Final Document

by

Jan Bobolz, Mike Czech, Andreas Dann, Johannes Geismann, Marcus Hüwe, Arthur Krieger, Goran Piskachev, David Schubert, and Rebekka Wohlrab

Supervisors: Prof. Dr. Wilhelm Schäfer, Stefan Dziwok, and Uwe Pohllmann

pg-cybertron@lists.upb.de

Paderborn, October 2014
## Contents

1 Introduction ................................................. 9  
   1.1 This Document .......................................... 9  
   1.2 The MechatronicUML Process: Extended ............... 10

2 Running Example ............................................. 13  
   2.1 Scenario description ..................................... 13  
   2.2 MUML PIM Model .......................................... 14  
   2.3 MUML Hardware description .............................. 17

3 Verification Concepts ........................................ 19  
   3.1 Verification Property Language ......................... 21  
       3.1.1 Goal of the Language and Design Idea ............. 21  
       3.1.2 Basic Semantics .................................... 22  
       3.1.3 Mappings ........................................... 23  
       3.1.4 Predicates ........................................... 25  
       3.1.5 Default Properties .................................. 26  
       3.1.6 MTCTL2English ...................................... 27  
   3.2 Forward Translation ..................................... 27  
       3.2.1 Simple Transformations .............................. 30  
       3.2.2 Protocol to CIC transformation ...................... 30  
       3.2.3 MTCTL Quantifier Normalization .................... 32  
       3.2.4 MTCTL Statically Evaluable Properties Normalization ... 33  
       3.2.5 MTCTL Split Properties Transformation ........... 33  
       3.2.6 MTCTL Properties in the Transformation Chain .... 34  
       3.2.7 Do Event Normalization .............................. 35  
       3.2.8 Urgency Normalization ............................... 36  
       3.2.9 Migration ........................................... 37  
   3.3 Backward Translation ................................... 43  
       3.3.1 Evaluation of MTCTL Property Results ............. 43  
       3.3.2 UPFAAL to MechatronicUML Trace Translation ... 44

4 Deployment Concepts ......................................... 47  
   4.1 Deployment Process ...................................... 47  
   4.2 Extension of the Platform-Description: Operating System ... 49  
       4.2.1 Goal of the Operating System Language .......... 49  
       4.2.2 Language Elements & Semantics ................... 49
Contents

4.3 Allocation .................................................. 51
  4.3.1 Goal of the Allocation Specification Language and its Design Idea .............................................. 51
  4.3.2 Allocation Specification Language Concrete Syntax .............................................................. 53
  4.3.3 Formal Preliminaries ......................................... 56
  4.3.4 Allocation Specification Language Semantics ................................................................. 59
  4.3.5 Transformation to 0-1 ILP ........................................ 61

4.4 Platform-Mapping ............................................ 65
  4.4.1 API-Mapping ................................................ 65

4.5 Derivation of a PSM ........................................... 68
  4.5.1 Step 1-2: Transformation of Continuous Components .......................................................... 69
  4.5.2 Step 3: Creation of PARAMETRIZED REAL-TIME STATE-CHARTS for the transformed Continuous Components ...................................................... 70
  4.5.3 Step 4: Creation of Behavior for the Transformed Hybrid Ports ................................................. 71
  4.5.4 Step 5: Transformation of Continuous Component Instances .................................................... 73

4.6 Middleware and Code Generation ................................. 75
  4.6.1 Middleware .................................................. 76
  4.6.2 Code Generation ............................................. 81

5 Evaluation ..................................................... 85
  5.1 Evaluation of Verification Concepts ............................... 85
    5.1.1 Complete Translation ........................................ 85
    5.1.2 Transparency ................................................ 86
    5.1.3 Semantics Preservation and Correctness ....................... 86
    5.1.4 Efficiency .................................................. 86
  5.2 Evaluation of Deployment Concepts ............................... 87
    5.2.1 Maintainability of the Deployment Process ....................... 87
    5.2.2 Appropriateness of the Deployment Process ....................... 87
    5.2.3 Semantics Preservation and Correctness ....................... 88
    5.2.4 Transparency ................................................ 88
    5.2.5 Efficiency of Code Generation ............................ 89

6 Related Work ................................................ 91
  6.1 Model Checking ............................................. 91
    6.1.1 Theseus ................................................. 91
    6.1.2 TOPCASED ............................................... 91
    6.1.3 First-Order TCTL ......................................... 92
    6.1.4 CurCuMA ................................................ 92
  6.2 Deployment .................................................. 93
    6.2.1 CHROMOSOME ........................................... 93
    6.2.2 AUTOSAR ............................................... 93
    6.2.3 EAST-ADL2 ............................................. 94
    6.2.4 C-Forge ................................................ 94
Contents

6.2.5 ArcheOpterix .................................................. 94

7 Lessons Learned .................................................. 97
7.1 Relying on Third Party Tools .................................. 97
7.2 GQM ................................................................. 97
7.3 Internal Organization ............................................. 98
7.4 Write Meaningful Documents .................................... 100

8 Future Work ....................................................... 101
8.1 Verification ........................................................ 101
8.2 Deployment ........................................................ 103

9 Conclusion ........................................................... 107

A Running Example: Diagrams ...................................... 109

B GQM-Goals .......................................................... 121
B.1 Verification Goals ................................................. 121
  B.1.1 Forward-Translation ....................................... 121
  B.1.2 Back-Translation ......................................... 123
  B.1.3 Verification Property Language .......................... 125
B.2 Deployment Goals ............................................... 126
  B.2.1 Definition of a Deployment Process ...................... 126
  B.2.2 Platform-Mapping and PSM Transformation ............ 128
  B.2.3 Allocation .................................................. 129
  B.2.4 Code Generation .......................................... 131
B.3 Results ............................................................ 131

C MTCTL Syntax ..................................................... 135

D Allocation Specification Language Syntax ...................... 141

E Implemented Accomplishments .................................. 145
E.1 Verification ........................................................ 145
E.2 Deployment ........................................................ 145
E.3 Running Example ................................................ 145
E.4 Misc ................................................................. 146
# List of Figures

1.1 Extended MechatronicUML process ................. 11

2.1 Overtaking scenario ................................. 13

2.2 Component instance configuration with Real-Time Coordination Protocols (VehicleDetection and Overtaking) ............ 14

2.3 Real-Time Statechart: Overtakee role behavior of the Overtaking Real-Time Coordination Protocol .............. 15

2.4 Real-Time Statechart: Overtaker role behavior of the Overtaking Real-Time Coordination Protocol .............. 16

2.5 Real-Time Statechart: Overtaker communicator component behavior ........................................ 16

2.6 Overtaking scenario with Lego robots ............... 17

2.7 Platform instance configuration diagram .............. 18

3.1 The verification process ............................... 20

3.2 Forward translation and participating transformations ........ 28

3.3 Transformation of the of Overtaking protocol to a component instance configuration .......................... 31

3.4 Do event normalized state (compare Figure 2.3 on page 15) ............................. 35

3.5 Excerpt of urgency normalized RTSCs (compare Figure 2.5) .................. 37

3.6 Overview of deriving UPPAAL processes from a CIC ........................................ 38

3.7 Illustration of an array in UPPAAL, which maps port instances to successors according to next references of the CIC .......... 39

3.8 Use of the next array in an UPPAAL template ............. 39

3.9 Multiple message buffers in UPPAAL ................... 40

3.10 Sending a parameterized message in UPPAAL ............... 42

3.11 Intermediate location in UPPAAL NTA .................. 43

3.12 Labeled MechatronicUML snapshot ..................... 46

4.1 Deployment process .................................... 48

4.2 The class OCLContext is the syntactical context for OCL expressions .......................... 58

4.3 Transformation of continuous components ................ 70

4.4 Behavior of the transformed continuous component Light .................. 72

4.5 Behavior of the component OvertakerDriver ............... 73

4.6 Parametrized discrete port instances and parameter bindings .... 74

4.7 Using a middleware, the application layer is independent from the platform layer ................................. 76
List of Figures

4.8 Diagram of middleware components and interfaces ............... 77
4.9 Sequence diagram for message exchange using a MECHATRONICUML-specific middleware .................................................. 79
4.10 Routing message over several ECUs ..................................... 80

9.1 Extended MECHATRONICUML process ................................. 107

A.1 REAL-TIME COORDINATION PROTOCOL Overtaking ............. 109
A.2 REAL-TIME COORDINATION PROTOCOL VehicleDetection ........ 109
A.3 Components ............................................................... 110
A.4 Component instance configuration ...................................... 111
A.5 REAL-TIME STATECHART OvertakerRole from the Overtaking REAL-TIME COORDINATION PROTOCOL ................................. 112
A.6 REAL-TIME STATECHART OvertakeeRole from the Overtaking REAL-TIME COORDINATION PROTOCOL ................................. 112
A.7 REAL-TIME STATECHART DetectorRole from the VehicleDetection REAL-TIME COORDINATION PROTOCOL ................................. 113
A.8 REAL-TIME STATECHART DelegatorRole from the VehicleDetection REAL-TIME COORDINATION PROTOCOL ................................. 113
A.9 REAL-TIME STATECHART OvertakerCommunicator .................. 114
A.10 REAL-TIME STATECHART OvertakeeCommunicator .................. 115
A.11 REAL-TIME STATECHART OvertakerDriveControl ................. 116
A.12 REAL-TIME STATECHART OvertakeeDriveControl ................... 117
A.13 Resource diagram ....................................................... 117
A.14 Resource instance diagram .............................................. 118
A.15 Platform diagram ....................................................... 119
A.16 Platform instance diagram (simplified) .............................. 120

B.1 Detailed overview of metrics for the verification part ............... 132
B.2 Detailed overview of metrics for the Deployment part ............... 133
B.3 Overall compliance of measured values with our hypotheses ....... 134
1 Introduction

As of today, software that controls physical systems must face and master difficult challenges. For example, a system must guarantee a failsafe function for autonomously overtaking vehicles. This means we need a collaboration between mechanical, electronic and software parts, which results in products known as cyber-physical systems. We dedicated our work to the software side of these cyber-physical systems. In particular, we develop software that must be capable of meeting hard real-time requirements in a guaranteed fashion. As part of the study curriculum, the one-year project group Cybertron consists of nine master students at the University of Paderborn. Our specified goal is to enhance confidence in the software of cyber-physical systems by improving the MECHATRONICUML method.

