state (see Figure 5.7). In contrast to the simple state’s region, the transition to Wait has the pre-existing entry point En1 as its source. Moreover, Wait and Done have the pre-existing exit point [Ex1] as their target.” [*DGH15]

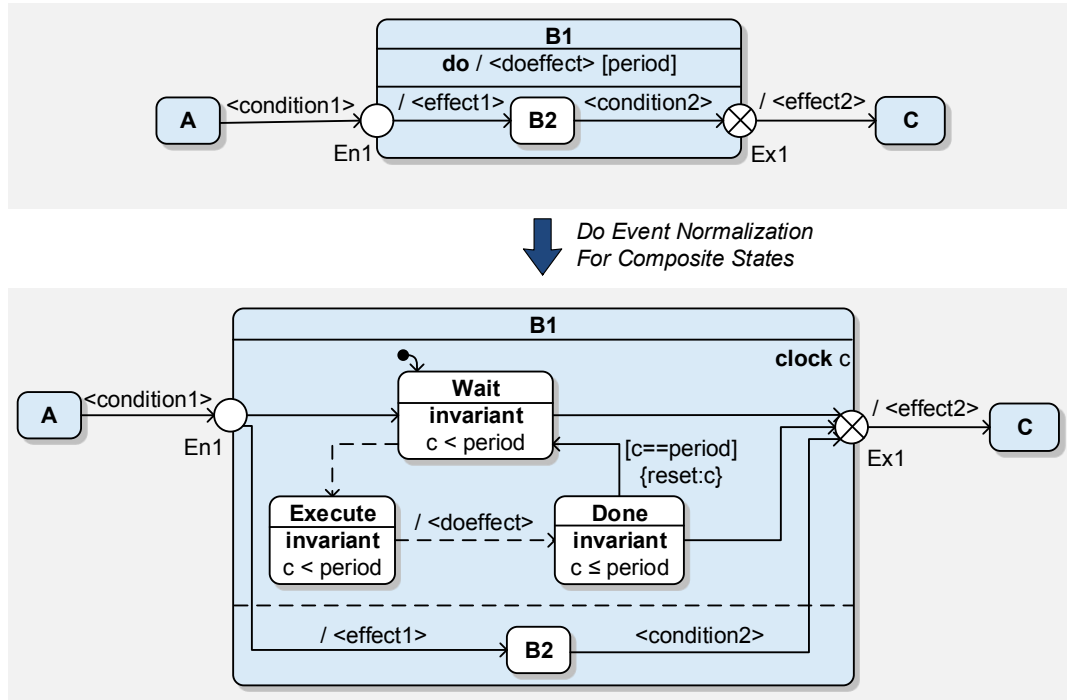


Figure 5.8: State Do-Effect Normalization for Composite States (Based on [*DGH15])

5.4.10 RTSC Hierarchy Normalization

“UPPAAL only supports flat automata. Therefore, we have to normalize all hierarchical RTSCs. Our approach for doing this is influenced by the concept of David et al. [DMY02] that realize the flattening in three [phases].” [*DGH15] In the first phase, they separate the hierarchy levels. Then, in the second phase, they handle the composite state deactivation. After the second phase, it may be possible that a formerly composite transition that starts in the composite state s can now synchronize with a transition that is defined inside s . Though, such a synchronization is a forbidden behavior. Therefore, in the third phase, they resolve this problem by introducing duplicate channels and transitions.

“Compared to David et al., we reduce the complexity of the flattening as we do not have to handle composite transitions because we already normalized them (see Section 5.4.8).” [*DGH15] As a positive side effect, the problem that phase 3 solves cannot occur in our transformation as the input for this normalization does not contain composite transitions anymore. Consequently, we do not have to realize phase 3. “Moreover, the goal of David et al. is to directly produce a valid UPPAAL model. In contrast to them, our goal is that the transformation result is a model that still adheres to the MECHATRONICUML [metamodel] because we want to apply further normalizations at the level of MECHATRONICUML afterwards.

In the following, we explain how we apply the phases 1 and 2 of the approach of David et al.” [*DGH15] We illustrate our concept using the example that we show in the upper part of Figure 5.9.

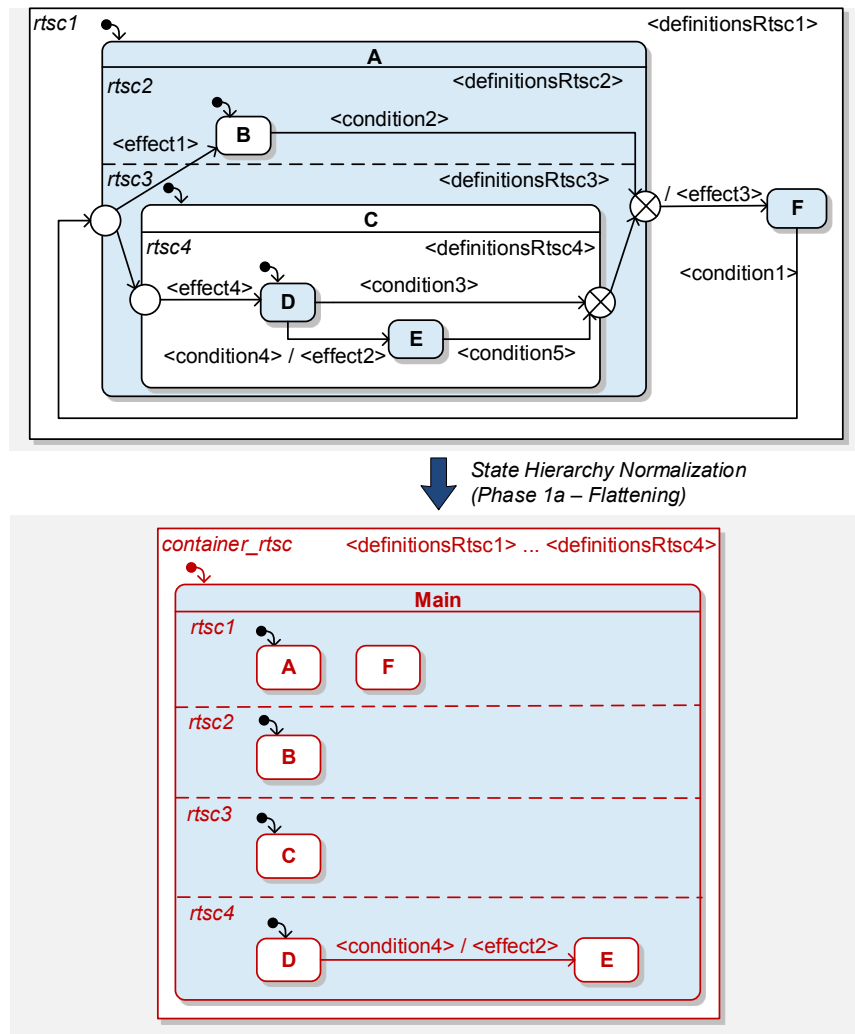


Figure 5.9: RTSC Hierarchy Normalization: Phase 1a (Based on [*DGH15])

5.4.10.1 Phase 1: Separating the Hierarchy Levels

“We split phase 1 into two different categories. First, we explain the flattening itself. Second, we explain how we realize the kickoff of the state machine and the inactivity of an embedded RTSC.” [*DGH15]

Phase 1a: Flattening

According to David et al., the first task is to remove the hierarchy. We realize this by applying the following steps:

1. We create for each (embedded) RTSC of the original RTSC one new RTSC. We take the name of the RTSC but not its definitions (i.e., the definitions of variables, clocks, and operations).
2. We embed all new RTSCs into a new composite state, which we call Main. This state is always active and belongs to a new RTSC, which we call container_rtsc.

3. We move all definitions of variables, clocks, and operations of all RTSCs into the new container_rtsc. This is necessary as MECHATRONICUML defines that embedded RTSCs may access variables, clocks, and operations of their parent RTSCs [*DPP+16, p. 19]. We avoid name clashes (e.g., two clocks have the name *c* and are now defined within the same RTSC) by giving all variables, clocks, and operations a unique name.
4. In each new RTSC, we embed the states of its original RTSC. Though, we transform all composite states to flat states. Therefore, we remove its regions, its channel definition, and its state connection points.
5. In each new RTSC, we embed the transitions of its original RTSC that connect states with each other. Therefore, we do not take transitions that have a state connection point as its source or target.

Note that embedding all flat RTSCs into a new composite state is a significant difference to David et al. We do this as it provides us several advantages:

- In MECHATRONICUML, a role of an RTCP has exactly one RTSC. With our approach, we can link the single RTSC container_rtsc to a role. Therefore, after the hierarchy normalization, the MECHATRONICUML model will still be valid according to the MECHATRONICUML metamodel.
- “In contrast to UPPAAL, MECHATRONICUML cannot store global declarations (e.g., variables, clocks, channels, operations) outside of an RTSC. By introducing the new RTSC, we can define “global” variables, channels, clocks, and operations. Thus, several flat RTSCs can access them, e.g., for specifying dependencies between them (see Phase 2).” [*DGH15]

We illustrate our flattening approach in Figure 5.9 (we highlight all changes in red). First, we create a new RTSC container_rtsc, which contains the new composite state Main. Then, we embed the four existing RTSCs (rtsc1, rtsc2, rtsc3, and rtsc4) into Main as parallel regions. In rtsc1, we embed the states A and F but no transitions as both transitions have a state connection point as source or target. Moreover, we transform A from a composite state to a simple state. In rtsc2, we only embed the state B and no transitions as both transitions have a state connection point as source or target. In rtsc3, we only embed the states C but no transitions as both transitions have a state connection point as source or target. Moreover, we transform C from a composite state to a simple state. In rtsc4, we embed the states D and E and the transition between D and E. However, we do not embed the other three transitions as they have a state connection point as source or target.

Phase 1b: RegionInactive-States

“In general, an embedded RTSC [resp. its region] must be [inactive] (i.e., none of its states or transitions are active) if the parent state is not active. David et al. propose to add a new state called *Idle* [(we prefer the name *RegionInactive*)] to *all* embedded state machines. Moreover, they define that all [*RegionInactive*] states are the initial states of the embedded state machines. However, an embedded state machine may be active from the start, e.g., rtsc2 in Figure 5.9 [is] initially active. As a consequence, David et al. add a dedicated kickoff state machine that enforces the relevant embedded state machines to switch from their [*RegionInactive*] state to the correct initial state.” [*DGH15]

In contrast to David et al., we refrain of using a dedicated kickoff state machine as this would require to fire a set of transitions at runtime to initialize the system. Consequently, the first snapshots of the zone graph are no official snapshots. Therefore, we would require additional concepts to ignore these snapshots when evaluating verification properties.

“Instead, we only add an [RegionInactive] state if the embedded state machine is not initially active or if the embedded state machine may be left while execution. Moreover, we only set the [RegionInactive] state to initial if the original initial state [of this region] is not active when initializing the system. We connect the [RegionInactive] states as follows: First, we collect all transitions of the [original embedded RTSCs] that [have] a state connection point as source or target. Second, we redirect the source [resp.] the target from the state connection point to the [RegionInactive] state.” [*DGH15] In summary, this concept does not produce unofficial snapshots as we identify exactly those states that must be active and define them as initial states. Moreover, compared to David et al., we do not require an additional kickoff RTSC (including states and transitions) and additional channels to realize our concept. Therefore, our concept does not inflate the model even more.

For example, based on Figure 5.9, we execute the following steps: “[W]e add [the state RegionInactive] to rtsc2, rtsc3, and rtsc4 because their former parent state A may become [inactive]. However, the [RegionInactive] states are not set as initial as all these three RTSCs are active at the start. rtsc1 does not get [a RegionInactive] state, as it will always stay active. Finally, we redirect the transitions. As a result, in rtsc2 and rtsc3, the outgoing and the incoming transition of state B and C both have the state [RegionInactive] as its second connector endpoint. Moreover, in rtsc4, the incoming transition of D as well as the two outgoing transitions of D and E that referred to an exit point now have the [RegionInactive] state as their second connector endpoint.” [*DGH15] We show the resulting RTSC after phase 1b in Figure 5.10.

5.4.10.2 Phase 2: Handling Composite State Activation and Deactivation

“In the second phase, we synchronize the hierarchy levels by handling the activation and deactivation of former composite states. Our concept of phase 2 is based on David et al. as we use synchronization channels in order to activate or deactivate composite states. However, as MECHATRONICUML differs to UPPAAL and as we do not have to handle composite transitions, the concrete algorithms differ.” [*DGH15] “[Moreover,] David et al. only handle the composite state deactivation in phase 2. Their activation was part of phase 1. Instead, we move the activation to phase 2 as it is very similar to the handling of the deactivation.” [*DGH15]

Phase 2a: Handling Composite State Activation

“A composite state activation may become complex as each composite state may consist of several regions with additional composite states that again may consist of several regions and so on. Consequently, the activation is like traversing a tree from its root to its leaves.” [*DGH15]

In particular, we proceed as follows for each activation tree that has a state *s* as its root, a composite state *cs* that shall be activated via an entry point *en* (*s* is on the same hierarchy level as *cs*), and a set of states that act as the leaves of the tree and, therefore, shall be activated:

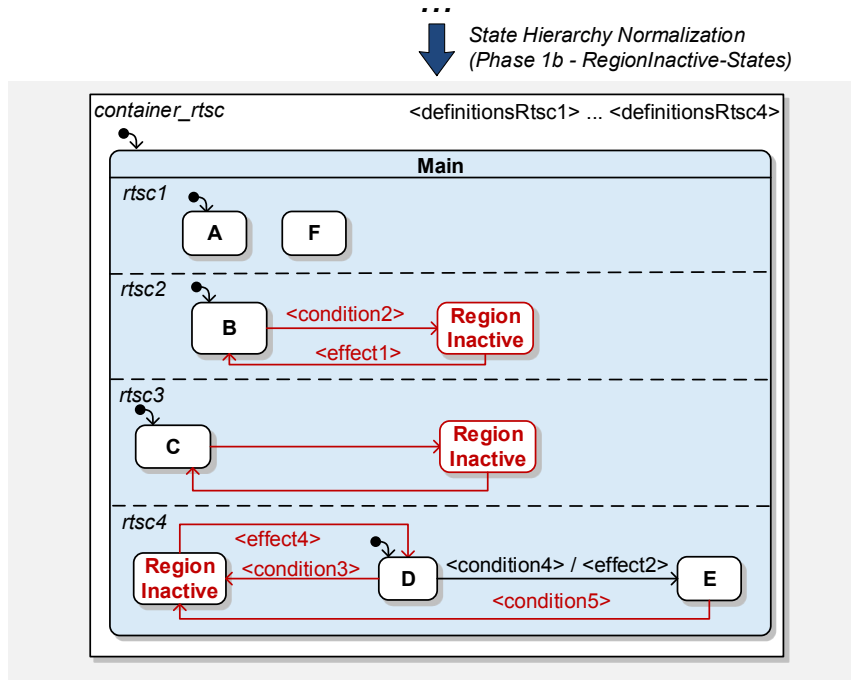


Figure 5.10: RTSC Hierarchy Normalization: Phase 1b

1. “For each region of *cs* [that we may enter via the activation tree], we add one synchronization channel *sc* to the single state of the new RTSC *r* that embeds all flat RTSCs.” [*DGH15]
2. For each region that we enter via the activation tree, we add one committed state to the region of *cs*. Moreover, we connect these states with urgent transitions in a sequence, whereby *s* is the start of the sequence and *cs* is the end of the sequence.
3. We move the condition label and the urgency from the transition that has state *s* as its source within the original RTSC to the first transition of our transition sequence.
4. We add a sender channel (“!”) of type *sc* to each outgoing transition of a committed state of this sequence.
5. We add a receiver channel (“?”) of type *sc* to each transition that shall activate a state due to the activation of *cs* via *en*. Hereby, we have to make sure [that we] respect the order of activation, i.e., we have to activate them from the root to leaves.

Noteworthy, MECHATRONICUML defines that a composite state activation chain is one common step that does not take time and where no effects may be executed [*DPP+16, p. 32]. We follow this definition as we only add committed states and as we only add urgent transitions that define no effects.

An example is given in Figure 5.11. In particular, we have to handle that the state *F* may fire its outgoing transition, which leads to the entry point of state *A*, and, therefore, to the activation of *A*. As *A* contains three regions (*rtsc2*, *rtsc3*, and *rtsc4*), we add three channels *entry1*, *entry2*, and *entry3* as well as three committed states *EntryA1*, *EntryA2*, and *EntryA3* to region *rtsc1*. Then, we connect the state *F* with *EntryA1*, *EntryA1* with *EntryA2*, *EntryA2*

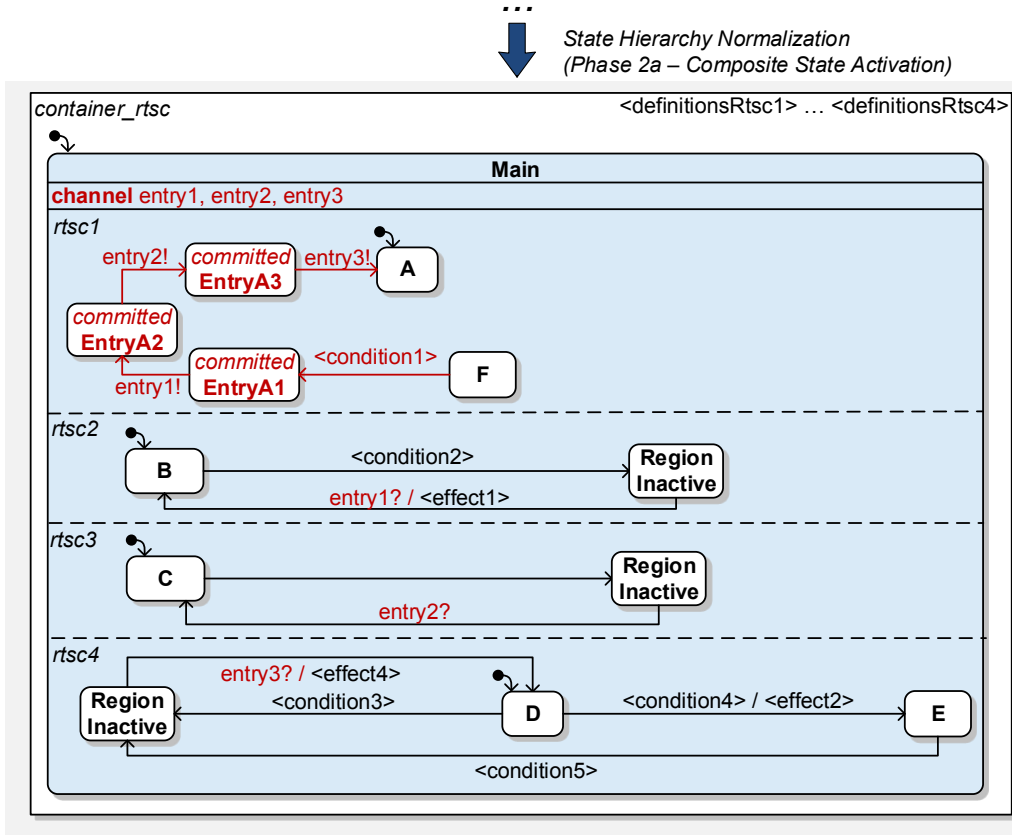


Figure 5.11: RTSC Hierarchy Normalization: Phase 2a

with EntryA3, and EntryA3 with A. The original RTSC defined a condition label $\langle \text{condition1} \rangle$ at the transition from F to the entry point of A. Therefore, we move this condition label to the transition that connects F with EntryA1. Afterwards, we add one sending channel to each succeeding transition, e.g., the transition between EntryA1 and EntryA2 receives the channel entry1!. As a consequence, we can now switch from state F to A while informing each region that it shall activate. Finally, we have to define that the regions are activated in correct order, i.e., the activation of rtsc4 must be activated at last as it has the deepest hierarchy level. Therefore, we add entry1? to the transition from [RegionInactive] to B in rtsc2, entry2? to the transition from [RegionInactive] to C in rtsc3, and entry3? to the transition from [RegionInactive] to D in rtsc4.

Phase 2b: Handling Composite State Deactivation

“The composite state deactivation may become even more complex than the composite state activation because *all* regions must be able to exit the parent state and each region may define its own conditions for being able to exit. Again, we face the fact that each composite state may consist of several regions with additional composite states that again may consist over several regions and so on. Consequently, the deactivation is like traversing a tree from its leaves to its root.” [*DGH15]

In particular, we proceed as follows for each deactivation tree that has a state s as its root, a composite state cs that shall be deactivated via an exit point ex (s is on the same hierarchy level as cs), and a set of states that act as the leaves of the tree and, therefore, shall be deactivated:

1. For each region of cs that we may exit via the deactivation tree, we add one synchronization channel sc to the single state of the new RTSC r that embeds all flat RTSCs.
2. For each region that we exit via the deactivation tree, we add one committed state to the region of cs . Moreover, we connect these states with urgent transitions in a sequence, whereby cs is the start of the sequence and s is the end of the sequence.
3. For each set of states that enable the composite state deactivation, we need one transition from cs to the first committed exit state. This is required, as each set of states may define different conditions in order to start the deactivation. Thus, we add additional transitions if necessary.
4. We add a sender channel (“!”) of type sc to each outgoing transition of a committed state of this sequence.
5. We collect the conditions of all transitions that belong to one set of states that enable the composite state deactivation, put them into a conjunction, and move them to one of the transitions that we defined in step 3.
6. Now we can add a receiver channel (“?”) of type sc to each transition that shall deactivate an RTSC due to the deactivation of cs via this deactivation tree. Hereby, we have to make sure to respect the order of deactivation, i.e., we have to deactivate them from the leaves to the root.
7. Without further modification, undefined behavior would be possible as we moved the conditions to the RTSC of cs in step 3. For example, if a former embedded RTSC of cs has two states $s1$ and $s2$ that enable the exiting under different conditions, then the RTSC may be left if $s1$ is active but the condition of $s2$ is fulfilled and the other way round. Therefore, for each state that enables the deactivation via this deactivation tree, we add a boolean variable that indicates whether the state is currently active. We set the initial value of each variable depending on the fact whether the state is initially active. Then, at each incoming transition of this state, we set its value to true; at each outgoing transition of this state, we set its value to false. Finally, we specify at each transition that we defined in step 3, which states must be active, as an additional condition. As a consequence, undefined behavior is no longer possible.

Noteworthy, MECHATRONICUML defines that a composite state deactivation chain is one common step that does not take time and where no effects may be executed [*DPP+16, p. 32]. We follow this definition as we only add committed states and as we only add urgent transitions that define no effects.

Due to this concept, we do not support RTSCs that have a composite state with an exit point and two or more transitions from different regions that define a synchronization and lead to this exit point. Otherwise, our concept would add more than one synchronization at one transition, which is forbidden in MECHATRONICUML [*DPP+16, p. 42].

“An example is given in Figure 5.12. In particular, we have to handle the deactivation of state A by its exit point, which leads to the state F. As A contains three regions (rtsc2, rtsc3, and rtsc4), we first add the channels exit1, exit2, and exit3.” [*DGH15] Second, due to the

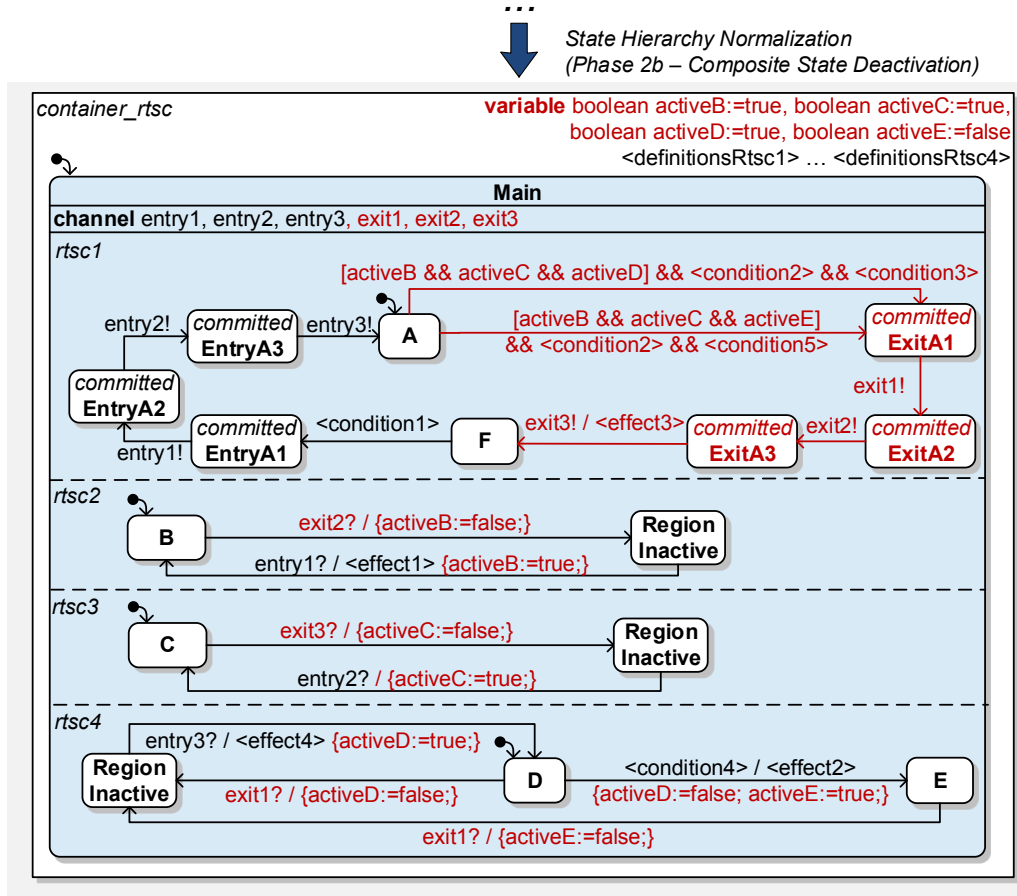


Figure 5.12: RTSC Hierarchy Normalization: Phase 2b (Based on [*DGH15])

three regions, we add three committed states ExitA1, ExitA2, and ExitA3 to region rtsc1 and connect them via transitions: we connect the state F with EntryA1, EntryA1 with EntryA2, EntryA2 with EntryA3, and EntryA3 with A. Third, we have to add a second transition from A to ExitA1 because there exist two sets of states that enable this deactivation: (B, C, D) and (B, C, E). Fourth, we add one sending channel to each outgoing transition of a committed state of the new sequence in rtsc1, e.g., the transition between ExitA1 and ExitA2 receives the channel entry1!. As a consequence, we can now switch from state A to F while informing each region that it shall deactivate. “Fifth, we put the [<condition2>] from transition B to [RegionInactive] in rtsc2 and [<condition3>] from D to [RegionInactive] in rtsc3 in a conjunction and move them to the first transition between A and ExitA1 in rtsc1 [as these are the conditions for the state combination (B, C, D)]. Analogously, we put the condition [<condition2>] from transitions B to [RegionInactive] in rtsc2 and [<condition5>] from E to [RegionInactive] in rtsc3 in a conjunction and move them to the second transition between A and ExitA1 in rtsc1 [as these are the conditions for the state combination (B, C, E)]. Sixth, we add the receiving channels: we add exit1? to the transitions from D to [RegionInactive] and E to [RegionInactive], exit2? to the transition from B to [RegionInactive], and exit3? to the transition from C to [RegionInactive]. Seventh, we add the boolean variables activeB,

activeC, activeD, and activeE to new_rtsc. All variables except activeE are initially true as their corresponding state is also initially true. Then, we add actions to all incoming and outgoing transitions of B, C, D, and E to set the correct value of the variables. Next, we add the condition that the variables activeB, activeC, and activeD must be true to the first transition between A and ExitA1. Analogously, we add the condition that the variables activeB, activeC, and activeE must be true to the second transition between A and ExitA1.” [*DGH15]

5.4.10.3 Adapting MTCTL Properties

“MECHATRONICUML defines that a connection point belongs to the composite state. Therefore, a composite state is still active while the state connection point is being traversed due to a firing transition. Consequently, the predicate `stateActive` of MTCTL is still true if a state connection point is being traversed.

As UPPAAL does not support state connection points, we replace some of the state connection points within this normalization to one or more committed states. Thus, the predicate `stateActive` will no longer return true if it refers to a composite state and such a committed state is being traversed. As a consequence, we have to replace all these `stateActive` predicates by a disjunction of `stateActive` predicates, where one predicate refers to the former composite state and the other predicates to the new committed states.

As an example, consider the MTCTL property `EF stateActive(A);`, which is specified for our example at the top of Figure 5.9. After the normalization, the property is transformed to `[EF ((stateActive(Main.rtsc1.A) or (stateActive(Main.rtsc1.EntryA1) or (stateActive(Main.rtsc1.EntryA2) or (stateActive(Main.rtsc1.EntryA3) or (stateActive(Main.rtsc1.ExitA1) or (stateActive(Main.rtsc1.ExitA2) or (stateActive(Main.rtsc1.ExitA3));].” [*DGH15]`

5.4.11 RTSC State Entry- / Exit-Effect Normalization

“As UPPAAL does not support entry- and exit-effects at states but only transition effects, we have to normalize this aspect, too. This is the task of this normalization.

MECHATRONICUML defines that the entry effect is a short hand notation for adding this effect to all incoming transitions of this state. In particular, the entry effect is executed after the transition effect and before activating the state. As a consequence, we move all entry effects of a state to all incoming transitions (they shall be executed directly after the already existing transition effect).

Analogously, MECHATRONICUML defines that the exit effect is a short hand notation for adding this effect to all outgoing transitions of this state. In particular, the exit effect is executed after the state deactivation but before the transition effect. As a consequence, we move all exit effects of a state to all outgoing transitions (they shall be executed directly before the already existing transition effect).

An example is given in Figure 5.13. Here, the entry effect of state C is moved to the transitions that start from A and B. Moreover, the exit effect of state C is moved to the transitions that start in C and have D and E as target.” [*DGH15]

5.4.12 RTSC Urgency Normalization

“MECHATRONICUML and UPPAAL both support urgent and non-urgent behavior. However, they support it at different [model] elements. While MECHATRONICUML distinguishes

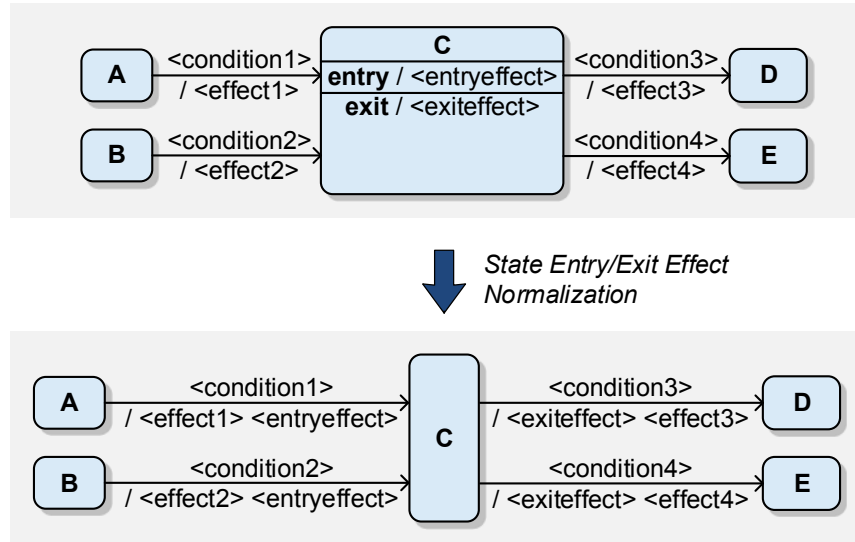


Figure 5.13: RTSC State Entry- / Exit-Effect Normalization (Based on [*DGH15])

between urgent and non-urgent transitions, UPPAAL distinguishes between urgent and non-urgent synchronization channels. As a consequence, in MECHATRONICUML, it is possible to define a synchronization between two transitions where one transition is urgent and the other one is non-urgent [(see Section 2.2.2)]. In UPPAAL, such synchronizations are not possible. Thus, we have to remove such synchronizations in an RTSC if we want to translate it to UPPAAL. This is the task of this normalization.

Our solution for the stated problem is as follows: First, we replace all synchronization channels where a non-urgent transition may synchronize with an urgent transition with two new synchronization channels. They both have the name of the former channel but have a different suffix: one gets the suffix “_urgent”, the other one gets “_nonurgent”. Then, all urgent transitions that use this channel get the new channel with the “urgent” suffix. Analogously, all non-urgent transitions get the new channel with the “nonurgent” suffix. Afterwards, we duplicate all urgent transitions that may synchronize with a non-urgent transition, i.e., we create a new transition *t* with the same source and target and the same conditions and effects. Finally, we change the urgency of *t* to non-urgent and replace the synchronization channel to the non-urgent one. As a result, it holds that all transitions synchronizing over the same channel are either all urgent or all non-urgent. Moreover, the result respects that a pair of transitions, where one is urgent and the other non-urgent, fires non-urgently.

We give an example for this normalization in Figure 5.14. Given is a state **A** that defines a synchronization channel *chan* and that consists of three regions. The regions differ in the urgency of their transition and if they send or receive the channel *chan*.” [*DGH15] Note that MECHATRONICUML only enables a pair-wise synchronization (see Section 2.2.2). Therefore, either region1 and region2 or region1 and region3 may synchronize with each other. If multiple pairs may synchronize with each other, then the region priority decides which pair synchronizes first. If no region priorities are defined, then the pairs may synchronize non-deterministically. “Within the normalization, we replace *chan* by *chan_urgent* and *chan_nonurgent* because the urgent transition of region1 and the non-urgent transition of

region2 can synchronize via this channel. The already existing transition from region1 and the transition of region3 get `chan_urgent`; the transition from region2 gets `chan_nonurgent`. Then, we duplicate the transition from region1 as it is an urgent transition that may synchronize via a non-urgent transition. Finally, for the newly created transition, we replace the `chan_urgent` to `chan_nonurgent` and switch the urgency to non-urgent. Noteworthy, we do not need to duplicate the transition from region3 as it cannot synchronize with the transition from region2 because both are defined as senders (i.e., both have a “!”).” [*DGH15]

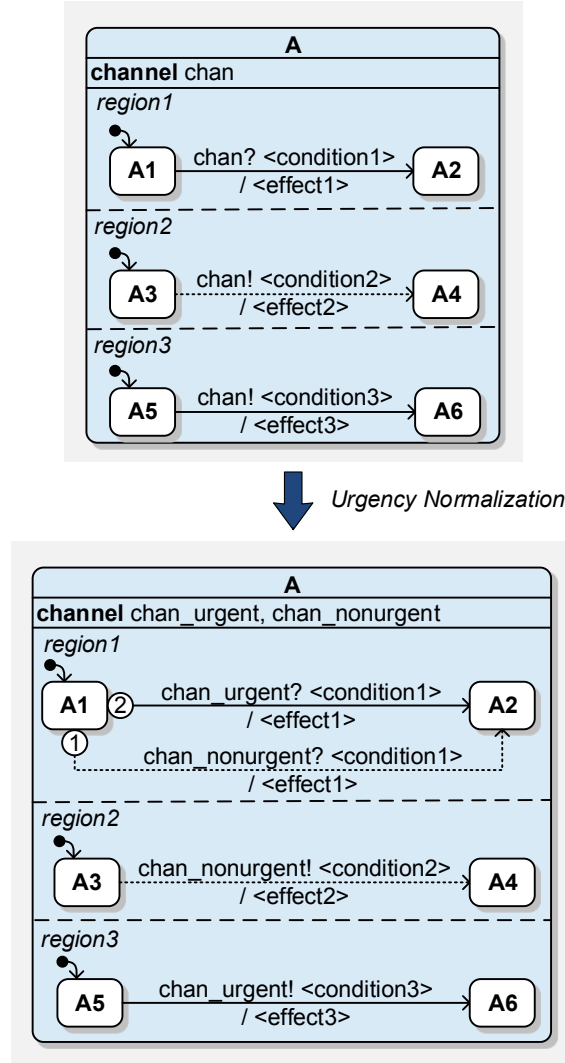


Figure 5.14: Urgent Transition Normalization (Based on [*DGH15])

5.4.12.1 Adapting MTCTL Properties

“[If] we have to duplicate a transition, we have to adapt `transitionFiring` predicates that refer to this transition. In particular, we replace the `transitionFiring` predicate by a disjunction of two

transitionFiring predicates, where one of the predicates refers to the old transition and one to the new one.” [*DGH15]

5.4.13 MechatronicUML to Uppaal Migration

“The task of the transformation is to migrate the MECHATRONICUML model (including the MTCTL properties) into a semantically equivalent UPPAAL model (including UTCTL properties). Due to the previous transformations, this task is significantly less complex as migrating the original MECHATRONICUML model into UPPAAL in one single model-to-model transformation step.

The migration consists of several sub-tasks namely the CIC migration to UPPAAL processes (Section 5.4.13.1), the RTSC migration to UPPAAL timed automata (Section 5.4.13.2), the encoding of the asynchronous message communication (Section 5.4.13.3), and the migration from MTCTL to UTCTL (Section 5.4.13.4).” [*DGH15]

5.4.13.1 Migrating of Component Instance Configurations To Uppaal Processes

“Currently, we do not support the transformation from a MECHATRONICUML component or an arbitrary CIC to UPPAAL but only the transformation from an RTCP to UPPAAL. Therefore, we only have two kinds of CICs to transform. Both consist of component instances with exactly one port (single or multi) only:

- If the input is a one-to-one RTCP, we have to transform a CIC that consists of two component instances, which differ in their component type, and that are connected via their single port instance.” [*DGH15]
- “If the input is a one-to-many RTCP, we have to transform a CIC that consists on the one hand of one component instance [of] type t1 and a multi port instance with n [subport] instances.” [*DGH15] Hereby, n is a fixed number that is equally to the role multiplicity that the DSMC user specified within the verification options (see Section 5.4). “On the other hand, the CIC consists of n component instances of type t2. Each of these n [component instances] have a single port instance of type p. Moreover, each single port [instance] is connected to one [subport instance].” [*DGH15]

“We illustrate our concept in Figure 5.15 for a one-to-many RTCP with a [role multiplicity] of 2 that we transformed into a CIC in the first translation step (see Section 5.4.1). As a result, the CIC consists of three component instances based on two different components (Comp1 and Comp2) and two port connector instances. In our illustration, the component Comp1 references a multi port RTSC that contains a flat coordinator RTSC and a flat [subport] RTSC. Moreover, the component Comp2 references a flat single port RTSC.

As depicted in Figure 5.15, we migrate each single port behavior to one UPPAAL template and a multi port behavior into three UPPAAL templates (one for the multi port, one for the coordinator, and one for the [subport])³. RTSC-based templates have two parameters: the first one is the component instance id; the second is the port instance id. In our

³For simplicity reasons, we assume in our descriptions that the single role as well as the coordinator and the subrole have a flat RTSC. If this is not the case, the hierarchy flattening will produce more RTSC (see Section 5.4.10). For each of these RTSCs we additionally create one template.

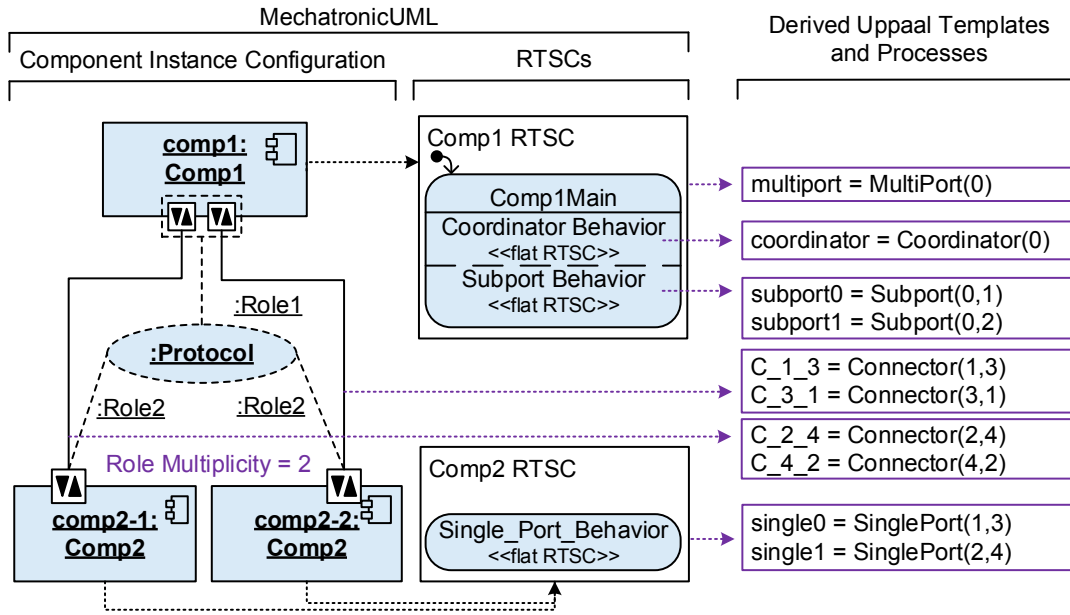


Figure 5.15: CIC Migration (Based on [*DGH15])

example, we generate four processes for Comp1: multiport, coordinator, subport0, and subport1.” [*DGH15] In addition, we generate two processes for Comp2: single0 and single1.

“The asynchronous communication between (sub-) port instances is defined via port connector instances that all have the same connector type and the same QoS assumptions. Therefore, we generate additional processes based on a common connector template for all these instances (see Section 5.4.13.3) to encode the transmission (including properties such as reliability and message delay) and the buffering of messages. In particular, we generate one UPPAAL process per connector and per communication direction. Each process gets, among others, as parameter the source and target port instance id. In our example, we generate four processes based on the common template as the multi port [multiplicity] is 2 and the communication direction is bidirectional.” [*DGH15]

5.4.13.2 Migrating RTSCs to Timed Automata Templates

“Migrating a flat RTSC to a timed automaton template is for the majority of [model] elements straight forward because RTSCs are based on timed automata. Moreover, all RTSC normalizations removed most of the [model] elements that are not available in timed automata. For example, we can directly transform states (including invariants) and transitions (their source and target) [of an RTSC] into [states and transitions of an UPPAAL timed automaton]. The transformation for the definition of variables, operations, clocks, data types, and synchronization channels is also trivial. Moreover, the migration of MECHATRONIC-UML’s action language is also straightforward as it has the same expressiveness as UPPAAL’s expression language. Nevertheless, some aspects [of the RTSC’s migration] still require more sophisticated concepts. We explain them in more detail in the following.” [*DGH15]

Hierarchical RTSCs

Due to the RTSC Hierarchy Normalization (see Section 5.4.10), all RTSCs are flat except the top most RTSC of each port. “These [so-called] port RTSCs contain a composite state that contains flat RTSCs. We do not migrate this composite state to UPPAAL because it is always active, has no invariants, and has no incoming or outgoing transitions. However, we migrate the variables, operations, clocks that are defined within its RTSC and the channels, which are defined in this state, into UPPAAL’s global declarations. Moreover, we migrate all contained flat RTSCs as described above.” [*DGH15]

RTSC Transition Labels

“An RTSC transition distinguishes between conditions and effects. Available conditions are clock constraints, guards, synchronizations, and an incoming message (i.e., the existence of a message in an incoming buffer); available effects are consuming a message (this effect is implicitly if the user defines an incoming message), sending a message, actions, and clock resets. UPPAAL categorizes its [transition] differently: it supports selects (that enable non-deterministic choices), guards, synchronizations, and updates. Our migration concept is as follows:

- We extract all non-deterministic choices from the RTSC transition action into UPPAAL’s select category.
- We merge RTSC clock constraints, guards, and trigger messages within UPPAAL’s guards. In particular, we concatenate them with the and operator. Thus, their order is not important.
- We transform MECHATRONICUML synchronizations into UPPAAL synchronizations (we give more details about this in Section 5.4.13.2).” [*DGH15]
- We merge all parts of an effect into UPPAAL’s update category. In particular, we concatenate them in the following order: (1) We store the incoming message into a temporary variable `msg`, (2) we execute the transition action (this action may access the parameters of `msg`), (3) we send a new message `msg2` (the parameter bindings may access the parameters of `msg` and variables, which may have changed values due to the transitions action), (4) we take `msg` from the incoming buffer and delete `msg`, and (5) we execute all clock resets. This order nearly obeys the specification of MECHATRONICUML for executing the transitions effects (see [*DPP+16, p. 48]). The only deviation is that we do not remove the incoming message from the buffer directly in step 1 but in step 4. Though, this has no semantic deviation as the steps 2 and 3 cannot access the message buffer directly but can only access the temporary variable `msg`.

Selector Expressions in Multi Ports RTSCs

“In MECHATRONICUML, the coordinator and the [subport] of the [multi port] can synchronize with each other. Though, when specifying the [multi port] RTSC, the [multi port multiplicity range] is not defined yet. Consequently, the user cannot define a synchronization between the coordinator and a concrete instance of a sub port. However, as [subport] instances are always ordered as a list, the user may specify the synchronization via the following reserved keywords as selectors: `self`, `next`, `previous`, `first` and `last` [*DPP+16, p. 60]. Each selector either refers at runtime to a [subport] instance or is null.

UPPAAL does not support selector keywords but only numeric selectors. Thus, we migrate them as follows: At first, we create for each of the five selector kinds one array. Each array has the size of the user-defined [multiplicity]. Therefore, we write at index i of the array the result of the selector expression of the [subport] instance i . The result is always a number. It is either greater or equal to zero and corresponds to another [subport] instance id or it is -1, which means that it does not refer to a concrete [subport] instance id (i.e., -1 represents null).

Figure 5.16 shows an example for the next array. We assume that the [multiplicity] of the RTCP is 2. Therefore, the array also has the size of 2. The follower of [subport] instance 0 is instance 1. Thus, we write at index 0 of the array the value 1. Instance 1 has no follower. Therefore, we write at index 1 of the array the value -1.

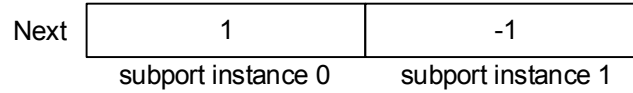


Figure 5.16: Defining the Next Array for Selector Expressions [*DGH15]

Within the timed automaton template, which is the same for all [subport] instances, we adapt the selector keyword of a synchronization by an access of the corresponding array and the [subport] instance id, which is given as a template parameter. An example is given in Figure 5.17. Here, we define that the [transition] shall synchronize using the channel *overtake* with the next subrole instance within the same (multi) port instance id.” [*DGH15]

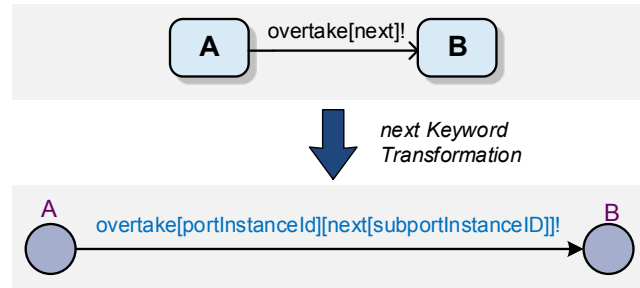


Figure 5.17: Using the next Keyword Within a Selector Expression (Based on [*DGH15])

Adding Intermediate States

“A transition in UPPAAL may not be active and may not take time in contrast to MECHATRONICUML. This has several consequences:

- It is not possible in UTCTL to evaluate the MTCTL predicate [transitionFiring](#).
- The UPPAAL counterexample cannot provide snapshots where a transition [of an RTSC] is firing but the MECHATRONICUML counterexample shall provide these snapshots.

” [*DGH15]

For solving these problems, we replace each transition of an UPPAAL timed automaton that corresponds to a non-time-consuming RTSC transition by an intermediate state (with a unique name) and two transitions (see Figure 5.18). Hereby, we move the transition label to the first transition (i.e., the transition that connects the source state with the intermediate state).

Note that we already applied the concept of intermediate states for transitions that are time-consuming (i.e., that have a deadline) during the deadline normalization (see Section 5.4.7). Therefore, we can reuse this intermediate state and do not have to add additional ones. As a consequence, we can evaluate whether a transition fires by verifying whether the intermediate state gets active. Moreover, for visualizing a counterexample, we can highlight that a transition fires if the intermediate state is active.

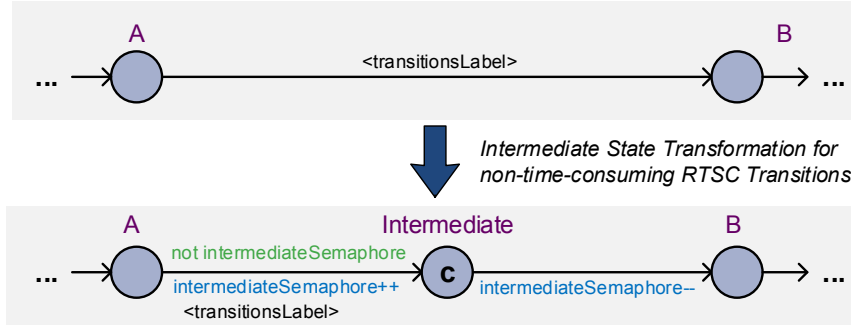


Figure 5.18: Adding of Intermediate States (Based on [*DGH15])

Though, this concept alone is not valid as MECHATRONICUML's semantics definition of non-time-consuming transition requires that the firing is one atomic step [*DPP+16, p. 46]. However, the current concept enables that other transitions (of other timed automata) fire while the intermediate state of a non-time-consuming transition is active. Therefore, we have to enforce that the intermediate state is immediately left before any other state may be left. We realize this by additionally applying the following changes:

1. For each intermediate state and its two transitions that correspond to a non-time-consuming transition of an RTSC:
 - a) We set the intermediate state to committed.
 - b) If the first transition (i.e., the transition that connects the source state with the intermediate state) is empty, we add the guard `intermediateSemaphore==0`, whereby `intermediateSemaphore` is a global variable that is initially 0. If the guard is not empty, then we concatenate this expression using the operator `and`.
 - c) We add the update statement `intermediateSemaphore++` to the first transition.
 - d) We add the update statement `intermediateSemaphore--` to the second transition (i.e., the transition that connects the intermediate state with the target state).
2. We add the guard `not intermediateSemaphore` (or concatenate this expression using the operator `and`) to all other transitions of the NTA.

Note that step 1a alone is not sufficient as more than one UPPAAL process may be in a committed state at the same time due to a synchronization of two timed automata. Then, UPPAAL non-deterministically chooses which transition fires first. We use the steps 1.b) to 1.d) and step 2 to implement a variant of a semaphore to handle this problem. We distinguish two cases of this problem:

- In the first case, only one transition wants to enter an intermediate state. Due to our implementation, the synchronization may only happen if `intermediateSemaphore` is

0. After the synchronization, `intermediateSemaphore` is 1. As all other transitions define that `intermediateSemaphore==0` must hold, only the outgoing transition of the intermediate state may fire.
- In the second case, both transitions want to enter their intermediate state. Due to our implementation, the synchronization may only happen, if `intermediateSemaphore` is 0. After the synchronization, `intermediateSemaphore` is 2. As all other transitions define that `intermediateSemaphore==0` must hold, the two outgoing transitions of the intermediate state have to fire first before any other transition may fire. Note that it is not defined which of the two transitions fires first. Though, this has no semantic impact as both transitions may not have actions or synchronizations. As UPPAAL is closed source, it is unclear whether this has a significant impact on the model checking performance.

Noteworthy, the deadline normalization (see Section 5.4.7) defines that the effect of the transition is part of the transition that has the intermediate state as its source. In contrary, we define here that we move the complete transition label to the first transition (the one that has the intermediate state as its target). This is possible as in this case no other transition (with synchronizations or effects) may fire and no time may progress.

Urgent Transitions

“MECHATRONICUML defines urgency at transitions while UPPAAL defines it at synchronization channels. We migrate this aspect by utilizing the UPPAAL modeling pattern *urgent edges* [BDL06, p. 34]. In short, the pattern defines that we have to add an urgent channel (we call the channel *hurry*) and a new template that defines a single [state] with a self-[transition] that only defines the sending (“!”) of *hurry*. Then, the pattern proposes that we add the receiving (“?”) of channel *hurry* to all [transitions] that shall fire urgently.” [*DGH15] However, two possible conflicts arise: (1) the [transition] may already define a synchronization and UPPAAL does not allow that [a transition] synchronizes with more than one channel. (2) UPPAAL does not allow that [a transition] may have a clock constraint if it shall synchronize via an urgent channel.

“We solve these two possible conflicts as follows: First, we collect all UPPAAL [transitions] that correspond to an urgent transition of the RTSC. Then, we check [whether] the RTSC transition is urgent and already synchronizes via a channel. If both [conditions are] true, we set the corresponding channel at the [UPPAAL transition] to urgent. If the transition is urgent but it does not synchronize via a channel, we use the newly created channel *hurry* [as recommended by the pattern]. Other cases, e.g., a channel is used by urgent and non-urgent transitions, may not occur due to our urgency normalization (see Section 5.4.12).” [*DGH15]

We show an example in Figure 5.19. Here, the RTSC defines an urgent transition between the states A and B. At the UPPAAL level, we define the urgent channel *hurry*. We assign the synchronization *hurry?* to the migrated transition within the UPPAAL model. Moreover, we add one timed automaton, which defines the state *Idle* and a self-transition that defines the synchronization *hurry!*.

5.4.13.3 Encoding Asynchronous Communication

“As already mentioned in Section 5.4.13.1, our idea is to generate one common connector template to encode the transmission (including properties such as [the reliability, the message

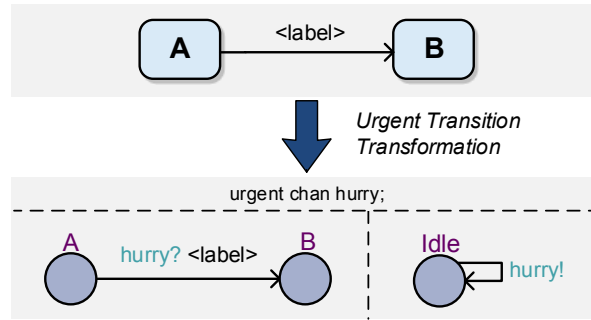


Figure 5.19: Migrating Urgent Transitions

delay, and if the message order shall be preserved)) and the buffering of message[s]. Then, we can generate a process per connector instance and per communication direction.

In particular, each connector process has the following tasks:

- Getting messages from the sending role
- Transmitting the message while considering the message delay (with or without preserving the message order)
- Enable that a message may get lost
- Storing a message in the correct incoming message buffer of the receiving role
- Discarding a message according to the defined strategy if the incoming buffer is full

To fulfill the stated tasks, we distinguish between [four] different data structures:

Connector Message Buffer Each connector process has exactly one such buffer that stores all messages (independent of their type) that are in transit. If the connector shall preserve the message order, then the buffer is conceptually a circular FIFO structure, otherwise it is [an unordered] set. We realize both variants by an array of messages that has a fixed size due to UPPAAL's language limitations (i.e., the language is C-like but does not enable pointers or linked lists). An array in UPPAAL has a static size. Therefore, it is possible that a connector may be full but a new message shall be sent, which would result in a connector buffer overflow if it is not handled. However, if the connector is reliable, then we assume that the connector can transport all messages. Therefore, the buffer must be large enough. In addition, if the connector is unreliable, we also require that the connector buffer is large enough as we only want a single point in the model, which we explain in the following, where a message may get lost due to the connector's unreliability. As a consequence, we have to identify a connector buffer size such that an overflow may never happen. Though, this is not possible without a dynamic analysis as this depends on the execution behavior of the RTSCs. Therefore, our solution is that the DSMC user has to specify the size by himself before the DSMC starts. We enable the DSMC user to check [whether] this size is too small using the MTCTL default property **AG not connectorOverflow**; (see Section 4.4). If the property is false, the user needs to increase the connector message buffer size [and verify the property again until it is true].

Connector Transmission Time Buffer Each connector process has exactly one such buffer to keep track of the transmission time of each message. Analogously to the connector

message buffer, we store the transmission times conceptually in a circular FIFO buffer or [an unordered] set [... We realize both variants] by an array of clocks. Moreover, this array has the same size as the connector message buffer.

Incoming Message Buffer Each connector process has n incoming message buffers that store the messages that the receiving role may consume, where n is defined within the role. The MECHATRONICUML user can define for each buffer a specific size, a discard strategy, and which message types the buffer may store. Analogously to the connector message buffer, we store the messages in a circular FIFO buffer, which we realize by an array of messages. This is sufficient as the buffer semantics define that the messages are stored via FIFO only.

Message Parameter Buffer Each connector process has exactly one such buffer per message type. It stores the parameter values for all message instances of this type that are currently in transit or within an incoming message buffer. Analogously to the connector message buffer, we store the transmission times in a circular FIFO buffer. We realize this by an array of structs, whereby the struct defines the parameter types of this message. The size of this buffer is the sum of the connector message buffer size plus the size of the incoming message buffer that stores this message type.

In the following, we define our concept for encoding the asynchronous communication based on these four data structures. We separate the concept in three phases: *dispatching*, *transmitting*, and *consuming*.” [*DGH15] In Section 3.3.2 and 3.4, we present the explicit and implicit QoS assumptions of an RTCP that we defined especially for applying model checking (as a reminder: the explicit ones are the maximum message transmission delay, the reliability, and the message order preservation). Therefore, for each phase, we do not only explain its functionality but also how we handle the QoS assumptions (especially the explicit ones). Noteworthy, most of the assumptions that we state in Section 3.4.3 do not require explicit model elements at the level of UPPAAL but are still essential as our model checking results are only valid if the middleware considers all of them. In fact, this concerns the following assumptions: 2, 3, 4, 5, 6, 7, 10, 11, 12, and 13. For example, assumption 5 states that we assume that the sender’s middleware cannot forbid the sending of the message. Therefore, we must not add behavior that prevents to dispatch a message.

“We [...] illustrate our concept using Figure 5.20, which shows a concrete example at the level of MECHATRONICUML, and using Figure 5.21, which shows the realization in UPPAAL.

Figure 5.20 depicts an unidirectional one-to-one RTCP that defines the roles Role1 and Role2. Role1 may send the message red, which has two parameters, and the message blue. Role2 may receive these messages via two incoming message buffers. The first message buffer has size 2 and may only store red messages, while the second has a size of three and may only store blue messages.” [*DGH15] Furthermore, the role connector of RTCP RTCP1 defines the following QoS assumptions: the maximum message delay is 7 ms, the connector is unreliable, and the message order shall be preserved (i.e., overtaken messages may be discarded). The figure also depicts additional concepts that we will explain in the following.

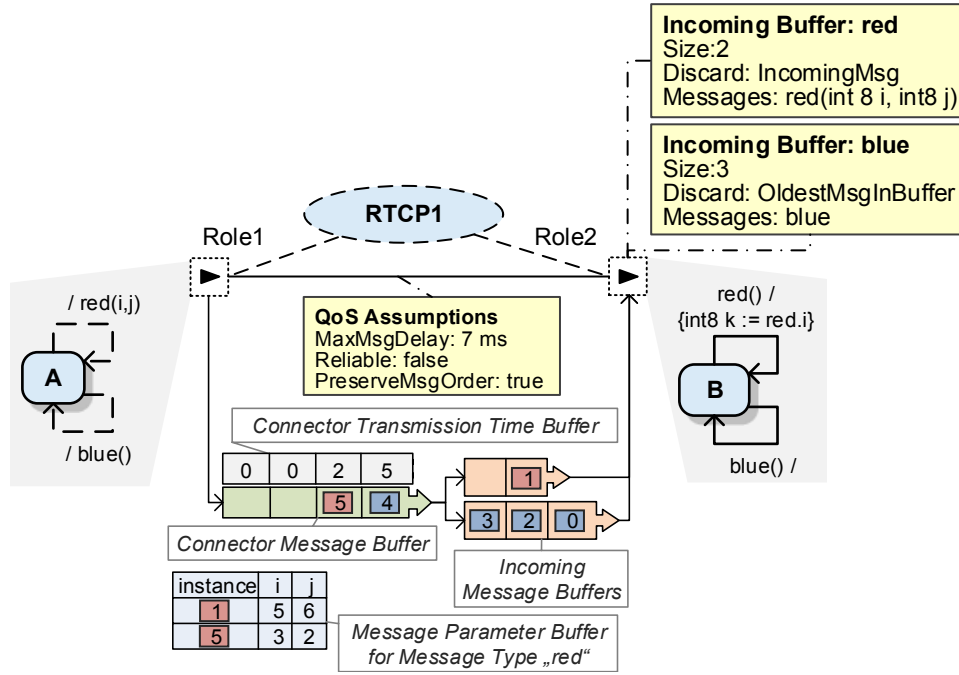


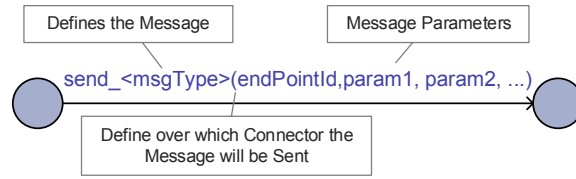
Figure 5.20: Asynchronous Message Exchange Concept (Based on [*DGH15])

Dispatching

“Dispatching at the level of MECHATRONICUML means that the role RTSC fires a transition that causes a message to be [sent] (including their parameters) over the role connector. An example is given in Figure 5.20, where Role1 can send the message red (along with the two parameters i and j) and message blue.

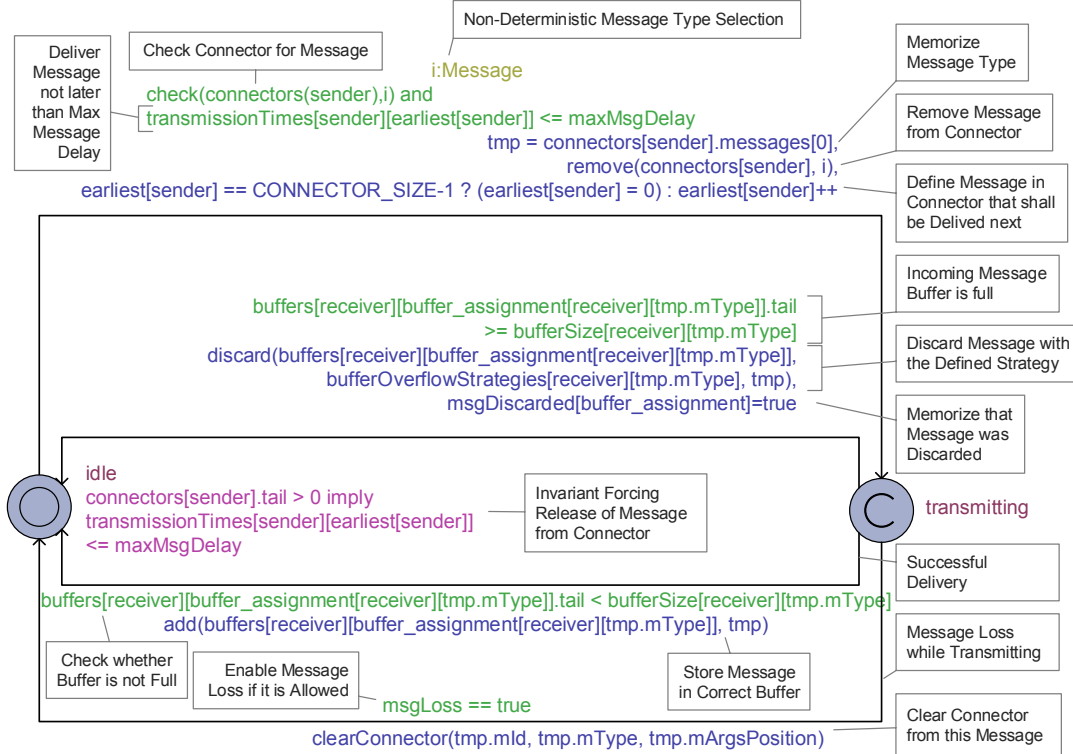
We show our schema for migrating a dispatch into UPPAAL in Figure 5.21(a). The raise message event is transformed into a function call. For each message type, we define an own function within the global declaration of the UPPAAL model. All functions require the endpoint instance as a first argument to distinguish over which connector the message shall be [sent]. Moreover, if the message type has message parameters, they appear as additional arguments in the function signature. Each function stores the message within the connector message buffer and the parameter values within the corresponding message parameter buffers. In addition, the function sets the corresponding clock within the connector transmission time buffer to zero.

Noteworthy, if the connector buffer is already full before we can insert our new message, we will not insert the message. This is an unwanted behavior as defined before. Therefore, we set in such a case a global variable named `connectorOverflow`, which is initially false, to true. Thus, we memorize that a `connectorOverflow` happened. A MECHATRONICUML user can check this value indirectly by formalizing the MTCTL default property **AG not `connectorOverflow`**.” [*DGH15] One global boolean variable is sufficient as we are not interested whether more than one connector overflow occurred or which connector message buffer had the buffer overflow. Instead, we only want to know whether any connector message buffer ever had a buffer overflow.

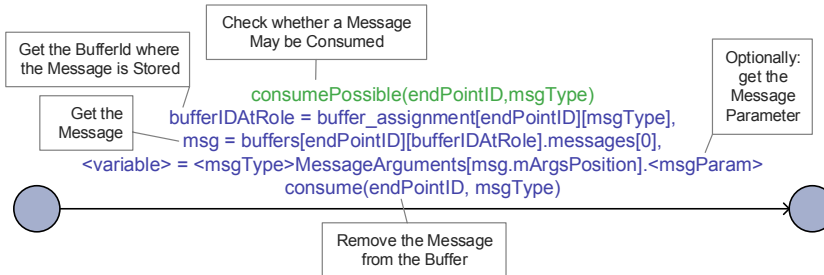


(a) Schema for Dispatching an Asynchronous Message in UPPAAL

Template Parameters: `const endPointInstance sender, const endPointInstance receiver,`
`const int maxMsgDelay, const bool msgLoss`



(b) Generic Template for Asynchronous Transmissions that Preserve the Message Order



(c) Schema for Consuming an Asynchronous Message in UPPAAL

Figure 5.21: Asynchronous Communication in UPPAAL (Based on [*DGH15])

Transmission

“If the connector shall preserve the message order, then it will deliver the first message (at position 0) in the connector message buffer to the receiving role instance not later than the maximum message delay. Note that the actual delay of each message is stored using the associated clock in the connector transmission time buffer. In particular, the connector removes the first message from the connector message buffer, defines which message will be the next one that shall be delivered, and defines that the associated clock is no longer assigned to a message.

If the connector does not need to preserve the message order, then the behavior is quite similar except that any of the messages in the connector message buffer may be selected to be transmitted. Noteworthy, the connector considers the actual delay of each message and, therefore, will transmit each message before its maximum delay.

As soon as the connector selected one message, three possible cases may occur:

1. If the connector QoS assumptions allow that messages may get lost [(i.e., the connector is unreliable)], then this is one possible case. If the message is lost, no further action is necessary [except the default behavior to clear the connector from this message including the message parameters].
2. If the incoming message buffer that is assigned to the message under delivery is not already full, then the message may be stored within the buffer.
3. If the incoming message buffer that is assigned to the message under delivery is full, then we execute the message discard strategy that is assigned to this buffer, i.e., we either discard the oldest message within the buffer and store the new message in the buffer or we discard the new message.

” [*DGH15]

We use Figure 5.20 again for an execution example at the level of MECHATRONICUML. “Here, RTCP1 is currently transmitting two messages: A blue message with id 4 is transmitting since 5 time units, while a red message with id 5 is transmitting since 2 time units.” [*DGH15] Therefore, the connector message buffer contains the blue message with id 4 at its head and the red message with id 5 on its second position. Moreover, in the connector transmission time buffer, the clock at the first position has the value 5 while the clock at the second position has the value 2. All other clock values of this buffer are currently not relevant. The parameters of the red message with id 5 are stored in the message parameter buffer for messages of type red. In our example, the message with id 5 has the parameter values $i==3$ and $j==2$. At this point in time, the connector, which shall preserve the message order, may start to transmit the blue message with id 4 into its incoming message buffer because this message is in the top position (i.e., it was dispatched before the red one). Though, the connector might also wait additional 2 time units as the maximum message delay is not reached yet. If the connector starts to deliver this message, it may either choose to lose the message as the QoS assumptions define that the connector is unreliable. Alternatively, the connector may choose to insert the message into the incoming message buffer. In the latter case, the defined discard strategy must be applied as the associated incoming message buffer is already full. According to the buffer specifications, the oldest message in the incoming message buffer shall be discarded, which is the one with id 0. Therefore, the message with id 0 is removed, the message with id 2 becomes the new head, and the message with id 4 is moved from the connector message buffer to the tail of this incoming message buffer. Afterwards, the red message with id 5 is at the head of the connector

message buffer. In parallel, the associated clock in the connector transmission time buffer is being released from the mapping to the message with id 4. Consequently, the clock with value 2, which is mapped to the message of id 5, is now at the head of the connector transmission time buffer. Afterwards, the connector can send the message with id 5 immediately but may also wait up to 5 time units (then the maximum delay of this message is reached).

“We show our UPPAAL connector template that contains all tasks for the transmission [in the case of preserving the message order] in Figure 5.21(b). [...] We define [the maximum message transmission delay and the reliability] as template parameters ([maxMsgDelay] is the maximum message delay, [...] msgLoss] indicates [whether] messages may be lost during transmission). Moreover, the template has two more parameters for defining the sending and receiving endpoint instance. The behavior definition of the template is as follows: Initially, all buffers are empty and the connector is in [state] idle.” [*DGH15] As long as no message shall be sent, the timed automaton remains in state idle. Though, if the connector message buffer is filled with at least one message, then the state may be left immediately. Moreover, the invariant of state idle enforces to leave this state when the maximum message delay is reached for the oldest message in the connector (i.e., the head of the FIFO queue). The single outgoing transition leads to the state transmitting. The transition removes the oldest message from the connector message buffer and defines the next message that shall be delivered. The state transmitting is committed as it shall be immediately left. It has three outgoing transitions—one per case that we stated before (discard a message, deliver the new message in the buffer, or loose the message). We discard the oldest message in the incoming message buffer if the strategy “Discard Oldest Message in Buffer” is selected for this buffer. However, we discard the incoming message if the strategy “Discard Incoming Message” or the strategy “Never Happens” is selected for this buffer. Discarding a message despite the strategy “Never Happens” is necessary as we would otherwise have to stop the model checking immediately because the model would be no longer valid, i.e., we would have to overwrite a still valid message in the incoming message buffer. Though, when discarding a message, we memorize that this buffer discarded a message. Therefore, the DSMC user may find out whether a buffer had to discard a message via the MTCTL predicate `messageDiscarded`. In the case of a successful delivery, we first verify that the buffer is not full and then store the message within this buffer. In the case of an unreliable connector, we have nothing to do, i.e., we do not have to process our temporary message stored in variable tmp any further. Finally, all three transitions return to the state idle.

The connector template that does not need to preserve the message order is quite similar. However, its invariant at state idle defines that the state must be left if any maximum message transmission delay of a message in the connector buffer expires. Moreover, the [transition] from idle to transmitting may choose any of the messages in the connector buffer. As a consequence, the update statements of this transition as well as the guard and the update statements of all other transitions are adapted such that they are able to process the selected message instead of the first message of the connector message buffer.

Both connector templates fulfill all QoS assumptions that we stated in Section 3.4.3. For example, our connector specifies the maximum message transmission delay as the time between dispatching/sending the message and storing it into the buffer because the associated clock is reset when dispatching/sending the message and the message is stored into the buffer before the clock has a value greater than its maximum message transmission delay. As another example, our connector does not forbid to send a message as the message may always

be put into the connector message buffer directly (otherwise the MTCTL default property **AG not connectorOverflow** would be violated, which is forbidden).

Consuming

“Consuming at the level of MECHATRONICUML means that the RTSC defines a message type as a transition firing condition. If a message with the correct type is at the front position of the associated message buffer, the transition can fire, remove the message from the buffer, and, access the message parameters (if they are defined for the message). An example is again given in Figure 5.20, where Role2 can consume the messages red and blue.

We show our schema for migrating the consuming into UPPAAL in Figure 5.21(c). The trigger message event is transformed into a call of the function `consumePossible` (defined in UPPAAL’s global declarations) within the [transition]’s condition part. The function returns true if a message of given type is available in the buffer of the defined endpoint instance. If a message may be consumed, the [transition] can fire. In particular, it will read the message, which is at the buffer position 0, retrieve the message, and retrieve the message parameters (optionally). Finally, the [transition]’s action will consume the message, i.e., remove it from the incoming message buffer.” [*DGH15]

5.4.13.4 MTCTL to UTCTL

“The MTCTL expressions that we have to transform during the migration are already quite similar [to] UTCTL expressions. This is due to the design of MTCTL, which is based on UTCTL, and due to our MTCTL normalizations that removed MECHATRONICUML-specific [model] elements. Consequently, the majority of the transformation is straight forward. The transformation of dynamic predicates and functions that are not supported by UPPAAL are the only non-trivial aspects that are left. We describe their transformation in the following.

deadlock UPPAAL already provides a keyword with the same name and the same semantics. Therefore, we do not have to transform it.

connectorOverflow As stated in Section 5.4.13.3, we provide a boolean variable in our UPPAAL model that switches to true if a connector overflow happens. Thus, we only have to replace the predicate by this variable.

stateActive(state) UPPAAL does not require a keyword to [express] that a [state] is active. It is sufficient [that] the name of the [state] is stated. Therefore, we replace the predicate with the name of the state.

transitionFiring(transition) As stated in Section 5.4.13.2, we added intermediate [states] to the timed automata. Therefore, we replace this predicate by referencing the corresponding intermediate [states].

messageDiscarded(buffer) As stated in Section 5.4.13.3, we provide a boolean array named `msgDiscarded` in our UPPAAL model. If a message is discarded, we switch the corresponding entry of the array to true. Thus, we only have to replace the predicate with a reference to the corresponding index of this array.

messageInBuffer(messageType, buffer) We replace this predicate with a function `checkMessageInBuffer` that has as parameters the buffers array (its indices are the role index and the buffer index) and the name of the message type. The [function] returns [true] if the message is in the given array.

messageInTransit(messageType) We replace this predicate with a function `checkMessageInConnector` that has as parameter the name of the message type. The [function] internally calls the function `checkMessageInBuffer` for all available connector buffers. Reusing this function is possible [because connector buffers and message buffers are both implemented as arrays]. `checkMessageInConnector` returns true if any call of the function `checkMessageInBuffer` returns true.

bufferMessageCounts(buffer) We replace this predicate by an UPPAAL function, which returns the current number of messages in the given incoming message buffer.

” [*DGH15]

5.5 Uppaal-Level Activities

The Activities 2–4 (see Figure 5.1) of our approach all happen at the level of UPPAAL. We explain Activity 2 (the model-to-text transformation) in Section 5.5.2, Activity 3 (the automation of UPPAAL) in Section 5.5.3, and Activity 4 (the model-to-text transformation) in Section 5.5.4. Before we execute Activity 2, we can optionally execute a layouting of the timed automata templates. We present our layouting concept in Section 5.5.1.

“Noteworthy, [these] three activities are realized as Java black-box units in QVTo [QVT; QVTo]. This is necessary as QVTo can only reuse traceability links of the same transformation. Consequently, the forward translation, these three steps, and the back-translation, which we discuss in the following, all belong to one QVTo transformation for the complete DSMC.” [*DGH15]

5.5.1 Layouting Timed Automata Templates

We provide the option to layout each timed automaton template of the NTA. By default, this option is disabled as it requires additional workload and is not necessary as all models shall be hidden to the DSMC user. Though, if the DSMC developer wants to investigate the UPPAAL text files within the UPPAAL development suite, then he will struggle to understand the templates because UPPAAL’s default layouting algorithm only arranges the states on a grid, connects the transitions as straight lines, and arranges labels without considering whether they overlap. In order to ease the work of the DSMC developer, we use the tool GraphViz [Gra] to improve the layouting of each timed automaton. In particular, we give GraphViz the set of states and transitions of our UPPAAL EMF model such that it can calculate a good layout. Then, we enlarge the given result by the factor 120, i.e., we multiply each coordinate with 120. In our experience, this factor is appropriate for good readable diagrams. Currently, we neglect that GraphViz also layouts the optimal position of each label. Instead, we rely on UPPAAL to layout them. Finally, we enrich the UPPAAL EMF model with the layout information.

For example, in Figure 5.22, we show the template of a timed automaton layouted by UPPAAL that represents the embedded RTSC of state Overtaking from role Overtaker of

RTCP Overtaking. Obviously, this timed automaton is hard to understand without layouting it manually. In contrary, our layouted version is shown in Figure 5.23. In our version, the transitions do not intersect anymore, transitions may be curved via bend points, and the labels are arranged correctly. Therefore, the DSMC developer can inspect the timed automata in UPPAAL's development suite more efficiently as he does not need to rearrange the states, transitions, and labels.



Figure 5.22: Timed Automata Template Layouted by UPPAAL Only

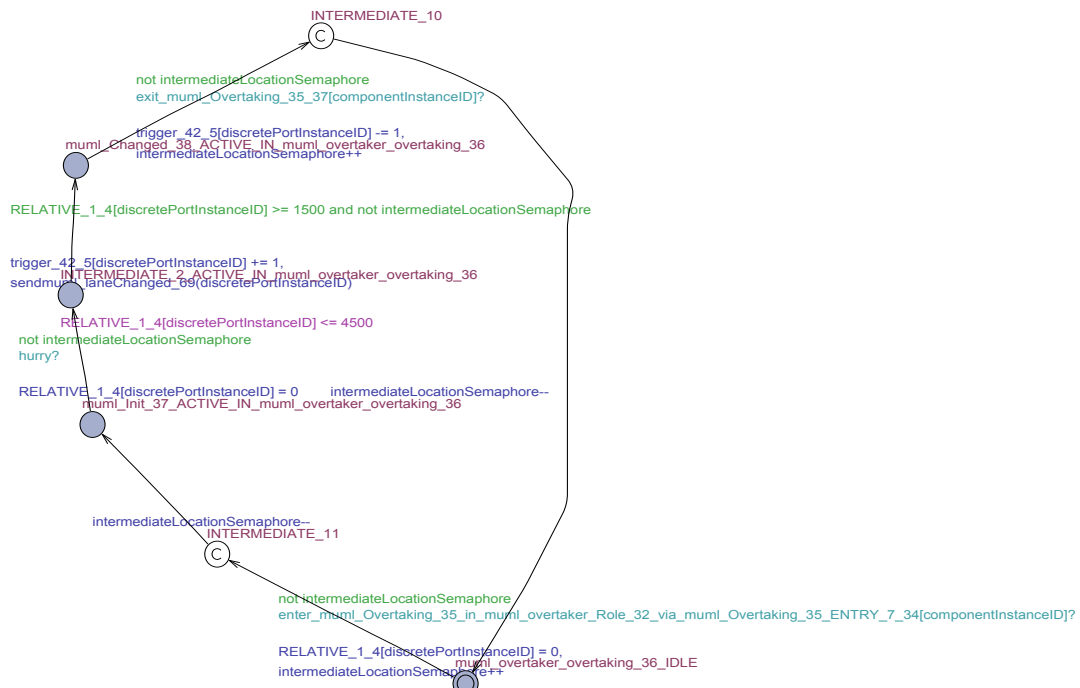


Figure 5.23: Timed Automata Template Layouted via GraphViz

5.5.2 Model-to-Text-Transformation

We carry out Activity 2 via a model-to-text transformation using Xtend2 [Xtend]. As a result, we get two files:

- an XML file that stores the NTA and respects the DTD defined by UPPAAL [UDTD] and
- a text file that stores the UTCTL properties.

Our translation is straightforward as we do not transform any additional (domain-specific) aspects but only change the format from the UPPAAL EMF model to the UPPAAL XML/text format and resolve expressions.

We rely on EMF's automatic XML serialization for showing examples of the transformation's input and output as this facilitates the comparison. As a first example, we show an excerpt of an NTA, which is the result of the model-to-model transformation of RTCP Overtaking:

```
<template name="UrgencyProvider" init="// @template .5/ @location.0">
  <location name="idle"/>
  <edge source="// @template .5/ @location.0" target="// @template .5/ @location
    .0">
    <guard xsi:type="expressions:NegationExpression">
      <negatedExpression xsi:type="expressions:IdentifierExpression"
        identifier="// @globalDeclarations / @declaration .30/ @variable.0"/>
    </guard>
    <synchronization kind="!">
      <channelExpression identifier="// @globalDeclarations / @declaration .0/
        @variable.0"/>
    </synchronization>
  </edge>
</template>
```

In particular, this excerpt defines the timed automaton template UrgencyProvider, which we created during the urgent transition migration (see Section 5.4.13.2). The template specifies a single state and a self-transition that specifies a guard as well as a synchronization (note that the EMF model uses UPPAAL's terminology of edges and locations, which we name state and transition within this thesis). We show the result of our model-to-text transformation for this excerpt in the following:

```
<template>
  <name>UrgencyProvider</name>
  <declaration>
  </declaration>
  <location id="idle_UrgencyProvider">
    <name>idle</name>
  </location>
  <init ref="idle_UrgencyProvider"/>
  <transition>
    <source ref="idle_UrgencyProvider"/>
    <target ref="idle_UrgencyProvider"/>
    <label kind="select"></label>
    <label kind="guard">not intermediateLocationSemaphore</label>
    <label kind="synchronisation">hurry!</label>
    <label kind="assignment"></label>
  </transition>
</template>
```

As the example shows, no additional information is added but only the XML structure is changed and expressions are resolved. As a second example, we show an UTCTL property that results of the MTCTL property **AG not deadlock**;

```
<properties xsi:type="requirements:UnaryProperty" comment="a_deadlock_
  never_occurs" quantifier="A" operator="[]">
  <expression xsi:type="expressions:NegationExpression">
    <negatedExpression xsi:type="requirements:DeadlockExpression"/>
  </expression>
</properties>
```

The result of our model-to-text transformation for this UTCTL property looks as follows:

```
/*
  a deadlock never occurs
*/
A[] not deadlock
```

Again, no additional information is added but only the XML structure is removed and expressions are resolved.

In Appendix D.1, we show the complete UPPAAL representation of RTCP Overtaking that we generated using our DSMC including the timed automata templates, the global declarations within UPPAAL, and the UTCTL properties.

5.5.3 Automating Uppaal

“For Activity 3, we automatically invoke UPPAAL’s command line verifier. As a result, UPPAAL will return textual results for each UTCTL property and—if available—UPPAAL’s textual traces (e.g., counterexamples). As defined in the beginning of Section 5.4, the domain expert may configure the UPPAAL model checking itself, e.g., UPPAAL’s hash table size, the state space reduction, and the (counterexample) trace kind. Moreover, we define that UPPAAL shall reuse the generated state-space when possible, which may speed up the verification.” [*DGH15] In addition, we choose the option that we do not want to print out an option summary or a progress indication. “We leave all other UPPAAL configurations at their default values, e.g., the search order is breadth first.” [*DGH15]

For example, we generate the following command line call when verifying the UPPAAL representation of RTCP Overtaking with a single property for a trace using the default DSMC options:

```
D:/UPPAAL/uppaal-4.1.19/uppaal-4.1.19/bin-Win32/verifyta.exe
-q -s -t 0 -H 27 -S 1
C:/Users/mer-sd/AppData/Local/Temp/muml_OvertakingCIC_81.xml
C:/Users/mer-sd/AppData/Local/Temp/muml_OvertakingCIC_81.q
```

The single parts of this command line call have the following meaning:

D:/UPPAAL/uppaal-4.1.19/uppaal-4.1.19/bin-Win32/verifyta.exe The path to UPPAAL’s command line verifier.

-q Suppress the option summary.

-s Suppress the progress indicator.

-t 0 Write a concrete trace to stderr with the option some.

-H 27 Set the hash table size for bit state hashing to 2^{27} .

-S 1 Set the state space reduction option to conservative.

C:/Users/mer-sd/AppData/Local/Temp/muml_OvertakingCIC_81.xml The path to the UPPAAL NTA file.

C:/Users/mer-sd/AppData/Local/Temp/muml_OvertakingCIC_81.q The path to the file, which defines the UTCTL property.

An example for a textual result of UPPAAL when verifying multiple properties without producing a trace looks as follows:

```
Verifying formula 1 at line 1
— Formula is NOT satisfied.
```

```
Verifying formula 2 at line 5
— Formula is satisfied.
```

As the example shows, UPPAAL states for each verification property (UPPAAL uses the terms formula and property synonymously) whether it is satisfied or not. Hereby, it references the actual property by referring to its line in the textual UTCTL file.

For illustrating an example for a textual trace, consider the RTCP Overtaking (cf. Figure 3.6) and the MTCTL property **AG** (*stateActive*(Overtaker.Overtaking) *implies* *stateActive*(Overtakee.NoAcceleration));. By default, the RTCP fulfills the given property. Therefore, to enforce a counterexample trace for this property, we change the RTCP as follows: We reduce the clock constraint of the transition originating in state *Overtakee.NoAcceleration* and leading to state *NoOvertaking* from $c \geq 21s + 2 \times \text{maxMsgDelay}$ to 21s. Consequently, the state *NoAcceleration* may now be left while the state *Overtaker.Overtaking* is still active, which leads to the violation of the property. In the following, we show an excerpt of UPPAAL's textual trace when verifying the mentioned property with the incorrect RTCP:

Transitions:

```
muml_overtakee_Role_49_46_1.muml_NoAcceleration_50_EXIT_13_66->
  muml_overtakee_Role_49_46_1.INTERMEDIATE_31 { !
    intermediateLocationSemaphore , exit_muml_NoAcceleration_50_52[
      componentInstanceID]!, intermediateLocationSemaphore++ }
muml_overtakee_noAcceleration_51_55_1.
  muml_Init_52_ACTIVE_IN_muml_overtakee_noAcceleration_51->
  muml_overtakee_noAcceleration_51_55_1.INTERMEDIATE_35 { !
    intermediateLocationSemaphore , exit_muml_NoAcceleration_50_52[
      componentInstanceID]?, trigger_57_19[discretePortInstanceID] -= 1,
    trigger_61_21[discretePortInstanceID] -= 1, trigger_65_23[
      discretePortInstanceID] -= 1, intermediateLocationSemaphore++ }
```

State:

```
( Connector_0_1.Idle Connector_1_0.Idle
  muml_OvertakeeCompRTSC_47_54_component_1.
  muml_Initial_48_ACTIVE_IN_muml_OvertakeeCompRTSC_47
  muml_OvertakerCompRTSC_30_40_component_0.
  muml_Initial_31_ACTIVE_IN_muml_OvertakerCompRTSC_30
  muml_overtakee_Role_49_46_1.INTERMEDIATE_31
  muml_overtakee_noAcceleration_51_55_1.INTERMEDIATE_35
  muml_overtakee_noOvertaking_55_56_1.muml_overtakee_noOvertaking_55_IDLE
  muml_overtaker_Role_32_33_0.
  muml_Overtaking_35_ACTIVE_IN_muml_overtaker_Role_32
  muml_overtaker_noOvertaking_41_41_0.muml_overtaker_noOvertaking_41_IDLE
  muml_overtaker_overtaking_36_36_0.
```

```

    muml_Changed_38_ACTIVE_IN_muml_overtaker_overtaking_36 UrgencyProvider.
    idle )
transmissionTimes[0][0]=21000 transmissionTimes[0][1]=15500
transmissionTimes[0][2]=21000 transmissionTimes[0][3]=21000
transmissionTimes[0][4]=21000 transmissionTimes[1][0]=21000
transmissionTimes[1][1]=21000 transmissionTimes[1][2]=21000
transmissionTimes[1][3]=21000 transmissionTimes[1][4]=21000 muml_c_46_3
[0]=20000 muml_c_46_3[1]=21000 RELATIVE_1_4[0]=20000 RELATIVE_1_4
[1]=21000 muml_c_62_18[0]=21000 muml_c_62_18[1]=21000 connectorOverflow
=0

```

This excerpt shows the last snapshot transition and the last snapshot before UPPAAL detects that the property is violated. In the last snapshot transition, two transitions of different automata synchronize each other. One automaton represents the RTSC Overtakee where the state NoAcceleration is active while the other one represents the RTSC contained by state NoAcceleration where the state Init is active. Due to the synchronization, both automata leave their states and switch to intermediate states. The snapshot shows UPPAAL’s active states of each timed automaton as well as the variable and clock assignments (we only show 17 of the 238 assignments due to space restrictions).

Our forward translation transforms the MTCTL property **AG** (`stateActive(Overtaker.Overtaking) implies stateActive(Overtakee.NoAcceleration)`); into the following UTCTL property:

```

A[]
( muml_overtaker_Role_32_33_0 . muml_overtaker_Role_32_AUX_muml
  _Overtaking_35_muml_Overtaking_35_ENTRY_7 or
  muml_overtaker_Role_32_33_0 . muml_exit3_39_43 or
  muml_overtaker_Role_32_33_0 . muml_Overtaking_35_ACTIVE_IN_muml
    _overtaker_Role_32 )
imply
( muml_overtakee_Role_49_46_1 . muml_NoAcceleration_50_EXIT_13_66 or
  muml_overtakee_Role_49_46_1 . muml_NoAcceleration_50_EXIT_11_62 or
  muml_overtakee_Role_49_46_1 . muml_NoAcceleration_50_EXIT_10_58 or
  muml_overtakee_Role_49_46_1 . muml_overtakee_Role_49_AUX_muml
    _NoAcceleration_50_muml_NoAcceleration_50_ENTRY_9 or
  muml_overtakee_Role_49_46_1 . muml_NoAcceleration_50_ACTIVE_IN_muml
    _overtakee_Role_49 )

```

Therefore, this property is no longer fulfilled because one of the listed overtaker states (i.e., `muml_overtaker_Role_32_33_0.muml_Overtaking_35_ACTIVE_IN_muml_overtaker_Role_32`) is active but none of the five listed states of the overtakee.

5.5.4 Text-to-Model-Transformation

“As defined for Activity 4, we need to parse the textual results of each UTCTL property and—optionally—the textual (counterexample) trace into a model.” [*DGH15] A grammar for UPPAAL’s console output is not available. Therefore, we reverse engineered the grammar based on various UPPAAL examples. We show the resulting grammar in Appendix E.3.5. A brief explanation is as follows: First of all, UPPAAL may output several traces that we collect within a trace repository. A trace first states the property and its verification result. If a trace was generated, then it is printed afterwards by stating a list of trace items. A trace item is either a state (in this thesis, we refer to this item as a snapshot) or a transitions (in this

thesis, we refer to this as a snapshot transition). States and transitions alternate each other. If it is a state, then UPPAAL lists for all processes, which location is currently active and the values of all variables (including clocks). If it is a transition, then UPPAAL distinguishes delay transitions and action transitions. A delay transition may only be shown in concrete traces. It states the delay of the NTA, i.e., how much all clocks progress. An action transition may be shown in concrete and symbolic traces. It consists of one or more edge activities (one per edge that was fired). Each edge activity states the location that was deactivated and the location that was activated. Moreover, each edge activity states additional details: First, it repeats the condition statement of the edge. Second, it states the synchronization statement or 'tau' if no synchronization was executed. Third, it states the update statement of the edge. Note that we do not define grammar rules for the details but handle them as one string as we only require these details to identify which edge was fired (if more than one edge connects the target and the source location).

Our grammar is specified using Xtext [Xtext]. Hereby, we specify references to our UPPAAL timed automata metamodel, which we explain in Appendix E.3.3. Then, we use Xtext to generate a metamodel for our grammar. We show the resulting metamodel in Appendix E.3.6. Like the grammar, the metamodel includes cross-references to the UPPAAL timed automata metamodel. Furthermore, we also define rules in Java for helping the Xtext's parser to identify which model element of the timed automata model needs to be referenced. In particular, these rules specify name matchings. For example, if the parser reads a variable with the name `connectorOverflow`, then we provide the list of all available variables. As our variable names are unique, the parser will identify the corresponding model element and reference it within the trace model. We provide this name matching for the following timed automata model elements: processes (an instance of a timed automaton template), states, clocks, and variables.

In the following, we show an exemplary UPPAAL trace model in its serialized form, which references the serialized UPPAAL model (here it has the name `OvertakingCIC.uppaal`). This model is the result when we parse the counterexample trace that we show in Section 5.5.3, which was generated for the incorrect RTCP Overtaking, which we introduce in the same section. In particular, we show an excerpt of the last snapshot transition and the last snapshot of the trace.

```
<traceItems xsi:type="diagnostictrace:ActionTransition" xmi:id="
  _2DB7XSMKEeeazvuJsUXzLQ">
  <edgeActivities xmi:id="_2DB7XiMKEeeazvuJsUXzLQ" details="{_!
    intermediateLocationSemaphore ,_exit_muml_NoAcceleration_50_52[
      componentInstanceID ]! ,_intermediateLocationSemaphore++_}">
    <source xmi:id="_2DB7XyMKEeeazvuJsUXzLQ">
      <process xmi:id="_2DB7YCMKEeeazvuJsUXzLQ">
        <template xsi:type="templates:RedefinedTemplate" href="
          OvertakingCIC.uppaal @systemDeclarations / @declaration.5 /
            @declaredTemplate"/>
        </process>
        <location href="OvertakingCIC.uppaal @template.6 / @location.11"/>
      </source>
      <target xmi:id="_2DB7YSMKEeeazvuJsUXzLQ">
        <process xmi:id="_2DB7YiMKEeeazvuJsUXzLQ">
          <template xsi:type="templates:RedefinedTemplate" href="
            OvertakingCIC.uppaal @systemDeclarations / @declaration.5 /
              @declaredTemplate"/>
          </process>
```

```
<location href="OvertakingCIC.uppaal###@template.6/@location.19"/>
</target>
</edgeActivities>
...
</traceItems>

<traceItems xsi:type="diagnostictrace:State" xmi:id="
  _2DB7aCMKEeeazvuJsUXzLQ">
  <locationActivities xmi:id="_2DB7aSMKEeeazvuJsUXzLQ">
    <process xmi:id="_2DB7aiMKEeeazvuJsUXzLQ">
      <template xsi:type="templates:RedefinedTemplate" href="OvertakingCIC
        .uppaal###@systemDeclarations/@declaration.8/@declaredTemplate"/>
    </process>
    <location href="OvertakingCIC.uppaal###@template.0/@location.0"/>
  </locationActivities>
  ...
  <variableValues xmi:id="_2DB7lCMKEeeazvuJsUXzLQ" value="21000">
    <firstNamedElementReference xmi:id="_2DB7lSMKEeeazvuJsUXzLQ">
      <singleNamedElementReference xmi:id="_2DB7liMKEeeazvuJsUXzLQ">
        <namedElement xsi:type="declarations:Variable" href="OvertakingCIC
          .uppaal###@globalDeclarations/@declaration.38/@variable.0"/>
        <indexes>1</indexes>
        <indexes>2</indexes>
      </singleNamedElementReference>
    </firstNamedElementReference>
  </variableValues>
  ...
</traceItems>
```

5.6 Model-To-Model Back-Translation from Uppaal to RTCPs

“According to Activity 5 in Figure 5.1, our back-translation has two tasks: First, it has to translate the verification results of all UTCTL properties back to the MTCTL properties. We explain this in Section 5.6.1. Second, if UPPAAL generated a (counterexample) trace, it has to translate it back to the level of MECHATRONICUML as well. This will be explained in Section 5.6.2.” [*DGH15]

“In both tasks, we resolve the traceability links that were automatically generated by QVTo within our multi-step forward translation.” [*DGH15] As already stated in Section 5.3, we execute the resolving by using QVTo’s *invresolve* operation [QVT], which returns all sources of a given model element. Therefore, we call *invresolve* on all elements of the model checker trace that we want to translate back to the DSL, i.e., processes, states, variable values, and clock values. *invresolve* only resolves one forward translation step back. Therefore, we recursively execute *invresolve* until we receive an element of the initial design model. Based on these information, we can build up [a] trace model at the level of the DSL. Furthermore, we also use QVTo’s *resolve* operation [QVT] recursively if we want to retrieve the UPPAAL model elements for a given source model element. “As a result, we can implement the [model-to-model] back-translation of a complete (counterexample) trace and of the verification results in one single step despite the 13 [model-to-model] steps from MECHATRONICUML to UPPAAL. In particular, our back-translation is very small as it needs only 609 QVTo LOCs,

compared to 9,239 LOCs for the translation from MECHATRONICUML to UPPAAL (ca. 7%). Moreover, our implementation is in most cases straightforward as we only have to define to which kind of MECHATRONICUML element a particular UPPAAL element needs to be transformed.” [*DGH15]

Note that we always identify exactly one element in the original RTCP model when using invresolve recursively. We achieve this as we define that our forward translation has the following limitations: (1) All RTCP model elements that we want to back-translate later on require an uninterrupted path of traceability links to one or more model elements of the UPPAAL model. (2) An UPPAAL model element may not belong to two or more RTCP model elements. Otherwise, it could be the case that we find none or more than one RTCP model element when back-translating an UPPAAL model element.

5.6.1 UTCTL to MTCTL

“For back-translating the UTCTL properties to MTCTL, we have to consider the case that we have to combine the result of multiple UTCTL properties into one MTCTL property result. This case exists because the “MTCTL Split Properties Translation” (see Section 5.4.4) may split an MTCTL property into multiple MTCTL properties while removing logical operators.

Our concept for this back-translation is as follows: After Activity [4], the UPPAAL output model contains the result (true or false) of each UTCTL property. By resolving the traceability links of the forward translation, we get for each UTCTL property the corresponding MTCTL property (we call it the “split property”) after the “MTCTL Split Properties Translation”. Next, we assign to each split property its result. Then, by resolving the traceability links again, we can match each split property to the MTCTL property that existed before the “MTCTL Split Properties Translation”. Thus, we now know for each sub-expression of these properties its result. [We] apply our “MTCTL Static Evaluation Normalization” (see Section 5.4.4) and simplify the complete property to one truth value (true or false). Finally, we can relate the truth value of each property to the original MTCTL property that the [DSMC user] specified initially by resolving the traceability links a last time.” [*DGH15] We store the results of each MTCTL property within MTCTL’s metamodel (see Appendix E.2.1), which provides the package results (see Figure E.17) for this purpose.

“Noteworthy, the effects of the two steps “MTCTL Set Quantifier Normalization” (see Section 5.4.2) and “MTCTL Static Evaluation Normalization” (see Section 5.4.3) of the forward translation do not require special treatments in the back-translation. The reason is that both are normalizations that only express the [MTCTL property] differently without changing its semantics.” [*DGH15]

We illustrate our concept by reusing the example of Section 5.4.4 which had two MTCTL properties as its result: **EF** *messageInTransit*(accept); and **AG** *messageInTransit*(decline) and *stateActive*(Overtaker.noOvertaking.Init);. “[Both] belong to the property **not** (**EF** *messageInTransit*(accept)) **or** (**AG** (*messageInTransit*(decline) and *stateActive*(Overtaker.NoOvertaking.Init)))”. After the model checking, we know the results of both split properties. For example, let us assume that the first property is **true** and the second is **false**. Then, we can insert these values into the original values. As a result, the combined property is: **not** (**true**) **or** (**false**);. After the static evaluation, we can simplify it to **false**. Consequently, the property that the [DSMC user] stated is **false** as well.” [*DGH15]

5.6.2 Uppaal Trace to MechatronicUML Trace

“As a prerequisite for the back-translation of UPPAAL traces, we require a [metamodel] at the level of MECHATRONICUML to specify traces.” [*DGH15] The metamodel of RTCPs (see Appendix E.1) is not sufficient as it is not designed to store runtime information, e.g., the value of a variable in a particular snapshot. Therefore, we use a separate metamodel for storing traces but reference the RTCP metamodel such that a MECHATRONICUML trace model can reference the RTCP model (the input of our DSMC).

We already defined a metamodel for storing traces and runtime information in [*HBDS15] to realize the second step of our compositional verification approach (see Section 2.2.1). Therefore, we only have to extend it slightly, e.g., to store clock values and to describe a snapshot of a component instance configuration. We show and explain the resulting metamodel in Appendix E.3.7.

Our MECHATRONICUML metamodel is similar to the UPPAAL trace metamodel (see Appendix E.3.6), i.e., it provides means to model a sequence of snapshots. However, MECHATRONICUML snapshots provide specific information of the MECHATRONICUML design model at runtime while the UPPAAL trace only provides specific information about the UPPAAL model. For example, the MECHATRONICUML trace defines which state of an RTSC is active in which port instance of which component instance. In contrary, an UPPAAL trace defines which state of a timed automaton process, which belongs to a timed automaton template, is active.

In the following, we explain the concept of our model-to-model back-translation of traces. As an example, we reuse our incorrect variant of RTCP Overtaking, which we introduced in Section 5.5.3, and the MTCTL property **AG** (*stateActive*(Overtaker.Overtaking) *implies* *stateActive*(Overtakee.NoAcceleration)); to produce a counterexample trace. We illustrate the back-translation with Figure 5.24, which sketches the back-translation of snapshots in the middle and focuses on the back-translation of the last snapshot of the trace. Moreover, in Appendix D.2, we show the complete counterexample trace visualized as an SVG. We explain the elements of our SVG representation within Section 5.7.1.

Our model-to-model back-translation concept is as follows: First of all, we translate each snapshot of UPPAAL to exactly one MECHATRONICUML snapshot and each snapshot transition, which stores the cause of the snapshot change, i.e., action or delay, to exactly one MECHATRONICUML snapshot transition. For example, Figure 5.24 sketches on the right side an UPPAAL trace consisting of 30 snapshots and on the left side a MECHATRONICUML trace consisting of 30 snapshots as well. We depict that each snapshot and each snapshot transition is translated back via green arrows.

Each MECHATRONICUML snapshot does not describe an RTCP at runtime but the runtime representation of a CIC. The reason for this is as follows: “As defined in Section 5.4.1, we translate an RTCP into a CIC as our DSMC shall support the model checking of components in the future as well (for realizing the third step of our compositional verification approach [see Section 2.2.1]). Therefore, we translate each UPPAAL snapshot into a CIC runtime model in order to reuse this back-translation for the model checking of components. Though, this [has no significant] negative influence to the analyzability of the (counterexample) trace because an RTCP at runtime and a CIC at runtime are very similar and as the RTSCs, the buffers, and the connector instances of [a CIC and an RTCP] are identical.” [*DGH15] For example, the RTCP Overtaking is of kind one-to-one. Therefore, we translate it into a CIC

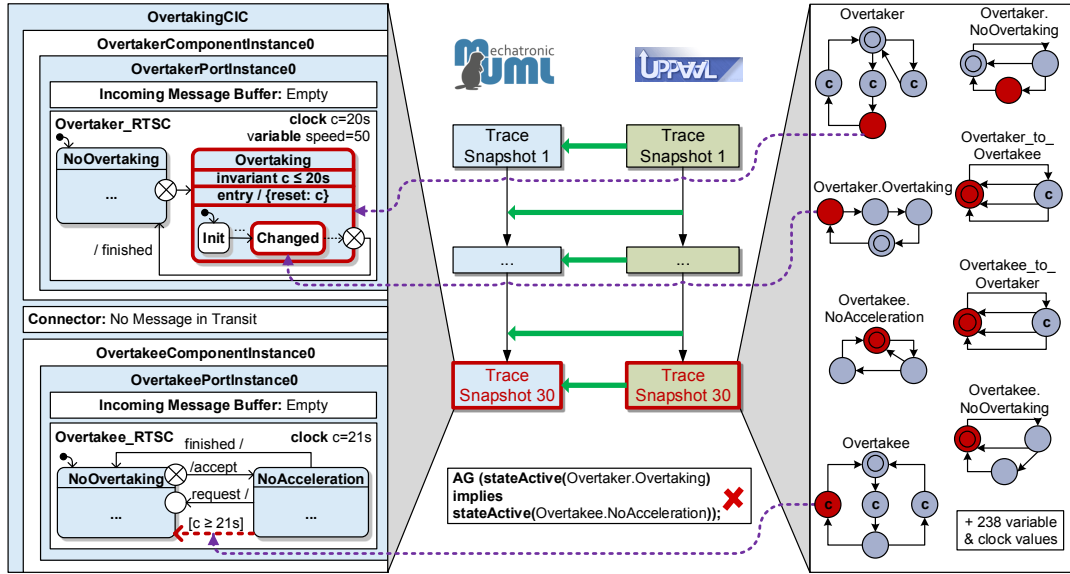


Figure 5.24: Back-Translation of an UPPAAL Counterexample Trace to MECHATRONICUML (Based on [*DGH15])

that consists of two component instances with one port instance each and a connector instance that connects these two port instances (see Section 5.4.1). Consequently, each snapshot of our MECHATRONICUML trace contains this structure. For example, Figure 5.24 shows on the left side the CIC structure of a MECHATRONICUML snapshot of RTCP Overtaking: The root element is the CIC of the overtaking scenario. The CIC contains two component instances (one for the overtaker, another one for the overtakee), whereby each component instance contains exactly one port instance. Moreover, the CIC contains the connector that connects the two component instances with each other.

For each snapshot, we translate all relevant runtime information back. These are:

- the active state (resp. its entry or exit point),
- the active (time-consuming) transition of each role RTSC,
- the values of all non-constant role RTSC variables,
- the values of all clocks of the role RTSC,
- the state of each incoming message buffer: the messages and their parameter values that each buffer currently stores, and
- the state of each port instance connector: the messages and their parameter values in transit (i.e., in the connector) including their source and destination port instance.

Figure 5.24 sketches several runtime information within the last snapshot of the trace for both languages. The correspondence (i.e., the traceability paths) between the model elements is partially highlighted by means of purple dashed arrows. Concerning the UPPAAL snapshot, we show the active state for each timed automaton in red. The UPPAAL model consists of 238 variable and clock values, whereby each snapshot defines the values of each variable and each clock. Using our back-translation, we can retrieve the relevant runtime information for each MECHATRONICUML snapshot. In particular, the last MECHATRONICUML snapshot shows that the state `Overtaker.Overtaking.Changed` and the non-urgent

transition from `Overtakee.NoAcceleration` to `NoOvertaking`, which specifies the too early clock constraint, are both active. Moreover, this snapshot shows that the incoming message buffers are empty, no message is in transit, that the clock `Overtaker.c` is at 20s, that the variable `Overtaker.speed` is 50, and that the clock `Overtaker.c` is at 21s. Note that this is the last snapshot of the counterexample trace as it violates the aforementioned MTCTL property.

In the following, we explain how we translate each runtime information back to the level of MECHATRONICUML:

Active State We can retrieve whether a state of an RTSC instance is active by resolving the states that the UPPAAL snapshot lists as active. If any of the resolved UPPAAL states is a state of this RTSC instance, then we can mark this RTSC state as active. In particular, we can find out whether a state or one of its entry or exit points is active.

Active Transition We can retrieve whether a transition of an RTSC instance is active by resolving the states that the UPPAAL snapshot lists as active. This is possible as we insert an intermediate state for all RTSC transitions (either during the RTSC deadline normalization, which we describe in Section 5.4.7, or during the migration to UPPAAL, which we describe in Section 5.4.13.2). If any of the resolved UPPAAL states is a transition of this RTSC instance, then we can mark this RTSC transition as active.

Variable and Clock Value We can retrieve the value of the variables and clocks of an RTSC instance by resolving the variables/clocks of the UPPAAL snapshot. This is possible as we transform RTSC variables and clocks directly to UPPAAL variables and clocks during the migration to UPPAAL (Section 5.4.13.2). The resolved UPPAAL variables and clocks that belong to this RTSC instance provide the values of the variables and clocks of this RTSC.

Buffer State We can retrieve the current messages (and parameter values) of a buffer of a port instance by resolving the variables of the UPPAAL snapshot. This is possible as we transform buffers to array variables during the migration to UPPAAL (Section 5.4.13.3).

Connector State We can retrieve the current messages (and parameter values) of a connector instance as well as the source and the target port instance by resolving the variables of the UPPAAL snapshot. This is possible as we transform connectors to array variables during the migration to UPPAAL (Section 5.4.13.3).

As we translate each UPPAAL snapshot back, we do not only get snapshots where the role RTSCs change but we also receive snapshots where only the asynchronous message connector or the message buffers change. Moreover, we receive multiple snapshots when a composite transition fires or a hierarchical RTSC gets (de-)activated as we normalize these elements during our forward translation into several states and transitions. For example, firing the transition from state `Overtaker.NoOvertaking.Idle` to `WaitingForAnswer` produces four snapshots (compare the snapshots 2–5 in Figure D.11 on page 352):

- The first snapshot shows that the transition is fired and the message is put into the connector.
- The second snapshot shows that the state `WaitingForAnswer` is activated.
- The third snapshot shows that the message is taken from the connector.

- The fourth snapshot shows that the message is put into the incoming message buffer of the role Overtakee.

In our opinion, the third snapshot must not necessarily be shown to the DSMC user as nothing interesting happens. Though, deciding which snapshot is interesting may be subjective. Hence, we currently do not hide a snapshot or merge several snapshots into one.

5.7 Implementation

In this section, we present excerpts of our prototypical implementation that covers the complete DSMC for RTCPs via UPPAAL. Our implementation is integrated into the MECHATRONIC-UML Tool Suite [*DGB+14].

In particular, we present the user interface in Section 5.7.1 and the plugin structure in Section 5.7.2. We provide the documentation of our metamodels and grammars in Appendix E.3.

5.7.1 User Interface

The DSMC user can start our DSMC in two different modes (see Figure 5.25). The first mode is called “Verify Multiple Properties”. Here, the DSMC user can verify a set of multiple properties that he may select out of all available MTCTL properties for this RTCP. At the level of UPPAAL, all selected properties will be verified. However, no trace is generated in order to save verification time. As a result, the DSMC user is informed concerning the fulfillment of each property.

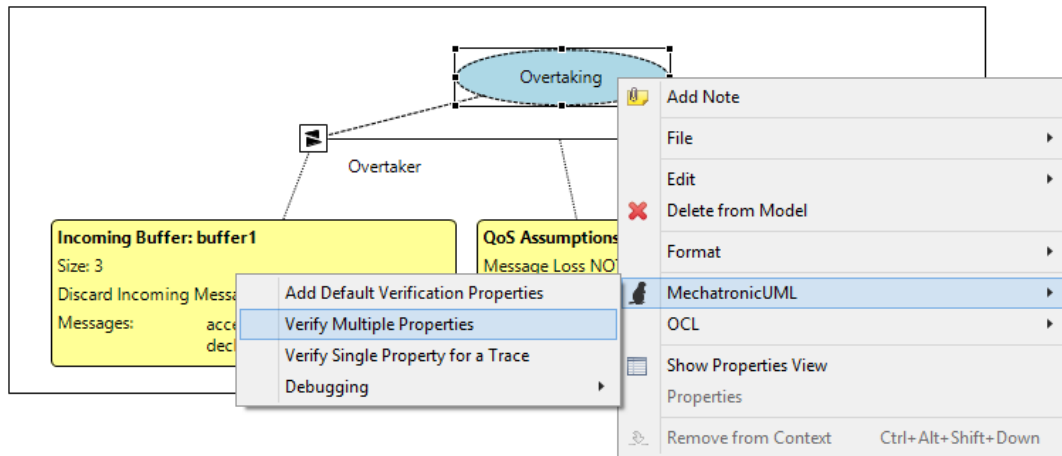


Figure 5.25: Selecting the Verification Mode

The second mode is called “Verify Single Property for a Trace”. Here, the DSMC user can verify a single atomic property (i.e., a property that does not contain set quantifiers and, therefore, results in exactly one UTCTL property). At the level of UPPAAL, the single property is verified and a trace will be generated (if the safety/liveness property is unfulfilled or the reachability property is fulfilled). Then, the trace of this property is translated back to MECH-

ATRONICUML. As a result, the DSMC user is informed whether the property is fulfilled and gets—if available—an SVG representation of the (counterexample) trace.

After the DSMC user selects a DSMC mode, he may change the default verification options, e.g., the role multiplicity, within the first page of our wizard (see Figure 5.26). In the second page of our wizard, the DSMC user has to choose the MTCTL properties to be verified (see Figure 5.27). Note that we state the MTCTL property as an English sentence into a tool tip box if the mouse cursor hovers over a property. By doing this, we aim to support the DSMC user as he might have less experience concerning TCTL-like languages (e.g., he might be an MTCTL reader). Therefore, we lower the probability that he misunderstands the available properties and, thus, selects the wrong ones for the DSMC.

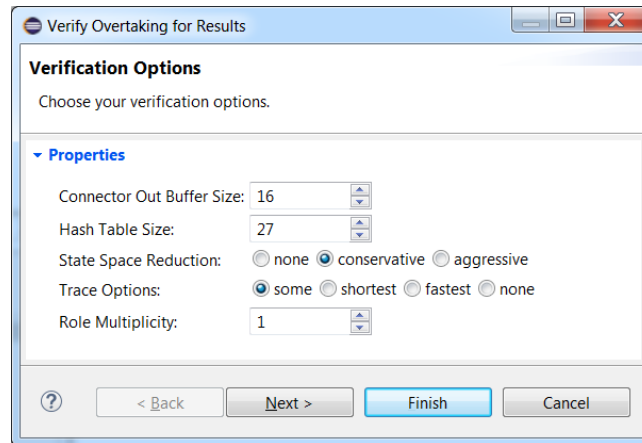


Figure 5.26: User Interface for Specifying DSMC Options

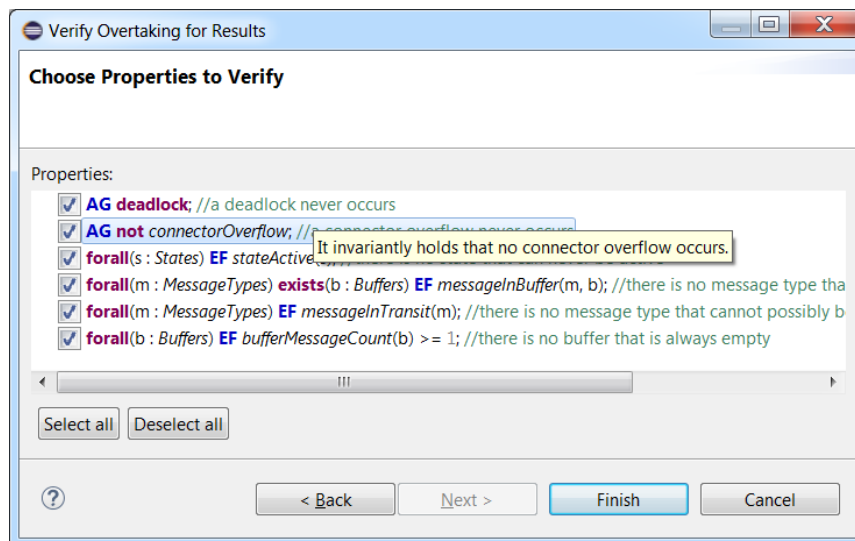


Figure 5.27: User Interface for Selecting MTCTL Properties

In addition, we also support two debugging modes for the DSMC developer: First, we enable to execute the mode “Verify Multiple Properties” but additionally store all intermediate

models and all intermediate files of the forward and backward translation. Second, we enable to execute the mode “Verify Single Property for a Trace” but additionally store all intermediate models and all intermediate files of the forward and backward translation. Therefore, the DSMC developer can (1) study the result after each transformation (model-to-model, text-to-model, model-to-text) in detail within the MECHATRONICUML Tool Suite, (2) use the model files for debugging a single QVTo transformation, which requires these models as inputs, and (3) inspect the generated UPPAAL text files within UPPAAL’s developer suite, e.g., to use its step by step simulator in order to debug an RTCP at the level of UPPAAL.

After the verification, we present the results to the DSMC user. Concerning mode “Verify Multiple Properties”, we present all verified properties (see Figure 5.28) and mark the fulfilled ones in green and the not fulfilled ones in red. In addition, if we had to split properties, the DSMC user may also see the result for each “split property”. For example, within the figure, we show the results of the split properties of property **forall**(m : **MessageTypes**) **EF** **messageInTransit**(m); Moreover, we aim to facilitate the job of the DSMC user as we provide an English sentence that expresses the result of the property. Hereby, our goal is that the DSMC user understands the result of the verified properties correctly. Again, we show this sentence into a tool tip box if the mouse cursor hovers over a property. Noteworthy, as we already explained in Section 4.5, if the property is not fulfilled, then we provide a sentence that negates the content of the property completely to state what holds for this RTCP.

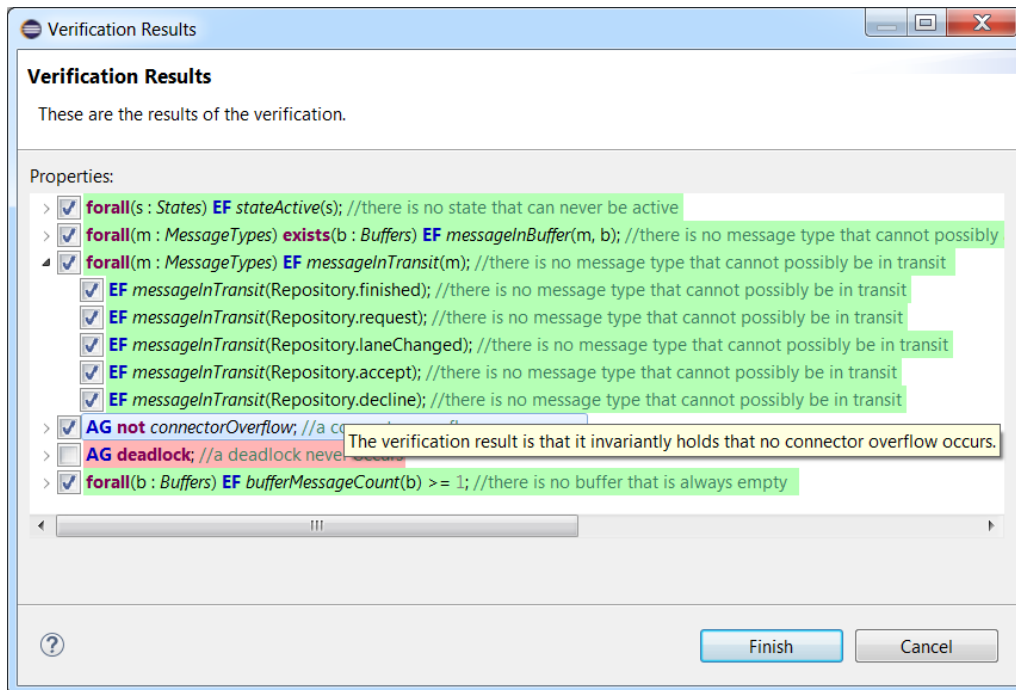


Figure 5.28: Result Presentation

In the case of mode “Verify Single Property for a Trace”, the DSMC user may open the generated SVG file of a (counterexample) trace if UPPAAL was able to produce it (UPPAAL only produces a trace if a safety or liveness property is not fulfilled or if a reachability property is fulfilled). We show a complete counterexample trace in its SVG representation for an incorrect variant of RTCP Overtaking in Appendix D.2. Within this section, we depict in

Figure 5.29 the last snapshot transition and the last snapshot of this trace. In addition, we explain the contents of the snapshot and its snapshot transition using white boxes.

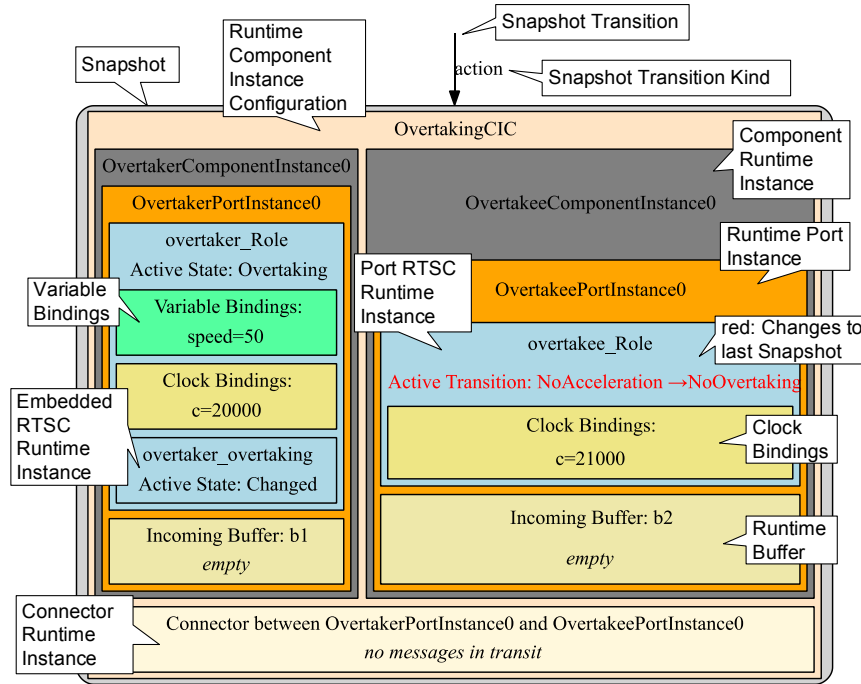


Figure 5.29: SVG Representation of a MECHATRONICUML Snapshot (Based on [BCD+14])

Based on [BCD+14], A detailed explanation for our SVG representation is as follows: We illustrate a snapshot transition as an arrow that points from one snapshot to its successor. Moreover, we annotate at each snapshot transition whether it is of kind *action* or *delay*. Concerning the snapshot, we choose the style of hierarchical boxes because it matches the hierarchical models of MECHATRONICUML. The most outer box (colored gray) is the snapshot itself. Within this box is another box (colored light orange) that represents the runtime version of the CIC to verify (OvertakingCIC). It contains boxes representing runtime versions of the contained component instances (colored dark gray) and the connector (colored light yellow). In our case, we have two component instances (OvertakerComponentInstance0 and OvertakeeComponentInstance0) and one connector between their ports that does not contain a message at the moment. Both runtime component instances may contain one or more port instances. As RTCP Overtaking is of kind one-to-one, we have one runtime port instance per runtime component instance: OvertakerPortInstance0 and OvertakeePortInstance0. A runtime port instance contains one runtime port RTSC instance and may contain runtime buffer instances. In our case, OvertakerPortInstance0 contains the runtime port RTSC instance *overtaker_Role* and the runtime buffer instance *b1*, which is currently empty. OvertakeePortInstance0 contains the runtime port RTSC instance *overtakee_Role* and the runtime buffer instance *b2*, which is currently empty as well. A runtime RTSC instance has an active state or an active transition and may contain variable bindings, clock bindings, and embedded runtime RTSC instances. In our case, the active state in *overtaker_Role* is *Overtaking*, the value of variable *speed* is 50, the clock *c* is 20000. Moreover, *overtaker_Role*

contains the embedded runtime RTSC instance `overtaker_overtaking`, where the state `Changed` is active. Concerning `overtakee_Role`, the transition from state `NoAcceleration` to `NoOvertaking` is active and the clock `c` is 21000. When transferring this information to our RTCP Overtaking, we can easily infer that the state `Overtaking.Changed` is active in role `Overtaker` while the transition from `NoAcceleration` to `NoOvertaking` is active in role `Overtakee`. Consequently, the MTCTL property **AG** (`stateActive(Overtaker.Overtaking)` *implies* `stateActive(Overtakee.NoAcceleration)`); is not fulfilled within the snapshot.

Noteworthy, our SVG representation does not visualize the active state of the component RTSCs as they always have only one active state that is always active. Furthermore, like UPPAAL, we do not show constant variables within the trace.

5.7.2 Plugin Structure

We show an excerpt of the plugins that we developed and adapted for our prototypical implementation in Figure 5.30. First of all, the plugin transformation is the core of our implementation as it contains all QVTo files that we use for realizing Activity 1 and 5 of our approach (see Figure 5.1).

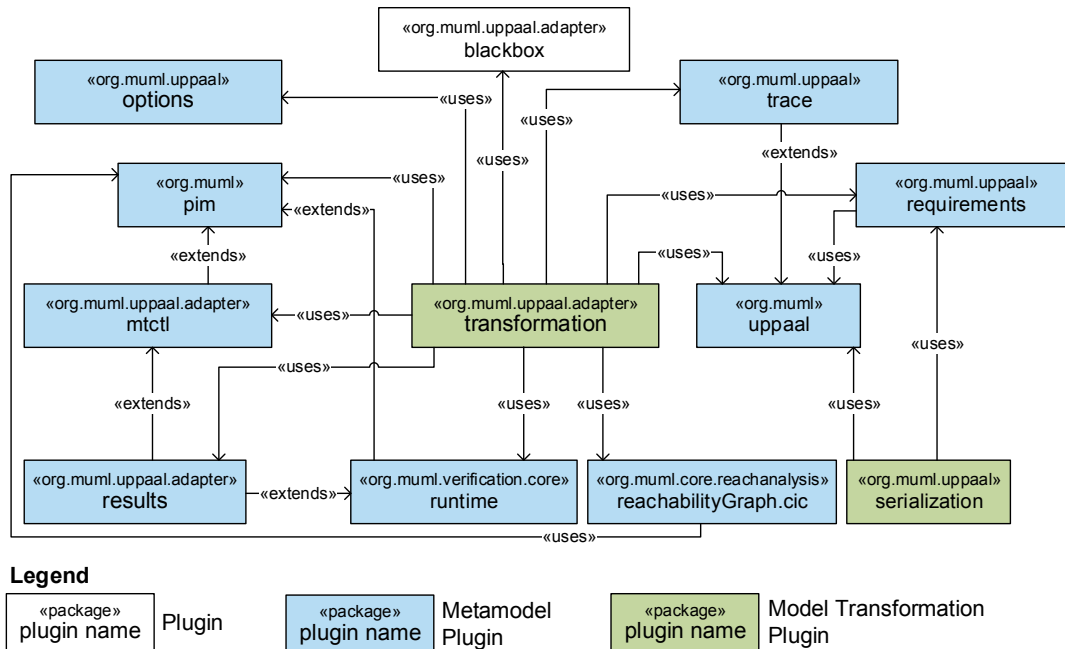


Figure 5.30: Plugins used for implementing the DSMC for RTCPs via UPPAAL

The inputs for Activity 1 are models of the metamodels for the platform-independent part of MECHATRONICUML (including RTCPs, RTSCs, and CICs), for MTCTL, and for the user-defined verification options concerning the DSMC. We define these metamodels in the plugins `pim`, `mtctl`, and `options` (the latter one contains a metamodel and graphical user interface for the verification options). The output of Activity 1 are models of the metamodels for UPPAAL timed automata and UTCTL, which are defined in the plugins `uppaal` and `requirements`.

During the complete DSMC, the plugin transformation uses the plugin `blackbox` for all tasks that may not be realized via QVTo but other technologies, e.g., Java. For simplicity reasons,

we do not show all plugin references of this plugin. Among others, the plugin blackbox executes the Activities 2, 3, and 4. In particular, for Activity 2, it starts the model-to-text transformation from the UPPAAL model to UPPAAL XML, which is defined in the plugin serialization. Then, the plugin blackbox automates UPPAAL in Activity 3. Afterwards, in Activity 4, the plugin blackbox starts the parsing of the textual outputs from UPPAAL and executes the text-to-model transformation. The result is a (counterexample) trace model. The parser, the text-to-model-transformation, as well as the metamodel for trace models are defined in the plugin trace.

As already stated above, the QVTo files for the back-translation (Activity 5) are contained in the plugin transformation as well. The input for the back-translation are the UPPAAL model, the UTCTL model, and the UPPAAL trace model. The output is on the one hand the results of each MTCTL property, which respects the metamodel that we defined in the plugin results. On the other hand, a second output may be available: the (counterexample) trace model at the level of MECHATRONICUML. These models are based on two metamodels: The first metamodel defines the snapshots and the snapshot transitions of a CIC trace. We define it in the plugin `reachabilityGraph.cic`. The second metamodel defines the contents of a snapshot, e.g., the active states and the messages in a buffer. We define it in the plugin runtime.

5.8 Evaluation

We conduct a case study based on the guidelines by Kitchenham et al. [KPP95] and Runeson et al. [RHRR12] for evaluating our approach. In particular, we evaluate our DSMC for realistic examples within the domain of CPSs but do not aim for generalizations. Analogous to our previous case studies, all examples of this case study are defined by ourselves or colleagues and are not taken from real industry examples. We present the core points of our evaluation in this section and give more details in Appendix C.3.

Our evaluation is structured as follows: First, we present the context of our case study in Section 5.8.1. Then, we set our hypotheses in Section 5.8.2 and prepare the validation in Section 5.8.3. Afterwards, we validate the hypotheses in Section 5.8.4. Finally, we discuss the threats to validity in Section 5.8.5 and analyze our results in Section 5.8.6.

5.8.1 Case Study Context

The first objective of our case study is to evaluate whether we fulfill the requirements R5 and R6 that we stated in Section 5.2. Therefore, this case study shall investigate whether our DSMC for RTCPs using the UPPAAL model checker (which is based on our general DSMC approach) is correct w.r.t. our concept. The second objective of this case study is to evaluate the performance of our DSMC approach (the translation, the model checking, and the back-translation). Our third objective is to evaluate whether RTCPs encapsulate the specification of a coordination protocol more effectively than the input language of UPPAAL.

“We conduct our case study using [16] existing RTCPs of different interconnected transportation systems (e.g., cars, railways, robots). These RTCPs focus on various use cases for coordination, such as collision avoidance or our overtaking scenario.” [*DGH15]

5.8.2 Setting the Hypotheses

Our case study has six evaluation hypotheses. The first three tackle the first objective (the correctness of the DSMC):

- H1** “Our DSMC approach correctly translates all RTCPs including all MTCTL properties to UPPAAL.” [*DGH15]
- H2** Our DSMC approach correctly translates all UPPAAL verification results (i.e., the fulfillment of each property) back to MECHATRONICUML.
- H3** Our DSMC approach correctly translates all traces of UPPAAL back to MECHATRONICUML.

Moreover, for a realistic RTCP with a role multiplicity of at most 2 and a standard computer, we additionally define 2 hypotheses to evaluate the performance of our DSMC:

- H4** When verifying MTCTL’s default properties, our DSMC approach is in most cases able to return a verification result within less than 1 minute.
- H5** When verifying a single MTCTL property, our DSMC approach is in most cases able to return a trace within less than 1 minute.

Our last hypothesis tackles our third evaluation objective:

- H6** An RTCP enables a more compact model representation of coordination protocols including verification properties than the input language of UPPAAL.

For evaluating hypothesis H1, we collect a set of pre-existing RTCPs that are correct w.r.t. a given set of MTCTL properties. We aim to cover a huge range of the expressiveness of RTCPs and MTCTL to increase the meaningfulness of our evaluation. We are able to determine the correctness of these RTCPs as we (a colleague and I) are experts of MECHATRONICUML and MTCTL. Then, we execute the forward translation of our DSMC for all RTCPs with all MTCTL properties and check whether all models are correct. Afterwards, we execute UPPAAL, and check whether all properties are fulfilled at the level of UPPAAL. We expect that this is always the case.

For evaluating hypothesis H2, we reuse our set of RTCPs and our set of MTCTL properties that we collected for evaluating hypothesis H1. Then, we execute our DSMC, which returns the results at the level of our DSL. We check whether all intermediate models are correct and whether all property results are correct at the level of MECHATRONICUML. We expect that this holds for all properties.

For evaluating hypothesis H3, we reuse our set of RTCPs and our set of MTCTL properties that we collected for evaluating hypothesis H1. From the set of MTCTL properties, we collect one MTCTL property (or a split property) per RTCP that states a fulfilled reachability property. Therefore, UPPAAL should be able to produce a trace that shows why this property is fulfilled. For consistency reasons, we always use the same kind of property. We execute our DSMC for each RTCP with its respective reachability property such that UPPAAL produces a trace. Then, we check whether all intermediate models are correct and whether the trace is correct. We expect that this is always the case.

For evaluating hypothesis H4 and H5, we reuse our set of RTCPs that we already use to evaluate H1. Concerning H4, we choose the set of MTCTL's default properties because they are valid properties for all of RTCPs. Therefore, this is a relevant performance use case. Concerning H5, we reuse the reachability properties that we define for evaluating H3. We expect that the hypotheses H4 and H5 are fulfilled.

For evaluating hypotheses H6, we reuse our set of RTCPs that we already use to evaluate H1 and specify MTCTL's default properties for each RTCP. We execute the forward translation of our DSMC for generating two models: the single RTCP EMF model and the EMF representation of the UPPAAL model. EMF has a built-in serialization to XML. We choose the number of XML elements (i.e., the number of nodes of the XML tree) of each serialization as our comparison metric because EMF serializes each model element (e.g., a state) into one XML element. We expect that the EMF serialization of a single RTCP has always less XML elements than the UPPAAL EMF serialization.

5.8.3 Preparing the Validation

For our evaluation, we consciously select 16 RTCPs from four different systems from the domain of CPS. All of the RTCPs were defined before this evaluation has been defined. Two of them are the RTCPs Overtaking and MultiOvertaking that we introduced in Section 3.5.1 and 3.5.2.4. Both RTCPs were defined by me. The other RTCPs are Navigation, Delegation, AllPositionTransmission, PositionTransmission, Distribution, Overtake, ChangeSection, Allow, VModeControl, Delegate, LimitVelocity, Inform, EnterSection, and NextSectionFree. We present them in Appendix C.1. We state in the appendix who defined each RTCP. In short, I was involved in designing 9 of the 16 RTCPs. Moreover, only the RTCPs Overtaker and MultiOvertaker influenced our DSMC concept. Therefore, 14 RTCPs have been designed independently of our concept.

Currently, our DSMC does not support RTCPs with one-to-many communication schemata. Therefore, we normalize the RTCP MultiOvertaking to the RTCP MultiOvertaking_wSubroleRegion that does not use one-to-many communication schemata anymore but an explicit subrole region (see Figure 3.17 on page 77).

Noteworthy, the appendix also presents the RTCPs ConvoyCoordination and ProfileDistribution. Though, these two RTCPs contain model elements that we currently do not support: ConvoyCoordination uses the data type Role and both RTCPs use the data type enum. The RTCP MultiOvertaking is also not fully supported by our DSMC as it uses transition priorities at state Overtaker.Main.subrole.Subrole_Main.Receiving. Though, in order to evaluate hypothesis H1 with this RTCP, we will manually change the resulting UPPAAL XML file. However, this missing feature of our DSMC hinders us to evaluate the hypotheses H2 to H5 with this RTCP.

The RTCPs cover a huge range of the expressiveness of RTCPs. In particular, they cover the following language properties:

- in-roles, out-roles, and in/out-roles
- roles with 0, 1, or 2 buffers
- buffers with a size of 1, 3, or 5
- buffers that shall never discard a message, discard the incoming message, or discard the oldest message in buffer

- buffers that may store 1, 2, 3, or 5 messages
- buffers that may store messages with and without parameters
- reliable and unreliable role connectors
- role connectors that preserve the message order
- various maximum message delays (0, 500 ms, 1s, ...)
- one-to-one RTCPs and one-to-many RTCPs with a role multiplicity of 1 and 2
- Various RTSCs:
 - flat and hierarchical RTSCs
 - variables and clocks in different hierarchy levels
 - zero, one, and more than one variable per RTSC
 - variables of different primitive types (boolean, int8, int32, ...) and user-defined types (e.g., an array of int32)
 - various variable kinds (default, constant, with initialize expression)
 - zero, one, and more than one clock per RTSC
 - varying number of states and transitions per RTSC
 - various state properties (initial, urgent, with invariant, with entry-event, with do-event, with exit-event)
 - various transition kinds (urgent and non-urgent, simple and composite, with entry or exit points as source or target)
 - varying number of entry and exit points
 - various transitions labels (only conditions, only effects, conditions and effects, with and without deadlines, ...)
 - all parts of the transition's conditions (synchronization, guard, clock constraint, receiving message)
 - all parts of the transition's effects (consuming a message, actions, clock resets, sending a message)

However, the following model properties are not covered:

- operations and operation repositories
- role connectors that do not preserve the message order (not implemented yet)
- transition priorities (not supported in our DSMC)
- region priorities (not supported in our DSMC)
- variables with data type Role (not supported in our DSMC)
- final states (not supported in our DSMC)
- variables with primitive type double (not supported in our DSMC)
- the complex data types enum and struct (not supported in our DSMC)

In order to prepare H1 and H2, we define MTCTL's default properties (see Section 4.3) for each RTCP. Moreover, for the RTCPs Overtaking and MultiOvertaking, we also use the additional MTCTL properties that we stated during our evaluation of MTCTL (see Section 4.7 and Appendix C.2). Hereby, we reformulate the four properties 24 to 27 that we specify for RTCP MultiOvertaking in Table C.5 on page 338 such that we can state the properties for RTCP MultiOvertaking_wSubroleRegion.

This list of MTCTL properties covers a huge range of MTCTL's expressiveness (see Section 4.7.4 for a list of covered model elements). Therefore, we consider this as an

appropriate test set for this evaluation of the DSMC. The set of 16 RTCPs contains three of the kind one-to-many. They all define a role multiplicity range of 1 to 2. We manually check whether all RTCPs (with all multiplicities) are correct w.r.t. their MTCTL properties and conclude that this is the case.

In order to prepare H3, we define for each RTCP an MTCTL reachability property that checks the reachability of the state that is activated after receiving the first message. This is an appropriate property as it always produces a trace that contains at least one firing transition per role, one state change per role, one message sending, one message transmission, one message buffering, and one message receiving. Moreover, in most cases, the trace also contains a progression of at least one clock. However, in order to formulate this property using MTCTL, we have to slightly adapt three of the RTCPs by inserting one intermediate state at the role that receives the message. The three roles that we change are (1) the role Section of RTCP ChangeSection, (2) the role Receiver of RTCP PositionTransmission, and (3) the role Receiver of RTCP AllPositionsTransmission. For example, we replace the self-transition of role Section of RTCP ChangeSection (see Appendix C.1.1.2) by two urgent transitions and one urgent state named Intermediate whereby the original transition label is assigned to the first transition. Then, we can state the following property: $\text{EF stateActive}(\text{Section.Intermediate});$.

We list each MTCTL property in Table C.7 and C.8 in Appendix C.3. For example, concerning RTCP Overtaking, we will use the MTCTL property $\text{EF stateActive}(\text{Overtakee.NoOvertaking.EvaluatingRequest});$ as the state Overtakee.NoOvertaking.EvaluatingRequest is the first one after receiving the first message (here: message request). We manually check whether each reachability property is fulfilled and conclude that this is the case.

In order to prepare H4 and H5, we enable to collect the duration of 19 steps of the DSMC:

1. the startup,
2. the protocol to CIC translation,
3. the MTCTL set quantifier normalization,
4. the MTCTL static evaluation normalization,
5. the MTCTL split properties translation,
6. the time unit normalization,
7. the renaming identifiers normalization,
8. the RTSC deadline normalization,
9. the RTSC composite transition normalization,
10. the RTSC state do-effect normalization,
11. the RTSC hierarchy normalization,
12. the RTSC state entry- / exit-effect normalization,
13. the RTSC urgency normalization,
14. the MECHATRONICUML to UPPAAL migration,
15. the UPPAAL model to text translation,
16. the UPPAAL model checking,
17. the UPPAAL text to model translation of results/traces,
18. the back-translation of results/traces from UPPAAL to MECHATRONICUML, and
19. the visualization of MECHATRONICUML results/traces.

Based on these single step durations, we can state various metrics:

- The sum of all 19 steps is the total duration of our DSMC.

- The sum of the steps 1 to 15 is the duration of the forward translation.
- Step 16 is the duration of the model checking itself.
- The sum of the steps 17 to 19 is the duration of the back-translation.

We measure the duration of these 19 steps as follows: We enrich the MECHATRONICUML Tool Suite by the possibility to measure the performance of our DSMC automatically. One can activate this in the dialog “Window»Preferences»MechatronicUML»Verification”. The dialog also enables to define how often a performance measurement shall be repeated automatically, i.e., it enables to define the number of DSMC iterations. We define that we execute each performance measurements 11 times per RTCP. Hereby, we always omit the first measurement as we discovered via several sample tests that the first one always requires more time than the others. We execute each DSMC with the following options: (1) connector out buffer size: 5, (2) hash table size: 27, (3) state space reduction: conservative, and (4) trace options: some.

When verifying multiple properties, our performance measurement verifies all available MTCTL properties. In our case, it verifies all default properties per RTCP. In contrary, when verifying a single property for a trace, our performance measurement verifies the first MTCTL property of the properties list. Here, we reuse the reachability properties from H3. Our performance measurement has 16 measuring points within our DSMC, where it logs the current time in ms since the start of the DSMC: one for each start of the 13 translation steps (see Figure 5.3 on page 149), the start of our UPPAAL plugin, the end of our UPPAAL plugin (which is simultaneously the start of our back-translation and trace visualization), and after the DSMC finished. The result of our performance measurement is a CSV file that lists each timestamp for each measuring point of each DSMC iteration. From the perspective of our performance measurement, the UPPAAL plugin is a black box. Though, within this plugin, the steps 15–17 happen. Therefore, we generate an additional CSV file that lists the timestamp for each of these three steps as well as the end of the UPPAAL plugin. Finally, based on the timestamps of the two CSV files, we can calculate the duration of our 19 steps and our metrics (the duration of each step is the difference of its own timestamp and the timestamp of the next step).

Concerning our hardware and software configuration, we select a Lenovo Thinkpad laptop. It has an Intel Core i7-4800MQ CPU @ 2.70 GHz and 8 GB main memory. We use Windows 7 Enterprise SP1 64 Bit as our operating system. Moreover, we use Java 1.8.0.121 (64 bit) and UPPAAL 4.1.19 with the 32 bit version of verifyta.exe. We launch the MECHATRONICUML Tool Suite from another Eclipse instance. Thus, while evaluating, two Eclipse instances are active. During the measurement, no other programs are open and all network/internet connections are offline.

We do not need to execute special activities to prepare hypotheses H6.

5.8.4 Validating the Hypotheses

In the following sections, we validate each hypothesis separately. For an improved readability, we write RTCP-Instance-Name[n] to name an RTCP instance with a role multiplicity n. For example, we write Distribution[2] for the RTCP instance Distribution with role multiplicity 2.

5.8.4.1 Validating Hypothesis H1

For validating H1, we execute the forward translation of our DSMC for all 16 RTCPs and all their MTCTL properties. For doing this, we use our implementation of the MECHATRONIC-UML Tool Suite with the option to generate intermediate models after each model-to-model transformation. As all three one-to-many RTCPs have a role multiplicity of 1 to 2, we evaluate 19 RTCP instances in total. As a result, we get for each RTCP 12 intermediate models of MECHATRONICUML, one pair of intermediate models of UPPAAL (it consists of one UPPAAL NTA model and one UTCTL model), and one pair of UPPAAL text files (one for the NTA, one for the UTCTL properties).

Afterwards, we execute the forward translation once more to investigate whether it is deterministic. The only difference in the (intermediate) models and the text files are the order of elements in unordered sets. This difference does not influence the semantics. Therefore, we conclude that the forward translation is deterministic for these RTCPs.

Then, we execute EMF's validate function on each intermediate model as well as on each UPPAAL model in order to check for syntactical problems and unfulfilled OCL constraints. EMF returns that no problem exists in any model. Next, we inspect each model for semantic deviations but do not find any problem (except the missing transition priorities of RTCP MultiOvertaking). Then, we load each resulting pair of text files into the UPPAAL's development suite to check for syntactic errors. UPPAAL states for each pair that no problems exist.

As a next step, we manually encode the transition priorities to the resulting UPPAAL text file for the NTA of RTCPs MultiOvertaking[1] and MultiOvertaking[2]. Our DSMC simply ignores the transition priorities. Therefore, the transition that corresponds to the RTSC transition with priority 1 currently may always be fired in both RTCP instances (even if the transition with priority 2 may be fired as well). We solve this by adding an additional guard to the transition that corresponds to the transition with priority 1. This guard ensures that the transition may only fire if no message of type answer may be consumed from the buffer. As a consequence, it is no longer possible that both transitions are enabled at the same time.

We manually search in UPPAAL's development suite for deviations between the original RTCPs and the resulting UPPAAL files. We conclude that no deviations exist. Finally, we execute the model checking within UPPAAL for all RTCPs and all their MTCTL properties. UPPAAL returns that all RTCPs fulfill all their MTCTL properties with one exception: UPPAAL is not able to verify all of the 12 MTCTL properties of RTCP MultiOvertaking[2]. In particular, it is only able to verify 127 of the 148 generated UTCTL properties. For all other UTCTL properties, UPPAAL reports an out of memory exception, i.e., our evaluation PC does not provide enough main memory to store the necessary state space that UPPAAL requires. For example, the check for deadlock freedom does not terminate. Concerning the verifiable properties, UPPAAL returns for all of them their expected verification result. Then, we change the reliability of the role connector of this RTCP to reliable in order to reduce the branches of the NTA and, thus, the state space. Though, UPPAAL is only able to verify two additional UTCTL properties but also fails to verify two properties that check whether a particular state may be active. Due to the change of the reliability, these states are not reachable anymore but UPPAAL cannot detect this. Next, we set up a virtual machine using Linux with 100 GB main memory and execute the model checking of the UPPAAL representation of RTCP MultiOvertaking[2] with the reliable connector again. Hereby, we use the 64 bit version of

verifyta as it enables us to go beyond 4 GB of main memory (UPPAAL provides for Windows only a 32 bit build, but for Linux a 64 bit build as well). However, UPPAAL is only able to verify one additional property compared to the variant with the reliable connector on the machine with 8 GB main memory and the 32 bit build.

5.8.4.2 Validating Hypothesis H2

For validating H2, we execute our DSMC for all RTCPs and all their MTCTL properties except the RTCP instances MultiOvertaking[1] and MultiOvertaking[2] due to their usage of transition priorities. Therefore, we evaluate 17 RTCP instances in total. As a result, our DSMC reports that all MTCTL properties are fulfilled.

We execute EMF's validate function on each generated model of the back-translation to check for syntactical problems and unfulfilled OCL constraints. EMF returns that no problem exists in any model. Next, we inspect each model of the back-translation for semantic deviations but do not find any problem. Afterwards, we execute the complete DSMC (including the back-translation) once more to investigate whether it is deterministic, i.e., whether it produces models that are semantically identical. The only difference in the (intermediate) models are the order of elements in unordered sets. This difference does not influence the semantics. Therefore, we conclude that the DSMC is deterministic for this use case and these RTCPs.

5.8.4.3 Validating Hypothesis H3

For validating H3, the MECHATRONICUML Tool Suite enables for our DSMC to additionally output the UPPAAL trace model (which includes the UTCTL verification results) and the MECHATRONICUML trace model (see Section 5.7.1).

We execute the DSMC for all RTCPs with each MTCTL reachability property except the RTCP instances MultiOvertaking[1] and MultiOvertaking[2] due to their usage of transition priorities. Therefore, we evaluate 17 RTCP instances in total. As a result, our DSMC returns for each RTCP instance a trace.

We execute EMF's validate function on each model of the back-translation to check for syntactical problems and unfulfilled OCL constraints. EMF returns that no problem exists in any model. Afterwards, we execute the DSMC once more to investigate whether it is deterministic, i.e., whether it produces models that are semantically identical. The only difference in the (intermediate) models and the SVG representation are the order of elements in unordered sets. This difference does not influence the semantics. Therefore, we conclude that the DSMC is deterministic for this use case and these RTCPs.

Next, we inspect each model of the back-translation for semantic deviations but do not find any problem. Finally, we investigate the SVG representation of each trace. We conclude that all information are visualized correctly.

5.8.4.4 Validating Hypothesis H4

For validating H4, we execute the performance measurement for each of our 15 RTCPs (we excluded the RTCP MultiOvertaking again). Moreover, we additionally verify a variant of RTCP Overtaking that has a reliable role connector as we expect that this language feature has a significant impact (for each message transmission, UPPAAL has to evaluate that the message

arrives and that it gets lost). In total, we get performance measurements of 18 RTCP instances. For each RTCP instance, we calculate the arithmetic mean based on its 10 total duration times and round it to an integer value. Each RTCP instance has a standard deviation of less than two percent. Therefore, we can conclude that our measurement environment is stable and that our measurements are reproduceable and deterministic.

The diagram in Figure 5.31 presents the results of our RTCP instances concerning their total DSMC duration. Due to the low standard deviation, we refrain from displaying boxplots. As the diagram shows, our DSMC is in most cases (17 of 18) able to return a verification result within less than 1 minute. In particular, 15 RTCP instances (among them the RTCP Overtaking variant with a reliable connector) only require less than 10 seconds. Only the RTCP instances of `ChangeSection` and `Distribution[2]` require ca. 15 seconds. The RTCP Overtaking, which has an unreliable connector, requires significantly more than one minute (ca. 92 seconds). Moreover, it seems to be that a higher role multiplicity increases the total duration significantly: the RTCP `Distribution[2]` requires 14,965 ms but `Distribution[1]` requires 6,467 ms (factor 2.3) and the RTCP `PositionTransmission[2]` requires 3,917 ms instead of 3,276 ms (factor 1.2).

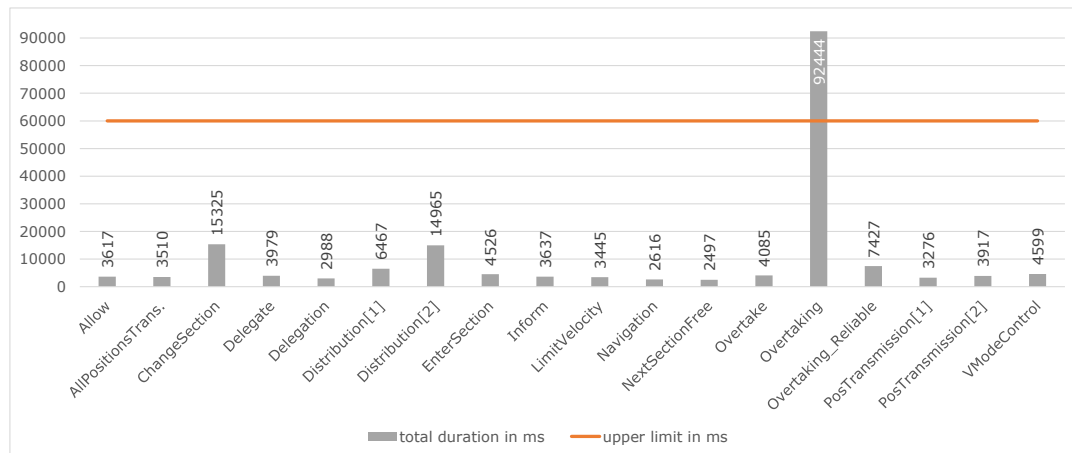


Figure 5.31: Total Duration of the RTCP instances when Verifying Multiple Properties for a Result

For a detailed analysis of this performance measurement, we show the duration of the three DSMC phases per RTCP instance in Figure 5.32. As the diagram shows, the forward translation requires the absolute majority of the time if the DSMC takes less than 10 seconds.

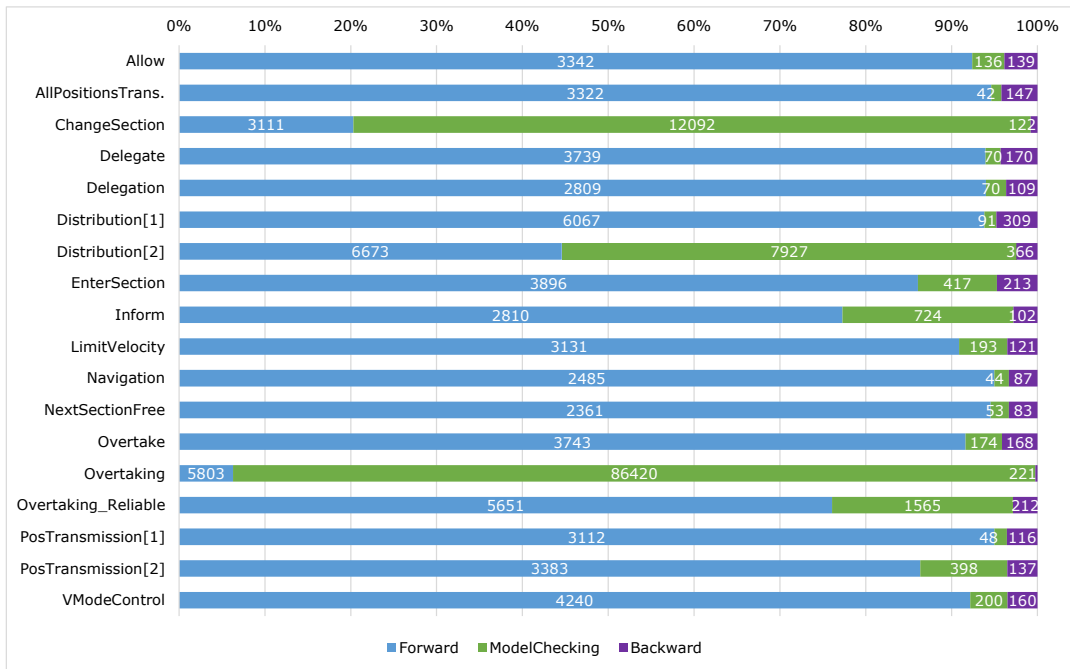


Figure 5.32: Duration of the Three DSMC Phases when Verifying Multiple Properties for a Result

The forward translation requires between approximately 2.3 and 6.6 seconds. It appears that the number of states and transitions of the RTCP instance influences the forward translation time. For example, the RTCP Inform has only 4 states and 4 transitions and requires 2.8 seconds. In contrary, the RTCP Overtaking has 12 states and 17 transitions and requires ca. 5.8 seconds while the RTCP Distribution with role multiplicity 2 has 26 states and 25 transitions and requires ca. 6.7 seconds. The back-translation is typically very fast as it is executed within 1 to 3 tenths of a second. Though, this is not surprising as it only has to translate the information back whether the properties are fulfilled or not. Consequently, it appears that if the DSMC requires more than 7 seconds, then the additional time is only due to UPPAAL's model checking.

The model checking time varies hugely depending on the RTCP instance as it ranges between 42 ms and 86,420 ms. Again, the number of states and transitions seems to influence the model checking time, e.g., the RTCP instances Overtaking and Distribution[2] have the highest model checking time. A reason for this is that two of the seven MTCTL default properties directly depend on the number of states and transitions. For example, the property **forall**(s: States) **EF** stateActive(s); is split into n UTCTL properties, where n is the number of states. Therefore, UPPAAL has to verify more UTCTL properties with a growing number of states and transitions. However, as the diagram in Figure 5.32 shows, the number of states and transitions is not the only depending factor for the total duration as the RTCP ChangeSection only has 4 states and 4 transitions but requires ca. 12 seconds (the deadlock check alone already requires ca. 6.5 seconds). In contrary, the RTCP Inform has the same number of states and transitions but only requires less than a second. As another example, the RTCP Overtaking has a smaller number of states and transitions than the RTCP Distribution[2] but requires 7.2

times more model checking time. Therefore, it seems to be that there are at least two additional important factors of the total duration time: (1) the role multiplicity and (2) the reliability of the connector. Changing the role multiplicity from 1 to 2 already increases the forward translation time as more model elements need to be translated. In addition, it significantly increases the model checking time. In our evaluation, the RTCP Distribution requires 7,927 ms for role cardinality 2 instead of 91 ms (factor 87) and the RTCP PositionTransmission requires 398 ms instead of 48 ms (factor 8). Changing the reliability of the RTCP has a significant effect on the model checking time as well. In our case, the model checking time of RTCP Overtaking increases from 1,565 ms to 86,420 ms (factor 55).

5.8.4.5 Validating Hypothesis H5

For validating H5, we execute the performance measurement for each of our 16 RTCPs. Noteworthy, we can include the RTCPs MultiOvertaking[1] and MultiOvertaking[2] as this property does not reach the transitions with the priorities. Moreover, UPPAAL is able to verify the reachability property that we selected in both RTCP instances. We verify all one-to-many RTCPs with role multiplicity 1 and 2. In total, we get performance measurements of 19 RTCP instances. For each RTCP instance, we calculate the arithmetic mean based on its 10 total duration times and round it to an integer value. Each RTCP instance has a standard deviation of less than two percent. Therefore, we can conclude that our measurement environment is stable and that our measurements are reproducible and deterministic.

The diagram in Figure 5.33 presents the results of our 19 RTCP instances concerning their total DSMC duration when verifying their single property for a trace. Due to the low standard deviation, we refrain from displaying boxplots. As the diagram shows, our DSMC is always able to return a trace at the level of MECHATRONICUML within less than 1 minute. In particular, the maximum duration is always below 30 seconds. Moreover, it seems to be that a higher role multiplicity increases the total duration significantly: the RTCP Distribution[2] requires 29,459 ms instead of 13,562 ms with role multiplicity 1 (factor 2.2), the RTCP MultiOvertaking[2] requires 20,488 ms instead of 14,295 ms (factor 1.4), and the RTCP PositionTransmission[2] requires 7,480 ms instead of 5,244 ms (factor 1.4).

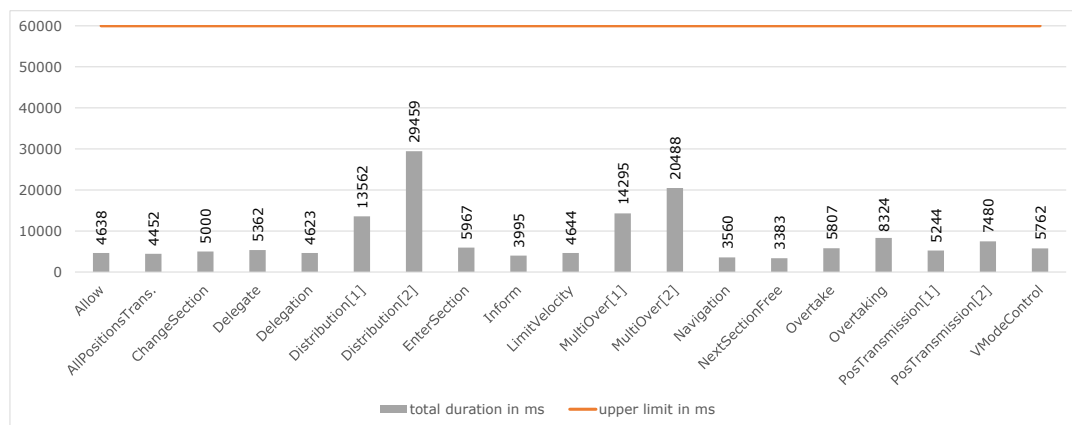


Figure 5.33: Total Duration of the RTCP Instances when Verifying a Single Property for a Trace

For a detailed analysis of this performance measurement, we show the duration of the three DSMC phases per RTCP instance in Figure 5.34. As the diagram shows, the forward translation and the back-translation require the absolute majority of the time while the model checking requires at most 4% of the total duration. The reason for this short model checking time is that we only verify one UTCTL property. Moreover, this property is a reachability property, which only has to identify one path to a state. In comparison, a property that checks for deadlock freedom would need to inspect all possible traces of the NTA.

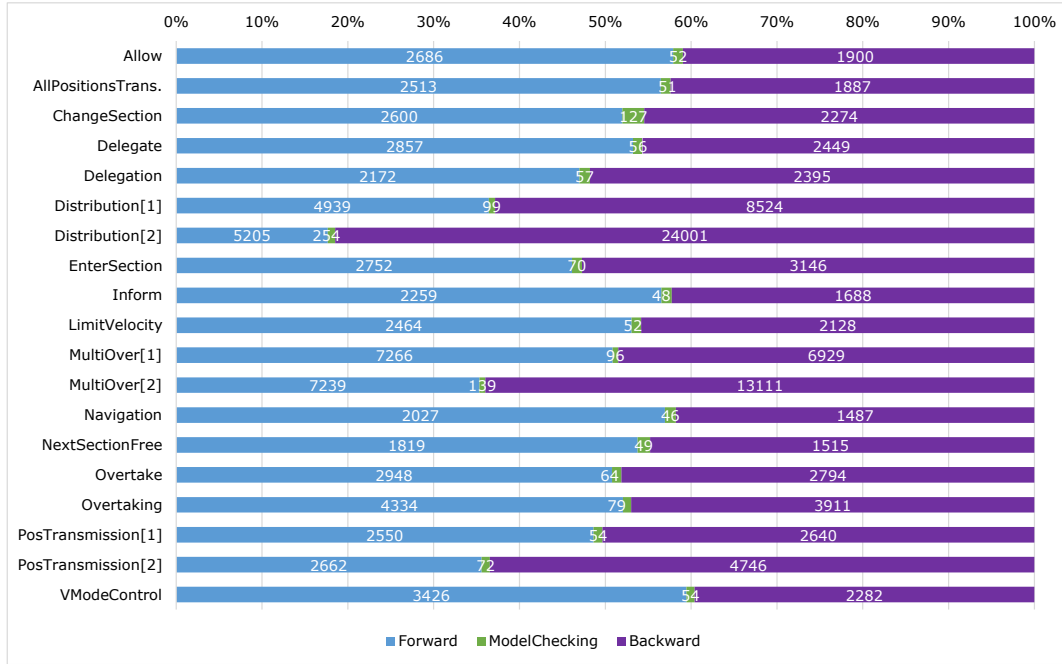


Figure 5.34: Duration of the Three DSMC Phases when Verifying a Single Property for a Trace

The forward translation requires between ca. 1.8 and 7.2 seconds. This is less time compared to the forward translation for verifying multiple properties for a result. The reason for this is that this forward translation needs to translate only one MTCTL property into one UTCTL property. Instead, the forward translation for verifying multiple properties for a result has to translate seven MTCTL properties into 14 to 70 UTCTL properties. The DSMC options of both translations differ only in one option: the forward translation when verifying a single property for a trace additionally has to define the trace kind (some, shortest, fastest). This should have only marginal impact on the forward translation time. The translation of the RTCP's structure and behavior is for both forward translations identical. Therefore, we can apply the same conclusions as for hypothesis H4: it appears that the number of states and transitions of the RTCP instance and the role multiplicity influence the forward translation time.

The model checking time slightly varies depending on the RTCP instance as it ranges between 46 and 254 ms. Again, the number of states and transitions seems to influence the model checking time, e.g., the RTCP Distribution[2] has the highest model checking time. However, the RTCP ChangeSection also requires a very high model checking time

compared to the average but only consists of four states and four transitions. Therefore, other dependencies exist.

The back-translation of traces ranges between 1,515 and 24,001 ms. The RTCP Distribution[2] and the RTCP MultiOvertaking[2] have the highest back-translation time while the RTCPs Navigation and NextSectionFree have the lowest back-translation time. As stated before, the back-translation consists of three steps: the textual trace provided by UPPAAL needs to be parsed by the text-to-model translation (step 17), the model-to-model back-translation (step 18), and the visualization of the trace in terms of an SVG file (step 19). In average, the RTCP Navigation requires 1,190 ms for step 17, 194 ms for step 18, and 14 ms for step 19; the RTCP NextSectionFree requires 1,224 ms, 194 ms, and 98 ms; the RTCP MultiOvertaking[2] requires 10,439 ms, 2,544 ms, and 128 ms; the RTCP Distribution[2] requires 17,093 ms, 6,736 ms, and 172 ms. A possible reason for this huge difference (especially for step 17 and 18) is the varying number of generated timed automata as UPPAAL prints in each snapshot of a textual trace the active states, the variable values, and the clock values. Therefore, the more automata, the more text is produced and the more text needs to be parsed in step 17. Moreover, the step 18 and 19 back-translate and visualize the information of each timed automaton. Therefore, their workload rises as well with an increasing number of timed automata. The number of generated timed automata depends on the number of state regions of each RTCP instance as well as on the role multiplicity. In particular, the latter does not only multiply the subrole RTSCs and the RTSCs of the single role but also the number of message connectors. Navigation and NextSectionFree have 2 regions only, a role multiplicity of 1, and generated 7 timed automata. In contrary, MultiOvertaking[2] has 15 regions, a role multiplicity of 2, and generated 31 timed automata while Distribution[2] has 11 regions, a role multiplicity of 2 as well, and generated 27 timed automata.

5.8.4.6 Validating Hypothesis H6

For validating H6, we execute the forward translation of our DSMC for all selected RTCPs and all their selected MTCTL properties. As all three one-to-many RTCPs have a role multiplicity of 1 to 2, we evaluate 19 RTCP instances in total. These RTCPs are contained in four different MECHATRONICUML model files. Thus, one MECHATRONICUML model may contain more than one RTCP. Moreover, a MECHATRONICUML also contains additional model elements like a component type definition.

We execute our implementation of the MECHATRONICUML Tool Suite with the option to generate intermediate models after each model-to-model transformation. The first intermediate model is serialized before the first forward translation step. In this model, all MECHATRONICUML model elements that are not relevant for the RTCP are deleted. Therefore, this model only contains the model elements for specifying the selected RTCP. As a consequence, we use this model for counting the number of XML elements of the RTCP EMF serialization. In particular, we measure the amount by using the SAX parser of Java 8.

The last intermediate model of the model-to-model forward translation is the UPPAAL EMF model. Therefore, we serialize this model for counting the number of XML elements of the UPPAAL EMF serialization. Again, we measure the amount by using Java 8's SAX parser.

The diagram in Figure 5.35 presents the results of our validation. As the diagram shows, the number of required XML elements increases significantly for each RTCP instance. In particular, it increases by a factor between 6.1 and 16.7 with an arithmetic mean of 9.9 and a

standard deviation of 3. Moreover, it seems that if the size of the RTCP EMF serialization increases, then the size of the UPPAAL EMF serialization increases in most cases as well.

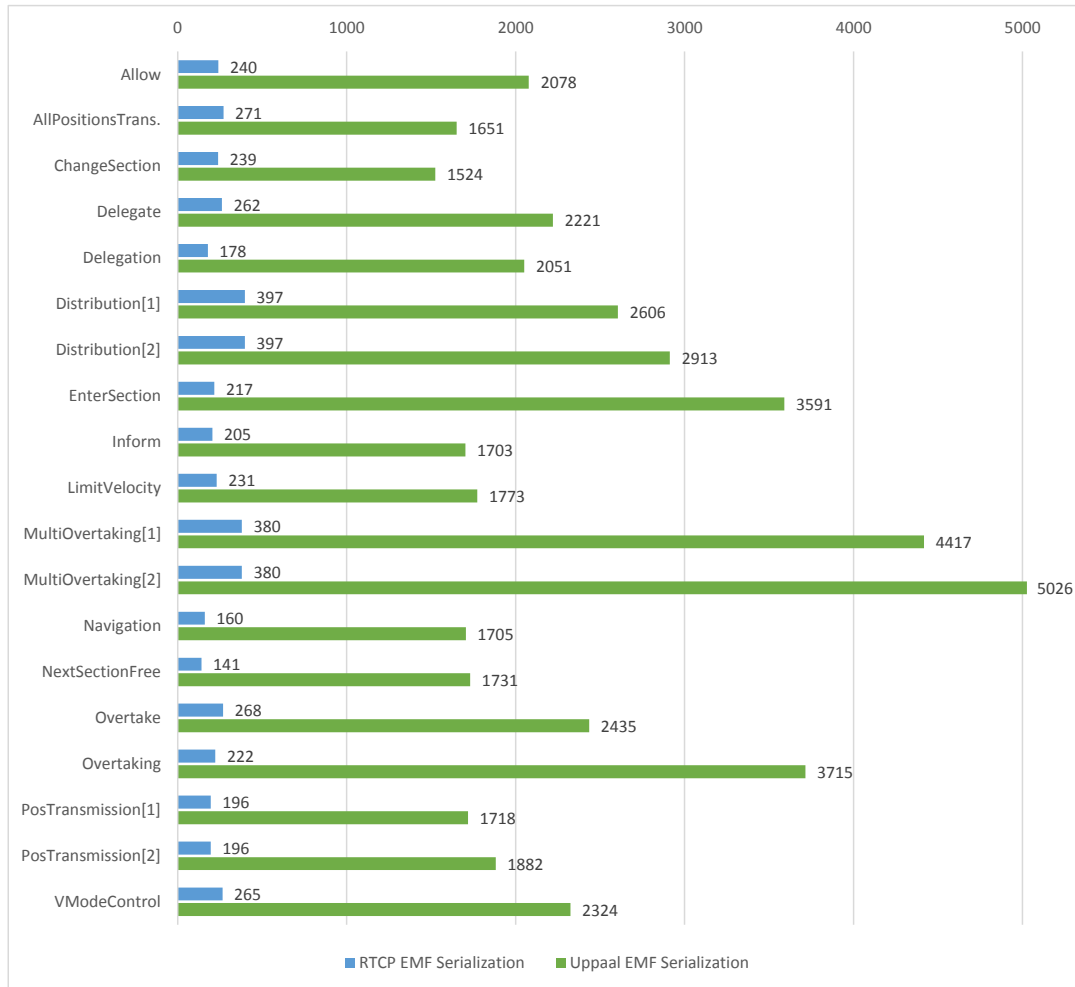


Figure 5.35: Number of XML Elements of the RTCP EMF Serialization and the UPPAAL EMF Serialization

5.8.5 Threats to Validity

According to Runeson et al. [RHRR12], we classify our threats to validity into three parts: construct validity (is the case study designed correctly to answer the hypotheses?), external validity (what hinders us to generalize our findings?), and reliability (would the result be the same with another set of researches?). In the following, we discuss our threats based on this classification schema and report how we consider these threats in the design of our study. Noteworthy, Runeson et al. also define a fourth class called internal validity that we do not need as the examination of causal relations is not one of our evaluation goals.

5.8.5.1 Construct Validity

- All examples of this case study are defined by ourselves, colleagues, or students. Therefore, our evaluation would have more significance if the specification of the RTCPs would have been executed by multiple software engineers (preferable from different companies), who are experts in CPSs and who were not involved in designing our DSMC.
- For validating hypotheses H5, we only verify one MTCTL property per RTCP instance. Instead, we could have also verified a set of MTCTL properties per RTCP instance. Moreover, we only verify reachability properties and no safety properties. For example, we could have additionally inserted an error within the RTCP that leads to a deadlock and verified for deadlock freedom. From the perspective of the back-translation, it is not relevant whether we verify a reachability property, a liveness property, or a safety property as the result is always a trace that consists of a set of snapshots and snapshot transitions. Therefore, only the model checking time of UPPAAL might have been different.
- For validating H6, we choose the UPPAAL EMF serialization (the result of our model-to-model forward translation) instead of the UPPAAL text serialization (the result of the model-to-text forward translation) for evaluating this hypothesis. There are several reasons why the UPPAAL EMF serialization is as good as or an even better choice than the UPPAAL text serialization:
 - The model-to-text transformation, which transforms the UPPAAL EMF model into UPPAAL text, does not translate additional language concepts but straightforwardly transforms one representation into the other. Consequently, the comparison is not distorted when using the UPPAAL EMF serialization.
 - The RTCP EMF model and UPPAAL EMF model both have a fine-grained metamodel with separate classes and, therefore, separate XML elements, for defining elements like clocks, channels, variables, types, and operations and for specifying textual expressions like a variable assignment (see Appendix E.1.7, E.1.4 and E.3.3). However, the DTD of the UPPAAL text serialization is not as fine-grained and requires significantly less XML elements for the same content. For example, all global declarations of the NTA (i.e., global definitions of variables, clocks, operations, etc.) are stored within one single XML element.
- When comparing the model size of RTCPs and UPPAAL models, we count all available XML elements. However, we do not compare the number of attributes or the content size of the XML elements. Therefore, this comparison could have been more fine-grained. However, both models are based on EMF and, therefore, use the identical serialization technique. Consequently, both models serialize each state, transition, clock definition, variable definitions, etc. as one XML element. Therefore, this is still a valid way to compare both models.

5.8.5.2 External Validity

- We only considered 16 RTCPs resp. 19 RTCP instances for our evaluation. We consider these RTCPs as realistic ones and explained that they already cover a huge range of

the expressiveness of RTCPs and MTCTL. Nevertheless, other realistic protocols and MTCTL properties could exist that disprove our findings. In particular, the following threats may be possible:

- It might be the case that we do not translate all RTCPs and all MTCTL properties correctly. However, we explained that our RTCPs and their MTCTL properties already cover a huge range of their expressiveness. Therefore, we rate this threat as low.
- We were not able to verify all properties for only one of our RTCP instances (i.e., RTCP MultiOvertaking[2]). However, it might be an exception that most of our RTCPs were verifiable. Other software engineers might specify their RTCPs differently such that the majority of their RTCPs might not be verifiable completely.
- From a set of 16 RTCP instances, only one RTCP instance required more than one minute to verify multiple properties for a result (RTCP Overtaking). This RTCP was the only one with an unreliable connector. It might be the case that many or most RTCPs with an unreliable connector cannot return a verification result within one minute.
- The single properties per RTCP that we select for producing a trace require between 7–20 snapshots. Their model checking time requires less than 3 tenths of a second and less than 25 seconds for the back-translation. Other properties (reachability or safety) can produce a trace with significantly more snapshots. Therefore, the required model checking time and the back-translation time may increase significantly as well. As a consequence, a total duration of under 1 minute may not be reachable anymore. Though, all our RTCPs require less than 30 seconds for returning a trace. Therefore, we still have over 30 seconds left before we reach our limit of 1 minute.
- We identified that the necessary model size of the UPPAAL EMF representation is always significantly larger. We do not prove that this is always the case. However, according to Section 3.3, an RTCP must define an asynchronous message exchange, where at least one message and one incoming message buffer is involved. Therefore, the UPPAAL EMF representation always has to encode this, which will always increase the necessary amount of XML elements.

5.8.5.3 Reliability

- We might have made mistakes while manually checking whether all RTCPs fulfill their MTCTL properties or while manually checking whether the DSMC has semantical deviations. We reduce the possibility of this threat via triangulation [RHRR12] as I did not evaluate the fulfillment only by myself. Instead, a colleague of our research group, who has two years experience with MECHATRONICUML, compared the semantics as well.
- We might have made mistakes while checking whether the DSMC is deterministic. We reduce the possibility of this threat as we use the tool KDiff3 that automatically compares the serialized models and the generated text files. Therefore, we only had to investigate the differences that our tool found.

- We might have made mistakes while implementing the performance measurements and while calculating the various metrics for H4 and H5. We reduce the possibility of this threat via triangulation as well because a colleague examined our implementation and recalculated our metrics.
- We might have made mistakes while implementing the XML element count of a serialized EMF model and while calculating the metrics. Again, we reduce the possibility of this threat via triangulation because a colleague examined our implementation and recalculated our metrics.

5.8.6 Analyzing the Results

Our case study shows that our DSMC correctly translates RTCPs and their MTCTL properties from MECHATRONICUML to UPPAAL and that it correctly translates verification results and traces back to MECHATRONICUML. Moreover, all translations are deterministic. Hence, we consider the evaluation hypotheses H1 to H3 as fulfilled. However, during our evaluation, we experienced that model checking one-to-many RTCPs with a role multiplicity of 2 or more may result in out of main memory exceptions for some of the properties, e.g., the check for deadlock freedom. Until now, the focus of our DSMC concept is correctness and not memory efficiency. Therefore, future work might improve our concepts such that more complex RTCPs are completely verifiable. Nevertheless, due to the state explosion problem [BCC98], we do not expect huge improvements like a successful verification of a variant of RTCP MultiOvertaking that enables a role multiplicity of 10.

Furthermore, our case study shows that one minute is in most cases sufficient to execute the complete DSMC for our given RTCPs. Hence, we consider the evaluation hypotheses H4 to H5 as fulfilled. In particular, our validation shows that an RTCP with a reliable connector and a role multiplicity of up to 2 requires less than 16 seconds when verifying the default properties without producing a trace. Though, an unreliable connector seems to significantly increase the model checking time. Moreover, our validation shows that we are able to return a trace for a single property within less than 30 seconds.

In addition, we show that our RTCPs always have a significantly more compact model representation of coordination protocols including verification properties than UPPAAL's input language. In particular, we show that the number of required XML elements is at least increased by factor 6. Hence, we consider the evaluation hypothesis H6 as fulfilled.

In summary, it seems that our DSMC correctly implements our concepts and, therefore, seems to successfully hide the UPPAAL model checker completely. Thus, it appears that the formal verification of a safety-critical RTCP may be executed by a software engineer who does not need to know how to encode concepts like asynchronous communication at the level of the model checker. Furthermore, as it seems that our DSMC has a good performance, a DSMC user may execute the DSMC to get immediate feedback during the development of an RTCP. This should help to improve the efficiency of the RTCP's development. Finally, due to our results of the model size comparison, it appears that our DSL's (RTCP and MTCTL) significantly increase the efficiency of specifying coordination protocols and verification properties as they encapsulate domain-specific aspects more effectively than UPPAAL's input language.

5.9 Limitations

Our DSMC approach for RTCP's using UPPAAL underlies the following conceptual limitations:

- Our DSMC approach is limited to one-to-many traceability links between the (intermediate) models. However, as already discussed in Section 5.3, this aspect does not impose any practical restrictions.
- We do not support final states in our DSMC.
- We do not support the usage of variables with data type Role within an RTSC, e.g., we cannot store a reference to a subrole instance within a variable and use this variable within a guard or an action. Moreover, we do not support the data types enumeration, struct, and double.
- For our DSMC, we do not support RTSCs that have a composite state with an exit point and two or more transitions from different regions that define a synchronization and lead to this exit point. This limitation is necessary to realize our concept of the hierarchy normalization (see Section 5.4.10).
- The result of the RTSC hierarchy normalization (see Section 5.4.10) is an RTSC that has one state, which is always active and that has no transitions. An RTSC of a multi role / multi port already fulfills these requirements. Nevertheless, we still flatten the hierarchy of these RTSCs first and integrate them into a new container RTSC. This is a superfluous task and could be improved.
- The UPPAAL semantics enforce firing a transition due to an invariant of another timed automata process. Therefore, an invariant in the RTSC of a role $r1$ of an RTCP may force UPPAAL to fire a transition in the RTSC of a role $r2$. This violates the MUML semantics. Currently, the software engineer has to check manually whether such a behavior is not possible within its RTCP.
- We only reverse engineered the grammar of the UPPAAL trace. Therefore, UPPAAL might produce traces that we cannot parse.
- We may get multiple MECHATRONICUML snapshots for a single state change if the source and/or the target state contains state connection points that must be traversed. The cause is that the RTSC hierarchy normalization (see Section 5.4.10) replaces each state connection point by at least one additional state. Thus, UPPAAL will also produce at least one additional snapshot for this state change.
- The evaluation showed that some of the MTCTL properties of the one-to-many RTCP MultiOvertaking are not verifiable for a role multiplicity 2. RTCPs with a similar or greater complexity might have the same problem.
- We do not provide a formal proof that our translation is always correct and preserves in all cases the semantics in MECHATRONICUML. However, our case study shows that many cases are already correct. Similar to the development of a code generator, we are convinced that remaining errors may be detected during the usage of our DSMC.

Moreover, we currently have the following implementation-specific limitations:

- We visualize counterexamples with an SVG file that contain all necessary information for understanding the trace at the level of MECHATRONICUML. Nevertheless, a preferred solution would be to enable a debugging of the (counterexample) traces

within the MECHATRONICUML Tool Suite. This could help the software engineer to understand counterexamples and identify the root cause of their fault.

- Our DSMC only accepts one-to-many RTCPs where the multi role has an explicit subrole. Therefore, if a multi role uses one-to-many communication schemata (see Section 3.5.2.1), then the DSMC user first has to normalize this RTSC to generate the explicit subrole. As a consequence, we can only provide (counterexamples) traces for one-to-many RTCPs where the multi role has an explicit subrole.
- We always preserve the message order while transmitting. Therefore, a message cannot overtake another message. As a consequence, we currently ignore the QoS assumption of the role connector concerning the message order preservation strategy.

5.10 Related Work

Several approaches exist that provide a translation from a DSL for communicating CPSs to a model checker. Though, none of them provides a DSMC that hides the model checking to the software engineer. In contrary, several DSMC approaches exist but all of them do not focus on communicating CPSs resp. coordination protocols of CPSs.

Therefore, we discuss related work in two categories: First, we present in Section 5.10.1 approaches that aim at formally verifying CPSs via a translation to a model checker. Then, we discuss approaches that completely hide formal methods by providing a back-translation for (counterexample) traces.

5.10.1 Translating DSLs for Communicating CPS to a Model Checking

“HUGO/RT [KMR02] is a prototypical tool for model checking timed UML state machines via UPPAAL. Their concept of having separate automata for the message transmission and buffering were the basis for our work. However, their concept for these automata has several limitations compared to us. First of all, they do not support the use case that a buffer is full but another message appears even though this is a realistic use case (besides MECHATRONIC-UML, AUTOSAR [AUTOSAR] supports this use case as well). If this happens, they stop the verification.” [*DGH15] Instead, we provide three buffer overflow avoidance strategies. For example, we are able to discard the incoming message or the oldest message in buffer. “[HUGO/RT] only enables to have a fixed message delay, which preserves the order of sending messages, or an upper bound message delay, which does not preserve the order of sending messages. Instead, we support the more realistic [assumption] of a [...] maximal message delay and are still able to always preserve the order of sending messages. In addition, in contrast to us, they are not able to handle the possibility of message loss, multiple message types per buffer, and message parameters. Furthermore, they do not describe whether they support multiple incoming message buffers per role.

David et al. [DMY02] translate a subset of UML Statecharts called hierarchical timed automata (HTA) with real-time extensions to the model checker UPPAAL. As HTAs are quite similar to UPPAAL timed automata, their main task is the translation from hierarchical to flat state machines. Noteworthy, their flattening approach was the basis for our concept. However, RTCPs and their RTSCs provide significantly more high-level modeling construct for easing the work of the domain expert, e.g., asynchronous communication, time-consuming transitions, state effects (entry, do, exit). In addition, we also provide with MTCTL a property

language at the level of the design model and translate all MTCTL properties into UTCTL. David et al. do not provide a modeling language for verification properties and, therefore, do not need to translate it.

The following works all enable a translation from a DSL to UPPAAL: Xu et al. enable it for COMDES-II [XPSA08], Furfaro and Nigro for RT-DEVS [FN08], Johnson et al. for AADL [JLPJ12], and Nadales Agut et al. for the Compositional Interchange Format (CIF) [NRS+11]. However, all these DSLs and, therefore, all translations do not require asynchronous communication including (multiple) incoming buffers. Moreover, only AADL enables to define a message delay. Though, the delay is constant and not variable as in our DSL. In addition, all these DSLs have no language [elements] like hierarchical state machines and state effects (entry, do, exit). Consequently, their translations are less complex and not reusable for our work.” [*DGH15]

As already mentioned in Section 3.10, Dragomir et al. [DOP15; Dra14] enable model checking for contracts via a translation from SysML block diagrams and state machines [SysML] to a variant of timed-input-output-automata (TIOA), which are similar to UPPAAL timed automata. Like us, their state machines support real-time requirements and sending asynchronous messages as well as consuming them from a message buffer. Nevertheless, Dragomir et al. have to deal with a less complex transformation: First of all, the SysML state machines that they use are less complex as they do not support state hierarchy (and, therefore, no entry- and exit-points), state invariants, or state effects (entry, do, exit). Furthermore, they do not have to deal with one-to-many communication, user-defined QoS assumptions, or properties of incoming message buffers (size, discard strategy). Moreover, they do not provide a DSVPL that needs to be transformed into the input language of the verification tool.

“Tiwari et al. [TSR03] develop “automated analysis techniques for asserting correctness of hybrid system designs” [TSR03, p.1] in embedded control systems. In particular, they define ‘a formal semantics for the Stateflow modeling language by translating a Stateflow model into a set of communicating pushdown automata’ [TSR03, p. 4]. However, in contrast to MECHATRONICUML, Stateflow does not enable an asynchronous message exchange (including message buffers, message delay, and message loss).” [*DGH15]

5.10.2 DSMC

“Guerra et al. [GdLMD09] present the *BaVeL* DSL for the modeling of DSMC workflows including back-translations. The language uses triple graphs [Sch95] for the specification of translation rules. During the forward translation, these rules produce the required traceability links that are later used by the back-translation. However, in case of multi-step translations, the approach requires a multi-step back-translation as well, because each step of the forward translation must be traced back individually by means of a separate triple graph. In contrast, our approach enables a single-step back-translation that is independent of the number of forward translation steps.

Zalila et al. [ZCP13] provide a methodology for DSMC that is based on a metamodeling pattern for executable DSLs. Shah et al. [SAB10] present a translation of structural *UML* models to *Alloy* for static consistency analysis. The authors use a meta transformation to generate the actual back-translation on-the-fly from the recorded traceability links. The *MADES* approach by Baresi et al. [BBK+15] uses traceability only to highlight the

correspondences between counterexample elements and the equivalent elements of the design model (without providing a proper back-translation). Common to these three approaches is that they require an additional instrumentation of the forward translation to produce a specific kind of traceability output. Thus, transformation engineers need to enrich the basic translation with additional traceability logic. Especially in case of multi-step translations, enriching a model transformation manually implies a huge development effort. In contrast, our approach demonstrates that traceability links generated implicitly by the transformation engine are sufficient even in case of multi-step translations.” [*DGH15]

“Combemale et al. [CGR11] provide a generic approach for the back-translation of execution traces such as counterexamples[...].” [*DGH15] Their approach requires a formal operational semantics definition of the DSL but not of the model checker. The main focus of the authors is the correct matching of the snapshots, e.g., in our case several UPPAAL trace snapshots may represent one MECHATRONICUML snapshot. The authors do not explain how to back-translate the elements of a single snapshot of the model checker to a single snapshot of the model checker. Instead, they only mention that they assume that the software engineer specifies this relation. In contrary, our approach explains how to back-translate the elements of a snapshot (even in the case of a multi-step forward translation).