There are two major areas that we worked on:

**Verification**  
Christopher Gerking introduced an approach for formal verification of Coordination Protocols in [1]. We improved upon this approach by expanding the set of MECHATRONICUML language constructs that are respected in the verification process. Furthermore, we created a domain-specific language that allows specifying verification properties for MECHATRONICUML.

**Deployment**  
We devised a process that bridges the gap between platform-independent models in MECHATRONICUML and deployable code. This new process employs domain-specific languages to specify platform parameters, a declarative language for software allocation onto hardware nodes, a MECHATRONICUML-specific middleware, and code generation.

Details about our goals can be found in the group’s requirements specification [2].

1.1 This Document

The purpose of this document is to provide information about the (conceptual) results of the project group, including an evaluation, as well as to share experiences concerning our (software-development) processes. Consequently, this document is aimed at readers who are familiar with MECHATRONICUML and want to understand or build upon our results. This includes future project groups, thesis workers, and MECHATRONICUML developers.
1 Introduction

This document is structured as follows: Next section introduces the extended MechatronicUML Process which provides the reader a high level overview of the method we worked on in this project group. Chapter 2 describes the running example scenario and its model which is used in the following chapters to illustrate the introduced concepts. In the following two chapters (3 and 4), we explain in details all concepts and implementations realized by Cybertron. The evaluation techniques which we used to evaluate our results are explained in Chapter 5. The related work to our domain is summarized in Chapter 6. In Chapter 7, we list relevant lessons we learned during whole year in this project group. We discuss the future work in Chapter 8. Finally, we conclude this document in Chapter 9.

1.2 The MechatronicUML Process: Extended

In this section, we provide a high level overview of the extended MechatronicUML process. Figure 1.1 shows the steps of the extended process by our project group. The process before the project group consisted of steps one to three. We added the step four.

Our extended MechatronicUML process describes the steps required to develop a software for cyber-physical systems using the MechatronicUML method. The MechatronicUML method focuses on the safety and correctness of component-based software. Moreover, MechatronicUML’s goal is the deployment of the software on distributed hardware with limited resources.

As shown in Figure 1.1, the input artifact is the requirements specification created by the requirements engineer and the customer. In step one of the process, the software architect uses the MechatronicUML method to model the software system. The architect creates a platform-independent model (PIM) of the system. MechatronicUML provides tooling for specification of the structure of the system, the behavior of the system, and the communication.

In step two, the software architect and optionally the safety engineer verify the modeled behavior of the system using the PIM as an artifact.

In step three, the hardware architect specifies the platform, which will be used to execute the software. Therefore, he uses MechatronicUML tooling to create a hardware model [3]. Notice that step three is executed in parallel to steps one and two.

In step four, the PIM should be deployed on the hardware described in the hardware model. In this step, first the allocation engineer needs to allocate the software components of the PIM to the hardware elements of the hardware model. Afterwards, he and optionally the target platform expert can generate code. Finally, the generated code is executed on the hardware.
1.2 The MechatronicUML Process: Extended

Figure 1.1: Extended MechatronicUML process
2 Running Example

In this chapter, we introduce our running example used for illustration of our concepts in the following chapters. We used the extended MECHATRONICUML process to develop the running example and create the final demonstration of the project group. The goal of this chapter is to present a scenario which is suitable for the MECHATRONICUML method and will be used in the following chapters to support the presented concepts. The following section provides a description of our scenario. In Section 2.2, we introduce the platform independent model for our scenario. Finally, in Section 2.3, we show the hardware model.

2.1 Scenario description

We introduce a scenario where one vehicle overtakes another vehicle. Both vehicles drive in the same direction. In this safety-critical scenario, many real-time requirements needs to be met. Moreover, these vehicles consists of distributed resources communicating via different communication protocols, which increase the complexity of the software deployed on the hardware. The vehicles are capable of exchanging messages in order to ensure the safety of the overtaking. The rear vehicle (overtaker) wants to overtake the front vehicle (overtakee). This is shown in Figure 2.1. When the overtaker gets close enough (fixed distance threshold) to the overtakee, the overtaking process is executed. Our main goal is to improve the safety from software perspective in such scenarios. For example, the overtakee should not increase its velocity while the overtaking process.

![Figure 2.1: Overtaking scenario](image-url)

2 Running Example

2.2 MUML PIM Model

In this section, we show the scenario from the previous section modelled in MECHATRONICUML. Since the modelling part of the process is not the focus of the project group, we omit some of the modelling details and show only the relevant for our work. For detailed description of MECHATRONICUML, the readers may refer to the MECHATRONICUML specification (v0.4) [4].

![Component instance configuration diagram]

Figure 2.2: Component instance configuration with Real-Time Coordination Protocols (VehicleDetection and Overtaking)
2.2 MUML PIM Model

The static structure of the system is represented as follows (Figure 2.2 shows the component instance configuration diagram). There are two structured component instances representing the overtaker and overtakee vehicles. Both structured component instances have broadly similar internal structure. In each structured component instance, there are two software component instances, the first one (driver) responsible for the driving behavior, sensor inputs, and actuator outputs, and the second one (communicator) responsible for the communication behavior with the communicator of the other structural component instance. There are also continuous component instances representing the sensors (light sensor and distance sensor) and actuators (motors). Additionally, we can see the two embedded REAL-TIME COORDINATION PROTOCOL (RTCP) instances: VehicleDetection and Overtaking. The RTCP instances define a communication contract between two entities, in this case two discrete port instances. They define which types of messages can be exchanged between the ports. An overview of all diagrams of the running example is available in Appendix A. In this chapter, we show only the most relevant.

In Figure 2.2, we show two REAL-TIME COORDINATION PROTOCOL instances. The VehicleDetection is contained in the overtaker vehicle. When the driver component instance detects the overtakee using the distance sensor, it triggers the communicator through VehicleDetection Coordination Protocol instance. It sends a request for overtaking to the overtakee. When the response is received, the communicator triggers the driver to execute the overtaking. Finally, when the overtaking is finished, the driver triggers the communicator to inform the overtakee. The second Coordination Protocol instance (Overtaking) is used for the communication between both vehicles, where the communicator component instances of both vehicle exchange messages. This protocol instance is used when the overtaker requests the overtakee to overtake.

![Real-Time Statechart: Overtakee role behavior of the Overtaking Protocol](image)

Figure 2.3: REAL-TIME STATECHART: Overtakee role behavior of the Overtaking Protocol
2 Running Example

Figure 2.4: **Real-Time Statechart**: Overtaker role behavior of the Overtaking Real-Time Coordination Protocol

Figure 2.5: **Real-Time Statechart**: Overtaker communicator component behavior
In the following, we show how the behavior of the overtakee and overtaker roles is modelled. Figures 2.3 and 2.4 show the Real-Time Statecharts (RTSCs) of these roles. In the overtakee role, when the Init state is active, it may receive requestOvertaking message from the overtaker role and the Requested state becomes active. Then, an acceptOvertaking message is sent and the NoAcceleration state becomes active. The velocity of the overtakee is continuously checked by the do-event in the action checkVel to make sure that it is not increased. When a finishedOvertaking message is consumed, the initial state becomes active. The overtaker role has a corresponding behavior where the same messages are exchanged in opposite direction.

The RTSC of the communicator component instance of the overtaker is shown on Figure 2.5. As defined by MechatronicUML method, there is one state which is initial and always active. This state contains one sub-region for the discrete port OvertakerPort (which is a refinement of the role OvertakerRole) and second sub-region for the DelegatorPort. On one hand, the OvertakerPort sub-region is communicating to the communicator of the overtakee by sending request message, receiving accept message, and sending finish message. On the other hand, the DelegatorPort sub-region communicates with the driver of the overtaker to inform it about the current state of the overtaking process. Both sub-regions communicate via three synchronization channels initiated, accepted, and executed, in order to coordinate.

For detailed overview of all diagrams of the system, the readers may refer to the Cybertron User guide\(^1\).

### 2.3 MUML Hardware description

We implemented the scenario described in Section 2.1 using LEGO mindstorms NXT. We built two LEGO robots to represent the two vehicles. The lane, on which both vehicles drive, is represented by a black line printed on the ground. Figure 2.6 illustrates the hardware and the environment of our scenario.

![Overtaking scenario with Lego robots](image)

Figure 2.6: Overtaking scenario with Lego robots

In order to define the hardware elements that we want to use, MechatronicUML offers hardware description diagrams to model the hardware. The plat-

\(^1\)https://trac.cs.upb.de/mechatronicuml/wiki/cybertron/userguide
form instance configuration of our example is shown in Figure 2.7. Internally these two platform instances contain several resource instances of different type. From the internal structure, we see that both robots have two bricks, two motors, and a light sensor (used for line following). Additionally, the Overtaker has an ultrasonic sensor (used to detect obstacles in the front). The Vehicles are connected via Bluetooth connection.

More details about the description of the hardware can be found in the MECHATRONICUML Hardware Platforms Description Method [3].

![Platform instance configuration diagram](image_url)

Figure 2.7: Platform instance configuration diagram
3 Verification Concepts

This chapter presents the developed verification concepts. As described in Chapter 1, we enhanced the existing verification approach \[1\] for \textsc{Real-Time Coordination Protocols} and implemented new concepts for it. These concepts achieve full transparency of the \textsc{Uppaal} model checker\(^1\), introduce support for a larger class of \textsc{MechatronicUML} models, and enable easier debugging of Coordination Protocol behavior.

We extended Gerking's process for verification \[1\]. Our new process works as follows: After a Coordination Protocol has been modeled with \textsc{MechatronicUML}, the software architect specifies formal verification properties based on informal requirements using a new Verification Property Language called MTCTL. Next, the user can verify the resulting MTCTL properties automatically (cf. Figure 3.1). To execute the verification, the model checker \textsc{Uppaal}\(^2\) is used (see [5, 6] for details about \textsc{Uppaal}). We assume the reader of this document to be familiar with \textsc{Uppaal}.

As depicted in Figure 3.1, we translate a Coordination Protocol to \textsc{Uppaal}'s NTA and MTCTL properties to \textsc{Uppaal}'s TCTL properties. Then, \textsc{Uppaal} checks the NTA and its properties. We implemented two use cases, corresponding to two different kinds of output from \textsc{Uppaal}: First, verify many properties at once and display their results. In this case, \textsc{Uppaal} verifies multiple properties at once and we translate these simple properties' results back to more complex MTCTL property results. Second, if one of these verification results is not expected, the user can choose a property to display a counterexample trace (in case a safety property is not fulfilled) or a reachability trace (in case a reachability property is fulfilled). In this case, \textsc{Uppaal} creates a trace for a single property and we translate this trace back to a \textsc{MechatronicUML} trace. Counterexample traces contain helpful information about which specific steps the \textsc{MechatronicUML} model passes to arrive in an unsafe situation. The software architect can use this information to fix a fault in the protocol's behavior. Then, the verification process can be repeated until the protocol is proven to fulfill its requirements.

The following sections explain our developed verification concepts in more detail. The Verification Property Language MTCTL is described in Section 3.1. The translation of \textsc{MechatronicUML} models to \textsc{Uppaal} NTA is presented in Section 3.2. In Section 3.3, we then explain the translation of \textsc{Uppaal} results and traces back to \textsc{MechatronicUML}.

\(^1\) Here, transparency means that the user does not need to have any knowledge about the \textsc{Uppaal} model checker to apply the verification process

\(^2\)http://www.uppaal.com/
Figure 3.1: The verification process
3.1 Verification Property Language

This section introduces the language MTCTL (MechatronicUML TCTL), which is used to describe the verification properties to be formally verified in the verification process. MTCTL is based on predicate logic and TCTL [7]. It is specifically tailored toward MechatronicUML and contains many MechatronicUML-specific predicates. For example, an MTCTL property is

\[ \text{AG (exists}\ b: \text{Buffers}) \quad \text{messageInBuffer(OvertakingMessages.requestOvertaking, b)} \quad \text{implies AF} \quad \text{transitionFiring(overtakee.OvertakeRoleBehavior.Init to Requested)}; \]

which describes that whenever the requestOvertaking message is in some buffer, eventually the Init_to_Requestted transition will fire. \(^3\)

In the extended MechatronicUML process, verification properties are specified after the initial MechatronicUML platform-independent design model of the software has been created. Software architects specify the formal MTCTL verification properties based on initially created top-level requirements [8] in order to verify them.

Because other stakeholders, like safety analysts [9], are also involved with the verification of the system, the language should be understandable even without a computer science background [8]. Therefore, we created the MTCTL2English feature, which allows us to automatically create more easily understandable natural language sentences from formal MTCTL properties.

This section is structured as follows. Section 3.1.1 describes the goals and the core design ideas of MTCTL. In Section 3.1.2, we introduce the reader to the semantics of our language. Based on this, we present the language elements and their meaning in Sections 3.1.3 and 3.1.4. In Section 3.1.5, we present default properties that are typically desirable to hold for any model. Finally, we introduce the MTCTL2English feature in Section 3.1.6.

3.1.1 Goal of the Language and Design Idea

In the previous verification approach of MechatronicUML, the properties to be verified were specified on the Uppaal level. This made it necessary for the user to know how MechatronicUML models were translated to Uppaal. To enable transparency of Uppaal, we created a Verification Property Language that allows the user to specify properties with the terminology of MechatronicUML, without requiring knowledge of the resulting Uppaal model.

We established the following high-level goals with MTCTL ( [8] describes these goals in more detail):

- Express safety, reachability, liveness, and deadlock properties

[^3]: Since Uppaal does not support nested temporal quantifiers, this is valid in MTCTL but not supported for verification. However, the following equivalent property is supported: exists(b: Buffers) messageInBuffer(OvertakingMessages.requestOvertaking, b) leadsTo transitionFiring(overtakee.OvertakeRoleBehavior.Init_to_Requestted);
3 Verification Concepts

- Reference the MECHATRONICUML platform-independent design model (e.g., states and messages)
- Automatically translate properties into the TCTL subset that UPPAADL is able to verify
- Be understandable for all involved stakeholders and provide an appropriate abstraction level

Based on these goals, we developed the Verification Property Language MTCTL. Our design idea is to base our language on both TCTL and predicate logic. The basis on TCTL facilitates the later translation to UPPAADL queries. Using predicate logic as a basis shall help to define simple and understandable, yet powerful properties. For example, it allows us to create simple statements checking, for example, whether each state can possibly become active:

\[
\text{forall}(s:\text{States}) \ \text{EF} \ \text{stateActive}(s)
\]

Because the syntax is very similar to predicate logic syntax (augmented with temporal quantifiers \text{AG}, \text{AF}, \text{EG}, \text{EF}, \text{leadsTo})

We refrain from a formal syntax description here and instead refer to the grammar (Appendix C) for details. The following sections explain the semantics of our language and its elements in further detail.

3.1.2 Basic Semantics

MTCTL is based on a combination of predicate logic and TCTL. The runtime state space of MECHATRONICUML can be described as an (infinite) tree. Each tree node references a system state (called \textit{snapshot} here to avoid confusion with states in REAL-TIME STATECHARTS). 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. For more information on MECHATRONICUML snapshots and (delay- or action-) snapshot transitions, we refer to future versions of the MECHATRONICUML technical report [4].

The semantics of MTCTL is defined by interpretations \( I : \text{MTCTL} \rightarrow \{\text{true}, \text{false}\} \) that map a valid MTCTL property to a truth value. Every MTCTL property is defined in the context of a verifiable MECHATRONICUML element (i.e., a Coordination Protocol, Atomic Component, or Component Instance Configuration) and the interpretation depends on the concrete element.

For a snapshot \( s \), we call \( I_s \) the interpretation for the snapshot subtree of \( s \). Let \( s \) be an arbitrary snapshot, \( \Phi, \Psi \in \text{MTCTL} \) sentences. Let \( D_s \) be the set of snapshots in the subtree of \( s \). Let \( P_s \) be the set of non-extendable paths through the subtree of \( s \), starting in the root node. We denote such a path \( p \in P_s \subseteq 2^{D_s} \) as:

\[\text{forall}(s:\text{States}) \ \text{EF} \ \text{stateActive}(s)\]

Because the syntax is very similar to predicate logic syntax (augmented with temporal quantifiers \text{AG}, \text{AF}, \text{EG}, \text{EF}, \text{leadsTo}), we refrain from a formal syntax description here and instead refer to the grammar (Appendix C) for details. The following sections explain the semantics of our language and its elements in further detail.

3.1.2 Basic Semantics

MTCTL is based on a combination of predicate logic and TCTL. The runtime state space of MECHATRONICUML can be described as an (infinite) tree. Each tree node references a system state (called \textit{snapshot} here to avoid confusion with states in REAL-TIME STATECHARTS). 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. For more information on MECHATRONICUML snapshots and (delay- or action-) snapshot transitions, we refer to future versions of the MECHATRONICUML technical report [4].