“Hegedüs et al. [HBRV10] use traceability links for the back-translation [...] but require the operational semantics of the DSL and the verification formalism (e.g., a model checker’s input language) to be formalized in terms of graph transformations.” [*DGH15] However, the UPPAAL model checker is closed source and a complete formal semantics definition is not available. Therefore, their approach is not applicable. In contrary, our approach does not require a formal semantics definition of the verification formalism.

Goldsby et al. present Theseus, a generic visualization framework that allows the automatic translation, verification, and the display of verification results and counterexamples [GCKK06]. The framework takes as input UML models and verification properties stated in natural language templates. Within a single step, they translate the input and execute the model checking. The framework is able to parse the textual model checker trace. In particular, it extracts four types of dynamic behavior: (1) a state is activated, (2) a transition is fired, (3) a message is sent, and (4) a message is received. The extracted dynamic behavior is exported into an XML format that may be imported by a UML CASE tool. Using this tool, the authors enable to inspect the trace via animated UML state machines or animated UML sequence diagrams. Noteworthy, the generated UML trace does not reference the input model. In this thesis, the focus is to deal with huge differences between the DSL (including the DSVPL) on the one hand and the input language of the model checker (including its verification property language) on the other hand. Therefore, in contrast to Goldsby et al., our concept enables a step-wise translation, which makes the specification significantly easier as the software engineer can focus on each difference within a separate translation step. As another difference to Goldsby et al., we enable a single step back-translation by utilizing the traceability links of the forward translation. As a consequence, we can—for example—identify that several states and transitions of the model checker’s trace relate to a single state of the design model. This is a feature that the approach of Goldsby et al. does not enable.

“Molotnikov et al. [MVR14] address the verification of domain-specific extensions to the C programming language. The verification is achieved by a multi-step translation to a basic form of verifiable C.” [*DGH15] The authors report that they can translate counterexamples

trace back to the domain-specific level by utilizing traceability information. However, they do not explain how they realize this on a conceptual level.

5.11 Summary

In the following, we summarize to what extent this thesis and the related approaches that we discussed in Section 5.10.1 fulfill the requirements that we stated in Section 5.2 for a DSMC for coordination protocols of CPSs. As a reminder, the requirements were based on our own experience and are not proven, e.g., by a systematic literature review or requirements workshops. Afterwards, we summarize the existing concepts to realize DSMC and compare them with our approach.

	<i>this thesis</i>	<i>HUGO/RT</i>	<i>David et al.</i>	<i>Xu et al.</i>	<i>Furfaro et al.</i>	<i>Johnson et al.</i>	<i>Nadales Agut et al.</i>	<i>Dragomir et al.</i>	<i>Tiwari et al.</i>
R1	✓	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)
R2	✓	×	×	×	×	×	×	×	×
R3	✓	×	×	×	×	×	×	×	×
R4	✓	×	×	×	×	×	×	×	×
R5	✓	✓	✓	✓	✓	✓	✓	✓	?
R6	✓	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
R7	(✓)	?	?	?	?	?	?	?	?
R8	✓	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
R9	✓	×	×	×	×	×	×	(✓)	×

Table 5.1: Comparison of the Requirements Fulfillment for a DSMC for CPSs

We provide an overview of the requirements fulfillment in Table 6.1 (Legend: ✓ = fulfilled, (✓) = partially fulfilled, × = not fulfilled, ? = unknown, n.a. = not applicable). As the table shows, this thesis fulfills nearly all requirements (one is only partially fulfilled) that we stated at the beginning of this chapter. Moreover, no other approach or a combination of approaches would fulfill all requirements.

The requirements R1 (translate the DSL model), R2 (translate the DSVPL model), R3 (automate the model checker), and R4 (translate results and traces back) provide the basis for hiding the model checking to the DSMC user. This thesis is the only work that fulfills all these requirements. We provide them in Section 5.4, 5.5, and 5.6. All other approaches only provide concepts to translate the DSL model but do not provide concepts to translate a DSVPL model, to automate the model checker, or to back-translate results or traces. Moreover, all approaches only fulfill R1 partially as their DSL does not primarily focus on verifying coordination protocols of CPSs but mostly on communicating embedded systems in general. For example, most of the translations do not need to cover domain-specific concepts like asynchronous message communication including message buffers or QoS assumptions. The translations that cover asynchronous message communication (HUGO/RT and Dragomir et al.) do not cover all our needs that we identified for RTCPs, e.g., buffer overflow avoidance

strategies, multiple buffers, or message parameters. Furthermore, except David et al., none of them has to deal with the flattening of the state machine hierarchy.

Concerning requirement R5 (correctness of the DSL translation), Nadales Agut et al. give a formal proof concerning the correctness. Our thesis and all other related approaches (except Tiwari et al.) use case studies to show the correctness of the translation of the DSL. Even though, a case study does not prove the correctness, it shows that the translation is correct for realistic examples. Therefore, we still consider that the requirement is fulfilled. In contrary, Tiwari et al. do not discuss the correctness nor do they provide a case study. Consequently, we define that it is unknown whether they fulfill this requirement. Moreover, during our evaluation in Section 5.8, we show that we correctly translate a huge range of MTCTL properties for a given set of realistic RTCPs. Therefore, we consider requirement R6 (correctness of the DSVPL translation) as fulfilled. The assessment of this requirement is for all related approaches not applicable as they do not translate a DSVPL.

In Section 5.9, we state that our DSMC currently does not support the complete expressiveness of RTCPs, e.g., final states and transition priorities are not supported. Therefore, this thesis only partially fulfills requirement R7 (support the complete expressiveness of the DSL). The related approaches make no statement about this. However, this thesis supports the complete expressiveness of MTCTL. Therefore, it fulfills requirement R8 (support the complete expressiveness of the DSVPL). The assessment of this requirement is for all related approaches not applicable as they do not translate a DSVPL.

The assessment of requirement R9 (enable to define the relevant DSMC options) is subjective as it is hard to decide which options are relevant and which are not. In our opinion, this thesis fulfills the requirement as it provides the relevant options for the translation (i.e., the role multiplicity and the connector buffer size.) and for the model checking (e.g., the hash table size and the trace kind). The only other approach that provides translation option is Dragomir et al. as the software engineer can specify the size of the incoming message buffer. All related approaches do not provide model checking options as they do not automate the model checking.

The related work that we discussed in Section 5.10.2 presents several approaches to realize a DSMC. However, they are either (1) not applicable for our DSMC of RTCPs via the model checker UPPAAL, (2) have disadvantages compared to our approach, or (3) provide too less conceptual details to apply them. Hegedüs et al. and Goldsby et al. belong to category (1): Hegedüs et al. require formal semantics definition of the model checker, which UPPAAL does not provide. Goldsby et al. only allows simple back-translations but we require complex ones as well (e.g., mapping several UPPAAL states to one RTCP state). Guerra et al., Zalila et al., Shah et al., and Baresi et al. belong to category (2): Guerra et al. require a multi-step back-translation but we only require a single-step. Zalila et al., Shah et al., and Baresi et al. require additional efforts by the software engineer to realize the back-translation. Our approach does not need this. Combemale et al. and Molotnikov et al. belong to category (3): Combemale et al. do not provide concepts for back-translating elements of a snapshot and Molotnikov et al. do not report about their back-translation on a conceptual level.

FORMAL DESIGN PATTERNS FOR RTCPS

In this chapter, we introduce our concept of formal design patterns of RTCPs, which we call Real-Time Coordination Patterns. These patterns capture well-proven and reusable solutions to a commonly occurring coordination problem within the domain of CPSs.

In particular, we start with the contributions of this chapter in Section 6.1. Then, we point out the stakeholders and requirements in Section 6.2. Afterwards, we introduce our concept Real-Time Coordination Patterns in Section 6.3. Based on our concept, we explain our pattern-based development method, which consists of two use-cases: (1) select and adapt an application-independent Real-Time Coordination Pattern to an application-specific RTCP and (2) abstract an application-specific RTCP to an application-independent Real-Time Coordination Pattern. Afterwards, we explain our implementation in Section 6.6. We describe our evaluation in Section 6.7 before stating the current limitations of our work in Section 6.8. Then, we discuss related work in Section 6.9. Finally, we summarize this chapter in Section 6.10.

The contents of this chapter have been published in [*DHT12; *DBHT12; *AGD+12; *PDM+14; *GTS+14]. Moreover, parts of this chapter were contributed by the project group SafeBots III [ABB+13] as well as the bachelor’s theses of Jannis Drewello [Dre11], Stefan Riepe [Rie11], Marcel Sander [San11], and Alexander Winkler [Win14]. The project group and the bachelor’s theses from Drewello, Riepe, and Sander were supervised by me. The bachelor’s thesis from Winkler was supervised by my research group colleague Uwe Pohlmann. In particular, the project group Safebots III provided an initial approach to abstract an RTCP to a pattern (see Section 6.4) and to select and adapt a pattern to an RTCP (see Section 6.5). Drewello, Riepe, and Sander identified in their theses new patterns or at least new variants of existing patterns. Furthermore, Sander and Winkler both helped to execute the case study that we present in Section 6.7.

6.1 Contributions

The main contributions of this chapter are as follows:

- We define a set of seven requirements for formal design patterns of RTCPs.
- We introduce the concept of formal design patterns for RTCPs of MECHATRONIC-UML, which we call Real-Time Coordination Patterns. These design patterns capture existing modeling experience, facilitate the correct specification (with a focus on formal verification), and ensure a good design of RTCPs. Currently, we identified nine patterns so far.

- We provide a uniform description format for Real-Time Coordination Patterns and define all existing patterns using this format.
- We define a development process for selecting and adapting Real-Time Coordination Patterns to RTCs as well as for abstracting RTCs to Real-Time Coordination Patterns.
- In an evaluation, we show that our patterns have a high reusability rate within two different applications and that the RTCs that were adapted from a pattern are still similar to the pattern and do not change their intent.

6.2 Stakeholder and Requirements

In this section, we define the stakeholders for design patterns of a DSL for safety-critical systems that enables formal verification. Then, we formulate and justify requirements that the design patterns as well as the development process have to fulfill for a given DSL and DSVPL as well as a given model checker. In our case, the DSL is RTCs, the DSVPL is MTCTL, and the model checker is UPPAAL.

We distinguish between two groups of stakeholders. The first group—the *pattern users*—consists of software engineers that have to specify models using the DSL. Moreover, they also shall verify their specified DSL models by applying DSMC. Therefore, they also need to specify the verification properties using the given DSVPL. This group does not need to have significant experience in the design and specification of DSL and DSVPL properties as this experience is encapsulated into the patterns. However, we assume that they have at least basic experience in this topic, i.e., they are familiar with both languages and with DSMC. Therefore, the idea is that the design patterns and an appropriate process facilitate their work to create valid (w.r.t. the syntax of the DSL and the verification properties) and well-designed models.

The second group—the *pattern specifiers*—are software engineers that identify, specify, and describe design patterns for RTCs. Thus, we assume that they have significant experience using the DSL, the DSVPL, and DSMC. They are able to identify design patterns for a given problem, can specify the models such that they fulfill all relevant verification properties, can describe all relevant aspects textually, and decide whether they have a good design.

In the following, we present requirements that design patterns for a DSL like MECHATRONICUML that enables formal verification shall have at least. We order the requirements according to their priority, where the first is the most important requirement. The requirements and their priorities are based on our experience. Therefore, we do not claim that this is a complete list.

In our opinion, a design pattern concept for a DSL that enables formal verification shall ...

- R1** provide proven and well-designed solutions for reoccurring design problems in this context,
- R2** provide textual descriptions using a uniform description format,
- R3** provide formal models that are valid w.r.t. the DSL,
- R4** enable a formal verification via DSMC, and
- R5** provide formal domain-specific verification properties stated with the given DSVPL.

Furthermore, we require that a software development method that enables the usage of such patterns shall ...

R6 provide concepts to select and adapt a formal design pattern to a concrete realization and

R7 provide concepts to identify and abstract an existing model into a formal design pattern.

In the following, we will justify our requirements. The requirement R1 is taken over from the definition of software design patterns by Gamma et al. [GHJV95] and Buschmann et al. [BMR+96] and is the most essential one. Otherwise, the term *pattern* would not fit. For example, if a pattern is only applicable in the application where it was identified, then the effort for defining the pattern is pointless as it would not provide any benefits for software engineers of other systems. Moreover, if the patterns are not proven or well-designed, then the patterns should not be a template for future software as the usage of the pattern may decrease the quality of the software under development.

The requirement R2 is also typical for a pattern catalog as the uniform description format facilitates to identify the most suitable pattern and enables to compare the patterns with each other. For example, Gamma et al. [GHJV95] and Buschmann et al. [BMR+96] both define their own description format.

In contrast to the requirements R1 and R2, the requirements R3, R4, and R5 are not stated by Gamma et al. [GHJV95] or Buschmann et al. [BMR+96]. However, related work, e.g., Taibi et al. [Tai07] as well as Ramirez and Cheng [RC10], states that these requirements are important for such domains. In particular, the requirements R3 is necessary as our design patterns shall be used to design models for safety-critical systems where formal verification is required. Having only informal descriptions or informal models would therefore require additional effort by the pattern user and, most importantly, raising the risk to introduce (safety-critical) errors in the model. Requirement R4 is important as we want to avoid that the pattern user requires the skills of the DSMC developer (see Section 5.2). Thus, the pattern user shall only need the skills of a DSMC user. Concerning requirement R5, we already stated in Section 4.4 that the choice of verification properties for an RTCP is highly important. Otherwise, existent (design) faults may remain undetected and may lead to safety-critical situations. Therefore, the pattern user needs support for specifying correct and relevant problems stated within a formal domain-specific verification properties language that references the formal design model.

The requirements R6 and R7 are essential to establish a pattern-based process that is efficient and reduces manual errors. Concerning R6, the pattern user should be supported to select / search a fitting pattern efficiently and to adapt it to a protocol that still contains the pattern's intent. Concerning R7, the pattern specifier should be supported to identify / harvest / mine new patterns and to abstract them correctly (such that they still solve the given problem correctly). For example, automatic model translations, guided graphical user interfaces (wizards), and/or a process consisting of mandatory and optional tasks can support the pattern user as well as the pattern specifier while doing their job.

6.3 Real-Time Coordination Patterns

In this section, we introduce our formal design patterns for RTCPs, which we call Real-Time Coordination Patterns. First, we start with introducing our concept of Real-Time Coordination

Patterns. Next, we give a definition for our patterns in Section 6.3.2. Then, we briefly describe the patterns that we identified so far in Section 6.3.3. Afterwards, Section 6.3.4 introduces our description format, which we use to describe each pattern in a consistent manner. Finally, in Section 6.3.5, we use our description format to define a pattern, which we use as a running example for the remainder of the chapter, in detail.

6.3.1 Concept

Our idea is to introduce application-independent design patterns for RTCPs, which we call Real-Time Coordination Patterns. These patterns are defined by software engineers, which we call pattern specifiers. As already described in Section 6.2, pattern specifiers have the necessary experience to identify and define such patterns. As a consequence, these patterns can facilitate the work of the pattern user significantly as the solved problem is stated explicitly and the solution is available as application-independent formal models.

We illustrate our general concept of Real-Time Coordination Patterns in Figure 6.1. First of all, it is necessary that the pattern specifier has to identify recurring design problems and corresponding solutions among the existing application-specific RTCPs. For example, the RTCP *ConvoyCoordination*, which is depicted in the left of our figure, tackles the problem that a system sometimes requires assurances by another system in order to execute a certain task or to change a mode (here: in order to use the slipstream of a convoy, the rear vehicle requires the assurance that the front vehicle will send a brake message before braking).

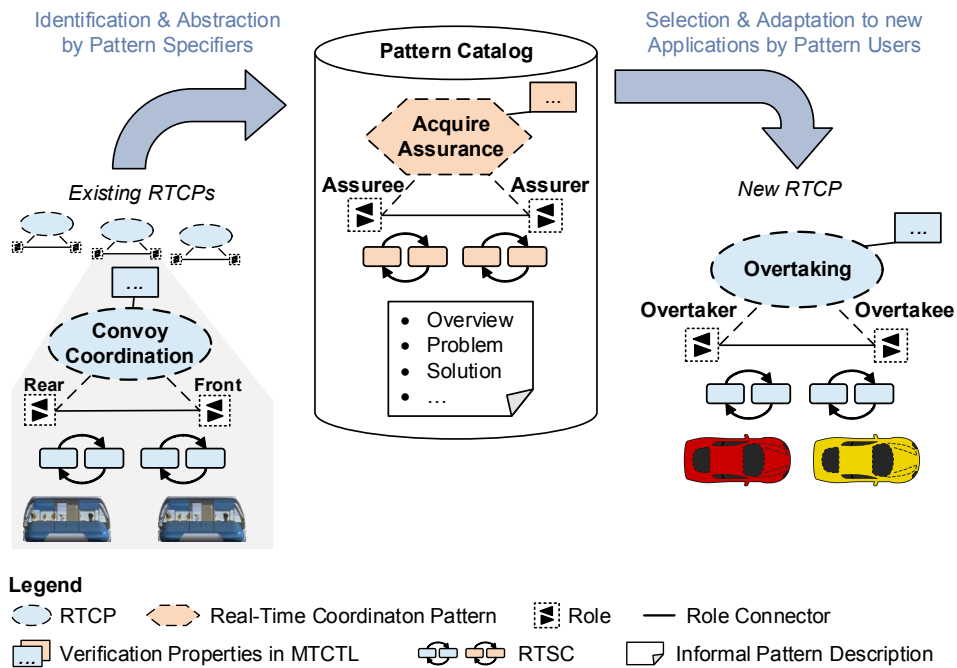


Figure 6.1: General Concept for Real-Time Coordination Patterns

If the pattern specifier identifies such a solution, he shall abstract the RTCP from all its application-specific properties in order to make it as reusable as possible (we define this process in detail in Section 6.4). The result is a Real-Time Coordination Pattern that is

application-independent and, therefore, ready to be reused in different applications. A Real-Time Coordination Pattern is very similar in its structural and behavioral definition to an RTCP. It consists of a formal model (including MTCTL properties) that enables formal verification via DSMC and a set of informal descriptions. For example, we abstracted the RTCP ConvoyCoordination to the Real-Time Coordination Pattern Acquire Assurance, which is no longer specific for convoys of vehicles. Instead, it enables to acquire an assurance that is necessary to execute a safety-critical task or to activate a safety-critical mode.

All identified Real-Time Coordination Patterns are stored in an electronic pattern catalog. The pattern user can access this catalog and, therefore, search for a pattern that fits to its coordination problem. Then, the pattern user selects and adapts the formal model to its needs and verifies it, among others, with the verification properties that were provided with the pattern. For example, we were able to reuse our pattern Acquire Assurance for several applications, e.g., for the running example of this thesis: the RTCP Overtaking. Here, the assurance is that as long as the Overtaker executes the overtaking, the Overtakee assures that it will not accelerate any further.

In summary, with this concept, we can reuse the existing experience for specifying safe and well-designed RTCPs in new applications and, therefore, facilitate the work of the pattern user.

6.3.2 Definition

A Real-Time Coordination Pattern follows the general definition of a pattern within software design as it “provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context” [BMR+96].

In addition, a “Real-Time Coordination Pattern describes a well-proven, reusable, [application-independent], and formal solution to a commonly occurring coordination problem within the domain of [CPSs]” [*DHT12]. Hence, Real-Time Coordination Patterns are software patterns that shall support inexperienced software engineers in specifying one-to-one and one-to-many RTCPs.

Because of its domain, a Real-Time Coordination Pattern focuses on the asynchronous message exchange of two roles, which represent communicating actors, under hard real-time constraints in a safety-critical environment. Due to the latter one, these patterns provide in their solution a formal design model as well as formal verification properties. The formal design model is based on the model representation of RTCPs (see Chapter 3) and only extends them by additional information (we will explain them in the following). The verification properties are stated in MTCTL (see Chapter 4), which reference the pattern model. As we use an RTCP-like DSL and MTCTL, we can reuse our DSMC via the model checker UPPAAL (see Chapter 5) in order to formally verify a pattern.

Compared to an RTCP, a Real-Time Coordination Pattern abstracts from application-specific details to be reusable in different application scenarios. For example, it enables to specify parameters instead of concrete values. These parameters are defined at the pattern itself and may be used within any expression of the pattern, i.e., within the RTSCs (invariants, guards, clock constraints, ...), the MTCTL properties, and the maximum message delay assumption of the role connector. However, if a pattern defines pattern parameters, then the

formal model of the pattern does not necessarily fulfill all stated MTCTL properties for any given set of parameter assignments. A pattern enables to store valid parameter assignments as so-called verified configurations. Hereby, each verified configuration stores one concrete value per pattern parameter.

Noteworthy, even though we enable that a pattern provides parameters, it is not a template but still a pattern because we do not restrict the pattern user to adapt it to its needs. The reason for this is that the necessary adaptations may vary significantly depending on the system under development and its requirements. Instead, we rather encourage the pattern user to do it by stating several alternatives for each pattern.

6.3.3 Identified Patterns

Until now, our pattern catalog consists of nine Real-Time Coordination Patterns that we identified in one-to-one and one-to-many RTCs. Each one of our identified patterns is applicable for one-to-one and one-to-many RTCs. For consistency reasons, we define them as one-to-one patterns but describe at least one one-to-many variant for each pattern. In fact, the necessary changes of the models between a one-to-one pattern and its one-to-many variant are comparable low due to our one-to-many communication schemata that we introduced in Section 3.5.2.1. Currently, our catalog does not contain any Real-Time Coordination Patterns that are applicable to one-to-many RTCs only. Though, we think that such patterns might exist.

We identified our patterns manually by inspecting existing RTCs from several applications, e.g., the RailCab system [HTS+08] and the BeBot system [HWR09; BeBot]. However, the number of existing RTCs is with currently less than 50 RTCs tremendously small compared to the available code basis that other pattern catalogs have, e.g., the catalogs of Buschmann et al. [BMR+96] and Gamma et al. [GHJV95]. Therefore, all patterns that we identified so far are rather pattern candidates that have to prove their quality in practice. In fact, better design solutions may exist for the given problems.

In Appendix B, we explain all our Real-Time Coordination Patterns in detail. A brief overview of our Real-Time Coordination Patterns is as follows:

Acquire Assurance (see Section 6.3.5) enables a role, which we call *Assuree*, to acquire an assurance by another role, which we call *Assurer*. The pattern assumes that a safety-critical situation appears if the *Assuree* assumes that the assurance is still valid, but the *Assurer* currently does not assure it. Therefore, the pattern ensures that this situation never happens despite message delay and message loss. In the remainder of this chapter, this pattern will be our running example.

Request Information (see Appendix B.1) enables that a role *Requester* can request information from a role *Provider*. The pattern considers that providing the information takes time. Moreover, only one request at a time is allowed.

Fail-operational Delegation (see Appendix B.2) “realizes a delegation of a task from a role *Master* to a role *Slave*. The *Slave* has to execute the task in a certain time and answers regarding success or failure. The pattern assumes that a failure is not safety-critical, though only one delegation at a time is allowed.” [*DBHT12]

Fail-safe Delegation (see Appendix B.3) “realizes a delegation of a task from a role Master to a role Slave. The Slave has to execute the task in a certain time and answers regarding success or failure. If the execution fails, no other task may be delegated until the Master ensures that the failure has been corrected. Moreover, only one delegation at a time is allowed.” [*DBHT12]

Event-based Transmission (see Appendix B.4) “is used to transmit information from a sender to a receiver if an event occurs (e.g., a condition becomes true). If the receiver does not get the information within a certain time, a specified behavior must be activated to prevent a safety-critical situation.” [*DBHT12]

Periodic Transmission (see Appendix B.5) is “used to periodically transmit information from a sender to a receiver. If the receiver does not get the information within a certain time, a specified behavior must be activated to prevent a safety-critical situation.” [*DBHT12]

Alternating Lock (see Appendix B.6) is used when two roles may not execute their actions at the same time. The roles have to alternate their executions. In addition, both roles may only execute their actions for a defined length of time. Then, they have to pass the permission to execute actions to the other role.

Block Execution (see Appendix B.7) is used when one role (the Guard) may block certain actions of the other role (the Executor), e.g., due to safety-critical reasons. In particular, the Guard may at any time start and end the blocking.

Limit Observation (see Appendix B.8) is used if one role shall notify the other role whether a certain measurement value violates a pre-defined limit or not.

6.3.4 Description Format of Real-Time Coordination Patterns

In this section, we define a uniform description format for Real-Time Coordination Patterns. Our main goal is that it facilitates the pattern selection for the pattern user. Therefore, we aim for understandable descriptions that enable to compare and combine the existing patterns with each other. Moreover, the description shall also cover all relevant information that facilitate the formal verification of the pattern.

Many description formats for software patterns already exist, e.g., the ones of Buschmann et al. [BMR+96], Gamma et al. [GHJV95], and Ramirez and Cheng [RC10]. For example, Buschmann et al. define the following aspects: name, also known as, example, context, problem, solution, structure, dynamics, implementation, example resolved, variants, known uses, consequences, and see also. However, none of them matches (exactly) with the contents we want to provide, e.g., that we also focus on the formal verification of the patterns. Therefore, we base our format on the one of Buschmann et al. [BMR+96] and adapt it to our needs. In particular, our changes to the format are as follows: First of all, we extract the summary of the pattern from the aspect name and provide it within a new aspect, which we call *overview*, in order to separate concerns. Moreover, we exchange the aspect *dynamics*, which shall describe the run-time behavior using scenarios, by the aspect *behavior*, which defines the behavior specification of the pattern. We do not use the aspect *implementation* as we provide models that shall enable a code generation later on. As we already explain

our models in the aspects **structure** and **behavior**, this aspect is no longer required. Next, we merge the contents of the aspects **example resolved** and **known uses** within the existing aspect **example**. Furthermore, we divide the aspect **see also** into the two aspects **alternative patterns** and **combinability** because they are—in our opinion—two different concerns. In addition, we add the new aspect **verification properties** to list and explain the verification properties (stated in MTCTL), which must hold for the pattern. Therefore, this aspect is similar to the aspect **constraints**, which is supported in the description format of Ramirez and Cheng [RC10]. However, in contrast to them, we also give at least one verified configuration within this aspect if the pattern defines at least one parameter. At last, we add the new aspect **search terms** that shall facilitate the search of a fitting pattern (see Section 6.5).

In summary, each Real-Time Coordination Pattern is described with the following aspects:

Name The name of the pattern.

Also Known As Existing synonyms or former names.

Overview A brief abstract of this pattern.

Context The context this pattern can be used in.

Problem The problems and design issues this pattern addresses.

Solution The solutions idea that tackles the given problem within its context.

Structure The structure of the pattern given as formal models and informal textual descriptions. This includes the properties of the roles (including their incoming message buffers), the message types, the incoming message buffer properties, the QoS assumptions of the role connector, and the parameters of the pattern.

Behavior The role behavior of the RTCP given as formal models (RTSCs) and informal textual descriptions.

Verification Properties A list of verification properties stated in MTCTL that this pattern has to fulfill and that an RTCP that adapts this pattern has to fulfill as well. Moreover, at least one verified configuration is stated that fulfills all given verification properties.

Consequences The consequences (benefits and liabilities) that arise if this pattern is used.

Examples Examples illustrate in which applications this pattern is useful.

Variants Possible variants of this patterns. Note that a variant only proposes minimal changes that do not effect the context or the problem and the solution in general.

Alternative Patterns A list of patterns (and corresponding explanations) that are related to this pattern and, therefore, may be used instead of it. For each alternative pattern, we state the differences as well the consequences of their use.

Combinability A list of patterns (and corresponding explanations) that may be used in combination with this pattern in order to fulfill more complex problems.

Search Terms A list of terms in verb-noun form that describe the coordination task of the pattern.

6.3.5 Example: The Pattern Acquire Assurance

In this section, we will exemplarily describe the pattern Acquire Assurance using our description format that we introduced in the previous section. The detailed description of all other patterns that we identified so far can be found in Appendix B.

Name Acquire Assurance

Overview This pattern enables a role, which we call *Assuree*, to acquire an assurance by another role, which we call *Assurer*. The pattern assumes that a safety-critical situation appears if the *Assuree* assumes that the assurance is still valid, but the *Assurer* currently does not assure it. Therefore, the pattern ensures that this situation never happens despite message delay and message loss.

Also Known As Its former name is Synchronized Collaboration [*DHT12; *DBHT12]. Though, it is not appropriate anymore as we realized that all our patterns define a collaboration and as most Real-Time Coordination Patterns synchronize the work of their roles. Moreover, a variant of this pattern is called Master-Slave-Assignment.

Context Time-bounded collaboration of independent systems.

Problem The pattern assumes that the role *Assuree* wants to execute a certain task or activate a certain mode (for a limited time). However, the task execution resp. the mode activation adds safety-critical hazards that may only be avoided if the role *Assurer* can assure a certain behavior. Consequently, the safety of the *Assuree* (and optionally the safety of the *Assurer*) depends on the behavior of the *Assurer*.

Therefore, as long as the task resp. the mode is active, the *Assurer* has to assure his behavior. In other words, the safety-critical situation occurs if the *Assuree* assumes that the assurance is still valid, but the *Assurer* does not. This situation may not occur even though (1) the transfer of the messages may have a delay, (2) the messages are stored in a limited buffer, and (3) the message transmission may be unreliable. While (1) could lead to the problem that the *Assurer* realizes too late that it has to assure certain behavior, (2) and (3) could lead to the problem that the *Assurer* never knows that it has to assure certain behavior (because the message was discarded from the buffer or got lost while transmission).

Solution Define an RTCP that distinguishes between the states that the assurance is given and not given. Moreover, the RTCP should act with different roles: one is the *Assuree* and one is the *Assurer*. The *Assuree* should be the role that is in charge to communicate that it needs the assurance and that it does no longer need it. In order to tackle the mentioned problems, the *Assuree* has to request for the assurance first. Only if the *Assurer* accepts the request, then the *Assurer* may execute its task resp. activate its modi. Otherwise, the *Assuree* could not be sure whether the *Assurer* really knows about the required assurance. If the assurance is no longer needed, it is sufficient that only the *Assuree* informs the *Assurer* about this (after it deactivates the task/mode). A request by the *Assuree* for deactivating and waiting for an acceptance message is optionally possible but not necessary.

Structure We show the structure of the pattern in the upper part of Figure 6.2. The pattern consists of the two roles Assuree and Assurer. Both roles are in/out roles. The role Assuree may send the messages requestAssurance and deactivateAssurance while the role Assurer may send the messages requestAccepted and requestRejected. Both roles have one buffer that store the all incoming messages of their role. The buffer of role Assuree has a size of 1, while the buffer of role Assurer has a size of 2. Both buffers shall never be forced to discard a message due to a full buffer. Moreover, the role connector has the following QoS assumptions: the connector is unreliable, the maximum message delay is given as a parameter, and the message order shall be preserved (i.e., overtaken messages will be discarded). Furthermore, there exist three parameters:

- `eval_time` defines how long the Assurer may evaluate the request.
- `maxMessageDelay` defines the maximum message delay.
- `offset` defines the additional time that the Assuree has to wait until it decides that no answer may be received by the Assurer anymore.

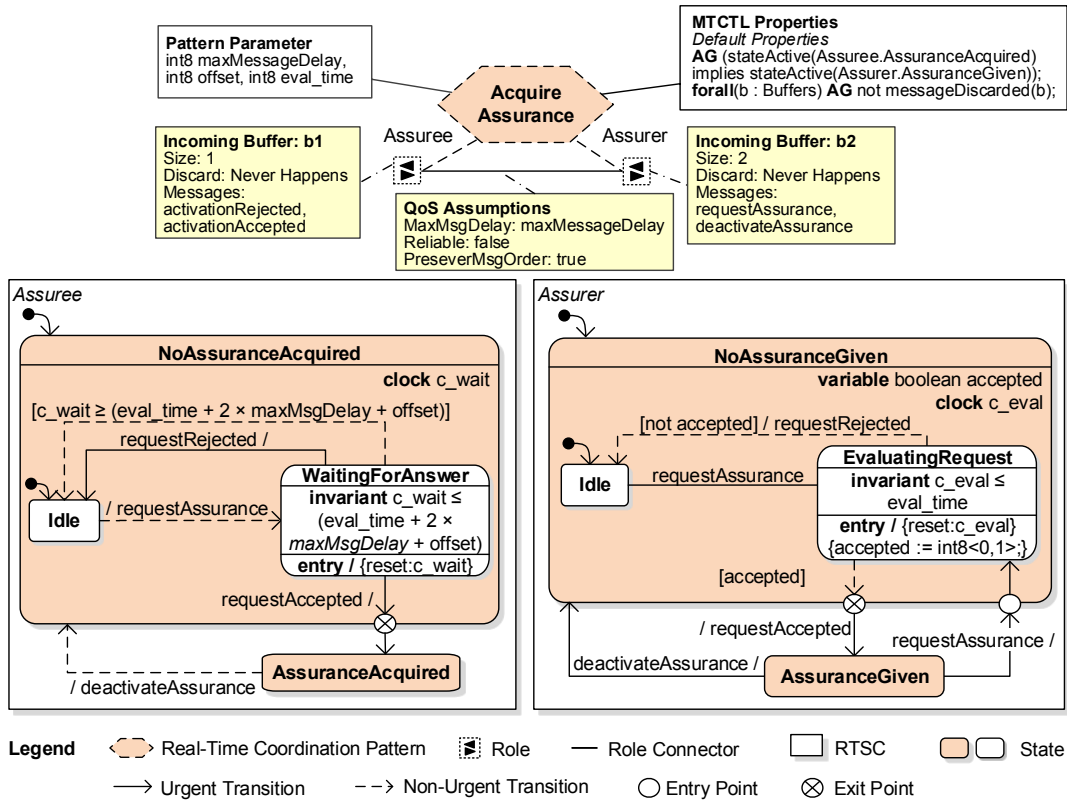


Figure 6.2: Real-Time Coordination Pattern Acquire Assurance

Behavior The behavior is shown in the lower part of Figure 6.2. Initially, both roles assume that the assurance is not given. The Assurer is passive and has to wait for the Assuree that it decides to send a request for acquiring the assurance. Thus, at any time, the Assuree may start the request (e.g., because the system wants to adapt its behavior).

If this is the case, the Assuree will change to state `WaitingForAnswer` and send the message `requestAssurance`.

If the Assuree receives no answer in a certain time (e.g., because the request message or the answer message of the Assurer got lost), it cancels its waiting and may send a new proposal. This waiting time is the sum of the evaluation time of the Assurer, two times the maximum delay of the connector, and an additional offset greater zero. The idea is that this waiting time is greater than the latest point in time where a message from the Assurer can arrive.

If the request message arrives, the Assurer changes to state `EvaluatingRequest` and has a limited time (defined via the pattern parameter `eval_time`) to evaluate and answer whether it accepts or rejects the request. If the Assurer rejects, it will send a corresponding message and return to state `Idle`. If the Assuree receives the reject message, it will change back to state `Idle` as well. However, if the Assurer accepts the request, it will send an acceptance message while activating its pre-defined behavior in order to provide the assurance (e.g., via an entry action in state `AssuranceGiven`). If the acceptance message arrives at the Assuree, it will change to the state `AssuranceAcquired` and then activate its predefined task resp. its predefined mode, e.g., via the entry action.

As soon as the Assuree decides that the assurance is no longer needed, the Assuree will deactivate its task / mode first (e.g., via the exit action of state `AssuranceAcquired`) before leaving this state, sending the deactivation message, and switching back to state `NoAssuranceAcquired.Idle`. If the Assurer receives the message, it will deactivate its assured behavior (e.g., via its exit action) and change back to state `NoAssuranceGiven.Idle`.

As the pattern allows that message loss may occur, it may be the case that the Assuree is in state `NoAssuranceAcquired.Idle` and the Assurer is in state `AssuranceGiven`. In that case, the Assuree may send a new request message. For handling this, the Assurer may consume the request via the transition from state `Assurer.AssuranceGiven` to `NoAssuranceGiven.EvaluateRequest`. As a consequence, this request is evaluated as a new request. The re-evaluation is especially important if the Assuree sends additional parameters with its message `requestAssurance`, which influences the decision of the result of the evaluation.

Noteworthy, in our formal model, the evaluation of the request by the Assurer is implemented via a non-deterministic choice such that the model checker will evaluate both possible evaluation results (true and false). We denote this choice via the expression $\langle 0,1 \rangle$.

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4).

Moreover, the pattern has to ensure that it invariantly holds that if the role Assuree is in state `AssuranceAcquired`, then the role Assurer is in state `AssuranceGiven` as well: $\text{AG}(\text{stateActive}(\text{Assuree.AssuranceAcquired}) \text{ implies } \text{stateActive}(\text{Assurer.AssuranceGiven}))$;

In addition, no buffer may discard a message: $\text{forall}(b : \text{Buffers}) \text{AG not messageDiscarded}(b)$;

A verified configuration of the parameters such that the pattern fulfills all verification properties simultaneously is as follows:

- eval_time: 600 ms
- maxMessageDelay: 199 ms
- offset: 2 ms

Consequences A benefit of the pattern is that it avoids a polling by the Assurer (whether it has to start or end the assurance) by putting the Assuree in the active role, i.e., the Assuree explicitly informs the Assurer when it shall start or end the assurance. As a second benefit, the Assurer cannot enforce that the task / mode of the Assuree shall remain active as the Assuree only sends a deactivation message and does not need an acceptance. Therefore, the Assuree can decide this independently of the Assurer.

As a first liability, both roles must have a pre-defined behavior and each role must exactly know the behavior that the other role will execute resp. assure. Otherwise, it would be unclear whether the safety-critical hazard can still occur. As a second liability, the Assurer cannot deactivate the assurance on its own but has to wait until the Assuree is willing to. As a third liability, if message loss may happen, it is possible the Assurer still assures the behavior but the Assuree does not need it anymore (i.e., the deactivation message was lost). As a fourth liability, the pattern assumes that the Assurer always activates the assured behavior successfully, i.e., it may never be the case that this action fails and, therefore, the behavior is not assured.

Examples The RailCab [HTS+08] (see Figure 6.3) is a novel transportation system. It is a small autonomous vehicle that drives on rails and may communicate with other systems (e.g., other RailCabs) via wireless asynchronous messages. Moreover, RailCabs can form a convoy at runtime without being mechanically coupled. This shall—among others—reduce the energy consumption of the RailCabs that drive behind another RailCab.

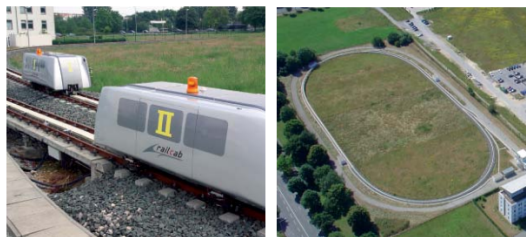


Figure 6.3: RailCab Prototypes (left) and RailCab Test Track (right) [*PDM+14]

In the scenario given in Figure 6.4, two RailCabs are driving on the same track in the same direction. The rear RailCab wants to create a convoy with the front RailCab. If such a convoy is formed, the rear RailCab will drive in close distance (i.e., only a few meters) to the front RailCab with a speed of up to 160 km/h. As long as the convoy is active, the front RailCab has to inform the rear RailCab via a message before it wants to brake hard. This is necessary as the distance sensors of the rear RailCab cannot detect a

hard braking of the front RailCab in time while driving at such a high speed and such a small distance. A collision would be the consequence.

Acquire Assurance enables to build a secure convoy that avoids the hazard of such a collision in convoy mode by making sure that the following assurance always holds: during convoy mode a brake message will always be sent by the front RailCab before hard braking. Therefore, the rear RailCab has to act as the *Assuree* and the front RailCab has to act as the *Assurer*.

The convoy protocol has been defined by Giese et al. in [GTB+03]. Though, this initial version did not support properties like message delay and message loss. A version that supports them is depicted in Figure 6.7.

Variants Instead of just informing that the assurance is no longer necessary, the *Assurer* could send a request to deactivate the assurance that the *Assuree* has to accept.

It is possible that the *Assurer* still assures the behavior though it is not longer necessary. This may happen due to message loss. For avoiding this, the *Assurer* can periodically request whether it may deactivate the assurance. If the *Assuree* agrees, then the assurance may be deactivated. As another alternative to avoid an unnecessary long assurance, the *Assuree* and *Assurer* may agree on a maximum time limit for the assurance, i.e., the *Assuree* defines a maximum time that the state *Assurance Acquired* may be active via a state invariant. As a consequence, the *Assurer* may leave its state *Assurance Given* as soon as this time limit is expired (via a state invariant that also considers the message delay and an additional transition with an appropriate clock constraint that leads to the state *NoAssurance Given*).

If the connector is reliable (i.e., message loss is not possible), then all stated verification properties are still useful. However, the pattern user should apply several changes to make the model more compact: (1) The state *NoAssurance Acquired* does not need the clock *c_wait* anymore. (2) The state *Assuree.NoAssurance Acquired.WaitingForAnswer* does not need its invariant and its entry action anymore. (3) The transition from state *Assuree.NoAssurance Acquired.WaitingForAnswer* to state *Idle* that defines the clock constraint should be removed. (4) The transition from state *Assurer.Assurance Given* to *NoAssurance Given.EvaluatingRequest* should be removed.

If the *Assuree* needs assurances from more than one *Assurer* at the same time, then this one-to-one pattern should be changed to a one-to-many one (see Figure 6.5; we do not show the new RTSC of role *Assurer* due to the small differences to the original RTSC). For doing this, the pattern user has to execute the following steps:

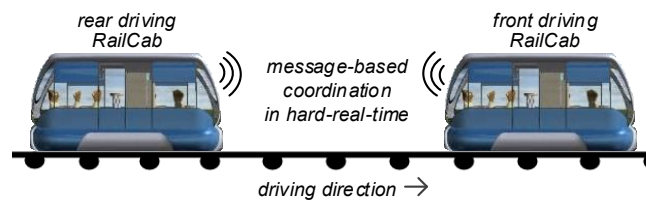


Figure 6.4: Convoy Scenario of the RailCab System (Based on [*PDM+14])

- He has to replace the fixed maximum role multiplicity range of the Assurer by a new parameter, which we call n .
- He has to change the RTSC of the Assuree as follows: (1) The Assuree should send the request to all assurers at once via the one-to-many communication schema multicast. (2) The transition from state `WaitingForAnswer` to `Idle` needs to be removed as message losses will be handled differently. (3) The receiving of the answer shall happen according to the one-to-many communication schema `multireceive`. In addition, the pattern user should replace the two answer messages (`requestAccepted` and `requestRejected`) by one message, which we call `answer`, that has a parameter of type `boolean` that indicates whether the request was accepted or not. Moreover, he has to define that the receiving does not start before the maximum delay for answering is over. By doing this, the Assuree can be sure that it does not miss an answer as it is still possible that a request message or an answer message got lost. (4) He has to add a new variable `allAccept` of type `boolean` to state `NoAssuranceAcquired`. The variable is reset via the entry action of state `WaitingForAnswer`. If the Assuree received all answers and all of them got accepted, then `allAccepts` becomes true, and the Assuree has to switch to state `AssuranceAcquired`. If the Assuree did not receive all answers or got at least one decline, then `allAccepts` becomes false, and the Assuree has to switch back to state `Idle`.
- He has to change the RTSC of role Assurer as follows: He has to replace the sending of message `requestRejected` by `answer(false)` and the sending of message `requestAccepted` by `answer(true)`.
- He has to replace the verification property **AG** (`stateActive`(Assuree.`AssuranceAcquired`) **implies** `stateActive`(Assurer.`AssuranceGiven`)); by **forall**($i : \text{Instances} < \text{Assurer} >$) **AG** (`stateActive`(Assuree.`AssuranceAcquired`) **implies** `stateActive`(Assurer.`AssuranceGiven`[i]));

As another alternative, the pattern user can adapt the pattern such that both roles can acquire an assurance from one another. In particular, each role can alternate between acquiring an assurance and giving an assurance. In an earlier version of the pattern catalog [DBHT12], this variant was a Real-Time Coordination Pattern on its own, which we called Master-Slave-Assignment. Here, the assurance is that a role will act as a slave that is willing to execute orders from the other role (which dynamically acts as the master) as long as the other role requires it.

Alternative Patterns If the pattern user only wants that the other role executes a certain task then he can use either the pattern Fail-operational Delegation (see Appendix B.2) or Fail-safe Delegation (see Appendix B.3).

Moreover, if the pattern user wants to block a certain action of another role without asking for permission, then he can also use the pattern Block Execution (see Appendix B.7).

Combinability As long as the assurance is active, several patterns may be used to exchange knowledge and to delegate tasks, e.g., Request Information (see Section B.1), Fail-operational Delegation (see Section B.2), Fail-safe Delegation (see Section B.3), Event-based Transmission (see Section B.4), and Periodic Transmission (see Section B.5). For

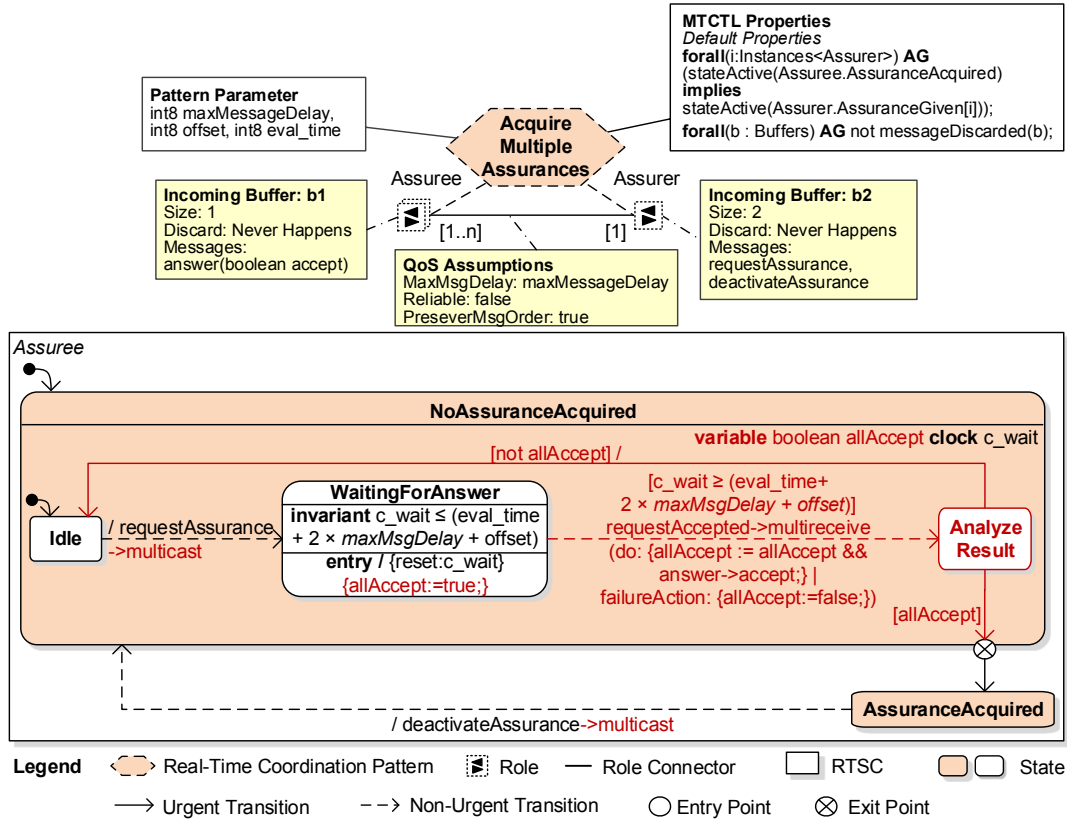


Figure 6.5: One-To-Many Variant of the Pattern Acquire Assurance (Changes to the One-to-One Pattern are Marked Red)

example, concerning the RailCab convoy example, the pattern user can use the pattern Fail-operational Delegation such that the front RailCab can order the rear RailCab that it has to brake.

Search Terms acquire assurance, assure state, promise safety, propose assurance, require behavior

6.4 Abstracting an RTCP to a Pattern

Our proposed method for abstracting an existing application-specific RTCP to a new application-independent Real-Time Coordination Pattern is illustrated in Figure 6.6. The only stakeholder for this process is the pattern specifier. This may be a single person but is preferable a group of experts that decide together concerning the pattern definition in order to increase the quality of the patterns.

At the beginning, the pattern specifier needs a set of verified application-specific RTCPs. Until now, the identification of new patterns is a manual task. Therefore, in *Step 1*, the pattern specifier has to select RTCPs with a common coordination problem that no pattern tackles yet.

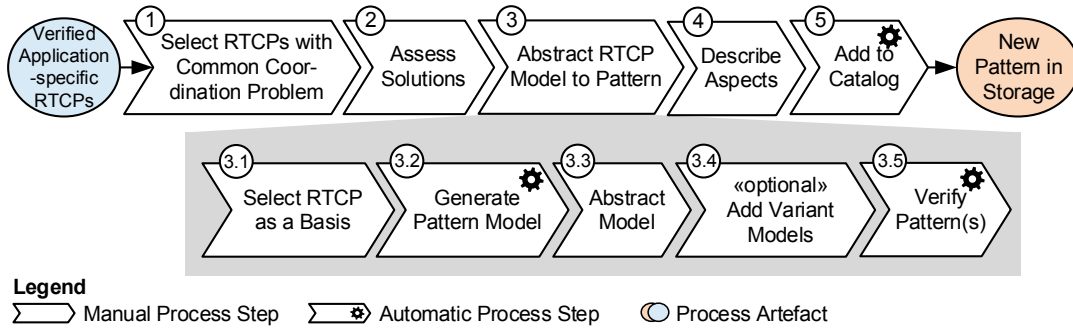


Figure 6.6: Process for Abstracting an RTCP to a Pattern

In *Step 2*, the pattern specifier shall assess the benefits and liabilities of each solution. Among others, this assessment has to take the RTSCs of the roles, the buffer properties, the QoS assumptions of the role connector, and the defined verification properties into account. Note that especially this task requires a huge set of experience in order to make correct statements.

Afterwards, in *Step 3*, the pattern specifier needs to abstract one of the RTCP models to the pattern model. We separate this step into four sub-steps. In *Step 3.1*, the pattern specifier has to select one RTCP model as a basis for the pattern model. Then, a new pattern model is automatically created out of the chosen one in *Step 3.2*. The abstraction of this model happens in *Step 3.3* and is a manual job. Note that this step does not define any mandatory adaptations. Though, we strongly recommend to execute the following tasks (other changes may also be useful):

- Rename all application-specific identifiers (statecharts, states, roles, role connector, buffer, message type, protocol name, variables, operations, synchronization channels) such that they are application-independent and as self-descriptive as possible.
- Replace all concrete values and time units that are application-specific by parameters (e.g., the message delay, invariants, clock constraints, guards, actions).
- Remove unnecessary elements that are only application-specific (e.g., message parameters, variables, operations, messages).
- Remove MTCTL properties that are application-specific only.
- Add additional MTCTL properties (e.g., all default properties) that ensure to preserve the idea of the pattern and its correctness.

Furthermore, if the pattern specifier defines pattern parameters, then he has to define at least one parameter configuration that shall fulfill all verification properties. This is crucial as we are not able to verify a pattern via model checking if it has parameters, i.e., a model checker would not be able to verify a pattern with all possible parameter bindings due to the state explosion problem (especially if more than one parameter is defined). Having at least one parameter configuration proves that at least one valid RTCP may be adapted from the pattern that fulfills all MTCTL properties.

Next, in *Step 3.4*, the pattern specifier may add a pattern model for each variant that is described in the aspect Variants of the informal description. This is especially recommended if the variant differs a lot. Otherwise, the pattern user has to make these changes by hand, which may introduce errors into the pattern model, e.g., if the variant description is ambiguous.

Step 3.5 is the final sub-step of *Step 3*. Within this step, the pattern and all its variants shall be verified with all specified parameter configurations for all given MTCTL properties. In particular, the pattern specifier can reuse our DSMC via the model checker UPPAAL (see Chapter 5) in order to execute the verification. Obviously, if any verification fails, then the pattern specifier has to adapt the model under verification, the parameter configuration, or the MTCTL properties until all verifications are successful.

Then, in *Step 4*, the pattern specifier describes the 15 description aspects (see Section 6.3.4). Noteworthy, our pattern metamodel (see Appendix E.4.1) enables to store all description aspects. Therefore, they are part of the pattern model itself and do not require a second storage concept. In particular, the pattern specifier can define all aspects using natural text. Additionally, for the aspects Alternatives and Combinability, our metamodel also enables to specify references to the corresponding patterns.

In the final *Step 5*, the pattern specifier has to add the new pattern (the formal models and the informal descriptions) to the pattern catalog. This catalog should be an online storage that is available for every software engineer via its development environment. In particular, we follow the idea by Henninger et al. [HA06; HC07] by providing a (rather simple) ontology for our patterns. In particular, our ontology is stated using the web ontology language (OWL, [OWL]). Storing the patterns and their descriptions into an ontology enables us to define several kinds of relationships between the patterns, and, therefore, facilitates the selection of a pattern by the pattern user. For example, we can link alternative and combinable patterns with each other, but we can also link the patterns by their search terms and by their context (we will explain this in Section 6.5). We present our ontology in detail in Appendix E.4.2. Noteworthy, this step is fully automatic as our tool suite transforms the pattern model (including the verification properties, the verified parameter configurations, and the informal descriptions) into data that the ontology understands.

In the following, we show the applicability of this method by explaining how we abstracted the pre-existing RTCP ConvoyCoordination, which we already introduced in Section 6.3.1 and in the aspect Example in Section 6.3.5, to the pattern Acquire Assurance. Therefore, we take the role as the pattern specifier.

As we already stated, the RTCP Convoy Coordination defines the establishing and dissolving of a convoy between two RailCabs. A convoy is beneficial for the rear RailCab as it can drive in the slipstream of the front RailCab and, therefore, can save energy. However, driving so close adds the hazard that the rear RailCab cannot react if the front RailCab suddenly brakes. Therefore, the rear RailCab needs the assurance from the front RailCab that it will send a braking message before the actual braking while driving in convoy mode. In general, the RTCP Convoy Coordination tackles the problem that a system sometimes requires assurances by another system in order to execute a certain task or to change a mode. As we have no pattern for this problem yet, we decide to define one based on this RTCP (Step 1 of our process).

Concerning Step 2, several variants of the RTCP Convoy Coordination exist (see [GTB+03; Bur06; Hir08; Tic09; Hen12; Pri13; Hei15]). One unpublished variant that follows our new syntax and semantics of RTCPs (see Chapter 3) is depicted in Figure 6.7. In contrary to the other variants (except [Hei15]), this variant considers message delay, message loss, and buffer discard strategies. Moreover, in contrast to some existent variants, it avoids the too strict constraint that the rear RailCabs needs an approval by the front RailCab in order to dissolve the convoy.

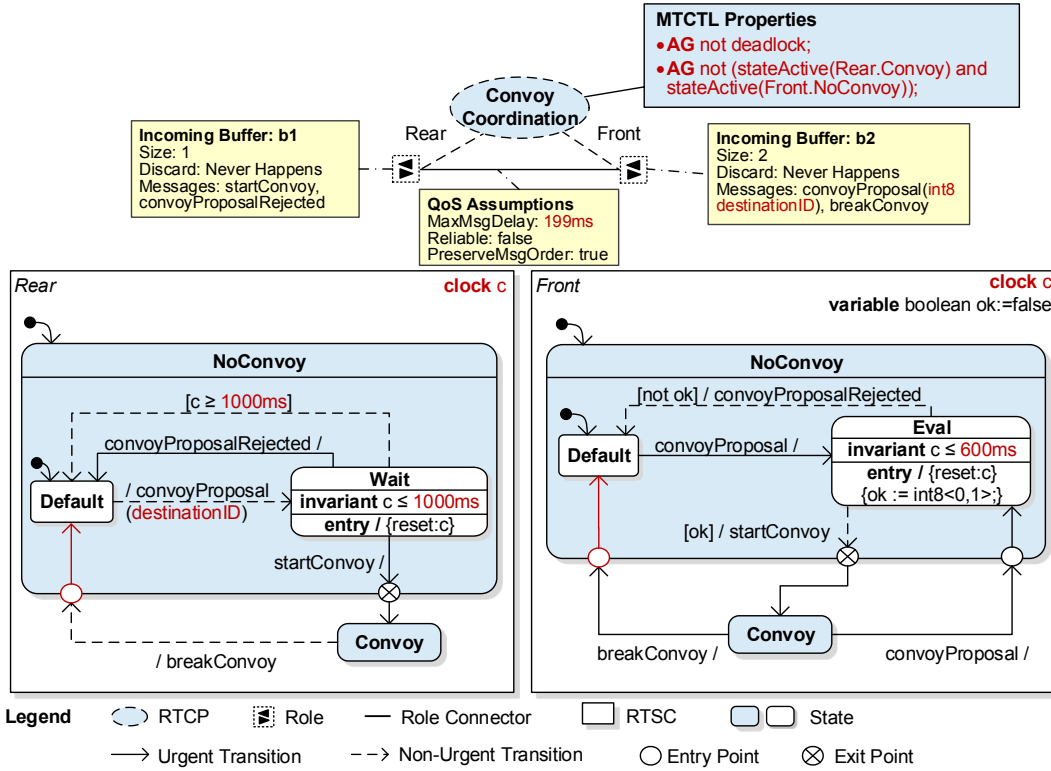


Figure 6.7: RTCP ConvoyCoordination (Changes While Abstracting—Except Renamings—are Marked in Red)

We decide to use the depicted variant as a basis for defining the model (Step 3.1) and generate the initial pattern model out of it (Step 3.2). Then, we abstract the pattern model to the one that we depict in Figure 6.2 (Step 3.3). In particular, we make the following changes: First, we replace all concrete values by parameters (`maxMessageDelay`, `eval_time`, and `offset`) or by expressions that use these parameters. For example, we replace the value 1000 ms of the invariant in state `Rear.NoConvoy.Wait` by the expression `eval_time + 2 × maxMessageDelay + offset`. Afterwards, we replace all occurrences of the parameter `maxMessageDelay` within both RTSCs with the keyword `maxMsgDelay`. Next, we add a parameter configuration with all three parameters, where each parameter has the value assigned that it replaced (`eval_time=600 ms`, `maxMessageDelay=199 ms`, `offset=2 ms`). Then, we remove the message parameter `destinationID` from the message type `convoyProposal` as this is only application-specific. Moreover, we remove in both RTSCs the entry point of state `Convoy` and the transition leading to the initial state as these elements are not necessary. Thus, the transitions from state `Convoy` has in both RTSCs the state `NoConvoy` as target. Afterwards, we add the remaining MTCTL default properties to the RTCP as all of them shall be fulfilled. Next, we add the MTCTL property **forall**(`b : Buffers`) **AG not messageDiscarded**(`b`); as the pattern assumes that both buffers never discard a message. Moreover, we reformulate the MTCTL property **AG not (stateActive**(`Rear.Convoy`) **and** **stateActive**(`Front.NoConvoy`)); into **AG stateActive**(`Rear.Convoy`) **implies stateActive**(`Front.Convoy`); as the important fact is that front is in convoy

mode as well. At last, we rename all identifiers, e.g., we change the name of the pattern to Acquire Assurance and the name of the roles from Rear to Assuree and from Front to Assurer.

In Step 3.4, we add a model for the one-to-many variant of the pattern, which we name AcquireMultipleAssurances. Next, in Step 3.5, we successfully verify that both patterns (the original and the variant) fulfill their MTCTL properties. Afterwards, in Step 4, we write the informal descriptions of the pattern (see Section 6.3.5) and define the references to the existing patterns (Master-Slave-Assignment is as an alternative pattern; Fail-safe Delegation, Fail-operational Delegation, Event-based Transmission, and Periodic Transmission are referenced as combinable patterns). Finally, in Step 5, we upload the pattern into our online pattern catalog.

6.5 Selecting and Adapting a Pattern to an RTCP

Our proposed method for selecting and adapting an existing application-independent Real-Time Coordination Pattern to a new application-specific RTCP is illustrated in Figure 6.8. The only stakeholder for this process is the pattern user.

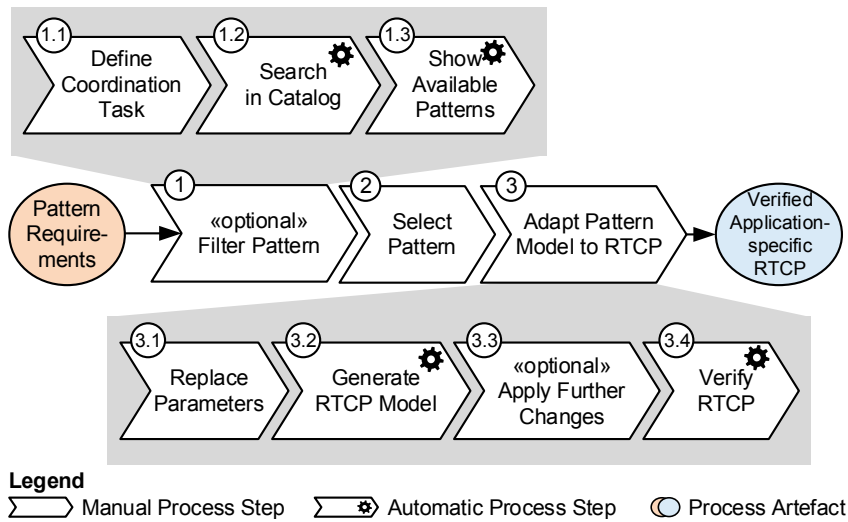


Figure 6.8: Process for Selecting and Adapting a Pattern to an RTCP

We expect that the requirements for the RTCP are available. Based on these requirements, the pattern user has to find a fitting pattern. As the number of available patterns may increase in the future, the pattern user may at first filter the list of all available patterns in *Step 1*. For doing this, we provide an automatic filtering mechanism, which requires the following three sub-steps: In *Step 1.1*, the pattern user has to describe the coordination tasks that the pattern shall fulfill. There are two possibilities to do this. First, the pattern user may define the coordination task by stating a search term in plain text. Second, the pattern user may decide to define the coordination task by selecting a verb-noun-pair, i.e., a wizard provides a list of verbs and a list of nouns from which the pattern user may choose from. Then, in *Step 1.2*, the pattern user can start the automatic search within the pattern catalog. If he stated plain text in Step 1.1, then we automatically search in all patterns whether any textual description of an aspect matches with the search string exactly. However, if he decided to

state a verb-noun-pair, then we compare the verb-noun-pair given by the pattern user with the verb-noun-pairs that are defined in the pattern aspect Search Terms. However, we do not only compare whether the verb and the noun are identical. Instead, we search for verb-noun pairs that match semantically, i.e., our ontology supports to define synonyms to each verb and each noun and to define words for each verb and noun that are more abstract or more concrete than the verb/noun. Therefore, we automatically generate and execute a query stated in SPARQL¹ that searches for related words within our ontology. As soon as the search is complete, we show the matching patterns to the pattern user in *Step 1.3*.

Afterwards, the pattern user has to select one pattern (or a variant of the pattern) in *Step 2*. Instead of providing an own selection process, we refer to the pattern selection process of Buschmann et al. [BMR+96, p. 368–370] that is also applicable for our pattern language as we reuse most of their description aspects (see Section 6.3.4). Next, the pattern user has to adapt the selected pattern model to his application in *Step 3*. This step consists of four sub-steps: At first, we automatically generate the RTCP model based on the given pattern model in *Step 3.1*. Then, in *Step 3.2*, the pattern user has to replace all existent parameters by concrete values, e.g., by defining concrete values on its own or by choosing one of the verified parameter configurations. Afterwards, the pattern user may apply further changes in *Step 3.3*. Among the possible changes, we distinguish between lightweight and heavyweight changes. Lightweight changes cannot change the intent of the pattern and cannot change its runtime behavior. These lightweight changes are:

- renaming identifiers (protocol, roles, buffer, RTSCs, states, asynchronous messages, clocks, variables, operations, . . .) to concretize their application-specific meaning,
- adding actions to states and transitions that do not influence the runtime behavior of the RTCP, and
- adding new message parameters, operations, variables. They may be used in existing and newly added actions but may not influence the runtime behavior of the RTCP.

Our lightweight changes match to the refinement definition of a timed bisimulation: “A timed bisimulation requires that the refined protocol includes exactly the same sequences of messages and specifies exactly the same time intervals as the abstract protocol. Therefore, timed bisimulation is a very strong refinement definition [...] and preserves all TCTL properties. The timed bisimulation is only fulfilled if the intervals of the refined behavior are identical to that of the abstract behavior [...]. Still, the timed bisimulation allows to modify the abstract protocol during the refinement step by inserting internal computations between the sent and received messages.” [*HBDS15, p. 14] In our case, the abstract protocol is the Real-Time Coordination Pattern; the refined protocol is the RTCP that is adapted from the pattern. Other refinement definitions presented in [*HBDS15], e.g., simulation, bisimulation, timed ready simulation, and relaxed timed bisimulation, are not suitable as they either do not consider clock constraints, allow a change of the maximum buffer size, or do not preserve reachability properties (see Section 2.1.2).

¹SPARQL [SPARQL] is a recursive acronym for SPARQL Protocol and RDF Query Language. RDF [RDF] is an acronym for the Resource Description Framework and enables to make statements about resources. Our ontology for patterns is defined using the web ontology language (OWL, [OWL]), which enriches RDF by additional model elements to increase the expressiveness. Therefore, queries in SPARQL may also be used to query knowledge in an OWL ontology.

We consider all other changes as heavyweight changes, e.g., adding / removing states, transitions, and messages or adding additional verification properties. The pattern user has to keep in mind that each heavyweight change could violate the intent of the pattern. Moreover, it cannot be assured that the provided MTCTL verification properties are still meaningful and sufficient for guaranteeing the safety and liveness of the resulting RTCP.

Finally, in *Step 3.4*, the pattern user has to verify the resulting RTCP using our DSMC via the model checker UPPAAL (see Chapter 5). Obviously, all stated verification properties should hold. Noteworthy, if the pattern user selected in Step 3.2 a verified parameter configuration and only applied lightweight changes, then this verification step may be left out as the runtime behavior compared to the verified pattern cannot be different.

In the following, we show the applicability of this method by developing the RTCP Overtaking, which we introduced in Section 1.1. Therefore, we will take the role as the pattern user. First of all, in Step 1, we choose to use our automatic filter mechanism. In particular, we choose to define a verb-noun-pair to describe the coordination task that this RTCP shall enable. In our case, the RTCP shall ensure that the overtakee does not accelerate any further while the overtaking happens. Therefore, the overtaker requires that the overtakee can guarantee it. Thus, we define the verb-noun-pair is “require guarantee”. We execute our catalog search via this pair. As a result, the catalog search suggests the pattern Acquire Assurance as the given pair is synonymous to the stated search term “acquire assurance”.

In Step 2, we review the proposed pattern by reading the descriptions and inspecting the formal models. As a result, we select the proposed pattern and adapt it to the RTCP Overtaking, which we illustrate in Figure 6.9. In particular, our wizard generates an initial RTCP model first (Step 3.1). Next, based on our requirements, we replace the three parameters in Step 3.2 as follows: `eval_time=3s`, `maxMessageDelay=1s`, and `offset=1s`. Afterwards, in Step 3.3, we apply additional lightweight and heavyweight changes. Concerning the lightweight changes, we rename most of the identifiers (e.g., we gave the RTCP the name Overtaking and rename the role Assuree to Overtaker and the role Assurer to Overtakee), we add the variable `speed` to the RTSC of role Overtaker, and we add the parameter `speed` to the existing message request.

Concerning the heavyweight changes, we apply the following changes: First, we restrict that the Overtaker may only take 20 seconds for the overtaking. We realize this by adding an invariant and a clock reset to state `Overtaker.Overtaking`. Second, we rename the clock `c_wait` to `c` and move it to the top level RTSC of this role. Third, we enable that the Overtakee must not stay in state `NoAcceleration` forever if the message finished does not arrive. We realize this by applying similar changes as we did to role Overtakee: We add an invariant and a clock reset to state `NoAcceleration`, add a non-urgent transition with an appropriate clock constraint from state `NoAcceleration` to state `NoOvertaking`, and rename the clock `c_eval` to `c` and move it to the top level RTSC of this role. Fourth, we define that the Overtaker shall inform the Overtakee as soon as it changed the lane. We realize this by implementing the pattern Event-based Transmission (see Appendix B.4) within the states `Overtaker.Overtaking` and `Overtakee.NoAcceleration`. In particular, we define that the Overtaker sends the message `laneChanged`, which is sent after 1500 to 4500 milliseconds.

After inspecting our changes, we conclude that we did not change the intent of the pattern and that the existing verification properties are still useful and correct. Finally, we verify the complete RTCP Overtaking using our DSMC. As a result, all verification properties are

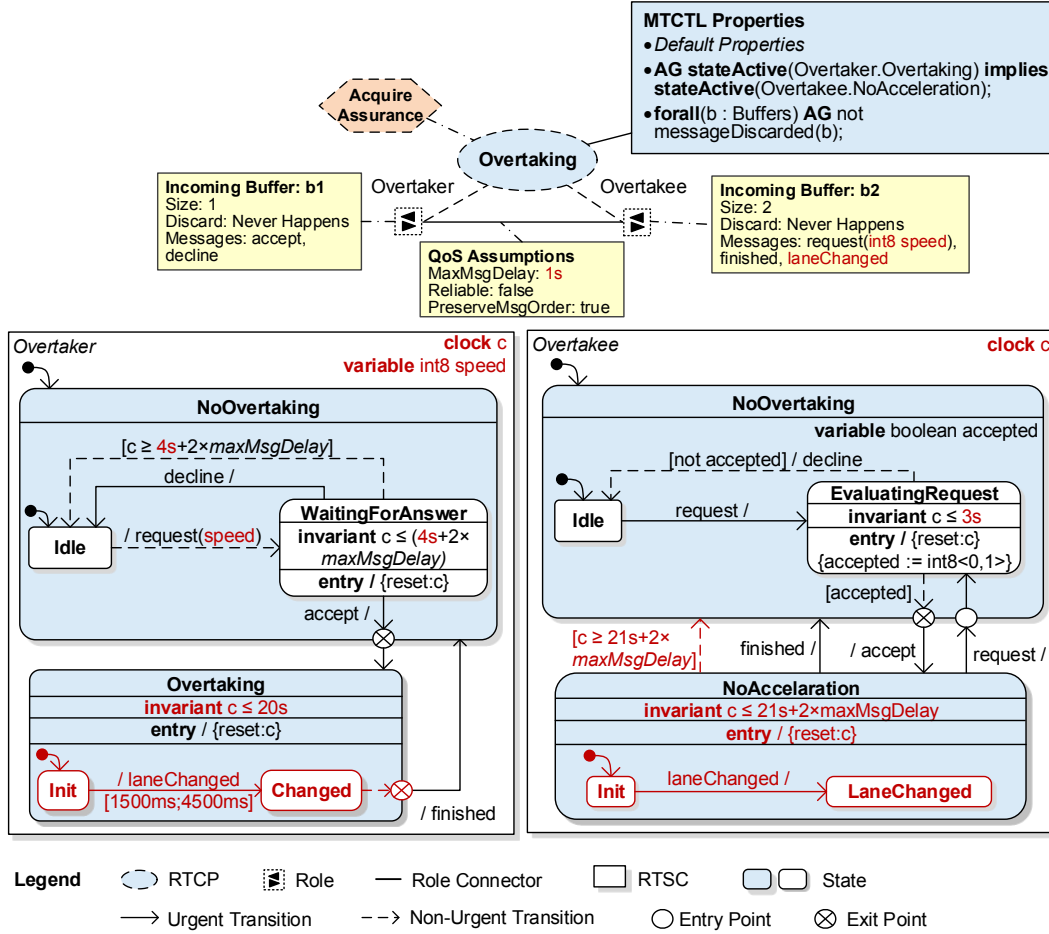


Figure 6.9: RTCP Overtaking Adapted from the Pattern Acquire Assurance (Changes While Adapting—Except Renamings—are Marked in Red)

fulfilled. Therefore, we have successfully defined the verified application-specific RTCP Overtaking based on the Real-Time Coordination Pattern Acquire Assurance.

6.6 Implementation

In this section, we present excerpts of our prototypical implementation of an editor for Real-Time Coordination Patterns. Moreover, we present the corresponding transformations and dialogs for adapting a pattern to protocol and abstracting a protocol to a pattern. Our implementation is integrated into the MECHATRONICUML Tool Suite [*DGB+14].

In particular, we present the user interface in Section 6.6.1 and the plugin structure in Section 6.6.2. We show and explain our metamodel for specifying Real-Time Coordination Patterns in Appendix E.4.1.

6.6.1 User Interface

Our user interface for modeling a Real-Time Coordination Pattern is analogously to the user interface of an RTCP (see Section 3.7). In fact, both user interfaces share a lot of common metamodel elements as well as implementation code. We show a screenshot of our Real-Time Coordination Pattern editor in Figure 6.10, which depicts the pattern **Acquire Assurance**. The editor follows the graphical syntax of a Real-Time Coordination Pattern (compare our illustration of Pattern **Acquire Assurance** in Figure 6.2).

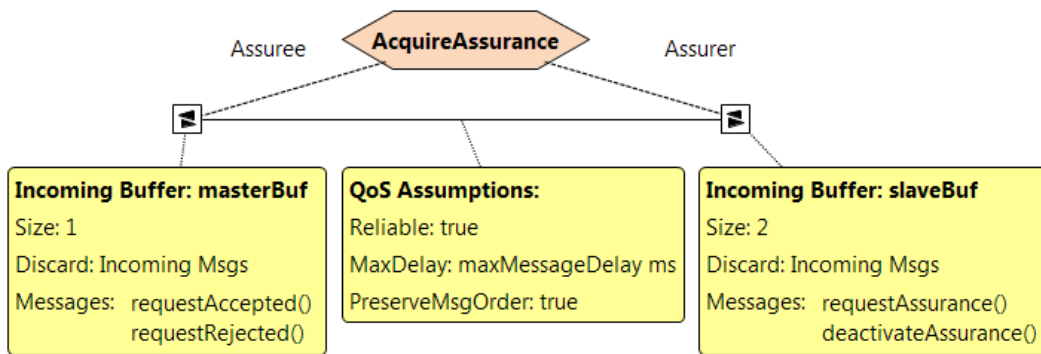


Figure 6.10: User Interface for Modeling the Structure of a Real-Time Coordination Pattern

The pattern specifier can specify the MTCTL properties for a Real-Time Coordination Pattern in the same manner as for an RTCP (see Figure 4.4). Moreover, like the RTCP editor, this editor supports to generate the MTCTL default properties. In particular, we generate the same set of default properties.

Our prototypical implementation supports all automatic process steps for abstracting an RTCP to a pattern (see Section 6.4) and for adapting a pattern to an RTCP (see Section 6.5). For example, we provide wizards for transforming the model of an RTCP to a pattern (Step 3.2 in Figure 6.6), where—among others—the pattern specifier may replace variables of the RTSC by pattern parameters. Moreover, we provide a wizard for transforming the model of a pattern to an RTCP (Step 3.1 in Figure 6.8), where—among others—the pattern user has to bind all pattern parameters to concrete (time) values either by selecting a verified configuration or by defining concrete values on its own.

For the formal verification of a Real-Time Coordination Pattern via DSMC (Step 3.5 in Figure 6.6 and Step 3.4 in Figure 6.8), we cannot simply reuse the wizard for executing a DSMC for an RTCP (see Section 5.7.1) because our DSMC does not support pattern parameters. Therefore, we require that the software engineer defines a concrete value for each parameter of the pattern first. For doing this, we reuse the wizard that we provide for adapting a pattern to an RTCP. Afterwards, we reuse our implementation to automatically transform a Real-Time Coordination Pattern model into an RTCP model. Only after this, we can reuse the wizard that we defined for verifying an RTCP via DSMC.

6.6.2 Plugin Structure

We show an excerpt of the plugins that we developed and adapted for our prototypical implementation in Figure 6.11.

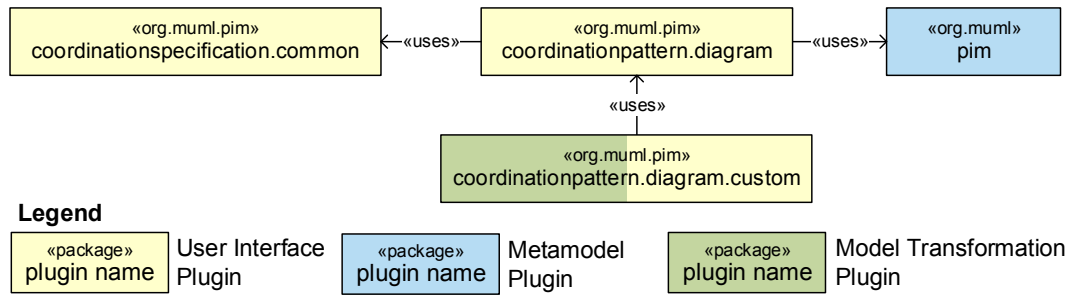


Figure 6.11: Plugins used for implementing Real-Time Coordination Patterns

First of all, we extend the already existing metamodel of the platform-independent modeling within the plugin `pim`. Among others, we define a new package, which we call `pattern`, that contains all relevant model elements for a Real-Time Coordination Pattern.

We provide three plugins for the user interface. The plugin `coordinationpattern.diagram` defines the GMF-based visual editor for specifying our Real-Time Coordination Patterns in their concrete syntax. Note that we store the commonalities with the visual editor for RTCPs (see Chapter 3) within the plugin `coordinationspecification.common`. Furthermore, the plugin `coordinationpattern.diagram.custom` exists. It contains manual written code for the visual editor due to limitations of GMF (it does not enable all editor functionalities that we need). In addition, this plugin contains our wizards and corresponding model transformations in QVTo to automatically transform a pattern to an RTCP and vice versa.

6.7 Evaluation

We conduct a case study based on the guidelines by Kitchenham et al. [KPP95] and Runeson et al. [RHRR12] for evaluating our approach. In particular, we evaluate our concept of Real-Time Coordination Patterns for realistic examples within the domain of CPSs but do not aim for generalizations.

Our evaluation is structured as follows: First, we present the context of our case study in Section 6.7.1. Then, we set our hypotheses in Section 6.7.2 and prepare the validation in Section 6.7.3. Afterwards, we validate the hypotheses in Section 6.7.4. Finally, we discuss the threats to validity in Section 6.7.5 and analyze our results in Section 6.7.6.

6.7.1 Case Study Context

The objective of our case study is to evaluate whether our patterns are reusable. Thus, we evaluate whether we fulfill the requirement R1 that we stated in Section 6.2.

We conduct our case study based on two different CPSs that we consider as realistic representatives of the domain as both systems consist of sub-systems that communicate with asynchronous messages under real-time constraints in a safety-critical environment. In the following, we give a brief description of these systems.

The first system is called *Cooperating Delta Robots* (see Figure 6.12) and was developed in the project ENTIME [*GTS+14] before we created our concept of Real-Time Coordination Patterns. This system consists of two independent robots that shall juggle a ball resp. pass a

ball to each other. Though, the ball may not touch the ground. Moreover, the robots cannot move their position and are not equipped with a camera system. Instead, the robots can only move their racket, can sense the ball by sensors on the racket while hitting the ball, and can communicate with each other via asynchronous messages. In particular, by utilizing their sensor data, the robots shall predict the flight trajectory and use this information either for themselves (if the robot shall juggle the ball) or to send it to the other robot via a message. In the latter case, the second robot must receive this prediction and compute and execute an appropriate strike trajectory before the ball arrives.

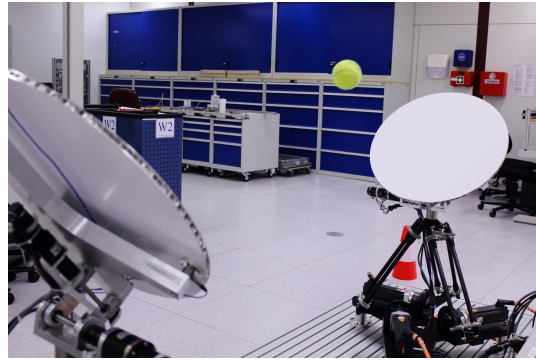


Figure 6.12: Cooperating Delta-Robots

The second system is called *Smart Universal Power Antenna* (SUPA, <http://www.supa-technology.de>). The idea of this system is to provide power for electronic devices cordless and anywhere on a big surface by using induction. For example, the system shall provide power for lamps, laptops, mobile phones, and monitors by locating the power supply on the underside of the tabletop (see Figure 6.12). In particular, the induction is realized by a network of antennas. This limits the radiation levels as a coil must only be active if a device is putted above it. Furthermore, each antenna may not only transmit energy but also data. However, each electronic device requires a different amount of energy. Otherwise, the hardware may get damaged. Therefore, the energy provider has to periodically check (1) whether a receiver is available for an antenna, (2) whether a receiver is no longer available, or (3) whether the receiver changed. In the cases (1) and (3), the energy provider has to request the power class of the system and has to react on the answer. Moreover, the energy provider has to react on several problems, e.g., whether the antenna does not deliver the correct amount of energy. Consequently, the energy receiver must be able to send its power class and to check whether the received energy is correct and send an error message to the power provider if not. The requirements for SUPA have been defined independently from us, i.e., the people that defined the requirements did not know our concept for Real-Time Coordination Patterns.

6.7.2 Setting the Hypotheses

Our case study has the following evaluation hypothesis:

- H1** The reusability rate of our patterns is 10%, i.e., for every 10 RTCPs, at least one of our patterns is applicable (applicable means that the design problem and the solution idea of the pattern fit to the given coordination task).



Figure 6.13: Smart Universal Power Antenna (SUPA, Images taken from: <http://www.supa-technology.de>)

H2 If a pattern is applicable, then the resulting RTCP model is in most cases still similar to the proposed pattern model.

Our hypothesis H1 is very conservative as the basis of existing RTCPs is with less than 50 considerable small. Therefore, it is very likely that several other patterns exist that are not identified yet. Moreover, we do not expect that each RTCP may be based on a pattern at all.

Concerning hypothesis H2, we want to evaluate how mature our formal models of the pattern already are. Typically, the less heavyweight changes the pattern user has to apply, the more development time can he save and the less errors he may make.

For validating both hypotheses, we give two students, which study computer science at bachelor level, the task to develop the software of the systems via MECHATRONICUML (each student was assigned to one system). Among others, they shall develop the RTCPs and use the patterns if they are applicable. Thus, we want that they execute the process for selecting and adapting a pattern to an RTCP in the role of a pattern user (see Section 6.5). After they specify the complete system (the RTCPs but also the component model), we analyze the result, e.g., whether the correct pattern was selected and whether the changes made by the student while adapting the pattern to an RTCP did not change the idea of the pattern. As we are experts of MECHATRONICUML and the existing pattern catalog, we can state whether a pattern is applicable for a given coordination task, whether a resulting RTCP still preserves the idea of the pattern, and whether an RTCP model is still similar to a pattern model. For the latter one, we mainly count the number of heavyweight changes and inspect the extent of each change in detail. We expect that we fulfill both hypotheses.

Noteworthy, at the time of executing this case study, the MECHATRONICUML Tool Suite did not enable to execute the process for selecting and adapting a pattern to an RTCP, did not enable to specify MTCTL properties, and did not provide our DSMC yet. Therefore, our focus in this case study is only the selection of a pattern and the manual adaptation to an RTCP.

6.7.3 Preparing the Validation

Both students had basic experience regarding model-based development (e.g., they have experience with UML state machines and UML component diagrams [UML]) and already knew the design patterns of Gamma et al. [GHJV95]. However, they were not familiar with MECHATRONICUML. Therefore, we gave them detailed introductions and provided them

with existing MECHATRONICUML models as well as our former version of the pattern catalog [*DBHT12] (as a PDF). Afterwards, we gave the students the informal requirements of the system's software as they both were not involved in defining the requirements.

6.7.4 Validating the Hypotheses

The results of both students are available in their bachelor's theses [San11; Win14]. The bachelor's thesis of Marcel Sander was supervised by me while the bachelor's thesis of Alexander Winkler was supervised by my colleague Uwe Pohlmann, who is an expert in MECHATRONICUML and Real-Time Coordination Patterns like me.

For the software of the Cooperating Delta Robots, our first student developed a total amount of 17 RTCs. Out of these 17, our student was able to reuse 2 different patterns for 5 RTCs. Thus, we have a reusability rate of approximately 29%. In particular, he applied the pattern Fail-operation Delegation four times and the pattern Request Information once. Furthermore, he only replaced the parameters of the patterns by concrete values and made only lightweight changes that did not effect the pattern intent or the original runtime behavior (i.e., the student renamed identifiers, and added message parameters, variables, and actions).

Afterwards, we evaluate the work of the first student. As a result, we conclude that it was correct to apply these patterns for these RTCs. Moreover, we agree with the changes that he made.

As a side note, the first student proposed six new patterns, whereby five of them were accepted as pattern candidates or valid variants of existing patterns. We list them in the following.

- Master-Slave-Assignment, which we accept as a variant of the pattern Acquire Assurance (see Appendix 6.3.5), occurred once in its system model.
- Turn Transmission, which we accept as a variant of the pattern Alternating Lock (see Appendix B.6), occurred once in its system model.
- Event-Listener, which we now call Event-based Transmission (see Appendix B.4), occurred nine times in its system model (eight times as a one-to-one and one time as a one-to-many variant).
- A one-to-many variant of Fail-operational Delegation (see Appendix B.2) occurred once in its system model. We list it now as an official variant but we recommend to use one-to-many communication schemata instead of modeling the multi role with an explicit subrole RTSC.
- Start-Stop, which we now call Block Execution (see Appendix B.7), occurred once in its system model.

For the software of the SUPA, our second student developed a total amount of four RTCs. Each protocol is based on an existing Real-Time Coordination Pattern. Thus, we have a reusability rate of 100%. In particular, the student used the pattern Fail-Operational Delegation twice, the pattern PeriodicTransmission once, and the pattern LimitObservation once as well. Furthermore, the second student also replaced the parameters of the patterns by concrete values. In addition, he applied several lightweight and one heavyweight change. We concluded that all lightweight changes did not effect the pattern intent or the original runtime behavior (i.e., the student renamed identifiers, and added message parameters, variables, and actions). The only heavyweight change was that the student removed the invariant of the initial state of

the role Provider. Therefore, we still consider the RTCP model as similar to the pattern model. In fact, we accept his heavyweight change as a valid variant of the pattern.

Afterwards, we evaluate the work of our second student as well. As a result, we conclude again that it was correct to apply these patterns for these RTCPs. Moreover, we agree with the changes that he made.

6.7.5 Threats to Validity

According to Runeson et al. [RHRR12], we classify our threats to validity into three parts: construct validity (is the case study designed correctly to answer the hypotheses?), external validity (what hinders us to generalize our findings?), and reliability (would the result be the same with another set of researches?). In the following, we discuss our threats based on this classification schema and report how we consider these threats in the design of our study. Noteworthy, Runeson et al. also define a fourth class called internal validity that we do not need as we do not examine causal relations.

6.7.5.1 Construct Validity

- Both students could have misunderstood the idea and the solution of a pattern. During the execution of their bachelor's theses, we provided support by explaining the patterns and answering the questions of the students. Nevertheless, it might be the case that not all misunderstandings were solved.
- Both students were aware that we want to reuse as much patterns as possible. Though, we defined that the results of their bachelor's theses do not depend on the number of pattern the students reuse or whether they reuse a pattern at all. Nevertheless, it might be the case that they reused a pattern even if they were not convinced that the pattern is the best solution. Sander does not state why he chose an existing pattern but Winkler gives reasonable explanations for his pattern selection. Therefore, in the case of Winkler's work, we are convinced that this threat does not apply.

6.7.5.2 External Validity

- We only consider two systems for our evaluation. The requirements for the system Coordinating Delta Robots were defined before we designed our patterns within the project ENTIME [*GTS+14] together with PhD students of mechanical engineering and industrial experts for mechatronic systems. The requirements for the system SUPA were defined independently from us by Fraunhofer ENAS, who are experts in mechatronic systems. Therefore, we consider our systems as realistic examples. However, other realistic examples could exist, where none of our patterns is applicable or where several heavyweight changes have to be applied.
- It might be that other software engineers would have designed our two systems in a different way than our two students such that our patterns could not have been applied.

6.7.5.3 Reliability

- The decision whether a pattern is applicable is partially subjective. In order to lessen this threat, I did not decide this on my own but with the help of my colleague Uwe

Pohlmann, who has several years of experience with MECHATRONICUML and Real-Time Coordination Patterns.

6.7.6 Analyzing the Results

In both examples, we have a reusability rate that is significantly higher than 10% (the Cooperating Delta Robots had 29%, the SUPA had 100%). Therefore, we consider our hypothesis H1 as fulfilled. Based on these high reusability rates, we are convinced that our pattern catalog already contains some of the most important patterns. Consequently, we believe that our current pattern catalog already facilitates the work of the software engineer to create valid (w.r.t. the syntax of the DSL and the verification properties) and well-designed RTCPs.

Moreover, we consider our hypothesis H2 as fulfilled as well, as nearly all changes to the patterns were only lightweight changes or replacing parameters by concrete values. Again, this shows that each pattern definition is already mature and actually defines an accepted solution for a reoccurring design problem. Moreover, even though the selection of a pattern and its adaption to an RTCP takes time, we are convinced that the software engineer saves development time as the number of additional changes are considerably small in most cases and the resulting RTCPs already provide well-designed and reasonable solutions for their problems.

In summary, it seems that our set of Real-Time Coordination Patterns provides reusable solutions for commonly reoccurring problems when designing verifiable RTCPs of CPSs. Therefore, we expect that our patterns should be able to increase the efficiency of the software engineer, the correctness of the resulting RTCPs, and their understandability.

6.8 Limitations

Our Real-Time Coordination Patterns underlie the following conceptual limitations:

- Currently, we do not support the pattern specifier in identifying a new pattern, e.g., we have no guidelines for this step. Therefore, the pattern specifier has to use his experience-based knowledge to conclude whether the RTCP provides a good solution for a commonly occurring problem.
- If the pattern specifier wants to add a new pattern, then he has to manually compare at first whether this pattern does not already exist. Thus, we currently do not provide automatic steps nor guidelines for this task.
- We have no concepts yet to detect a pattern within a MECHATRONICUML specification if it is not explicitly referenced. This could help the software engineer to understand the purpose of an unknown RTCP.
- We do not enable that a parameter of a verified configuration is valid for a certain range of values. This could significantly reduce the maximum number of possible distinct verified configurations for one pattern.
- Our current ontology for storing the knowledge of a pattern is rather simple. Currently, the ontology does not contain the semantics of the description aspects, the models, and the verification properties of a pattern but only describes the functionality using verb-

noun-pairs and stores the relationships between the pattern (alternatives and combinable patterns).

Moreover, we currently have the following implementation-specific limitations:

- The functionalities for storing, editing, searching, and loading a pattern via an ontology and appropriate wizards are not part of the MECHATRONICUML Tool Suite but were only developed in a variant of this tool suite by the project group SafeBots III [ABB+13]. Instead, the MECHATRONICUML Tool Suite only enables to store the patterns in a separate EMF-based model file and provides functionalities to import and export them.

6.9 Related Work

First of all, no related design pattern catalog exists that aims at the same domain as we: the formal (state-based) specification of an asynchronous message exchange under real-time constraints between independent (cyber-physical) systems while respecting QoS assumptions like message delay and message loss and enabling formal verification, e.g., via model checking. However, related work defines informal and formal design patterns that are similar to our domain.

6.9.1 Informal Design Patterns

“Patterns regarding the coordination and communication between classes, components, and systems already exist and were a great help for defining our own patterns. However, most of them only illustrate the communication informally using sequence diagrams. If at all, they only define simple timing behavior. Examples for these are the patterns *Chain of Responsibility*, *Command*, and *Observer* by Gamma et al. [GHJV95] and the patterns *Master-Slave*, *Forwarder-Receiver*, *Client-Dispatcher-Server*, and *Publisher-Subscriber* by Buschmann et al. [BMR+96].” [*DHT12]

Several approaches define design patterns for real-time systems, e.g., Douglass [Dou99; Dou02] defines so-called real-time design patterns for—among others—the collaboration between components. They describe the behavior of several patterns via sequence diagrams and state machines, e.g., the patterns *Watchdog* and *Rendezvous*. In particular, the sequence diagrams and state machines include real-time constraints and define an exchange of events.