The semantics of MTCTL is defined by interpretations \( I : \text{MTCTL} \rightarrow \{\text{true}, \text{false}\} \) that map a valid MTCTL property to a truth value. Every MTCTL property is defined in the context of a verifiable MECHATRONICUML element (i.e., a Coordination Protocol, Atomic Component, or Component Instance Configuration) and the interpretation depends on the concrete element.

For a snapshot \( s \), we call \( I_s \) the interpretation for the snapshot subtree of \( s \). Let \( s \) be an arbitrary snapshot, \( \Phi, \Psi \in \text{MTCTL} \) sentences. Let \( D_s \) be the set of snapshots in the subtree of \( s \). Let \( P_s \) be the set of non-extendable paths through the subtree of \( s \), starting in the root node. We denote such a path \( p \in P_s \subseteq 2^{D_s} \) as:

4While MTCTL supports all these elements, the verification with UPPAADL is currently restricted to Coordination Protocols.
5 The subtree of a snapshot \( s \) is the (usually infinite) subtree of any node in the TCTL tree that references \( s \). This is well-defined because all of these node subtrees are equal.
the set of visited snapshots (meaning that two paths visiting different tree nodes but the same snapshots are equivalent for our purposes).

The interpretation $I$ of an MTCTL property is defined recursively as follows:

- $I = I_r$, where $r$ is the initial snapshot.
- $I_s(\text{forall}(x:\text{SET}) \ \Phi) = \text{true}$ if and only if for all $\bar{x}$ in SET, $I_s(\Phi[x/\bar{x}]) = \text{true}$
- $I_s(\text{exists}(x:\text{SET}) \ \Phi) = \text{true}$ if and only if a $\bar{x}$ in SET exists such that $I_s(\Phi[x/\bar{x}]) = \text{true}$
- $I_s(\Phi \ \text{and} \ \Psi) = \text{true}$ if and only if $I_s(\Phi) = \text{true}$ and $I_s(\Psi) = \text{true}$
  - Analogously for or, not, and implies
- $I_s(\text{AG} \ \Phi) = \text{true}$ if and only if $\forall p \in P_s \ \forall s \in p: I_s(\Phi) = \text{true}$
- $I_s(\text{AF} \ \Phi) = \text{true}$ if and only if $\forall p \in P_s \ \exists s \in p: I_s(\Phi) = \text{true}$
- $I_s(\text{EG} \ \Phi) = \text{true}$ if and only if $\exists p \in P_s \ \forall s \in p: I_s(\Phi) = \text{true}$
- $I_s(\text{EF} \ \Phi) = \text{true}$ if and only if $\exists p \in P_s \ \exists s \in p: I_s(\Phi) = \text{true}$
- $I_s(\Phi \ \text{leadsTo} \ \Psi) = I_s(\text{AG}(\Phi \ \text{implies} \ \text{AF} \ \Psi))$

Using this recursive definition, combined with interpretation definitions of predicates (Section 3.1.4), the interpretation of a valid MTCTL property can be computed. For the interpretation of predicates, it should be noted that predicates may be parameterized with mappings. This is explained in Section 3.1.3.

### 3.1.3 Mappings

In MTCTL, mappings can be used as arguments to predicates. As a (contrived) example, the property

$$\text{forall}(t:\text{Transitions}) \ (\text{stateActive}(\text{sourceState}(t)) \ \text{leadsTo} \ \text{transitionFiring}(t))$$

expresses that whenever the source state of a transition is active, this transition will eventually fire. Here, the interpretation $I_s$ of stateActive depends on the value of the mapping sourceState. Furthermore, sourceState and transitionFiring both depend on the value of the (placeholder) mapping $t$. More generally, all parameters of predicates are mappings, and so are the parameters of mappings themselves.

Formally, the value of a mapping is defined by functions $V_s : \text{MAPPINGS} \rightarrow U$, for some universe $U$, potentially depending on a snapshot $s$. A mapping may be either a constant (like $t$ above) or its value may depend on its parameters (like sourceState). Furthermore, a mapping may be either dynamic or static. The value of a dynamic mapping (potentially) depends on a snapshot (like bufferMessageCount). The value of a static mapping does not depend on a snapshot (like sourceState). This section lists supported mappings and defines their values.

23
Direct References to the PIM  MTCTL supports referencing elements from the PIM model, such as states, transitions, buffers, clocks, variables, etc. Direct references consist of a qualified name of the static element, optionally followed by a reference to an instance hosting the instantiated static element at runtime (in brackets). For example,

\texttt{overtaker.Overtaking[overtakerComponent.overtakerPortInstance]}

references the state \texttt{Overtaking} of port \texttt{overtaker} belonging to \texttt{overtaker-PortInstance} of component instance \texttt{overtakerComponent}.

The reference to an instance disambiguates the direct reference. For example, a single (static) state may be instantiated multiple times in a 1:n communication protocol. If a static element is only instantiated once, the instance reference can be omitted.

For specifying instances, we assume that a subrole behavior \texttt{REAL-TIME STATECHART} (and its contents) belong to a \texttt{DiscreteSingleInteractionEndpointInstance}, adaptation behavior belongs to a \texttt{DiscreteMultiInteractionEndpointInstance}, and all other \texttt{REAL-TIME STATECHARTS} belong to an \texttt{AtomicComponentInstance}.

Formally, a direct reference $x$ is a constant mapping that can be static or dynamic. Its value $V_s(x)$ is:

- Its static value, if $x$ is an integer, or message type
- Its referenced instantiated element (static), if $x$ is a state, transition, message buffer, statechart, or instance (e.g., ConnectorEndpointInstance)
- Its dynamic value (i.e. depending on a snapshot), if $x$ is a clock or a \texttt{MECHATRONICUML} variable

Sets  \texttt{forall} and \texttt{exists} expressions reference a set of \texttt{MECHATRONICUML} elements, allowing the user to easily specify properties over a large number of elements without enumerating them. Where applicable, the following sets contain all possible values connected to the verifiable element (e.g., the Coordination Protocol). The following sets are supported:

- Integer intervals, such as $[0,10]$: The set of integers in the specified bounds
- \texttt{Buffers}: The set of instantiated incoming message buffers
- \texttt{Clocks}: The set of instantiated clocks throughout all \texttt{REAL-TIME STATECHARTS}
- \texttt{MessageTypes}: The set of all message types that may be sent or received according to the ports/roles
- \texttt{States}: The set of instantiated states throughout all \texttt{REAL-TIME STATECHARTS}
- \texttt{Transitions}: The set of instantiated transitions throughout all \texttt{REAL-TIME STATECHARTS}
3.1 Verification Property Language

- **Instances< Element >**: The set of instantiated instances of the Connector Endpoint or Atomic Component 'Element'
- **Subinstances< Element >**: The set of instantiated DiscreteSingleInteractionEndpoints of a multi-port or multi-role 'Element'

Formally, these sets contain constant mappings that may be used as parameters of predicates or other mappings. It should be noted that for Buffers, Clocks, States, and Transitions, the set contains all runtime instances of these element kinds. For example, \( \text{forall}(s:\text{States}) \) contains all states that exist at runtime (i.e. all meaningful static state - instance pairs).

**Non-constant Mappings**  Currently, MTCTL supports the following non-constant mappings (i.e. mappings with a parameter):

- \( \text{bufferMessageCount}(\text{buffer}) \), where \( V_s(\text{bufferMessageCount}(\text{buffer})) \) is the number of messages contained in \( V_s(\text{buffer}) \) in snapshot \( s \)
- \( \text{sourceState}(\text{transition}) \), where \( V_s(\text{sourceState}(\text{transition})) \) is a reference to the source state of \( V_s(\text{transition}) \) (instance-aware)
- \( \text{targetState}(\text{transition}) \), defined analogously to \( \text{sourceState} \)

### 3.1.4 Predicates

MTCTL supports dynamic and static predicates that can be used in expressions. Static predicates are those that can be evaluated independently of a snapshot context (i.e. before model checking time). On the other hand, the interpretation of dynamic predicates (potentially) depends on a specific snapshot. Therefore, dynamic predicates are enforced to be (indirectly) bound to a temporal quantifier. The supported dynamic predicates and their interpretations are:

- \( I_s(\text{deadlock}) = \text{true if and only if all snapshot transitions reachable from } s \) are delay transitions
- \( I_s(\text{bufferOverflow}) = \text{true, if and only if at some point before } s, \) a buffer (or a connector) had to reject a message while utilizing its full capacity
- \( I_s(\text{messageInBuffer}(\text{messageType}, \text{buffer})) = \text{true, if and only if in } s, \) a message of type \( V_s(\text{messageType}) \) is currently included in the specified buffer
- \( I_s(\text{messageInTransit}(\text{messageType})) = \text{true, if and only if in } s, \) a message of type \( V_s(\text{messageType}) \) is currently being transmitted to the receiver’s buffer, but has not arrived yet (the message may still be lost)
- \( I_s(\text{stateActive}(\text{state})) = \text{true if and only if } V_s(\text{state}) \) is active in \( s \)
- \( I_s(\text{transitionFiring}(\text{transition})) = \text{true if and only if } V_s(\text{transition}) \) is currently firing in \( s \). This predicate also covers non-time-consuming transitions, i.e. there is a dedicated snapshot whenever such a transition is firing.
- Comparisons (see next paragraph)
3 Verification Concepts