Real-Time Coordination Patterns are design patterns for the asynchronous message exchange in the domain of CPSs. Other domain-specific languages also provide patterns for a message-based communication. “For example, AGENTUML is a modeling language [for the domain of multi-agent systems] to specify agent interaction protocols (AIP) [BMO01]. For such protocols the Foundation of Intelligent Physical Agents defines so-called protocol templates, e.g., the *Propose Interaction Protocol*, which proposes an interaction that can be accepted or rejected.” [*DHT12] In particular, AIPs combine sequence diagrams with the notion of state diagrams. Therefore, they enable to specify the exchange of messages and to define time constraints. Moreover, they enable to specify an interaction between two agents (as a one-to-one communication) but also between several agents (e.g., as a one-to-many communication). In summary, AIPs are quite similar to our RTCPs but miss our domain-specific properties like asynchronous messages (including the need of message buffers) and continuous clocks (including state invariants).

In contrast to all these pattern systems, we provide formal models for each pattern. Therefore, we directly support the formal verification of a pattern and the resulting RTCP via DSMC using parameter configurations. Moreover, we provide formal verification properties that the pattern and the resulting RTCP has to fulfill. Hence, the pattern user does not need to come up with a set of useful properties on his own.

6.9.2 Formal Design Patterns

Gil de la Iglesia [dIIgl14] provides “[...] a set of formally specified templates for the specification and verification of self-adaptive behaviors of a family of distributed self-adaptive systems. The templates are based on the MAPE-K reference model (Monitor-Analyze-Plan-Execute plus Knowledge). The templates comprise: (1) behavior specification patterns for modeling the different MAPE components of a feedback loop, and (2) property specification patterns that support verification of the correctness of the adaptation behaviors. The target domain are distributed applications in which self-adaptation is used for managing resources for robustness and openness requirements. The templates are derived from expertise with developing several self-adaptive systems [...]” [dIIgl14] He demonstrates the reusability of his templates with several of case studies. In summary, his pattern concept is similar to ours. Though, in contrast to his work, we do not focus on MAPE components but on coordination protocols for asynchronous message exchange. In addition, we state verification properties for each pattern but he provides property specification patterns that do not refer to a particular template but to the complete system. Moreover, we provide a more detailed description catalog (we additionally describe consequences, examples, alternatives, variants, etc.). Furthermore, we also provide concepts and process steps to search patterns, to adapt them to a protocol, and to abstract existing protocols to a pattern.

Ramirez and Cheng [RC10] provide 12 formal design pattern for the design of self-adaptive systems. They identified their patterns by studying over thirty adaptation-related implementations. Their description template is similar to the one of Gamma et al. [GHJV95]. In particular, they describe the structural aspects of the patterns using class and component diagrams. Furthermore, the behavioral aspects are described using state and sequence diagrams. Like us, they provide verification properties that their models have to fulfill. In their case, the property language is LTL [BK08]. The patterns are categorized into monitoring, decision-making and reconfiguration patterns. In summary, their pattern approach is very similar to us. However, they tackle another design aspect by focusing on self-adaptation. Thus, one cannot directly apply these patterns for the design of CPSs as they do not consider aspects like real-time constraints and asynchronous message exchange (including buffers and QoS assumptions).

Taibi et al. [Tai07] describe several approaches regarding the formalization of patterns and their subsequent adaptation and refinement, but they do not focus on asynchronous communication or coordination protocols of CPSs. All these approaches do not focus on model checking design patterns resp. the models that are adapted from the pattern. They only provide own languages or use OCL [OCL] in order to define static constraints. Therefore, in contrast to us, they do not provide verification properties concerning the behavior, e.g, given in a CTL-based language like MTCTL, that the patterns as well as the resulting models that adapt this pattern shall fulfill.

As already discussed in Section 4.9.6, several pattern catalogs exist for verification properties, e.g., by Dwyer et al. [DAC98; DAC99] and Konrad and Cheng [KC05]. However, in contrast to them, we do not only provide a fitting verification properties for a given problem but also the appropriate design model.

Kemper [Kem11] defines a model-based formalism that enables formal verification. She calls this formalism “Real-Time Coordination Patterns”. However, her patterns are no software design patterns like ours that follow the idea of Gamma et al. [GHJV95]. Instead, their patterns describe formally which information two specific components have to exchange under real-time constraints. Therefore, their patterns do not describe a solution for a reoccurring design problem.

6.10 Summary

In the following, we summarize to what extent this thesis and the most relevant approaches that we discussed in Section 6.9 fulfill the requirements that we stated in Section 6.2. As a reminder, the requirements were based on our own experience and are not proven, e.g., by a systematic literature review or requirements workshops.

	<i>this thesis</i>	<i>Gamma et al.</i>	<i>Buschmann et al.</i>	<i>Douglass</i>	<i>AGENTUML</i>	<i>Gil de la Iglesia</i>	<i>Ramirez et al.</i>
R1	✓	(✓)	(✓)	(✓)	(✓)	×	×
R2	✓	✓	✓	✓	✓	✓	✓
R3	✓	×	×	×	×	✓	✓
R4	✓	×	×	×	×	✓	✓
R5	✓	×	×	×	(✓)	(✓)	✓
R6	✓	✓	(✓)	(✓)	×	×	×
R7	(✓)	×	×	×	×	×	(✓)

Table 6.1: Comparison of the Requirements Fulfillment for Design Patterns of CPSs

We provide an overview of the requirements fulfillment in Table 6.1 (Legend: ✓ = fulfilled, (✓) = partially fulfilled, × = not fulfilled). As the table shows, this thesis fulfills nearly all requirements (one is only partially fulfilled) that we stated at the beginning of this chapter. Moreover, no other approach fulfills all requirements. A combination of the related approaches could theoretically fulfill the requirements to the same extent as our approach but would—according to our experience—require huge conceptual efforts.

The most important requirement is R1, which states that the pattern formalism shall provide proven and well-designed solutions for reoccurring design problems in the context of verifiable coordination protocols of CPSs. The assessment of this requirement is subjective as it is hard to decide at which point a solution is proven, whether it is a well-designed solution or not, and whether it is really reusable. In our opinion, this thesis fulfills R1 because of four reasons: (1) Our patterns are especially designed to consider the relevant aspects of our domain. (2) All our solutions are originated from existing RTCPs. Therefore, they are a proven solution.

(3) As we are experts for specifying verifiable coordination protocols of CPSs, we can decide whether the solution is well-designed. (4) In our evaluation in Section 6.7, we show that our pattern are reusable. In contrary, no related approach that we studied completely fulfills this requirement. Gamma et al., Buschmann et al., Douglass, and AGENTUML provide patterns concerning the exchange of data. However, they do not or only partially consider the relevant aspects of our domain like real-time constraints or asynchronous message exchange (including the specification of message buffers and QoS assumptions).

All pattern formalisms provide a textual description and, therefore, fulfill requirement R2. The requirements R3 (provide formal models) and R4 (enable a formal verification) are fulfilled by this thesis and the approaches of Gil de la Iglesia as well as Ramirez et al.

This thesis and Ramirez et al. enable a formal verification of their patterns and, therefore, fulfill requirement R5. AGENTUML and Gil de la Iglesia only fulfill this requirement partially: AGENTUML only provides informal properties that shall hold and Gil de la Iglesia only provides property patterns that are independent of his templates.

Concerning requirement R6 (provide concepts to select and adapt a pattern), this thesis and Gamma et al. provide a process for both selecting and adapting a pattern. We present our concepts in Section 6.5. Buschmann et al. and Douglass fulfill this requirement only partially as they do not provide concepts to adapt a pattern to an application-specific solution. In contrast to Gamma et al., Buschmann et al., and Douglas, we do not only provide a process but also provide search terms and an ontology to further facilitate the section process.

No approach (even ours) completely fulfills requirement R7 (provide concepts to identify and abstract a pattern). In particular, this thesis only focuses on abstracting a protocol correctly to a pattern and not on identifying/mining/harvesting the pattern. We present our concepts in Section 6.4. Ramirez et al. only partially fulfill this requirement: they do not provide a process for identifying patterns or abstracting application-specific solutions to patterns but only document the process they have applied to develop their patterns.

CONCLUSION

This chapter presents the conclusions that we draw from this thesis. In particular, we summarize our work in Section 7.1 and point out future research challenges in Section 7.2, which can be derived from this thesis' results.

7.1 Summary

As stated by the German National Academy of Science and Engineering (acatech), “[f]uture cyber-physical systems will contribute to safety, efficiency, comfort and human health like never before.” [aca11, p. 5] These systems may achieve this by coordinating their behavior via an asynchronous message exchange, forming a system of systems. However, enabling such a coordination increases the complexity of designing their software: it introduces more sources of safety-critical faults at runtime because a single system now also depends on the correct (timing) behavior of the other systems as well as on the (timing) behavior of the communication network. The contributions of this thesis enable software engineers of CPSs to manage the complexity when specifying platform-independent models of such systems and when formally verifying their safety. Within this thesis, our contributions focus on the MECHATRONICUML method [*DPP+16]. Though, they may also be transferred to other model-driven approaches that provide support for specifying and verifying software models of CPSs. We successfully implemented all our contributions into the MECHATRONICUML Tool Suite [*DGB+14].

As our first contribution, we define a DSL for specifying verifiable coordination protocols of CPSs to reduce the potential of modeling errors and to avoid ambiguous and, therefore, unsafe specifications. In particular, we improve the initial definition of RTCPs by Giese et al. [GTB+03; BGHS04] and Hirsch [Hir08] to decrease errors during the specification. First of all, we provide incoming message buffers as explicit model elements. Therefore, a software engineer does not need to specify them using RTSCs anymore but can define them in a more compact manner. Moreover, we enable a more compact behavioral definition for one-to-many RTCPs by providing new keywords for specifying coordination aspects. As a consequence, a software engineer can specify the same behavior with significantly less model elements than before. We refine the existing modeling assumptions such that no ambiguities concerning the model checking of RTCPs exist anymore. In particular, we define the assumed software layers, the assumed steps of a message transmission, and the QoS assumptions. In a case study, we show that our new behavioral definition for one-to-many RTCPs is sufficient to specify existing examples and that they enable to model the behavior in a more compact manner.

Finally, we summarize that this thesis fulfills all our requirements but no other approach or a combination of approaches would fulfill all requirements.

As our second contribution, we define a DSVPL for specifying the verification properties for verifiable coordination protocol of CPSs at the domain level while respecting the capabilities of the selected model checker. Having such a DSVPL shall reduce the necessary experience to specify and understand the properties to be verified. This avoids wrong properties that lead to false conclusions of the system's safety. In our case, the DSVPL is called MTCTL, the protocol formalism are RTCPs, and the model checker is UPPAAL. Among others, MTCTL enables to make statements about (real-time) aspects that are no longer defined as RTSCs but using explicit model elements, e.g., message buffers. Moreover, we define a set of application-independent verification properties (called default properties) that should always hold for all RTCPs as they check important safety, reachability, and design aspects. In addition, we provide a translation from MTCTL to English sentences. This shall enable all involved stakeholders that do not necessarily have model checking expertise, e.g., requirements engineers, to understand the stated MTCTL properties and link them to other artifacts of the system. In a case study, we show that we are able to state a vast majority of realistic informal requirements in MTCTL and that we can generate valid English sentences for each MTCTL property. Finally, we summarize that this thesis fulfills all our requirements but no other approach or a combination of approaches would fulfill all requirements.

As our third contribution, we define a model-driven DSMC approach for verifiable coordination protocols of CPSs in order to hide the details of the model checking. Therefore, we reduce the necessary experience to apply model checking, which may lead to an improved applicability of model checking and, therefore, to an improved safety of the system. In particular, we define a DSMC for MECHATRONICUML's RTCPs using the model checker UPPAAL. The inputs of the DSMC are the RTCP and MTCTL properties. The output is the verification result per property and—if available—a trace (e.g., a counterexample) at the level of MECHATRONICUML. Our DSMC is fully automatic and completely hides the model checking from the software engineer. Moreover, our DSMC concept is especially appropriate for large differences between the DSL and the model checker's language as it is the case for RTCPs and UPPAAL because the concept enables to specify a multi-step translation to the model checker but only requires a single step for the back-translation to the DSL. In a case study, we show that our DSMC correctly translates 16 RTCPs including their MTCTL properties to UPPAAL, that it correctly translates results and (counterexample) traces produced by UPPAAL back to MECHATRONICUML, that one minute is in most cases sufficient to execute the complete DSMC, and that our DSL's encapsulate domain-specific aspects more effectively than UPPAAL, which results in an efficient specification of coordination protocols and verification properties. Finally, we summarize that this thesis fulfills nearly all our requirements but no other approach or a combination of approaches would fulfill all requirements.

As our fourth contribution, we define formal design patterns that consider the need to formally verify coordination protocols of CPSs. By doing this, we enable a more efficient design of such protocols. In particular, we introduce the concept of Real-Time Coordination Patterns for RTCPs. These patterns capture existing modeling experience, facilitate the correct specification (with a focus on formal verification), and ensure a good design of RTCPs. Currently, we identified nine patterns so far and describe them with a uniform description format. Moreover, we define a development process for selecting and adapting Real-Time

Coordination Patterns to RTCPs as well as for abstracting RTCPs to Real-Time Coordination Patterns. In a case study, we show that our patterns have a high reusability rate within two different applications and that the RTCPs that we adapted from a pattern are still similar to the pattern and do not change the pattern's intent. Finally, we summarize that this thesis fulfills nearly all our requirements but no other approach or a combination of approaches would fulfill all requirements.

7.2 Future Work

The contributions of this thesis provide several aspects that may be the topic of future research. In general, future work can remove the limitations that we state in the sections 3.9, 4.8, 5.9, and 6.8. Furthermore, although we already did four case studies (see the section 3.8, 4.7, 5.8, and 6.7), some of our contributions are not evaluated yet. In addition, all our contributions require deep evaluations using real-world examples from different domains (automotive, avionics, factory automation, etc.). In the following, we specify ten aspects for future research:

Buffer Specification and QoS of RTCPs Further evaluations (e.g., user acceptance tests [Cim06]) can answer whether the expressiveness of our definition of RTCPs is appropriate for the domain of CPSs. For example, concerning message buffers, AUTOSAR [AUTOSAR] enables that the owner of a buffer is informed if a message is discarded and MARTE [MARTE] enables additional sorting strategies for the messages in the buffer besides FIFO (e.g., Last-In First Out). Currently, we do not provide these features as we did not find use cases for them but this might change in the future. In addition, further evaluations can show whether we currently miss any QoS assumptions (implicit and explicit) concerning the asynchronous message transfer or whether some assumptions are not necessary or too strict.

Expressiveness of MTCTL Our evaluation in Section 4.7 shows that we miss important language elements of MTCTL, e.g., dynamic predicates concerning sending and receiving a message. Therefore, future work should enrich our language. Further evaluations, e.g., user acceptance tests, can show whether we need more changes concerning the expressiveness of our new language. For example, we currently do not support all language elements of UTCTL as we did not find relevant use cases yet, e.g., UTCTL provides bit-shift operators [BDL06] but MTCTL does not.

Formal Proof of the Correctness of our DSMC In this thesis, we show that our DSMC is correct for a wide range of RTCPs via a case study in Section 5.8. Nevertheless, we do not formally prove that our DSMC always preserves the semantics of RTCPs. A formal proof could raise the trust in the verification results. This is especially relevant as CPSs are typically safety-critical. Future work could realize such a formal proof by reusing concepts of Hegedüs et al. [HBRV10]. However, this approach requires a formal semantics definition of the DSL and the model checker. Formal semantics of RTSCs are already defined by Heinzemann [Hei15] but a formal definition of the remaining parts of an RTCP (buffer behavior, message exchange, ...) is not defined yet. Moreover, a definition for all model elements of UPPAAL timed automata that we use is not publicly available as well.

Applicability of the DSMC for One-To-Many RTCPs Our evaluation in Section 5.8 shows that our DSMC may require too much main memory when verifying a one-to-many RTCP in UPPAAL—supposedly due to the state-explosion-problem [CGP00] as a one-to-many RTCP with a role multiplicity higher than 1 increases the parallelism in the model. For example, we are not able to prove significant verification properties like deadlock freedom for our one-to-many RTCP *MultiOvertaking* with role multiplicity 2 on a machine that has 100 GB of main memory. In this thesis, we focused on a correct translation from RTCPs to UPPAAL rather than a translation that requires as less main memory as possible. Therefore, future work has to identify whether our translation from RTCPs to UPPAAL may be improved such that we still preserve the semantics of RTCPs but require less main memory. Moreover, as described in Section 2.1.3, UPPAAL provides several options to influence the model checking process. Currently, we only use the default options and did not try to optimize them. Altering these options in general or specifically for each RTCP or each MTCTL property may improve the current situation as well. In addition, future work might inspect whether other types of performance improvements might be possible with respect to abstractions, e.g., by applying symbolic model checking [BK08, p. 381]. For example, the approach of Morb   et al. [MPS11] enables fully symbolic model checking for timed automata that outperforms model checkers like UPPAAL.

Feature Models for Real-Time Coordination Patterns We already identified several variants for each of our nine Real-Time Coordination Patterns. Though, a pattern specifier currently has to specify one model for each variant of a Real-Time Coordination Pattern (including its MTCTL properties, verified configurations, and informal descriptions). This is a time-consuming task and leads to redundancy problems as a variant often only changes a small part of the original pattern. Therefore, future work may provide a formal variant management of Real-Time Coordination Patterns by using feature models [CE00]. Then, a pattern user could select for one pattern a set of features that he needs in order to fulfill its requirements. Then, an appropriate model (including its MTCTL properties, verified configurations, and informal descriptions) could be generated based on the pattern user’s selection.

Improve Pattern Filtering Our concept for filtering the list of Real-Time Coordination Patterns within a catalog describes the possibility to search a term within the textual descriptions and to search within an ontology based on a given verb-noun-pair. A third option that we currently do not support is to use the resulting models of the requirements phase as inputs for the filtering. For example, *MECHATRONICUML* defines for the requirements phase formal message sequence diagrams called Modal Sequence Diagram (MSD) [HFK+16; Gre11] that specify the message-exchange between two or more roles including real-time constraints. Furthermore, *MECHATRONICUML* defines for the platform-independent design phase that one or more RTCPs have to be defined for the given set of MSDs per message connector. As we now provide Real-Time Coordination Patterns for RTCPs, the project group *Safebots III* [ABB+13] proposes that a software engineer should select one or more pattern for each MSD first and adapt the pattern(s) to RTCPs afterwards. In particular, they propose an automatic selection process that takes an MSD as an input and returns a list of Real-Time Coordination Patterns that

fulfill the sequence specified by the MSD. Though, their concept is an early draft and needs additional conceptual definitions.

Usability Within this thesis, we provide several aspects that require user interaction. Here, usability is an important software quality factor that—among others—influences the understandability and the efficiency of the software engineer. However, our concepts do not consider usability factors. Therefore, intensive usability tests [DR99] are required. For example, we decided that MTCTL is a TCTL-like language. Though, an OCL-based language like [FM02; CK02] or a language where the software engineer defines structured English sentences like [FMR00; KC05] might be more useable solutions. As another example, we provide concrete (graphical) syntax definitions for RTCPs and Real-Time Coordination Patterns but consider them as a first draft that needs to be evaluated.

Support for Reconfiguration A CPS is typically self-adaptive (i.e., it may adapt its behavior by reconfiguring its software architecture) as the environment and the set of participants of a CPS change at runtime. Hirsch [Hir08] already provides concepts for reconfiguring the number of roles instances of an RTCP at runtime. In this thesis, we do not focus on reconfiguration. Consequently, future work may enrich our contributions with this aspect. In particular, the following actions could be done: (1) identify whether use cases exist to reconfigure the number of incoming message buffers per role instance at runtime (if this is the case, then define their syntax and semantics), (2) define the QoS assumptions during a reconfiguration of the number of role instances, (3) evaluate whether the explicit QoS assumptions may be reconfigured by the software engineer (e.g., if the message transfer in the underlying platform switches from LAN to WLAN, then the maximum message transmission delay might be adapted), (4) increase the expressiveness of MTCTL (e.g., a software engineer should be able to state that a certain configuration fulfills certain characteristics), (5) adapt the DSMC such that it considers the changes of the actions 1-4, and (6) add for each one-to-many Real-Time Coordination Pattern useful variants that enable reconfiguration.

Testing Concepts After specifying an RTCP, the software engineer has to verify its design using model checking, which can automatically analyze all possible execution traces of the RTCP. In subsequent development stages, the software engineer has to ensure that the definitions of the RTCP are still respected (which includes that all MTCTL properties are still fulfilled). Otherwise, the model checking results are obsolete and may ultimately result in safety-critical errors in the final system. However, model checking may not be applied in all subsequent stages. For example, the process of MECHATRONICUML defines that a domain-spanning system model, where discrete and continuous behavior (feedback controller and environment models) are combined, shall be defined later on. Due to the continuous behavior, model checking may not be used and hybrid model checking [Hen96] is not applicable for realistic models [ERNF12]. Therefore, traditional testing (e.g., simulations) must be applied to analyze the models. As testing only covers some execution traces, the selection of good test cases is very important. Thus, future work could come up with an automatic test case generation based on the specified RTCPs and its MTCTL properties. For example, one valid test case could be that no message sequences except the ones that the RTCP allows are

possible. Furthermore, future work should define a test method that checks whether all stated (implicit and explicit) QoS assumptions that we specified in Section 3.4 are fulfilled.

Consider Security Currently, our concepts only address the safety of the system and not its security. However, security threats exist and need to be addressed. As an example, the asynchronous message communication between roles of an RTCP may be the target for a security attack (especially if the communication is wireless), e.g., a man-in-the-middle attack [Kiz05] may threaten the confidentiality of the communication (data is sent to the wrong receiver), the integrity (the accuracy and the completeness) of the data being sent, and the availability of the receiver (e.g., the man in the middle could start a denial-of-service attack). All these threats may comprise the safety of the systems. Therefore, as Heinzemann [Hei15] already suggests, “[...] future works first need to integrate an authentication mechanism [DVW92; BCK98] into the instantiation of RTCPs on system level to ensure that no unauthorized system [may communicate with the CPS]. Second, future works need to integrate the use of encryption standards like the Advanced Encryption Standard (AES, [AES; DR02]) into RTCPs to enable secure communication. Such security measures may probably be generated into the system automatically when deriving the platform-specific model.” [Hei15]

CATALOG OF ONE-TO-MANY COMMUNICATION SCHEMATA

In Section 3.5.2, we briefly introduce our six one-to-many communication schemata that we currently support. Moreover, we give a detailed definition of the one-to-many communication schemata multicast and multireceive in Section 3.5.2.2 and Section 3.5.2.3. Furthermore, for both schemata, we already define one of the two normalization templates (Section 3.5.2.8 and 3.5.2.8).

In this appendix, we first describe the second normalization template of the schema multicast in Section A.1. Then, we describe the second normalization template of the schema multireceive in Section A.2. Afterwards, we describe the remaining four schemata (their definition as well as their normalization templates). In particular, we describe unicast in Section A.3, singlereceive in Section A.4, iterate in Section A.5, and loadbalancing in Section A.6.

Note that all normalizations are formally defined within our implementation using QVTo (see Section 3.7.2).

A.1 Normalization Template for Multicast with Subrole Condition

We show the normalization template for the one-to-many communication schema multicast with subrole-specific conditions in Figure A.1. Our template covers not only the case that the transition specifies the message that shall be sent via a multicast, but also the conditions for enabling the transition and the effects that shall be executed within the coordinator after the multicast is executed. Furthermore, the template covers the subrole-specific attributes condition, retryAfter, and action.

The template defines that we add in Step 3.1 (see Figure 3.12) four synchronization channels `multicastCheck<i>`, `multicastCheckDone<i>`, `multicast<i>`, and `multicastDone<i>`, where `<i>` is a unique number of all multicasts that are used within this multi role. Moreover, the channels `multicastCheck` and `multicast` are typed over the multi role.

In Step 3.2., we adapt the coordinator as follows:

1. The source state (here: A) receives an additional region that checks whether the multicast is possible. In particular, it only starts the check if the subrole-independent conditions of the original transition are fulfilled. Then, the region instructs each subrole instance

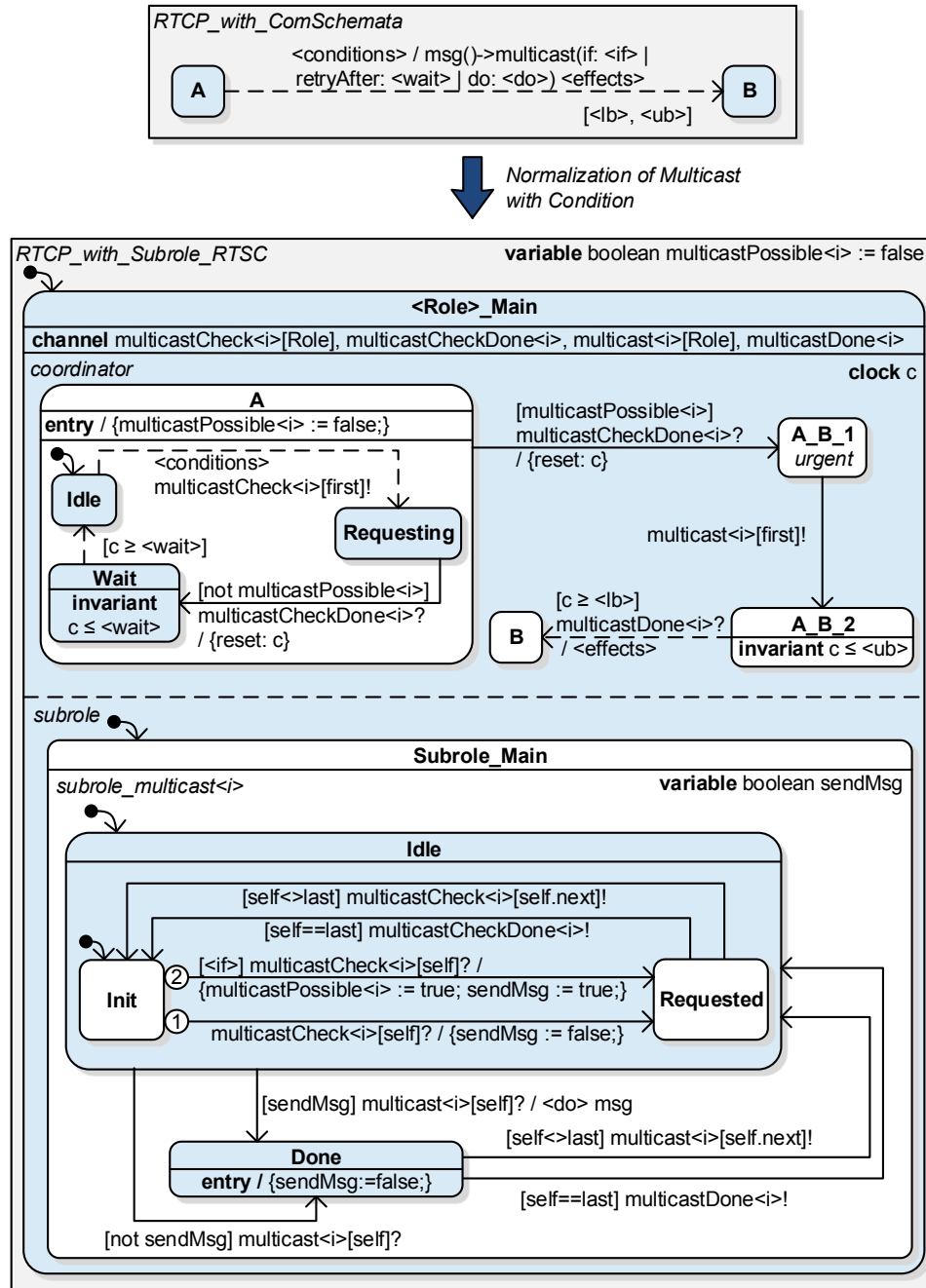


Figure A.1: Normalization Template for Multicast with Conditions

to check whether the multicast is possible (i.e., if at least one subrole fulfills its subrole-specific condition). If it is possible, then each subrole that fulfilled its subrole-specific condition shall send its message. If it is not possible, then the region waits the user-defined time before it checks the condition and all subrole-specific conditions once again.

2. The transition from *Idle* to *Requesting* adopts the urgency of the original transition, e.g., if the original transition is urgent, then this transition is urgent as well.
3. We replace the original transition by two intermediate states and three transitions. The first transition is fired if at least one subrole instance can send the multicast. The second transition initiates the sending of the multicast. The third transition is fired as soon as the sending is done and executes the subrole-independent effects of the original transition.
4. If the original transition defines a deadline, then we execute the same five changes as specified in the template of the schema multicast that does not use subrole-specific conditions (see Section 3.5.2.8).

In Step 3.3, we add a region with the name `subrole_multicast<i>` to the state `Subrole_Main`, where `<i>` is a unique number of all multicasts that are used within this multi role.

Then, in Step 3.4, we add the states *Idle* and *Done* and four transitions. The state *Idle* is a composite state that contains the states *Init* and *Requested* and four transitions as well. *Idle* handles the check whether the multicast is possible. If this is the case, this information is stored in two variables: the subrole-specific variable `sendMsg` and the multi role variable `multicastPossible<i>`, where `<i>` is a unique number of all multicasts that are used within this multi role.

Note that the transition priorities that we define explicitly in this template are important as the behavior would otherwise differ from the definition.

A.2 Normalization Template for Multireceive with Subrole Condition

We show the normalization template for the one-to-many communication schema multireceive with subrole-specific conditions in Figure A.2. Our template covers not only the case that the transition specifies the message that shall be received via a multireceive, but also the conditions for enabling the transition and the effects that shall be executed within the coordinator after the multireceive is executed. Furthermore, the template covers the subrole-specific attributes condition, `retryAfter`, `action`, and `failureAction`.

The template defines that we add in Step 3.1 (see Figure 3.12) four synchronization channels `multiRcvCheck<i>`, `multiRcvCheckDone<i>`, `multiRcv<i>`, and `multiRcvDone<i>`, where `<i>` is a unique number of all multireceives that are used within this multi role. Moreover, the channels `multiRcvCheck` and `multiRcv` are typed over the multi role.

In Step 3.2., we adapt the coordinator as follows:

1. The source state (here: *A*) receives an additional region that checks whether the multireceive is possible. In particular, it only starts the check if the subrole-independent conditions of the original transition are fulfilled. Then, the region instructs each subrole instance to check whether the multireceive is possible (i.e., if at least one subrole

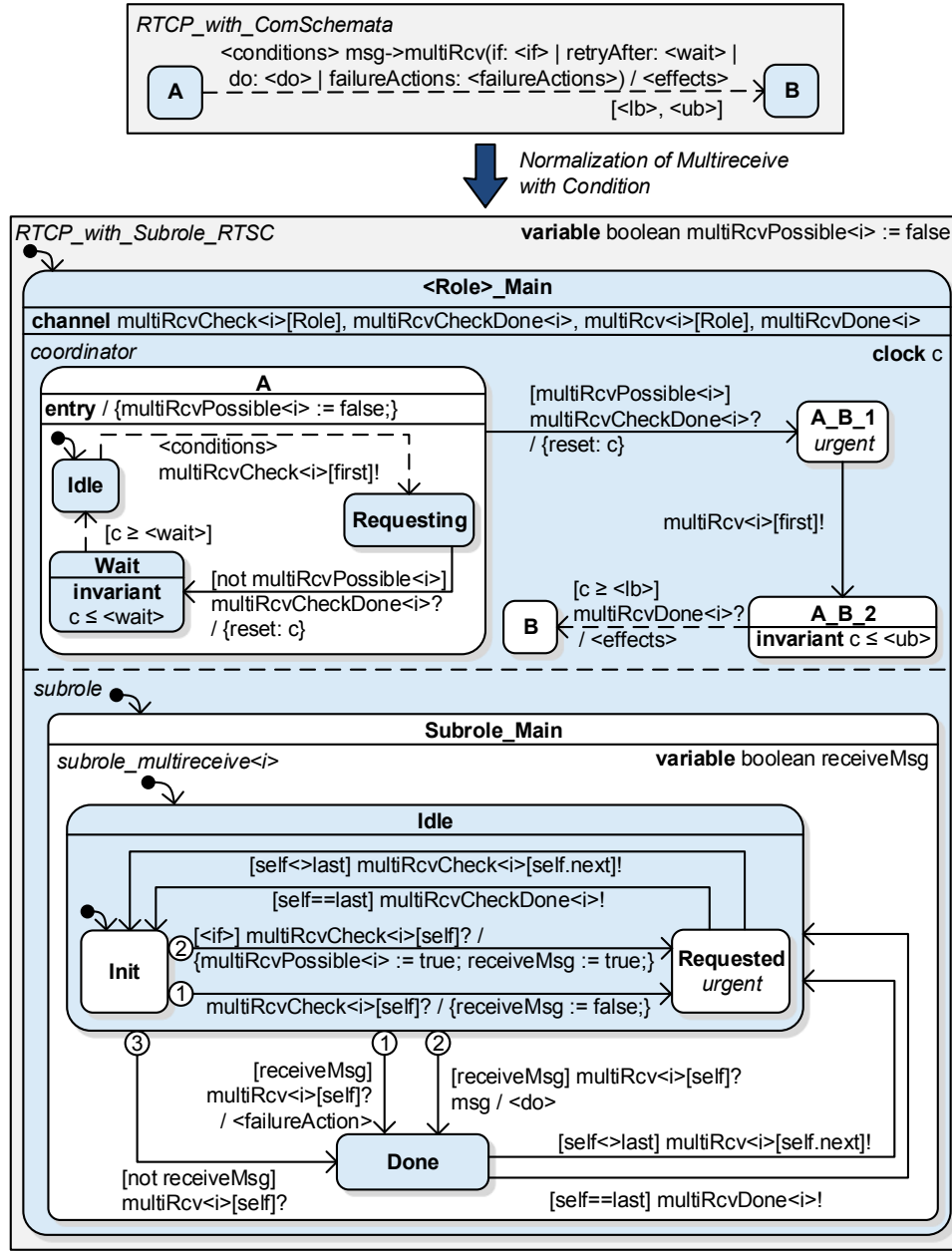


Figure A.2: Normalization Template for Multireceive with Conditions

fulfills its subrole-specific condition). If it is possible, then each subrole that fulfilled its subrole-specific condition shall try to receive its message. If it is not possible, then the region waits the user-defined time before it checks the condition and all subrole-specific conditions once again.

2. The transition from *Idle* to *Requesting* adopts the urgency of the original transition, e.g., if the original transition is urgent, then this transition is urgent as well.
3. We replace the original transition by two intermediate states and three transitions. The first transition is fired if at least one subrole instance can receive the *multireceive*. The second transition initiates the receiving of the *multireceive*. The third transition is fired as soon as the receiving is done and executes the subrole-independent effects of the original transition.
4. If the original transition defines a deadline, then we execute the same five changes as specified in the template of the schema *multireceive* that does not use subrole-specific conditions (see Section 3.5.2.8).

In Step 3.3, we add a region with the name `subrole_multireceive<i>` to the state `Subrole_Main`, where `<i>` is a unique number of all *multireceives* that are used within this multi role.

Then, in Step 3.4, we add the states *Idle* and *Done* and five transitions. We distinguish three cases: (1) the subrole shall not try to receive a message as its subrole-specific condition was not fulfilled, (2) the subrole shall try to receive its message as its subrole-specific condition was fulfilled and consumes its from the buffer, and (3) the subrole shall try to receive its message as its subrole-specific condition was fulfilled but no message is available.

The state *Idle* is a composite state that contains the states *Init* and *Requested* and four transitions. *Idle* handles the check whether the *multireceive* is possible. If this is the case, this information is stored in two variables: the subrole-specific variable `receiveMsg` and the multi role variable `multiRcvPossible<i>`, where `<i>` is a unique number of all *multireceives* that are used within this multi role.

Note that the transition priorities that we define explicitly in this template are important as the behavior would otherwise differ from the definition.

A.3 Unicast

We define the one-to-many communication schema *unicast* in Section A.3.1. Afterwards, we define its normalization template in Section A.3.2.

A.3.1 Definition

Description The one-to-many communication schema *unicast* enables the multi role instance to send a message to exactly one instance of the single roles. The RTCP specifier must define a condition that the subrole instance has to fulfill before it may send the message. Otherwise, either the same single role instance would always get the message or a random one. Both possibilities are not desired. The RTCP specifier may specify the condition via the attribute condition that contains textual expressions of MECHATRONICUML's action language (see Section 2.2.2). As an important fact, the transition that specifies a *unicast* with a condition may only fire if at least one subrole instance fulfills

the stated condition. Otherwise, the transition is not enabled. Furthermore, the subrole instances are always evaluated in ascending order. The first subrole instance that fulfills the condition will send the message. If no subrole instance fulfills the condition, then the RTSC has to wait a fixed time before it may evaluate the subrole instances again. The RTCP specifier has to define this time span (as a time value expression) using the attribute `retryAfter`. It has to be greater than zero.

In addition, this schema enables to define an action that shall be executed by the subrole instance after sending the message. The RTCP specifier may define this via the attribute `action` that contains textual expressions of MECHATRONICUML's action language as well.

In summary, the attributes `condition` and `retryAfter` are mandatory while the attribute `action` is optional.

Example The coordinator of a platoon of cars that does not drive at the front of the platoon uses an RTCP to coordinate the platoon. Therefore, the coordinator may send messages to all other cars of the platoon. The one-to-many communication schema `unicast` is appropriate if the coordinator wants to send a message concerning a new platoon speed (via this RTCP) to the frontmost car only. We depict this example using a sequence diagram in Figure A.3.

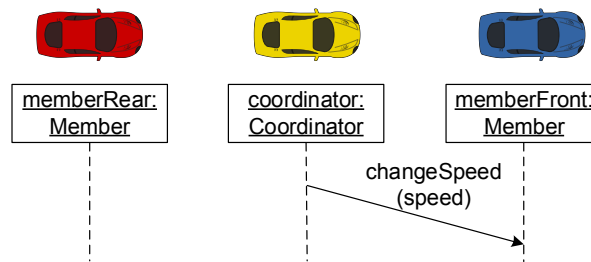


Figure A.3: Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Unicast

Concrete Syntax If the one-to-many communication schema `unicast` is applied, then we extend the transition label as follows. At first, we append the string `'->unicast'` to the name of the message. Then, if one of the three attributes is defined, we extend the string by round parentheses. The attribute `condition` is given first with the prefix `'if: '`, the attribute `retryAfter` is given next with the prefix `"retryAfter: "`, and the action attribute `action` is given last with the prefix `'do: '`. If more than one attribute is used, we separate them with a pipe. As an example, we write `m1->unicast(if: <conditionExpr>|retryAfter: <retryAfterExpr>|do: <actionExpr>)` for a message `m1` that shall be sent via a unicast and that has the condition `<conditionExpr>`, the retry after delay `<retryAfterExpr>`, and the action `<actionExpr>`.

Alternative Schemata If the message may be sometimes relevant for more than one instance of the single role, then the RTCP specifier should either use `multicast` or `iterate` and define a condition to filter the set of receivers. If it is necessary to send messages (e.g., a task that the receiver shall execute) to multiple instances of the single role, then

the RTCP specifier can use the one-to-many communication schema loadbalancing (including a condition that must hold) instead of multiple unicasts to avoid that the same receiver gets all messages.

Combinability A typical combination is that the multi role sends a unicast first and awaits an answer by one instance of the single role via a singlereceive.

A.3.2 Normalization Template

We show the normalization template for the one-to-many communication schema unicast in Figure A.4. Our template covers not only the case that the transition specifies the message that shall be sent via a unicast, but also the conditions for enabling the transition and the effects that shall be executed within the coordinator after the unicast is executed. Furthermore, the template covers the subrole-specific attributes condition, `retryAfter`, and action.

The template defines that we add in Step 3.1 (see Figure 3.12) four synchronization channels `unicastCheck<i>`, `unicastCheckDone<i>`, `unicast<i>`, and `unicastDone<i>`, where `<i>` is a unique number of all unicasts that are used within this multi role. The channel `unicastCheck` is typed over the multi role.

In Step 3.2., we adapt the coordinator as follows:

1. The source state (here: A) receives an additional region that checks whether the unicast is possible. In particular, it only starts the check if the conditions of the original transition are fulfilled. Then, the region instructs each subrole instance to check whether the unicast is possible (i.e., if at least one subrole fulfills its subrole-specific condition). Hereby, the coordinator region iterates all subrole instances in ascending order. As soon as a subrole instance fulfills its condition, the iteration is stopped. Afterwards, the coordinator instructs this subrole instance to send the message. If no subrole instance fulfills its condition, then the region waits the user-defined time before it checks once again.
2. The transition from `Idle` to `Requesting` adopts the urgency of the original transition, e.g., if the original transition is urgent, then this transition is urgent as well.
3. We replace the original transition by two intermediate states and three transitions. The first transition is fired if one subrole instance can send the unicast. The second transition initiates the sending of the unicast. The third transition is fired as soon as the sending is done and executes the subrole-independent effects of the original transition.
4. If the original transition defines a deadline, then we execute the same five changes as specified in the template of the schema multicast that does not use subrole-specific conditions (see Section 3.5.2.8).

In Step 3.3, we add a region with the name `subrole_unicast<i>` to the state `Subrole_Main`, where `<i>` is a unique number of all unicasts that are used within this multi role.

Then, in Step 3.4, we add the states `Idle` and `Sent` and two transitions. The state `Idle` is a composite state that contains the states `Init` and `Requested` and four transitions. `Idle` handles the check whether the unicast is possible. If this is the case, this information is stored in two variables: the subrole-specific variable `sendMsg` and the multi role variable `unicastPossible<i>`, where `<i>` is a unique number of all unicasts that are used within this multi role. The state `Sent` is only visited by the subrole instance that sends the unicast.

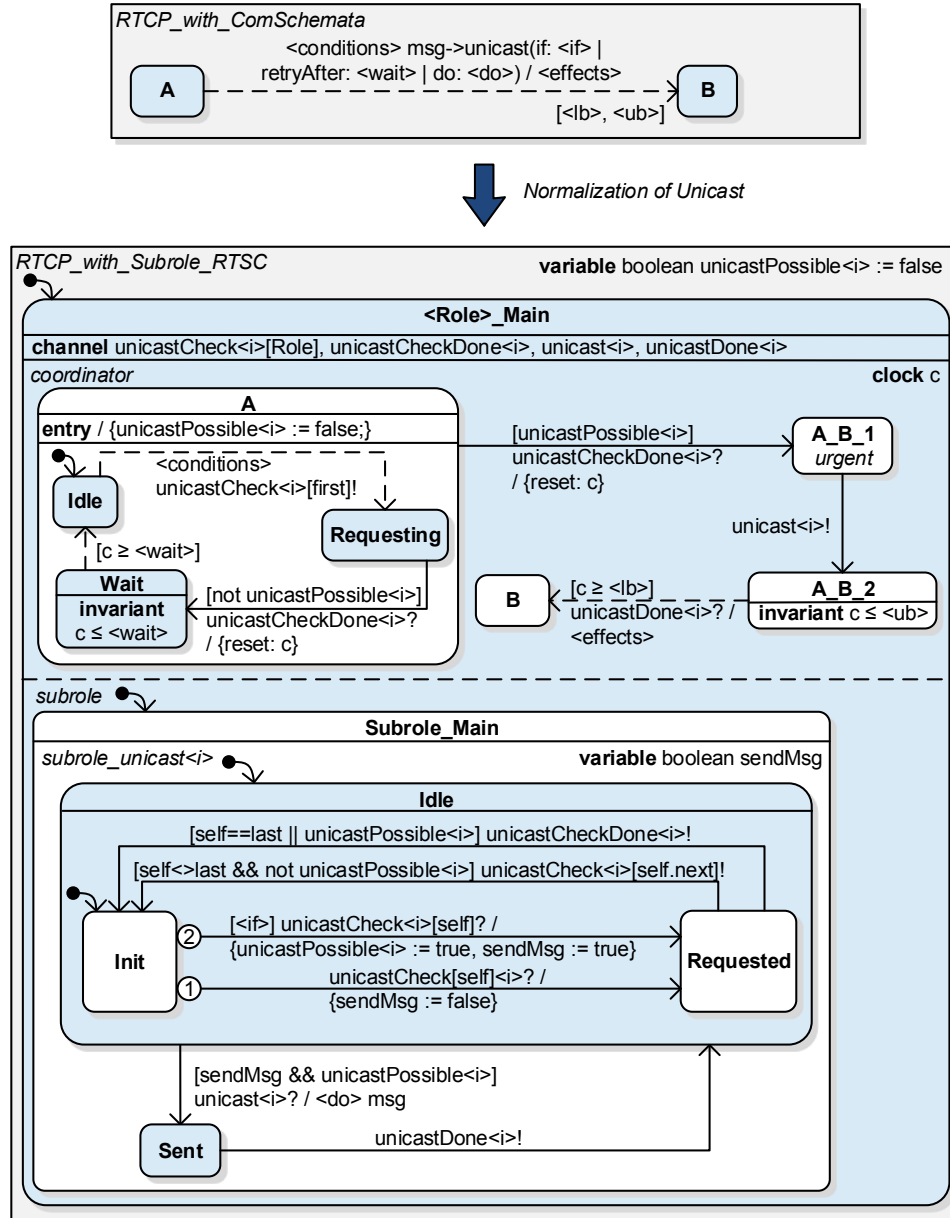


Figure A.4: Normalization Template for Unicast

Note that the transition priorities that we define explicitly in this template are important as the behavior would otherwise differ from the definition.

A.4 Singlereceive

We define the one-to-many communication schema *singlereceive* in Section A.4.1. Afterwards, we define its normalization template in Section A.4.2.

A.4.1 Definition

Description The one-to-many communication schema *singlereceive* enables the multi role instance to receive a message from exactly one instance of the single roles. The RTCP specifier must define a subrole-specific condition that a subrole instance has to fulfill before it may receive the message. He may specify this via the attribute condition that contains textual expressions of MECHATRONICUML's action language (see Section 2.2.2). As an important fact, the transition that specifies a *singlereceive* with a condition may only fire if at least one subrole instance fulfills the stated condition. Otherwise, the transition is not enabled. Furthermore, the subrole instances are always evaluated in ascending order. The first subrole instance that fulfills the condition and has a message in its buffer that may be received shall receive the message. If no subrole instance was able to receive a message, then the RTSC has to wait a fixed time before it may evaluate the subrole instances again. The RTCP specifier has to define this time span (as a time value expression) using the attribute *retryAfter*. It has to be greater than zero. In addition, this schema enables to define an action that shall be executed by the subrole instance after receiving the message. An RTCP specifier may define this via the attribute *action* that may contain textual expressions of MECHATRONICUML's action language as well.

In summary, the attribute *retryAfter* is mandatory while the attributes *condition* and *action* are optional.

Example The coordinator of a platoon of cars that does not drive at the front of the platoon wants to find out which car is driving at the front. Therefore, it sends a multicast first to all members of the platoon. Then, the coordinator awaits a single answer by the one member that drives at the front via a *singlereceive*. We depict this example using a sequence diagram in Figure A.5.

Concrete Syntax If the one-to-many communication schema *singlereceive* is applied, then we extend the transition label as follows. At first, we append the string `'->singlereceive'` to the name of the message. Then, if one of the three attributes is defined, we extend the string by round parentheses. The attribute *condition* is given first with the prefix `'if: '`, the attribute *retryAfter* is given next with the prefix `'retryAfter: '`, and the attribute *action* is given last with the prefix `'do: '`. If more than one attribute is used, we separate them with a pipe. As an example, we write `m1->singlereceive(if: <conditionExpr>|retryAfter: <retryAfterExpr>|do: <actionExpr>)` for a message *m1* that shall be sent via a *singlereceive* and that has the condition `<conditionExpr>`, the retry after delay `<retryAfterExpr>`, and the action `<actionExpr>`.

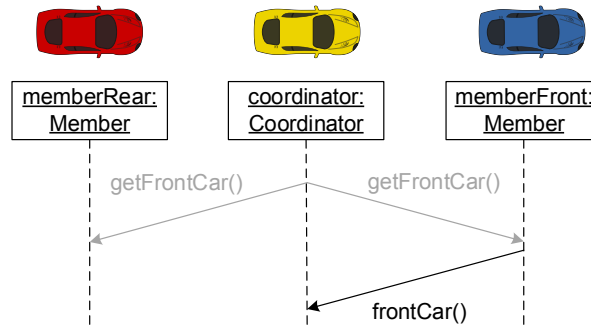


Figure A.5: Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Singlereceive

Alternative Schemata If it may be the case that more than one instance of the single role answers, then the RTCP specifier should use the schema *multireceive*.

Combinability A typical combination is that the multi role sends a multicast or unicast first and awaits an answer by one instance of the single role via a *singlereceive*.

A.4.2 Normalization Template

We show the normalization template for the one-to-many communication schema *singlereceive* in Figure A.6. Our template covers not only the case that the transition specifies the message that shall be received and consumed via *singlereceive*, but also the conditions for enabling the transition and the effects that shall be executed within the coordinator after the *singlereceive* is executed. Furthermore, the template covers the subrole-specific attributes condition, *retryAfter*, and action.

The template defines that we add in Step 3.1 (see Figure 3.12) three synchronization channels *singleRcv*<i>, *singleRcvNotPossible*<i>, and *singleRcvDone*<i>, where <i> is a unique number of all *singlereceive*s that are used within this multi role. Moreover, the channel *singleRcv* is typed over the multi role.

In Step 3.2., we adapt the coordinator as follows:

1. The source state (here: A) receives an additional region that checks whether the *singlereceive* is possible and—if this is the case—receives the message. In particular, it first checks whether the conditions of the original transition are fulfilled. Then, the region instructs each subrole instance to check whether the *singlereceive* is possible (i.e., if at least one subrole fulfills its subrole-specific condition and may receive the message). Hereby, the coordinator region iterates all subrole instances in ascending order. The first subrole instance that fulfills the condition and has a message in its buffer that may be received shall directly receive the message. If no subrole instance was able to receive a message, then the region waits the user-defined time before it checks once again.
2. The transition from Idle to Requesting in A adopts the urgency of the original transition, e.g., if the original transition is urgent, then this transition is urgent as well.
3. We replace the original transition by one intermediate state and two transitions. The first transition is fired if one subrole instance executes the receiving and consuming of

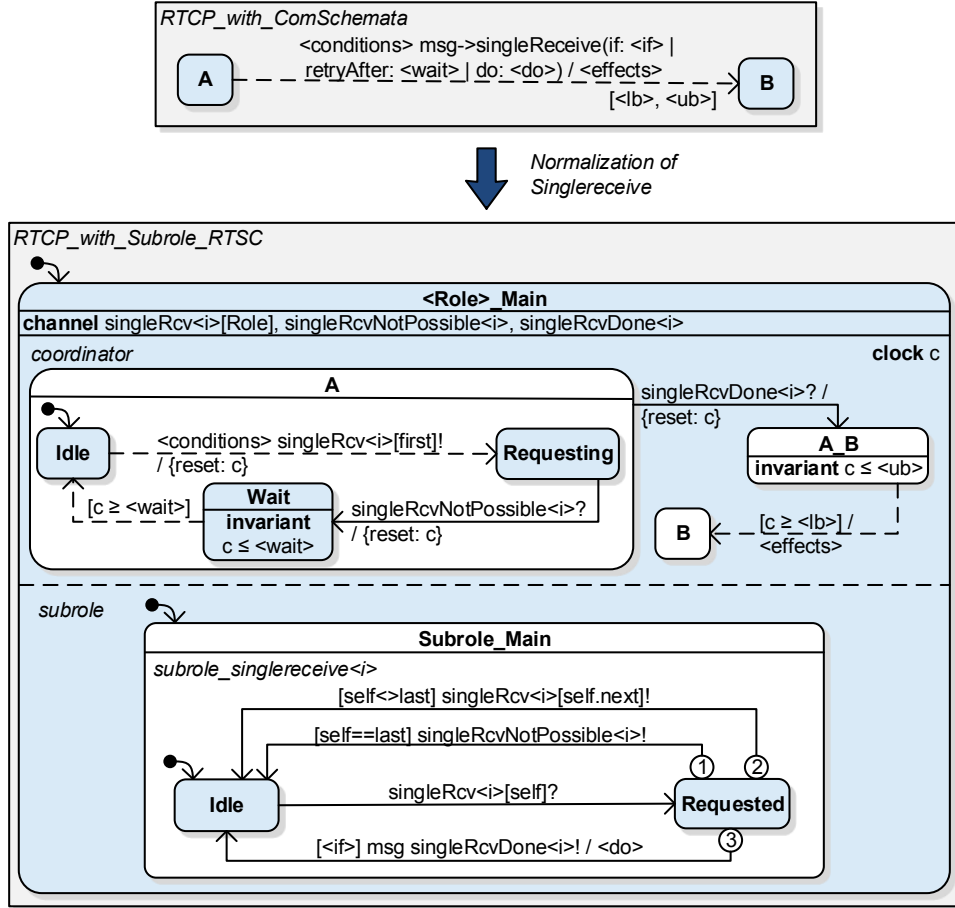


Figure A.6: Normalization Template for Singlereceive

the message. The second transition only executes the subrole-independent effects of the original transition.

4. If the original transition defines a deadline, then we execute the following four changes:
 - (1) we reuse the clock c that exists for the attribute `retryAfter` and reset it in the transition from `Idle` to `Requesting`,
 - (2) we define an invariant in the intermediate state (here: `A_B`),
 - (3) we add a clock constraint in the transition to the target state of the original transition (here: `B`), and
 - (4) define that this transition is non-urgent as MECHATRONICUML does not allow to define a clock constraint at an urgent transition. If the original transition does not define a deadline, then we do not need to apply these changes.

In Step 3.3, we add a region with the name `subrole_singlereceive<i>` to the state `Subrole_Main`, where $<i>$ is a unique number of all `singlereceives` that are used within this multi role.

Then, in Step 3.4, we add the states `Idle` and `Requested` and four transitions to the new region. These states and transitions handle the check whether the `singlereceive` is possible. If this is the case, then the message is directly consumed while the coordinator gets informed about this via the synchronization channel `singleRcvDone`.

Note that the transition priorities that we define explicitly in this template are important as the behavior would otherwise differ from the definition.

A.5 Iterate

We define the one-to-many communication schema *iterate* in Section A.5.1. Afterwards, we define its normalization template in Section A.5.2.

A.5.1 Definition

Description The one-to-many communication schema *iterate* enables the multi role instance to send the same message to (a subset of) all instances of the single role. The parameters of each message may vary. Furthermore, the messages are not sent at the same time but a certain delay must happen until the next message may be sent, i.e., the clocks must progress after sending one message. By default, the subrole instances are always visited in ascending order. However, the schema enables that the subrole instances are visited in descending order. The RTCP specifier may define the order via the boolean attribute *startFromFirst* (it is true by default).

Moreover, this one-to-many communication schema enables to define that not all subrole instances have to send this message but only these that fulfill a certain condition. Defining such a condition may reduce network traffic (depending on the platform-specific model) and may reduce the workload on the receiver side. An RTCP specifier may define this via the attribute *condition* that contains textual expressions of MECHATRONICUML's action language (see Section 2.2.2). As an important fact, the transition that specifies an *iterate* with a condition may only fire if at least one subrole instance fulfills the stated condition. Otherwise, the transition is not enabled. If no subrole instance fulfills the condition, then the RTSC has to wait a fixed time before it may evaluate the subrole instances again. The RTCP specifier has to define this time span (as a time value expression) using the attribute *retryAfter*. It must be defined if a condition is defined and has to be greater than zero.

The RTCP specifier defines the delay between sending two messages via the attribute *iterateDelay*, which defines a time value expression. We assume that the delay is always greater than zero. Furthermore, the schema enables to cancel the sending of the *iterate*, i.e., as soon as a certain condition is fulfilled, the iteration stops and no further message is sent. The RTCP specifier may define this condition via the attribute *until*, which contains textual expressions of MECHATRONICUML's action language (as a consequence, the condition may only reference variables and operations and no clocks).

In addition, this schema enables to define an action that shall be executed by each subrole after sending its message. An RTCP specifier may define this via the attribute *action* that contains textual expressions of MECHATRONICUML's action language as well.

In summary, the attribute *iterateDelay* is mandatory and the attributes *condition*, *until*, *startFromFirst*, and *action* are optional attributes while the attribute *retryAfter* must be defined if and only if the attribute *condition* is defined.

Example The coordinator of a platoon of cars that drives at the front of the platoon wants to start a hard braking maneuver. As the distance sensors of the members that drive behind the coordinator cannot react in time to such a braking, the coordinator has to inform the members using the message `brake` one after another beginning with the member at the rear of the platoon. Between each sending of message `brake`, the coordinator waits an appropriate amount of time such that the cars create a safety zone between each other. This is necessary as cars cannot brake with the exact same strength. We depict this example using a sequence diagram in Figure A.7.

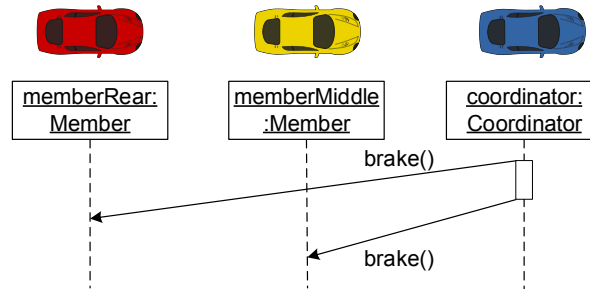


Figure A.7: Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Iterate

Concrete Syntax If the one-to-many communication schema `iterate` is applied, then we extend the transition label as follows. At first, we append the string `'->iterate'` to the name of the message. Then, we extend the string by round parentheses that include the following: First, we show the attribute condition with the prefix `'if: '`. Second, we show the attribute `retryAfter` with the prefix `'retryAfter: '`. Third, we show the attribute action with the prefix `'do: '`. Fourth, we show the attribute `until` with the prefix `'until: '`. Fifth, we show the attribute `iterateDelay` with the prefix `'iterateDelay: '`. Lastly, we show the attribute `startFromFirst` with the prefix `'startFromFirst: '`. We separate the attributes with a pipe. As an example, we write `m1->iterate(if: <conditionExpr>|retryAfter: <retryAfterExpr>|do: <actionExpr>|until: <untilExpr>|iterateDelay: <iterateDelayExpr>)` for a message `m1` that shall be sent via an `iterate` that has the condition `<conditionExpr>`, the retry after delay `<retryAfterExpr>`, the action `<actionExpr>`, the until condition `<untilExpr>`, and the iterate delay of `<iterateDelayExpr>`.

Alternative Schemata If the message is only relevant for one instance of the single role, then use either `unicast` or `loadbalancing` as this may reduce the network traffic (this depends on the underlying platform). If the multi role instance shall send all messages at the same time, then use `multicast`.

Combinability A typical combination is that the multi role sends an `iterate` first and awaits an answer by all instances of the single role via a `multireceive` or by one instance of the single role via a `singlereceive`.

A.5.2 Normalization Template

We show the normalization template for the one-to-many communication schema `iterate` in Figure A.8. Our template covers not only the case that the transition specifies the message that shall be sent via `iterate`, but also the conditions for enabling the transition and the effects that shall be executed within the coordinator after the `iterate` is executed. Furthermore, the template covers the subrole-specific attributes `condition`, `retryAfter`, `action`, `iterateDelay`, and `until`. Concerning the attribute `startFromFirst`, the template is only valid for the value `true`.

The template defines that we add in Step 3.1 (see Figure 3.12) four synchronization channels `iterateCheck<i>`, `iterateCheckDone<i>`, `iterate<i>`, and `iterateDone<i>`, where `<i>` is a unique number of all iterates that are used within this multi role. Moreover, the channels `iterateCheck` and `iterate` are typed over the multi role.

In Step 3.2., we adapt the coordinator as follows:

1. The source state (here: `A`) receives an additional region that checks whether the iteration is possible. In particular, it only starts the check if the conditions of the original transition are fulfilled. Then, the region instructs each subrole instance to check whether the `iterate` is possible (i.e., if at least one subrole fulfills its subrole-specific condition). If it is possible, then each subrole that fulfilled its subrole-specific condition shall send its message. If it is not possible, then the region waits the user-defined time before it checks once again.
2. The transition from `Idle` to `Requesting` adopts the urgency of the original transition, e.g., if the original transition is urgent, then this transition is urgent as well.
3. We replace the original transition by two intermediate states and three transitions. The first transition is fired if at least one subrole instance can send the `iterate`. The second transition initiates the sending of the `iterate`. The third transition is fired as soon as the sending is done and executes the subrole-independent effects of the original transition.
4. If the original transition defines a deadline, then we execute the same five changes as specified in the template of the schema `iterate` that does not use subrole-specific conditions (see Section 3.5.2.8).

In Step 3.3, we add a region with the name `subrole_iterate<i>` to the state `Subrole_Main`, where `<i>` is a unique number of all iterates that are used within this multi role.

Then, in Step 3.4, we add the states `Idle`, `Done_1`, and `Done_2` and seven transitions. The state `Idle` is a composite state that contains the states `Init` and `Requested` and four transitions. `Idle` handles the check whether `iterate` is possible. If this is the case, this information is stored in two variables: the subrole-specific variable `sendMsg` and the multi role variable `iteratePossible<i>`, where `<i>` is a unique number of all iterates that are used within this multi role. If the iteration shall start (because at least one subrole instance fulfills its conditions), then the states `Done_1` and `Done_2` are visited by all subrole instances. This is necessary as the subrole instances do not know if the next subrole instance can send its message or not. Though, a subrole instance that does not send a message will not wait the specified delay (using the parameter `delayTime`). Furthermore, each subrole instance checks whether it shall terminate the iteration. If a subrole instance can send its message, then it first sends the message but checks whether it has to terminate the complete iteration before it delays.

If the RTCP specifier uses `iterate` and sets the attribute `startFromFirst` to the value `false`, then the template changes as follows: (1) We exchange all synchronization channel parameters

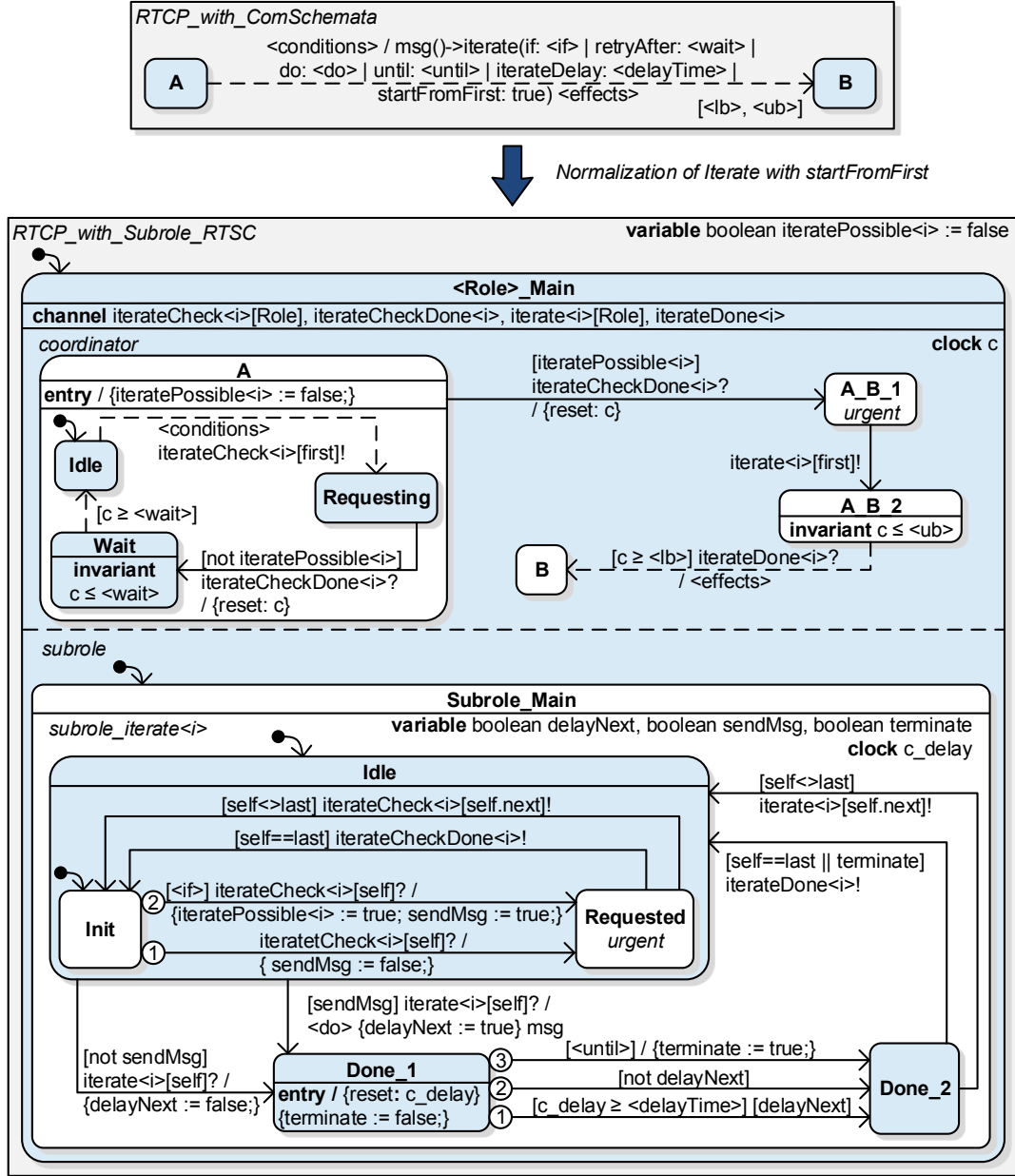


Figure A.8: Normalization Template for Iterate with startFromFirst

that have the value first with the value last. (2) We exchange all synchronization channel parameters that have the value last with the value first. (3) We exchange all synchronization channel parameters that have the value `self.next` with the value `self.prev`.

Note that the transition priorities that we define explicitly in this template are important as the behavior would otherwise differ from the definition.

A.6 Loadbalancing

We define the one-to-many communication schema *loadbalancing* in Section A.6.1. Afterwards, we define its normalization templates in Section A.6.2.

A.6.1 Definition

Description The one-to-many communication schema *loadbalancing* enables the multi role instance to send a message to exactly one instance of the single roles. We assume that each message is a task that needs a certain amount of time. As a consequence, the schema considers the workload of its receivers if they already received a task from the multi role instance that is not done yet. Then, the multi role instance will not send a new task to such a receiver. The schema provides two options to identify whether a receiver still has a task that is not done yet: (1) The task has a pre-defined maximum time limit. Therefore, the receiver will not get a new task before this time limit is expired. The RTCP specifier may define the maximum working time limit via the attribute `maxWT`, which defines a time value expression. (2) The task has no pre-defined maximum time limit but the receiver of the tasks sends a message back as soon as its task is done. The RTCP specifier has to reference the message that indicates that the job is done via the attribute `responseMsg`. Moreover, he may specify a response action via the attribute `responseAction` that the subrole instance has to execute as soon as the job is done. The action is a textual expression of MECHATRONICUML's action language (see Section 2.2.2).

Furthermore, the RTCP specifier may define a condition that the subrole instance has to fulfill before it may send the message. He can specify this via the attribute `condition` that contains textual expressions of MECHATRONICUML's action language. As an important fact, the transition that specifies a *loadbalancing* with a condition may only fire if at least one subrole instance fulfills the stated condition. Otherwise, the transition is not enabled. Furthermore, the subrole instances are always evaluated in ascending order. The first subrole instance that fulfills the condition will send the message. If no subrole instance fulfills the condition, then the RTSC has to wait a fixed time before it may evaluate the subrole instances again. The RTCP specifier has to define this time span (as a time value expression) using the attribute `retryAfter`. It has to be greater than zero. In addition, this schema enables to define an action that shall be executed by the subrole instance after sending the message. The RTCP specifier may define this via the attribute `action` that contains textual expressions of MECHATRONICUML's action language as well.

In summary, the RTCP specifier either has to use the attribute `maxWT` or the attribute `responseMsg`. If the attribute `responseMsg` is used, then he may optionally specify

the attribute `responseAction`. Moreover, the attributes `condition` and `action` are optional attributes while the attribute `retryAfter` must be defined if and only if the attribute `condition` is defined.

Example The coordinator of a platoon of cars wants to externalize the task to compute the optimal route to a specific destination. In our example, we assume that the member that drives at the end of the platoon shall do this task. After some time, the coordinator decides that the maximum convoy speed may be even higher. As a consequence, it wants to start another computation for the optimal route. As the first computation is not done yet, the coordinator sends this new task to the member that drives at the front of the platoon. We depict this example using a sequence diagram in Figure A.9.

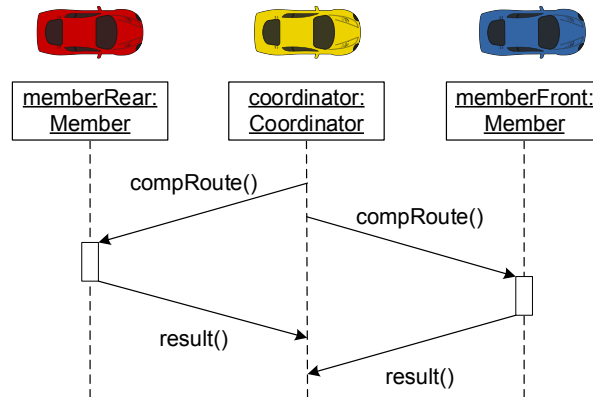


Figure A.9: Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Loadbalancing

Concrete Syntax If the one-to-many communication schema loadbalancing is applied, then we extend the transition label as follows. At first, we append the string `'->loadbalancing'` to the name of the message. Then, we extend the string by round parentheses that include the following: First, we show the attribute condition with the prefix `'if: '`. Second, we show the attribute `retryAfter` with the prefix `'retryAfter: '`. Third, we show the attribute action with the prefix `'do: '`. If loadbalancing defines a maximum working time, then we additionally show the attribute `maximumWorkingTime` with the prefix `'maxWT: '`. If loadbalancing defines a response message, then we additionally show the attribute `responseMsg` with the prefix `'responseMsg: '` and, if it is defined, the attribute `responseAction` with the prefix `'responseAction: '`. We separate the attribute with a pipe. As an example, we write `m1->loadbalancing(if: <conditionExpr>|retryAfter: <retryAfterExpr>|do: <actionExpr>|maxWT: <maxWTExpr>)` for a message `m1` that shall be sent via a loadbalancing that has the condition `<conditionExpr>`, the retry after delay `<retryAfterExpr>`, the action `<actionExpr>`, the until condition `<untilExpr>`, and the maximum working time `<maxWTExpr>`.

Alternative Schemata If a balancing is not necessary, then use the schema `unicast`.

Combinability A typical combination is that the multi role receive a `singlereceive` or a `multireceive` first and sends several messages afterwards via the one-to-many communication schema loadbalancing.

A.6.2 Normalization Templates

We provide two different templates for the two options that this one-to-many communication schema provides. In particular, we define the template for the option *maximum working time* in Section A.6.2.1. Then, we define the template for the option *message response* in Section A.6.2.2.

A.6.2.1 Option: Maximum Working Time

We show the normalization template for the one-to-many communication schema loadbalancing and the option *maximum working time* in Figure A.10. Our template covers not only the case that the transition specifies the message that shall be sent via loadbalancing, but also the conditions for enabling the transition and the effects that shall be executed within the coordinator after the loadbalancing is executed. Furthermore, the template covers the subrole-specific attributes condition, retryAfter, action, and maxWT.

The template defines that we add in Step 3.1 (see Figure 3.12) four synchronization channels `loadbalancingCheck<i>`, `loadbalancingCheckDone<i>`, `loadbalancing<i>`, and `loadbalancingDone<i>`, where `<i>` is a unique number of all loadbalancings that are used within this multi role. Moreover, the channel `loadbalancingCheck` is typed over the multi role.

In Step 3.2., we adapt the coordinator as follows:

1. The source state (here: A) receives an additional region that checks whether the loadbalancing is possible. In particular, it only starts the check if the conditions of the original transition are fulfilled. Then, the region instructs each subrole instance to check whether the loadbalancing is possible (i.e., if at least one subrole fulfills its subrole-specific condition). Hereby, the coordinator region iterates all subrole instances in ascending order. As soon as a subrole instance fulfills its condition and has no open task (anymore), the iteration is stopped. Afterwards, the coordinator instructs this subrole instance to send the message. If no subrole instance fulfills its condition, then the region waits the user-defined time before it checks once again.
2. The transition from `Idle` to `Requesting` adopts the urgency of the original transition, e.g., if the original transition is urgent, then this transition is urgent as well.
3. We replace the original transition by two intermediate states and three transitions. The first transition is fired if at least one subrole instance can send the loadbalancing. The second transition initiates the sending of the loadbalancing. The third transition is fired as soon as the sending is done and executes the subrole-independent effects of the original transition.
4. If the original transition defines a deadline, then we execute the same five changes as specified in the template of the schema multicast that does not use subrole-specific conditions (see Section 3.5.2.8).

In Step 3.3, we add a region with the name `subrole_loadbalancing_maxworktime<i>` to the state `Subrole_Main`, where `<i>` is a unique number of all loadbalancings that are used within this multi role.

Then, in Step 3.4, we add the states `Idle` and `Send` and two transitions. The state `Idle` is a composite state that contains the states `Init`, `Check`, and `Requested` as well as six transitions. `Idle` handles the check whether loadbalancing is (already) possible (either the subrole instance

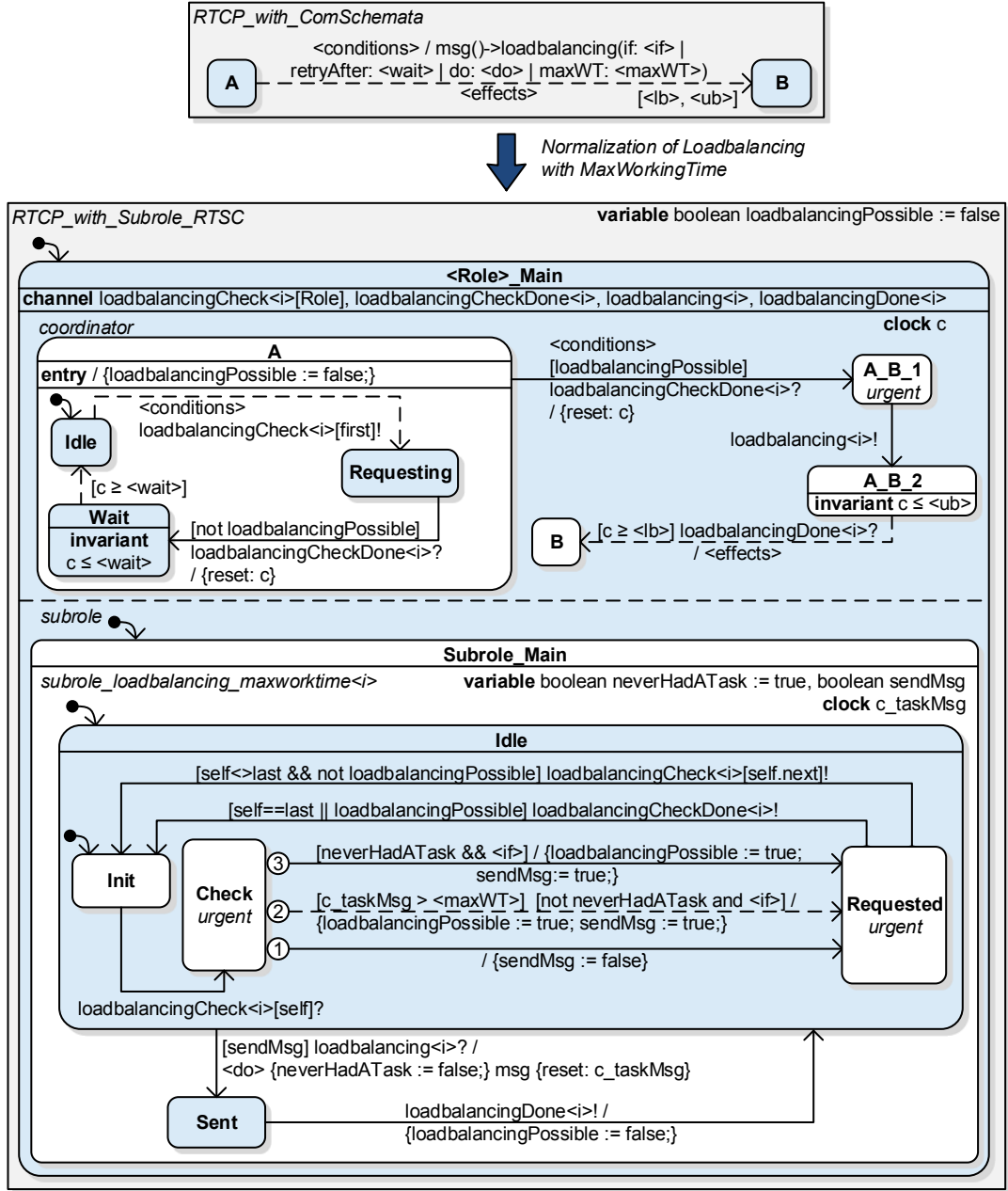


Figure A.10: Normalization Template for LoadBalancing with MaxWorkingTime

never had a task or the maximum working time has expired). If this is the case, this information is stored in two variables: the subrole-specific variable `sendMsg` and the multi role variable `loadbalancingPossible<i>`, where `<i>` is a unique number of all loadbalancings that are used within this multi role. The state `Sent` is only visited by the subrole instance that sends the message.

Note that the transition priorities that we define explicitly in this template are important as the behavior would otherwise differ from the definition.

A.6.2.2 Option: Message Response

We show the normalization template for the one-to-many communication schema `loadbalancing` and the option *message reponse* in Figure A.11. Our template covers not only the case that the transition specifies the message that shall be sent via `loadbalancing`, but also the conditions for enabling the transition and the effects that shall be executed within the coordinator after the `loadbalancing` is executed. Furthermore, the template covers the subrole-specific attributes `condition`, `retryAfter`, `action`, `responseMsg`, and `responseAction`.

The template defines that we add in Step 3.1 (see Figure 3.12) four synchronization channels `loadbalancingCheck<i>`, `loadbalancingCheckDone<i>`, `loadbalancing<i>`, and `loadbalancingDone<i>`, where `<i>` is a unique number of all loadbalancings that are used within this multi role. Moreover, the channel `loadbalancingCheck` is typed over the multi role.

In Step 3.2., we adapt the coordinator as follows:

1. The source state (here: `A`) receives an additional region that checks whether the `loadbalancing` is possible. In particular, it only starts the check if the conditions of the original transition are fulfilled. Then, the region instructs each subrole instance to check whether the `loadbalancing` is possible (i.e., if at least one subrole fulfills its subrole-specific condition). Hereby, the coordinator region iterates all subrole instances in ascending order. As soon as a subrole instance fulfills its condition and has no open task (anymore), the iteration is stopped. Afterwards, the coordinator instructs this subrole instance to send the message. If no subrole instance fulfills its condition, then the region waits the user-defined time before it checks once again.
2. The transition from `Idle` to `Requesting` adopts the urgency of the original transition, e.g., if the original transition is urgent, then this transition is urgent as well.
3. We replace the original transition by two intermediate states and three transitions. The first transition is fired if at least one subrole instance can send the `loadbalancing`. The second transition initiates the sending of the `loadbalancing`. The third transition is fired as soon as the sending is done and executes the subrole-independent effects of the original transition.
4. If the original transition defines a deadline, then we execute the same five changes as specified in the template of the schema `multicast` that does not use subrole-specific conditions (see Section 3.5.2.8).

In Step 3.3, we add the two regions `subrole_loadbalancing_responseA<i>` and `subrole_loadbalancing_responseB<i>` to the state `Subrole_Main`, where `<i>` is a unique number of all loadbalancings that are used within this multi role. The region `subrole_loadbalancing_responseA<i>` is responsible for sending a message to the connected

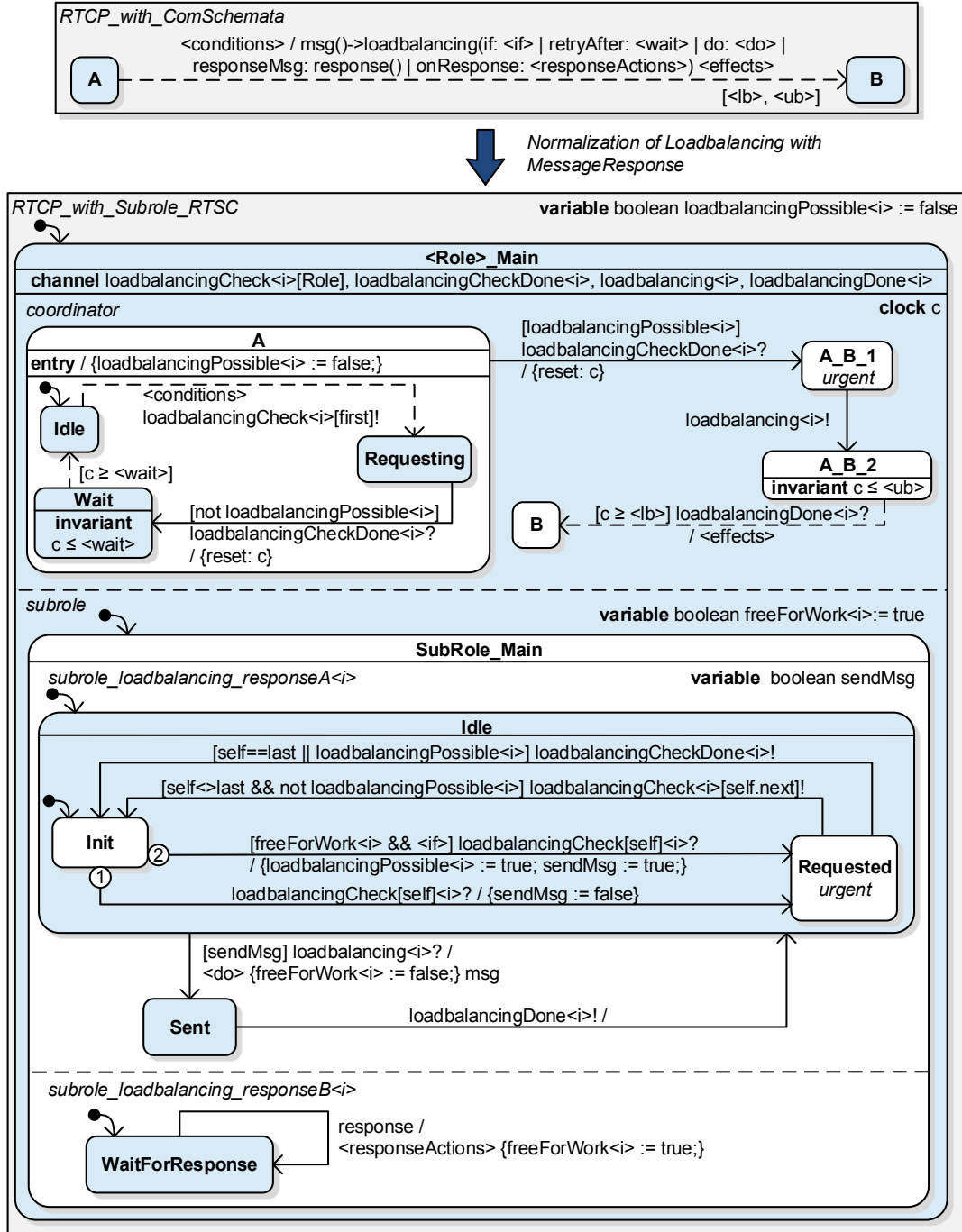


Figure A.11: Normalization Template for LoadBalancing with MessageResponse

instance of the single role, while the region `subrole_loadbalancing_responseB<i>` is responsible for receiving the response message from the connected instance of the single role.

Then, we execute Step 3.4 as follows: We add the states `Idle` and `Sent` and two transitions to the region `subrole_loadbalancing_responseA<i>`. The state `Idle` is a composite state that contains the states `Init`, `Check`, and `Requested` as well as four transitions. `Idle` handles the check whether loadbalancing is (already) possible (either the subrole instance never had a task or the response message came already back). If this is the case, this information is stored in two variables: the subrole-specific variable `sendMsg` and the multi role variable `loadbalancingPossible<i>`, where `<i>` is a unique number of all loadbalancings that are used within this multi role. The state `Sent` is only visited by the subrole instance that sends the message. Concerning the region `subrole_loadbalancing_responseB<i>`, we add a single state `WaitForResponse` and a self-transition. As soon as a response message is available, the transition fires, executes the response action (if it was defined), and stores in the subrole variable `freeForWork<i>` (`<i>` is a unique number of all loadbalancings that are used within this multi role) that it is available for a new task.

Note that the transition priorities that we define explicitly in this template are important as the behavior would otherwise differ from the definition.

B

CATALOG OF REAL-TIME COORDINATION PATTERNS

In Section 6.3, we briefly introduced our nine Real-Time Coordination Patterns that we identified so far. Moreover, we already gave a detailed description of the pattern *Acquire Assurance* using our descriptions format. In this appendix, we describe the remaining patterns using our descriptions format as well. We published a former version of this catalog in [*DBHT12].

B.1 Request Information

Name Request Information

Overview This pattern enables that a role *Requester* can request information from a role *Provider*. The pattern considers that providing the information takes time. Moreover, only one request at a time is allowed.

Also Known As Event-Listener [San11]

Context Information exchange between two roles

Problem A role *Provider* has information, e.g., sensor values, that another role *Requester* requires. The *Requester* does not need this information periodically. Moreover, as the role connector is unreliable, the *Requester* cannot be sure whether the role *Provider* receives the request. Furthermore, the reply of the *Provider* may get lost during transmission.

Solution Define an RTCP that enables the role *Requester* to request an information at any time. The *Provider* has to reply in a certain time that is known to the *Requester*. Moreover, the *Requester* needs to know the maximum message delay. As a consequence, the *Requester* can calculate the latest time when an answer may arrive (two times the maximum message delay plus the maximum reply time of the *Provider*). Therefore, the *Requester* can be sure whether a message got lost and, thus, can send a new request. Noteworthy, the pattern assumes that the *Requester* cannot start a new request as long as a request is ongoing.

Structure We show the structure of the pattern in the upper part of Figure B.1.

The pattern is bidirectional and consists of the two roles Requester and Provider. Both roles are in/out roles. The role Requester may send the message request while the role Provider may send the message reply.

Both roles have a single buffer of size 1. In addition, both buffers shall never be forced to discard a message. Moreover, the role connector has the following QoS assumptions: the connector is unreliable, the maximum message delay is given as a parameter, and the message order shall be preserved (i.e., overtaken messages will be discarded).

The pattern defines three parameters:

- `maxMessageDelay` defines the maximum message delay.
- `replytime` defines how much time the Provider has to retrieve the information.
- `offset` defines the additional time that the Requester has to wait until it decides that no answer may be received by the Provider anymore.

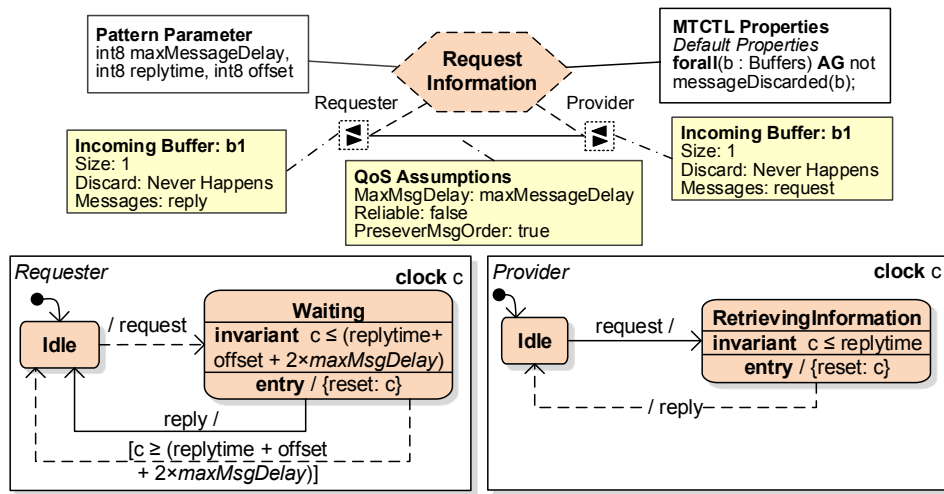


Figure B.1: Real-Time Coordination Pattern Request Information

Behavior We show the behavior of the pattern in the lower part of Figure B.1.

The role Requester is initially in state `Idle`. At any point in time, the Requester can send the message request to the role Provider and change to state `Waiting`. Upon the activation of `Waiting`, an entry-action resets the clock `c`. If the message reply arrives in the buffer, then the role Requester may consume it and switch back to state `Idle`. The second outgoing transition gets only enabled if either the message request or the message reply got lost during transmission, i.e, this transition and the state invariant encode a timeout. As soon as this transition is enabled, the invariant of state `Waiting` enforces the role Requester to leave the state and switch back to state `Idle`.

The role Provider is initially in state `Idle` as well. At any point in time, the role Provider may receive the message request from the role Requester. Then, the role Provider has to change to state `RetrievingInformation` to start with the retrieving of the information. Upon the activation of `RetrievingInformation`, the clock `c` is reset via an entry-action. When the role has retrieved the information, it may at any point in time answer via

message reply but not later than the stated invariant. After the message reply has been sent, the role Provider switches back to state Idle.

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4). In addition, no buffer may discard a message: **forall**(**b** : Buffers) **AG not** messageDiscarded(**b**);

A verified configuration of the parameters such that the pattern fulfills all verification properties simultaneously is as follows:

- replytime: 60 ms
- maxMessageDelay: 10 ms
- offset: 1 ms

Consequences A benefit of the pattern is that it avoids a polling by the Requirer if the request is already done. Moreover, it avoids a polling by the Provider if an order is available.

As a liability, the Requirer may only make one request at a time. Moreover, if one of the messages get lost, then the Requirer has to wait until the timeout happens, i.e., until an answer of the Provider is no longer possible. If the probability of message loss is high and the maximum working time of the Provider is very long, then the efficiency of processing several requests is significantly low.

Examples We illustrate an example for this pattern in the Coordinating Delta Robots System [San11]. Here the pattern is used for RTCP HMI_STRATEG_REQ that retrieves the user-defined strategy from the component human-machine-interface.

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role Provider to a multi role. Therefore, multiple instances of role Requirer are available that state a request to the Provider. The Requirer may consume each message via the one-to-many communication schemata singlereceive and may answer to each request via the one-to-many communication schemata unicast. (2) The pattern user can change the role Requirer to a multi role. Therefore, multiple instances of role Provider are available that receive an request from the Requirer. Hereby, it is possible that all instances of role Provider have to do the same request: Then, the role Requirer should send the request via the one-to-many communication schemata multicast and wait for the answers via the one-to-many communication schemata multireceive. As another possibility, every request is stated to only one Provider: Then, the role Requirer can send the request via the one-to-many communication schemata unicast or loadbalancing and wait for the answer via the one-to-many communication schemata unireceive.

If the Requirer is able to recognize that a request is no longer necessary, then the Requirer should be able to send the message cancel in order to cancel the work of the Provider.

If the connector is reliable (i.e., message loss is not possible), then all stated verification properties are still useful. However, the pattern user should apply several changes to make the model more compact: (1) The role Requirer does not need the clock *c* anymore. (2) The state Requirer.Waiting does not need its invariant and its entry action

anymore. (3) The transition Requester.Waiting→Idle that defines the clock constraint should be removed.

Alternative Patterns If the pattern user wants to delegate a job that may fail, he should either use Fail-safe Delegation (see Section B.3) or Fail-operational Delegation (see Section B.2).

Combinability The pattern user can combine it with the pattern Acquire Assurance (see Section 6.3.5): As long as the assurance is active, the pattern may be used to request additional information.

Search Terms request information, retrieve data, get values

B.2 Fail-operational Delegation

Name Fail-operational Delegation

Also Known As -

Overview “This pattern realizes a delegation of [an order] from a role Master to a role Slave. The Slave executes the [order] in a certain time and answers regarding success or failure. The pattern assumes that a failure is not safety-critical. [Moreover,] only one delegation at a time is allowed.” [*DBHT12]

Context Delegate orders to another role

Problem “If the communication is asynchronous and the communication channel is unreliable, [then] the role that sends the [order] does not know [whether] the other role has received it. Though, the [order] has to be done.” [*DBHT12] A failed execution of the order does not need to be handled before a new order can be delegated.

Solution “Define an RTCP that enables a role Master to delegate [orders] to a Slave. [...] The Master delegates the [order] and waits for its completion. After a specified time, the Master cancels the waiting. The Slave executes this [order] in a certain time and reports [whether] the [order] was done successfully or [whether] the execution failed. [...] If the execution of an order failed, the next order can be delegated immediately. Therefore, this pattern should only be used, when a failed order does not produce an unsafe state that must be repaired before the next [order] can be delegated.” [*DBHT12]

Structure We show the structure of the pattern in the upper part of Figure B.2.