The supported static predicates and their interpretations are:

- \( I_s(substateOf(state, superstate)) = true \) if and only if \( V_s(state) \) is a (direct or indirect) substate of \( V_s(superstate) \) (non-reflexive, instance-aware).
- \( I_s(stateInStatechart(state, statechart)) = true \) if and only if \( V_s(state) \) is (directly or indirectly) embedded in \( V_s(statechart) \) (instance-aware).
- Comparisons (see next paragraph)

Comparisons Comparisons may be either static or dynamic predicates, depending on what is being compared. In MTCTL, you can compare elements using the operators \( =, \neq, \leq, <, \geq \) and \( > \):

A comparison compares the values of two supplied mappings. Formally, \( I_s(x = y) = true \) for two mappings \( x \) and \( y \), if and only if \( V_s(x) = V_s(y) \). (analogously for the other operators)

3.1.5 Default Properties

This section introduces the default MTCTL properties we specified. Each of these properties is valid on any verifiable element. In the MechatronicUML TOOL SUITE, you can easily add these default properties as an initial assortment of generic verification properties, creating an easy starting point for the verification. Table 3.1 shows the default properties and explanations.

<table>
<thead>
<tr>
<th>MTCTL Property</th>
<th>English Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG not deadlock</td>
<td>A deadlock never occurs.</td>
</tr>
<tr>
<td>AG not bufferOverflow</td>
<td>A buffer overflow never occurs.</td>
</tr>
<tr>
<td>forall(t : Transitions) EF transitionFiring(t)</td>
<td>There is no transition that can never be fired.</td>
</tr>
<tr>
<td>forall(s : States) EF stateActive(s)</td>
<td>There is no state that can never be active.</td>
</tr>
<tr>
<td>forall(m : MessageTypes) exists(b : Buffers) EF messageInBuffer(m, b)</td>
<td>There is no message type that cannot possibly arrive in a buffer.</td>
</tr>
<tr>
<td>forall(m : MessageTypes) EF messageInTransit(m)</td>
<td>There is no message type that cannot possibly be in transit.</td>
</tr>
<tr>
<td>forall(b : Buffers) EF bufferMessageCount(b) &gt;= 1</td>
<td>There is no incoming message buffer that is always empty.</td>
</tr>
</tbody>
</table>

Table 3.1: Default Properties with their explanation in natural language
3.1.6 MTCTL2English

This section introduces our MTCTL2English feature. The feature has been introduced to provide easily understandable properties for stakeholder communication in the form of structured English sentences.

In the extended MECHATRONICUML process, explained in Section 1.2, the verification step allows to verify MECHATRONICUML models with formal MTCTL properties. To allow safety analysts and other stakeholders without knowledge of TCTL to understand these formal properties more easily and to check the consistency to the initial requirements, we developed the MTCTL2English feature.

MTCTL2English has been developed as an Xtend\(^6\) Model-to-Text transformation. We also considered the implementation of a structured English parser, but decided to limit the effort and increase the flexibility of our approach and choose the Model-to-Text transformation instead. It was an advantage that we did not have to consider the parseability of our language. For instance, this allowed us to generate more natural sentences and include negations of properties.

For example, consider the following MTCTL property:

\[
\text{AG}(\text{stateActive}(\text{Overtaker.overtaking}) \implies \text{stateActive}(\text{Overtakee.noAcceleration}))
\]

The MTCTL2English feature creates the following equivalent sentence:

“It invariantly holds that if Overtaker.overtaking is active then Overtakee.noAcceleration is active.”

If this property is not fulfilled, the MTCTL2English feature creates the following English sentence when showing the results:

“The verification result is that it can possibly happen that Overtaker.overtaking is active and Overtakee.noAcceleration is not active.”

3.2 Forward Translation

This section describes the transformation of MECHATRONICUML’s REAL-TIME COORDINATION PROTOCOLS (RTCP) with MTCTL properties into UPPAAL models with UPPAAL queries. We call this transformation forward translation. The concept and an initial implementation were developed by Gerking in the course of his master’s thesis [1].

Figure 3.2 visualizes the forward translation and the participating transformations. These transformations are named after the syntax elements or modeling concept they are mainly concerned with. Figure 3.2 distinguishes between transformations that existed before the start of CYBERTRON and were not adapted by the project group (blue), preexisting transformations that were adapted (yellow), and transformations that were newly created (green). We do not explain concepts of the first kind of transformations in this document ( [1] describes these in detail).

\(^6\)https://www.eclipse.org/xtend/
Figure 3.2: Forward translation and participating transformations
Additionally, certain transformations are marked as being *normalizations*. A normalization is a transformation “used to decrease the syntactic complexity, e.g., by translating syntactic sugar into more primitive language constructs” [10]. In our case, all normalizations operate on the MECHATRONICUML models and substitute model elements that have no corresponding UPPAAL element with elements or constructs that can easier be translated to UPPAAL. As an advantage, these transformations are also particularly well suited for reuse in the context of other transformation chains (e.g., for translations to other model checking languages or to simulation languages).

Generally, Section 3.2.1 describes simple transformations of the transformation chain. MTCTL-specific transformations are covered in Sections 3.2.3 - 3.2.5. Complex transformations that operate on the MECHATRONICUML level and mainly concern RTSC elements are covered in the Sections 3.2.2, 3.2.7, and 3.2.8. Finally, the actual migration to UPPAAL is explained in Section 3.2.9. The transformations in the transformation chain are:

- **Protocol to CIC**: creation of a component instance configuration for the protocol to verify (see Section 3.2.2)
- **Quantifier**: normalizes models w.r.t. `forall` or `exists` quantifiers of MTCTL properties (see Section 3.2.3)
- **Statically Evaluable Properties**: evaluates parts of properties that are evaluable without using UPPAAL (see Section 3.2.4)
- **Split Properties**: splits complex MTCTL properties into multiple simple UPPAAL queries (see Section 3.2.5)
- **Names**: adds prefixes the names of named elements (see Section 3.2.1)
- **Deadline**: normalizes models w.r.t. transition deadlines (see [1])
- **Composite Transitions**: normalizes models w.r.t. transitions to composite states (see [1])
- **Do-Events**: normalizes models w.r.t. do events of states (see 3.2.7)
- **Hierarchy**: flattens the RTSCs (see [1])
- **Entry/Exit Events**: normalizes models w.r.t. entry and exit events of states (see Section 3.2.1)
- **Time Units**: scales time values to a common time unit (see Section 3.2.1)
- **Urgent Transitions**: preparations to translate urgent transitions to UPPAAL’s edges (see Section 3.2.8)
- **Migration**: migrates the MECHATRONICUML models to UPPAAL (see Section 3.2.9)
3 Verification Concepts

3.2.1 Simple Transformations

This section covers transformations of the transformation chain that are conceptually simple and therefore can be described very briefly.

**Names** There are no restrictions on the names that named elements of MECHATRONICUML can have. The migration (see Section 3.2.9) translates some of these elements into variables that are declared in UPPAAL’s global declarations. The language these declarations are defined in supports certain keywords. To avoid conflicts between keywords and variable names, the Names transformation adds a prefix (namely 'MumlElement') to all names of named elements.

Additionally, certain characters (like '-') must not be used in Uppaal identifiers. Therefore, the Names transformation substitutes these with an '_'.

**Entry/Exit Events** MECHATRONICUML supports entry and exit events of states as you can see in the Requested and NoAcceleration states of Figure 2.3 (page 15). This syntax feature is not supported by UPPAAL. Hence, the Entry/Exit Events normalization copies entry events to all incoming transitions and exit events to all outgoing transitions of the corresponding state. This functionality was already realized by the preexisting implementation of the migration by Gerking (i.e. entry/exit events were directly translated to UPPAAL). However, to reduce the complexity of the migration, we extracted the functionality into a normalization (i.e. entry/exit are now resolved on the MECHATRONICUML level).

**Time Units** MECHATRONICUML supports several time units, e.g., seconds or milliseconds, whereas UPPAAL does not support time units at all. To ensure a semantics-preserving translation from MECHATRONICUML models to UPPAAL models, all time values are scaled to the smallest time unit in use.

Exactly this is done by the Time Units normalization. This functionality was also already existing in the pre-CYBERTRON state of the migration, but extracted into a normalization for complexity-reduction reasons.

3.2.2 Protocol to CIC transformation

This section covers the protocol to CIC transformation, which is the first transformation of the forward translation (see Figure 3.2). Here, we discuss the purpose of this transformation and show its functionality on the running example introduced in Section 2.2.

The state-based behavior of REAL-TIME COORDINATION PROTOCOLS’ (RTCPs’) [4] roles is defined by REAL-TIME STATECHARTS (RTSCs). In the preexisting version of the forward translation, each of these RTSCs was translated into an

---

7In some (uncommon) cases, this may make two different names equal. A better solution is to forbid these characters in MECHATRONICUML identifiers as well.
UPPAAL template [11]. Since only 1:1 RTCPs were supported, each template was instantiated exactly once.

To support 1:n RTCPs, we need a more sophisticated instantiation concept. This mainly results from the need of certain RTSCs being instantiated multiple times, because the elements they specify a behavior for are instantiated multiple times (see Section 3.2.9 for details).

The first decision we had to make was whether to let UPPAAL choose the concrete role multiplicity nondeterministically during model checking or to verify only a single multiplicity in a verification run and let the user choose it. We decided for the user input, because the state space can grow exponentially with the role multiplicity. Choosing a role multiplicity nondeterministically means that all (already large) valid multiplicity state spaces are combined into a single state space. This can easily result in a state space size that is unfeasible to check. In our chosen approach, only the state space for a single specific role multiplicity is included in the checked state space. We note that no functionality is lost using this approach, since all multiplicities can be checked separately.

The next decision was whether to handle instantiating certain objects multiple times completely in the course of the migration (see Section 3.2.9) or to move certain parts of the instantiation logic into a precedent transformation. The migration is conceptually and also implementation-wise very complex [11]. To reduce this complexity, we decided for a precedent transformation, which is the protocol to CIC transformation.

![Figure 3.3: Transformation of the Overtaking protocol to a component instance configuration](image)

This transformation creates a component instance configuration (CIC) (see [4]) for the protocol to verify while respecting the role cardinality given by the user. Let us assume that the Overtaking RTCP (see Figure 2.2) is a 1:n RTCP with a role cardinality of [1..3] for the overtaker role. Furthermore, consider a role multiplicity of 2 given by the user. Figure 3.3 demonstrates how this protocol
3 Verification Concepts

is transformed into a CIC by the protocol to CIC transformation for the user-defined role multiplicity two. Additionally, the corresponding component model is generated.

In this generated CIC, there is one overtaker component instance with a multiport instance that refines the overtaker role. Also, there are two overtakee component instances with one single port instance each that refines the overtakee role. The behavior of the ports is exactly the same as the behavior of the roles. Moreover, the MTCTL properties (see Section 3.1) defined for the Overtaking RTCP are copied and adapted to the newly created CIC.

There are two advantages resulting from this transformation. First, we were able to reduce the complexity of the migration compared to a solution that handles the role instantiation within the migration itself (for more migration specific information please see Section 3.2.9). Second, all following transformations of the forward translation take this CIC model as an input. Meaning, CICs of this structure (only two component types, maximal one multi port, etc.) are already supported by all of these transformations (including the migration). Thereby, crucial steps to the verification of CICs are already reflected in the current state of the forward translation.

3.2.3 MTCTL Quantifier Normalization

This section introduces the quantifier normalization which is the first one of the MTCTL normalizations. The goal of the quantifier normalization is to remove forall and exists quantifiers from MTCTL properties and replace them with equivalent statements.

Since all sets that can be referenced in MTCTL quantifiers are finite, this transformation simply replaces forall(x:SET) Φ with Φ[x/x_1] and ... and Φ[x/x_n] for x_1, ..., x_n ∈ SET (analogously for exists with a disjunction).

This normalization step is needed because UPPAAL does not have a concept of the sets referenced in quantifier expressions (such as States, Transitions, Buffers, ...).

We use the following MTCTL property to illustrate the quantifier normalization step:

forall \( s: \text{States} \) \( (\text{stateInStatechart}(s, \text{Overtakee.OvertakeeRoleBehavior}) \implies \text{EF stateActive}(s)) \)

It expresses that each state contained in the REAL-TIME STATECHART OvertakeeRoleBehavior can possibly become active at some point in time.

The quantifier normalization step transforms the property to the following:

\( (\text{stateInStatechart}(\text{Overtaker.Init, Overtakee.OvertakeeRoleBehavior}) \implies \text{EF stateActive}(\text{Overtaker.Init})) \)

and \( (\text{stateInStatechart}(\text{Overtakee.Requested, Overtakee.OvertakeeRoleBehavior}) \implies \text{EF stateActive}(\text{Overtakee.Requested})) \)

... and ...

and \( (\text{stateInStatechart}(\text{Overtakee.Init, Overtakee.OvertakeeRoleBehavior})) \)

32
implies EF stateActive(Overtakee.Init)).}

The quantifier normalization ensures that the quantifiers exists and forall are removed from the MTCTL properties.

### 3.2.4 MTCTL Statically Evaluable Properties Normalization

In this section, we describe the *statically evaluable properties normalization* which simplifies MTCTL expressions that can be statically evaluated.

The goal of the *statically evaluable properties normalization* is to evaluate predicates that are independent of a snapshot (such as stateInStatechart) and to simplify logics expressions.

To illustrate this normalization, we consider the property specified in Section 3.2.3. The *statically evaluable properties normalization* now evaluates all static predicates and mappings. In the example, the predicate stateInStatechart (...) is statically evaluated. For instance, Overtaker.Init is not contained in the Real-Time Statechart OvertakeeRoleBehavior. Therefore, 

\[
(stateInStatechart(Overtaker.Init, Overtakee.OvertakeeRoleBehavior) \ \text{implies} \ EF \ stateActive(Overtaker.Init))
\]

is equivalent to

\[
(false \ \text{implies} \ EF \ stateActive(Overtakee.Init))
\]

As its premise is false, this implication is equivalent to true. Therefore, this operand is removed from the overall conjunction. All operands are evaluated like this, so that after the *statically evaluable properties normalization step*, the property is reduced to the following:

**EF stateActive(Overtakee.Init)**

**and EF stateActive(Overtakee.Requested)**

**and EF stateActive(Overtakee.NoAcceleration)**

(with one **EF stateActive(...)** expression for all states contained in Overtakee.OvertakeeRoleBehavior). Notice that this normalization step removes **EF stateActive()** expressions from the property that are not required to evaluate it. This is a potentially large performance benefit because model checking only has to be applied to this smaller number of expressions.

The *statically evaluable properties normalization* requires that universal and existential quantifiers have already been resolved by the quantifier normalization. The post-condition of this normalization is that all static predicates and static mappings (cf. Section 3.1.3 and Section 3.1.4) are removed from the properties.

### 3.2.5 MTCTL Split Properties Transformation

The *split properties transformation* transforms complex MTCTL properties into simple properties. UPPAAL’S TCTL does not allow the specification of several tem-
3 Verification Concepts

poral quantifiers in one property. For example, UPPAAL cannot evaluate an UPPAAL query like \( (A[] \text{not deadlock}) \) or \( (E<> \text{deadlock}) \). However, using an arbitrary number of temporal quantifiers in a single property is valid in MTCTL. Therefore, we split the MTCTL properties in this step, so that afterwards each property contains exactly one temporal quantifier. Later on, after the split properties have been verified, we recompose the property again as part of the backward translation (Section 3.3.1).

Now we consider an example property for the split properties transformation. For the single complex property from Section 3.2.4

\[
\begin{align*}
\text{EF stateActive} \left( \text{Overtakee.Init} \right) \\
\text{and EF stateActive} \left( \text{Overtakee.Requested} \right) \\
\text{and EF stateActive} \left( \text{Overtakee.NoAcceleration} \right)
\end{align*}
\]

we create three simple properties:

1. \(\text{EF stateActive} \left( \text{Overtakee.Init} \right)\)
2. \(\text{EF stateActive} \left( \text{Overtakee.Requested} \right)\)
3. \(\text{EF stateActive} \left( \text{Overtakee.NoAcceleration} \right)\)

The split properties transformation assumes that the properties do not contain any universal and existential quantifiers. Furthermore, for the implementation, temporal quantifiers must not be nested (this is an overall necessary assumption, since it is not supported by UPPAAL). After the split properties transformation, the set of MTCTL properties can be mapped 1:1 to UPPAAL queries.

### 3.2.6 MTCTL Properties in the Transformation Chain

Following the MTCTL transformations, several additional normalization steps are done to simplify the model before migrating it to UPPAAL (Figure 3.2). Each of these transformation steps of the forward translation needs to ensure that properties stay equivalent in the transformed model.

Therefore, we adapted some of the transformations to ensure that properties are correctly transformed during the normalizations. One example for this is the hierarchy normalization. In this transformation, there is the special case of transitions from an exit point to a non-exit-point (i.e. the last transition in a transition chain when leaving a hierarchical state). This kind of transition is mapped to several alternatives (basically one for each possibility to leave a hierarchical state) \[1\]. \(^8\) The semantics of this is that the transition in the original model is firing if any of the alternative transitions in the transformed model is firing. Consequently, transitionFiring predicates referring to such a transition are transformed to a disjunction of transitionFiring predicates for the corresponding alternative transitions in the transformed model.

\(^8\)This will be changed in the future to comply with MECHATRONICUML semantics for leaving hierarchical states depth-first
3.2 Forward Translation

![](image)

**Figure 3.4:** Do event normalized state (compare Figure 2.3 on page 15)

### 3.2.7 Do Event Normalization

This section explains the *do event normalization*. In the overtaking example, there is a *do event* in the state **NoAcceleration** (Figure 2.3 on page 15). Semantically, this means that while **NoAcceleration** is active, the action **checkVel** is executed periodically (with a period of 20ms) [4]. The execution time of the action cannot be defined in **MechatronicUML** (it is assumed to fit inside a period). Furthermore, the starting point of an execution during a period is nondeterministic.