“The pattern [is bidirectional and] consists of the two roles Master and Slave. Both roles are in/out roles. The Master may send the message order to the Slave. The Slave may send the messages done and fail to the Master.” [*DBHT12]

Both roles have a single buffer of size 1. In addition, both buffers shall never be forced to discard a message. Moreover, the role connector has the following QoS assumptions: the connector may be unreliable, the maximum message delay is given as a parameter, and the message order shall be preserved (i.e., overtaken messages will be discarded).

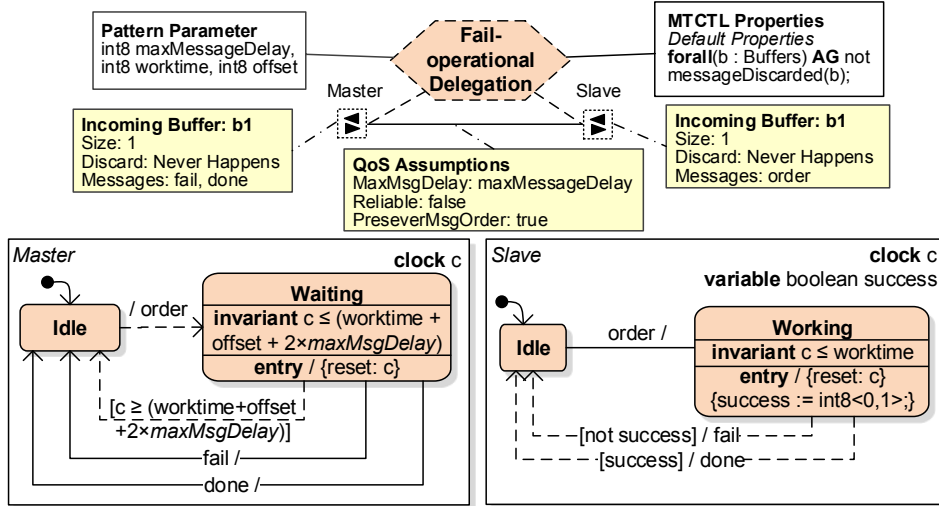


Figure B.2: Real-Time Coordination Pattern Fail-operational Delegation

The pattern defines three parameters:

- `maxMessageDelay` defines the maximum message delay.
- `worktime` defines how long the Provider may work in order to fulfill the order.
- `offset` defines the additional time that the Master has to wait until it decides that no answer may be received by the Slave anymore.

Behavior We show the behavior of the pattern in the lower part of Figure B.2.

The role Master is initially in state `Idle`. At any point in time, the Master can send the message `order` to the Slave and change to state `Waiting`. Upon the activation of state `Waiting`, an entry-action resets the clock `c`. If the message `done` or the message `fail` arrives in the buffer, then the role Master has to consume it and has to switch back to state `Idle`. The third outgoing transition gets only enabled if either the message `order`, the message `done`, or the message `fail` got lost during transmission, i.e., this transition and the state invariant encode a timeout. As soon as this transition is enabled, the invariant of state `Waiting` enforces the role Master to leave the state and switch back to state `Idle`.

The role Slave consists of the initial state `Idle` and the state `Working`. At any point in time, the role Slave may receive the message `order` from the Master. This message triggers the transition from `Idle` to `Working` and leads to the consumption of the message. Upon the activation of state `Working`, the clock `c` is reset via an entry-action. An invariant using the clock `c` ensures that `Working` is left not later than the allowed working time after its activation. There are two outgoing transitions that both lead to state `Idle`. The message `done` shall be sent to the Master if the order could be completed successfully. If the order cannot be completed in time or an error occurs, the message `fail` will be sent to the Master.

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4). In addition, no buffer may discard a message: **forall**(*b* : Buffers) **AG** not messageDiscarded(*b*);

A verified configuration of the parameters such that the pattern fulfills all verification properties simultaneously is as follows:

- maxMessageDelay: 5 ms
- worktime: 10 ms
- offset: 1 ms

Consequences A benefit of the pattern is that it avoids a polling by the Master if the order is already done. Moreover, it avoids a polling by the Slave if an order is available.

As a liability, the Master may only make one order at a time. Moreover, if one of the messages get lost, then the Master has to wait until the timeout happens, i.e., until an answer of the Slave is no longer possible. If the probability of message loss is high and the maximum working time of the Slave is very long, then the efficiency of processing several requests is significantly low.

Examples “[We illustrate] an example for this pattern in the BeBot Coordination Example [*DPP+16]. Here the pattern is used [in RTCP Delegation] to delegate the [order] of checking the validity of a target position [of the BeBot].” [*DBHT12]

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role Slave to a multi role so that he can have multiple instances of role Master that send an order to the Slave. The Slave may consume each message via the one-to-many communication schemata singlereceive and may answer to each order via the one-to-many communication schemata unicast. (2) The pattern user can change the role Master to a multi role so that he can have multiple instances of role Slave that receive an order from the Master. Hereby, it is possible that all instances of role Slave have to do the same order: Then, the role Master should send the order via the one-to-many communication schemata multicast and wait for the answers via the one-to-many communication schemata multireceive. As another possibility, every order is assigned to only one Slave: Then, the role Master can send the order via the one-to-many communication schemata unicast or loadbalancing and wait for the answers via the one-to-many communication schemata unireceive.

If the Master is able to recognize that a failure in the system occurred and the Slave should / must stop its work (e.g., to prevent a safety-critical situation), then the Master should / must be able to send the message cancel in order to cancel the work of the Slave.

If the connector is reliable (i.e., message loss is not possible), then all stated verification properties are still useful. However, the pattern user should apply several changes to make the model more compact: (1) The role Master does not need the clock *c* anymore. (2) The state Master.Waiting does not need its invariant and its entry action anymore. (3) The transition Master.Waiting→Idle that defines the clock constraint should be removed.

Alternative Patterns “Use Fail-safe Delegation (see Section B.3), if a failure leads to a safety-critical situation that must be corrected first before a new [order] can be delegated.” [*DBHT12]

Moreover, if the order is to block a certain action of the other role, then another alternative might be the pattern Block Execution (see Section B.7).

In addition, if the approval of the other role is required (if it may execute an order), then an alternative is the pattern Acquire Assurance (see Section 6.3.5).

Furthermore, if no reply is necessary (e.g., as the task can always be successfully completed), then another alternative is the pattern Event-based Transmission (see Section B.4).

Combinability The pattern user can combine it with the pattern Acquire Assurance (see Section 6.3.5): As long as the assurance is active, the pattern may be used to delegate additional orders.

Search Terms delegate order, send task, define job

B.3 Fail-safe Delegation

Name Fail-safe Delegation (initially defined by [Rie11] within the software model of an existing kneading machine)

Also Known As -

Overview “This pattern realizes a delegation of [an order] from a role Master to a role Slave. The Slave executes the [order] in a certain time and answers regarding success or failure. If the execution fails, no other [order] may be delegated until the Master ensures that the failure has been corrected. Moreover, only one [order] at a time is allowed.” [*DBHT12]

Context Delegate orders to another role

Problem “If the communication is asynchronous and the communication channel is unreliable, the role that sends the [order] does not know [whether] the other role has received it. Though, the [order] has to be done.” [*DBHT12] A failed execution of the order must be handled before a new order can be delegated.

Solution “Define [an RTCP] that enables a role Master to delegate [orders] to a Slave. A failed [order] execution is handled before a new [order] can be delegated. The Master delegates the [order] and wait for its completion. After a specified time, the Master cancels the waiting. The Slave executes this [order] in a certain time and reports [whether] the [order] was done successfully or [whether] the execution failed. If it failed, the Slave does not execute new [order]s until the Master sends the signal that the error is resolved.” [*DBHT12]

Structure We show the structure of the pattern in the upper part of Figure B.3.

The pattern is bidirectional and consists of the two roles Master and Slave. Both roles are in/out roles. The Master may send the messages order and continue. The Slave may send the messages done and fail.

Both roles have a single buffer of size 1. In addition, both buffers shall never be forced to discard a message. Moreover, the role connector has the following QoS assumptions: the connector may be unreliable, the maximum message delay is given as a parameter, and the message order shall be preserved (i.e., overtaken messages will be discarded).

The pattern defines three parameters:

- `maxMessageDelay` defines the maximum message delay.
- `worktime` defines how long the Provider may work in order to fulfill the order.
- `offset` defines the additional time that the Master has to wait until it decides that no answer may be received by the Slave anymore.

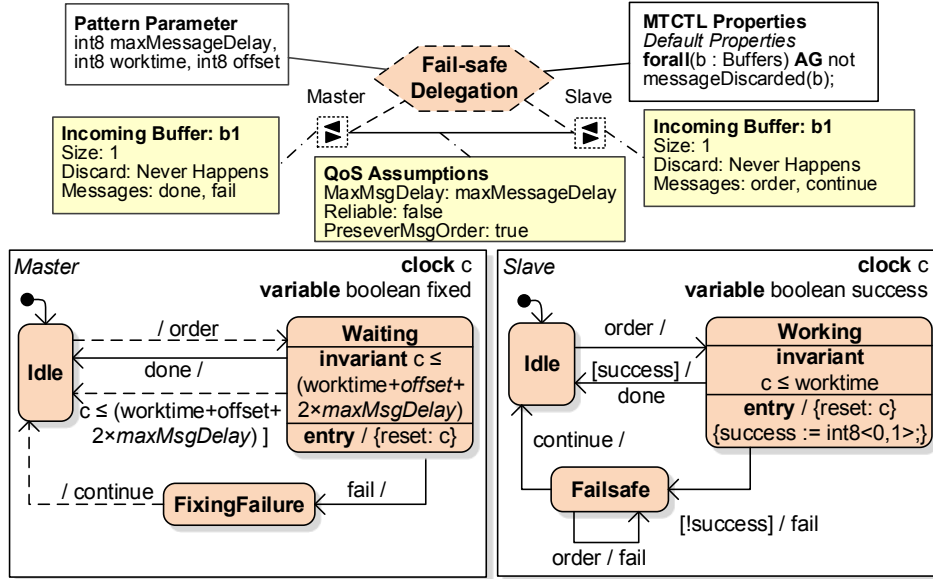


Figure B.3: Real-Time Coordination Pattern Fail-safe Delegation

Behavior We show the behavior of the pattern in the lower part of Figure B.3.

The role Master is initially in state `Idle`. At any point in time, the Master can send the message `order` to the Slave and change to state `Waiting`. Upon the activation of state `Waiting`, an entry-action resets the clock `c`. If the message `done` arrives in the buffer, then the role Master has to consume it and has to switch back to state `Idle`. If the message `fail` arrives in the buffer, then the role Master has to consume it and has to switch to state `FixingFailure`. The third outgoing transition gets only enabled if either the message `order`, the message `done`, or the message `fail` got lost during transmission, i.e., this transition and the state invariant encode a timeout. As soon as this transition is enabled, the invariant of state `Waiting` enforces the role Master to leave the state and switch back to state `Idle`. If the Master is in state `FixingFailure`, then the pattern assumes that the role executes internal actions to resolve the failure. As soon as the failure is resolved, the role Master sends message `continue` and changes back to state `Idle`.

The role Slave consists of the initial state `Idle` and the states `Working` and `Failsafe`. At any point in time, the Slave may receive the message `order` from the Master. This

message triggers the transition from *Idle* to *Working* and leads to the consumption of the message. Upon the activation of state *Working*, an entry-action resets the clock *c*. An invariant using the clock *c* ensures that the state *Working* is not left later than the allowed working time after its activation. The message *done* shall be sent to the *Master* if the order could be completed successfully. Then, the role should switch back to state *Idle*. Instead, the message *fail* shall be sent to the *Master* if the order cannot be completed in time or an error occurs. Then, the role should switch to state *Failsafe*. As long as the role is in state *Failsafe* and a new message of type *order* arrives, the role will reply with the message *fail*. However, if the role receives the message *continue*, then it will switch back to state *Idle*.

Verification Properties The pattern has to ensure all default properties (see Section 4.4). In addition, no buffer may discard a message: **forall**(*b* : *Buffers*) **AG** not *messageDiscarded*(*b*);

A verified configuration of the parameters such that the pattern fulfills all verification properties simultaneously is as follows:

- *maxMessageDelay*: 50 ms
- *worktime*: 40 ms
- *offset*: 1 ms

Consequences A benefit of the pattern is that it avoids a polling by the *Master* if the order is already done. Moreover, it avoids a polling by the *Slave* if an order is available.

As a liability, the *Master* may only make one order at a time. Moreover, if one of the messages get lost, then the *Master* has to wait until the timeout happens, i.e., until an answer of the *Slave* is no longer possible. If the probability of message loss is high and the maximum working time of the *Slave* is very long, then the efficiency of processing several requests is significantly low.

Examples An autonomous transportation robot is working in a warehouse and has to move goods. A centralized system (the *Master*) delegates the robot (the *Slave*) to move goods to a certain position. The robot may crash, which causes a blocking of the gangway or errors at the hardware of the robot. If this is the case, the system must clean the place of accident and has to make sure that the robot is free of errors before it may delegate a new order to the robot.

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role *Slave* to a multi role so that he can have multiple instances of role *Master* that send an order to the *Slave*. The *Slave* may consume each message via the one-to-many communication schemata *singlereceive* and may answer to each order via the one-to-many communication schemata *unicast*. (2) The pattern user can change the role *Master* to a multi role so that he can have multiple instances of role *Slave* that receive an order from the *Master*. Hereby, it is possible that all instances of role *Slave* have to do the same order: Then, the role *Master* should send the order via the one-to-many communication schemata *multicast* and wait for the answers via the one-to-many communication schemata *multireceive*. As another possibility, every order is assigned to only one *Slave*: Then, the role *Master* can send

the order via the one-to-many communication schemata unicast or loadbalancing and wait for the answers via the one-to-many communication schemata unireceive.

If the Master is able to recognize that a failure in the system occurred and the Slave should / must stop its work (e.g., to prevent a safety-critical situation), then the Master should / must be able to send the message `cancel` in order to cancel the work of the Slave.

If the connector is reliable (i.e., message loss is not possible), then all stated verification properties are still useful. However, the pattern user should apply several changes to make the model more compact: (1) The role Master does not need the clock `c` anymore. (2) The state `Master.Waiting` does not need its invariant and its entry action anymore. (3) The transition `Master.Waiting→Idle` that defines the clock constraint should be removed.

Alternative Patterns If the pattern user wants to realize a fail-operational strategy, then he should use the pattern `Fail-operational Delegation` (see Section B.2), i.e., the pattern assumes that a failed order does not need to be handled and the Master can directly send a new order.

Moreover, if the order is to block a certain action of the other role, then the pattern user can also use the pattern `Block Execution` (see Section B.7).

In addition, if the pattern user needs the approval of the other role whether it may execute an order, then he can also use the pattern `Acquire Assurance` (see Section 6.3.5).

Combinability The pattern user can combine it with the pattern `Acquire Assurance` (see Section 6.3.5): As long as the assurance is active, the pattern may be used to delegate additional orders.

Search Terms `delegate order`, `send task`, `define job`

B.4 Event-based Transmission

Name Event-based Transmission

Also Known As -

Overview This pattern can be used to transmit information from a sender to a receiver if an event occurs (e.g., a condition becomes true).” [*DBHT12]

Context Information exchange between two roles

Problem A role Receiver requires information from a role Sender. However, the Sender shall only send an information to the Receiver if a new information is available.

Solution The transmission of the information should be event-based such that the role Sender may decide when it sends the message without having an upper time bound when the message shall be sent.

Structure We show the structure of the pattern in the upper part of Figure B.4.

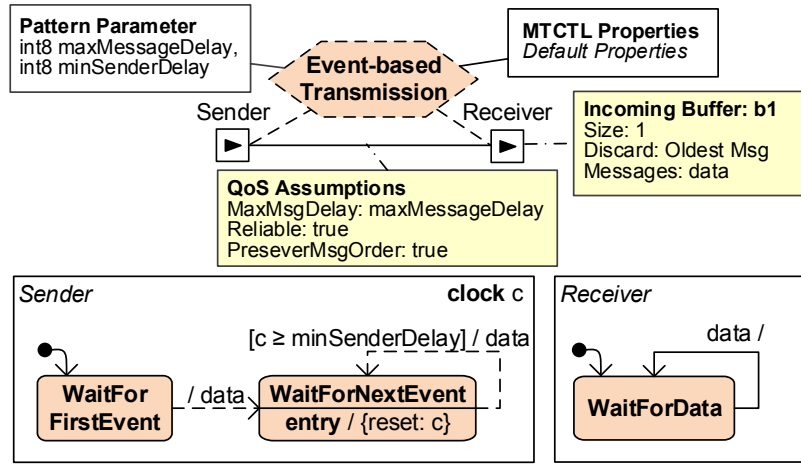


Figure B.4: Real-Time Coordination Pattern Event-based Transmission

The pattern is unidirectional and consists of the two roles Sender and Receiver. The role Sender is an out-role and the role Receiver is an in-role. The role Sender may only send one message to the receiver, which we call data.

The role Receiver defines a single buffer of size 1. In addition, the buffer shall discard the oldest message in the buffer if it is already full. This is useful as the role Receiver is typically interested in the newest data. Moreover, the role connector has the following QoS assumptions: the connector must be reliable, the maximum message delay is given as a parameter, and the message order shall be preserved (i.e., overtaken messages will be discarded).

The pattern defines two parameters:

- maxMessageDelay defines the maximum message delay.
- minSenderDelay defines a time span that the Sender has to wait after sending the message data before it may send another message.

Behavior The behavior is shown in the lower part of Figure B.4.

“The role Sender consists of the initial state WaitForFirstEvent and the state WaitForNextEvent. If the first event happens (e.g., a condition gets true), the role sends the message data to role Receiver and switches to state WaitForNextEvent. Then, the role Sender has to wait a pre-defined minimum time before it may send another message of type data. This minimum waiting time is necessary due to the semantics of RTSCs. Otherwise, the RTSC could send the message infinite times without clock progress.

The role Receiver only consists of the initial state WaitForData. If the message data arrives in the buffer, the role has to urgently consume it via the self-transition.

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4).

A verified configuration such that the pattern fulfills all verification properties simultaneously is as follows:

- maxMessageDelay: 50 ms
- minSenderDelay: 55 ms

Consequences As a benefit, this pattern reduces the communication exchange to a minimum, i.e., data is only sent if a certain condition is fulfilled or if an event occurs and not periodically. Therefore, the role Sender never has to send a single message if the event never occurs. Moreover, the Receiver does not need to ask whether new data is available.

As a liability, the pattern user has to pre-define the minimum sender delay of the role Sender.

Examples In an extended version of our overtaking scenario (see Section 1.1), this pattern could be used for the communication between the red car and a satellite that has information about the safety of the street, e.g., whether glaze ice is on it. When using this pattern, the satellite would act as the Sender and the red car as the Receiver. Then, the satellite would be able to inform the red car as soon as the safety of the street changes.

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role Receiver to a multi role so that he can have multiple instances of role Sender that send data to the single instance of role Receiver. The receiving may be realized via the one-to-many communication schemata `singlereceive` or `multireceive`. (2) The pattern user can change the role Sender to a multi role so that he can have multiple instances of role Receiver that receive data from the single instance of role Sender. Hereby, it is possible that all instances of role Receiver shall receive the same data (this may be realized via the one-to-many communication schemata `multicast` or `iterate`) or every data is sent to only one Receiver (this may be realized via the one-to-many communication schemata `loadbalancing` or `unicast`).

If the pattern user adds conditions to the transition of the Receiver, then the consuming of the receiving message may be delayed. Therefore, a buffer size of one can lead to the fact that incoming messages are discarded. If the pattern user does not want that messages are discarded, then he needs to add the following MTCTL property: **AG not messageDiscarded(b1)**; If this property fails, then the pattern user needs to increase the buffer size. Moreover, the pattern user should then change the discard strategy of the buffer to “Never Happens”.

Instead of discarding the oldest message at role Receiver, the pattern user can also define that he wants to discard the incoming message.

The pattern user can also use this pattern with an unreliable connector. However, the role Sender will not be able to know whether any of the messages will ever be received.

Alternative Patterns A specialized version of this pattern is Limit Observation (see Section B.8) as it also sends the messages to a role if a certain event (a limit is violated or redeemed) occurs.

If a periodic sending is necessary, then the pattern user can use the pattern Periodic Transmission (see Section B.5).

If the data being sent is a request that needs a reply, then the pattern user should use the pattern Request Information (see Section B.1). In contrary, if the data being sent is an order that needs an answer whether it succeeded or failed, then the pattern user should either use the pattern Fail-operational Delegation (see Section B.2) or Fail-safe Delegation (see Section B.3).

Combinability The pattern user can combine it with the pattern Acquire Assurance (see Section 6.3.5): As long as the assurance is active, the pattern may be used to request additional information.

Moreover, the pattern user can use the pattern Block Execution (see Section B.7) in order to block the Sender to send new data to the Receiver.

Search Terms transmit data, send information, receive data

B.5 Periodic Transmission

Name Periodic Transmission (initially defined by [*BDG+11] and [Dre11])

Overview “This pattern can be used to periodically transmit information from a sender to a receiver. If the receiver does not get the information within a certain time, a specified behavior must be activated to prevent a safety-critical situation.” [*DBHT12]

Also Known As -

Context Information exchange between two roles

Problem The receiver requires a periodical update of a certain information. Otherwise, a safety-critical situation may occur. Though, the message exchange is unreliable. Therefore, if the update is not received, the receiver has to execute actions in order to prevent safety-critical situations.

Solution Define an RTCP that enables a role Sender to periodically transmit data to a role Receiver. The Receiver has to handle the consequences of the unreliable connector: it must be able to detect that a message got lost using the information of the period of the role Sender and the maximum message delay. If the role Receiver detects that a message got lost, then it should switch its state and execute actions that prevent safety-critical situations. As soon as the data arrives again, it can return to its default behavior.

Structure We show the structure of the pattern in the upper part of Figure B.5.

The pattern is unidirectional and consists of the two roles Sender and Receiver. The role Sender is an out-role and the role Receiver is an in-role. The role Sender may only send the message data.

The role Receiver defines a single buffer of size 1. In addition, the buffer of role Receiver shall never be forced to discard a message. This is useful as the role Receiver

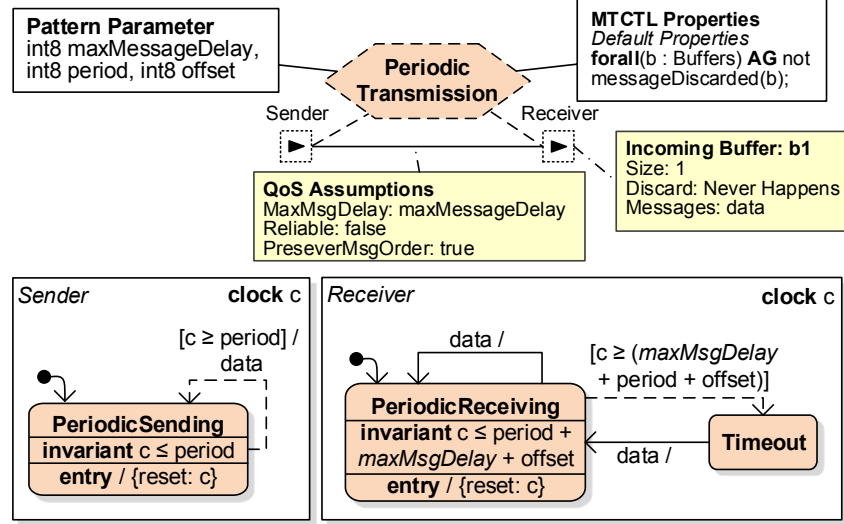


Figure B.5: Real-Time Coordination Pattern Periodic Transmission

is typically interested in the newest data. Moreover, the role connector has the following QoS assumptions: the connector may be unreliable, the maximum message delay is given as a parameter, and the message order shall be preserved (i.e., overtaken messages will be discarded).

The pattern defines three parameters:

- `maxMessageDelay` defines the maximum message delay.
- `period` defines the time span between two sendings of message data from the role Sender to the role Receiver.
- `offset` defines the additional time that the Receiver has to wait until it decides that the message will not arrive anymore.

Behavior The behavior is shown in the lower part of Figure B.5.

“The role Sender consists of the initial state `PeriodicSending` only. The Sender must send each period time units a message data to the receiver.

The role Receiver consists of the initial state `PeriodicReceiving` and the state `Timeout`. The [default behavior] is that the Receiver receives a message data periodically. Though, if the message data got lost or the Sender falls out, the receiver changes to state `Timeout` and activates a certain behavior to avoid the safety-critical situation. As soon as the receiver receives a message data again, it changes back to state `PeriodicReceiving`.” [*DBHT12]

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4). In addition, no buffer may discard a message: **forall**(`b : Buffers`) **AG not** `messageDiscarded(b)`;

A verified configuration such that the pattern fulfills all verification properties simultaneously is as follows:

- period: 20 ms
- maxMessageDelay: 5 ms
- offset: 1 ms

Consequences As a benefit, this pattern reduces the communication exchange to a minimum, i.e., the Receiver does not need to ask whether new data is available.

As a liability, both roles need to know the period. Moreover, the role Receiver also needs to know the maximum message delay.

Examples An autonomous vehicle drives in an unknown area. It uses its distance sensors to detect obstacles on its way. The measurements is exchanged periodically between two components of the vehicle: a component that gathers and sends the measurements and one that receives it and influences the movements of the vehicle based on the measurements. If the latter component does not receive its measurements in time, it has to stop the movement of the vehicle to avoid collisions.

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role Receiver to a multi role so that he can have multiple instances of role Sender that send data to the single instance of role Receiver. The receiving may be realized via the one-to-many communication schemata `singlereceive` or `multireceive`. (2) The pattern user can change the role Sender to a multi role so that he can have multiple instances of role Receiver that receive data from the single instance of role Sender. Hereby, it is possible that all instances of role Receiver shall receive the same data (this may be realized via the one-to-many communication schemata `multicast` or `iterate`) or every data is sent to only one Receiver (this may be realized via the one-to-many communication schemata `loadbalancing` or `unicast`).

If the pattern user adds conditions to the transitions of the Receiver that receive message data, then the consuming of the receiving message may be delayed. Therefore, a buffer size of one can lead to the fact that incoming messages are discarded. The pattern user either can define that the buffer shall discard the oldest message or the incoming message. If the pattern user does not want that messages are discarded, then he should check the following MTCTL property: `AG not messageDiscarded(b1)`; If this property fails, then the pattern user needs to increase the buffer size.

If the connector is reliable (i.e., message loss is not possible), then all stated verification properties are still useful. However, the pattern user should apply several changes to make the model more compact: (1) The role Receiver does not need the clock `c` anymore. (2) The state `Receiver.Timeout` and its transitions are not necessary anymore. (3) The invariant and clock reset at state `Receiver.PeriodicReceiving` is not necessary anymore. In addition, the following MTCTL property should be fulfilled: `AG not messageDiscarded(b1)`;

Alternative Patterns The pattern user should use the pattern Event-based Transmission (see Section B.4) if a periodic sending is not necessary as the data only needs to be sent if a certain event occurs (e.g., a condition becomes true). Then, the pattern user can reduce the exchanged messages to a minimum.

“Use the pattern Limit Observation (see Section B.8), if there is a pre-defined limit and the Sender informs event-based [whether] the limit is redeemed or violated.” [*DBHT12]

Combinability The pattern user can combine it with the pattern Acquire Assurance (see Section 6.3.5): As long as the assurance is active, the pattern may be used to request additional information.

Moreover, he can use the pattern Block Execution (see Section B.7) in order to block the Sender to send new data to the Receiver.

Search Terms transmit data, send information, receive data

B.6 Alternating Lock

Name Alternating Lock (initially defined by [Rie11]).

Also Known As It is also known as Producer Consumer, Mutual Exclusion, and Turn Transmission.

Overview This pattern is used when two roles may not execute their actions at the same time. The roles have to alternate their executions. In addition, both roles may only execute their actions for a defined length of time. Then, they have to pass the permission to execute the actions to the other role.

Context Prevent actions of another role

Problem Two roles shall not be able to execute a certain action together, e.g., accessing a shared section, to prevent safety-critical situations. Instead, they shall alternate the execution of their actions. In addition, the time each role should be allowed to execute the action shall be limited.

Solution Define an RTCP that specifies a bidirectional alternating lock between two roles: PartnerA and PartnerB. At first, one of the partners may have execute its actions but the other one shall not. Via message exchange, the partners may switch their turns, i.e., the permission to execute its actions. Furthermore, the pattern user should define an upper time limit that limits the time that a role may execute its actions. Otherwise, one role could execute its actions infinitely long, which is not intended.

Structure We show the structure of the pattern in the upper part of Figure B.6.

The pattern is bidirectional and consists of the two roles PartnerA and PartnerB. Both roles are in/out roles. Both, the role PartnerA and the role PartnerB may only send the message `switch`.

Both roles define a single buffer of size 1. In addition, both buffers shall never be forced to discard a message. Moreover, the role connector has the following QoS assumptions: the connector is reliable, the maximum message delay is given as a parameter, and the message order shall be preserved.

The pattern defines two parameters:

- `maxMessageDelay` defines the maximum message delay.

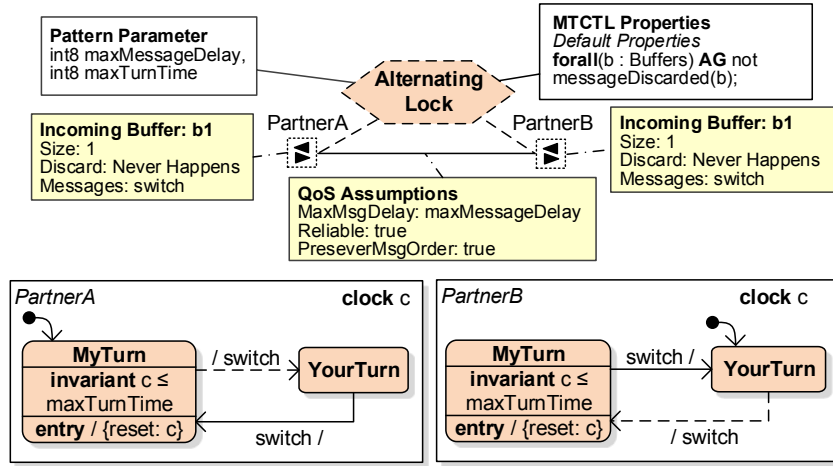


Figure B.6: Real-Time Coordination Pattern Alternating Lock

- maxTurnTime defines the time how long each turn may be.

Behavior We show the behavior of the pattern in the lower part of Figure B.6.

The role **PartnerA** is initially in state **MyTurn** and may execute its actions. The invariant of this state defines an upper time limit for the turn of this role (via the parameter maxTurnTime). Before this invariant expires, it has to end its actions, switch to state **YourTurn**, and send the message **switch** to the role **PartnerB**. If the role receives the message **switch** from role **PartnerB**, then it has to consume this message and return to state **MyTurn**. Then, it may execute its actions again.

The role **PartnerB** is initially in state **MyTurn** and may not execute its actions. If the role receives the message **switch** from role **PartnerA**, then it has to consume this message and return to state **MyTurn** and may execute its actions. The invariant of this state defines an upper time limit for the turn of this role (via the parameter maxTurnTime). Before this invariant expires, it has to end its actions, switch back to state **YourTurn**, and send the message **switch** to the role **PartnerA**.

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4). In addition, no buffer may discard a message: forall(b : Buffers) AG not messageDiscarded(b);

A verified configuration of the parameters such that the pattern fulfills all verification properties simultaneously is as follows:

- maxMessageDelay: 20 ms
- maxTurnTime: 50 ms

Consequences A benefit of the pattern is that it reduces the necessary message exchange, e.g., both roles do not need to request whether they may execute their actions.

One liability is that the maximum time of each turn must be pre-defined. Furthermore, it must be pre-defined which role may start with executing its actions. As another liability, the pattern user can only use this pattern if the message exchange is reliable.

Examples “Two components of the BeBot have access to a shared memory. The first components can write on the memory, the second can read. While writing, it is forbidden to read, while reading it is forbidden to write.” [*DBHT12]

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role PartnerB to a multi role so that he can have multiple instances of role PartnerA that all have may start their actions at start. Typically, the role PartnerB may only start to execute its safety-critical actions as soon as all instances of role PartnerA have sent the message `switch`. This may be detected via the one-to-many communication schemata `multireceive`. If role PartnerB leaves the state `MyTurn`, it should inform all instances of role PartnerA that they may execute their actions again, e.g., using the one-to-many communication schemata `multicast`. (2) The pattern user can change the role PartnerA to a multi role so that he can have multiple instances of role PartnerB. Then, all instances of role PartnerB may not execute their actions at the start. If role PartnerA leaves the state `MyTurn`, it should inform all instances of role PartnerB, e.g., using the one-to-many communication schemata `multicast`, that they may execute their actions. Eventually, all instances have to end their turn and send the message `switch` to role PartnerA. The role PartnerA may only start to execute its safety-critical actions as soon as it received a message `switch` from all instances of role PartnerB. This may be detected via the one-to-many communication schemata `multireceive`.

Instead of using the same state invariant, the two roles can use different state invariants.

Sander [San11] proposes a variant where it is not defined at the start which role may execute its actions. Thus, both roles start in a state `Inactive` and they have to negotiate which role may start.

Alternative Patterns “This pattern is related to the pattern Block Execution (see Section B.7), which provides also the possibility to lock actions [...]” [*DBHT12]

Combinability The pattern user can combine it with a one-to-many variant of pattern Fail-safe Delegation (see Section B.3) or Fail-operational Delegation (see Section B.2), where the role Slave is a multi role with a multiplicity of 2. Using Alternating Lock, the masters can avoid that they both send an order to role Slave.

Search Terms switch work, prevent actions, alternate access, alternate work

B.7 Block Execution

Name Block Execution (initially defined by [Dre11]).

Also Known As “Also known as Start-Stop and Guard.” [*DBHT12]

Overview This pattern is used when one role (the Guard) may block certain actions of another role (the Executor), e.g., due to safety-critical reasons. In particular, the Guard may at any time start and end the blocking.

Context Prevent actions of another role

Problem A role must be able to block and unblock the actions of another role, e.g., to prevent a safety-critical situation or if it is not necessary that the other role operates.

Solution “[...] [D]efine [an RTCP] between a Guard and an Executor. Enable the Guard to monitor the environment resp. the current situation. Only if acting is safe resp. necessary, the Guard grants permission to the Executor to act. At first, the permission is denied because the Guard first has to explore the situation.” [*DBHT12]

Structure We show the structure of the pattern in the upper part of Figure B.7.

The pattern is unidirectional and consists of the two roles Guard and Executor. The role Guard is an out-role and the role Executor is an in-role. The role Guard may send the messages free and block.

The role Executor defines a single buffer of size 1. In addition, the buffer shall never be forced to discard a message. Moreover, the role connector has the following QoS assumptions: the connector must be reliable, the maximum message delay is given as a parameter, and the message order shall be preserved.

The pattern defines two parameters:

- maxMessageDelay defines the maximum message delay.
- minSenderDelay defines a time span that the Guard has to wait before sending a message.

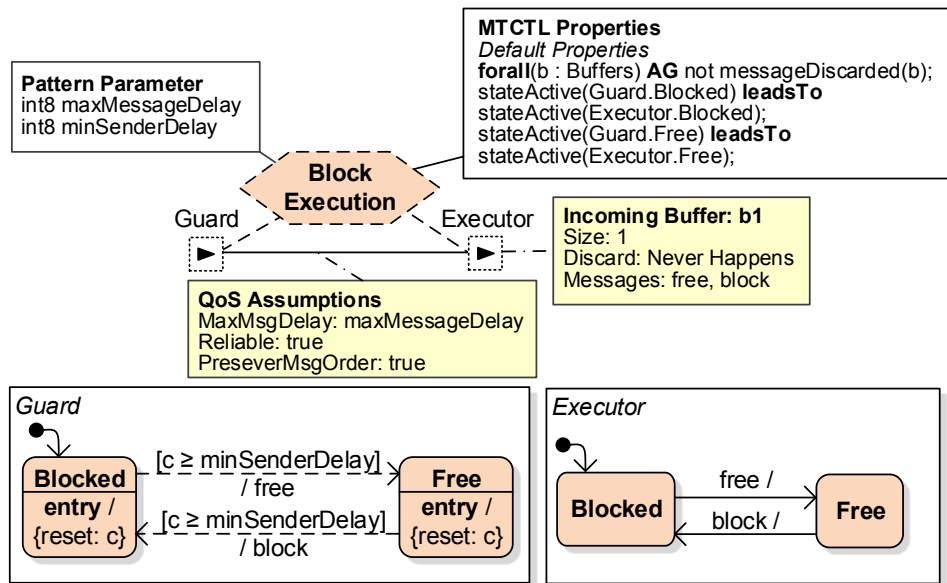


Figure B.7: Real-Time Coordination Pattern Block execution

Behavior “The behavior is shown in the lower part of Figure B.7

Initially, the role Guard consists of the initial state Blocked and the state Free. The guard sends the message free to the executor as soon as the executor may work and changes to state Free. As soon as the guard detects that the executor must stop his work,

it sends the message block and changes to state Blocked.” [*DBHT12] The role Guard has to wait a pre-defined minimum time before it may send a message. This minimum waiting time is necessary due to the semantics of RTSCs. Otherwise, the RTSC could send infinite messages without timing progress.

The role Executor consists of the initial state Blocked and the state Free. When the Executor receives the message free, it changes to state Free and starts its work. When the executor is in state Free and receives the message block, it changes to state Block and stops its work.” [*DBHT12]

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4). In addition, no buffer may discard a message: **forall**(*b* : Buffers) **AG not** messageDiscarded(*b*);

If any of the following properties is not fulfilled, then the role Executor does not stay in sync with the instructions of the role Guard.

- Whenever the state Guard.Blocked is active, then eventually the state Executor.Blocked is active as well: **stateActive**(Guard.Blocked) **leadsTo** **stateActive**(Executor.Blocked);
- Whenever the state Guard.Free is active, then eventually the state Executor.Free is active as well: **stateActive**(Guard.Free) **leadsTo** **stateActive**(Executor.Free);

A verified configuration of the parameters such that the pattern fulfills all verification properties simultaneously is as follows:

- maxMessageDelay: 50 ms
- minSenderDelay: 60 ms

Consequences A benefit of the pattern is that it reduces the necessary message exchange, e.g., the Executor does not need to request whether it may work on or not and the Guard does not need to send a status message periodically but only sends a message that the execution shall be stopped or that it may resume.

One liability is that the Guard has to know which actions the Executor executes. Furthermore, the pattern assumes that it does not take additional time to block and resume the actions of the Executor. As another liability, the pattern user can only use this pattern if the message exchange is reliable.

Examples In an extended version of our overtaking scenario (see Section 1.1), this pattern could be used for the communication between the red car and a satellite that has information about the safety of the street, e.g., whether glaze ice is on it. When using this pattern, the satellite would act as the Guard and the red car as the Executor. Then, if the street is unsafe, the satellite would be able to forbid the red car to overtake.

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role Executor to a multi role so that he can have multiple instances of role Guard that are allowed to block the actions of the role Executor. This may be realized via the one-to-many communication schemata *singlereceive* (if each instance of role Guard has the rights to block the actions) and *multireceive* (if all instances of role Guard must agree to block or free the actions). In

both cases, the instances of role Guard need to be informed about the changes of role Block Execution. (2) The pattern user can change the role Guard to a multi role so that he can have multiple instances of role Executor that receive a message from the Guard whether their execution is blocked or not. Typically, the one-to-many communication schemata multicast is appropriate for this in order to block or free the actions of all instances of role Executor.

It may be useful to switch the initial states such that the states Guard.Free and Executor.Free are initially active.

It may be useful to restrict the time that the states Blocked and Free of role Guard may be active.

If the blocking and resuming of the actions takes time, then the pattern user has to insert additional timing constraints at role Executor and increase the value of parameter minSenderDelay.

Alternative Patterns The patterns Fail-safe Delegation (see Section B.3) and Fail-operational Delegation (see Section B.2) both enable a Master to delegate a task to a Slave. In contrary, in the pattern Block Execution, the role Executor has a pre-defined task and does not need to report the result of the task to the role Guard. In particular, the role Guard even does not need to know which task the role Executor executes in detail.

Moreover, if the pattern user wants to block a certain action of another role but need to ask for permission first, then he can also use the pattern Acquire Assurance (see Section 6.3.5).

Combinability The pattern Block Execution can be combined with several patterns to block one of the acting roles. For example, the role Master of pattern Fail-safe Delegation (see Section B.3) may be blocked such that it can no longer delegate new tasks to a slave. Other examples are the role Master of pattern Fail-operational Delegation (see Section B.2) and the role Sender of pattern Event-based Transmission (see Section B.4).

Search Terms block execution, prevent actions, stop work, pause job

B.8 Limit Observation

Name Limit Observation (initially defined by [Dre11]).

Also Known As -

Overview This pattern is used if one role shall notify another role whether a certain measurement value violates a pre-defined limit or not.

Context Information exchange between two roles

Problem A role Provider measures numerical information. The other role Observer wants to be informed as soon as the numerical information violates a certain limit and as soon as the limit is redeemed again.

Solution The goal is to avoid as much communication as possible. Therefore, the role Provider does not periodically send the information whether the limit is violated or not. Moreover, the Observer does not need to request whether the limit is violated or redeemed. Instead, the Provider only sends a message as soon as the limit is violated and as soon as it is redeemed again. The pattern respects that at the start it is unknown to both roles whether the limit is violated. The limit may be a single value or defined by an upper and lower bound.

Structure We show the structure of the pattern in the upper part of Figure B.8.

The pattern consists of the roles Provider and Observer. The role Provider is an out-role; the role Observer is an in-role. The role Provider may send the messages `limitViolated` and `limitRedeemed`.

The role Observer has a single buffer of size 1. In addition, the buffer shall never be forced to discard a message. The role connector has the following QoS assumptions: the connector must be reliable, the maximum message delay is given as a parameter, and the message order shall be preserved.

The pattern defines three parameters:

- `maxMessageDelay` defines the maximum message delay.
- `worktime` defines the maximum working time for the first measurement of role Provider.
- `minSenderDelay` defines a time span that the Provider has to wait before sending a message (except the first message).

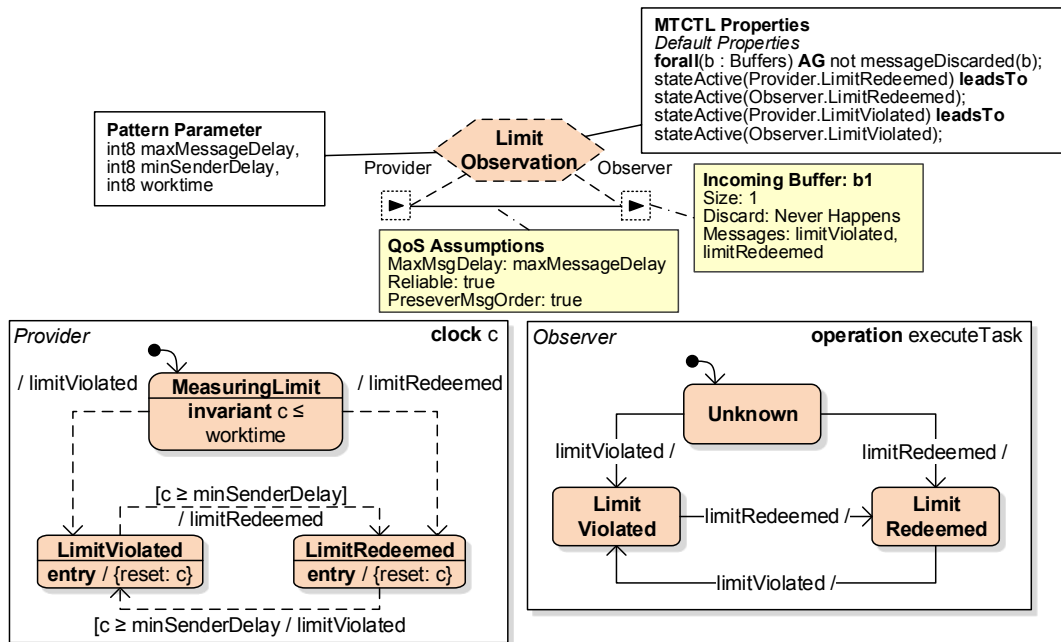


Figure B.8: Real-Time Coordination Pattern Limit Observation

Behavior “The behavior is shown in the lower part of Figure B.8.

Initially, the role Provider starts in state `MeasuringLimit` and stays there not longer than `[worktime]` units of time. In this state, the first measurement will be done and the provider checks [whether] the limit is redeemed or violated. If it is redeemed, the state changes to `LimitRedeemed` and the message `limitRedeemed` is sent to the observer. If the limit is violated, the state changes to `LimitViolated` and the message `limitViolated` is sent to the observer. If the provider is in state `LimitViolated` and [detects] that the result of the measurements changes so that the limit is not violated anymore, the provider changes to state `LimitRedeemed` and sends the message `limitRedeemed`. If the provider is in state `LimitRedeemed` and [detects] that the result of the measurements changes so that the limit is violated, the provider changes to state `LimitViolated` and sends the message `limitViolated`.” [*DBHT12] The role Provider has to wait a pre-defined minimum time before it may send a message (except before sending the first message). This minimum waiting time is necessary due to the semantics of RTSCs. Otherwise, the RTSC could send infinite messages without timing progress.

The role Observer is initially waiting in state `Unknown` for a message from the role Provider that informs whether the limit is violated or redeemed. If the message `limitViolated` arrives, the role will consume it and change to state `LimitViolated`. If the message `limitRedeemed` arrives, the role will consume it and change to state `LimitRedeemed`. Afterwards, the role Observer will switch between the states `LimitViolated` and `LimitRedeemed` when the corresponding message arrives.

Verification Properties The Real-Time Coordination Pattern has to ensure all default properties (see Section 4.4). In addition, no buffer may discard a message: **forall**(`b : Buffers`) **AG not** `messageDiscarded(b)`;

If any of the following properties is not fulfilled, then the role Observer does not stay in sync with the instructions of the role Provider.

- If the Provider is in state `LimitViolated`, the Observer will be in state `LimitRedeemed` in the future: `stateActive(provider.LimitViolated) leadsTo stateActive(observer.LimitRedeemed)`;
- If the Provider is in state `LimitRedeemed`, the Observer will be in state `LimitViolated` in the future: `stateActive(provider.LimitRedeemed) leadsTo stateActive(observer.LimitViolated)`;

A verified configuration of the parameters such that the pattern fulfills all verification properties simultaneously is as follows:

- `maxMessageDelay`: 1 ms
- `period`: 3 ms
- `minSenderDelay`: 4 ms

Consequences A benefit of the pattern is that it reduces the number of messages that shall be delivered to a minimum. For example, the Observer does not have to ask via polling whether the limit has changed and the Provider only sends a message if the limit is violated or not. Thus, the Provider does not send all measurements or sends a period information whether the limit is violated or not.

One liability is that the Observer is not informed which limit the Provider uses. Having the knowledge of the limit is not always mandatory for the Observer but it may be in some cases. Moreover, the Observer is not getting informed when the limit that the Provider uses changes. Furthermore, the Observer is not able to change the limit. In addition, the pattern user can only use this pattern if the message exchange is reliable.

Examples The pattern user can use this pattern to realize a distance control for a car platoon such that the upper and lower limit concerning the distance to a car is always respected. In particular, the role Provider represents a sensor that measures the distance between two cars and sends a message of the distance is too small or too high, i.e., the message `limitViolated` gets a parameter that defines whether the distance is too small or too high. In contrary, the role Observer represents a component that controls the speed of the car that drives behind based on the received information from the Provider. If the upper limit is violated, then the component that controls the speed has to decelerate. If the lower limit is violated, then the component that controls the speed has to accelerate.

Variants It is possible to extend the given one-to-one pattern to a one-to-many pattern in two different ways: (1) The pattern user can change the role Observer to a multi role so that he can have multiple instances of role Provider that inform the Observer concerning limit violations. This may be realized via the one-to-many communication schemata `singlereceive` or `multireceive`. (2) The pattern user can change the role Provider to a multi role so that he can have multiple instances of role Observer that receive a message from the Requester as soon as a limit is violated or redeemed. Hereby, it is possible that all instances of role Observer want to know each change (this may be realized via the one-to-many communication schemata `multicast` or `iterate`) or that each change should be sent to only one Observer (this may be realized via the one-to-many communication schemata `unicast` or `loadbalancing`).

Alternative Patterns A related pattern is Block Execution (see Section B.7), where the role Executor is blockable by the role Guard. Using this pattern, the sending role Guard decides whether the Executor should be blocked. However, using the pattern Limit Observation, the receiving role Observer can decide which actions should be done, if the limit exceeds. Moreover, the Observer may also execute actions if the limit is redeemed.

A more abstract version of this pattern is Event-based Transmission (see Section B.4) as it also sends its data event-based. However, using pattern Event-based Transmission, the data being sent is not restricted to inform about the violation of a limit.

Combinability “Combine it with Fail-Operational-Delegation (see Section B.2), if [the pattern user wants] that the Provider can change the limit at run-time.” [*DBHT12]

Using the pattern Block Execution (see Section B.7), the pattern user can instruct the role Provider to pause and continue the observation.

Using the pattern Event-based Transmission (see Section B.4), the role Provider can inform the Observer when the limit changes.

Search Terms observe limit, receive data, send data, deliver information



EVALUATION DETAILS

In this appendix, we present details to our evaluations. At first, we introduce in Section C.1 our examples that we use for our evaluation and that we did not already introduce in the main chapters like the RTCP Overtaking. Then, we present the evaluation details of MTCTL in Section C.2. Finally, in Section C.3, we present the details of our evaluation of our DSMC.

C.1 Examples

In this section, we present RTCPs that we use in our evaluation and that we did not already present within this work. In particular, we briefly explain their task and show their structural and behavioral definition.

Our RTCPs are taken from four different systems from the domain of CPS: (1) Overtaking Cars, (2) Overtaking Cars (Extended), (3) RailCab, and (4) BeBot. *Overtaking Cars* is the running example from this thesis. One car wants overtake another car. Optionally, a car from the contraflow is approaching as well. For this system, we use two RTCPs for our evaluation that we both already presented within the thesis: the RTCP Overtaking, which we depict in Figure 3.6 on page 56, and the RTCP MultiOvertaking, which we depict in Figure 3.9 on page 64. We introduce the other three systems and their RTCPs in the following.

C.1.1 Overtaking Cars (Extended)

This is an extended version of the overtaking scenario, where not just the cars but also a street information provider is involved. Furthermore, the car component is separated into several subcomponents that have to communicate via message as well. As a result, there exist three RTCPs for the internal communication between the subcomponents and four RTCPs for the communication between the cars and between the car and the street information provider. All RTCPs were created by the project group Cybertron [BCD+14] independently of our DSMC concept and the evaluations of this thesis.

C.1.1.1 RTCP Allow

The RTCP Allow consists of the two roles Requestor and Controller. As soon as it is allowed, the Requestor sends a request to the Controller whether overtaking is safe or unsafe. The Controller immediately answers this request.

The structure of the RTCP is shown in Figure C.1 while the role behaviors are shown in the Figures C.2 and C.3. The depicted figures are not published. Hence, we created them based on the given models.

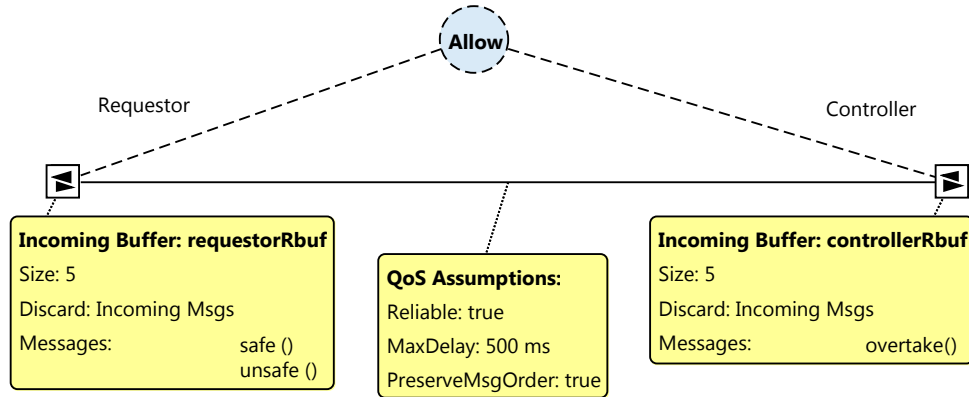


Figure C.1: Structure of RTCP Allow

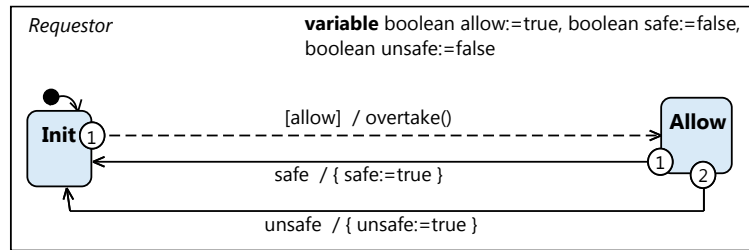


Figure C.2: Role Requestor of RTCP Allow

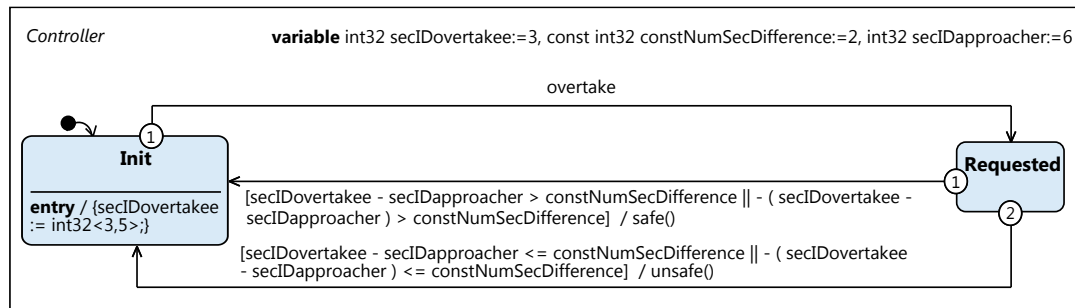


Figure C.3: Role Controller of RTCP Allow

C.1.1.2 RTCP ChangeSection

The RTCP ChangeSection consists of the two roles Vehicle and Section. In this RTCP, the Vehicle is able to detect if it changes its section (by detecting a change of the color on the street). If this happens, it informs the role Section that the section has been changed.

The structure of the RTCP is shown in Figure C.4 while the role behaviors are shown in the Figures C.5 and C.6. The depicted figures are not published. Hence, we created them based on the given models.

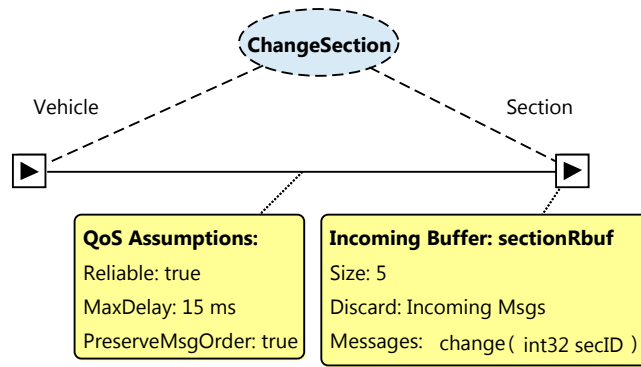


Figure C.4: Structure of RTCP ChangeSection

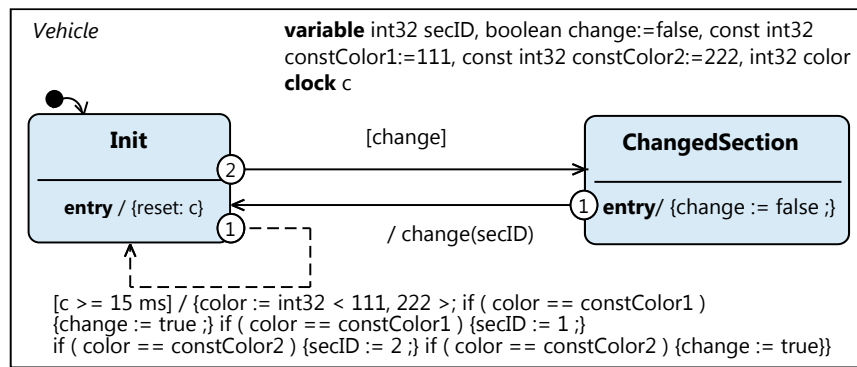


Figure C.5: Role Vehicle of RTCP ChangeSection

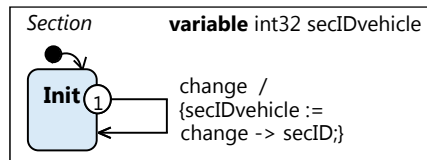


Figure C.6: Role Section of RTCP ChangeSection

C.1.1.3 RTCP Delegate

The RTCP Delegate consists of the two roles Initiator and Examiner. The task of the Initiator is to define when the overtaking shall start (e.g., based on the distance to the car that shall be overtaken). Then, the Initiator delegates the task of deciding whether overtaking is safe to the role Examiner. The Examiner will immediately start with the check. The Examiner does not answer as long as the overtaking is unsafe. As soon as overtaking is safe, the Examiner answers the Initiator that it may start the overtaking. Finally, the Initiator informs the Examiner when the overtaking is done.

The structure of the RTCP is shown in Figure C.7 while the role behaviors are shown in the Figures C.8 and C.9. The depicted figures are not published. Hence, we created them based on the given models.

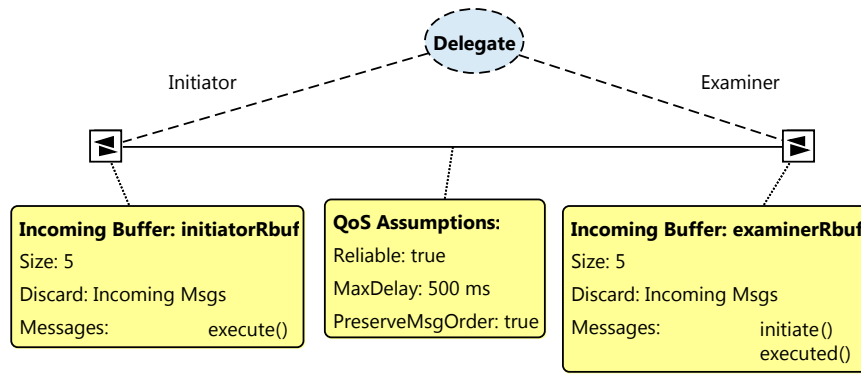


Figure C.7: Structure of RTCP Delegate

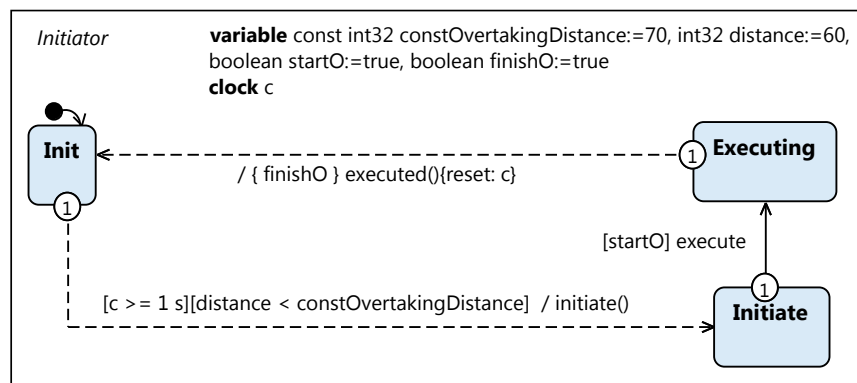


Figure C.8: Role Initiator of RTCP Delegate

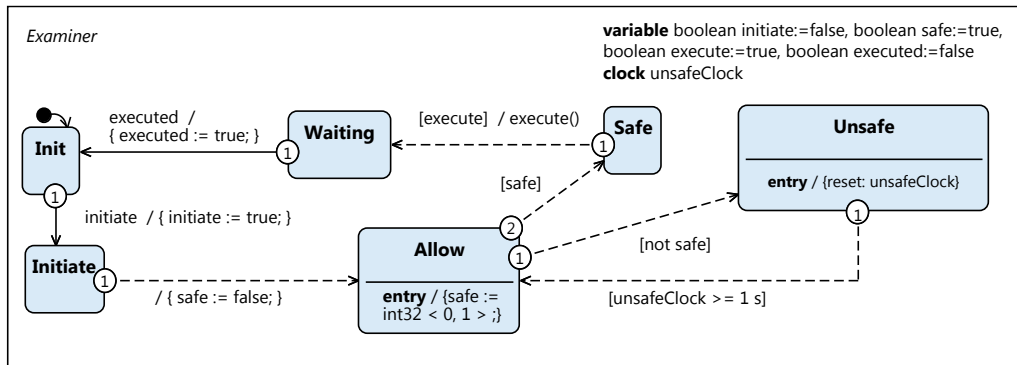


Figure C.9: Role Examiner of RTCP Delegate

C.1.1.4 RTCP Inform

The RTCP Inform consists of the two roles Overtakee and Approacher. The task of the Overtakee is to inform the Approacher when the overtaking starts and when it ends. As long as the overtaking happens, the Approacher assures to limit its acceleration.

The structure of the RTCP is shown in Figure C.10 while the role behaviors are shown in the Figures C.11 and C.12. The depicted figures are not published. Hence, we created them based on the given models.

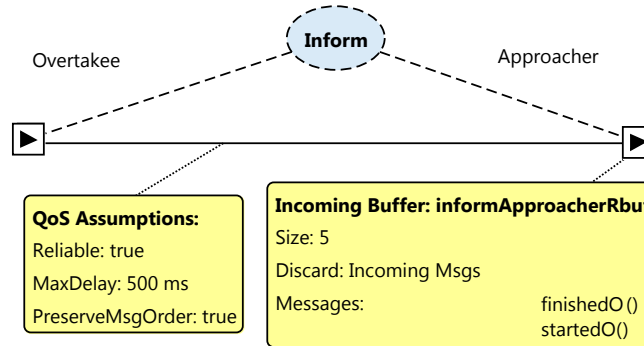


Figure C.10: Structure of RTCP Inform

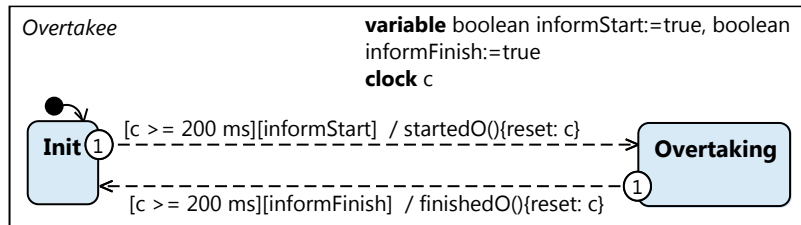


Figure C.11: Role Overtakee of RTCP Inform

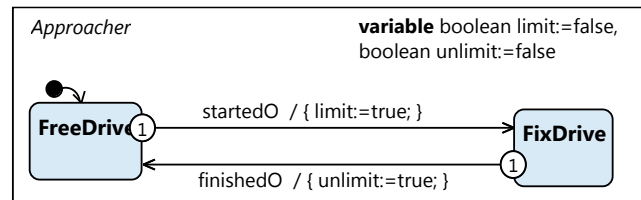


Figure C.12: Role Approacher of RTCP Inform

C.1.1.5 RTCP LimitVelocity

The RTCP LimitVelocity consists of the two roles Limiter and LimitDriving. The task of role LimitDriving is to limit the velocity of the role Limiter as long as the overtaking happens. In particular, LimitDriving sends the message fixVelocity if the role Limiter shall fix its velocity to a constant value as long as the message freeVelocity arrives.

The structure of the RTCP is shown in Figure C.13 while the role behaviors are shown in the Figures C.14 and C.15. The depicted figures are not published. Hence, we created them based on the given models.

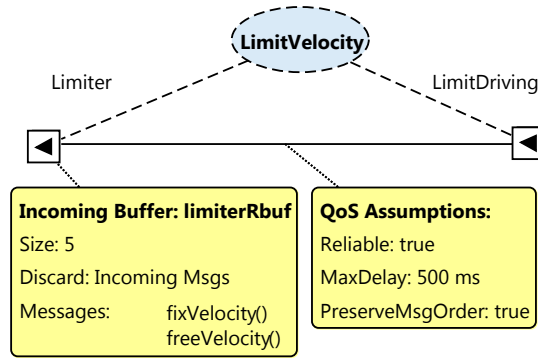


Figure C.13: Structure of RTCP LimitVelocity

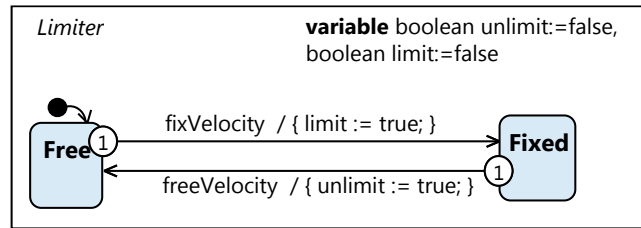


Figure C.14: Role Limiter of RTCP LimitVelocity

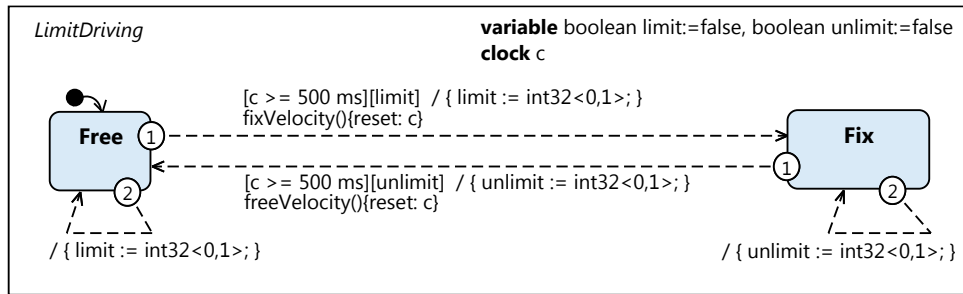


Figure C.15: Role LimitDriving of RTCP LimitVelocity

C.1.1.6 RTCP Overtake

The RTCP Overtake consists of the two roles Overtakee and Overtaker. The purpose of this protocol is to coordinate the start and the end of the overtaking. Therefore, this RTCP is similar to our running example RTCP Overtaking (see Figure 3.6 on page 3.6). In particular, the task of the Overtaker is to request whether overtaking is possible. If it is possible, then the Overtaker activates the overtaking. If it is not possible, then the Overtaker will wait at least one second until it requests again. As soon as the overtaking is finished, the Overtaker informs the Overtakee about this.

The structure of the RTCP is shown in Figure C.16 while the role behaviors are shown in the Figures C.17 and C.18. The depicted figures are not published. Hence, we created them based on the given models.

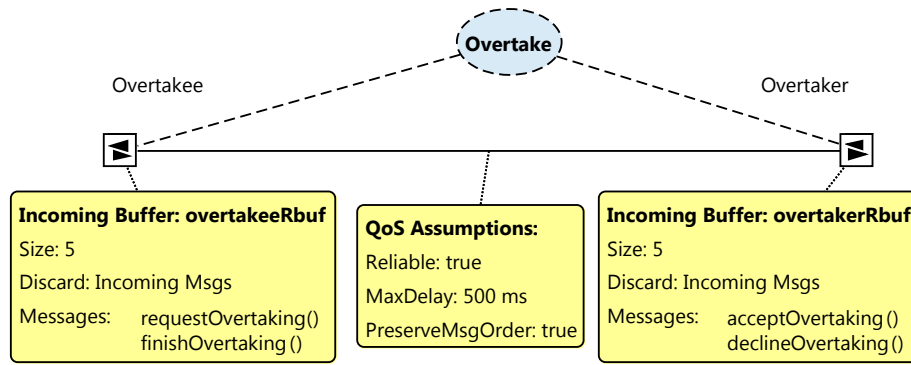


Figure C.16: Structure of RTCP Overtake

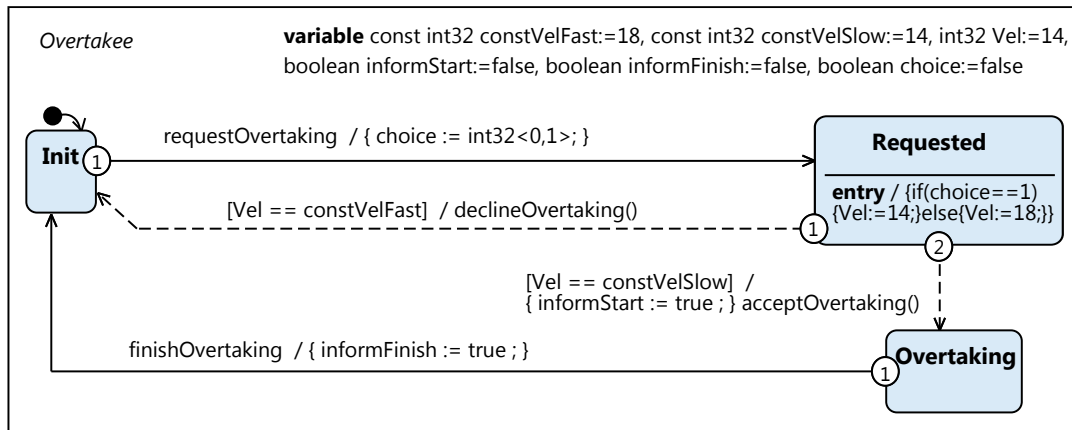


Figure C.17: Role Overtakee of RTCP Overtake

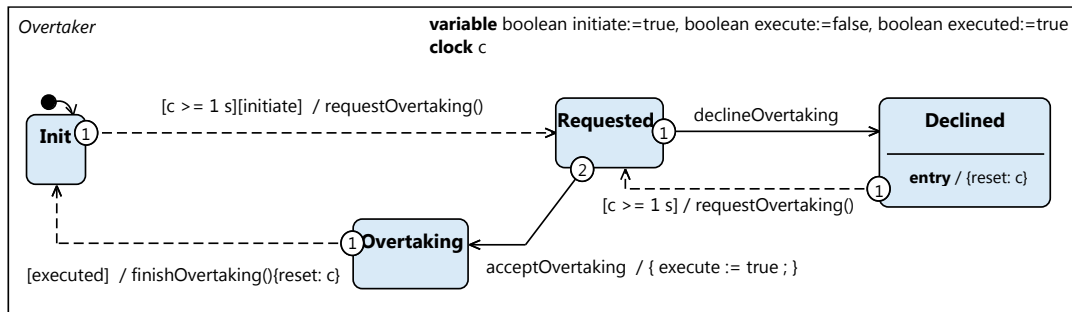


Figure C.18: Role Overtaker of RTCP Overtake

C.1.1.7 RTCP VModeControl

The RTCP VModeControl consists of the two roles VelocitySetter and VelocityGetter. The purpose of this RTCP is to coordinate the velocity of the car. In particular, the role VelocitySetter is able to switch between a mode where the velocity may change (between slow and fast) and where the velocity is fixed (i.e., it is always slow). If the VelocitySetter switches its mode or if it switches between slow and fast, then it informs the role VelocityGetter about this.

The structure of the RTCP is shown in Figure C.19 while the role behaviors are shown in the Figures C.20 and C.21. The depicted figures are not published. Hence, we created them based on the given models.

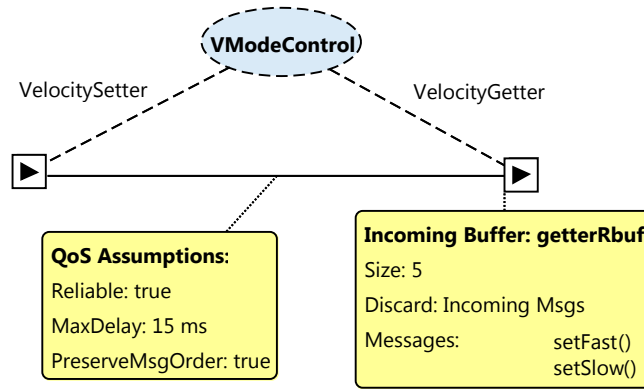


Figure C.19: Structure of RTCP VModeControl

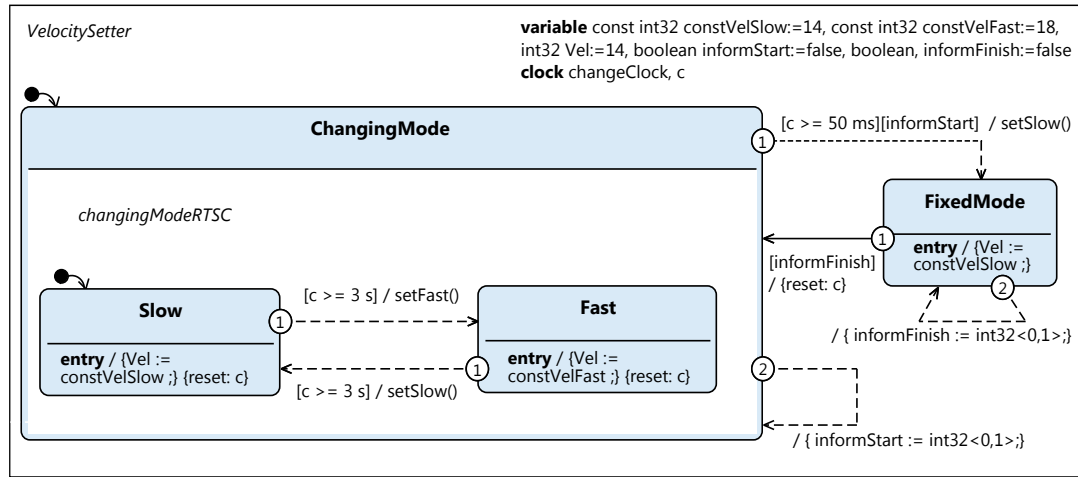


Figure C.20: Role VelocitySetter of RTCP VModeControl

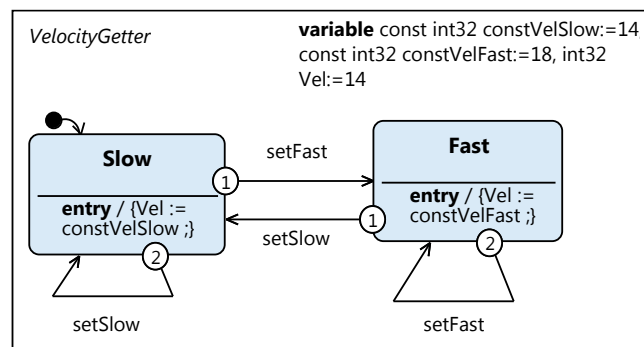


Figure C.21: Role VelocityGetter of RTCP VModeControl

C.1.2 RailCab

A RailCab [HTS+08] is an autonomous railway system that consists of only one cab. RailCab may build convoys dynamically with other RailCabs. As a consequence, the rear driving RailCab can save fuel as it drives within the slipstream of the other RailCab. RailCabs can communicate wirelessly for building such a convoy. Moreover, we assume that the tracks are separated into track sections, e.g., a switch. Track sections may communicate wirelessly via RailCabs, e.g., the RailCab needs permission for driving into the switch.

For our evaluation, we use four RTCPs from the RailCab system: EnterSection, NextSectionFree, ProfileDistribution, and ConvoyCoordination. EnterSection and NextSectionFree are both taken from [*HBDS15] and were designed by Christian Heinzemann and me. ProfileDistribution and ConvoyCoordination are taken from [Hei15] and were only designed by Christian Heinzemann. Note that ConvoyCoordination by Heinzemann is a variant of the RTCP that we depict in Figure 6.7. The variant by Heinzemann can coordinate a convoy of two or more RailCabs while the RTCP in Figure 6.7 can only coordinate a convoy of two RailCabs. All RTCPs have been designed independently of our DSMC concept and the evaluations of this thesis.

C.1.2.1 RTCP EnterSection

The RTCP EnterSection consists of the two roles Railcab and Section. “The purpose of this protocol is to enable a coordination for traversing a section in a safe and efficient manner.” [*HBDS15]. In particular, the role Railcab has to request for entering the section, which is represented by role Section, as soon as it approaches it. The role Section may allow and deny the entering. If it is denied, then the Railcab has to wait until the Section allows the entering. If the entering is allowed, then the Railcab will eventually enter the section, drive on it, and leave it while informing the Section about this.

The structure of the RTCP is shown in Figure C.22 while the role behaviors are shown in the Figures C.23 and C.24. We created the figures based on the given models. The RTCP is also published in [*HBDS15].

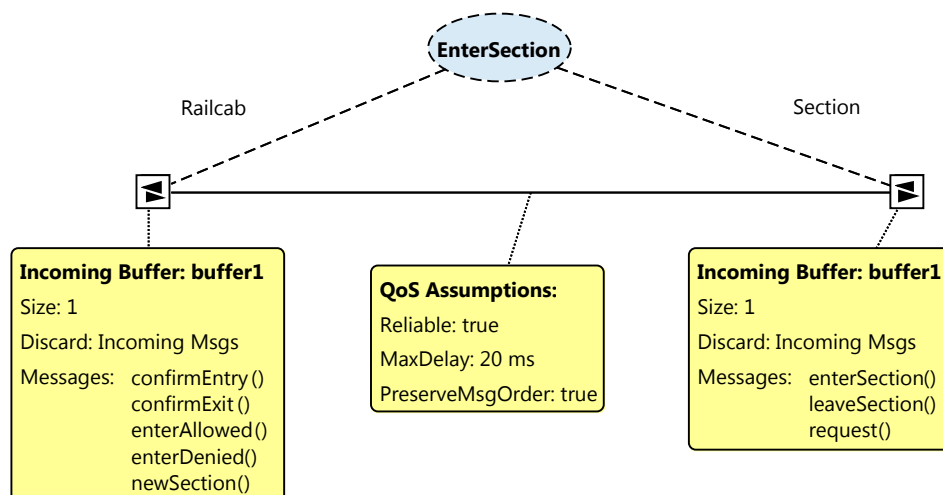


Figure C.22: Structure of RTCP EnterSection (Based on [*HBDS15])

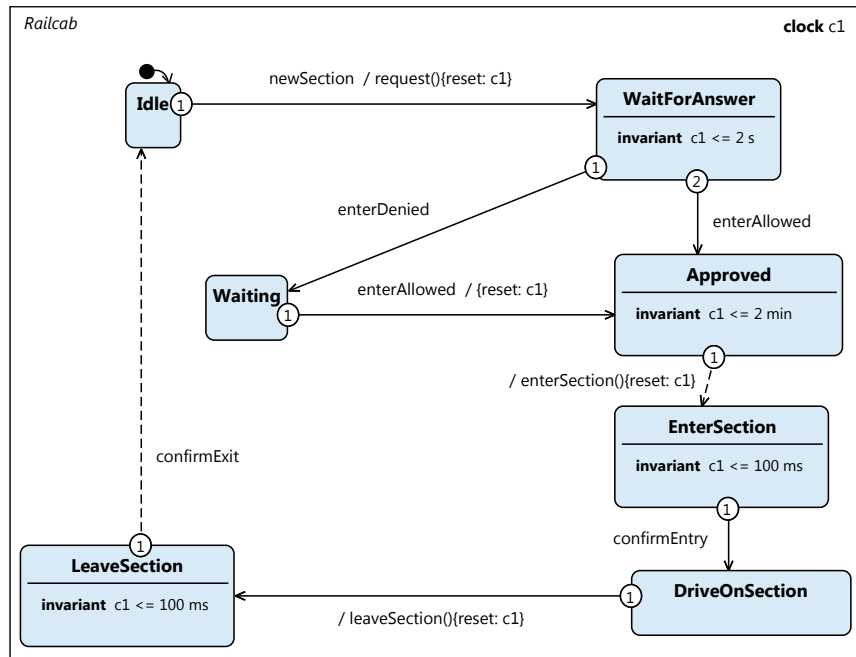


Figure C.23: Role RailCab of RTCP EnterSection (Based on [*HBDS15])

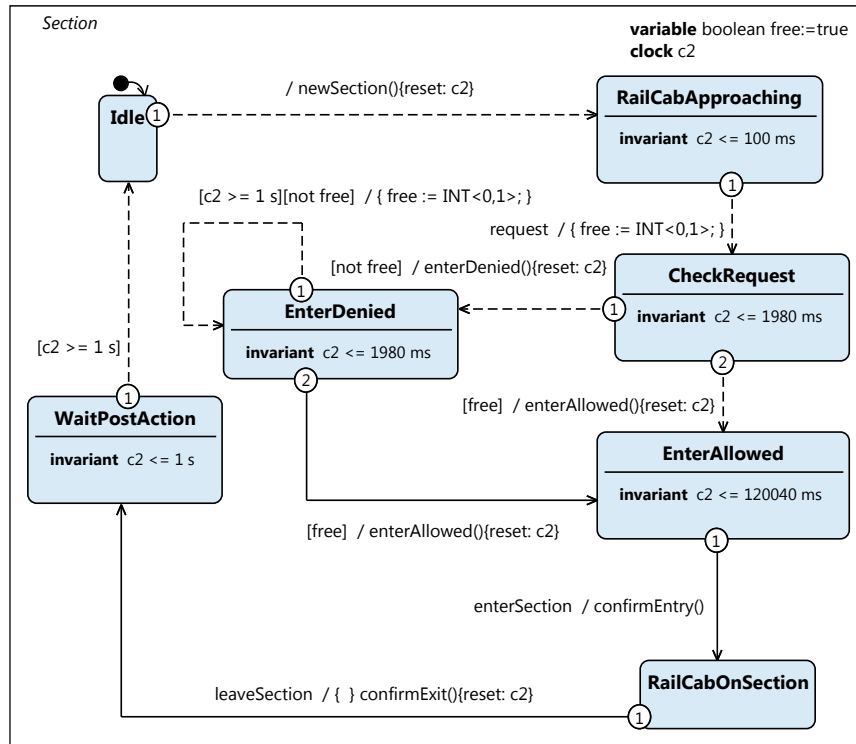


Figure C.24: Role Section of RTCP EnterSection (Based on [*HBDS15])

C.1.2.2 RTCP NextSectionFree

The RTCP NextSectionFree consists of the two roles Tracksection and Switch. The purpose of this RTCP is to acquire the knowledge if the next section is free. In particular, the role Tracksection sends a request to role Switch (a switch is a special kind of a track section), whether it is currently free or not. If the Switch receives the request, then it will answer after 20 ms.

The structure of the RTCP is shown in Figure C.25 while the role behaviors are shown in the Figures C.26 and C.27. The depicted figures are not published. Hence, we created them based on the given models.

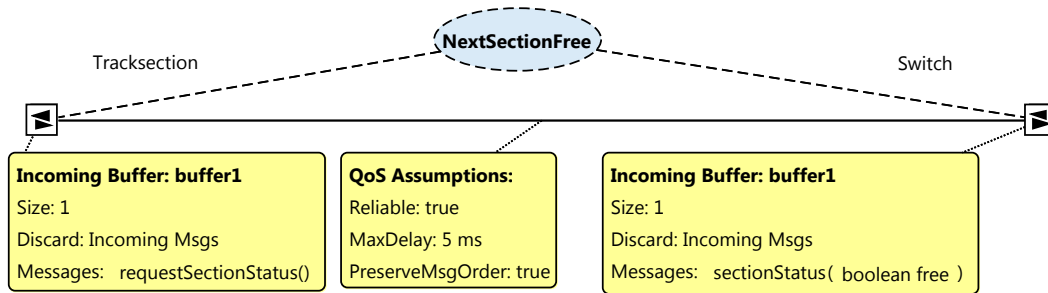


Figure C.25: Structure of RTCP NextSectionFree

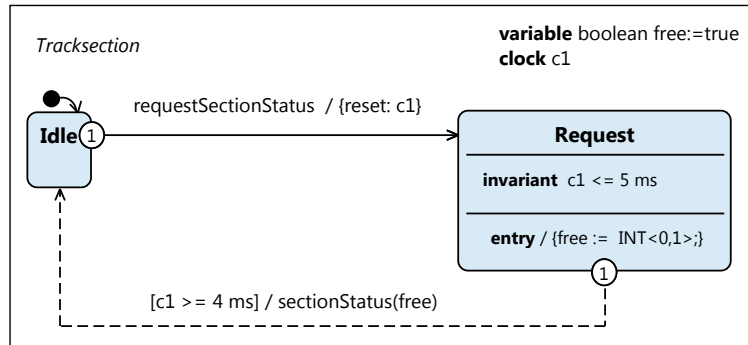


Figure C.26: Role Tracksection of RTCP NextSectionFree

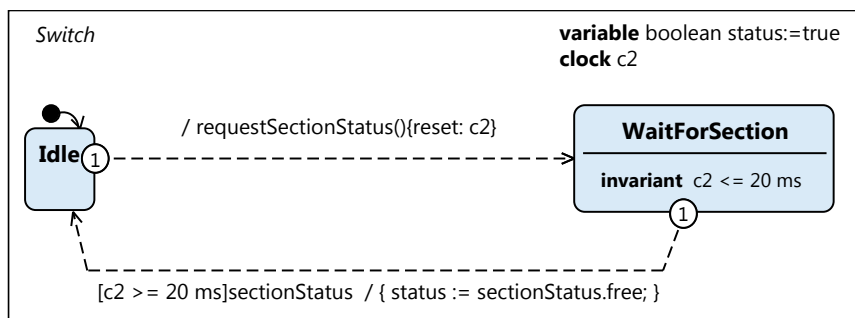


Figure C.27: Role Switch of RTCP NextSectionFree

C.1.2.3 RTCP ProfileDistribution

“The RTCP ProfileDistribution [...] is responsible for propagating profiles and the data, which is necessary for using the profile, inside the coordinator RailCab. [...] The multi role [ProfileProvider] sends the profile information to many [instances of role ProfileReceiver] and receives information about the current maximum speeds for the [existing instances of role ProfileReceiver]. The latter information may be used for adjusting the convoy speed after a profile change.” [Hei15]

The structure of the RTCP is shown in Figure C.28 while the role behaviors are shown in the Figures C.29 and C.30.

Compared to us, Heinzemann uses a slightly different concrete syntax as he indicates an urgent state by the letter U in the top right corner of the state. Moreover, he assumes that messages are never reordered during transmission and that they never get discarded while enqueueing into a buffer. Though, he does not state for the RTCP whether the connector is reliable. Furthermore, Heinzemann names the two regions of a multi role RTSC adaptation and subrole instead of coordinator and subrole. In order to be compliant with the RTSC definition, we extracted the clock resets of the action compartment. This thesis does not focus on reconfiguring RTCPs. Therefore, we removed the action `createSubRoleInstance(self)`; at the self-transition of state `adaptation.Idle`. In addition, we removed the entry effect of state `ProfileProvider.adaptation.Idle` that resets clock `c5` and add a clock reset of clock `c5` to transition `ProfileProvider.adaptation.sendUpdate`. Without this change, the RTSC would not comply to the informal definition of Heinzemann that the role sends every second an update as the self-transition of state `adaptation.Idle` could delay the update.

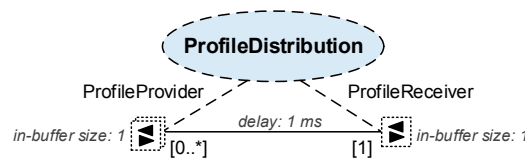


Figure C.28: Structure of RTCP ProfileDistribution (Based on [Hei15])

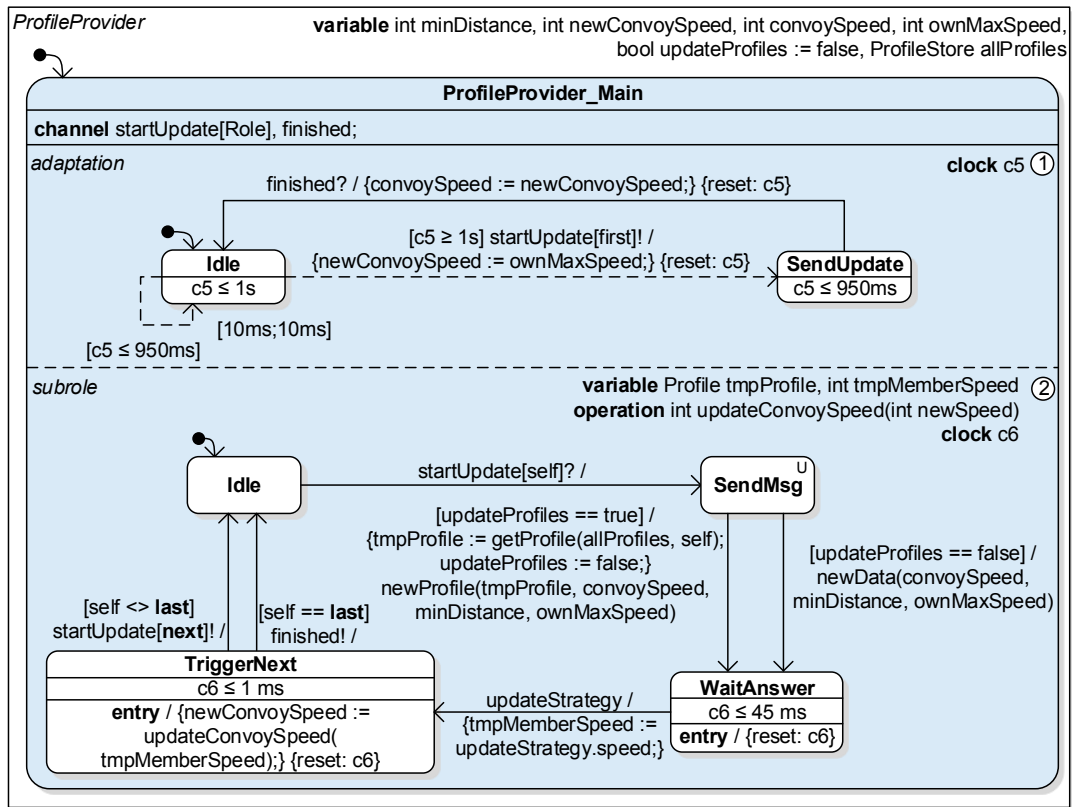


Figure C.29: Changed Role ProfileProvider of RTCP ProfileDistribution (Based on [Hei15])

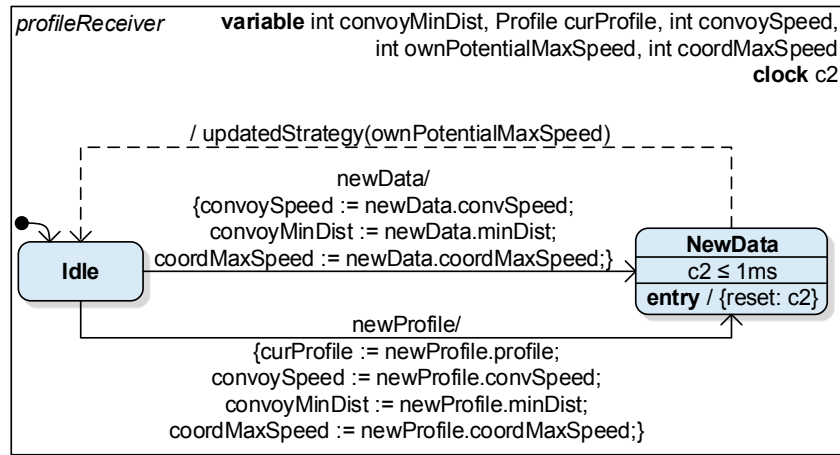


Figure C.30: Role ProfileReceiver of RTCP ProfileDistribution (Based on [Hei15])

C.1.2.4 RTCP ConvoyCoordination (2 or more RailCabs)

“The RTCP ConvoyCoordination [...] is responsible for managing the convoy [of two or more RailCabs]. 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 [...] [Hei15]

“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.” [Hei15]

As the description indicates, this RTCP is able to reconfigure the number of roles at runtime. The concepts for specifying reconfiguration rules in RTCPs are specified in [Hei15].

The structure of the RTCP is shown in Figure C.31 while the role behaviors are shown in the Figures C.32 and C.33.