**Uppaal** does not support a do event equivalent syntax feature. Hence, we added the *do event normalization* to the *forward translation* (see Figure 3.2). It substitutes the do event with an embedded RTSC executing the do event behavior.

Figure 3.4 shows the **NoAcceleration** state after the *do event normalization*. The do event is removed and a new embedded and semantically equivalent RTSC is added. This RTSC has three states **Wait**, **Release**, and **Done**. **Wait** is active in every time interval of a period where the action is not yet executed. The time spent in **Release** expresses the execution time of the action. When **Release** is exited the side effects of **checkVel** are applied. For the rest of the period, **Done** is active.

The **NoAcceleration** state can only be exited, if the **Wait** or the **Done** state is active. This matches the semantics of do events, which are not interruptible. Additionally, the time guards, trigger message events, and synchronizations of the originally outgoing transition are copied to the transitions targeting the exit point. This ensures that the adapted state can only be exited whenever this is possible for the original one.

In the case of states that embed other RTSCs (hierarchical states), the behavior of the *do event normalization* is slightly different. Please note that the *do event normalization* is executed after the *composite transitions normalization* (see Figure 3.2). The latter normalization replaces transitions having a hierarchical state as source or target state by a certain construct of entry/exit points and transitions.
Therefore, we can assume that for each originally incoming transition there exists an entry point and for each originally outgoing transitions there exists an exit point of an hierarchical state. To ensure the proper activation of the do event RTSC, a transition from each entry point to Wait is created. Similarly, transitions from Wait and Done to all exit points are created to ensure proper deactivation. In contrast to the case of a flat RTSC, these transitions to the exit points do not contain guards, synchronizations, or trigger message events. As pointed out in [4], they can only fire if one transition from each of the other embedded RTSCs can also fire. Hence, the deactivation of a state is not altered by this do event RTSC construct and additional guards are not needed.

### 3.2.8 Urgency Normalization

MECHATRONICUML distinguishes between urgent and non-urgent transitions [4]. Urgent transitions are fired immediately when they are enabled, whereas non-urgent transitions may postpone the firing even if they are enabled. UPPAAAL does not have urgent edges, but applies the urgency concept to its synchronization channels. The pre-CYBERTRON version of the forward translation was already able to translate urgent and non-urgent transitions to UPPAAAL. Problems arise when urgent and non-urgent transitions synchronize [1].

MECHATRONICUML defines for the synchronization between an urgent and a non-urgent transition non-urgent semantics. This means that the synchronization may be postponed even if both transitions are enabled. However, translating the corresponding MECHATRONICUML synchronization channel into a non-urgent UPPAAAL synchronization channel results in inconsistent semantics when two urgent transitions use the same synchronization channel to synchronize. In this case, the synchronization would be urgent in MECHATRONICUML but non-urgent in UPPAAAL.

The urgency normalization solves this problem. We use Figure 2.5 (page 16) to illustrate this normalization. In that example, there is a transition from Init to Requested that synchronizes over the initiated synchronization channel with a transition from Init to InitiationReceived. Here, the first transition is urgent and the second one is non-urgent.

Figure 3.5 shows an excerpt of the RTSC after the urgency normalization. Two new synchronization channels (initiated_urgent and initiated_nonurgent) are created and initiated is deleted. Additionally, there is a new non-urgent transition from Init to Requested. All actions and guards are copied from the preexisting transition to the new one. However, the synchronization is changed to the initiated_nonurgent synchronization channel.

Moreover, the synchronizations of the other transitions were changed to the synchronization channel corresponding to their respective urgency properties. This results in synchronization channels over which only urgent or non-urgent transitions synchronize respectively, while preserving the semantics of the original
RTSC. These synchronization channels can now safely be declared as urgent/non-urgent on the UPPAAL level.

### 3.2.9 Migration

This subsection introduces the adaptations we have made to the migration. Those range from improvements for the overall semantics preservation to added support of further MECHATRONICUML language features.

**Migration of Component Instance Configurations** The decision to translate Coordination Protocols into component instance configurations (CICs) as part of the transformation chain (see Figure 3.2) raised the need for a migration of CICs to UPPAAL. Therefore, we have adapted the existing migration which now produces an UPPAAL NTA according to a given CIC.

Fundamental changes in the migration concern the creation and instantiation of UPPAAL processes. Instead of deriving UPPAAL processes from roles, we now derive them from component instances and their discrete single and multi port instances. Thus, this approach expresses the semantics of a CIC in terms of UPPAAL’s NTA.

The creation of UPPAAL processes [12] according to a given CIC works as follows: Each component instance provided to the migration is associated to a REAL-TIME STATECHART (RTSC) with exactly one orthogonal composite state
containing a number of flat RTSCs [1]. Hence, for component instances with one single port instance it is sufficient to generate an Uppaal template from each flat RTSC and instantiate it once as an Uppaal process. However, if a component instance contains a multi port instance, the Uppaal template of the corresponding subport behavior RTSC is instantiated multiple times according to the multi port’s multiplicity including RTSCs on which the subport behavior RTSC depends on. Figure 3.6 shows the 1:n protocol situation from Section 3.2.2 with the corresponding RTSCs and Uppaal processes of the resulting Uppaal NTA.

Figure 3.6: Overview of deriving Uppaal processes from a CIC

Each dotted arrow line in the illustration represents an association. For instance, it shows that the overtaker’s component instance is associated to a RTSC with one orthogonal composite state containing the (flattened) component, adaptation and subport behavior RTSCs of the multi port instance. Since the multi port instance has a multiplicity of two, we generate exactly two Uppaal processes from the template associated to this subport behavior RTSC. In contrast, the adaptation behavior RTSC is associated to one corresponding Uppaal process only. Furthermore, each Uppaal process is associated to a component instance by a unique identifier. For example, the Uppaal process for the adaptation behavior gets the identifier of its corresponding component instance (in this case 0) as first argument, and the identifier of it’s corresponding multi port instance as second argument (here 1). Both are internally used in the templates.

9Since the full migration of arbitrary CICs is not a goal of this project group, we assume here the existence of only one single or multi port instance within each component instance to simplify the instantiation of Uppaal processes.
3.2 Forward Translation

Selector Expressions  The adapted migration supports the concept of MECHATRONICUML selector expressions in port behavior [4, p. 70]. For this, we generate additional arrays to UPPAAL’s declarations which map port instances (represented by their unique identifier) to the self, next, previous, first, and last port instance according to the given CIC. Concretely, position i of the generated array contains the ID of the corresponding selector expression’s value for port instance i. Figure 3.7 illustrates this mapping by arrays in case of the 1:n protocol situation as seen before.

\[
\begin{array}{ccc}
\text{Next} & \text{Overtakee2 port instance ID} & \text{Overtakee2 port instance ID} \\
\text{Overtakee1 port instance ID} & \text{(no successor)} & \ldots
\end{array}
\]

Figure 3.7: Illustration of an array in UPPAAL, which maps port instances to successors according to next references of the CIC

In this example, the given CIC has a multi port instance with a multiplicity of two. MECHATRONICUML demands an order with respect to both Overtakee single port instances. This order is given by the illustrated array, which defines the single port instance with the identifier 1 as successor of the one with identifier 0. Since these are the only port instances connected to a multi port instance, all other entries of the array contain null values.

Occasionally, subport and adaptation behaviors make use of selector expressions inside their RTSCs to synchronize each other over synchronization channels [4]. Figure 3.8 illustrates how generated UPPAAL templates from subport and adaptation behaviors make use of the newly introduced arrays.

\[
\text{overtakeOK}[\text{portInstanceID}][\text{next}[\text{portInstanceID}]]!
\]

Figure 3.8: Use of the next array in an UPPAAL template

The illustration above shows that a selector expression is simply converted to an array access by means of the corresponding array (named self, next, previous, first, or last) and a given port instance ID. Hence, our approach makes the use of selector expressions in UPPAAL templates very explicit, since it is visually similar to MECHATRONICUML. This eases the debugging process for developers of MECHATRONICUML in case of bugs in our transformations.
Multiple Message Buffers Per Port  MechatronicUML supports multiple message buffers per port, where each buffer is associated to one message type [4, p. 20]. However, the original verification process did not support this language feature. Therefore, we have extended the migration by several aspects in order to give developers the ability to verify Coordination Protocols with multiple message buffers per role as well.

To this end, we remodeled the UPPAAL NTA declarations to be generated as follows: First, we model the concept of multiple message buffers per port by means of several two-dimensional arrays very similar as we did for supporting selector expressions. Then, we map the combination of a port instance and a message type to the concrete buffer of the given port instance. Figure 3.9 shows an illustration of the overall concept in the context of the Overtaking Protocol assuming an arbitrary assignment of port instances to several buffers.