Compared to us, Heinzemann uses a slightly different concrete syntax as he indicates an urgent state by the letter U in the top right corner of the state. Moreover, he assumes that messages are never reordered during transmission and that they never get discarded while enqueueing into a buffer. Though, he does not state for the RTCP if the connector is reliable. Furthermore, Heinzemann names the two regions of a multi role RTSC adaptation and subrole instead of coordinator and subrole. In order to be compliant with the RTSC definition we extracted the clock resets of the action compartment. This thesis does not focus on reconfiguring RTCPs. Therefore, we removed the action `createSubRoleInstance(self);` at the transition between state `Coordinator.adaptation.Idle` and `HandleNewMember` as well as the action `deleteSubRoleInstance(curSubRole);` at the transition between state `Coordinator.adaptation.DeleteSR` and `Idle`. Furthermore, we fixed the following definitions as they would lead to a deadlock within the original RTSC:

- We add a clock reset of clock `c1` at the two transitions leading to state `Coordinator.adaptation.DeleteSR`.
- We add a clock reset of clock `c1` at the transition from `Coordinator.adaptation.UpdateRequired` to `UpdateProfiles`.
- We changed the clock constraint at the transition from state `Coordinator.adaptation.DeleteSR` to `Idle`: Instead of `c2 >= 200ms` we define that `c1 >= 200ms` must hold.

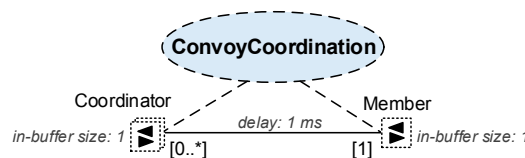


Figure C.31: Structure of RTCP ConvoyCoordination (Based on [Hei15])

327

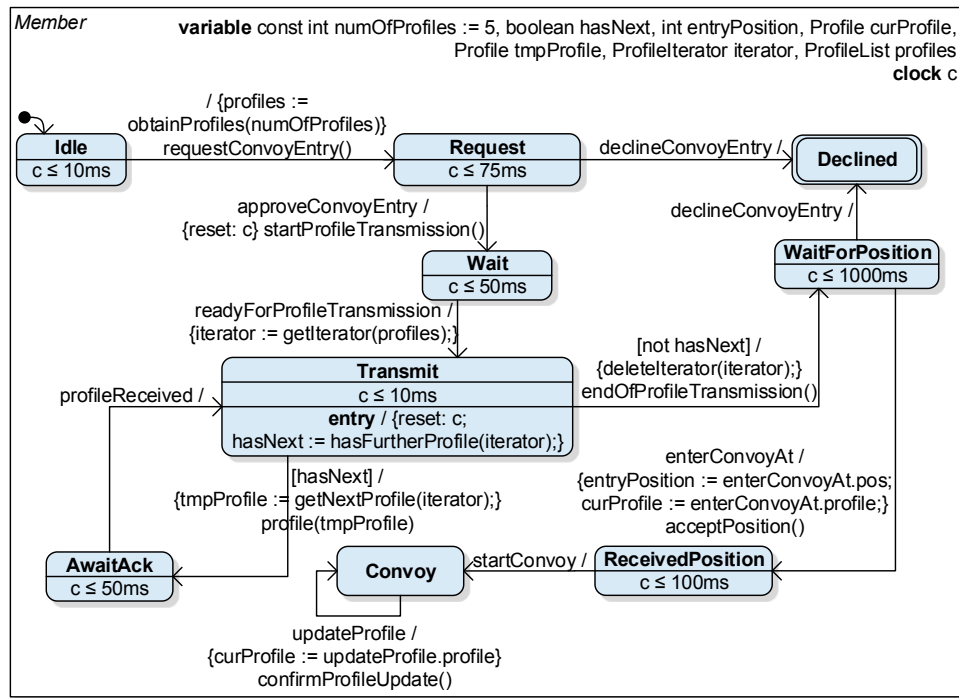


Figure C.33: Role Member of RTCP ProfileDistribution (Based on [Hei15])

C.1.3 BeBot

The BeBot [HWR09; BeBot] is a small autonomous robot that uses sensors to perceive its environment. In our scenario, BeBots use RTCPs within their system in order to separate the internal behavior. Moreover, they use RTCPs to communicate with other BeBots, e.g., for collision avoidance.

For our evaluation, we use five RTCPs from the RailCab system: Navigation, Delegation, Distribution, PositionTransmission, and AllPositionsTransmission. All RTCPs have been initially designed by our student Jannis Drewello [Dre11]. Later on, they have been refined by students, colleagues, and myself as they are the running example of the MECHATRONICUML PIM specification [*BDG+11; *BBD+12; *BBB+12; *BDG+14b; *DPP+16]. In this thesis, we use the newest variant from [*DPP+16]. Noteworthy, all RTCPs have been designed independently of our DSMC concept and the evaluations of this thesis.

C.1.3.1 RTCP Navigation

The RTCP Navigation consists of the two roles Navigator and Provider. The task of the role Navigator is to inform the Provider that it shall move to a certain target position. As soon as the position is reached, the Provider informs the Navigator about this.

The structure of the RTCP is shown in Figure C.34 while the role behaviors are shown in the Figures C.35 and C.36.

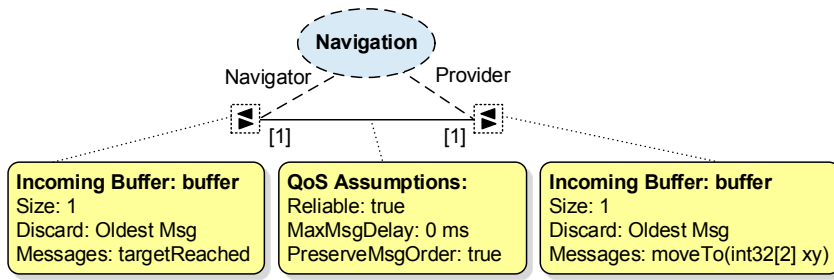


Figure C.34: Structure of RTCP Navigation [*DPP+16]

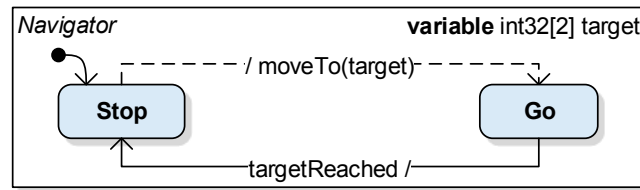


Figure C.35: Role Navigator of RTCP Navigation [*DPP+16]

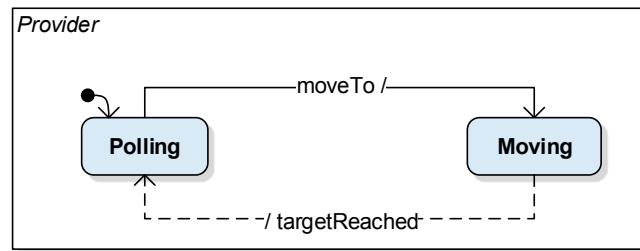


Figure C.36: Role Provider of RTCP Navigation [*DPP+16]

C.1.3.2 RTCP Delegation

The RTCP Delegation consists of the two roles Master and Slave. In this RTCP, the Master delegates the task to check whether a target position is acceptable or not to the Slave. The Slave informs within 100 ms about the result of this check.

The structure of the RTCP is shown in Figure C.37 while the role behaviors are shown in the Figures C.38 and C.39.

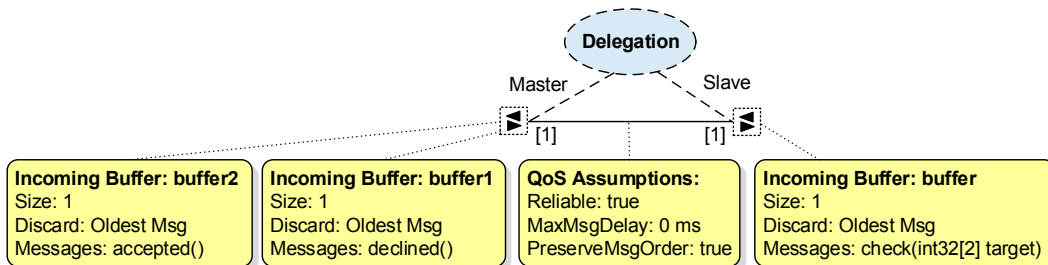


Figure C.37: Structure of RTCP Delegation [*DPP+16]

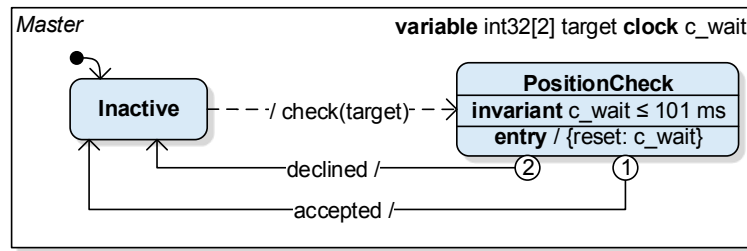


Figure C.38: Role Master of RTCP Delegation [*DPP+16]

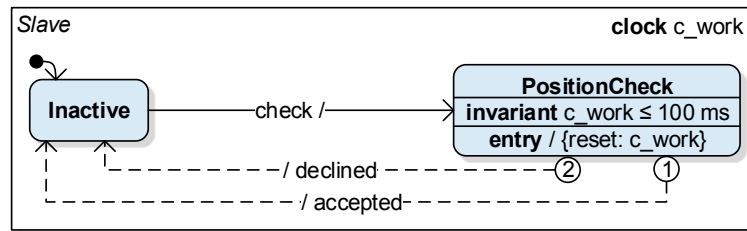


Figure C.39: Role Slave of RTCP Delegation [*DPP+16]

C.1.3.3 RTCP Distribution

The RTCP Distribution consists of the two roles Distributor and Client. The purpose of this RTCP is to distribute the position of all BeBots. In particular, the Distributor collects from each Client its position. Then, it combines these positions and its own position into a list of positions. This list is sent to each Client. Compared to [*DPP+16], we made a few changes to the RTCP as we identified several problems. For example, we added a state Blocked to the region send of role Client such that all clients send their position first to the role Distributor before the distributor sends all positions to all clients.

The structure of the RTCP is shown in Figure C.40 while the role behaviors are shown in the Figures C.41 and C.42.

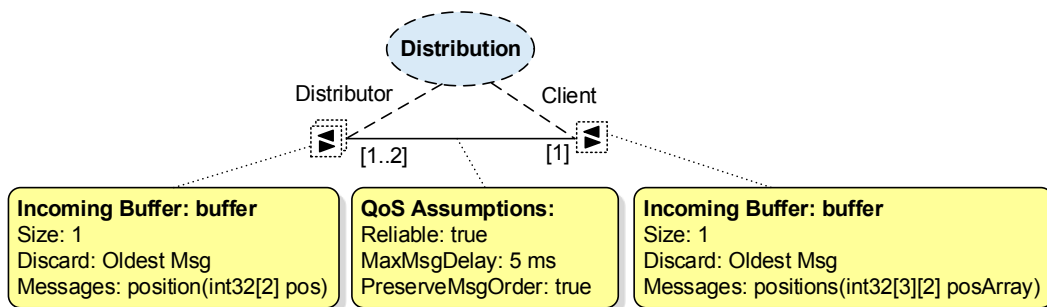


Figure C.40: Structure of RTCP Distribution [*DPP+16]

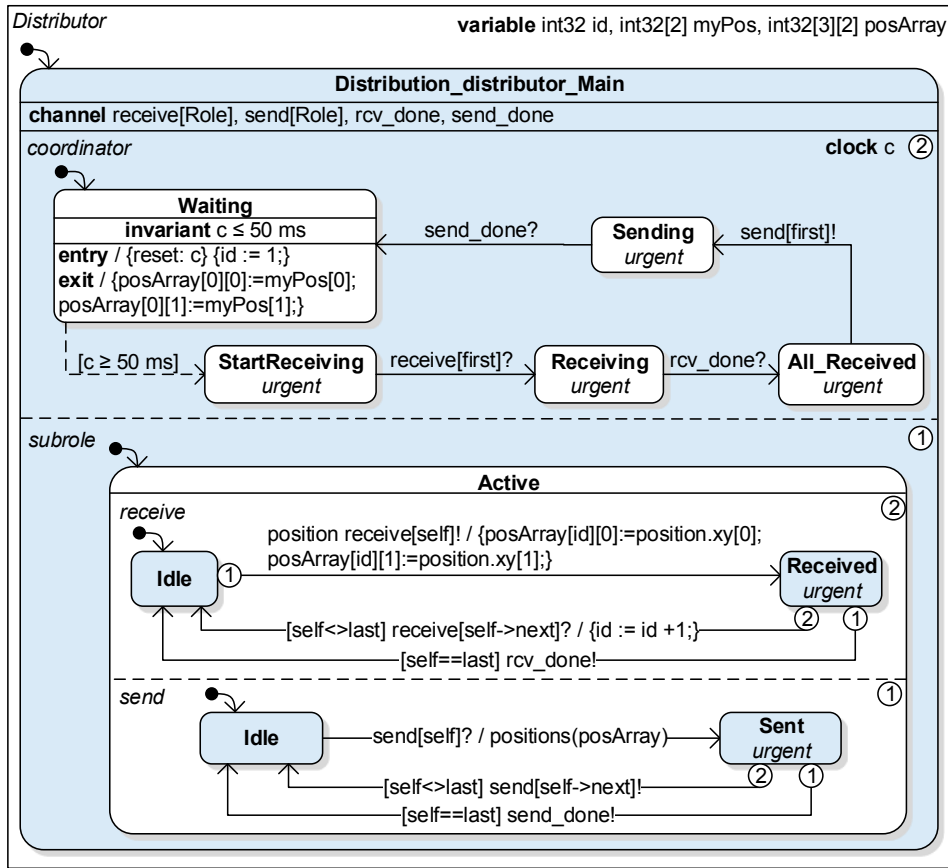


Figure C.41: Role Distributor of RTCP Distribution (Based on [*DPP+16])

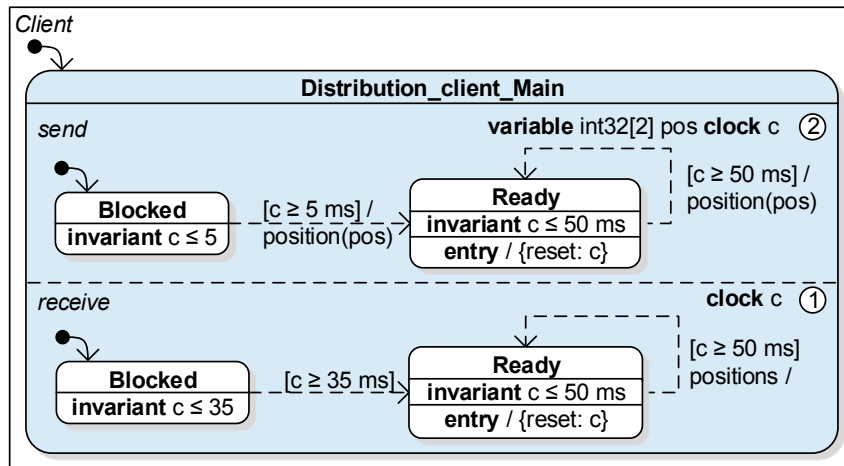


Figure C.42: Role Client of RTCP Distribution (Based on [*DPP+16])

C.1.3.4 RTCP PositionTransmission

The RTCP PositionTransmission consists of the two roles Sender and Receiver. In this RTCP, the Sender multicasts a position of a BeBot to all instances of role Receiver. In particular, the position is sent periodically each 30 ms.

The structure of the RTCP is shown in Figure C.43 while the role behaviors are shown in the Figures C.44 and C.45.

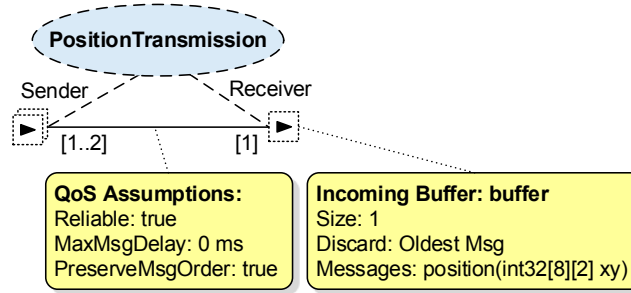


Figure C.43: Structure of RTCP PositionTransmission [*DPP+16]

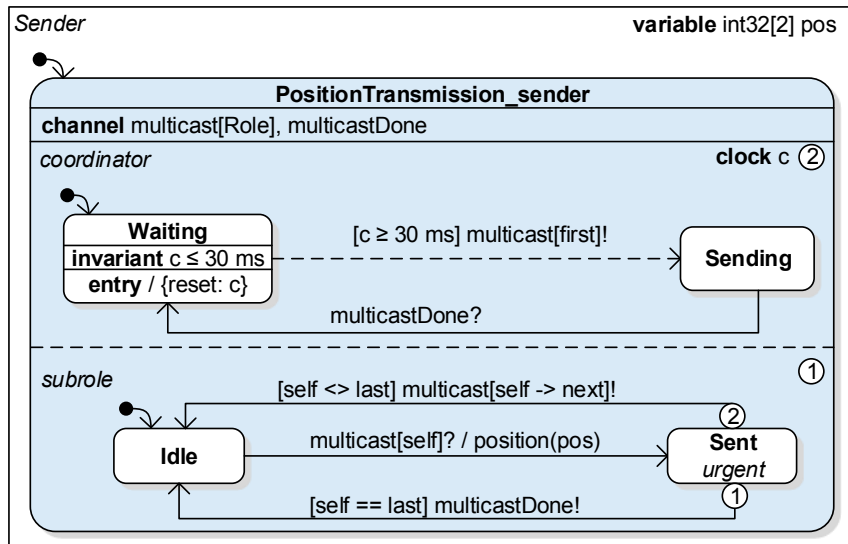


Figure C.44: Role Sender of RTCP PositionTransmission [*DPP+16]

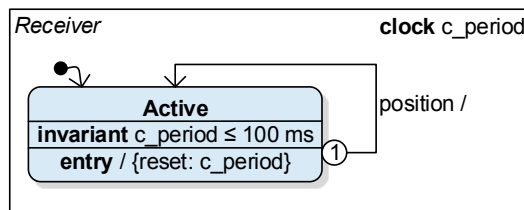


Figure C.45: Role Receiver of RTCP PositionTransmission [*DPP+16]

C.1.3.5 RTCP AllPositionsTransmission

The RTCP PositionTransmission consists of the two roles Sender and Receiver. In this RTCP, the Sender sends a list of positions to the role Receiver. In particular, the list of positions is sent periodically after 50 to 100 ms.

The structure of the RTCP is shown in Figure C.46 while the role behaviors are shown in the Figures C.47 and C.48.

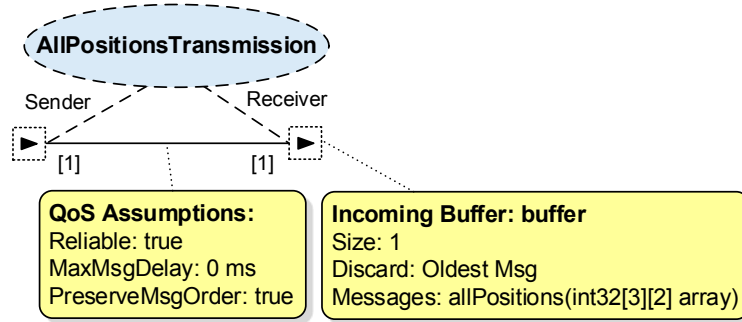


Figure C.46: Structure of RTCP AllPositionsTransmission [*DPP+16]

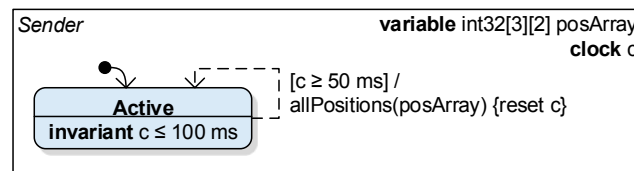


Figure C.47: Role Sender of RTCP AllPositionsTransmission [*DPP+16]

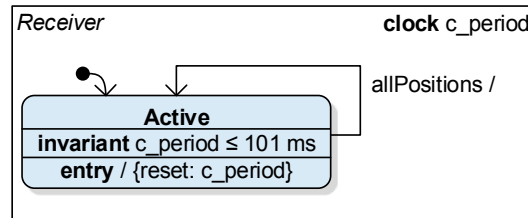


Figure C.48: Role Receiver of RTCP AllPositionsTransmission [*DPP+16]

C.2 MTCTL

In this section, we present details from our evaluation in Section 4.7. In particular, we show the results in Tables C.1, C.2, C.3, C.4, C.5, and C.6. Each column of each table defines a unique index in column 1, defines an informal requirement in column 2, states our manually translated MTCTL property in column 3, and states the automatically translated English sentences of our MTCTL property in column 4.

Index	Informal Requirement	MTCTL	MTCTL to English
1	If the overtaker is in state Overtaking, then the overtakee is in state NoAcceleration.	AG (stateActive(Overtaker.Overtaking) implies stateActive(Overtakee.NoAcceleration));	It invariantly holds that if the state Overtaking.Overtaker.Overtaking is active, then the state Overtaking.Overtakee.NoAcceleration is active as well.
2	If the overtakee receives an overtaking request, it will either send an accept or decline message.	transitionFiring(Overtakee.NoOvertaking.Idle_to_EvaluatingRequest) leadsTo (transitionFiring(Overtakee.NoOvertaking.EvaluatingRequest_to_Idle) or transitionFiring(Overtakee.NoOvertaking.EvaluatingRequest_to_exit1));	Whenever the transition Overtaking.Overtakee.NoOvertaking.Idle_to_EvaluatingRequest is firing, then eventually the transition Overtaking.Overtakee.NoOvertaking.EvaluatingRequest_to_Idle is firing or the transition Overtaking.Overtakee.NoOvertaking.EvaluatingRequest_to_exit1 is firing as well.
3	If the overtaker receives an accept message in its buffer within 6 seconds after sending the request, it will switch to state Overtaking.	messageInBuffer(Repository.accept, Overtaker.b1) and Overtaker.c <= 6 secs leadsTo stateActive(Overtaker.Overtaking);	Whenever the message Repository.accept is in the buffer Overtaking.Overtaker.buffer1 and Overtaking.Overtaker.c1 is at most 6 seconds, then eventually Overtaking.Overtaker.Overtaking is active.
4	It is possible to initiate the overtaking more than once.	Not possible in MTCTL without additional variables in the RTCP.	–

Table C.1: Evaluation Results for RTCP Overtaking (Part 1)

Index	Informal Requirement	MTCTL	MTCTL to English
5	If an overtaking request is declined, it is possible to request again and get an accept.	Not possible in MTCTL without additional variables in the RTCP.	–
6	There shall never be a deadlock in the protocol.	AG not deadlock;	It invariantly holds that no deadlock occurs.
7	The connector will never overflow.	AG not connectorOverflow;	It invariantly holds that no connector overflow occurs.
8	All states are reachable.	forall (s : States) EF stateActive (s);	For all states s it can possibly happen that the state s is active.
9	All transitions may be fired.	forall (t : Transitions) EF transitionFiring (t);	For all transitions t it can possibly happen that the transition t is firing.
10	No message will ever be discarded in any buffer.	forall (b : Buffers) AG not messageDiscarded (b);	For all buffers b it invariantly holds that no message in the buffer b gets discarded.
11	There is no buffer that is always empty.	forall (b : Buffers) EF bufferMessageCount (b) >= 1;	For all buffers b it can possibly happen that the number of messages in buffer b is at least 1.
12	All message buffers may reach their capacity limit.	EF bufferMessageCount (Overtaker.b1) == 1 and EF bufferMessageCount (Overtakee.b2) == 2;	It can possibly happen that the number of messages in buffer Overtaking.Overtaker.b1 is 1 and it can possibly happen that the number of messages in buffer Overtaking.Overtakee.b2 is 2.
13	The number of message in buffer b1 of role Overtaker is always smaller than two.	AG bufferMessageCount (Overtaker.b1) < 2;	It invariantly holds that the number of messages in buffer Overtaking.Overtaker.b1 is less than 3.

Table C.2: Evaluation Results for RTCP Overtaking (Part 2)

Index	Informal Requirement	MTCTL	MTCTL to English
14	The number of message in buffer b2 of role Overtakee will never be four.	AG <code>bufferMessageCount(Overtakee.b2)= 4;!</code>	It invariantly holds that the number of messages in buffer Overtaking.Overtakee.b2 is not 4.
15	Each message may be in transit.	forall (m : <code>MessageTypes</code>) EF <code>messageInTransit(m)</code> ;	For all messages m it can possibly happen that the message m is in transit.
16	There is no message type that cannot possibly arrive in a buffer.	forall (m : <code>MessageTypes</code>) exists (b : <code>Buffers</code>) EF <code>messageInBuffer(m, b)</code> ;	For all messages m there exists a buffer b so that it can possibly happen that the message m is in the buffer b.
17	The overtaker may never send a request.	EG <code>stateActive(Overtaker.NoOvertaking.Idle)</code> ;	It might always hold that the state Overtaking.Overtaker.NoOvertaking.Idle is active.
18	The state Overtaking of role Overtaker is contained in the RTSC <code>overtaker_rtsc</code> .	AG <code>stateInStatechart(Overtaker.NoOvertaking, Overtaker.overtaker_Role)</code> ;	It invariantly holds that the state Overtaking.Overtaker.NoOvertaking is in statechart Overtaking.Overtaker.overtaker_Role.
19	The state Overtaker.Overtaking.Init is a substate of the state Overtaker.Overtaking.	AG <code>substateOf(Overtaker.Overtaking.Init, Overtaker.Overtaking)</code> ;	It invariantly holds that the state Overtaking.Overtaker.Overtaking.Init is a substate of Overtaking.Overtaker.Overtaking.
20	The source state of the transition from state Overtaking.Init to Changed of role Overtaker may be active.	EF <code>stateActive(sourceState(Overtaker.Overtaking.Init_to_Changed))</code> ;	It can possibly happen that the state, which is the source of transition Overtaking.Overtaker.Overtaking.Init_to_Changed, is active.

Table C.3: Evaluation Results for RTCP Overtaking (Part 3)

Index	Informal Requirement	MTCTL	MTCTL to English
21	The target state of the transition from state Overtaking.Init to Changed of role Overtaker may be active.	EF stateActive(targetState (Overtaker.Overtaking.Init_to_Changed));	It can possibly happen that the state, which is the target of transition Overtaking.Overtaker.Overtaking.Init_to_Changed, is active.
22	The overtakee instance is at most 10 seconds in state NoAcceleration.LaneChanged.	Not possible in MTCTL without an additional clock in the RTCP.	–
23	The clock Overtakee.c is eventually greater than one second.	AF Overtakee.c >= 0 secs;	Eventually it holds that Overtaking.Overtakee.c is at least 0 seconds.

Table C.4: Evaluation Results for RTCP Overtaking (Part 4)

Index	Informal Requirement	MTCTL	MTCTL to English
24	It may be the case that all overtakees are in state NoAcceleration at the same time.	forall (j : Instances<Overtakee>) EF stateActive (Overtakee.NoAcceleration[j]);	For all instances j of Overtaking.Overtakee it can possibly happen that the state Overtaking.Overtakee.NoAcceleration of instance j is active.
25	At the same time, each instance of the overtakee may have a message of type request in its buffer.	forall (j : Instances<Overtakee>) EF messageInBuffer (Repository.request, Overtakee.b2[j]);	For all instances j of Overtaking.Overtakee it can possibly happen that the message Repository.request is in the buffer Overtaking.Overtakee.b2 of instance j.
26	At the same time, each instance of the overtakee may have 2 messages in its buffer.	forall (j : Instances<Overtakee>) EF bufferMessageCount (Overtakee.b2[j]) == 2;	For all instances j of Overtaking.Overtakee it can possibly happen that the number of messages in buffer Overtaking.Overtakee.b2 of instance j is 2.
27	If the overtaker is in state Overtaking, then all overtakee instances are in state NoAcceleration.	AG forall (i:Instances<Overtaker>) forall (j: Instances<Overtakee>) (stateActive (Overtaker.Main.coordinator.Overtaking[i]) implies stateActive (Overtakee.NoAcceleration[j]));	It invariantly holds that for all instances i of Overtaking.Overtaker for all instances j of Overtaking.Overtakee it holds that if the state Overtaking.Overtaker.Overtaking of instance i is active, then the state Overtaking.Overtakee.NoAcceleration of instance j is active as well.

Table C.5: Evaluation Results for RTCP MultiOvertaking.

Index	Informal Requirement	MTCTL	MTCTL to English
28	One of the subrole instances of role Overtaker may reach the state Sent in region subrole_multicast1.	exists (i:Subinstances<Overtaker>) (EF stateActive (Overtaker.Main.subrole.subrole_Main.subrole_multicast1.Sent[i]));	There exists a subinstance i of Overtaking.Overtaker so that it can possibly happen that the state Overtaking.Overtaker.Main.subrole.Subrole_Main.subrole_multicast1.Sent of instance i is active.

Table C.6: Evaluation Results for RTCP MultiOvertaking_wSubroleRegion

C.3 DSMC

In this section, we present details from our evaluation in Section 5.8. In particular, the tables C.7 and C.8 show the reachability properties that we selected for each of our RTCPs in order to validate the hypotheses H2 and H5.

System	Protocol	Protocol Form	Reachability Property
Overtaking Cars	Overtaking	one-to-one	EF <code>stateActive</code> (Overtakee.NoOvertaking.EvaluatingRequest);
Overtaking Cars	Multi-Overtaking _wSubrole-Region	one-to-many	exists (i : <code>Instances</code> <Overtakee>) EF <code>stateActive</code> (Overtakee.NoAcceleration.BrakingAllowed[i]);
Overtaking Cars (Extended)	Allow	one-to-one	EF <code>stateActive</code> (Controller.Requested);
Overtaking Cars (Extended)	Change-Section	one-to-one	EF <code>stateActive</code> (Section.Intermediate);
Overtaking Cars (Extended)	Delegate	one-to-one	EF <code>stateActive</code> (Executor.Initiate);
Overtaking Cars (Extended)	Inform	one-to-one	EF <code>stateActive</code> (Approacher.FixDrive);
Overtaking Cars (Extended)	Limit-Velocity	one-to-one	EF <code>stateActive</code> (Limiter.Fixed);
Overtaking Cars (Extended)	Overtake	one-to-one	EF <code>stateActive</code> (Overtakee.Requested);
Overtaking Cars (Extended)	VMode Control	one-to-one	EF <code>stateActive</code> (VelocityGetter.Fast);
RailCab	EnterSection	one-to-one	EF <code>stateActive</code> (Railcab.WaitForAnswer);
RailCab	NextSection-Free	one-to-one	EF <code>stateActive</code> (Tracksection.Request);

Table C.7: MTCTL Reachability Properties for Evaluating Hypotheses H3 and H5 (Part 1)

System	Protocol	Protocol Form	Reachability Property
BeBot	Navigation	one-to-one	EF stateActive(Provider.Moving);
BeBot	Delegation	one-to-one	EF stateActive(Slave.Position_Check);
BeBot	Distribution	one-to-many	exists (i : Subinstances<Distributor>) EF stateActive(Distributor.Main.sub_role_template.Active.receiveSinglePositions.Received[i]);
BeBot	Position-Transmission	one-to-many	exists (i : Instances<receiver>) EF stateActive(Receiver.Intermediate[i]);
BeBot	AllPositions-Transmission	one-to-one	EF stateActive(Receiver.Intermediate);

Table C.8: MTCTL Reachability Properties for Evaluating Hypotheses H3 and H5 (Part 2)

DETAILS OF THE RTCP OVERTAKING

In this appendix, we show details of our one-to-one RTCP Overtaking, which we introduced formally in Section 3.5.1. The appendix is structured as follows: First, in Section D.1, we present the UPPAAL representation of RTCP Overtaking. Afterwards, in Section D.2, we present an example of a counterexample trace for this RTCP.

D.1 Uppaal Representation

In this section, we show the UPPAAL representation of RTCP Overtaking and its MTCTL properties. At first, the following 10 figures show the generated timed automata templates in UPPAAL's concrete syntax. All templates were initially layouted by GraphViz. Afterwards, we did some slight layout modifications, e.g., rearranging labels.

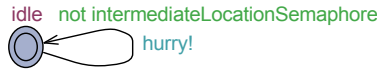


Figure D.1: Timed Automata Template UrgencyProvider

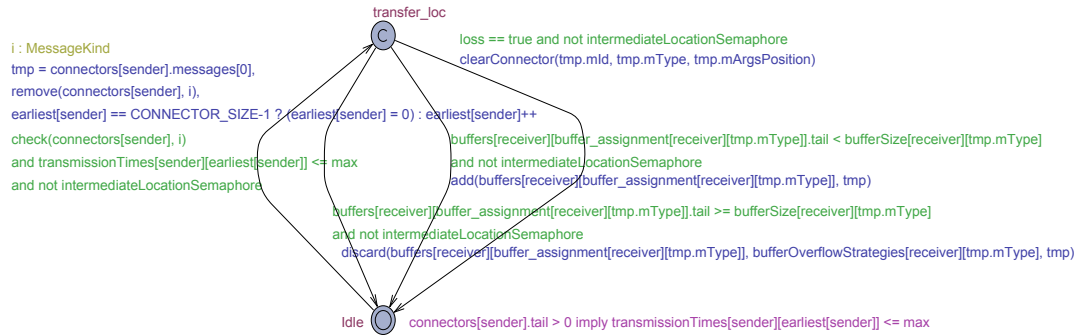


Figure D.2: Timed Automata Template Connector

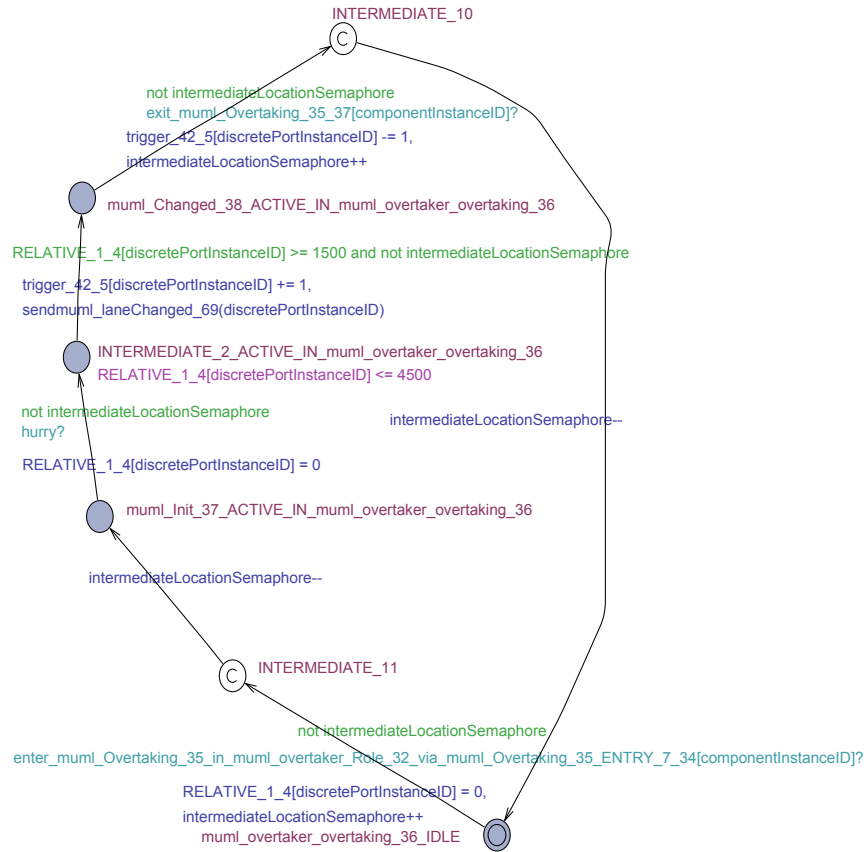


Figure D.3: Timed Automata Template `muml_overtaker_overtaking_36_36`

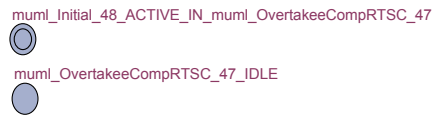


Figure D.4: Timed Automata Template `muml_OvertakeeCompRTSC_47_54`

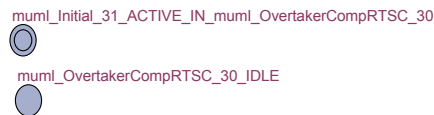
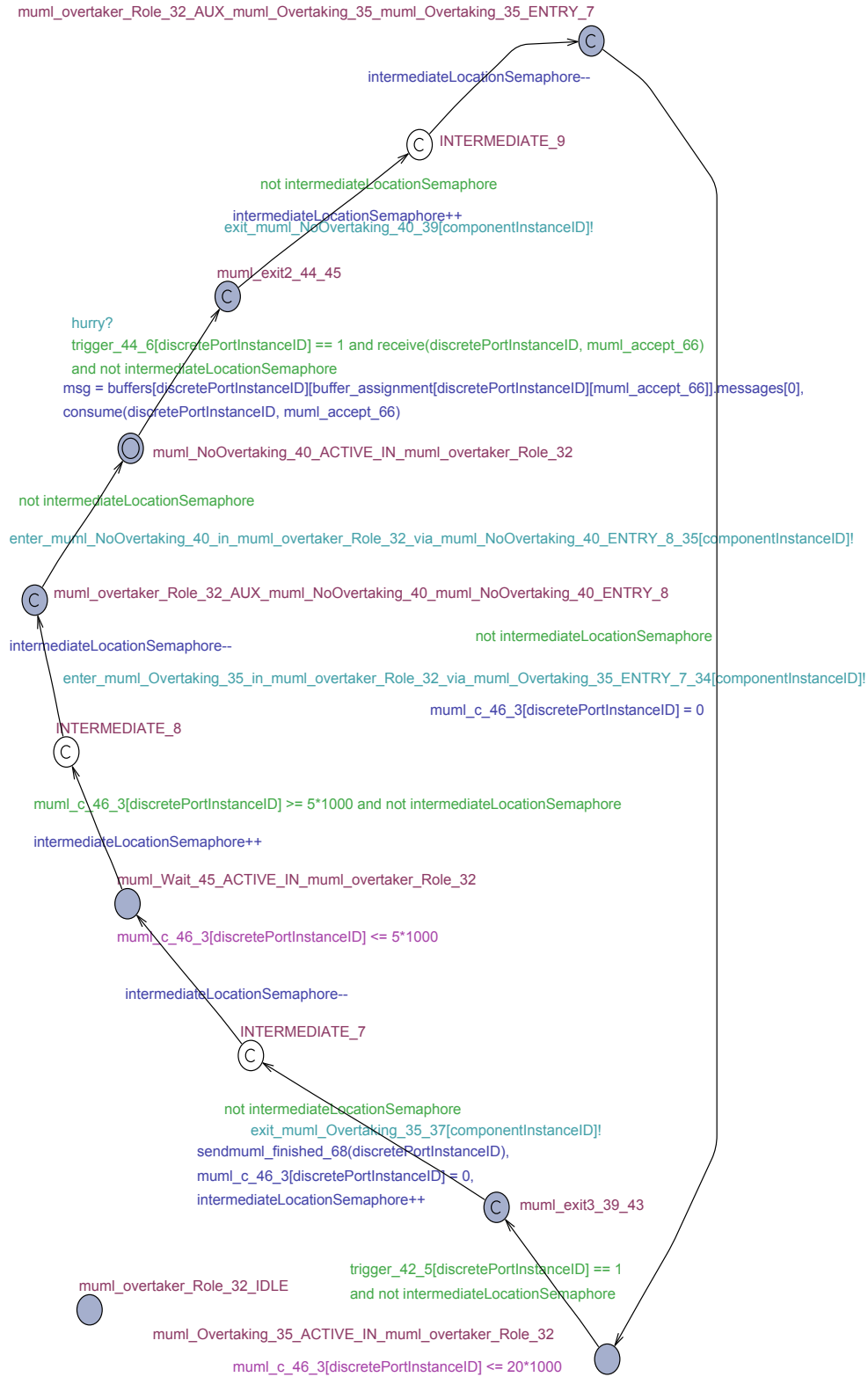


Figure D.5: Timed Automata Template `muml_OvertakerCompRTSC_30_40`

Figure D.6: Timed Automata Template `muml_overtaker_Role_32_33`

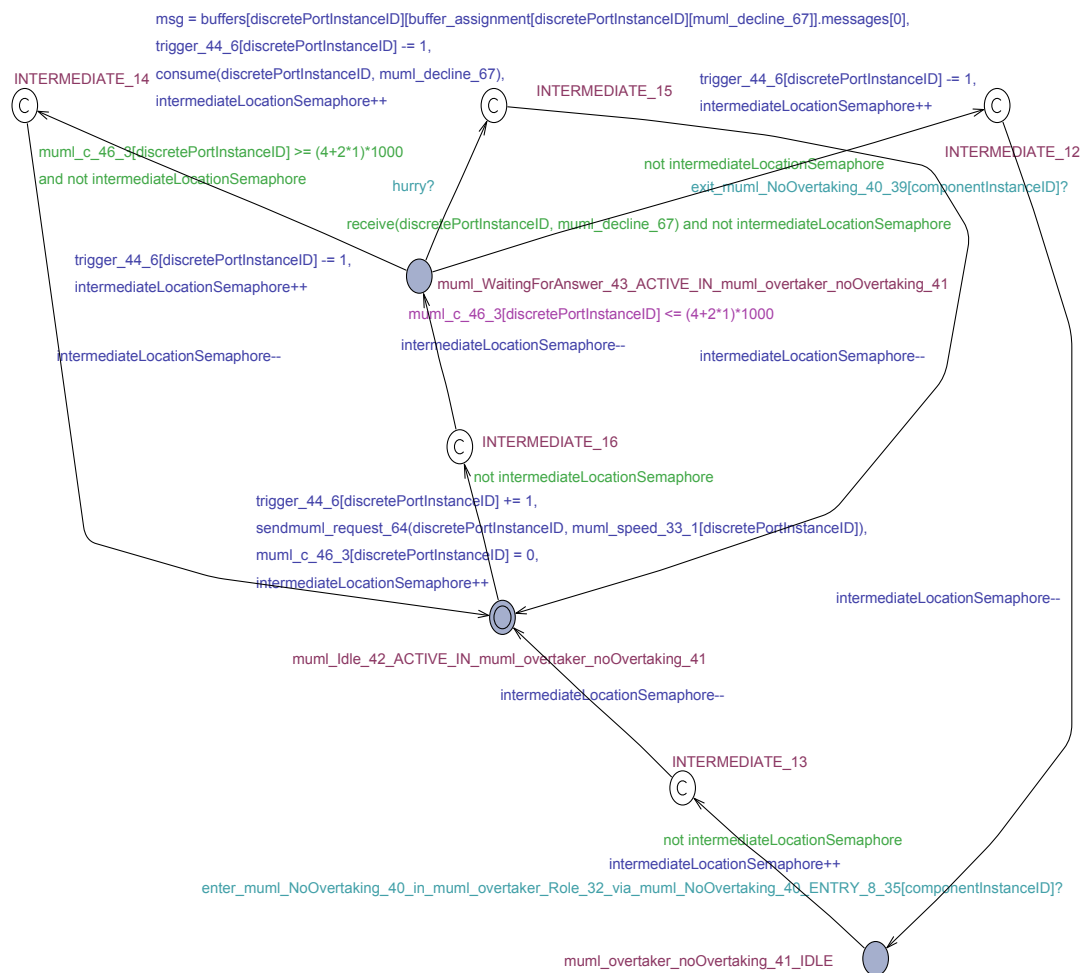


Figure D.7: Timed Automata Template `muml_overtaker_noOvertaking_41_41`



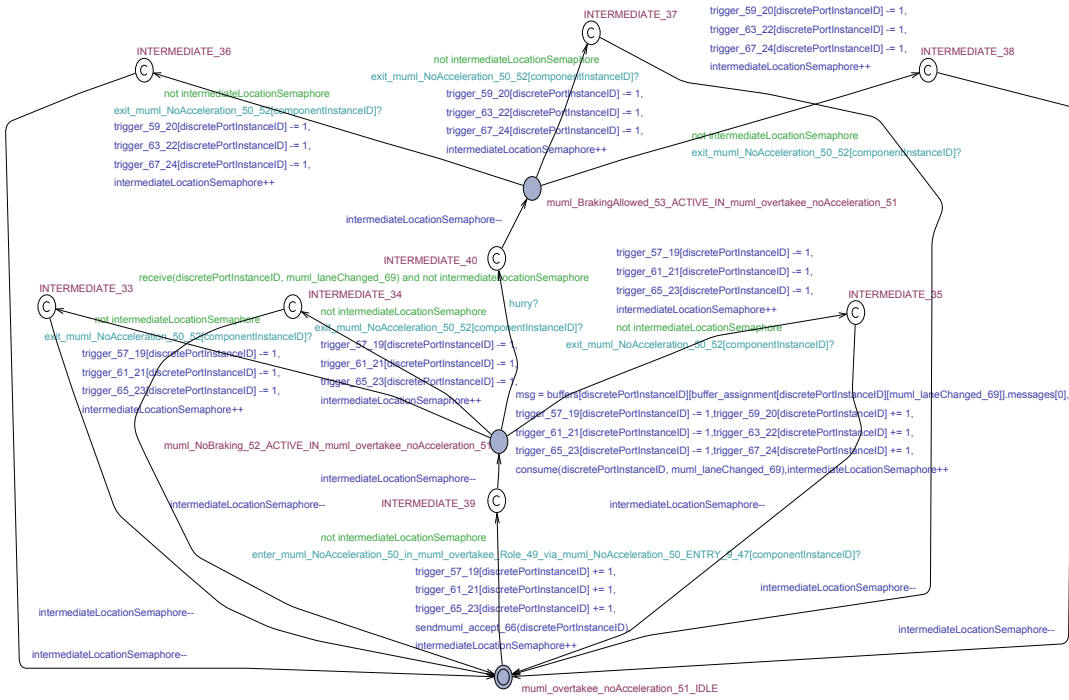


Figure D.9: Timed Automata Template muml_overtakee_noAcceleration_51_55

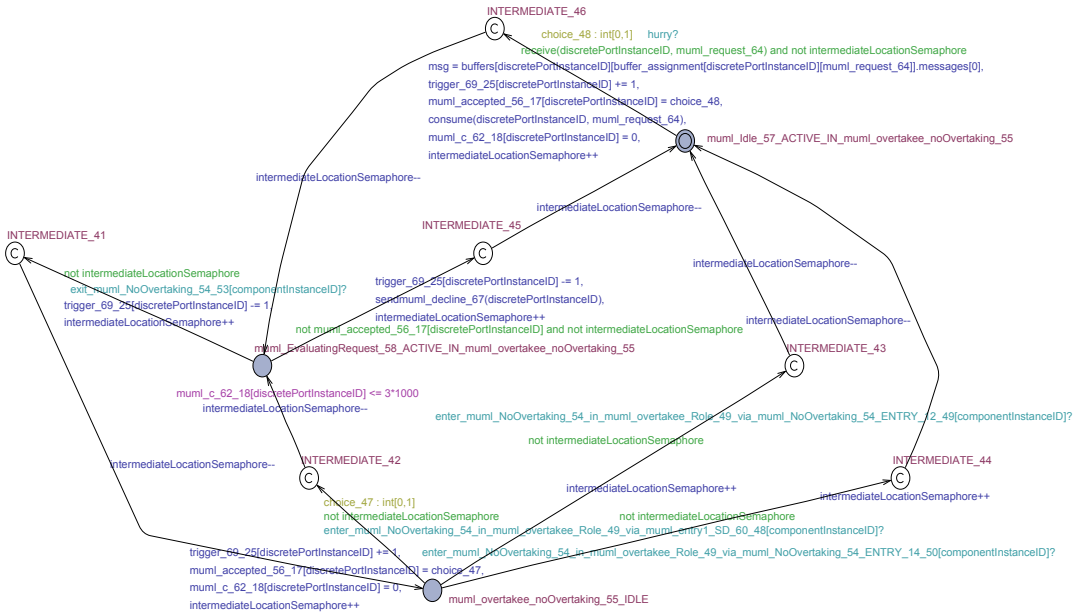


Figure D.10: Timed Automata Template muml_overtakee_noOvertaking_55_56

Next, we show the corresponding global declarations for these timed automata. Compared to the original transformation output, we only removed empty lines and removed some line breaks.

```
typedef int[-2147483648,2147483647]
MUMLint; typedef int[-32768,32767]
MUMLShort; typedef int[-128,127] MUMLByte;
const int NUM_OF_MESSAGE_KINDS = 5;
const int NUM_MAX_MESSAGES_IN_TRANSIT = 11;
const int MAX_OF_BUFFER_AND_CONNECTOR_SIZE = 5;
const int CONNECTOR_SIZE = 5;
const int MAX_NUM_OF_MESSAGE_BUFFERS_PER_ROLE = 1;
typedef int[0,NUM_MAX_MESSAGES_IN_TRANSIT] MessageId;
typedef int[0,NUM_OF_MESSAGE_KINDS] MessageKind;
typedef struct {MessageId mId; MessageKind mType; int mArgsPosition;} Message;
typedef struct {
    int[0,MAX_OF_BUFFER_AND_CONNECTOR_SIZE] tail; Message messages[MAX_OF_BUFFER_AND_CONNECTOR_SIZE];
    bool messageDiscarded;} Buffer;
typedef struct {
    MessageId muml_request_64MessageID; MUMLByte muml_speed_65_;
} muml_request_64Parameters;
typedef struct {MessageId muml_finished_68MessageID;} muml_finished_68Parameters;
typedef struct {MessageId muml_laneChanged_69MessageID;} muml_laneChanged_69Parameters;
typedef struct {MessageId muml_decline_67MessageID;} muml_decline_67Parameters;
typedef struct {MessageId muml_accept_66MessageID;} muml_accept_66Parameters;
const int NUM_OF_DISCRETE_PORT_INSTANCES = 2;
const int NUM_OF_COMPONENT_INSTANCES = 2;
typedef int[0,NUM_OF_DISCRETE_PORT_INSTANCES-1] discretePortInstance;
typedef int[0,NUM_OF_COMPONENT_INSTANCES-1] componentInstance;
const int[-1,NUM_OF_DISCRETE_PORT_INSTANCES] next[discretePortInstance] = {-1,-1};
const int[-1,NUM_OF_DISCRETE_PORT_INSTANCES] previous[discretePortInstance] = {-1,-1};
const int[-1,NUM_OF_DISCRETE_PORT_INSTANCES] first[discretePortInstance] = {-1,-1};
const int[-1,NUM_OF_DISCRETE_PORT_INSTANCES] last[discretePortInstance] = {-1,-1};
const int[-1,NUM_OF_DISCRETE_PORT_INSTANCES] self[discretePortInstance] = {0,1};
const int[-1,MAX_NUM_OF_MESSAGE_BUFFERS_PER_ROLE-1] buffer_assignment[discretePortInstance][NUM_OF_MESSAGE_KINDS+1] = {{-1,-1,-1,-1,0,0},{-1,0,0,0,-1,-1}};
const int[-1,MAX_OF_BUFFER_AND_CONNECTOR_SIZE] bufferSize[discretePortInstance][NUM_OF_MESSAGE_KINDS+1] = {{-1,-1,-1,-1,1,1},{-1,3,3,3,-1,-1}};
const int DISCARD_INCOMING = 0;
const int DISCARD_OLDEST = 1;
const int[-1,1] bufferOverflowStrategies[discretePortInstance][NUM_OF_MESSAGE_KINDS+1] = {{-1,-1,-1,-1,-1,1},
DISCARD_INCOMING, DISCARD_INCOMING},{-1,DISCARD_INCOMING,DISCARD_INCOMING,DISCARD_INCOMING,-1,-1}};
bool connectorOverflow;
int[0,31] intermediateLocationSemaphore;
Buffer buffers[discretePortInstance][MAX_NUM_OF_MESSAGE_BUFFERS_PER_ROLE];
Buffer connectors[discretePortInstance];
int earliest[discretePortInstance];
int latest[discretePortInstance];
clock transmissionTimes[discretePortInstance][CONNECTOR_SIZE];
const MessageKind nullMessageKind = 0, muml_request_64 = 1, muml_finished_68 = 2, muml_laneChanged_69 = 3,
muml_decline_67 = 4, muml_accept_66 = 5;
const MessageId nullMessageId = 0;
const Message nullMessage = {nullMessageId, nullMessageKind, 0};
MessageId freeIds[NUM_MAX_MESSAGES_IN_TRANSIT] = {1,2,3,4,5,6,7,8,9,10,11};
muml_request_64Parameters muml_request_64NullMessage;
muml_finished_68Parameters muml_finished_68NullMessage;
muml_laneChanged_69Parameters muml_laneChanged_69NullMessage;
muml_decline_67Parameters muml_decline_67NullMessage;
muml_accept_66Parameters muml_accept_66NullMessage;
int[0,NUM_MAX_MESSAGES_IN_TRANSIT] muml_request_64Tail;
int[0,NUM_MAX_MESSAGES_IN_TRANSIT] muml_finished_68Tail;
int[0,NUM_MAX_MESSAGES_IN_TRANSIT] muml_laneChanged_69Tail;
int[0,NUM_MAX_MESSAGES_IN_TRANSIT] muml_decline_67Tail;
int[0,NUM_MAX_MESSAGES_IN_TRANSIT] muml_accept_66Tail;
muml_request_64Parameters muml_request_64MessageArguments[NUM_MAX_MESSAGES_IN_TRANSIT+1];
muml_finished_68Parameters muml_finished_68MessageArguments[NUM_MAX_MESSAGES_IN_TRANSIT+1];
muml_laneChanged_69Parameters muml_laneChanged_69MessageArguments[NUM_MAX_MESSAGES_IN_TRANSIT+1];
muml_decline_67Parameters muml_decline_67MessageArguments[NUM_MAX_MESSAGES_IN_TRANSIT+1];
muml_accept_66Parameters muml_accept_66MessageArguments[NUM_MAX_MESSAGES_IN_TRANSIT+1];
void add(Buffer &buffer, Message message) {
    buffer.messages[buffer.tail] = message;
    buffer.tail++;
}
int getNumberOfElementsInBuffer(Buffer b) {
    return b.tail;
}
bool check(Buffer buffer, MessageKind mKind) {
    Message nextMessage = buffer.messages[0];
    return mKind != nullMessageKind and nextMessage.mType == mKind;
}
bool checkMessageInBuffer(Buffer buffer, MessageKind mKind) {
    int i;
    for(i = 0; i < MAX_OF_BUFFER_AND_CONNECTOR_SIZE; i++)
        if(buffer.messages[i].mType == mKind)
            return true;
    return false;
}
```

```

}
void remove(Buffer &buffer, MessageKind m) {
    if (check(buffer, m)) {
        int i;
        for (i = 0; i < MAX_OF_BUFFER_AND_CONNECTOR_SIZE-1; i++)
            buffer.messages[i] = buffer.messages[i+1];
        buffer.messages[MAX_OF_BUFFER_AND_CONNECTOR_SIZE-1] = nullMessage;
        buffer.tail--;
    }
}
bool receive(discretePortInstance receiver, MessageKind mKind) {
    return check(buffers[receiver][buffer_assignment[receiver][mKind]], mKind);
}
MessageId requestId() {
    int i;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT; i++)
        if (freeIds[i] != nullMessageId) {
            MessageId result;
            result = freeIds[i];
            freeIds[i] = nullMessageId;
            return result;
        }
    return nullMessageId;
}
bool releaseId(MessageId id) {
    int i;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT; i++)
        if (freeIds[i] == id)
            return false;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT; i++)
        if (freeIds[i] == nullMessageId) {
            freeIds[i] = id;
            return true;
        }
    return false;
}
int addmuml_request_64Arguments(muml_request_64Parameters m) {
    int i;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT+1; i++)
        if (muml_request_64MessageArguments[i].muml_request_64MessageID == 0) {
            muml_request_64MessageArguments[i] = m;
            return i;
        }
    return -1;
}
int addmuml_finished_68Arguments(muml_finished_68Parameters m) {
    int i;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT+1; i++)
        if (muml_finished_68MessageArguments[i].muml_finished_68MessageID == 0) {
            muml_finished_68MessageArguments[i] = m;
            return i;
        }
    return -1;
}
int addmuml_laneChanged_69Arguments(muml_laneChanged_69Parameters m) {
    int i;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT+1; i++)
        if (muml_laneChanged_69MessageArguments[i].muml_laneChanged_69MessageID == 0) {
            muml_laneChanged_69MessageArguments[i] = m;
            return i;
        }
    return -1;
}
int addmuml_decline_67Arguments(muml_decline_67Parameters m) {
    int i;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT+1; i++)
        if (muml_decline_67MessageArguments[i].muml_decline_67MessageID == 0) {
            muml_decline_67MessageArguments[i] = m;
            return i;
        }
    return -1;
}
int addmuml_accept_66Arguments(muml_accept_66Parameters m) {
    int i;
    for (i = 0; i < NUM_MAX_MESSAGES_IN_TRANSIT+1; i++)
        if (muml_accept_66MessageArguments[i].muml_accept_66MessageID == 0) {
            muml_accept_66MessageArguments[i] = m;
            return i;
        }
    return -1;
}
bool removemuml_request_64Arguments(int mArgsPosition) {
    muml_request_64MessageArguments[mArgsPosition] = muml_request_64NullMessage;
    return true;
}
bool removemuml_finished_68Arguments(int mArgsPosition) {
    muml_finished_68MessageArguments[mArgsPosition] = muml_finished_68NullMessage;
    return true;
}

```



```

}
bool removemuml_laneChanged_69Arguments(int mArgsPosition) {
    muml_laneChanged_69MessageArguments[mArgsPosition] = muml_laneChanged_69NullMessage;
    return true;
}
bool removemuml_decline_67Arguments(int mArgsPosition) {
    muml_decline_67MessageArguments[mArgsPosition] = muml_decline_67NullMessage;
    return true;
}
bool removemuml_accept_66Arguments(int mArgsPosition) {
    muml_accept_66MessageArguments[mArgsPosition] = muml_accept_66NullMessage;
    return true;
}
void sendmuml_request_64(discretePortInstance sender, MUMByte muml_speed_65_0) {
    if(connectors[sender].tail < CONNECTOR_SIZE) {
        muml_request_64Parameters muml_request_64Instance = {nullMessageId, muml_speed_65_0};
        Message message = nullMessage;
        MessageId mId = nullMessageId;
        mId = requestId();
        muml_request_64Instance.muml_request_64MessageID = mId;
        message.mId = mId;
        message.mType = muml_request_64;
        message.mArgsPosition = addmuml_request_64Arguments(muml_request_64Instance);
        add(connectors[sender], message);
        transmissionTimes[sender][latest[sender]] = 0;
        latest[sender] == CONNECTOR_SIZE-1 ? (latest[sender] = 0) : latest[sender]++;
    }
    else
        connectorOverflow = true;
}
void sendmuml_finished_68(discretePortInstance sender) {
    if(connectors[sender].tail < CONNECTOR_SIZE) {
        muml_finished_68Parameters muml_finished_68Instance = {nullMessageId};
        Message message = nullMessage;
        MessageId mId = nullMessageId;
        mId = requestId();
        muml_finished_68Instance.muml_finished_68MessageID = mId;
        message.mId = mId;
        message.mType = muml_finished_68;
        message.mArgsPosition = addmuml_finished_68Arguments(muml_finished_68Instance);
        add(connectors[sender], message);
        transmissionTimes[sender][latest[sender]] = 0;
        latest[sender] == CONNECTOR_SIZE-1 ? (latest[sender] = 0) : latest[sender]++;
    }
    else
        connectorOverflow = true;
}
void sendmuml_laneChanged_69(discretePortInstance sender) {
    if(connectors[sender].tail < CONNECTOR_SIZE) {
        muml_laneChanged_69Parameters muml_laneChanged_69Instance = {nullMessageId};
        Message message = nullMessage;
        MessageId mId = nullMessageId;
        mId = requestId();
        muml_laneChanged_69Instance.muml_laneChanged_69MessageID = mId;
        message.mId = mId;
        message.mType = muml_laneChanged_69;
        message.mArgsPosition = addmuml_laneChanged_69Arguments(muml_laneChanged_69Instance);
        add(connectors[sender], message);
        transmissionTimes[sender][latest[sender]] = 0;
        latest[sender] == CONNECTOR_SIZE-1 ? (latest[sender] = 0) : latest[sender]++;
    }
    else
        connectorOverflow = true;
}
void sendmuml_decline_67(discretePortInstance sender) {
    if(connectors[sender].tail < CONNECTOR_SIZE) {
        muml_decline_67Parameters muml_decline_67Instance = {nullMessageId};
        Message message = nullMessage;
        MessageId mId = nullMessageId;
        mId = requestId();
        muml_decline_67Instance.muml_decline_67MessageID = mId;
        message.mId = mId;
        message.mType = muml_decline_67;
        message.mArgsPosition = addmuml_decline_67Arguments(muml_decline_67Instance);
        add(connectors[sender], message);
        transmissionTimes[sender][latest[sender]] = 0;
        latest[sender] == CONNECTOR_SIZE-1 ? (latest[sender] = 0) : latest[sender]++;
    }
    else
        connectorOverflow = true;
}
void sendmuml_accept_66(discretePortInstance sender) {
    if(connectors[sender].tail < CONNECTOR_SIZE) {
        muml_accept_66Parameters muml_accept_66Instance = {nullMessageId};
        Message message = nullMessage;
        MessageId mId = nullMessageId;
        mId = requestId();
        muml_accept_66Instance.muml_accept_66MessageID = mId;
    }
}

```

```

    message.mId = mId;
    message.mType = muml_accept_66;
    message.mArgsPosition = addmuml_accept_66Arguments(muml_accept_66Instance);
    add(connectors[sender], message);
    transmissionTimes[sender][latest[sender]] = 0;
    latest[sender] == CONNECTOR_SIZE-1 ? (latest[sender] = 0) : latest[sender]++;
}
else
    connectorOverflow = true;
}

void consume(discretePortInstance receiver, MessageKind mKind) {
    if (receive(receiver, mKind)) {
        Message nextMessage = buffers[receiver][buffer_assignment[receiver][mKind]].messages[0];
        releaseId(nextMessage.mId);
        remove(buffers[receiver][buffer_assignment[receiver][mKind]], mKind);
        if (mKind == muml_request_64)
            removemuml_request_64Arguments(nextMessage.mArgsPosition);
        if (mKind == muml_finished_68)
            removemuml_finished_68Arguments(nextMessage.mArgsPosition);
        if (mKind == muml_laneChanged_69)
            removemuml_laneChanged_69Arguments(nextMessage.mArgsPosition);
        if (mKind == muml_decline_67)
            removemuml_decline_67Arguments(nextMessage.mArgsPosition);
        if (mKind == muml_accept_66)
            removemuml_accept_66Arguments(nextMessage.mArgsPosition);
    }
}

void clearConnector(MessageId id, MessageKind messageTypeToDiscard, int argsPosition) {
    releaseId(id);
    if (messageTypeToDiscard == muml_request_64)
        removemuml_request_64Arguments(argsPosition);
    if (messageTypeToDiscard == muml_finished_68)
        removemuml_finished_68Arguments(argsPosition);
    if (messageTypeToDiscard == muml_laneChanged_69)
        removemuml_laneChanged_69Arguments(argsPosition);
    if (messageTypeToDiscard == muml_decline_67)
        removemuml_decline_67Arguments(argsPosition);
    if (messageTypeToDiscard == muml_accept_66)
        removemuml_accept_66Arguments(argsPosition);
}

void discard(Buffer &buffer, int bufferOverflowStrategy, Message message) {
    MessageId messageToDiscard = message.mId;
    MessageKind messageTypeToDiscard = message.mType;
    if (bufferOverflowStrategy == DISCARD_OLDEST) {
        messageToDiscard = buffer.messages[0].mId;
        messageTypeToDiscard = buffer.messages[0].mType;
        remove(buffer, messageTypeToDiscard);
        buffer.messages[buffer.tail] = message;
        buffer.tail++;
    }
    releaseId(messageToDiscard);
    buffer.messageDiscarded = true;
    clearConnector(messageToDiscard, message.mArgsPosition, messageTypeToDiscard);
}

MUMLByte muml_speed_33_1[discretePortInstance] = {50,50};
MUMLByte muml_assumedEvalTime_34_2[discretePortInstance] = {4,4};
clock muml_c_46_3[discretePortInstance];
clock RELATIVE_1_4[discretePortInstance];
urgent chan exit_muml_Initial_31_38[componentInstance];
urgent chan exit_muml_Overtaking_35_37[componentInstance];
urgent chan enter_muml_Overtaking_35_in_muml_overtaker_Role_32_via_muml_Overtaking_35_ENTRY_7_34[
    componentInstance];
urgent chan exit_muml_NoOvertaking_40_39[componentInstance];
urgent chan enter_muml_NoOvertaking_40_in_muml_overtaker_Role_32_via_muml_NoOvertaking_40_ENTRY_8_35[
    componentInstance];
MUMLInt trigger_42_5[discretePortInstance] = {0,0};
MUMLInt trigger_44_6[discretePortInstance] = {0,0};
bool muml_accepted_56_17[discretePortInstance];
clock muml_c_62_18[discretePortInstance];
urgent chan exit_muml_Initial_48_51[componentInstance];
urgent chan exit_muml_NoAcceleration_50_52[componentInstance];
urgent chan enter_muml_NoAcceleration_50_in_muml_overtakee_Role_49_via_muml_NoAcceleration_50_ENTRY_9_47[
    componentInstance];
urgent chan exit_muml_NoOvertaking_54_53[componentInstance];
urgent chan enter_muml_NoOvertaking_54_in_muml_overtakee_Role_49_via_muml_entry1_SD_60_48[componentInstance];
urgent chan enter_muml_NoOvertaking_54_in_muml_overtakee_Role_49_via_muml_NoOvertaking_54_ENTRY_12_49[
    componentInstance];
urgent chan enter_muml_NoOvertaking_54_in_muml_overtakee_Role_49_via_muml_NoOvertaking_54_ENTRY_14_50[
    componentInstance];
MUMLInt trigger_57_19[discretePortInstance] = {0,0};
MUMLInt trigger_59_20[discretePortInstance] = {0,0};
MUMLInt trigger_61_21[discretePortInstance] = {0,0};
MUMLInt trigger_63_22[discretePortInstance] = {0,0};
MUMLInt trigger_65_23[discretePortInstance] = {0,0};
MUMLInt trigger_67_24[discretePortInstance] = {0,0};
MUMLInt trigger_69_25[discretePortInstance] = {0,0};

```

Finally, we show the generated UTCTL properties that are based on the following properties: **AG** (stateActive(Overtaker.Overtaking) implies stateActive(Overtakee.NoAcceleration));, **forall**(b : Buffers) **AG** not messageDiscarded(b);, and MTCTL's default properties. Compared to the original transformation output, we removed empty lines and comments.

```
A[] (muml_overtaker_Role_32_33_0.muml_overtaker_Role_32_AUX_muml_Overtaking_35_muml_Overtaking_35_ENTRY_7 or
muml_overtaker_Role_32_33_0.muml_exit3_39_43 or muml_overtaker_Role_32_33_0.
muml_Overtaking_35_ACTIVE_IN_muml_overtaker_Role_32) imply (muml_overtakee_Role_49_46_1.
muml_NoAcceleration_50_EXIT_13_66 or muml_overtakee_Role_49_46_1.muml_NoAcceleration_50_EXIT_11_62 or
muml_overtakee_Role_49_46_1.muml_NoAcceleration_50_EXIT_10_58 or muml_overtakee_Role_49_46_1.
muml_overtakee_Role_49_AUX_muml_NoAcceleration_50_muml_NoAcceleration_50_ENTRY_9 or
muml_overtakee_Role_49_46_1.muml_NoAcceleration_50_ACTIVE_IN_muml_overtakee_Role_49)
A[] not buffers[0][0].messageDiscarded
A[] not buffers[1][0].messageDiscarded
A[] not deadlock
A[] not connectorOverflow
E< muml_overtaker_Role_32_33_0.INTERMEDIATE_9
E< muml_overtakee_noOvertaking_55_56_1.INTERMEDIATE_45
E< muml_overtaker_noOvertaking_41_41_0.INTERMEDIATE_16
E< muml_overtaker_overtaking_36_36_0.INTERMEDIATE_2_ACTIVE_IN_muml_overtaker_overtaking_36
E< muml_overtakee_noOvertaking_55_56_1.INTERMEDIATE_42
E< muml_overtaker_noOvertaking_41_41_0.INTERMEDIATE_14
E< muml_overtakee_Role_49_46_1.INTERMEDIATE_32 or muml_overtakee_Role_49_46_1.INTERMEDIATE_31
E< muml_overtakee_Role_49_46_1.INTERMEDIATE_27 or muml_overtakee_Role_49_46_1.INTERMEDIATE_28
E< muml_overtaker_noOvertaking_41_41_0.INTERMEDIATE_12
E< muml_overtakee_noOvertaking_55_56_1.INTERMEDIATE_41
E< muml_overtakee_Role_49_46_1.INTERMEDIATE_26
E< muml_overtakee_noOvertaking_55_56_1.INTERMEDIATE_46
E< muml_overtaker_Role_32_33_0.INTERMEDIATE_7
E< muml_overtakee_Role_49_46_1.INTERMEDIATE_30 or muml_overtakee_Role_49_46_1.INTERMEDIATE_29
E< muml_overtaker_overtaking_36_36_0.INTERMEDIATE_10
E< muml_overtakee_noAcceleration_51_55_1.INTERMEDIATE_40
E< muml_overtaker_Role_32_33_0.INTERMEDIATE_8
E< muml_overtaker_noOvertaking_41_41_0.INTERMEDIATE_15
E< muml_OvertakerCompRTSC_30_40_component_0.muml_Initial_31_ACTIVE_IN_muml_OvertakerCompRTSC_30
E< muml_overtaker_overtaking_36_36_0.muml_Init_37_ACTIVE_IN_muml_overtaker_overtaking_36
E< muml_overtakee_noAcceleration_51_55_1.muml_BrakingAllowed_53_ACTIVE_IN_muml_overtakee_noAcceleration_51
E< muml_overtaker_noOvertaking_41_41_0.muml_Idle_42_ACTIVE_IN_muml_overtaker_noOvertaking_41
E< muml_overtakee_noOvertaking_55_56_1.muml_Idle_57_ACTIVE_IN_muml_overtakee_noOvertaking_55
E< muml_overtakee_noOvertaking_55_56_1.muml_EvaluatingRequest_58_ACTIVE_IN_muml_overtakee_noOvertaking_55
E< muml_overtaker_Role_32_33_0.muml_overtaker_Role_32_AUX_muml_NoOvertaking_40_muml_NoOvertaking_40_ENTRY_8 or
muml_overtaker_Role_32_33_0.muml_exit2_44_45 or muml_overtaker_Role_32_33_0.
muml_NoOvertaking_40_ACTIVE_IN_muml_overtaker_Role_32
E< muml_overtakee_Role_49_46_1.muml_NoAcceleration_50_EXIT_13_66 or muml_overtakee_Role_49_46_1.
muml_NoAcceleration_50_EXIT_11_62 or muml_overtakee_Role_49_46_1.muml_NoAcceleration_50_EXIT_10_58 or
muml_overtakee_Role_49_46_1.
muml_overtakee_Role_49_AUX_muml_NoAcceleration_50_muml_NoAcceleration_50_ENTRY_9 or
muml_overtakee_Role_49_46_1.muml_NoAcceleration_50_ACTIVE_IN_muml_overtakee_Role_49
E< muml_overtaker_noOvertaking_41_41_0.muml_WaitingForAnswer_43_ACTIVE_IN_muml_overtaker_noOvertaking_41
E< muml_overtaker_Role_32_33_0.muml_Wait_45_ACTIVE_IN_muml_overtaker_Role_32
E< muml_overtakee_noAcceleration_51_55_1.muml_NoBraking_52_ACTIVE_IN_muml_overtakee_noAcceleration_51
E< muml_overtaker_Role_32_33_0.muml_overtaker_Role_32_AUX_muml_Overtaking_35_muml_Overtaking_35_ENTRY_7 or
muml_overtaker_Role_32_33_0.muml_exit3_39_43 or muml_overtaker_Role_32_33_0.
muml_Overtaking_35_ACTIVE_IN_muml_overtaker_Role_32
E< muml_OvertakeeCompRTSC_47_54_component_1.muml_Initial_48_ACTIVE_IN_muml_OvertakeeCompRTSC_47
E< muml_overtakee_Role_49_46_1.muml_overtakee_Role_49_AUX_muml_NoOvertaking_54_muml_NoOvertaking_54_ENTRY_14 or
muml_overtakee_Role_49_46_1.muml_overtakee_Role_49_AUX_muml_NoOvertaking_54_muml_NoOvertaking_54_ENTRY_12
or muml_overtakee_Role_49_46_1.muml_exit1_61_70 or muml_overtakee_Role_49_46_1.
muml_overtakee_Role_49_AUX_muml_NoOvertaking_54_muml_entry1_SD_60 or muml_overtakee_Role_49_46_1.
muml_NoOvertaking_54_ACTIVE_IN_muml_overtakee_Role_49
E< muml_overtaker_overtaking_36_36_0.muml_Changed_38_ACTIVE_IN_muml_overtaker_overtaking_36
E< checkMessageInBuffer(buffers[1][0], muml_laneChanged_69)
E< checkMessageInBuffer(buffers[1][0], muml_finished_68)
E< checkMessageInBuffer(buffers[1][0], muml_request_64)
E< checkMessageInBuffer(buffers[0][0], muml_request_64)
E< checkMessageInBuffer(buffers[0][0], muml_finished_68)
E< checkMessageInBuffer(buffers[0][0], muml_decline_67)
E< checkMessageInBuffer(buffers[1][0], muml_decline_67)
E< checkMessageInBuffer(buffers[1][0], muml_accept_66)
E< checkMessageInBuffer(buffers[0][0], muml_accept_66)
E< checkMessageInBuffer(buffers[0][0], muml_laneChanged_69)
E< exists (i : discretePortInstance) checkMessageInBuffer(connectors[i], muml_finished_68)
E< exists (i : discretePortInstance) checkMessageInBuffer(connectors[i], muml_laneChanged_69)
E< exists (i : discretePortInstance) checkMessageInBuffer(connectors[i], muml_request_64)
E< exists (i : discretePortInstance) checkMessageInBuffer(connectors[i], muml_accept_66)
E< exists (i : discretePortInstance) checkMessageInBuffer(connectors[i], muml_decline_67)
E< getNumberOfElementsInBuffer(buffers[1][0]) >= 1
E< getNumberOfElementsInBuffer(buffers[0][0]) >= 1
```

D.2 Exemplary SVG Counterexample Trace

In this section, we present an example of counterexample trace visualized using our SVG representation. The trace is generated by our DSMC based on the incorrect version of RTCP Overtaking, which we introduced in Section 5.5.3, verified using the MTCTL property **AG** (`stateActive(Overtaker.Overtaking)` *implies* `stateActive(Overtakee.NoAcceleration)`);

The counterexample trace consists of 30 snapshots and 29 snapshot transitions. Due to size limitation, we distribute the trace on the figures D.11, D.12, D.13, D.14, and D.15.

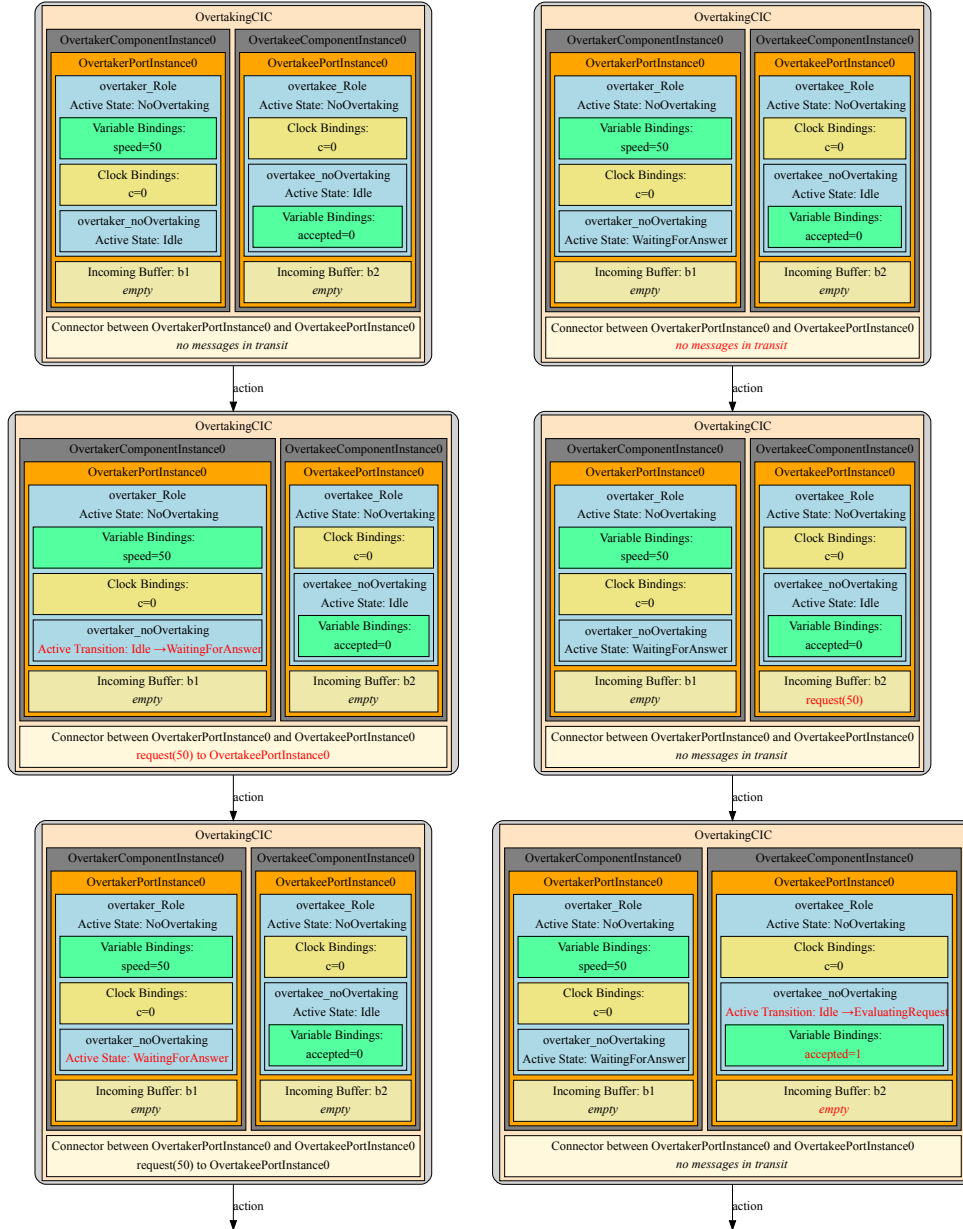


Figure D.11: Snapshots 1–6 of an Exemplary Counterexample Trace

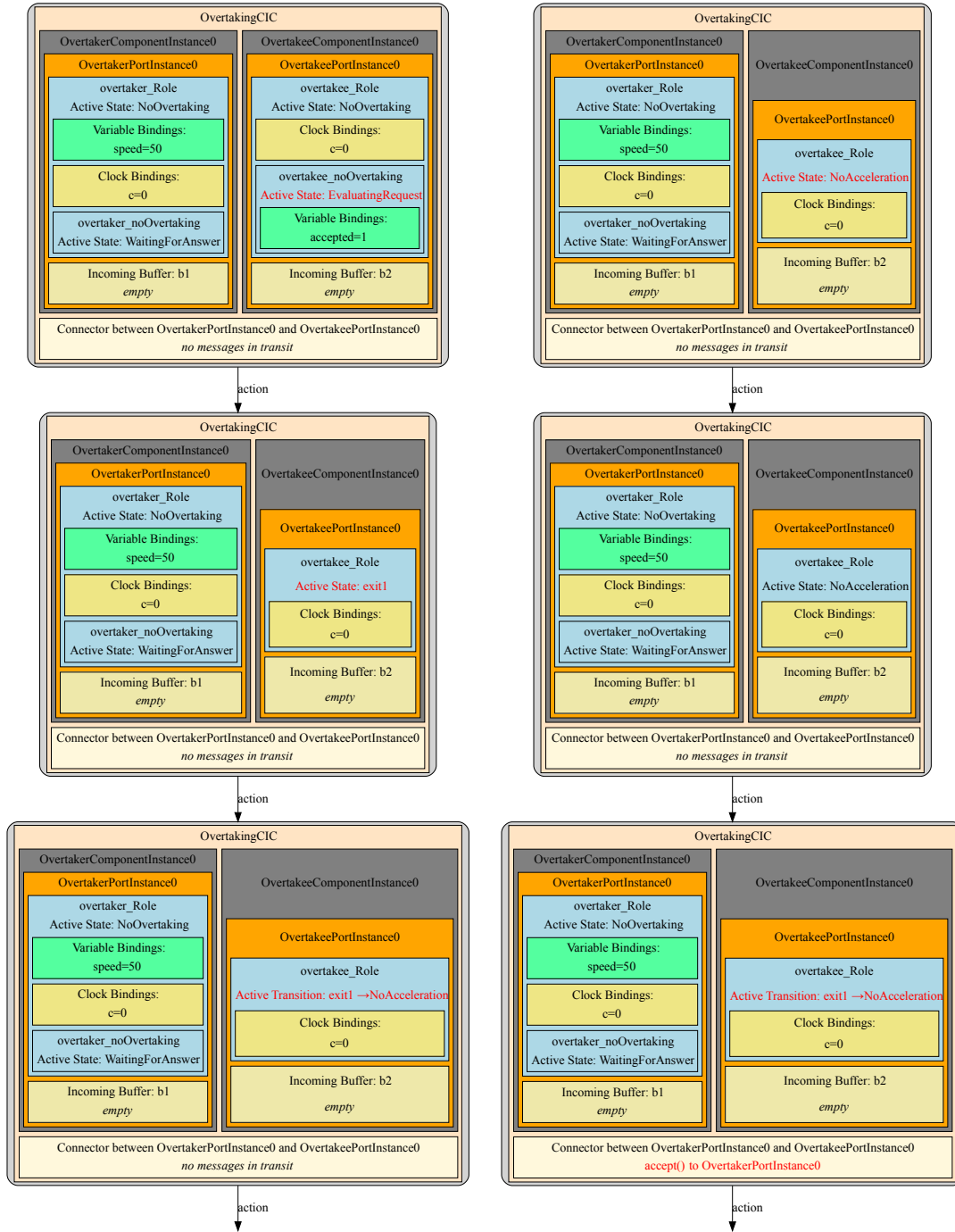


Figure D.12: Snapshots 7–12 of an Exemplary Counterexample Trace

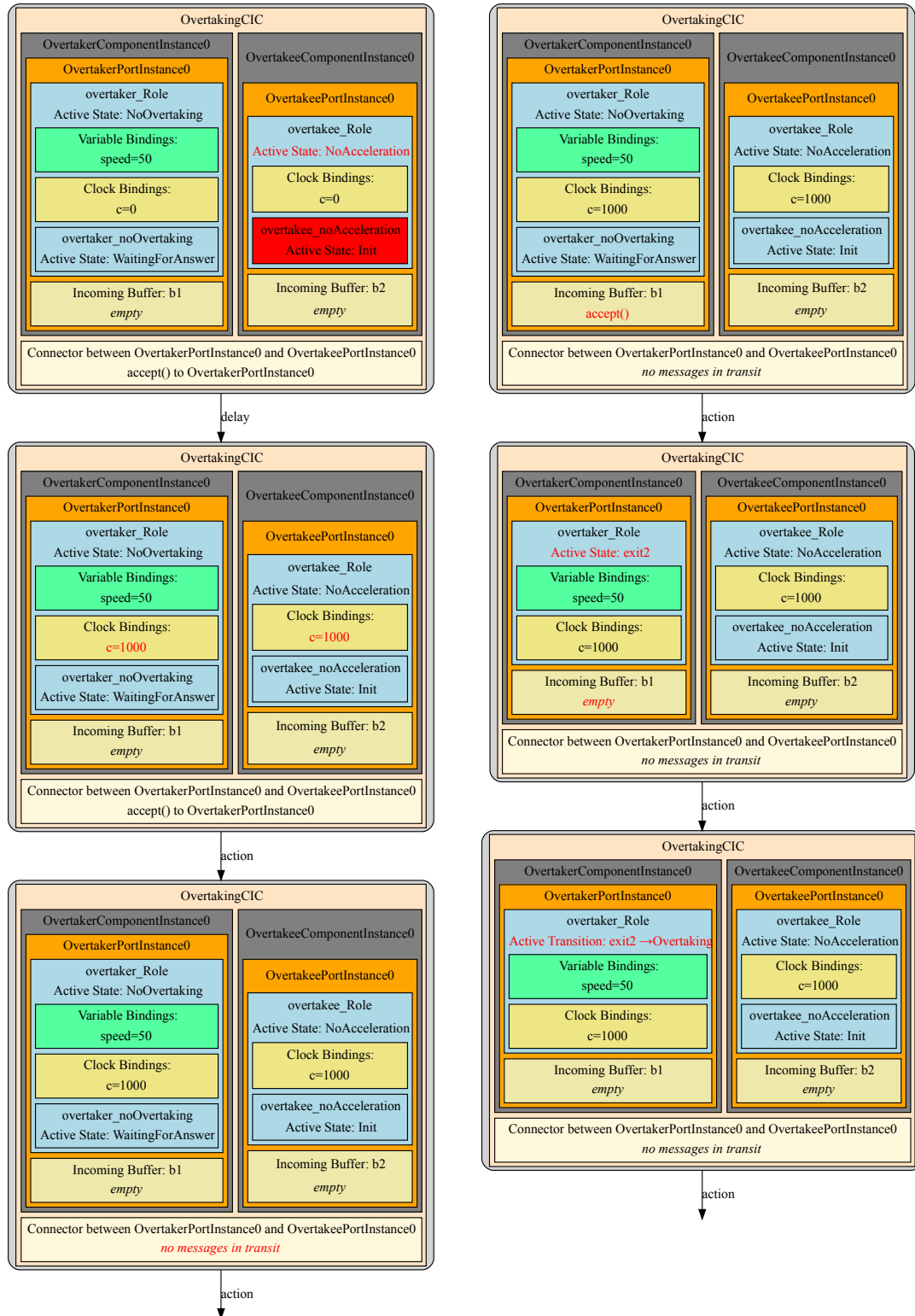


Figure D.13: Snapshots 13–18 of an Exemplary Counterexample Trace

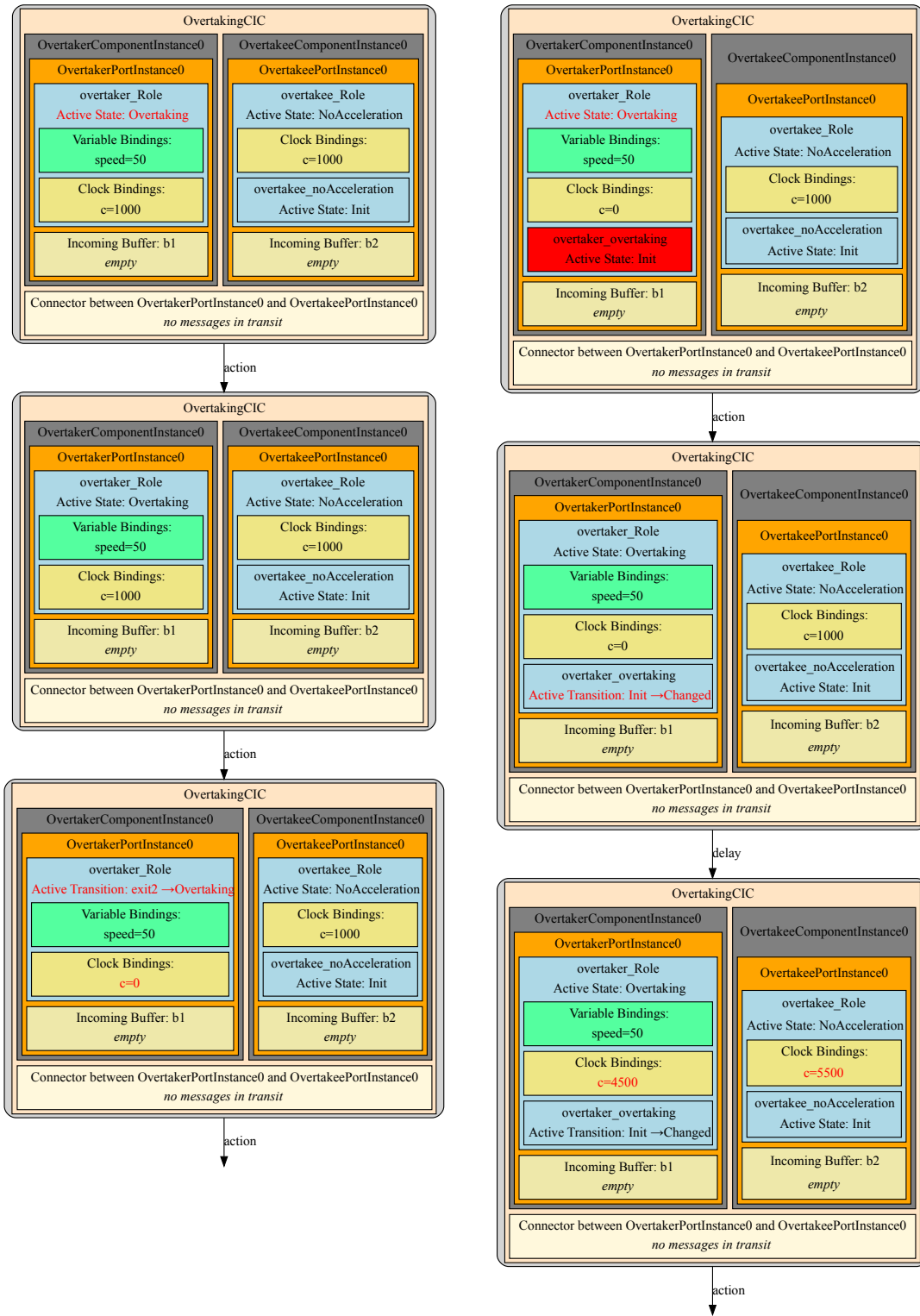


Figure D.14: Snapshots 19–24 of an Exemplary Counterexample Trace



Figure D.15: Snapshots 25–30 of an Exemplary Counterexample Trace

IMPLEMENTATION DETAILS

In this appendix, we show the metamodels and textual grammars that we defined during our implementation. They provide a formal definition of the abstract syntax and the static semantics of our modeling languages. We have specified the metamodels using EMF [EMF], the textual grammars using Xtext [Xtext], and the static semantics using OCL constraints [OCL] (the OCL constraints are contained within the metamodels). For our implementation, we used the Eclipse Neon.2 Modeling Tools. We illustrate our metamodels using class diagrams that we created with the class diagram editor of EcoreTools [EcoreTools]. Note that each diagram only shows inheritance relations to classes that are part of the same diagram. Therefore, a class may have more inheritance relations.

This appendix is structured as follows: First, in Section E.1, we present the metamodels for specifying RTCs (see Chapter 3). Afterwards, in Section E.2, we present the grammar and the metamodel of MTCTL (see Chapter 4). Next, in Section E.3, we present the grammar and the metamodels that we require to realize our DSMC for RTCs via UPPAAL (see Chapter 5). Finally, in Section E.4, we present our metamodel and our ontology for specifying and searching Real-Time Coordination Patterns (see Chapter 6).

E.1 RTCP

The metamodel of MECHATRONICUML for platform-independent modeling consists of several packages that are described in detail in [*DPP+16] (including the OCL constraints, the implementations of operations, and the definition of derived references and derived attributes). In this section, we only present the most important packages, classes, references, attributes, and OCL constraints that we require for specifying RTCs. Some of our explanations are based on [Hei15]. All (derived) attributes and references that start with the prefix “gmf” only exist within our metamodels as we require them for visualizing our models using Eclipse GMF.

The metamodels were defined by the authors of the MECHATRONICUML PIM specification [*BDG+11; *BBD+12; *BBB+12; *BDG+14b; *DPP+16]. All RTCP-specific aspects were mainly defined by me. An early draft of the package for one-to-many communication schemata (see Section E.1.8) has been defined in the master’s thesis of Bröggelwirth [Brö14], which I supervised. Afterwards, I improved her definitions.

E.1.1 Package Core

The classes of package core (see Figure E.1) are super classes for all classes in MECHATRONICUML's metamodel. The root element is the class `ExtendableElement` that provides—in combination with the class `Extension`—an extension mechanism. Each class that inherits from `ExtendableElement` may have arbitrary many extensions. Therefore, we enable that a metamodel stores additional information without modifying the existing classes.

An `ExtendableElement` has the subclasses `NamedElement` and `CommentableElement` that enable to define a name as well as a comment to an element. Furthermore, the class `Expression` is a subclass of `CommentableElement` that is the root class for MECHATRONICUML's action language (see Section E.1.4).

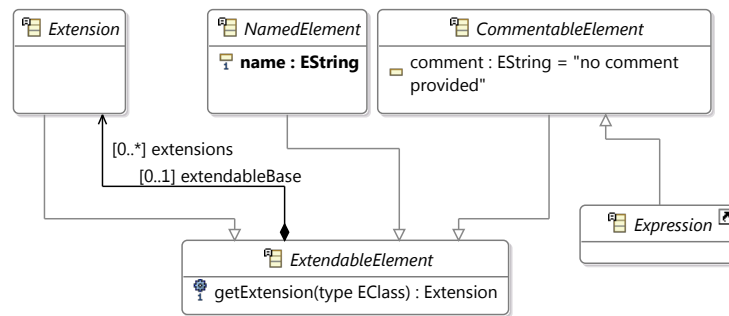


Figure E.1: Class Diagram of Package core of MECHATRONICUML's Metamodel

E.1.2 Package ValueType

The package `valuetype` (see Figure E.2) enables to specify different types of values that are used in the MECHATRONICUML metamodel. First of all, this package defines the class `NaturalNumber` that enables to specify a natural number that may even be infinite. Via an OCL constraint, we define that the value may not be negative. The class `Cardinality` enables to define a range of natural numbers while the class `Range` enables to define a range of long values. Via OCL constraints, we define that the lower value of a cardinality / range must be smaller or equal than the upper value. Moreover, this package also defines the class `TimeValue` that consists of a value expression (stated using the action language) and a time unit (based on Java's class `java.util.concurrent.TimeUnit`).

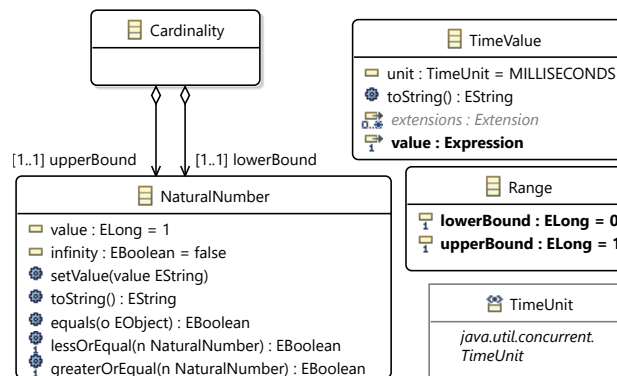
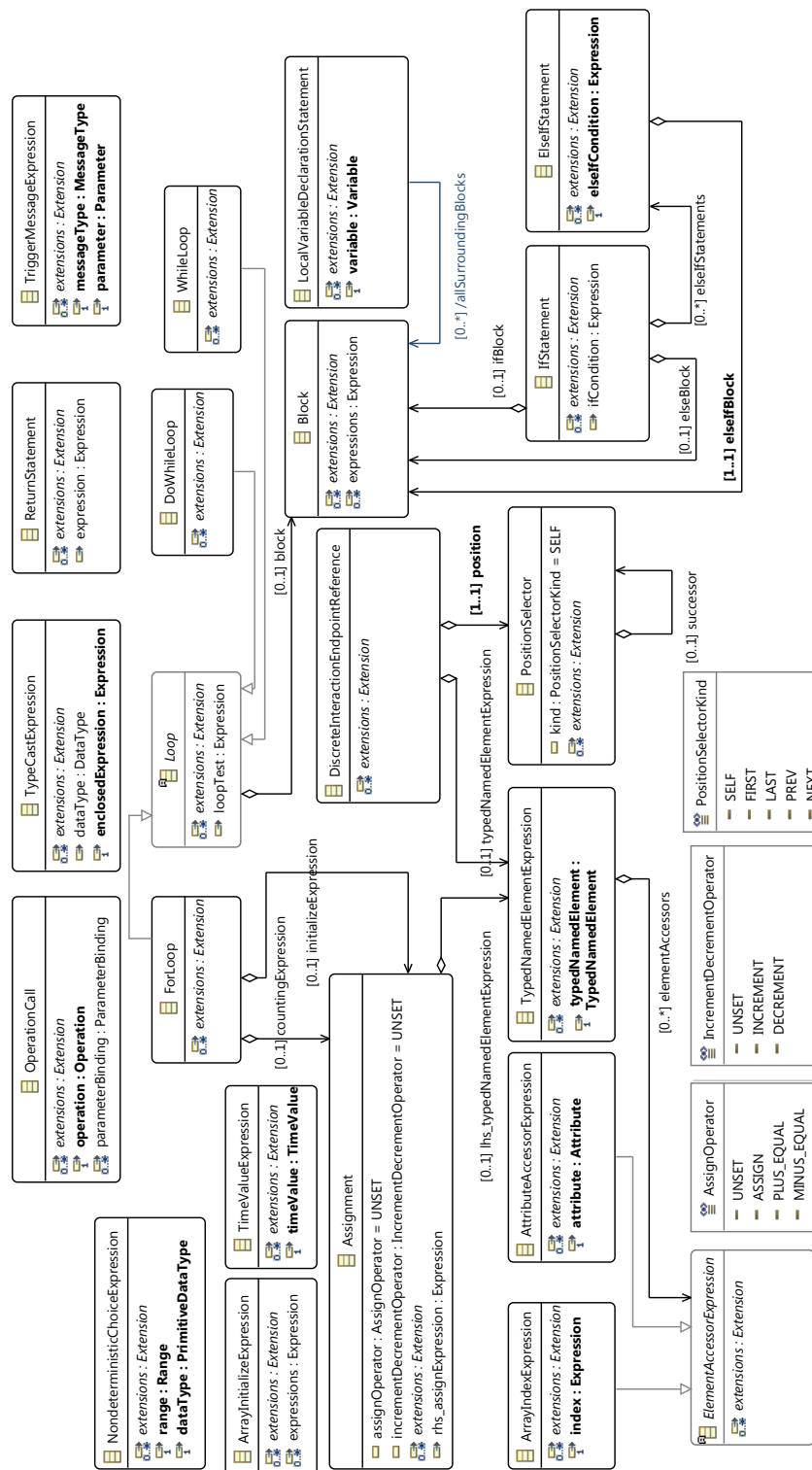


Figure E.2: Class Diagram of Package `valuetype` of MECHATRONICUML's Metamodel



Concerning the instantiation of discrete interaction endpoints, we provide two classes that inherit from `DiscreteInteractionEndpointInstance`: `DiscreteSingleInteractionEndpointInstance` and `DiscreteMultiInteractionEndpointInstance`. Via OCL constraints, we enforce that a discrete single interaction endpoint instance may only have a discrete single interaction endpoint as its type, if the discrete single interaction endpoint is not a multi endpoint. In contrary, a discrete multi interaction endpoint instance may only have a multi endpoint as its type. Furthermore, a discrete multi interaction endpoint instance may contain an ordered set of a subinteraction endpoint instances that are typed over the class `DiscreteSingleInteractionEndpointInstance`. As this set is ordered, a subinteraction endpoint instance knows the next and previous subinteraction endpoint instance. The discrete multi interaction endpoint instance knows the first and the last subinteraction endpoint instance of the set. Consequently, each subinteraction endpoint instance can derive the first and the last subinteraction endpoint instance of this set. Among others, the references `first`, `last`, `next`, and `prev` enable us to provide keywords for referring to subrole instances within a multi role RTSC that uses an explicit subrole RTSC (see Section 3.5.2.7).

The number of allowed subinteraction endpoint instances must be within the limits of discrete interaction endpoint's multiplicity. We enforce this via OCL constraints.

E.1.6 Packages Protocol and Constraint

The packages protocol and constraint (see Figure E.6) enable to specify an RTCP and to store its MTCTL properties.

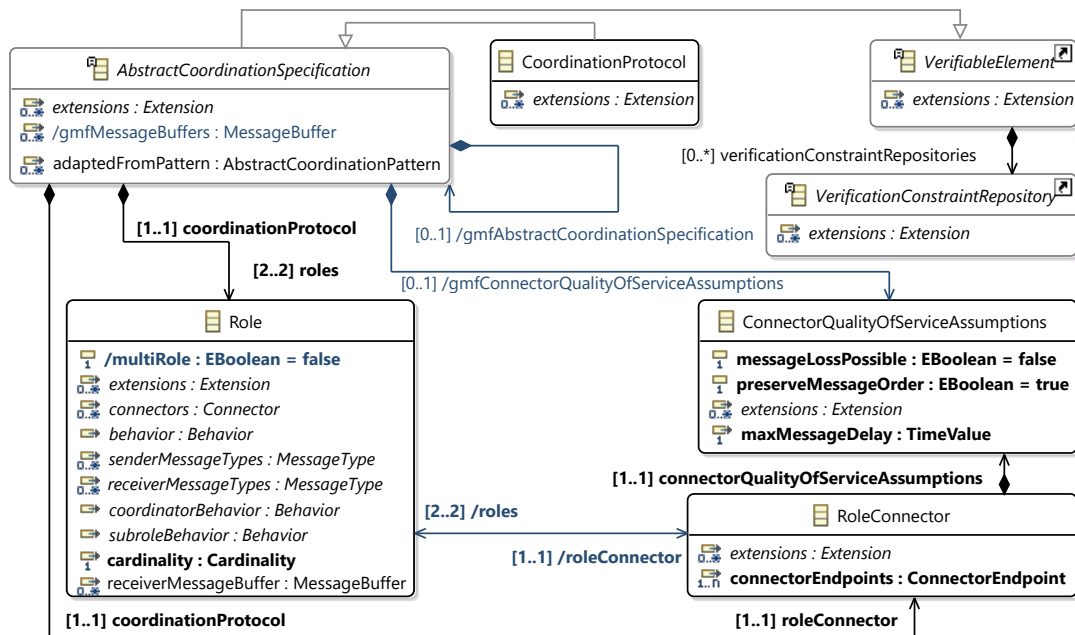


Figure E.6: Class Diagram of Packages protocol and constraint of MECHATRONICUML's Metamodel

The class `CoordinationProtocol` itself has no attributes or references but inherits them from the abstract class `AbstractCoordinationSpecification`, which defines the attributes and

references that an RTCP and a Real-Time Coordination Pattern (see Chapter 6 and E.4.1) have in common. In particular, the class `AbstractCoordinationSpecification` defines that an RTCP and a pattern have exactly two roles, one role connector, and may be adapted from one or more patterns. In addition, this class inherits from `VerifiableElement` and may, therefore, define verification constraint repositories (via class `VerificationConstraintRepository`). A verification constraint repository stores a list of MTCTL properties.

The class `Role` represents a role (see Section 3.3.1) and inherits from class `DiscreteInteractionEndpoint`, which inherits from the classes `NamedElement` (see Section E.1.1), `ConnectorEndpoint` (see Section E.1.5), and `BehavioralElement` (see Section E.1.3). Therefore, a role has a name, a connector, a multiplicity, may have a behavior (an RTSC), may send and receive messages, and may define an incoming message buffer. Using OCL constraints, we additionally define the following:

- The two roles must have different names.
- A role must have a behavior.
- A multi role requires that either all messages in the RTSC use one-to-many communication schemata or that the references `coordinatorBehavior` and `subroleBehavior` are set.
- Every role must have the `senderMessageTypes` of all the other role as its `receiverMessageTypes`.
- We forbid that a role can neither send nor receive messages. Therefore, it can receive, send, or send and receive messages.

The class `RoleConnector` represents a role connector (see Section 3.3.2) and inherits from class `Connector`. Therefore, it references connector endpoints. Via an OCL constraint, we define that the role connector may only connect roles and that both roles must be contained by the same RTCP. In addition, a role connector enables to specify the following explicit QoS assumptions: `messageLossPossible`, `preserveMessageOrder`, and `maxMessageDelay` (via class `ConnectorQualityOfServiceAssumptions`). The reference `messageLossPossible` is deprecated and will be replaced by a reference `reliable` in future version of the metamodel to be consistent with the concept of Section 3.3.2. If `messageLossPossible` is true, then `reliable` will be false and vice versa.

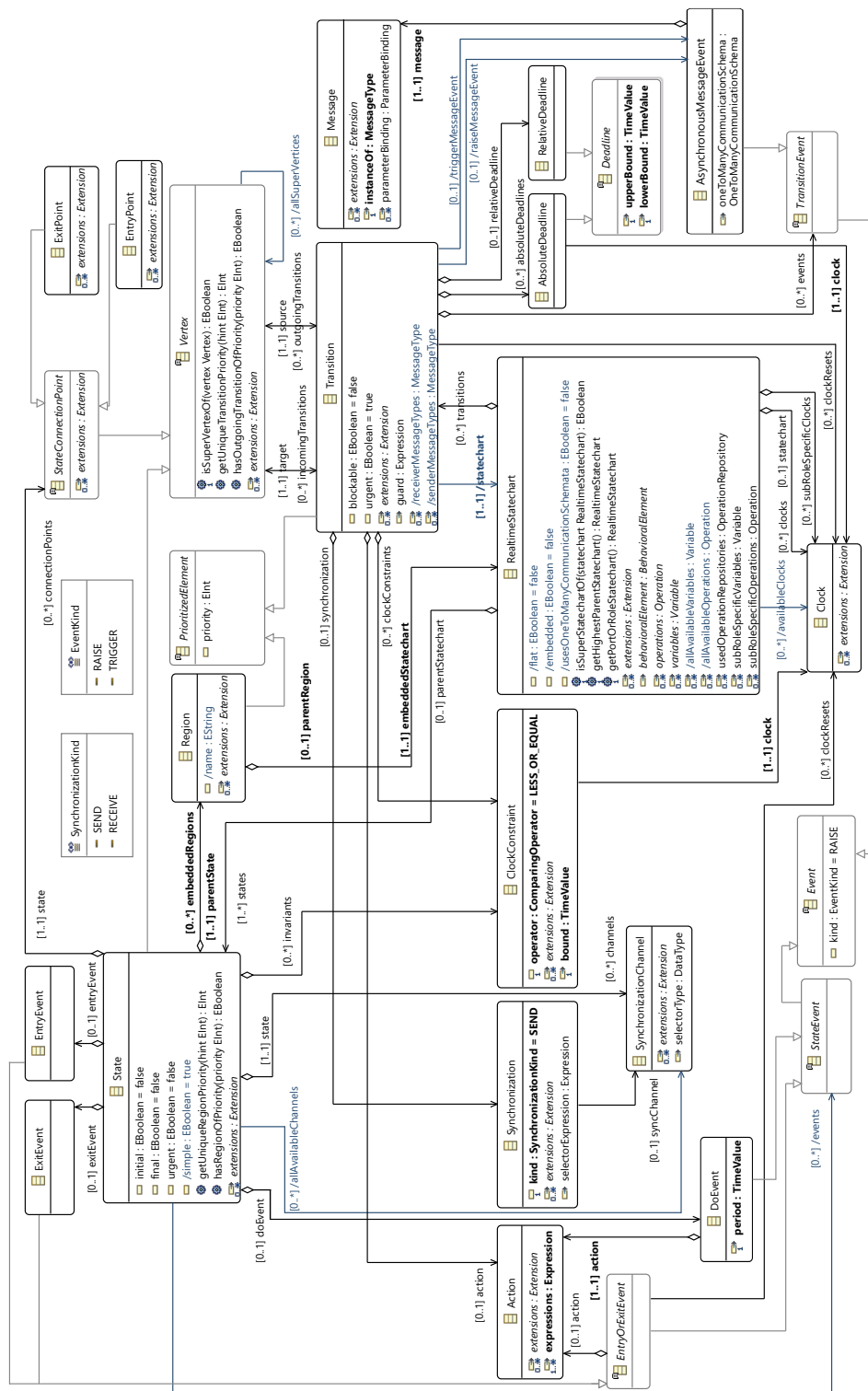
E.1.7 Package `Realtimestatechart`

The package `realtimestatechart` (see Figure E.7) enables to specify all aspects of an RTSC (see Section 2.2.2). The base class of this package is the class `RealtimeStatechart`. Among others, the class `Behavior` (see Section E.1.3) is a super type of this class. Therefore, an RTSC may define variables and operations. Additionally, an RTSC may define clocks and specifies the operation repositories that it will access. If the RTSC defines the behavior of a multi role with an explicit subrole, then the RTSC may also define subrole-specific variables, operations, and clocks. An RTSC may contain vertices (states or state connection points; via class `Vertex`) and transitions that connect the vertices with each other. The class `State` represents a state. It may be initial, final, and urgent. Furthermore, it may define invariants, synchronization channels, and entry-/do-/exit-actions. A state that is not simple embeds at least one region (via class `Region`). Each region contains exactly one RTSC. A state connection point (via class `StateConnectionPoint`) is either an entry or an exit point. A transition (class `Transition`) may

be urgent or non-urgent. It defines several conditions (a guard, a receiver message type, clock constraints, a priority, and a synchronization) and effects (an action, a sender message type, clock resets). Furthermore, a transition may either define a relative or an absolute deadline (via classes `Deadline`, `AbsoluteDeadline`, and `RelativeDeadline`).

Within this package, we use several OCL constraints. The most important ones are as follows (some of the explanations are taken or adapted from [*DPP+16]):

- An RTSC may not be the parent of the RTSC that it contains.
- An RTSC has exactly one initial state.
- Final states must not have outgoing transitions nor regions.
- The RTSC of a component behavior must contain exactly one state and no transitions at the top-level.
- The names of hybrid ports must differ from the variable names of the RTSC.
- A clock constraint that defines an invariant of a state may only use the operators `LESS` and `LESS_OR_EQUAL`.
- If a multi port/role RTSC has no explicit subrole, then each message that is sent or received requires a one-to-many communication schema.
- If a multi port/role RTSC has an explicit subrole, then it is not allowed to specify asynchronous message events in the coordinator region.
- If a multi port/role RTSC does not use one-to-many communication schemata, then it must contain exactly one state and no transitions on top-level. Furthermore, the top level state defines exactly two regions: one region defines the coordinator behavior, the other one defines the subrole behavior.
- When using one-to-many communication schemata, a trigger message event may only use receiving schemata and a raise message event may only use sending schemata.
- State connection points are only allowed at composite states.
- An entry point needs at least one incoming transition.
- All regions of the parent state must have exactly one vertex that the entry point connects to and one vertex that connects to the exit point.
- An exit point must have exactly one outgoing transition.
- A non-simple state may only have one region with transitions that have a trigger message event and connect (directly or indirectly) to the exit point.
- There must be at most one region with synchronizing transitions that connect (directly or indirectly) to the exit point.
- Inter-level transitions are invalid.
- A transition may only send and receive messages that the behavioral element (e.g., the role) defines.
- The trigger message event of a transition may not have parameters.
- A transition must not specify a received synchronization and a trigger message at the same time.
- A transition defines at most one raise message event and at most one trigger message event.
- Transitions to state connection points must not define side effects or deadlines.
- Transitions from state connection points must not have conditions but must be urgent.
- Defining both relative and absolute deadlines is forbidden.



E.1.8 Package One-To-Many Communication Schemata

The package `one_to_n_schemata` (see Figure E.8) enables to specify one-to-many communication schemata (see Section 3.5.2.1). For doing this, we provide the abstract class `OneToManyCommunicationSchema`. A one-to-many communication schema is specified for an asynchronous message event that is specified at a transition of an RTSC (see Section E.1.7). Each schema may define three schemata attributes: `condition`, `action`, and `retryAfter`. Though, we define via an OCL constraint that the attribute `retryAfter` may only be specified if and only if a condition is defined.

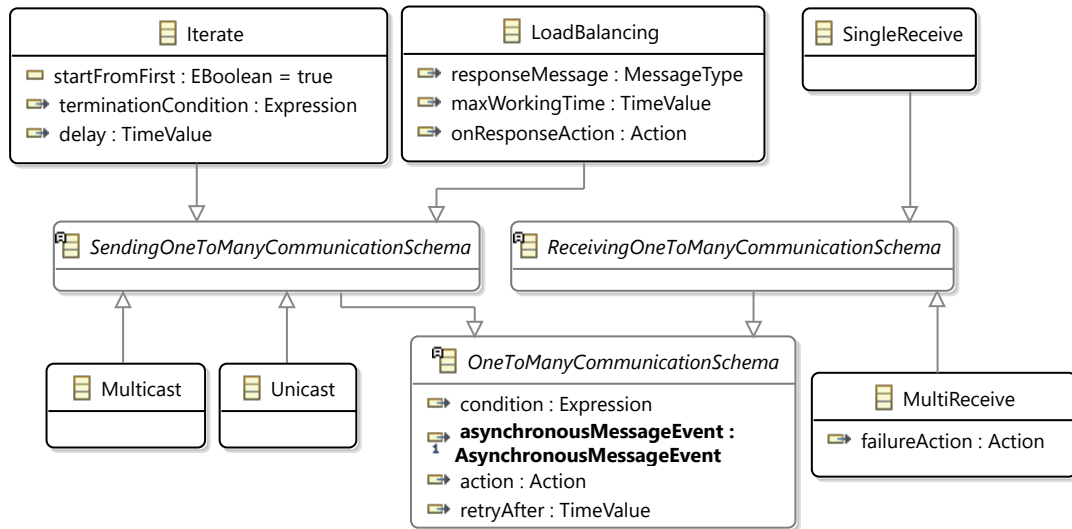


Figure E.8: Class Diagram of Package `one_to_n_schemata` of MECHATRONICUML's Metamodel

Two abstract classes inherit from `OneToManyCommunicationSchema`: `SendingOneToManyCommunicationSchema` and `ReceivingOneToManyCommunicationSchema`. The classes `Multicast`, `Unicast`, `Iterate`, and `LoadBalancing` inherit from `SendingOneToManyCommunicationSchema` as they all aim to send one or more messages. A multicast and a unicast do not provide additional schemata attributes (see Sections 3.5.2.2 and A.3). However, we enforce using an OCL constraint that a unicast must define the attributes `condition` and `retryAfter` (they are not optional anymore). The schema `iterate` additionally provides the optional schemata attributes `startFromFirst`, `terminationCondition`, and `delay` (see Section A.5). The schema `loadbalancing` provides the schemata attributes `maxWorkingTime`, `onResponseAction`, and `onResponseMessage` (see Section A.5). Using OCL constraints, we define (1) that either `onResponseMessage` or `maxWorkingTime` may be defined and (2) that if `onResponseAction` is set, then `onResponseMessage` must be set as well.

The classes `MultiReceive` and `SingleReceive` inherit from `ReceivingOneToManyCommunicationSchema` as they both aim to receive one or more messages. `SingleReceive` does not provide additional schemata attributes (see Section A.4) but `multiReceive` additionally provides the optional schemata attribute `failureAction` (see Section 3.5.2.2).

E.1.9 Package Component

The package component (see Figure E.9) enables to specify component types of MECHATRONICUML.

The root class for this package is the abstract class `Component`. A component may be a software component, which defines discrete behavior, a continuous component, or a hybrid component. Components define ports to communicate with each other. MECHATRONICUML separates between discrete, hybrid, and continuous ports. Furthermore, component types in MECHATRONICUML may be hierarchical. Therefore, a component type may be atomic (flat) or structured. If it is a structured component, then it may embed at least one component part (a component that is partially instantiated), which is typed over another component. A component part defines port parts (a port that is partially instantiated), which are typed over the port of the referenced component type.

A structured component has to define how its internal component parts and their port parts communicate with each other. The metamodel enables this by providing a port connector that defines which ports / port parts may interact with each other. We distinguish between assembly connectors and delegation connectors. Assembly connectors may connect ports of the same hierarchy level (i.e., they connect two or more port parts); delegation connectors connect ports of different hierarchy levels (i.e., they connect a port part and a port). Noteworthy, ports and port connectors inherit from the abstract class (Discrete-) `ConnectorEndpoint` and `Connector` of package connector.

In this thesis, we focus on the definition of RTCPs. They are applied within the component model as follows: Each discrete port has to refine a role of an RTCP¹. Consequently, each port part that is typed over a discrete port indirectly references a role as well. However, we still provide the class `CoordinationProtocolPart` to visualize the RTCP within our GMF editor. Concerning the message buffer specification, we enable that a discrete port may overwrite the role's incoming message buffers.

¹(We only raise an OCL warning and not an OCL error if no role is referenced as we have users that want to specify a non-safety-critical system with MECHATRONICUML. Typically, these users do not want to apply model checking. Then, specifying an RTCP is not mandatory anymore.

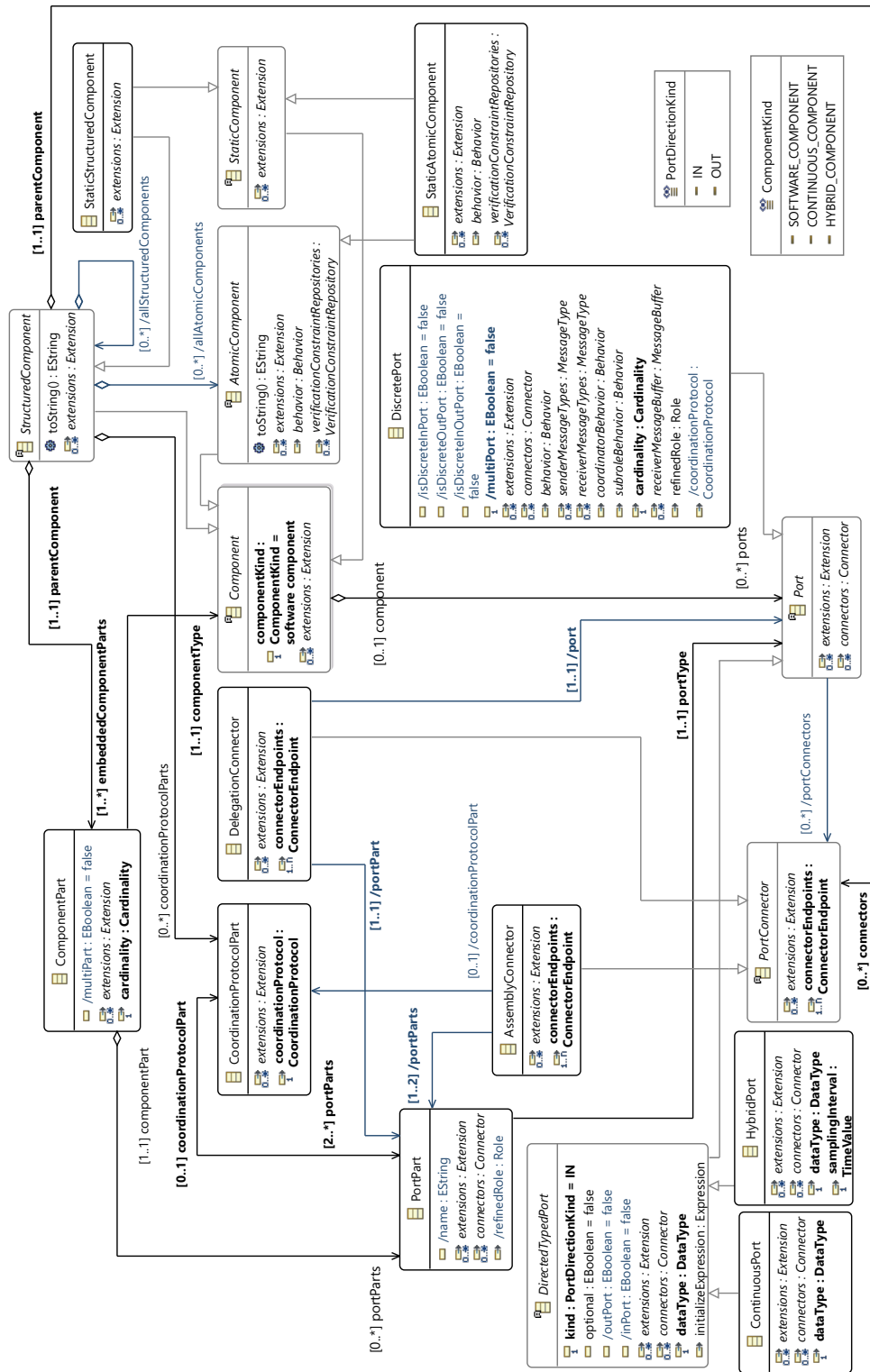


Figure E.9: Class Diagram of Package component of MECHATRONICUML's Metamodel

E.1.10 Package Instance

The package instance (see Figure E.10) enables to specify a component instance configuration (CIC) of MECHATRONICUML.

The root class for this package is `ComponentInstanceConfiguration`. A CIC contains instances of RTCPs (see Section E.1.6) as well as components, ports, and connectors (see Section E.1.9). Therefore, a CIC also contains instances of protocol parts, component parts, port parts, and connector parts.

A component instance may either be atomic or structured, where the latter one embeds another CIC. A port instance may be discrete, continuous, or hybrid. In particular, a discrete port instance is either a discrete single port instance or a discrete multi port instance. A role connector instance may be an assembly connector instance or a delegation connector instance. A delegation connector instance is always typed over a delegation connector. However, an assembly connector instance may be typed over an assembly connector but does not have to (if the assembly connector instance connects two port instances directly instead of two port parts). Noteworthy, port instances and port connector instances inherit from the abstract class (Discrete-) `ConnectorEndpoint` and `Connector` of package connector.

In this thesis, we focus on the definition of RTCPs. They are applied within a CIC as follows: Each discrete port instance is (indirectly) typed over a discrete port, which has to refine a role of an RTCP. The port instances that are connected via assembly connector instances belong to the same coordination protocol instance. Noteworthy, we only provide the class `CoordinationProtocolInstance` to visualize it within our GMF editor. Concerning the specification of QoS assumptions, we enable that an assembly connector instance may overwrite the QoS assumptions of the role connector.



E.2 MTCTL

In the following, we provide the metamodel of MTCTL (see Chapter 4) in Section E.2.1 and our Xtext grammar in Section E.2.2. Both the metamodels and the grammar were defined by the project group Cybertron [BCD+14] and me.

E.2.1 Metamodel

The core classes of MTCTL are stored within the package `mtctl` (see Figure E.11). The class `PropertyRepository` enables to store a list of MTCTL properties. Each property is expressed using the class `Property`. A property defines an expression, where the class `Expression` is the common super type of all elements of the language (except `Property` and `PropertyRepository` and the classes of package `result`).

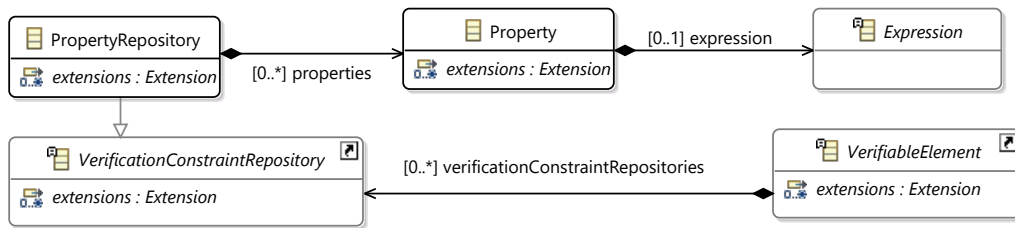


Figure E.11: Class Diagram of Package `mtctl` of MTCTL's Metamodel

The package `booleanlogic` (see Figure E.12) defines the four logical operators that MTCTL supports: **and**, **implies**, **not**, and **or**. They support to reference all expressions of MTCTL.



Figure E.12: Class Diagram of Package `booleanlogic` of MTCTL's Metamodel

The package `predicates` (see Figure E.13) defines all predicates of MTCTL (see Section 4.3.5). The base class of this package is class `PredicateExpr`. A predicate may either be dynamic, static, or a comparison expression. We define seven dynamic predicates: **deadlock**, **connectorOverflow**, **messageInBuffer**, **messageInTransit**, **stateActive**, **transitionFiring**, and **messageDiscarded**. Moreover, we define four static predicates **true**, **false**, **substateOf**, and **stateInStatechart**. MTCTL supports six kinds of comparisons: equals, greater, greater or equal, less, less or equal, and not equal.

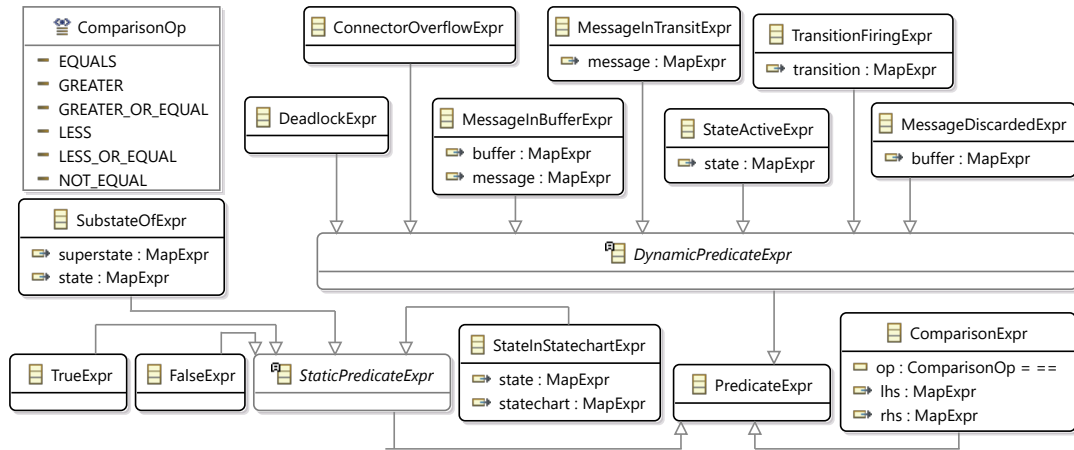


Figure E.13: Class Diagram of Package predicates of MTCTL's Metamodel

The package comparables (see Figure E.14) contains expressions that can be used in a `ComparisonExpr`. The class `MapExpr` is an abstract superclass for expressions that evaluate to some value. Comparables may be the three supported operations (`bufferMessageCount`, `sourceState`, and `targetState`) as well as a constant value and a reference to an element of the design model (via class `MumlElemExpr`).

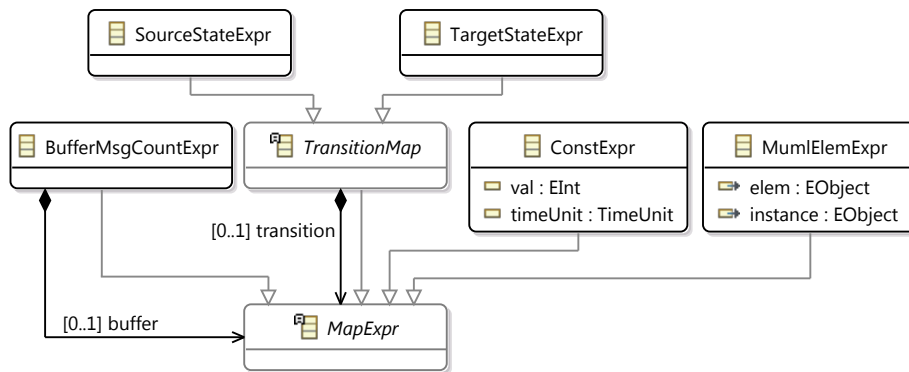


Figure E.14: Class Diagram of Package comparables of MTCTL's Metamodel

The package quantifiers (see Figure E.15) defines the temporal and set quantifiers of MTCTL. We define five temporal quantifiers: **AG**, **AF**, **EF**, **EG**, and **leadsTo**. Moreover, we define two set quantifiers (**forall** and **exists**) that both define a bounded variable and a set.

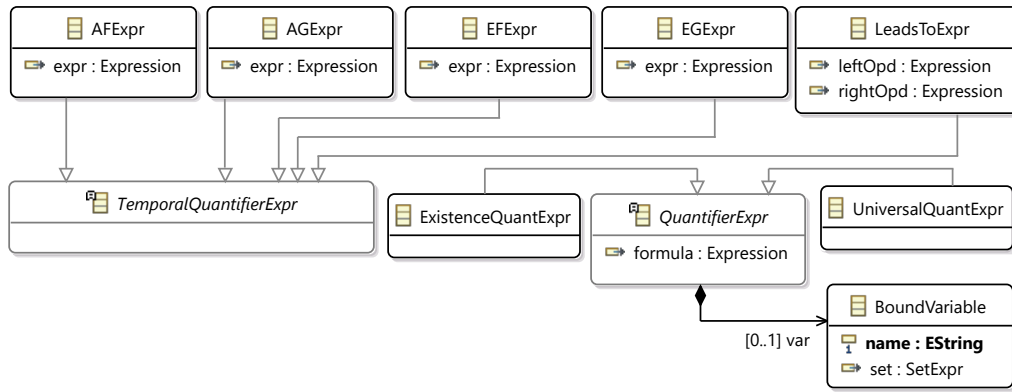


Figure E.15: Class Diagram of Package quantifiers of MTCTL's Metamodel

The package sets (see Figure E.16) defines the possible sets that a set quantifier supports. In particular, it supports eight sets: **Clocks**, integers intervals, **MessageTypes**, **States**, **Transitions**, **Buffers**, **Instances**, and **Subinstances**.

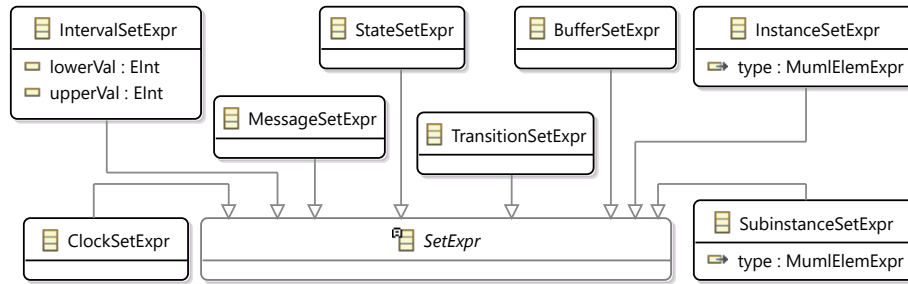


Figure E.16: Class Diagram of Package sets of MTCTL's Metamodel

The package results (see Figure E.17) is not used for specifying an MTCTL property. Instead, we require it to store the verification result of a list of MTCTL properties. In particular, the class **PropertyResultRepository** is the root class that contains all results. Each verification result is stored in class **PropertyResult** (we store if the property is fulfilled and the property itself). Moreover, due to our MTCTL split property normalization (see Section 5.4.4), it might be the case that we have to split a property into several subproperties. We store them also using the class **PropertyResult**. They are contained within their parent property.

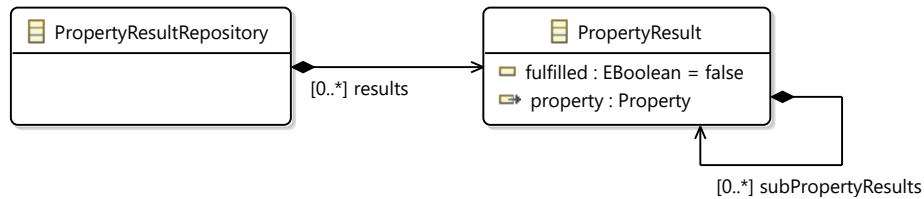


Figure E.17: Class Diagram of Package results of MTCTL's Metamodel

E.2.2 Grammar

In the following, we list the grammar of MTCTL as it is specified in Xtext. The grammar references the MTCTL metamodel (see Section E.2.1) and the package valuetype (see

Section E.1.2) of MECHATRONICUML's metamodel to express time units. We provide an explanation for a simplified variant of this grammar in Section 4.3.1.

Please note that additional language specifications exist, which we had to specify in Java due to the limitations of Xtext. For example, we do not directly define within our grammar that nesting of temporal quantifiers is forbidden.

```
1 grammar org.muml.uppaal.adapter.mtctl.xtext.Mtctl with org.eclipse.xtext.
   common.Terminals
2
3 import "platform:/resource/org.muml.uppaal.adapter.mtctl/model/Mtctl.ecore
   " as mtctl
4 import "platform:/resource/org.muml.uppaal.adapter.mtctl/model/Mtctl.ecore
   ///booleanlogic" as mtctl
5 import "platform:/resource/org.muml.uppaal.adapter.mtctl/model/Mtctl.ecore
   ///comparables" as mtctl
6 import "platform:/resource/org.muml.uppaal.adapter.mtctl/model/Mtctl.ecore
   ///predicates" as mtctl
7 import "platform:/resource/org.muml.uppaal.adapter.mtctl/model/Mtctl.ecore
   ///quantifiers" as mtctl
8 import "platform:/resource/org.muml.uppaal.adapter.mtctl/model/Mtctl.ecore
   ///sets" as mtctl
9
10 import "platform:/resource/org.muml.pim/model/pim.ecore ///valuetype" as
   valuetype
11 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
12
13 PropertyRepository returns mtctl::PropertyRepository: (properties+=
   Property)*;
14
15 Property returns mtctl::Property hidden (WS, ML_COMMENT): expression=
   Expression ';' (comment=SL_COMMENT)?;
16
17 Expression returns mtctl::Expression: LeadsToExpr;
18
19 //Binary operators (increasing precedence)
20 LeadsToExpr returns mtctl::Expression:
21   ImplExpr ({ mtctl::LeadsToExpr.leftOpd=current } 'leadsTo' rightOpd=
   ImplExpr)*;
22
23 ImplExpr returns mtctl::Expression:
24   OrExpr ({ mtctl::ImplExpr.leftOpd=current } 'implies' rightOpd=OrExpr)*;
25
26 OrExpr returns mtctl::Expression:
27   AndExpr ({ mtctl::OrExpr.leftOpd=current } 'or' rightOpd=AndExpr)*;
28
29 AndExpr returns mtctl::Expression:
30   NotExpr ({ mtctl::AndExpr.leftOpd=current } 'and' rightOpd=NotExpr)*;
31
32 //Unary operators
33 NotExpr returns mtctl::Expression: 'not' { mtctl::NotExpr } opd=NotExpr |
   QuantifierExpr;
34
35 QuantifierExpr returns mtctl::Expression: UniversalQuantExpr |
   ExistentialQuantExpr | TemporalQuantifierExpr | AtomExpr;
36
```

```

37 UniversalQuantExpr returns mtctl::QuantifierExpr: 'forall' '(' {mtctl::
    UniversalQuantExpr} var=VariableBinding ')' formula=(NotExpr);
38
39 ExistentialQuantExpr returns mtctl::QuantifierExpr: 'exists' '(' {mtctl::
    ExistenceQuantExpr} var=VariableBinding ')' formula=(NotExpr);
40
41 VariableBinding returns mtctl::BoundVariable: name=ID ':' set=SetExpr;
42
43 TemporalQuantifierExpr returns mtctl::Expression: EFExpr | AFExpr | EGExpr
    | AGExpr;
44
45 EFExpr returns mtctl::TemporalQuantifierExpr: ('EF' | 'E<>') {mtctl::
    EFExpr} expr=NotExpr;
46
47 AFExpr returns mtctl::TemporalQuantifierExpr: ('AF' | 'A<>') {mtctl::
    AFExpr} expr=NotExpr;
48
49 EGExpr returns mtctl::TemporalQuantifierExpr: ('EG' | 'E[]') {mtctl::
    EGExpr} expr=NotExpr;
50
51 AGExpr returns mtctl::TemporalQuantifierExpr: ('AG' | 'A[]') {mtctl::
    AGExpr} expr=NotExpr;
52
53 //Bottom of precedence chain
54 AtomExpr returns mtctl::Expression: '(' Expression ')' | PredicateExpr |
    ComparisonExpr;
55
56 //Predicates
57 PredicateExpr returns mtctl::Expression: TrueExpr | FalseExpr |
    DeadlockExpr | ConnectorOverflowExpr | StateExpr | MessageExpr |
    TransitionExpr;
58
59 DeadlockExpr returns mtctl::DeadlockExpr: {mtctl::DeadlockExpr} 'deadlock'
    ;
60
61 ConnectorOverflowExpr returns mtctl::ConnectorOverflowExpr: {mtctl::
    ConnectorOverflowExpr} 'connectorOverflow';
62
63 TrueExpr returns mtctl::TrueExpr: {mtctl::TrueExpr} 'true';
64
65 FalseExpr returns mtctl::FalseExpr: {mtctl::FalseExpr} 'false';
66
67 StateExpr returns mtctl::PredicateExpr: StateActiveExpr | SubstateOfExpr |
    StateInStatechartExpr;
68
69 StateActiveExpr returns mtctl::StateActiveExpr: 'stateActive' '(' state=
    StateMapExpr ')';
70
71 SubstateOfExpr returns mtctl::SubstateOfExpr: 'substateOf' '(' state=
    StateMapExpr ',' superstate=StateMapExpr ')';
72
73 StateInStatechartExpr returns mtctl::StateInStatechartExpr: '
    stateInStatechart' '(' state = StateMapExpr ',' statechart =
    StatechartMapExpr ')';
74

```

```
75 MessageExpr returns mtctl::PredicateExpr: MessageInBufferExpr |
    MessageInTransitExpr | MessageDiscardedExpr;
76
77 MessageInTransitExpr returns mtctl::MessageInTransitExpr: '
    messageInTransit' '(' message=MessageMapExpr ')';
78
79 MessageInBufferExpr returns mtctl::MessageInBufferExpr: 'messageInBuffer'
    '(' message=MessageMapExpr ',' buffer=BufferMapExpr ')';
80
81 MessageDiscardedExpr returns mtctl::MessageDiscardedExpr: '
    messageDiscarded' '(' buffer = BufferMapExpr ')';
82
83 TransitionExpr returns mtctl::PredicateExpr: TransitionFiringExpr;
84
85 TransitionFiringExpr returns mtctl::TransitionFiringExpr: '
    transitionFiring' '(' transition=TransitionMapExpr ')';
86
87 //Comparisons
88 ComparisonExpr returns mtctl::Expression: { mtctl::ComparisonExpr } lhs=
    MapExpr op=ComparisonOp rhs=MapExpr;
89
90 enum ComparisonOp returns mtctl::ComparisonOp: EQUALS='==' | GREATER='>' |
    GREATER_OR_EQUAL='>=' | LESS='<' | LESS_OR_EQUAL='<=' | NOT_EQUAL='!='
    ;
91
92 //Expressions usable in comparisons. Starting with MapExpressions arranged
    by return type
93 MapExpr returns mtctl::MapExpr: MumlElemExpr | BufferMessageCountExpr |
    ConstExpr | SourceStateExpr | TargetStateExpr;
94
95 IntegerMapExpr returns mtctl::MapExpr: MumlElemExpr |
    BufferMessageCountExpr | ConstExpr;
96
97 TransitionMapExpr returns mtctl::MapExpr: MumlElemExpr;
98
99 StateMapExpr returns mtctl::MapExpr: MumlElemExpr | SourceStateExpr |
    TargetStateExpr;
100
101 StatechartMapExpr returns mtctl::MapExpr: MumlElemExpr;
102
103 BufferMapExpr returns mtctl::MapExpr: MumlElemExpr;
104
105 MessageMapExpr returns mtctl::MapExpr: MumlElemExpr;
106
107 BufferMessageCountExpr returns mtctl::BufferMsgCountExpr: '
    bufferMessageCount' '(' buffer=BufferMapExpr ')';
108
109 SourceStateExpr returns mtctl::SourceStateExpr: 'sourceState' '('
    transition=TransitionMapExpr ')';
110
111 TargetStateExpr returns mtctl::TargetStateExpr: 'targetState' '('
    transition=TransitionMapExpr ')';
112
113 MumlElemExpr returns mtctl::MumlElemExpr: elem=[ecore::EObject|
    QualifiedName] ('[' instance=[ecore::EObject|QualifiedNames] ']' )?;
114
```

```

115 ConstExpr returns mtctl::ConstExpr: val=EInt (timeUnit=TimeUnitExpr)?;
116
117 TimeUnitExpr returns valuetype::TimeUnit: 'days' | 'hrs' | 'mins' | 'secs'
    | 'msecs' | 'usecs' | 'nsecs';
118
119 //Sets
120 SetExpr returns mtctl::SetExpr: InstanceSetExpr | SubinstanceSetExpr |
    IntervalSetExpr | {mtctl::StateSetExpr} 'States' | {mtctl::
    TransitionSetExpr} 'Transitions' | {mtctl::MessageSetExpr} '
    MessageTypes' | {mtctl::ClockSetExpr} 'Clocks' | {mtctl::BufferSetExpr}
    'Buffers';
121
122 IntervalSetExpr returns mtctl::SetExpr: {mtctl::IntervalSetExpr} '['
    lowerVal=INT ',' upperVal=INT ']';
123
124 InstanceSetExpr returns mtctl::SetExpr: {mtctl::InstanceSetExpr} '
    Instances' '<' type=MumlElemExpr '>';
125
126 SubinstanceSetExpr returns mtctl::SetExpr: {mtctl::SubinstanceSetExpr} '
    Subinstances' '<' type=MumlElemExpr '>';
127
128 //Other necessary definitions
129 QualifiedName: QNAME | ID;
130
131 terminal QNAME: ID ( '.' ID )+;
132
133 EInt returns ecore::EInt: '-'? INT;

```

E.3 DSMC of RTCPs via Uppaal

For realizing our DSMC for RTCPs via UPPAAL (see Chapter 5), we define and use several metamodels that we will present in the following. In particular, we start with presenting our metamodel for defining the verification options in Section E.3.1. We continue with presenting a metamodel that extends the MECHATRONICUML model with additional information in Section E.3.2. Then, we present our metamodels concerning UPPAAL timed automata in Section E.3.3 and UTCTL in Section E.3.4. Afterwards, we present our textual grammar for UPPAAL traces in Section E.3.5 and the resulting metamodel in Section E.3.6. Finally, we present the metamodel for MECHATRONICUML traces in Section E.3.7.

The metamodel for the DSMC options has been defined by the project group Cybertron [BCD+14], which I supervised, and by me. The metamodel for UPPAAL timed automata is based on the UPPAAL Tiga metamodel by Greenyer [Gre11]. Gerking [Ger13] significantly extended this metamodel. Moreover, he initially defined the metamodels for MECHATRONICUML extensions, UTCTL, UPPAAL traces, and the grammar for UPPAAL traces. All his works were produced within his master's thesis, which was supervised by me. Later on, the project group Cybertron and I improved the metamodels of UPPAAL (timed automata, UTCTL, trace) and the MECHATRONICUML extensions as well as the grammar for traces. The metamodels for MECHATRONICUML traces were initially defined by Heinzemann [*HBDS15; Hei15] to model counterexamples of MECHATRONICUML's refinement check. Later on, the project group Cybertron and I enriched these models with

small details to additionally model (counterexample) traces of RTCPs resp. of component instance configurations.

E.3.1 DSMC Options Metamodel

In Section 5.7.1, we show in Figure 5.26 a wizard, where the DSMC user may define the verification options. We collect all options within one metamodel, which we show in Figure E.18.

The class Options contains the general options to verify a verifiable element (i.e., an RTCP, a discrete component, or a CIC). These options are the size of the connector buffer, the hash table size, the state space reduction strategy, and the trace option. If a one-to-many RTCP shall be verified, then the DSMC user may also define a role multiplicity. We enable to hide this option if a one-to-one RTCP shall be verified using the class HideOptionExtension (the attribute option has to name the option that shall be hidden).

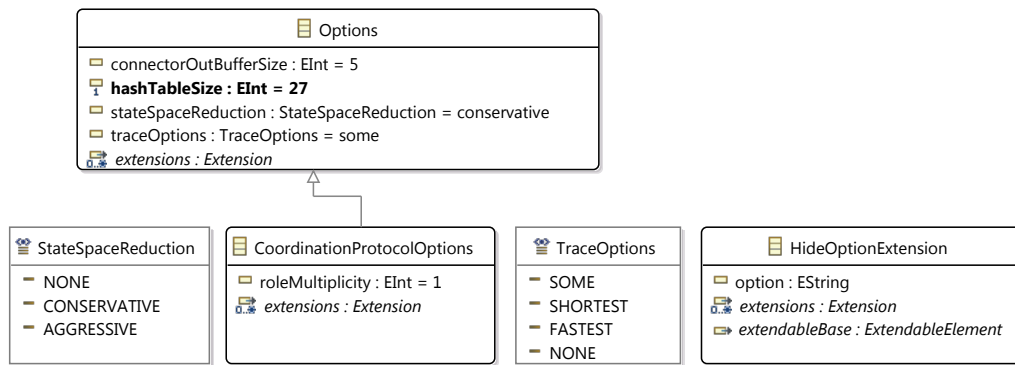


Figure E.18: Class Diagram of the DSMC-Options Metamodel

E.3.2 MechatronicUML Verification Extension Metamodel

The package verificationextension (see Figure E.19) provides several classes that extend existing classes of MECHATRONICUML's metamodels for specifying an RTCP or a trace:

- The class CommittedExtension enables us to declare a state of an RTSC as committed (we do not enable that the RTCP specifier may declare it but we require it for our verification).
- The class ElementToVerifyExtension enables us to mark a verifiable element (e.g., an RTCP) that it shall be verified. All other verifiable elements of the MECHATRONICUML model will not be verified and, therefore, not transformed to UPPAAL.
- The classes ClockBindingsExtension and ClockBinding enable us to bind concrete values to all clocks within MECHATRONICUML's trace. Concrete clock values appear within concrete traces, which we use for MECHATRONICUML counterexamples. As the runtime metamodel (see Section E.3.7) focuses on symbolic traces, we realized to store this information using an extension.

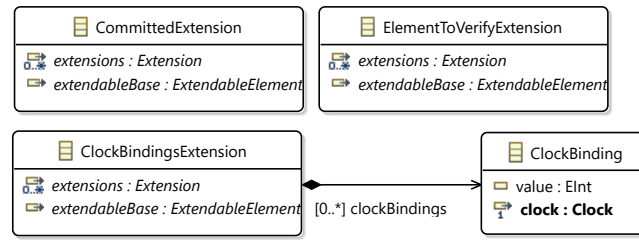
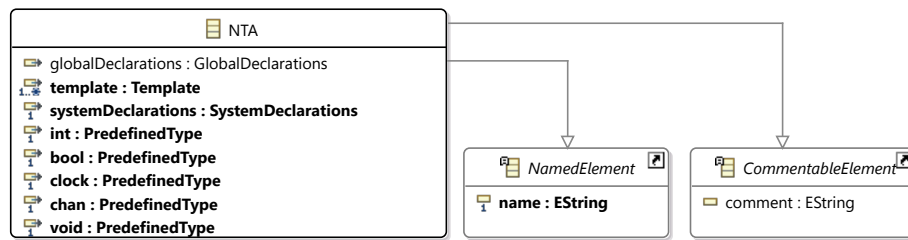


Figure E.19: Class Diagram of the DSMC-Options Metamodel

E.3.3 Uppaal Timed Automata Metamodel

The package `uppaal` (see Figure E.20) defines the class for a network of timed automata (class `NTA`). An `NTA` may define global declarations, templates, and must define one system declaration. Moreover, this class defines five predefined types: `int`, `bool`, `clock`, `chan`, and `void`. The class `NTA` inherits from the classes `NamedElement` and `CommentableElement`. These classes are contained in the package `core` and enable to add a name and a comment to an `NTA`. We use both classes for several other classes within our UPPAAL metamodel. We depict both classes in Figure E.20 as well.

Figure E.20: Class Diagram of Packages `uppaal` and `core` of UPPAAL's Metamodel

The package `declarations` (see Figure E.21) enables to specify declarations. The base class is the abstract class `Declarations`. Three kinds of declarations exist: local, global, and system declarations. An `NTA` may define a set of global declarations and a set of system declarations while a timed automaton may define local declarations. The class `Declarations` may define multiple instances of class `Declaration`. A declaration may be a variable declaration, a type declaration (see package `types`), and a function declaration. A variable declaration may declare channel variables, clock variables, and data variables.

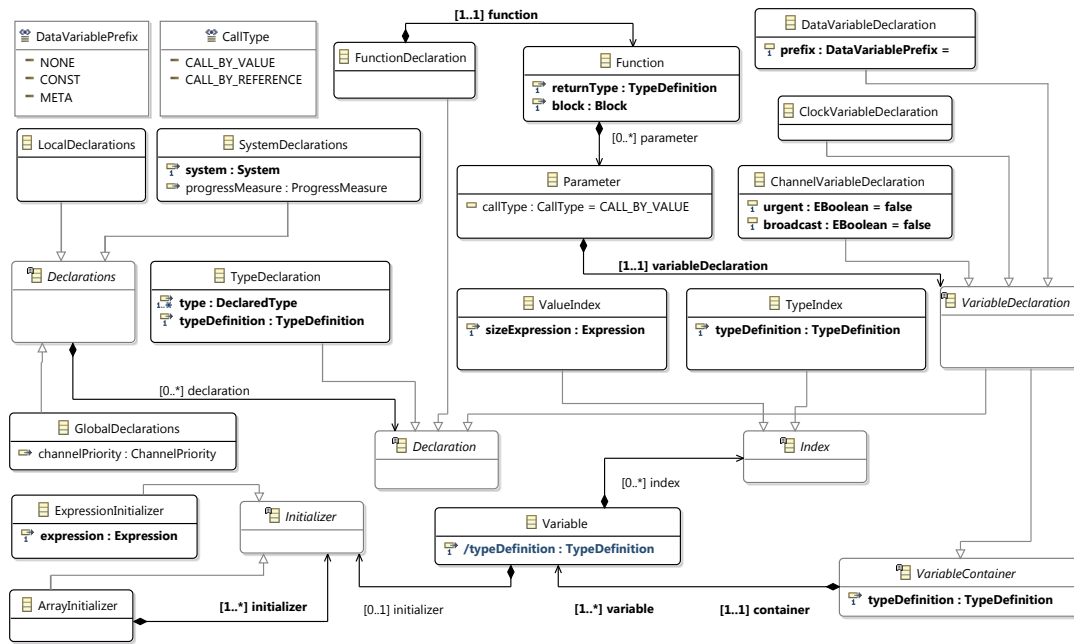


Figure E.21: Class Diagram of Package declarations of UPPAAL's Metamodel

The package global (see Figure E.22) defines classes that are relevant for the global declaration. The class ChannelList enables to specify the list of available channel while the class ChannelPriority enables to define a channel priority list that consists of one or more channel priority items. While specifying channel priorities, the class DefaultChannelPriority enables to state that the respective channel has a default priority.

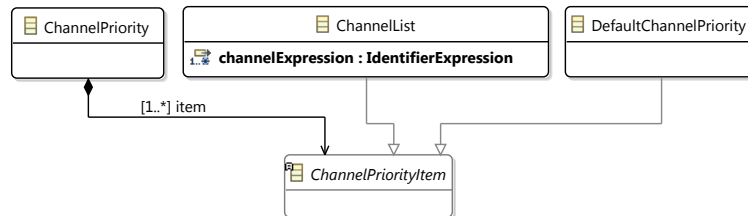


Figure E.22: Class Diagram of Package global of UPPAAL's Metamodel

The package system (see Figure E.23) enables to specify the parts of a system declaration (the class SystemDeclaration is specified in package declarations). A system declaration consists of a declaration, a system, and a progress measure:

- A system declaration may declare all available declarations. Among others, it may specify a template declaration that redefines an existing template by altering its parameters and its name.
- A system defines one or more instantiation lists. Each list defines a set of (redefined) templates that shall be instantiated. The order of the instantiation lists is very important as they define the priority of the processes: all processes of one instantiation list have the same priority but the processes of different instantiation lists have different priorities. An instantiation list that is ordered before another list also has a higher priority.

- A progress measure contains a list of expressions. Each expression typically refers to variables that indicate a progress in the model, which helps UPPAAL to reduce the memory usage.

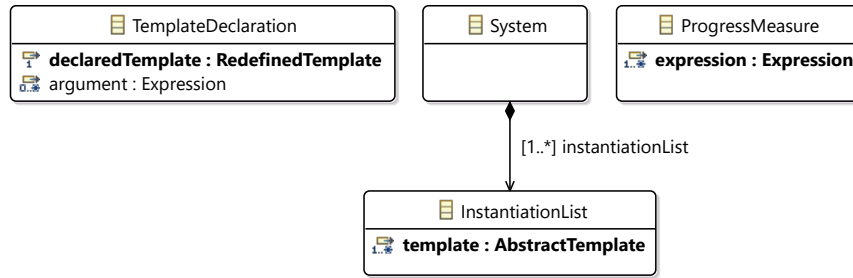


Figure E.23: Class Diagram of Package system of UPPAAL's Metamodel

The package statements (see Figure E.24) enables to specify a statement. UPPAAL supports nine different statements: blocks, empty statements, for loops, iterations, while loops, do while loops, if-else statements, return statements, and expression statements. Some of the statements define one or more expressions. The classes for specifying an expression are specified in the package expressions.

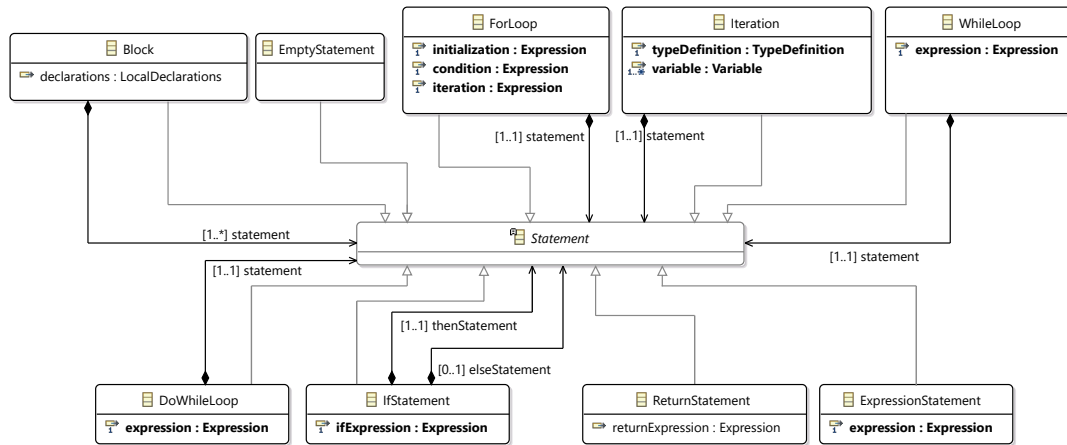


Figure E.24: Class Diagram of Package statements of UPPAAL's Metamodel

The package templates (see Figure E.25) enables to specify a template of an UPPAAL timed automaton. The abstract base class of this package is **AbstractTemplate**. It defines that a template may have parameters. Two classes inherit from this class: **RedefinedTemplate** and **Template**. A redefined template may redefine another template by altering its name and its parametrization (UPPAAL often refers to a redefined template as a partial instantiation).

A template may define a set of local declarations. It consists of at least one location (one of them must be the initial location) and may contain several edges. A location may be urgent, committed, or none of both. Furthermore, it may define an invariant and has a position. An edge connects two locations. It may have bend points and may define a guard, an update, a synchronization, and a selection.

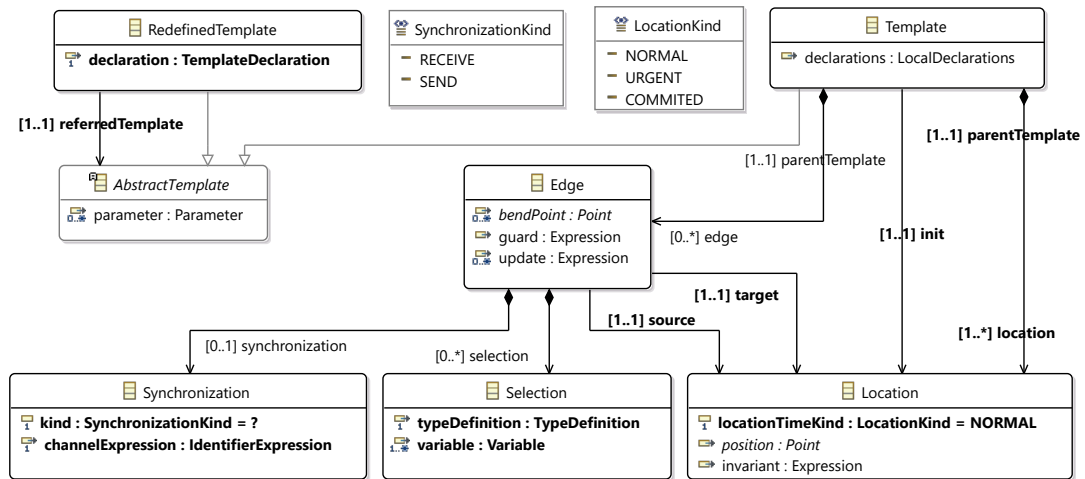


Figure E.25: Class Diagram of Package templates of UPPAAL's Metamodel

The package types (see Figure E.26) defines the types of an NTA. UPPAAL provides five pre-defined types: int, clock, chan, bool, and void. Moreover, UPPAAL enables to define additional types that may be based on the pre-defined types. In particular, it enables to define arrays, a range of integers, structs (records), and scalars (a scalar is similar to an unordered list). Furthermore, this package provides the class `TypeReference` that enables to reference a specific type, e.g., within a type definition.

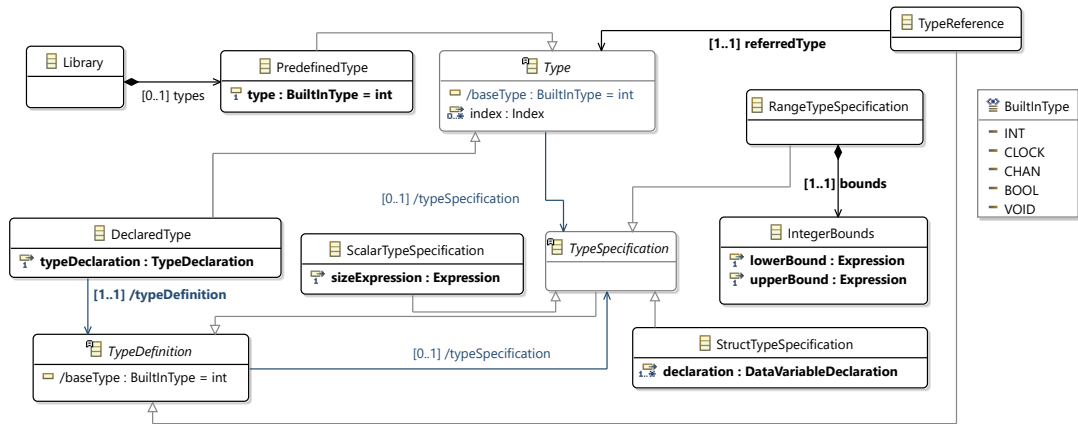


Figure E.26: Class Diagram of Package types of UPPAAL's Metamodel

The package expressions (see Figure E.27) enables to specify expressions. Its base class is the abstract class `Expression`. UPPAAL supports a broad range of expressions. Among others, it supports negation expressions, function call expressions, quantifier expressions (existential and universal), literal expressions, condition expressions, identifier expressions, and binary expressions. Moreover, several binary expressions exist as well, e.g., arithmetic expressions, assignment expressions, logical expressions, and comparison expressions.

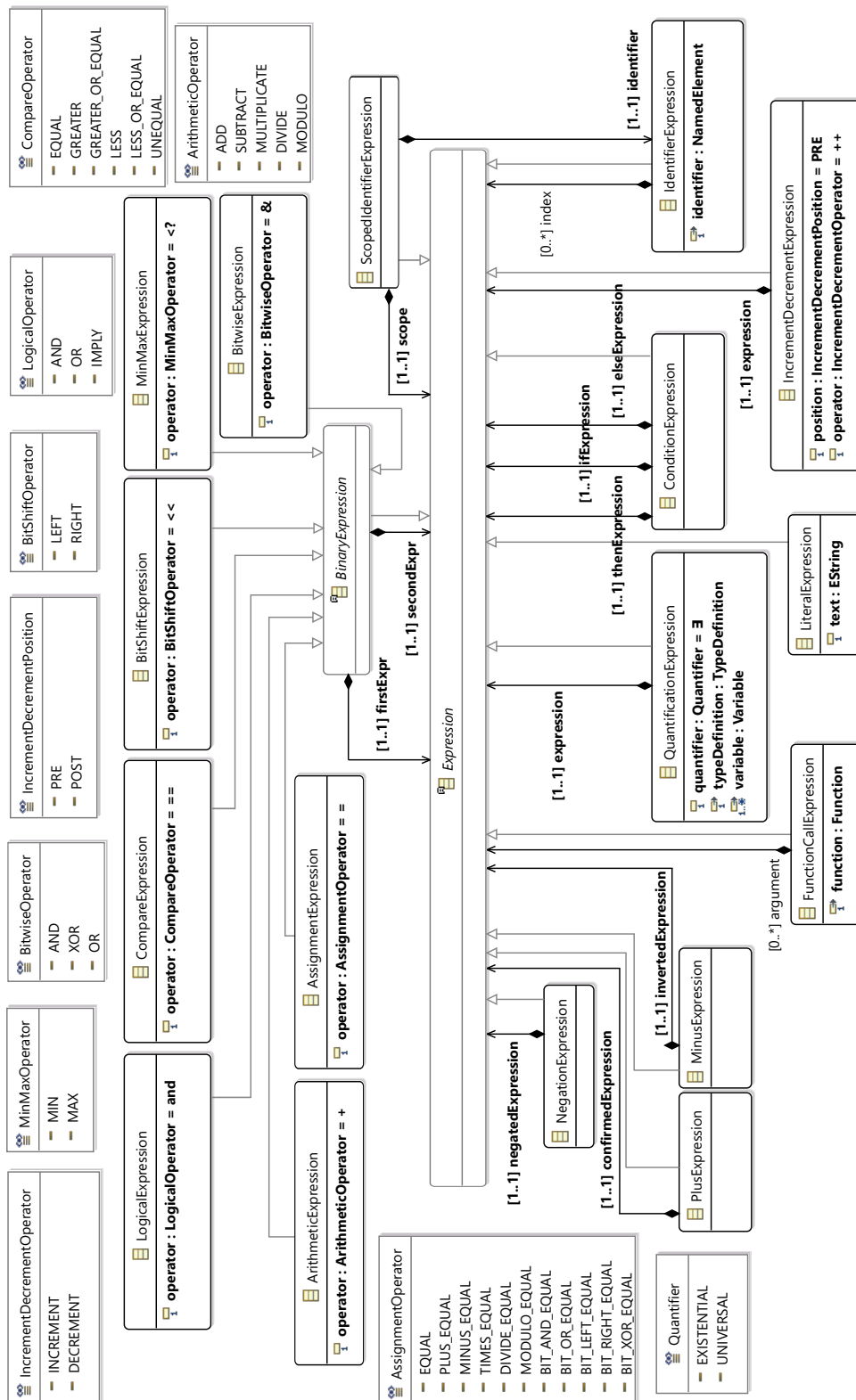


Figure E.27: Class Diagram of Package expressions of UPPAAL’s Metamodel

The package `visuals` (see Figure E.28) defines classes that are only relevant for visualizing a timed automaton. A graphical element of a timed automaton may have a default color, one of the 12 pre-defined colors, or a user-defined color code. Furthermore, graphical elements have defined points on the graphical surface: planar elements, e.g., a location, have a position and linear elements, e.g., an edge, may have multiple bend points.

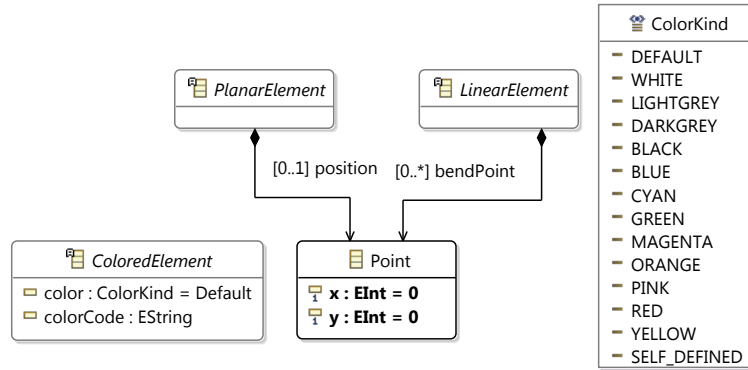


Figure E.28: Class Diagram of Package `visuals` of UPPAAL's Metamodel

E.3.4 UTCTL Metamodel

We show a class diagram of the UTCTL metamodel in Figure E.29. A property is represented by the abstract class `Property`. Properties are contained within a property repository. We distinguish two kinds of properties: unary and binary properties. Unary properties use one of these four temporal quantifiers: `A[]`, `A<>`, `E[]`, and `E<>`. The only binary property kind that UTCTL supports is the `leadsTo` property (via operator `->`).

The expression resp. the expressions that a property may define are specified using the class `Expression`, which is defined in package `expressions` in the metamodel of UPPAAL timed automata (see Section E.3.3).

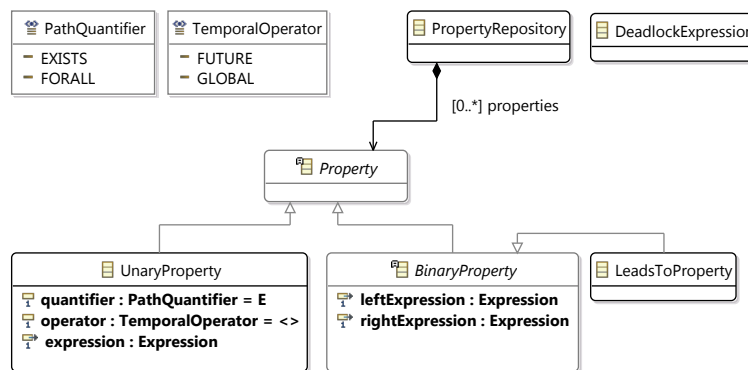


Figure E.29: Class Diagram of the UTCTL Metamodel

E.3.5 Uppaal Trace Grammar

In the following, we show the grammar of an UPPAAL trace that we specified in Xtext. The grammar references the UPPAAL timed automata metamodel (see Section E.3.3).

```

1  grammar org.muml.uppaal.trace.DiagnosticTrace with org.eclipse.xtext.
   common.Terminals
2
3  import "http://www.eclipse.org/emf/2002/Ecore" as.ecore
4  import "platform:/resource/org.muml.uppaal/model/uppaal.ecore" as.uppaal
5  import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//core" as
   core
6  import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//types" as
   types
7  import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//
   declarations" as.declarations
8  import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//
   declarations/global" as.global
9  import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//
   declarations/system" as.system
10 import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//templates"
   as.templates
11 import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//statements
   " as.statements
12 import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//
   expressions" as.expressions
13 import "platform:/resource/org.muml.uppaal/model/uppaal.ecore#//visuals"
   as.visuals
14
15 generate diagnostictrace "http://www.muml.org/uppaal/trace/1.0.0"
16
17 TraceRepository: ('Cannot_reuse_state_space_when_trace_length_optimisation
   _is_used.')(traces+=Trace)*;
18
19 Trace:
20   'Verifying'('property' | 'formula') property=INT 'at_line' line=INT
21   result=Result
22   (('Showing_counter_example.' | 'Showing_example_trace.') traceItems+=
   TraceItem+)?;
23
24 enum Result:
25   SUCCESS='_Property_is_satisfied.' | FAILURE='_Property_is_NOT_
   satisfied.' | SUCCESS='_Formula_is_satisfied.' | FAILURE='_
   Formula_is_NOT_satisfied.';
26
27 TraceItem: State | Transition;
28
29 State hidden(WS,DEPTH): 'State' (':')? // declare colon as optional since
   some UPPAAL traces come without a colon behind the 'State' keyword
30   '(' (locationActivities+=LocationActivity)+ ')'
31   (variableValues+=VariableValue ((',')? variableValues+=VariableValue)*)?
   // declare comma as optional since the list of UPPAAL variable
   values is sometimes comma-separated, sometimes not;
32
33 terminal DEPTH: '#depth'=INT;
34
35 // override rule to support signed negative integers as variable values
36 terminal INT returns.ecore::EInt: ('-')? ('0'..'9')+;
37
38 LocationActivity: process=ProcessIdentifier '.' location=[templates::
   Location];

```

```

39
40 ProcessIdentifier: template=[ templates :: AbstractTemplate ] ( '(' arguments +=
    INT ( ',' arguments += INT ) * ')' ) ? ;
41
42 VariableValue: firstNamedElementReference=NamedElementReference ( '-'
    secondNamedElementReference=NamedElementReference ) ? operator =
    CompareOperator value=INT;
43
44 NamedElementReference: singleNamedElementReference=
    SingleNamedElementReference ( '.' namedElementReference=
    NamedElementReference ) ? ;
45
46 SingleNamedElementReference: namedElement=[ core :: NamedElement ] ( '['
    indexes += INT ']' ) * ;
47
48 enum CompareOperator: EQ='=' | LEQ='<=' | GEQ='>=';
49
50 Transition: ActionTransition | DelayTransition;
51
52 ActionTransition: 'Transitions:' edgeActivities += EdgeActivity +;
53
54 EdgeActivity: source=LocationActivity '->' target=LocationActivity details
    =EDGE;
55
56 terminal EDGE: '{ ' -> ' }';

```

E.3.6 Uppaal Trace Metamodel

The metamodel for UPPAAL traces is generated out of the UPPAAL trace grammar (see Section E.3.5). The metamodel only references elements from the UPPAAL timed automata metamodel (see Section E.3.3).

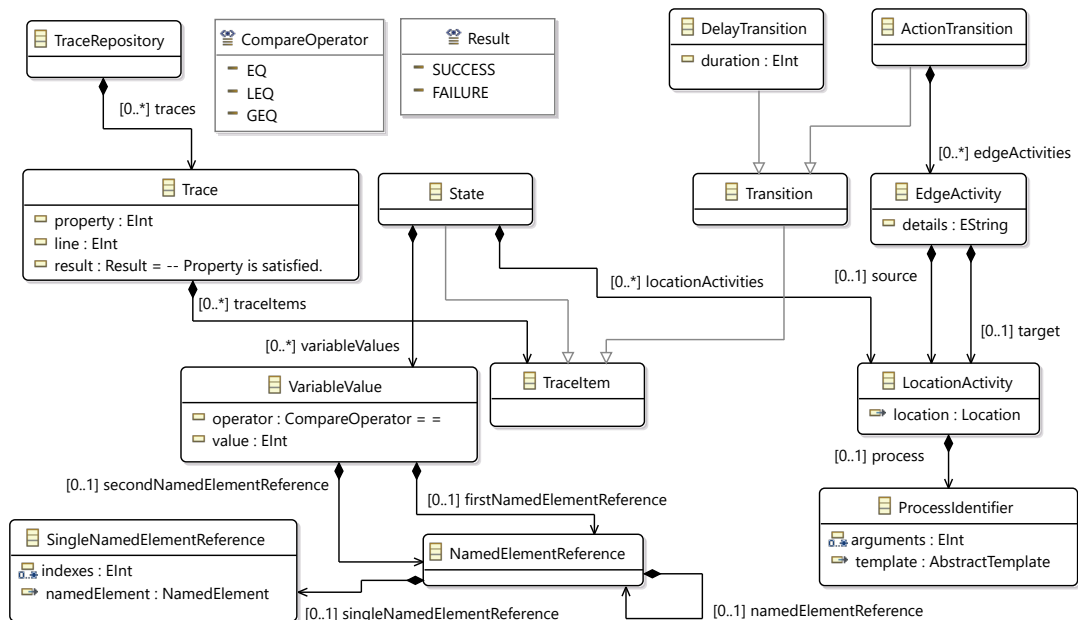


Figure E.30: Class Diagram of the UPPAAL Trace Metamodel

E.3.7 MechatronicUML Trace Metamodel

The MECHATRONICUML trace metamodel consists of several packages. We show the classes of the three packages (reachabilitygraph, rtsc, and cic) in Figure E.31. The classes `ActionTransition`, `HashToStateList`, `HashToStateListMapEntry`, `ReachabilityGraph`, `ReachabilityGraphState`, and `ReachabilityGraphTransition` belong to package `reachabilitygraph`; the classes `DelayTransition`, `ZoneGraph`, `ZoneGraphState`, and `ZoneGraphTransition` belong to package `rtsc`; the class `CICZoneGraphState` belongs to package `cic`.

The classes of package `reachabilitygraph` were defined by Heinzemann to store the result of the reachability analysis during a refinement check [HBDS15; Hei15]. Its base class is `ReachabilityGraph` that encapsulates the whole graph. Among others, it stores the states and transitions of this graph. States are expressed using the class `ReachabilityGraphState`; transitions are expressed using the class `ReachabilityGraphTransition`. The class `ActionTransition`, which inherits from `ReachabilityGraphTransition`, represents a snapshot transition that executes an action, e.g., firing a transition of an RTSC.

The classes of package `rtsc` inherit from the classes of package `reachabilitygraph`. They enable to specify a zone graph for a set of RTSC instances. An RTSC instance is specified via class `RealtimeStatechartInstance`, which is defined in the package `runtime`. A zone graph specifies a set of clocks and consists of zone graph states (class `ZoneGraphState`) and zone graph transitions (classes `ZoneGraphTransition` and `DelayTransition`). A zone graph state defines, among others, the set of RTSC instances.

The class `CICZoneGraphState` of package `cic` has class `ZoneGraphState` as its superclass. It additionally stores the component instance configuration (CIC) for the list of RTSC instances. As a consequence, we can infer the behavioral elements (the runtime discrete port instances and the runtime component instances) of the statecharts.

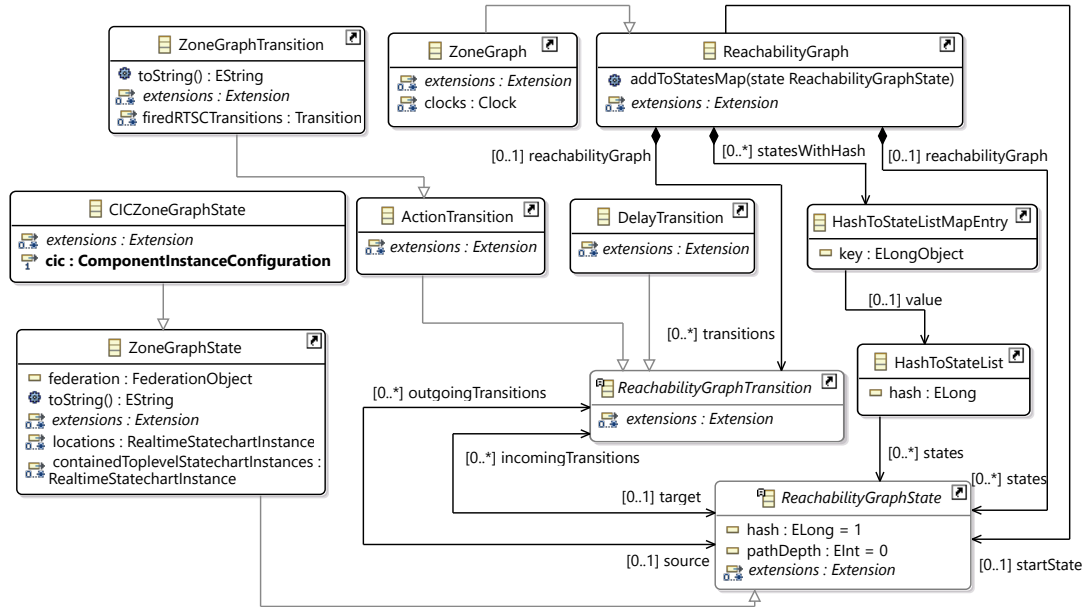


Figure E.31: Class Diagram of Packages `reachability`, `rtsc`, and `cic` of MECHATRONICUML's Trace Metamodel

The package runtime specifies the runtime state of a zone graph snapshot state. The classes of this package extend the package instance of MECHATRONICUML's PIM metamodel (see Section E.1.10). The base classes of this package are `RealtimeStatechartInstance` and `RuntimeBehavioralElement`. An RTSC instance knows its RTSC type, its parent instance, and its runtime behavior element. Moreover, the RTSC instance either stores its active vertex (a state or a connection point) or its active transitions. In addition, it binds a value to each variable of the RTSC (via class `VariableBinding`). Noteworthy, the clock values of an RTSC instance are not stored directly within this package but within the MECHATRONICUML verification extension metamodel (see Section E.3.2).

A runtime behavioral element knows its behavior (its RTSC instance) and stores its runtime message buffers (class `RuntimeMessageBuffer`). A runtime message buffer knows its message buffer type and may contain several runtime messages (via class `RuntimeMessage`). Each message knows its message type and may contain runtime parameters (via class `RuntimeParameter`), whereby a runtime parameter knows its type and has a value.

We distinguish two kinds of runtime behavioral elements: runtime component instances (class `RuntimeComponentInstance`) and runtime discrete interaction endpoint instances (class `RuntimeDiscreteInteractionEndpointInstance`). Among others, a runtime component instance knows its type, its port instances, its component part, and its CIC. In contrary, a runtime discrete interaction endpoint instance knows its type and its connector instances. As in the MECHATRONICUML PIM metamodel, we distinguish two kinds of runtime discrete interaction endpoint instances: role instances (class `RoleInstance`) and runtime discrete port instances (class `RuntimeDiscretePortInstance`). Among others, a role instance knows its role type and its runtime role connector instance (class `RuntimeRoleConnectorInstance`), which is one of the two kinds of a runtime connector instance (class `RuntimeConnectorInstance`). A runtime role connector instance knows its role connector type and its role instances. Moreover, it also stores the runtime messages that are currently in transit including their receiver. A runtime assembly connector instance (class `RuntimeAssemblyConnectorInstance`) also inherits from the class `RuntimeConnectorInstance`. Therefore, it knows its connector instance, assembly connector type (if it exists), and its port instances. In addition, this class also stores the connector QoS assumptions.

A role instance may be a single role instance (class `SingleRoleInstance`) or a multi role instance (class `MultiRoleInstance`). A single and a multi role instance both know their type. A multi role instance also knows its subrole instances including which subrole instance is the first and which is the last. We do not provide an extra class for a subrole instance. Instead, we reuse the class `SingleRoleInstance`. In contrary to a single role instance, a subrole instance also knows its multi port instance, the first and the last subrole instance of the multi port instance, and its next and previous subrole instance (if they exist).

A runtime discrete port instance knows—among others—its port type and its component instance. We distinguish two kinds of runtime discrete port instances: runtime discrete single port instances (class `RuntimeDiscreteSinglePortInstance`) and runtime discrete multi port instances (class `RuntimeDiscreteMultiPortInstance`). For their definition, we reused our metamodel concept for single role instances and multi role instances.

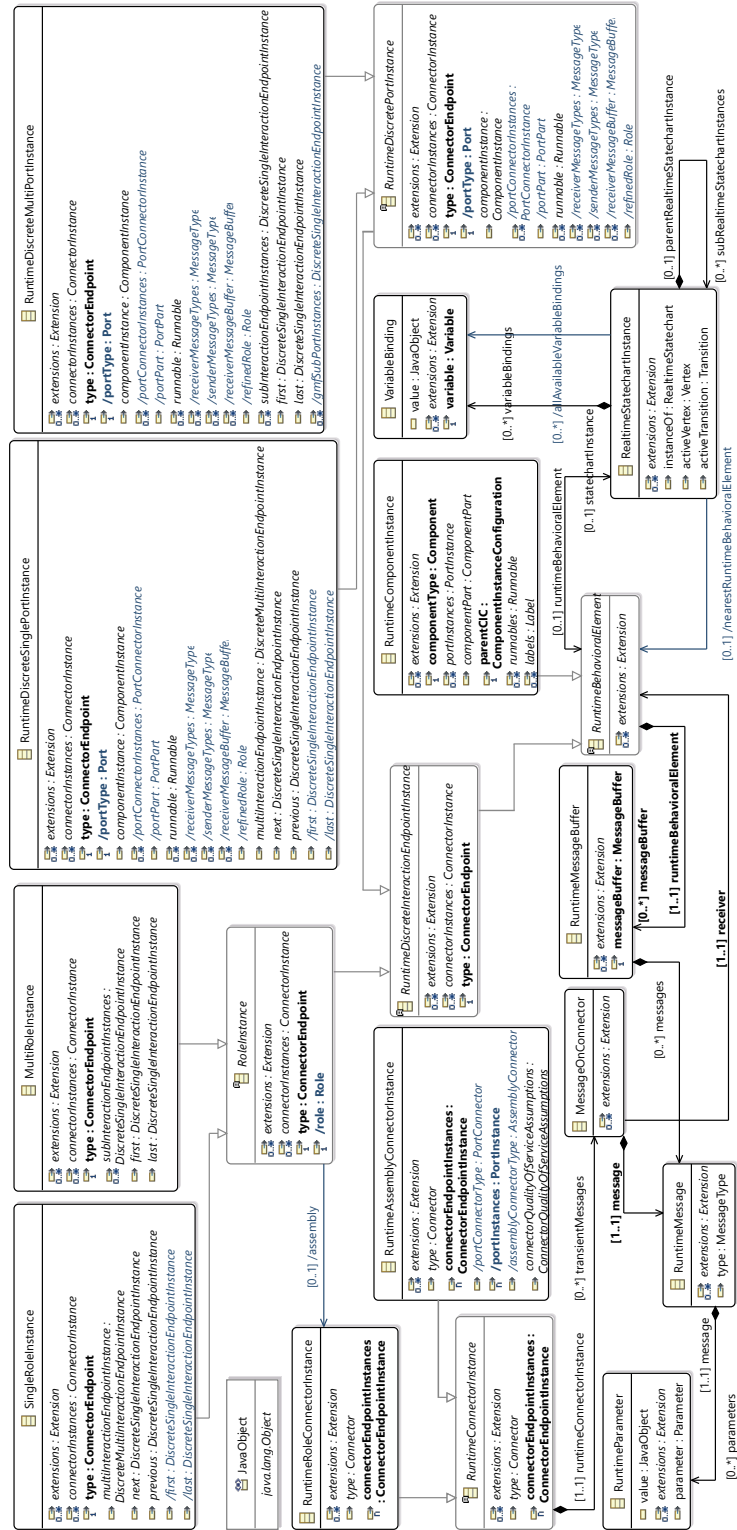


Figure E.32: Class Diagram of Package runtime of MECHATRONICUML's Trace Metamodel

E.4 Real-Time Coordination Patterns

In the following, we present in Section E.4.1 the metamodel for Real-Time Coordination Patterns (see Chapter 6) and in Section E.4.2 our ontology for Real-Time Coordination Patterns. The metamodel as well as the ontology were initially defined by the project group SafeBots III [ABB+13], which I supervised. Afterwards, I revised both.

E.4.1 Metamodel

The base class of this metamodel is the class `AbstractCoordinationPattern`, which inherits from class `AbstractCoordinationSpecification` of package `protocol` of MECHATRONICUML's PIM metamodel (see Section E.1.6). Like an RTCP, an abstract pattern may contain two roles, one role connector, verification constraint repositories, and may store if it was adapted from one or more patterns. In addition, an abstract pattern may also store pattern parameters and verified configurations (via class `VerifiedConfiguration`). A verified configuration contains a set of parameter bindings (one for each pattern parameter).

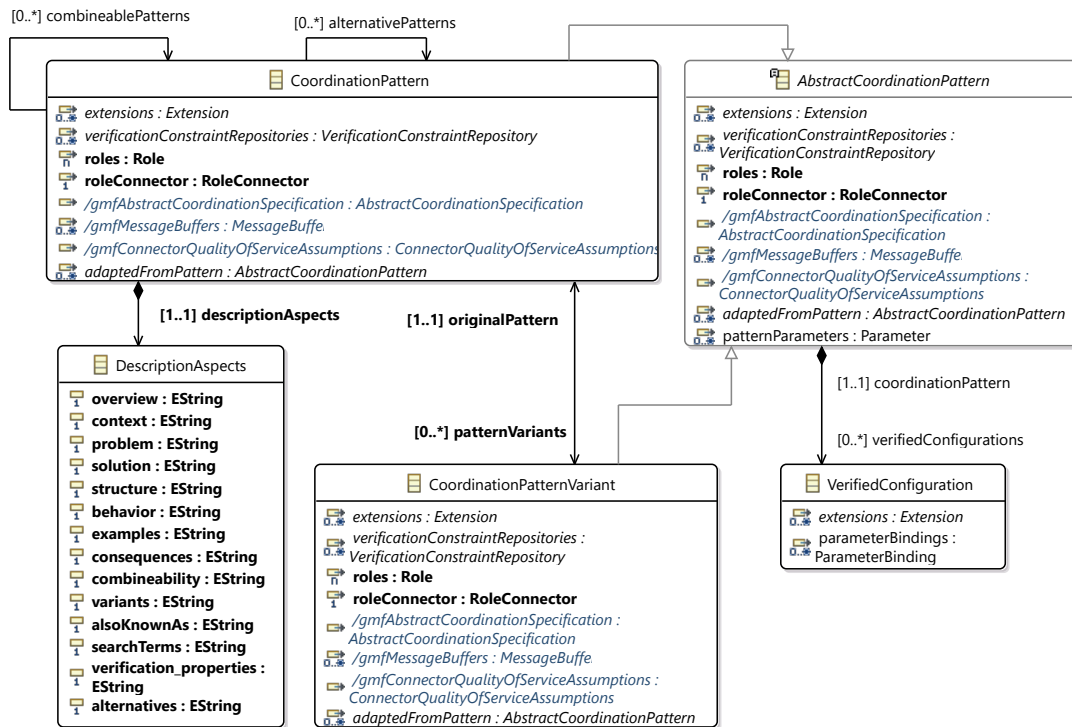


Figure E.33: Class Diagram of the Pattern Metamodel

An abstract pattern may either be a Real-Time Coordination Pattern (short: pattern; class `CoordinationPattern`) or a pattern variant (class `CoordinationPatternVariant`). A pattern may reference other patterns that are alternatives or that are combinable with this pattern. Moreover, a pattern defines description aspects (via class `DescriptionAspects`) and may define a set of pattern variants. The class `DescriptionAspects` stores 14 informally described pattern aspects (see Section 6.3.4; the aspect name is already defined in the class `CoordinationPattern`).

In contrast to a pattern, a pattern variant only stores the original pattern to which it belongs. Therefore, it does not store the information of a pattern (the alternative and combinable pattern and the description aspects) again.

E.4.2 Ontology

We visualize our OWL ontology for Real-Time Coordination Patterns via a class diagram for EMF metamodels in Figure E.34. A variant of our ontology has been published in [*GTS+14].

A brief explanation of our ontology is as follows: The class `CoordinationPattern` represents a Real-Time Coordination Pattern. `CoordinationPattern` defines the data property name and seven object properties:

- The object properties `is_less_concrete_than` and `is_more_concrete_than` state if other patterns exist that are less or more concrete than this pattern. Both properties refer to the same class and are transitive.
- The object properties `is_combinable_with` and `is_alternative_to` state with which other patterns this pattern is combinable and its alternatives. Both properties refer to the same class and are transitive and symmetric.
- The object property `fulfills` states at least one functionality of the pattern by referencing class `Function`.
- The object property `has_description` states at least one functionality of the pattern by referencing class `InformalDescription`.

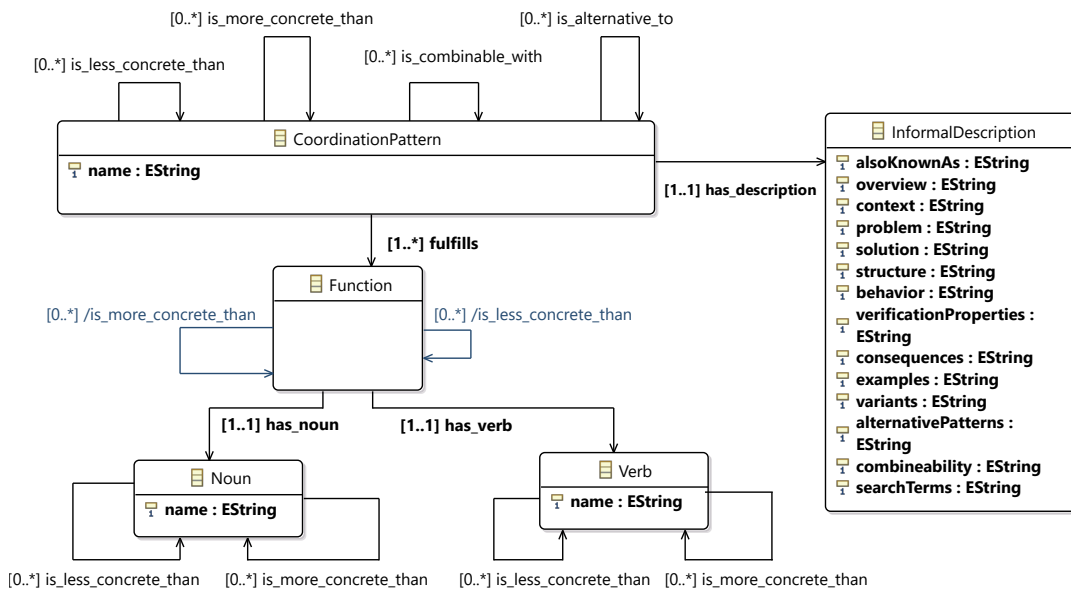


Figure E.34: OWL Ontology for Real-Time Coordination Patterns Visualized in EMF

A function consists of a noun (class `Noun`) and a verb (class `Verb`). Both, a noun and a verb, may know other nouns/verbs that are less or more concrete than themselves. Consequently, we can infer whether a function is more or less concrete than another function. Finally, the class `InformalDescription` defines 14 data properties to store the remaining description aspects of the pattern.

F

LIST OF ABBREVIATIONS

CIC	Component Instance Configuration
CPS	Cyber-physical System
CTL	Computation Tree Logic
DSL	Domain-specific Language
DSMC	Domain-specific Model Checking
DSVPL	Domain-specific Verification Property Language
EMF	Eclipse Modeling Framework
FIFO	First In – First Out
FO-TCTL	First-Order TCTL
GMF	Graphical Modeling Framework
LIFO	Last In – First Out
LTL	Linear-Time Temporal Logic
MDA	Model-driven Architecture
MSD	Modal Sequence Diagram
MTCTL	MECHATRONICUML TCTL
M2M	Model-to-Model
NTA	Network of Timed Automata
OCL	Object Constraint Language
OWL	Web Ontology Language
PIM	Platform-independent Model
QoS	Quality-of-Service

QVT	Query View Transformation
QVTo	Query View Transformation Operational
RDF	Resource Description Framework
RTCP	Real-Time Coordination Protocol
RTSC	Real-Time Statechart
SPARQL	SPARQL Protocol and RDF Query Language
SVG	Scalable Vector Graphics
TCTL	Timed Computation Tree Logic
UML	Unified Modeling Language
UTCTL	Uppaal TCTL
w.r.t.	with respect to

PAPER CONTRIBUTIONS

My contribution to each of my publication is as follows:

[*BDG+11; *BBD+12; *BBB+12; *BDG+14b; *DPP+16] We published the MECHATRONICUML method in five versions: 0.1, 0.2, 0.3, 0.4, and 1.0. My contributions to these versions are as follows: (1) In all versions, I was the only author for the section about RTCPs, the metamodel of RTCPs, their process, and the related work about contract-based design and behavioral connectors. (2) In [*BDG+14b], Christopher Gerking and myself completely rewrote and partially redefined the specification of RTSCs. (3) I extensively contributed in defining (but not writing) the remaining parts of the language (except reconfiguration), the metamodel, and the process of MECHATRONICUML. (4) I reviewed the introduction, the overview, the language definition, the process, and the example in several versions. (5) I supervised the writing and reviewing of the versions [*BDG+14b] and [*DPP+16]. (6) In all versions, I contributed in specifying the running example.

[*AGD+12] Harald Anacker and I were the main authors and jointly developed the concepts of this paper. I wrote Chapter 3 (Solution Elements of the Software Engineering Domain) and jointly wrote Chapter 4 (System Design Based on Solution Patterns) with Harald Anacker.

[*BDG+14a] This paper has been written in combination with a poster for the poster session of this conference. The paper summarizes the MECHATRONICUML method. I contributed in specifying the concepts that the paper presents. Moreover, I supervised the writing and reviewing of this paper and jointly wrote it with my colleagues. The poster has been created by me.

[*DBHT12] This technical report is an extended version of [*DHT12]. In contrast to the conference paper, this report presents the complete catalog of Real-Time Coordination Patterns. The patterns have been identified by my students (i.e., within the bachelor's theses that I supervised) and by me. My student Kathrin Bröker provided initial drafts for some of the pattern descriptions. Later on, I revised and unified all pattern descriptions and wrote descriptions for additional patterns.

[*DGB+14] I was one of the main authors and supervised the writing and reviewing of this paper. Moreover, I contributed in writing Chapter 2 (MechatronicUML Tool Suite) and I wrote Chapter 4 (Conclusion). Moreover, I revised and reviewed the paper. The

MECHATRONICUML Tool Suite is a joint product developed by the authors of this paper and the help of several students. The concept of the tool demo has been defined by Sebastian Thiele and me but only Sebastian Thiele created the tool demo.

[*DGB14] I was the single author of this paper (the other two only reviewed the paper). The concepts of this paper were developed by me and by Sebastian Goschin (within his master's thesis [Gos14] that I supervised).

[*DGH15] This technical report is an extended version of [*GSDH15]. Compared to the workshop paper, I additionally wrote the chapters 6 (Model-To-Model Translation from RTCPs to Uppaal), 7 (Automating Uppaal), and 8 (Model-To-Model Back-Translation from Uppaal to RTCPs). The concepts of this chapter were defined by the master's thesis of Christopher Gerking [Ger13], by the project group Cybertron [BCD+14], and by me. I supervised the master's thesis as well as the project group.

[*DHT12] I was the main author of this paper and developed most of the concepts on my own. Moreover, I supervised the writing and reviewing of this paper. In particular, I wrote the chapters 3 (Patterns for Real-Time Coordination Protocols), 5 (Case Study: Cooperating Robots), and 6 (Related Work). The case study was supervised by me as well. In addition, I reviewed and revised all other chapters.

[*DJS+13] Viktor Just, Thomas Schierbaum, and I were the main authors of this paper. We developed these concepts together and with the help of other colleagues. I wrote major parts of Chapter 2 (Integrierter Regelungs- und Softwareentwurf). Furthermore, I also wrote the chapter 3 (Verwandte Arbeiten) and 4 (Resümee und Ausblick). In addition, I reviewed and revised all other chapters.

[*GSA+11] The main authors of this paper were Harald Anacker, Frank Bauer, and me. We jointly developed the concepts of this paper (along with additional colleagues). I wrote section 5.4 (Konkretisierung der diskreten Softwareanteile) and reviewed the other parts of the paper.

[*GSDH15] Christopher Gerking and I were the main authors of this paper. The paper is based on the master's thesis of Gerking [Ger13], the work of the project group Cybertron [BCD+14], and concepts by me. The master's thesis as well as the project group were both supervised by me. For this paper, I wrote the abstract, major parts of Chapter 2 (Foundations), major parts of Chapter 4 (DSMC for MUML's RTCPs Using Uppaal), Chapter 5 (Case Study), and Chapter 7 (Conclusion and Future Work). Furthermore, I reviewed and revised the complete paper. The case study was executed by me with the help of one of my students.

[*GTS+14] I was one of the main authors of this book. I wrote major parts of Chapter 2 (Grundlagen) and Chapter 3 (Instrumentarium und dessen praktische Anwendung an einem Demonstrator). Most of the parts that I wrote about in Chapter 3 were conceptually defined by myself. Moreover, I reviewed several parts of the book.

[*HBDS15] Christian Heinzemann, Christian Brenner, and I were the main authors of this journal paper. Most of the concepts of this paper were defined by Heinzemann and Brenner. Though, the concepts about RTCPs, their formal verification, and

their QoS assumptions were contributed by me. I wrote Chapter 2 (Modeling with MechatronicUML), Chapter 3 (Verifying Real-Time Coordination Protocols), and Chapter 7 (Case Study). Furthermore, I reviewed the complete paper.

[*HSD+15] I was one of the main authors of this technical report. In particular, I jointly wrote Chapter 2 (Description), 3 (Engineering and Adaptation Challenges), and 6 (Related Problems). Furthermore, I reviewed the complete report.

[*ODB+12] I was one of the main authors of this paper and developed the concepts of this paper together with Felix Oestersötebier and Frank Bauer. In particular, I wrote Section 2.2 (Lösungselemente der Softwaretechnik) and jointly wrote Chapter 3 (Semantische Technologien zur Beschreibung von Lösungswissen) and 4 (Einbindung in den Entwurfsprozess). Furthermore, I reviewed and revised the complete paper.

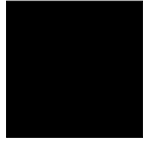
[*OJT+12] I was co-author of this paper and wrote its abstract as well as its Chapter 1 (Introduction). Furthermore, I reviewed and revised parts of the paper. In addition, I was partially involved in developing the concepts of this paper.

[*PDS+12] I jointly developed the concepts with Uwe Pohlmann and Boris Wolf and was one of the main authors. In particular, I wrote Chapter 2 (Running Example) and reviewed as well as revised all other chapters.

[*PDM+14] Uwe Pohlmann and I were the main authors of this paper and jointly developed its concepts. In particular, I wrote the Chapters 2 (Running Example), 3 (Real-Time Coordination Patterns), and 8 (Conclusion). Moreover, I revised all other chapters. The Modelica pattern library was developed by Sebastian Thiele, who was supervised by Uwe Pohlmann and me.

[*PTD+14] I was one of the main authors of this paper and developed its concepts together with my colleagues. In particular, I wrote—together with Uwe Pohlmann—the Section 3.b (Application Layer) and the Chapter 4 (Experience). Furthermore, I reviewed and revised the paper.

[*TJD+12] I was a co-author of this paper and only wrote Section 3.4 (Integration von Regelungs- und Softwarekonzept) together with Uwe Pohlmann.



BIBLIOGRAPHY

The bibliography is structured into four parts: my own publication, the theses that I have supervised, external literature, and norms and specification.

Own Publications

In this section, I do not only list publications that contribute to my PhD thesis but all publications that I wrote during my time of a PhD student. The publication key of all my publications have the prefix * to identify them easily within this thesis.

- [*AGD+12] H. Anacker, J. Gausemeier, R. Dumitrescu, S. Dziwok, and W. Schäfer, “Solution patterns of software engineering for the system design of advanced mechatronic systems,” in *Proceedings of the 9th France-Japan / 7th Europe-Asia Congress, 13th International Workshop on Research and Education in Mechatronics (MECATRONICS-REM 2012)*, Supméca, Paris, France, 2012, pp. 101–108. DOI: 10.1109/MECATRONICS.2012.6450994 (cit. on pp. 227, 395).
- [*BBB+12] S. Becker, C. Brenner, C. Brink, S. Dziwok, C. Heinzemann, R. Löffler, U. Pohlmann, W. Schäfer, J. Suck, and O. Sudmann, “The MechatronicUML design method – process, syntax, and semantics,” Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-12-326, Aug. 2012, Vers. 0.3 (cit. on pp. 27, 87, 88, 328, 357, 395).
- [*BBD+12] S. Becker, C. Brenner, S. Dziwok, T. Gewering, C. Heinzemann, U. Pohlmann, C. Priesterjahn, W. Schäfer, J. Suck, O. Sudmann, and M. Tichy, “The MechatronicUML method – process, syntax, and semantics,” Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-12-318, Feb. 2012, Vers. 0.2 (cit. on pp. 27, 87, 88, 328, 357, 395).
- [*BDG+11] S. Becker, S. Dziwok, T. Gewering, C. Heinzemann, U. Pohlmann, C. Priesterjahn, W. Schäfer, O. Sudmann, and M. Tichy, “MechatronicUML – syntax and semantics,” Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-11-325, Aug. 2011, Vers. 0.1 (cit. on pp. 27, 87, 88, 301, 328, 357, 395).

- [*BDG+14a] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann, “The MechatronicUML method: Model-driven software engineering of self-adaptive mechatronic systems,” in *Proceedings of the 36th International Conference on Software Engineering (Posters)*, ser. ICSE’14, Hyderabad, India: ACM, May 2014 (cit. on pp. 3, 24, 59, 395).
- [*BDG+14b] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, S. Thiele, W. Schäfer, M. Meyer, U. Pohlmann, C. Priesterjahn, and M. Tichy, “The MechatronicUML design method - process and language for platform-independent modeling,” Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-14-337, Mar. 2014, Version 0.4 (cit. on pp. 27, 68, 87, 88, 328, 357, 395).
- [*DBHT12] S. Dziwok, K. Bröker, C. Heinzemann, and M. Tichy, “A catalog of real-time coordination patterns for advanced mechatronic systems,” Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-12-319, Feb. 2012 (cit. on pp. 227, 232, 233, 235, 240, 253, 289, 292, 294, 295, 298, 301, 302, 304, 306–308, 311, 312, 395).
- [*DGB+14] S. Dziwok, C. Gerking, S. Becker, S. Thiele, C. Heinzemann, and U. Pohlmann, “A tool suite for the model-driven software engineering of cyber-physical systems,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hongkong, China: ACM, Nov. 2014, pp. 715–718, ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2661665 (cit. on pp. 11, 31, 84, 129, 199, 248, 261, 395).
- [*DGB14] S. Dziwok, S. Goschin, and S. Becker, “Specifying intra-component dependencies for synthesizing component behaviors,” in *Proceedings of the 1st International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp 2014)*, F. Ciccozzi, M. Tivoli, and J. Carlson, Eds., Valencia, Spain: CEUR-WS.org Vol-1281, Sep. 2014, pp. 16–25 (cit. on pp. 27, 396).
- [*DGH15] S. Dziwok, C. Gerking, and C. Heinzemann, “Domain-specific model checking of MechatronicUML models using UPPAAL,” Paderborn University, Tech. Rep. tr-ri-15-346, Jul. 2015 (cit. on pp. 143, 146–153, 155–179, 181–187, 190, 192, 194–197, 204, 205, 222–224, 396).
- [*DHT12] S. Dziwok, C. Heinzemann, and M. Tichy, “Real-time coordination patterns for advanced mechatronic systems,” in *Proceedings of the 14th International Conference on Coordination Languages and Models (COORDINATION 2012)*, IFIP International Federation for Information Processing (2012), M. Sirjani, Ed., ser. LNCS 7274, Stockholm, Sweden, Jun. 2012, pp. 166–180 (cit. on pp. 227, 231, 235, 256, 395, 396).
- [*DJS+13] S. Dziwok, V. Just, T. Schierbaum, W. Schäfer, A. Trächtler, and J. Gausemeier, “Integrierter Regelungs- und Softwareentwurf für komplexe mechatronische Systeme,” in *9. Paderborner Workshop Entwurf mechatronischer Systeme*, Heinz Nixdorf Institut, vol. 310, Verlagsschriftenreihe des Heinz Nixdorf Instituts, Paderborn, Apr. 2013, pp. 375–394 (cit. on p. 396).

-
- [*DPP+16] S. Dziwok, U. Pohlmann, G. Piskachev, D. Schubert, S. Thiele, and C. Gerking, “The MechatronicUML design method: Process and language for platform-independent modeling,” Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Zukunftsmeile 1, 33102 Paderborn, Germany, Tech. Rep. tr-ri-16-352, Dec. 2016, Version 1.0 (cit. on pp. 3, 24, 25, 27–31, 33, 37–44, 46–49, 52, 55, 67–71, 78, 80, 82, 87, 88, 97, 98, 136, 150, 151, 157–161, 165, 167, 169, 176, 178, 261, 294, 328–333, 357, 359, 364, 395).
- [*GSA+11] J. Gausemeier, W. Schäfer, H. Anacker, F. Bauer, and S. Dziwok, “Einsatz semantischer Technologien im Entwurf mechatronischer Systeme,” in *8. Paderborner Workshop Entwurf mechatronischer Systeme (EMS)*, Paderborn, Germany, ser. HNI-Verlagsschriftenreihe, vol. 294, Heinz Nixdorf Verlag, May 2011, pp. 7–35 (cit. on p. 396).
- [*GSDH15] C. Gerking, W. Schäfer, S. Dziwok, and C. Heinzemann, “Domain-specific model checking for cyber-physical systems,” in *Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2015)*, M. Famelis, D. Ratiu, M. Seidl, and G. Selim, Eds., Ottawa, Sep. 2015. [Online]. Available: <http://ceur-ws.org/Vol-1514/paper3.pdf> (cit. on pp. 25–27, 33, 143, 396).
- [*GTS+14] J. Gausemeier, A. Trächtler, W. Schäfer, H. Anacker, F. Bauer, H. Borchert, S. Dziwok, U. Frank, R. Herden, G. Hoppe, V. Just, M. Kieler-Dunsche, D. Kruse, F. Oestersötebier, J. Papenfort, U. Pohlmann, H. Reddehase, J. Rieke, T. Schierbaum, L. Seifert, H. Stichweh, H. Teichrieb, R. Wagner, and S. Wessels, *Semantische technologien im entwurf mechatronischer systeme: Effektiver austausch von lösungswissen in branchenwertschöpfungsketten*. München: Carl Hanser Verlag, Jun. 2014, ISBN: 978-3446436305 (cit. on pp. 227, 250, 254, 391, 396).
- [*HBDS15] C. Heinzemann, C. Brenner, S. Dziwok, and W. Schäfer, “Automata-based refinement checking for real-time systems,” *Computer Science - Research and Development*, 2015, Published Online June 2014, ISSN: 1865-2034. DOI: 10.1007/s00450-014-0257-9 (cit. on pp. 4, 24–26, 78–80, 196, 246, 321, 322, 377, 387, 396).
- [*HSD+15] C. Heinzemann, D. Schubert, S. Dziwok, U. Pohlmann, C. Priesterjahn, C. Brenner, and W. Schäfer, “Railcab convoys: An exemplar for using self-adaptation in cyber-physical systems,” Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-15-344, Jan. 2015. DOI: 10.13140/2.1.4726.8163 (cit. on p. 397).
- [*ODB+12] F. Oestersötebier, F. Bauer, S. Dziwok, A. Trachtler, W. Schäfer, and J. Gausemeier, “Unterstützung des mechatronischen entwurfs durch die effektive suche nach lösungselementen mithilfe von semantischen technologien,” in *Tag des Systems Engineering*, M. Maurer and S.-O. Schulze, Eds., Gesellschaft für Systems Engineering e.V., Sep. 2012 (cit. on p. 397).

- [*OJT+12] F. Oestersötebier, V. Just, A. Trächtler, F. Bauer, and S. Dziwok, “Model-based design of mechatronic systems by means of semantic web ontologies and reusable solution elements,” in *Proceedings of the ASME 2012 11th Biennial Conference on Engineering Systems Design and Analysis*, ASME, Nantes, France: ASME, Jul. 2012, pp. 647–656. DOI: doi:10.1115/ESDA2012-82324 (cit. on p. 397).
- [*PDM+14] U. Pohlmann, S. Dziwok, M. Meyer, M. Tichy, and S. Thiele, “A modelica coordination pattern library for cyber-physical systems,” in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools’14, Lisbon, Portugal: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Mar. 2014, pp. 76–85, ISBN: 978-1-63190-007-5. DOI: 10.4108/icst.simutools.2014.254640 (cit. on pp. 227, 238, 239, 397).
- [*PDS+12] U. Pohlmann, S. Dziwok, J. Suck, B. Wolf, C. C. Loh, and M. Tichy, “A modelica library for real-time coordination modeling,” in *Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany*, DLR - Robotics and Mechatronics Center, Modelica Association, Linköping University Electronic Press, Linköpings universitet, 2012, pp. 365–374, ISBN: 978-91-7519-826-2. DOI: 10.3384/ecp12076365 (cit. on p. 397).
- [*PTD+14] U. Pohlmann, H. Trsek, L. Dürkop, S. Dziwok, and F. Oestersötebier, “Application of an intelligent network architecture on a cooperative cyber-physical system: An experience report,” in *19th IEEE Conference on Emerging Technologies and Factory Automation (Work-in-Progress)*, IEEE, Jan. 2014 (cit. on p. 397).
- [*TJD+12] H. Teichrieb, V. Just, S. Dziwok, U. Pohlmann, T. Schierbaum, and A. Trächtler, “Modellbasierter Entwurf zweier kooperierender Delta-Roboter,” in *Berichtsband des Tag des Systems Engineering 2012*, Jan. 2012 (cit. on p. 397).

Supervised Theses

Noteworthy, in this section, I do not only list theses that contribute to my PhD thesis but all theses that I supervised during my time of a PhD student.

For each bachelor’s or master’s thesis, I executed the same process: First, I defined the topic of the thesis including motivation, problems, and goals. Then, the students had to write their thesis. During this time, I discussed with each student their progress at least one hour per week.

A project group typically consists of 8 to 12 students at master level that have to execute a project for two semesters. In the project group, each student has to write a seminar thesis about a topic. Then, the group has to define a solution for a given problem, implement the solution, and evaluate it. Typically, the students execute tasks given by not one but several supervisors. Therefore, not all students of the project groups worked on my topics. In the project group SafeBots III [ABB+13], I supervised Vinay Akkasetty Gopal, Jana Bröggelwirth, and Sijia

Li. In the project group Cybertron [BCD+14], I supervised Jan Bobolz, Mike Czech, David Schubert, and Rebekka Wohlrab. Therefore, I defined their topic of the seminar thesis and their topic of the project group including motivation, problems, and goals. During the project group, I discussed with my students their progress at least one hour per week.

- [ABB+13] V. Akkasetty Gopal, J. Bröggelwirth, C. Brune, F. Ghassemi, A. Kollias, S. Li, S. Merschjohann, S. Schwichtenberg, and D. Shipchanov, *Final documentation of the project group safebots iii*, Paderborn University, Feb. 2013 (cit. on pp. 227, 256, 264, 390, 402).
- [BCD+14] J. Bobolz, M. Czech, A. Dann, J. Geismann, M. Hüwe, A. Krieger, G. Piskachev, D. Schubert, and R. Wohlrab, *Final documentation of the project group cybertron*, Paderborn University, Sep. 2014 (cit. on pp. 103, 109, 114–116, 143, 202, 313, 371, 377, 396, 403).
- [Brö14] J. Bröggelwirth, “Design of real-time coordination protocols for multilateral communication,” master’s thesis, Institute, Paderborn University, Aug. 2014 (cit. on pp. 33, 34, 357).
- [Dre11] J. Drewello, “Musterbasierter Entwurf für das Verhalten von BeBot-Robotern mit MechatronicUML,” bachelor’s thesis, Paderborn University, 2011 (cit. on pp. 227, 301, 306, 309, 328).
- [Ger13] C. Gerking, “Transparent UPPAAL-based verification of MechatronicUML models,” master’s thesis, Paderborn University, May 2013 (cit. on pp. 143, 377, 396).
- [Gos14] S. Goschin, “Synthesis of extended hierarchical real-time behavior,” master’s thesis, Paderborn University, Feb. 2014 (cit. on pp. 27, 396).
- [Jah11] T. Jahn, “Funktionsbeschreibung von Softwarelösungselementen in Ontologien,” bachelor’s thesis, Paderborn University, Sep. 2011.
- [Kas15] H. Kassner, “Protocol-compliant synthesis of real-time component behavior,” Master’s thesis, Paderborn University, Jun. 2015 (cit. on p. 27).
- [Kri13] A. Kriger, “Modellierung von Konnektor- und Nachrichtenpuffereigenschaften in MechatronicUML,” bachelor’s thesis, Paderborn University, 2013 (cit. on p. 33).
- [Rie11] S. Riepe, “Zustandsbasierte Verhaltensmodellierung von Steuerungssystemen mit Entwurfsmustern,” bachelor’s thesis, Paderborn University, Jun. 2011 (cit. on pp. 59, 227, 295, 304).
- [San11] M. Sander, “Prozessevaluierung der Software-Konkretisierung nach ENTIME,” bachelor’s thesis, Paderborn University, Nov. 2011 (cit. on pp. 59, 227, 253, 289, 291, 306).
- [SDP+15] W. Schäfer, S. Dziwok, U. Pohlmann, J. Bobolz, M. Czech, A. P. Dann, J. Geismann, M. Hüwe, A. Krieger, G. Piskachev, D. Schubert, and R. Wohlrab, “Seminar theses of the project group cybertron,” Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Projektgruppenbericht tr-ri-15-345, Jul. 2015 (cit. on pp. 103, 104).

- [Wol11] B. Wolf, “Automatische Transformation von Modellen der MechatronicUML nach Modelica,” master’s thesis, Paderborn University, Nov. 2011.

Literature

- [Abr74] J.-R. Abrial, “Data semantics,” in *IFIP Working Conference Data Base Management*, 1974, pp. 1–60 (cit. on p. 38).
- [ABRS04] F. Arbab, C. Baier, J. Rutten, and M. Sirjani, “Modeling component connectors in reo by constraint automata: (extended abstract),” *Electronic Notes in Theoretical Computer Science*, vol. 97, no. 0, pp. 25–46, 2004, Proceedings of FOCLASA 2003, the Foundations of Coordination Languages and Software Architectures, a satellite event of CONCUR 2003, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.04.028. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S157106610405039X> (cit. on pp. 7, 97).
- [aca11] acatech - National Academy of Science, Ed., *Cyber-physical systems – driving force for innovation in mobility, health, energy and production*. Springer, 2011, ISBN: 978-3-642-29090-9 (cit. on pp. 1, 261).
- [ACD93] R. Alur, C. Courcoubetis, and D. L. Dill, “Model-checking in dense real-time,” *Information and Computation*, vol. 104, no. 1, pp. 2–34, May 1993, ISSN: 0890-5401. DOI: 10.1006/inco.1993.1024 (cit. on pp. 12, 17, 21, 104).
- [AD94] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, Apr. 1994, ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8 (cit. on p. 17).
- [Alu99] R. Alur, “Timed automata,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. A. Peled, Eds., vol. 1633, Springer Berlin Heidelberg, Jul. 1999, pp. 8–22, ISBN: 978-3-540-66202-0. DOI: 10.1007/3-540-48683-6_3 (cit. on p. 19).
- [AMB+12] M. Asplund, A. Manzoor, M. Bouroche, S. Clarke, and V. Cahill, “A formal approach to autonomous vehicle coordination,” in *FM 2012: Formal Methods*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Méry, Eds., vol. 7436, Springer Berlin Heidelberg, 2012, pp. 52–67, ISBN: 978-3-642-32758-2. DOI: 10.1007/978-3-642-32759-9_8. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32759-9_8 (cit. on p. 96).
- [Arb06] F. Arbab, “Coordination for component composition,” *Electronic Notes in Theoretical Computer Science*, vol. 160, no. 0, pp. 15–40, 2006, Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005) Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005), ISSN: 1571-0661. DOI: 10.1016/j.entcs.2006.05.013. [Online]. Available:

-
- <http://www.sciencedirect.com/science/article/pii/S1571066106003756> (cit. on p. 97).
- [BBK+15] L. Baresi, G. Blohm, D. S. Kolovos, N. Matragkas, A. Motta, R. F. Paige, A. Radjenovic, and M. Rossi, “Formal verification and validation of embedded systems: The uml-based mades approach,” *Software & Systems Modeling*, vol. 14, no. 1, pp. 343–363, Feb. 2015 (cit. on pp. 9, 146, 223).
- [BCC+08] T. Bures, J. Carlson, I. Crnkovic, S. Sentilles, and A. Vulgarakis, “Procom - the progress component model reference manual, version 1.0,” Mälardalen University, Tech. Rep. ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Jun. 2008. [Online]. Available: <http://www.mrtc.mdh.se/index.php?choice=publications&id=1508> (cit. on p. 100).
- [BCC98] S. Berezin, S. Campos, and E. M. Clarke, “Compositional reasoning in model checking,” English, in *Compositionality: The Significant Difference*, ser. Lecture Notes in Computer Science, W.-P. de Roever, H. Langmaack, and A. Pnueli, Eds., vol. 1536, Springer Berlin Heidelberg, 1998, pp. 81–102, ISBN: 978-3-540-65493-3. DOI: 10.1007/3-540-49213-5_4 (cit. on pp. 3, 25, 220).
- [BCK98] M. Bellare, R. Canetti, and H. 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*, ser. STOC ’98, Dallas, Texas, USA: ACM, May 1998, pp. 419–428, ISBN: 0-89791-962-9. DOI: 10.1145/276698.276854 (cit. on p. 266).
- [BDL06] G. Behrmann, A. David, and K. G. Larsen, *A tutorial on UPPAAL 4.0*, Department of Computer Science, Aalborg University, Denmark, Nov. 2006 (cit. on pp. 18, 19, 21–23, 30, 179, 263).
- [BEH+97] E. Basturk, R. Engel, R. Haas, V. Peris, and D. Saha, “Using network layer anycast for load distribution in the internet,” in *TECH. REP., IBM T.J. WATSON RESEARCH CENTER*, 1997 (cit. on p. 66).
- [BGHS04] S. Burmester, H. Giese, M. Hirsch, and D. Schilling, “Incremental design and formal verification with uml/rt in the fujaba real-time tool suite,” in *Proceedings of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, Oct. 2004, pp. 1–20 (cit. on pp. 6, 8, 24, 27, 28, 33, 35, 42, 150, 261).
- [BGK+02] J. Bengtsson, W. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Automated verification of an audio-control protocol using UPPAAL,” *The Journal of Logic and Algebraic Programming*, vol. 52, pp. 163–181, 2002, ISSN: 1567-8326. DOI: [http://dx.doi.org/10.1016/S1567-8326\(02\)00036-X](http://dx.doi.org/10.1016/S1567-8326(02)00036-X). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S156783260200036X> (cit. on p. 30).

- [BK08] C. Baier and J.-P. Katoen, *Principles of model checking*. Cambridge, MA, USA: MIT Press, 2008, ISBN: 978-0-262-02649-9 (cit. on pp. 2, 3, 17, 25, 26, 138, 257, 264).
- [BLL+96] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, “UPPAAL — a tool suite for automatic verification of real-time systems,” English, in *Hybrid Systems III*, ser. Lecture Notes in Computer Science, R. Alur, T. A. Henzinger, and E. Sontag, Eds., vol. 1066, Springer Berlin Heidelberg, 1996, pp. 232–243, ISBN: 978-3-540-61155-4. DOI: 10.1007/bfb0020949 (cit. on pp. 4, 17).
- [BMO01] B. Bauer, J. P. Müller, and J. Odell, “Agent uml: A formalism for specifying multiagent interaction,” in *Agent-Oriented Software Engineering*, P. Ciancarini and M. J. Wooldridge, Eds., Springer, 2001, pp. 91–103 (cit. on pp. 96, 256).
- [BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture, volume 1: A system of patterns*. Chichester, UK: John Wiley & Sons, 1996, ISBN: 978-0-471-95869-7 (cit. on pp. 10, 59, 60, 229, 231–233, 246, 256).
- [Brö11] K. Bröker, “Modellierung und verifikation von kommunikationsprotokollen für den betrieb des railcab systems,” Master’s thesis, Paderborn University, Nov. 2011 (cit. on p. 59).
- [Bru15] C. Brune, “Domain-specific property specification for timed model checking,” Master’s thesis, Paderborn University, Feb. 2015 (cit. on p. 139).
- [Bur06] S. Burmester, “Model-driven engineering of reconfigurable mechatronic systems,” PhD thesis, Paderborn University, Aug. 2006, ISBN: 9783832512989 (cit. on p. 243).
- [BY04] J. Bengtsson and W. Yi, “Timed automata: Semantics, algorithms and tools,” in *Lectures on Concurrency and Petri Nets*, ser. Lecture Notes in Computer Science, J. Desel, W. Reisig, and G. Rozenberg, Eds., vol. 3098, Springer Berlin Heidelberg, 2004, pp. 87–124, ISBN: 978-3-540-22261-3. DOI: 10.1007/978-3-540-27755-2_3 (cit. on pp. 17, 19, 27).
- [CBF+10] T. Correa, L. B. Becker, J.-M. Farines, J.-P. Bodeveix, M. Filali, and F. Vernadat, “Supporting the design of safety critical systems using AADL,” in *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS ’10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 331–336, ISBN: 978-0-7695-4015-3. DOI: 10.1109/iceccs.2010.56 (cit. on p. 100).
- [CDKB11] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed systems: Concepts and design*, 5th. USA: Addison-Wesley Publishing Company, 2011, ISBN: 9780132143011 (cit. on p. 66).
- [CE00] K. Czarnecki and U. W. Eisenecker, *Generative programming: Methods, tools, and applications*. Addison-Wesley, 2000 (cit. on p. 264).
- [CE82] E. M. Clarke and E. A. Emerson, “Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons,” *Science of Computer Programming*, vol. 2, no. 3, pp. 241–266, Dec. 1982 (cit. on p. 137).

-
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, Apr. 1986, ISSN: 0164-0925. DOI: 10.1145/5397.5399 (cit. on p. 21).
- [CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 2000, ISBN: 978-0262032704 (cit. on pp. 3, 17, 264).
- [CGR11] B. Combemale, L. Gonnord, and V. Rusu, “A generic tool for tracing executions back to a DSML’s operational semantics,” in *Modelling Foundations and Applications*, Proc. (Birmingham, Jun. 6–9, 2011), R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige, Eds., ser. LNCS, vol. 6698, Springer, 2011, pp. 35–51, ISBN: 978-3-642-21469-1 (cit. on p. 224).
- [Cim06] R. Cimperman, *Uat defined: A guide to practical user acceptance testing*, First. Addison-Wesley Professional, 2006, ISBN: 0321498143 (cit. on p. 263).
- [CK02] M. V. Cengarle and A. Knapp, “Towards ocl/rt,” in *FME 2002: Formal Methods - Getting IT Right*, ser. Lecture Notes in Computer Science, L.-H. Eriksson and P. Lindsay, Eds., vol. 2391, Springer Berlin Heidelberg, 2002, pp. 390–409, ISBN: 978-3-540-43928-8. DOI: 10.1007/3-540-45614-7_22 (cit. on pp. 8, 138, 265).
- [CKNZ12] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, “Model checking and the state explosion problem,” in *Tools for Practical Software Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds., vol. 7682, Springer Berlin Heidelberg, 2012, pp. 1–30, ISBN: 978-3-642-35745-9. DOI: 10.1007/978-3-642-35746-6_1 (cit. on p. 3).
- [DAC98] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Property specification patterns for finite-state verification,” in *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*, M. A. Ardis and J. M. Atlee, Eds., ACM, 1998, pp. 7–15 (cit. on pp. 124, 140, 258).
- [DAC99] —, “Patterns in property specifications for finite-state verification,” in *Proceedings of the 21st International Conference on Software Engineering (ICSE’99)*, 1999, pp. 411–420 (cit. on pp. 140, 258).
- [DCC+13] J. A. Davis, M. Clark, D. D. Cofer, A. Fifarek, J. Hinchman, J. Hoffman, B. Hulbert, S. P. Miller, and L. Wagner, “Study on the barriers to the industrial adoption of formal methods,” in *Formal Methods for Industrial Critical Systems*, (Madrid, Sep. 23–24, 2013), C. Pecheur and M. Dierkes, Eds., ser. LNCS, vol. 8187, Springer, 2013, pp. 63–77, ISBN: 978-3-642-41009-3 (cit. on p. 2).
- [dIIgl14] D. G. de la Iglesia, “A formal approach for designing distributed self-adaptive systems,” Dissertation No 171/2014, PhD thesis, Linnaeus University, Apr. 2014, ISBN: 978-91-87427-85-5 (cit. on p. 257).

- [DLL+11] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. van Vliet, and Z. Wang, “Statistical model checking for networks of priced timed automata,” in *Formal Modeling and Analysis of Timed Systems: 9th International Conference, FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*, U. Fahrenberg and S. Tripakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 80–96, ISBN: 978-3-642-24310-3. DOI: 10.1007/978-3-642-24310-3_7. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24310-3_7 (cit. on p. 53).
- [DLL+12] A. David, K. G. Larsen, A. Legay, M. Møller, U. Nyman, A. Ravn, A. Skou, and A. Wasowski, “Compositional verification of real-time systems using ecdar,” English, *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 6, pp. 703–720, 2012, ISSN: 1433-2779. DOI: 10.1007/s10009-012-0237-y (cit. on p. 3).
- [DMMV14] R. Deshayes, B. Meyers, T. Mens, and H. Vangheluwe, “Promobox in practice : A case study on the GISMO domain-specific modelling language,” in *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014.*, 2014, pp. 21–30. [Online]. Available: <http://ceur-ws.org/Vol-1237/paper3.pdf> (cit. on p. 138).
- [DMY02] A. David, M. O. Möller, and W. Yi, “Formal verification of UML statecharts with real-time extensions,” in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, R.-D. Kutsche and H. Weber, Eds., vol. 2306, Springer Berlin / Heidelberg, 2002, pp. 208–241, ISBN: 978-3-540-43353-8. DOI: 10.1007/3-540-45923-5_15. [Online]. Available: <http://www.springerlink.com/content/0xgx6u9trljp32wp> (cit. on pp. 150, 163, 222).
- [DOP15] I. Dragomir, I. Ober, and C. Percebois, “Contract-based modeling and verification of timed safety requirements within sysml,” *Software & Systems Modeling*, pp. 1–38, 2015, ISSN: 1619-1374. DOI: 10.1007/s10270-015-0481-1 (cit. on pp. 2, 3, 95, 223).
- [Dor08] R. Dorociak, “Hybride Verifikation von Mechatronic UML Modellen durch Integration des Modelcheckers PHAVer,” bachelor’s thesis, Paderborn University, Jan. 2008 (cit. on p. 9).
- [Dou02] B. P. Douglass, *Real-time design patterns: Robust scalable architecture for real-time systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0201699567 (cit. on pp. 10, 256).
- [Dou99] —, *Doing hard time: Developing real-time systems with uml, objects, frameworks, and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN: 0-201-49837-5 (cit. on pp. 10, 256).
- [DR02] J. Daemen and V. 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 (cit. on p. 266).

-
- [DR99] J. S. Dumas and J. Redish, *A practical guide to usability testing*. Intellect Books, 1999 (cit. on p. 265).
- [Dra14] I. Dragomir, “Contract-based modeling and verification of timed safety requirements for system design in sysml,” PhD thesis, University of Toulouse, Dec. 2014 (cit. on pp. 95, 223).
- [dSWP11] O. M. dos Santos, J. Woodcock, and R. Paige, “Using model transformation to generate graphical counter-examples for the formal analysis of xuml models,” in *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, Apr. 2011, pp. 117–126. DOI: 10.1109/iceccs.2011.19 (cit. on p. 8).
- [DVW92] W. Diffie, P. C. Van Oorschot, and M. J. Wiener, “Authentication and authenticated key exchanges,” *Designs, Codes and Cryptography*, vol. 2, no. 2, pp. 107–125, 1992, ISSN: 1573-7586. DOI: 10.1007/BF00124891. [Online]. Available: <http://dx.doi.org/10.1007/BF00124891> (cit. on p. 266).
- [EABP12] A. Etien, V. Aranega, X. Blanc, and R. F. Paige, “Chaining model transformations,” in *Proceedings of the First Workshop on the Analysis of Model Transformations*, Innsbruck: ACM, 2012, pp. 9–14, ISBN: 978-1-4503-1803-7. DOI: 10.1145/2432497.2432500 (cit. on p. 146).
- [EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of algebraic graph transformation*, ser. Monographs in Theoretical Computer Science. Springer, 2006, ISBN: 978-3540311874. DOI: 10.1007/3-540-31188-2 (cit. on p. 137).
- [EH10] T. Eckardt and S. Henkler, “Component behavior synthesis for critical systems,” in *Architecting Critical Systems*, ser. Lecture Notes in Computer Science, H. Giese, Ed., vol. 6150, Springer Berlin Heidelberg, Jun. 2010, pp. 52–71, ISBN: 978-3-642-13555-2. DOI: 10.1007/978-3-642-13555-2_4 (cit. on p. 27).
- [EHH+13] T. Eckardt, C. Heinzemann, S. Henkler, M. Hirsch, C. Priesterjahn, and W. Schäfer, “Modeling and verifying dynamic communication structures based on graph transformations,” *Computer Science - Research and Development*, vol. 28, no. 1, pp. 3–22, Feb. 2013, Published online July 2011, ISSN: 1865-2034. DOI: 10.1007/s00450-011-0184-y (cit. on p. 137).
- [ERNF12] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle, “Improving the sat modulo ode approach to hybrid systems analysis by combining different enclosure methods,” English, *Software & Systems Modeling*, pp. 1–28, Nov. 2012, ISSN: 1619-1366. DOI: 10.1007/s10270-012-0295-3 (cit. on pp. 25, 265).
- [FC03] N. Fornara and M. Colombetti, “Defining interaction protocols using a commitment-based agent communication language,” in *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS ’03, Melbourne, Australia: ACM, 2003, pp. 520–527, ISBN: 1-58113-683-8. DOI: 10.1145/860575.860659. [Online].

- Available: <http://doi.acm.org/10.1145/860575.860659> (cit. on p. 140).
- [FM02] S. Flake and W. Mueller, “An ocl extension for real-time constraints,” English, in *Object Modeling with the OCL*, ser. Lecture Notes in Computer Science, T. Clark and J. Warmer, Eds., vol. 2263, Springer Berlin Heidelberg, 2002, pp. 150–171, ISBN: 978-3-540-43169-5. DOI: 10.1007/3-540-45669-4_8 (cit. on pp. 8, 137, 265).
- [FMR00] S. Flake, W. Mueller, and J. Ruf, “Structured english for model checking specification,” in *TRANS. AMER. MATH. SOC*, VDE Verlag, 2000, pp. 2547–2552 (cit. on pp. 139, 265).
- [FN08] A. Furfaro and L. Nigro, “Embedded control systems design based on RT-DEVS and temporal analysis using UPPAAL,” in *Proc. of the Intern. Multiconf. on Computer Science and Information Technology*, (Wisła, Oct. 20–22, 2008), M. Ganzha, M. Paprzycki, and T. Pelech-Pilichowski, Eds., vol. 3, Los Alamitos: IEEE, 2008, pp. 601–608, ISBN: 978-83-60810-14-9 (cit. on p. 223).
- [For03] N. Fornara, “Interaction and communication among autonomous agents in multiagent systems,” PhD thesis, Università della Svizzera italiana, Facoltà di Scienze della Comunicazione, Dec. 2003.
- [GB03] H. Giese and S. Burmester, “Real-time statechart semantics,” Software Engineering Group, Paderborn University, Paderborn, Germany, Tech. Rep. tr-ri-03-239, Jun. 2003 (cit. on pp. 8, 27, 28, 150, 157, 161).
- [GCKK06] H. J. Goldsby, B. H. C. Cheng, S. Konrad, and S. Kamdoun, “A visualization framework for the modeling and formal analysis of high assurance systems,” in *Model Driven Engineering Languages and Systems, 9th Intern. Conf. Proc.* (Genova, Oct. 1–6, 2006), O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., ser. LNCS, vol. 4199, Springer, 2006, pp. 707–721, ISBN: 978-3-540-45772-5 (cit. on p. 224).
- [GdLMD09] E. Guerra, J. de Lara, A. Malizia, and P. Díaz, “Supporting user-oriented analysis for multi-view domain-specific visual languages,” *Information and Software Technology*, vol. 51, no. 4, pp. 769–784, 2009 (cit. on pp. 146, 223).
- [GFZ13] J. A. Guerrero-Ibáñez, C. Flores-Cortés, and S. Zeadally, “Vehicular ad-hoc networks (vanets): Architecture, protocols and applications,” English, in *Next-Generation Wireless Technologies*, ser. Computer Communications and Networks, N. Chilamkurti, S. Zeadally, and H. Chaouchi, Eds., Springer London, 2013, pp. 49–70, ISBN: 978-1-4471-5163-0. DOI: 10.1007/978-1-4471-5164-7_5 (cit. on p. 66).
- [GHJV95] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley, 1995, ISBN: 0201633612 (cit. on pp. 3, 10, 59, 60, 229, 232, 233, 252, 256–258).
- [Gie03] H. Giese, “A formal calculus for the compositional pattern-based design of correct real-time systems,” Software Engineering Group, University of Paderborn, Tech. Rep. tr-ri-03-240, Jul. 2003 (cit. on p. 27).

-
- [Gre11] J. Greenyer, “Scenario-based design of mechatronic systems,” PhD thesis, Paderborn University, Oct. 2011 (cit. on pp. 149, 264, 377).
- [GRSS14] J. Gausemeier, F.-J. Rammig, W. Schäfer, and W. Sextro, Eds., *Dependability of self-optimizing mechatronic systems*, ser. Lecture Notes in Mechanical Engineering. Springer, Mar. 2014, ISBN: 978-3642537417 (cit. on p. 1).
- [GTB+03] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake, “Towards the compositional verification of real-time UML designs,” in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE’03, Helsinki, Finland: ACM, Sep. 2003, pp. 38–47, ISBN: 1-58113-743-5. DOI: 10.1145/940071.940078 (cit. on pp. 4, 6, 10, 24, 25, 33, 35, 42, 239, 243, 261).
- [HA06] S. Henninger and P. Ashokkumar, “An ontology-based metamodel for software patterns,” in *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE 2006)*, Jul. 2006, pp. 327–330 (cit. on p. 243).
- [HBRV10] Á. Hegedüs, G. Bergmann, I. Ráth, and D. Varró, “Back-annotation of simulation traces with change-driven model transformations,” in *Software Engineering and Formal Methods*, Proc. (Pisa, Sep. 13–18, 2010), J. L. Fiadeiro, S. Gnesi, and A. Maggiolo-Schettini, Eds., Los Alamitos: IEEE, 2010, pp. 145–155, ISBN: 978-0-7695-4153-2 (cit. on pp. 224, 263).
- [HC07] S. Henninger and V. Corrêa, “Software pattern communities: Current practices and challenges,” in *Proceedings of the 14th Conference on Pattern Languages of Programs 2007 (PLoP ’07)*, Monticello, Illinois, USA, September 5-8, 2007, Sep. 2007 (cit. on p. 243).
- [Hei15] C. Heinzemann, “Verification and simulation of self-adaptive mechatronic systems,” PhD thesis, Paderborn University, Sep. 2015. [Online]. Available: <http://digital.ub.uni-paderborn.de/hs/content/titleinfo/1775471> (cit. on pp. 17, 19, 20, 27, 34, 53, 54, 79, 88, 243, 263, 266, 321, 324–328, 357, 377, 387).
- [Hen12] S. Henkler, “Ein komponentenbasierter, modellgetriebener Softwareentwicklungsansatz für vernetzte, mechatronische Systeme,” PhD thesis, Paderborn University, Jun. 2012 (cit. on p. 243).
- [Hen96] T. A. Henzinger, “The theory of hybrid automata,” in *Proc. of the 11th IEEE Symposium on Logic in Computer Science*, ser. LICS ’96, New Brunswick, NJ, USA: IEEE Computer Society, Jul. 1996, pp. 278–292, ISBN: 0-8186-7463-6. DOI: 10.1109/lics.1996.561342 (cit. on pp. 25, 265).
- [HFK+16] J. Holtmann, M. Fockel, T. Koch, D. Schmelter, C. Brenner, R. Bernijazov, and M. Sander, “The mechatronicuml requirements engineering method: Process and language,” Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Tech. Rep. tr-ri-16-351, Dec. 2016 (cit. on p. 264).

- [HH11] C. Heinzemann and S. Henkler, “Reusing dynamic communication protocols in self-adaptive embedded component architectures,” in *Proceedings of the 14th International Symposium on Component Based Software Engineering*, ser. CBSE ’11, Boulder, Colorado, USA: ACM, Jun. 2011, pp. 109–118, ISBN: 978-1-4503-0723-9. DOI: 10.1145/2000229.2000246 (cit. on p. 79).
- [HHD09] N. Hadj Kacem, A. Hadj Kacem, and K. Drira, “A Formal Model of a Multi-step Coordination Protocol for Self-adaptive Software Using Coloured Petri Nets,” English, *IJCIS*, vol. 7, no. 1, Feb. 2009. [Online]. Available: <http://hal.archives-ouvertes.fr/hal-00361145> (cit. on pp. 7, 95).
- [HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay, “Contracts: Specifying behavioral compositions in object-oriented systems,” in *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA/ECOOP ’90, Ottawa, Canada: ACM, 1990, pp. 169–180, ISBN: 0-89791-411-2. DOI: 10.1145/97945.97967 (cit. on p. 4).
- [Hir04] M. Hirsch, “Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit UPPAAL,” Diploma Thesis, Paderborn University, Jun. 2004 (cit. on pp. 8, 150).
- [Hir08] —, “Modell-basierte Verifikation von vernetzten mechatronischen Systemen,” PhD thesis, Paderborn University, Sep. 2008 (cit. on pp. 6, 33, 35, 36, 53, 58, 59, 67, 70, 243, 261, 265).
- [HMP92] T. A. Henzinger, Z. Manna, and A. Pnueli, “Timed transition systems,” English, in *Real-Time: Theory in Practice*, ser. Lecture Notes in Computer Science, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds., vol. 600, Springer Berlin Heidelberg, 1992, pp. 226–251, ISBN: 978-3-540-55564-3. DOI: 10.1007/bfb0031995 (cit. on p. 109).
- [HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic model checking for real-time systems,” *Information and Computation*, vol. 111, no. 2, pp. 193–244, Jun. 1994, ISSN: 0890-5401. DOI: 10.1006/inco.1994.1045 (cit. on p. 17).
- [Hof07] T. Hoffmann, “Spezifikation und Synthese von Konnektorverhalten für Mechatronik UML,” Studienarbeit, Master’s thesis, Paderborn University, Feb. 2007 (cit. on p. 52).
- [Hol90] G. J. Holzmann, “Design and validation of protocols,” *Tutorial Computer Networks and ISDN Systems*, vol. 25, pp. 981–1017, 1990 (cit. on p. 138).
- [Hol97] —, “The model checker spin,” *Software Engineering, IEEE Transactions on*, vol. 23, no. 5, pp. 279–295, May 1997, ISSN: 0098-5589. DOI: 10.1109/32.588521 (cit. on p. 138).

-
- [HPB+10] P. Hosek, T. Pop, T. Bures, P. Hnetynka, and M. Malohlava, “Comparison of component frameworks for real-time embedded systems,” in *Component Based Software Engineering*, ser. Lecture Notes in Computer Science, L. Grunske, R. H. Reussner, and F. Plasil, Eds., vol. 6092, Berlin/Heidelberg: Springer, 2010, pp. 21–36, ISBN: 978-3-642-13237-7. DOI: 10.1007/978-3-642-13238-4_2 (cit. on p. 98).
- [HR04] M. Huth and M. Ryan, *Logic in computer science: Modelling and reasoning about systems*, 2nd. Cambridge, UK: Cambridge University Press, 2004, ISBN: 978-0-521-54310-1 (cit. on p. 21).
- [HRS13] C. Heinzemann, J. Rieke, and W. Schäfer, “Simulating self-adaptive component-based systems using MATLAB/Simulink,” in *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, ser. SASO ’13, Philadelphia, PA, USA: IEEE Computer Society, Sep. 2013, pp. 71–80. DOI: 10.1109/SASO.2013.17 (cit. on p. 25).
- [HSST13] C. Heinzemann, O. Sudmann, W. Schäfer, and M. Tichy, “A discipline-spanning development process for self-adaptive mechatronic systems,” in *Proceedings of the 2013 International Conference on Software and System Process*, ser. ICSSP 2013, May 18–19, San Francisco, USA: ACM, May 2013, pp. 36–45, ISBN: 978-1-4503-2062-7. DOI: 10.1145/2486046.2486055 (cit. on p. 31).
- [HTS+08] C. Henke, M. Tichy, T. Schneider, J. Böcker, and W. Schäfer, “Organization and control of autonomous railway convoys,” in *Proceedings of the 9th International Symposium on Advanced Vehicle Control*, ser. AVEC’08, Kobe, Japan, Oct. 2008, pp. 318–323 (cit. on pp. 88, 232, 238, 321).
- [HWR09] S. Herbrechtsmeier, U. Witkowski, and U. Rückert, “Bebot: A modular mobile miniature robot platform supporting hardware reconfiguration and multi-standard communication,” in *Progress in Robotics*, ser. Communications in Computer and Information Science, vol. 44, Berlin, Heidelberg: Springer-Verlag, 2009, pp. 346–356 (cit. on pp. 87, 232, 328).
- [JF13] M. Jaksic and B. Freitag, “Temporal patterns for document verification,” in *WWV 2010. 6th International Workshop on Automated Specification and Verification of Web Systems*, L. Kovacs and T. Kutsia, Eds., ser. EasyChair Proceedings in Computing, vol. 18, EasyChair, 2013, pp. 3–18 (cit. on p. 139).
- [JLPJ12] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, “Automated verification of AADL-specifications using UPPAAL,” in *IEEE 14th Intern. Symposium on High-Assurance Systems Engineering*, Proc. (Omaha, Oct. 25–27, 2012), V. Winter, R. Gandhi, and A. Parakh, Eds., IEEE, 2012, pp. 130–138, ISBN: 978-1-4673-4742-6 (cit. on p. 223).
- [KC05] S. Konrad and B. H. C. Cheng, “Real-time specification patterns,” in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, USA, May 15–21, 2005, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds., ACM, May 2005, pp. 372–381. DOI: 10.1145/1062455.1062526 (cit. on pp. 106, 124, 139, 140, 258, 265).

- [Kem11] S. Kemper, "Modelling and analysis of real-time coordination patterns," PhD thesis, Faculteit der Wiskunde en Natuurwetenschappen, 2011 (cit. on p. 258).
- [Kiz05] J. M. Kizza, *Computer network security*. Springer US, May 2005, ISBN: 978-0-387-20473-4 (cit. on pp. 51, 52, 266).
- [KMR02] A. Knapp, S. Merz, and C. Rauh, "Model checking timed UML state machines and collaborations," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Proc. (Oldenburg, Sep. 9, 2002–Sep. 12, 2012), W. Damm and E.-R. Olderog, Eds., ser. LNCS, vol. 2469, Springer, 2002, pp. 395–416, ISBN: 978-3-540-44165-6 (cit. on p. 222).
- [Kop11] H. Kopetz, *Real-time systems design principles for distributed embedded applications*, 2nd, J. A. Stankovic, Ed., ser. Real-Time Systems Series. Springer, 2011, ISBN: 978-1-4419-8237-7. DOI: 10.1007/978-1-4419-8237-7 (cit. on p. 1).
- [Koy90] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990 (cit. on p. 139).
- [KPP95] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Softw.*, vol. 12, no. 4, pp. 52–62, Jul. 1995, ISSN: 0740-7459. DOI: 10.1109/52.391832. [Online]. Available: <http://dx.doi.org/10.1109/52.391832> (cit. on pp. 86, 131, 204, 250).
- [Lam77] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, Mar. 1977, ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904 (cit. on p. 22).
- [LBJ15] A. Louati, K. Barkaoui, and C. Jerad, "Temporal properties verification of real-time systems using uml/marte/ocl-rt," English, in *Formalisms for Reuse and Systems Integration*, ser. Advances in Intelligent Systems and Computing, T. Bouabana-Tebibel and S. H. Rubin, Eds., vol. 346, Springer International Publishing, 2015, pp. 133–147, ISBN: 978-3-319-16576-9. DOI: 10.1007/978-3-319-16577-6_6 (cit. on p. 138).
- [LWF03] A. Lopes, M. Wermelinger, and J. L. Fiadeiro, "Higher-order architectural connectors," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 1, pp. 64–104, Jan. 2003, ISSN: 1049-331X. DOI: 10.1145/839268.839272 (cit. on pp. 97, 102).
- [MB08] L. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Ramakrishnan and J. Rehof, Eds., vol. 4963, Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24 (cit. on p. 96).
- [Mea55] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, Sep. 1955, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1955.tb03788.x (cit. on p. 17).

-
- [MPS11] G. Morb , F. Pigorsch, and C. Scholl, “Fully symbolic model checking for timed automata,” in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 616–632, ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_50. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_50 (cit. on p. 264).
- [MRK+97] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon, “A graphical environment for the design of concurrent real-time systems,” *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 1, pp. 31–79, Jan. 1997, ISSN: 1049-331X. DOI: 10.1145/237432.237438 (cit. on p. 139).
- [MSH02] H. Mazouzi, A. E. F. Seghrouchni, and S. Haddad, “Open protocol design for complex interactions in multi-agent systems,” in *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, ser. AAMAS ’02, Bologna, Italy: ACM, 2002, pp. 517–526, ISBN: 1-58113-480-0. DOI: 10.1145/544862.544866. [Online]. Available: <http://doi.acm.org/10.1145/544862.544866>.
- [MSKA14] Y.-J. Moon, A. Silva, C. Krause, and F. Arbab, “A compositional model to reason about end-to-end qos in stochastic reo connectors,” *Science of Computer Programming*, vol. 80, Part A, pp. 3–24, 2014, Special section on foundations of coordination languages and software architectures (selected papers from FOCLASA’10), Special section - Brazilian Symposium on Programming Languages (SBLP 2010) and Special section on formal methods for industrial critical systems (Selected papers from FMICS’11), ISSN: 0167-6423. DOI: 10.1016/j.scico.2011.11.007. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167642311002073> (cit. on p. 97).
- [MVR14] Z. Molotnikov, M. V lter, and D. Ratiu, “Automated domain-specific C verification with mbeddr,” in *ASE’14, Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, (V ster s, Sep. 15–19, 2014), I. Crnkovic, M. Chechik, and P. Gr nbacher, Eds., ACM, 2014, pp. 539–550, ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642938 (cit. on p. 224).
- [MWVD13] B. Meyers, M. Wimmer, H. Vangheluwe, and J. Denil, “Towards domain-specific property languages: The promobox approach,” in *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, ser. DSM ’13, Indianapolis, Indiana, USA: ACM, 2013, pp. 39–44, ISBN: 978-1-4503-2600-1. DOI: 10.1145/2541928.2541936 (cit. on p. 138).
- [NRS+11] D. E. Ndales Agut, M. A. Reniers, R. R. Schiffelers, K. Y. J rgensen, and D. van Beek, “A semantic-preserving transformation from the compositional interchange format to UPPAAL,” in *Proc. of the 18th IFAC World Congress*, (Milano, Aug. 28–Sep. 2, 2011), S. Bittanti, A. Cenedese, and S. Zampieri, Eds., 2011, pp. 12 496–12 502 (cit. on p. 223).

- [OPB01] J. J. Odell, H. V. D. Parunak, and B. Bauer, “Representing agent interaction protocols in UML,” in *IN OMG DOCUMENT AD/99-12-01. INTELLICORP INC*, Springer-Verlag, 2001, pp. 121–140 (cit. on p. 96).
- [PG06] R. G. Pettit IV and H. Gomaa, “Modeling behavioral design patterns of concurrent objects,” in *ICSE '06: Proceedings of the 28th international conference on Software engineering*, Shanghai, China: ACM, 2006, pp. 202–211.
- [PHMG14] U. Pohlmann, J. Holtmann, M. Meyer, and C. Gerking, “Generating Modelica models from software specifications for the simulation of cyber-physical systems,” in *SEAA '14*, IEEE Computer Society, Aug. 2014, pp. 191–198. DOI: 10.1109/SEAA.2014.18 (cit. on p. 25).
- [Pnu85] A. Pnueli, “In transition from global to modular temporal reasoning about programs,” English, in *Logics and Models of Concurrent Systems*, ser. NATO ASI Series, K. R. Apt, Ed., vol. 13, Springer Berlin Heidelberg, 1985, pp. 123–144, ISBN: 978-3-642-82455-5. DOI: 10.1007/978-3-642-82453-1_5 (cit. on p. 3).
- [Pri13] C. Priesterjahn, “Analyzing self-healing operations in mechatronic systems,” Dissertation, Paderborn University, Paderborn, Germany, Aug. 2013 (cit. on p. 243).
- [PT98] H. Peng and S. Tahar, “A survey on compositional verification,” Tech. Rep., 1998 (cit. on p. 3).
- [RC10] A. J. Ramirez and B. H. C. Cheng, “Design patterns for developing dynamically adaptive systems,” in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '10, Cape Town, South Africa: ACM, 2010, pp. 49–58, ISBN: 978-1-60558-971-8. DOI: 10.1145/1808984.1808990 (cit. on pp. 3, 10, 229, 233, 234, 257).
- [RHRR12] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*, 1st. Wiley Publishing, 2012, ISBN: 9781118104354 (cit. on pp. 86, 92, 93, 131, 134, 135, 204, 217, 219, 250, 254).
- [RK97] J. Ruf and T. Kropf, “Symbolic model checking for a discrete clocked temporal logic with intervals,” in *Proceedings of the IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 146–163, ISBN: 0-412-81330-0 (cit. on pp. 137, 139).
- [RM15] D. Romuald and T. Mens, “Gismo: A domain-specific modelling language for executable prototyping of gestural interaction,” in *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '15, Duisburg, Germany: ACM, 2015, pp. 34–43, ISBN: 978-1-4503-3646-8. DOI: 10.1145/2774225.2774838 (cit. on p. 138).

-
- [RMM+96] Y. Ramakrishna, P. Melliar-Smith, L. Moser, L. Dillon, and G. Kutty, "Interval logics and their decision procedures: Part i: An interval logic," *Theoretical Computer Science*, vol. 166, no. 1–2, pp. 1–47, 1996, ISSN: 0304-3975. DOI: [http://dx.doi.org/10.1016/0304-3975\(95\)00254-5](http://dx.doi.org/10.1016/0304-3975(95)00254-5). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/0304397595002545> (cit. on p. 139).
- [RvKT+00] E. E. Roubtsova, J. van Katwijk, W. J. Toetenel, C. Pronk, and R. C. M. de Rooij, "Specification of real-time systems in UML," *Electronic Notes in Theoretical Computer Science*, vol. 39, no. 3, pp. 293–305, 2000, MTCS 2000 (Satellite Workshop of CONCUR 2000), ISSN: 1571-0661. DOI: [10.1016/S1571-0661\(05\)80753-4](http://dx.doi.org/10.1016/S1571-0661(05)80753-4). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1571066105807534> (cit. on pp. 136, 137).
- [SAB10] S. M. A. Shah, K. Anastakis, and B. Bordbar, "From UML to Alloy and back again," in *Models in Software Engineering, Workshops and Symposia at MODELS 2009*, Reports and Revised Selected Papers (Denver, Oct. 4–9, 2009), S. Ghosh, Ed., ser. LNCS, vol. 6002, Springer, 2010, pp. 158–171, ISBN: 978-3-642-12260-6 (cit. on pp. 9, 146, 223).
- [SACO02] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Propel: An approach supporting property elucidation," in *Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002.*, May 2002, pp. 11–21 (cit. on p. 140).
- [SBF+97] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl, "Formal verification made easy," *IBM J. Res. Dev.*, vol. 41, no. 4-5, Jul. 1997, ISSN: 0018-8646. DOI: [10.1147/rd.414.0567](http://dx.doi.org/10.1147/rd.414.0567). [Online]. Available: <http://dx.doi.org/10.1147/rd.414.0567>.
- [Sch06] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006, ISSN: 0018-9162. DOI: [10.1109/mc.2006.58](http://dx.doi.org/10.1109/mc.2006.58) (cit. on p. 1).
- [Sch95] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science*, (Herrsching, Jun. 16–18, 1994), E. W. Mayr, G. Schmidt, and G. Tinhofer, Eds., ser. LNCS, vol. 903, Springer, 1995, pp. 151–163, ISBN: 978-3-540-59071-2 (cit. on p. 223).
- [SHS11] J. Suck, C. Heinzemann, and W. Schäfer, "Formalizing model checking on timed graph transformation systems," Software Engineering Group, Heinz Nixdorf Institute, Paderborn University, Tech. Rep. tr-ri-11-316, Sep. 2011 (cit. on p. 137).
- [SK03] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, 2003 (cit. on p. 146).
- [Sta08] F. Stallmann, "A model-driven approach to multi-agent system design," PhD thesis, Paderborn University, Apr. 2008 (cit. on pp. 8, 139).

- [STA98] R. L. Spelberg, H. Toetenel, and M. Ammerlaan, "Partition refinement in real-time model checking," English, in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, A. P. Ravn and H. Rischel, Eds., vol. 1486, Springer Berlin Heidelberg, 1998, pp. 143–157, ISBN: 978-3-540-65003-4. DOI: 10.1007/bfb0055344 (cit. on p. 136).
- [Ste05] A. Stecker, "Model Checking von Real-Time Statecharts mit RAVEN," Studienarbeit, Paderborn University, 2005 (cit. on p. 9).
- [Suc11] J. Suck, "Model Checking zeitbehafteter Graphtransformationssysteme," Master's thesis, Paderborn University, May 2011 (cit. on p. 137).
- [SW07] W. Schäfer and H. Wehrheim, "The challenges of building advanced mechatronic systems," in *Future of Software Engineering*, ser. FOSE '07, Minneapolis, MN, USA: IEEE Computer Society, May 2007, pp. 72–84, ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.28 (cit. on p. 1).
- [Tai07] T. Taibi, Ed., *Design patterns formalization techniques*. Hershey, Pennsylvania, USA: IGI Publishing, Mar. 2007, ISBN: 9781599042190 (cit. on pp. 229, 257).
- [Tan10] A. S. Tanenbaum, *Computer networks*. Prentice Hall, Sep. 2010, vol. Fifth Edition, ISBN: 978-0-13-212695-3 (cit. on pp. 53, 59).
- [Tic09] M. Tichy, "Gefahrenanalyse selbstoptimierender Systeme," Dissertation, Paderborn University, Warburger Str. 100, May 2009. [Online]. Available: <http://ubdok.uni-paderborn.de/servlets/DocumentServlet?id=10091> (cit. on p. 243).
- [TSR03] A. Tiwari, N. Shankar, and J. M. Rushby, "Invisible formal methods for embedded control systems," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 29–39, 2003 (cit. on p. 223).
- [VBD+13] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *Dsl engineering - designing, implementing and using domain-specific languages*. dslbook.org, 2013, pp. 1–558, ISBN: 978-1-4812-1858-0. [Online]. Available: <http://www.dslbook.org> (cit. on p. 2).
- [VDW12] W. Visser, M. B. Dwyer, and M. W. Whalen, "The hidden models of model checking," *Software & Systems Modeling*, vol. 11, no. 4, pp. 541–555, 2012 (cit. on pp. 1–3).
- [VSC+09] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seculeanu, and P. Pettersson, "Formal semantics of the procom real-time component model," in *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, ser. SEEA '09, Patras, Greece: IEEE Computer Society, Aug. 2009, pp. 478–485, ISBN: 978-0-7695-3784-9. DOI: 10.1109/seaa.2009.53 (cit. on pp. 30, 100).
- [WDR11] M. T. B. Waez, J. Dingel, and K. Rudie, "Timed automata for the development of real-time systems," Queen's University, Tech. Rep. 2011-579, 2011 (cit. on p. 17).

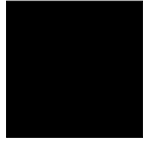
-
- [Wei08] F. Weigl, “Document verification with temporal description logics,” PhD thesis, University of Passau, 2008. [Online]. Available: <http://nbn-resolving.de/urn:nbn:de:bvb:739-opus-12528> (cit. on p. 139).
- [Win14] A. Winkler, “Modellgetriebene Softwareentwicklung zur induktiven Energieübertragung,” bachelor’s thesis, Paderborn University, Mar. 2014 (cit. on pp. 227, 253).
- [XPSA08] K. Xu, P. Pettersson, K. Sierszecki, and C. Angelov, “Verification of COMDES-II systems using UPPAAL with model transformation,” in *The 14th IEEE Intern. Conf. on Embedded and Real-Time Computing Systems and Applications*, Proc. (Kaohsiung, Aug. 25–27, 2008), Los Alamitos, 2008, pp. 153–160, ISBN: 978-0-7695-3349-0 (cit. on p. 223).
- [ZCP13] F. Zalila, X. Crégut, and M. Pantel, “Formal verification integration approach for DSML,” in *Model-Driven Engineering Languages and Systems*, ser. LNCS, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds., vol. 8107, Berlin and Heidelberg: Springer, 2013, pp. 336–351, ISBN: 978-3-642-41532-6 (cit. on pp. 9, 223).

Norms and Specifications

- [AADL] P. Feiler, D. Gluch, and J. Hudak, “The architecture analysis & design language (aadl): An introduction,” Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2006-TN-011, 2006. [Online]. Available: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879> (cit. on p. 99).
- [AES] *Advanced encryption standard (aes)*, National Institute of Standards, Nov. 2001. [Online]. Available: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf> (cit. on p. 266).
- [AUTOSAR] AUTOSAR development cooperation, “Release 4.2 Overview and Revision History,” Specification, version 4.2, 2014. [Online]. Available: <http://www.autosar.org/specifications/release-42/> (cit. on pp. 42, 49, 98, 222, 263).
- [BeBot] *BeBot miniature robot*. [Online]. Available: <https://www.hni.uni-paderborn.de/en/system-and-circuit-technology/projects/abgeschlossene-projekte/mini-robot-bebot> (cit. on pp. 87, 232, 328).
- [Car2Car] *CAR 2 CAR Communication Consortium*. [Online]. Available: <https://www.car-2-car.org> (cit. on p. 1).
- [EcoreTools] *EcoreTools*. [Online]. Available: <http://www.eclipse.org/ecoretools/> (cit. on p. 357).
- [EMF] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *Emf: Eclipse modeling framework*, 2nd, ser. The Eclipse Series. Addison-Wesley, Dec. 2008, ISBN: 978-0321331885 (cit. on pp. 32, 149, 357).

- [GMF] *Graphical Modeling Framework (GMF)*. [Online]. Available: <http://www.eclipse.org/gmf/> (cit. on p. 32).
- [Gra] *Graphviz – Graph Visualization Software*. [Online]. Available: www.graphviz.org (cit. on p. 187).
- [ISO14977] International Organization for Standardization, Ed., *Isotiec 14977:1996 information technology - syntactic metalanguage - extended bnf*, 1996 (cit. on pp. 45, 107).
- [ISO24765] ISO, “Systems and software engineering – vocabulary,” *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec. 2010. DOI: 10.1109/ieeestd.2010.5733835 (cit. on p. 1).
- [MARTE] OMG MARTE, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*, Specification, version 1.1, Object Management Group, 2011. [Online]. Available: <http://www.omg.org/spec/MARTE/1.1/PDF> (cit. on pp. 3, 7, 42, 98, 99, 263).
- [MDA] OMG MDA, *MDA Guide*, Specification, version 2.0, Object Management Group, 2014. [Online]. Available: <http://www.omg.org/cgi-bin/doc?ormsc/14-06-01> (cit. on p. 48).
- [OCL] OMG OCL, *Object Constraint Language (OCL)*, Specification, version 2.4, Object Management Group, 2014. [Online]. Available: <http://www.omg.org/spec/OCL/2.4/PDF/> (cit. on pp. 8, 32, 106, 137, 149, 257, 357).
- [OWL] P. F. Patel-Schneider, P. Hayes, and I. Horrocks, “Owl web ontology language semantics and abstract syntax,” *W3C recommendation*, vol. 10, 2004 (cit. on pp. 243, 246).
- [QVT] Object Management Group (OMG), *Meta object facility (mof) 2.0 query/view/transformation, v1.3*, formal/2016-06-03, Jun. 2016. [Online]. Available: <http://www.omg.org/spec/QVT/1.3/> (cit. on pp. 32, 146, 150, 187, 194).
- [QVTo] *QVT Operation (QVTo)*. [Online]. Available: <https://projects.eclipse.org/projects/modeling.mmt.qvt-oml> (cit. on pp. 32, 146, 150, 187).
- [RDF] D. Beckett, *RDF/XML Syntax Specification (Revised)*, Feb. 2004. [Online]. Available: <http://www.w3.org/TR/rdf-syntax-grammar/> (cit. on p. 246).
- [SI] B. N. Taylor and A. Thompson, *The international system of units (si). national institute of standards and technology special publication 330 (2008 edition)*. National Institute of Standards and Technology / U.S. Department of Commerce, 2008 (cit. on p. 29).
- [SPARQL] E. Prud’hommeaux, S. Harris, and A. Seaborne, Eds., *SPARQL Query Language for RDF*, W3C Recommendation, <https://www.w3.org/TR/sparql11-overview/>, Mar. 2013. [Online]. Available: <https://www.w3.org/TR/sparql11-overview/> (cit. on p. 246).

-
- [SysML] O. M. Group, *Omg systems modeling language (omg sysml)*, Document formal/10-06-02, Jun. 2010. [Online]. Available: <http://www.sysml.org/specs/> (cit. on pp. 95, 223).
- [UDTD] UPPAAL *DTD*. [Online]. Available: http://www.it.uu.se/research/group/darts/uppaal/flat-1_2.dtd (cit. on p. 189).
- [UML] OMG UML, *Unified Modeling Language (UML)*, version 2.5, Object Managment Group (OMG), Mar. 2015. [Online]. Available: <http://www.omg.org/spec/UML/2.5/> (cit. on pp. 24, 27, 38, 96, 252).
- [Xtend] *Xtend*. [Online]. Available: <https://eclipse.org/xtend/> (cit. on pp. 32, 188).
- [Xtext] *Xtext*. [Online]. Available: <http://www.eclipse.org/Xtext/> (cit. on pp. 32, 107, 129, 193, 357).



LIST OF FIGURES

1.1	Example Scenario: Coordinated Overtaking	5
1.2	Enhanced Design Process for the Software Development of CPS	12
1.3	Thesis Structure	15
2.1	Network of Timed Automata Specifying a Coordinated Overtaking	18
2.2	Zone Graph (Based on [Hei15])	20
2.3	Temporal Quantifier Supported by UTCTL and Exemplary Illustrations. The Proposition p is Fulfilled in Yellow Snapshots, the Proposition q is Fulfilled in Blue Snapshots (Based on [BDL06])	22
2.4	Overview of the Compositional Verification Approach (Based on [*GSDH15])	26
2.5	The RTCP SimpleOvertaking and its Role Behavior Defined Using RTSCs	28
3.1	The RTCP MultiOvertaking (Based on [*DPP+16])	37
3.2	Concrete Syntax of Roles and their Messages Type Specification [*DPP+16]	38
3.3	The Five Classes of an RTCP [*DPP+16]	41
3.4	Instance of RTCP MultiOvertaking (Role Multiplicity = 2) [*DPP+16]	47
3.5	The Assumed Layer Model of an RTCP [*DPP+16]	49
3.6	RTCP Overtaking (Structure and Behavior)	56
3.7	Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Multicast	61
3.8	Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Multireceive	63
3.9	RTCP MultiOvertaking (Differences to RTCP Overtaking are Highlighted in Red)	64
3.10	Example for Modeling and Instantiating a Multi Role RTSC With an Explicit Subrole RTSC	68
3.11	Simplified Version of RTSC Overtaker for RTCP Overtaking that Uses an Explicit RTSC for the Subrole	69
3.12	Process for the Normalization of RTSCs with One-To-Many Communication Schemata	71
3.13	Split Transition Normalization	71
3.14	State Connection Point Normalization	72
3.15	Normalization Template for Multicast without Subrole Condition	73
3.16	Normalization Template for MultiReceive without Subrole Condition	75

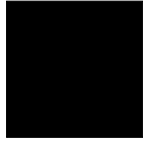
3.17	RTSC Overtaker without One-To-Many Communication Schemata (Changes are Marked Red)	77
3.18	Atomic Software Components for the Overtaking Scenario	81
3.19	CIC of Two Cars that Apply RTCP Overtaking	82
3.20	Component Behavior for RedSW and YellowSW (Refinements are Marked Red)	83
3.21	User Interface for Specifying the Structure of an RTCP	85
3.22	User Interface for Specifying the Behavior of a Role via RTSCs	86
3.23	Plugins used for implementing RTCPs	86
3.24	Variant of Role Sender of RTCP PositionTransmission that uses One-To-Many Communication Schemata	89
3.25	Variant of Role Distributor of RTCP Distribution that uses One-To-Many Communication Schemata	89
3.26	Variant 1 of Role ProfileProvider of RTCP ProfileDistribution that uses One-To-Many Communication Schemata	91
3.27	Variant 2 of Role ProfileProvider of RTCP ProfileDistribution that uses One-To-Many Communication Schemata	91
3.28	Variant of Role Coordinator of RTCP ConvoyCoordination that uses One-To-Many Communication Schemata	92
4.1	MTCTL: the Counterpart to UTCTL at the Level of MECHATRONICUML	106
4.2	Tree Representation of the MTCTL Property <code>not (AG (stateActive(Overtaker.Overtaking) implies stateActive(Overtakee.noAcceleration)))</code> ;	125
4.3	Traversing the Tree Representation for Producing the English Sentence	128
4.4	MTCTL Editor and MTCTL to English as Mouse-Over Event	130
4.5	Live Syntax Check and Error Messages	130
4.6	Adding Default Verification Properties	131
4.7	Plugins used for implementing MTCTL	131
5.1	Utilizing M2M Traceability for DSMC (Based on [*DGH15])	147
5.2	Resolving of Traceability Paths (Based on [*DGH15])	148
5.3	Concept for Forward Translation Chain (Based on [*DGH15])	149
5.4	Generating a CIC for a One-To-Many RTCPs (Based on [*DGH15])	152
5.5	RTSC Deadline Normalization (Based on [*DGH15])	157
5.6	RTSC Composite Transition Normalization (Based on [*DGH15])	159
5.7	RTSC Do-Effect Normalization for Simple States (Based on [*DGH15])	162
5.8	State Do-Effect Normalization for Composite States (Based on [*DGH15])	163
5.9	RTSC Hierarchy Normalization: Phase 1a (Based on [*DGH15])	164
5.10	RTSC Hierarchy Normalization: Phase 1b	167
5.11	RTSC Hierarchy Normalization: Phase 2a	168
5.12	RTSC Hierarchy Normalization: Phase 2b (Based on [*DGH15])	170
5.13	RTSC State Entry- / Exit-Effect Normalization (Based on [*DGH15])	172
5.14	Urgent Transition Normalization (Based on [*DGH15])	173
5.15	CIC Migration (Based on [*DGH15])	175
5.16	Defining the Next Array for Selector Expressions [*DGH15]	177
5.17	Using the next Keyword Within a Selector Expression (Based on [*DGH15])	177
5.18	Adding of Intermediate States (Based on [*DGH15])	178

5.19	Migrating Urgent Transitions	180
5.20	Asynchronous Message Exchange Concept (Based on [*DGH15])	182
5.21	Asynchronous Communication in UPPAAL (Based on [*DGH15])	183
5.22	Timed Automata Template Layouted by UPPAAL Only	188
5.23	Timed Automata Template Layouted via GraphViz	188
5.24	Back-Translation of an UPPAAL Counterexample Trace to MECHATRONIC- UML (Based on [*DGH15])	197
5.25	Selecting the Verification Mode	199
5.26	User Interface for Specifying DSMC Options	200
5.27	User Interface for Selecting MTCTL Properties	200
5.28	Result Presentation	201
5.29	SVG Representation of a MECHATRONICUML Snapshot (Based on [BCD+14])	202
5.30	Plugins used for implementing the DSMC for RTCPs via UPPAAL	203
5.31	Total Duration of the RTCP instances when Verifying Multiple Properties for a Result	212
5.32	Duration of the Three DSMC Phases when Verifying Multiple Properties for a Result	213
5.33	Total Duration of the RTCP Instances when Verifying a Single Property for a Trace	214
5.34	Duration of the Three DSMC Phases when Verifying a Single Property for a Trace	215
5.35	Number of XML Elements of the RTCP EMF Serialization and the UPPAAL EMF Serialization	217
6.1	General Concept for Real-Time Coordination Patterns	230
6.2	Real-Time Coordination Pattern Acquire Assurance	236
6.3	RailCab Prototypes (left) and RailCab Test Track (right) [*PDM+14]	238
6.4	Convoy Scenario of the RailCab System (Based on [*PDM+14])	239
6.5	One-To-Many Variant of the Pattern Acquire Assurance (Changes to the One-to-One Pattern are Marked Red)	241
6.6	Process for Abstracting an RTCP to a Pattern	242
6.7	RTCP ConvoyCoordination (Changes While Abstracting—Except Renamings—are Marked in Red)	244
6.8	Process for Selecting and Adapting a Pattern to an RTCP	245
6.9	RTCP Overtaking Adapted from the Pattern Acquire Assurance (Changes While Adapting—Except Renamings—are Marked in Red)	248
6.10	User Interface for Modeling the Structure of a Real-Time Coordination Pattern	249
6.11	Plugins used for implementing Real-Time Coordination Patterns	250
6.12	Cooperating Delta-Robots	251
6.13	Smart Universal Power Antenna (SUPA, Images taken from: http://www.supa- technology.de)	252
A.1	Normalization Template for Multicast with Conditions	268
A.2	Normalization Template for Multireceive with Conditions	270
A.3	Exemplary Sequence Diagram that Illustrates the Behavior of the One-To- Many Communication Schema Unicast	272

A.4	Normalization Template for Unicast	274
A.5	Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Singlereceive	276
A.6	Normalization Template for Singlereceive	277
A.7	Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Iterate	279
A.8	Normalization Template for Iterate with startFromFirst	281
A.9	Exemplary Sequence Diagram that Illustrates the Behavior of the One-To-Many Communication Schema Loadbalancing	283
A.10	Normalization Template for LoadBalancing with MaxWorkingTime	285
A.11	Normalization Template for LoadBalancing with MessageResponse	287
B.1	Real-Time Coordination Pattern Request Information	290
B.2	Real-Time Coordination Pattern Fail-operational Delegation	293
B.3	Real-Time Coordination Pattern Fail-safe Delegation	296
B.4	Real-Time Coordination Pattern Event-based Transmission	299
B.5	Real-Time Coordination Pattern Periodic Transmission	302
B.6	Real-Time Coordination Pattern Alternating Lock	305
B.7	Real-Time Coordination Pattern Block execution	307
B.8	Real-Time Coordination Pattern Limit Observation	310
C.1	Structure of RTCP Allow	314
C.2	Role Requestor of RTCP Allow	314
C.3	Role Controller of RTCP Allow	314
C.4	Structure of RTCP ChangeSection	315
C.5	Role Vehicle of RTCP ChangeSection	315
C.6	Role Section of RTCP ChangeSection	315
C.7	Structure of RTCP Delegate	316
C.8	Role Initiator of RTCP Delegate	316
C.9	Role Examiner of RTCP Delegate	316
C.10	Structure of RTCP Inform	317
C.11	Role Overtakee of RTCP Inform	317
C.12	Role Approacher of RTCP Inform	317
C.13	Structure of RTCP LimitVelocity	318
C.14	Role Limiter of RTCP LimitVelocity	318
C.15	Role LimitDriving of RTCP LimitVelocity	318
C.16	Structure of RTCP Overtake	319
C.17	Role Overtakee of RTCP Overtake	319
C.18	Role Overtaker of RTCP Overtake	319
C.19	Structure of RTCP VModeControl	320
C.20	Role VelocitySetter of RTCP VModeControl	320
C.21	Role VelocityGetter of RTCP VModeControl	320
C.22	Structure of RTCP EnterSection (Based on [*HBDS15])	321
C.23	Role RailCab of RTCP EnterSection (Based on [*HBDS15])	322
C.24	Role Section of RTCP EnterSection (Based on [*HBDS15])	322
C.25	Structure of RTCP NextSectionFree	323

C.26	Role Tracksection of RTCP NextSectionFree	323
C.27	Role Switch of RTCP NextSectionFree	323
C.28	Structure of RTCP ProfileDistribution (Based on [Hei15])	324
C.29	Changed Role ProfileProvider of RTCP ProfileDistribution (Based on [Hei15])	325
C.30	Role ProfileReceiver of RTCP ProfileDistribution (Based on [Hei15])	325
C.31	Structure of RTCP ConvoyCoordination (Based on [Hei15])	326
C.32	Changed Role Coordinator of RTCP ConvoyCoordination (Based on [Hei15])	327
C.33	Role Member of RTCP ProfileDistribution (Based on [Hei15])	328
C.34	Structure of RTCP Navigation [*DPP+16]	329
C.35	Role Navigator of RTCP Navigation [*DPP+16]	329
C.36	Role Provider of RTCP Navigation [*DPP+16]	329
C.37	Structure of RTCP Delegation [*DPP+16]	329
C.38	Role Master of RTCP Delegation [*DPP+16]	330
C.39	Role Slave of RTCP Delegation [*DPP+16]	330
C.40	Structure of RTCP Distribution [*DPP+16]	330
C.41	Role Distributor of RTCP Distribution (Based on [*DPP+16])	331
C.42	Role Client of RTCP Distribution (Based on [*DPP+16])	331
C.43	Structure of RTCP PositionTransmission [*DPP+16]	332
C.44	Role Sender of RTCP PositionTransmission [*DPP+16]	332
C.45	Role Receiver of RTCP PositionTransmission [*DPP+16]	332
C.46	Structure of RTCP AllPositionsTransmission [*DPP+16]	333
C.47	Role Sender of RTCP AllPositionsTransmission [*DPP+16]	333
C.48	Role Receiver of RTCP AllPositionsTransmission [*DPP+16]	333
D.1	Timed Automata Template UrgencyProvider	341
D.2	Timed Automata Template Connector	341
D.3	Timed Automata Template muml_overtaker_overtaking_36_36	342
D.4	Timed Automata Template muml_OvertakeeCompRTSC_47_54	342
D.5	Timed Automata Template muml_OvertakerCompRTSC_30_40	342
D.6	Timed Automata Template muml_overtaker_Role_32_33	343
D.7	Timed Automata Template muml_overtaker_noOvertaking_41_41	344
D.8	Timed Automata Template muml_overtakee_Role_49_46	345
D.9	Timed Automata Template muml_overtakee_noAcceleration_51_55	346
D.10	Timed Automata Template muml_overtakee_noOvertaking_55_56	346
D.11	Snapshots 1–6 of an Exemplary Counterexample Trace	352
D.12	Snapshots 7–12 of an Exemplary Counterexample Trace	353
D.13	Snapshots 13–18 of an Exemplary Counterexample Trace	354
D.14	Snapshots 19–24 of an Exemplary Counterexample Trace	355
D.15	Snapshots 25–30 of an Exemplary Counterexample Trace	356
E.1	Class Diagram of Package core of MECHATRONICUML’s Metamodel	358
E.2	Class Diagram of Package valuetype of MECHATRONICUML’s Metamodel	358
E.3	Class Diagram of Package behavior of MECHATRONICUML’s Metamodel	359
E.4	Class Diagram of Package actionlanguage of MECHATRONICUML’s Metamodel	360
E.5	Class Diagram of Package connector of MECHATRONICUML’s Metamodel	361

E.6	Class Diagram of Packages protocol and constraint of MECHATRONICUML's Metamodel	362
E.7	Class Diagram of Package realltimestatechart of MECHATRONICUML's Metamodel	365
E.8	Class Diagram of Package one_to_n_schemata of MECHATRONICUML's Metamodel	366
E.9	Class Diagram of Package component of MECHATRONICUML's Metamodel	368
E.10	Class Diagram of Package instance of MECHATRONICUML's Metamodel	370
E.11	Class Diagram of Package mtctl of MTCTL's Metamodel	371
E.12	Class Diagram of Package booleanlogic of MTCTL's Metamodel	371
E.13	Class Diagram of Package predicates of MTCTL's Metamodel	372
E.14	Class Diagram of Package comparables of MTCTL's Metamodel	372
E.15	Class Diagram of Package quantifiers of MTCTL's Metamodel	373
E.16	Class Diagram of Package sets of MTCTL's Metamodel	373
E.17	Class Diagram of Package results of MTCTL's Metamodel	373
E.18	Class Diagram of the DSMC-Options Metamodel	378
E.19	Class Diagram of the DSMC-Options Metamodel	379
E.20	Class Diagram of Packages uppaal and core of UPPAAL's Metamodel	379
E.21	Class Diagram of Package declarations of UPPAAL's Metamodel	380
E.22	Class Diagram of Package global of UPPAAL's Metamodel	380
E.23	Class Diagram of Package system of UPPAAL's Metamodel	381
E.24	Class Diagram of Package statements of UPPAAL's Metamodel	381
E.25	Class Diagram of Package templates of UPPAAL's Metamodel	382
E.26	Class Diagram of Package types of UPPAAL's Metamodel	382
E.27	Class Diagram of Package expressions of UPPAAL's Metamodel	383
E.28	Class Diagram of Package visuals of UPPAAL's Metamodel	384
E.29	Class Diagram of the UTCTL Metamodel	384
E.30	Class Diagram of the UPPAAL Trace Metamodel	386
E.31	Class Diagram of Packages reachability, rtsc, and cic of MECHATRONIC-UML's Trace Metamodel	387
E.32	Class Diagram of Package runtime of MECHATRONICUML's Trace Metamodel	389
E.33	Class Diagram of the Pattern Metamodel	390
E.34	OWL Ontology for Real-Time Coordination Patterns Visualized in EMF	391



LIST OF TABLES

3.1	Comparison of the Requirements Fulfillment for Verification Protocols of CPSs	101
4.1	English Templates and Negated English Templates (Part 1)	126
4.2	English Templates and Negated English Templates (Part 2)	127
4.3	English Templates and Negated English Templates (Part 3)	128
4.4	Comparison of the Requirements Fulfillment for a DSVPL for CPSs	141
5.1	Comparison of the Requirements Fulfillment for a DSMC for CPSs	225
6.1	Comparison of the Requirements Fulfillment for Design Patterns of CPSs	258
C.1	Evaluation Results for RTCP Overtaking (Part 1)	334
C.2	Evaluation Results for RTCP Overtaking (Part 2)	335
C.3	Evaluation Results for RTCP Overtaking (Part 3)	336
C.4	Evaluation Results for RTCP Overtaking (Part 4)	337
C.5	Evaluation Results for RTCP MultiOvertaking.	338
C.6	Evaluation Results for RTCP MultiOvertaking_wSubroleRegion	338
C.7	MTCTL Reachability Properties for Evaluating Hypotheses H3 and H5 (Part 1)	339
C.8	MTCTL Reachability Properties for Evaluating Hypotheses H3 and H5 (Part 2)	340