![Figure 3.9: Multiple message buffers in UPPAAL](image)

In order to deliver a message, UPPAAL needs to know which buffer shall be addressed by it. For this, our implemented internal delivery system of MechatronicUML messages in UPPAAL makes use of the illustrated array structure. For example, the Buffer Assignment array maps the message types finishedOvertaking() and requestOvertaking() to different buffers with respect to both Overtakee port instances expressed by the arrows in the illustrating figure.

Furthermore, we preserve the semantics of individual buffer sizes. For that, we use a similar strategy providing an additional array which maps a combination of a port instance and message type to the size of the associated buffer. This is used by the connector template [1, p. 81] to identify buffer overflows.

MechatronicUML Data Types  MechatronicUML supports a wide variety of different data types like Short, Byte, and Integer [4, p. 11]. Since the initial implementation did not support all types supported in MechatronicUML or simply provided different semantics, we have introduced reasonable UPPAAL equivalents.
3.2 Forward Translation

to preserve the overall semantics of a given Coordination Protocol. For example, the MECHATRONICUML Integer supports values ranging from \(-2,147,483,648\) to \(2,147,483,647\) whereas default built-in UPPAAL Integers (without bounds) range from \(-32,768\) to \(32,768\) [5]. However, MECHATRONICUML’s Long and Double types still are not supported yet, because of UPPAAL limitations\(^{10}\).

Listing 3.1 shows the UPPAAL equivalent of the MECHATRONICUML Integer type.

```
typedef int[−2147483648,2147483647] MUMLInt;
```

Listing 3.1: MECHATRONICUML integer expressed in UPPAAL

The next paragraph introduces our implementation of message parameters in UPPAAL, which makes use of the previously defined MECHATRONICUML data types.

**Message Parameters** MECHATRONICUML supports the passing of message parameters. For example, the developer could decide to add a parameter to the message type `acceptOvertaking()` of our Overtaking Protocol resulting in the parameterized message type `acceptOvertaking(int v)`. In order to make it possible to verify protocols that contain parameterized message types, we have extended the migration to UPPAAL by several elements.

This extension consists of the generation of meaningful structures out of each MECHATRONICUML message type. These structures are used to store arguments for parameters associated with a message, which can be identified by its unique identifier. Figure 3.2 shows for the message type `acceptOvertaking(int v)` the corresponding structure in UPPAAL.

```
typedef struct {
    MessageId mID;
    MUMLInt param_v;
} acceptOvertaking_Parameters;
```

Listing 3.2: Structure for the message type acceptOvertaking(int v)

As seen in the illustration, the generated UPPAAL structure of `acceptOvertaking(int v)` supports the additional parameter by the variable `param_v` and is associated to a message by the identifier `mID`.

Having structures as representation for parameterized message types, we completely revised the handling of messages in UPPAAL. For this, the migration generates several new helper functions. Figure 3.10 illustrates the process of sending a parameterized message using these functions.

\(^{10}\)Floating point data types and operations on them could potentially be manually implemented in UPPAAL. However, implementing this is tedious and models that rely on floating point numbers are often too complex for model checking.
The illustration above presents the sending process as an act between four actors: The RTSC template where the sending is initiated, the ID Management System, which provides functions for assigning unique identifiers to messages, the Argument Management System, which manages the storage of arguments associated to message parameters, and the connector which delivers a message. In turn, the process of receiving a parameterized message works in a similar way.

**Intermediate Locations** Originally, the concept of firing transitions in MECHATRONICUML was missing a meaningful UPPAAL equivalent. Therefore, we introduced intermediate locations (ILs) in generated UPPAAL NTA to preserve the semantics of MECHATRONICUML RTSCs. The decision to generate ILs was made in the context of both the Backward Translation and MTCTL which makes use of ILs for evaluating the transitionFiring predicate (see Section 3.1 and Section 3.3). Moreover it is required by MECHATRONICUML semantics that transitions can fire even if a target state’s invariant is not fulfilled. ILs are a reasonable mechanism to preserve this semantics.

ILs are generated for each MECHATRONICUML transition. Figure 3.11 shows an IL in the context of the generated UPPAAL NTA with respect to our running example. As Figure 3.11 illustrates, an IL is placed between the location which represents the source state Requested and the location which represents the target state Overtaking. In addition, an IL is always committed and the variable intermediateLocationSemaphore is used to ensure that intermediate locations are left immediately and before anything else can happen in the UPPAAL model.
3.3 Backward Translation

The previous section focuses on creating a model for UPPAAL that corresponds to a Real-Time Coordination Protocol. This forward translation enables model checking for MechatronicUML’s Real-Time Coordination Protocols. However, to achieve transparency of the UPPAAL model checker, UPPAAL’s results have to be translated to the level of MechatronicUML (backward translation).

Corresponding to the two supported verification use cases (cf. Figure 3.1 on page 20), there are two UPPAAL artifacts that have to be translated back to MechatronicUML:

1. When verifying (multiple) MTCTL properties, UPPAAL yields several truth values that have to be composed to yield the MTCTL results. This is explained in Section 3.3.1

2. When verifying a single property, UPPAAL may yield a trace (sequence of system states) as a (counter-)example for the property. These system states (simply called states in UPPAAL) need to be translated to corresponding MechatronicUML system states (called snapshots). This is explained in Section 3.3.2

3.3.1 Evaluation of MTCTL Property Results

This section explains how the UPPAAL query results are translated back to MechatronicUML using the composite result evaluation transformation. This is non-trivial because the split properties transformation induces a 1:n relationship between complex MTCTL properties and simple UPPAAL queries (cf. Section 3.2.5). Consequently, (potentially) multiple UPPAAL results have to be composed to yield the result of an original MTCTL property.

The process roughly works as follows: We obtain the model checking results from UPPAAL (its output is parsed by an Xtext parser). These UPPAAL query results correspond (1:1) to the split properties’ results (Section 3.2.5). Next, for each of the split property results, we replace the corresponding expressions in the output of the statically evaluable properties normalization with true or false. After this substitution, the properties are fully statically evaluable. Consequently, the
statically evaluable properties normalization will simplify each property to true or false. These results correspond to the interpretations of the (complex) properties that the user initially specified.

Using the example of Section 3.2.5, the result of the statically evaluable properties normalization is

\[
\text{EF } \text{stateActive}(\text{Overtakee.Init}) \\
\text{and EF } \text{stateActive}(\text{Overtakee.Requested}) \\
\text{and EF } \text{stateActive}(\text{Overtakee.NoAcceleration})
\]

After the other transformations (most notably split properties), UPPAAL yields the results of \( \text{EF } \text{stateActive}(\text{Overtakee.Init}) \), \( \text{EF } \text{stateActive}(\text{Overtakee.Requested}) \), and \( \text{EF } \text{stateActive}(\text{Overtakee.NoAcceleration}) \), respectively. The composite result evaluation transformation substitutes these expressions in the original model above with UPPAAL’s results, yielding, for example:

\[
\text{true} \\
\text{and true} \\
\text{and true}
\]

Running the statically evaluable properties normalization again yields true, which is the result of the initial complex MTCTL property

\[
\text{forall } (s:\text{States}) \ (\text{stateInStatechart}(s,\text{Overtakee.OvertakeeRoleBehavior}) \text{ implies EF } \\
\text{stateActive}(s))
\]

### 3.3.2 Uppaal to MechatronicUML Trace Translation

In the case of not fulfilled safety properties, there is an trace leading from an initial state to a state that is actually violating the property. Similarly, for a fulfilled reachability property, there is an trace fulfilling the property. Especially in the case of not fulfilled safety properties, the information contained in the trace can considerably ease the task of debugging the model and finding/fixing the cause of the violation. We support traces for split MTCTL properties (Section 3.2.5), because these are actually translated to UPPAAL and verified. Hence, UPPAAL can only return a trace for these properties.

Section 3.2.2 introduces the protocol to CIC transformation as the first transformation of the forward translation. Additionally, it explains that actually the verification is performed on the component instance configuration (CIC) resulting from this first transformation. Therefore, we had to decide whether to translate the UPPAAL trace into a trace of the original RTCP or into a trace of the created CIC.

We decided for a backward translation to the CIC. Mainly, because in this way the implementation of the backward translation can also be used when the forward translation is adapted to the verification of CICs. Additionally, since the behavior of the roles is the same as the one of the ports created by the protocol to CIC transformation, all the information to debug the original RTCP is still available.
3.3 Backward Translation

Furthermore, we decided to translate one UPPAAL state into one MECHATRONICUML snapshot. Thereby, certain action that on the MECHATRONICUML level can be assumed to be performed within one snapshot-change are stretched across several MECHATRONICUML snapshots. This is done mainly because there does not yet exist a formal MECHATRONICUML snapshot semantics definition.

Implementation-wise, the UPPAAL trace is translated into an instance of a CIC-specific runtime meta-model. To do so, the textual UPPAAL trace is parsed into an UPPAAL trace model by utilizing XText\(^\text{11}\). This model is then translated into an CIC-specific runtime model while making heavy use of the QVTo\(^\text{12}\) trace model. For more information about these concepts, please see [1]. Finally, we visualize this CIC-specific model by utilizing Graphviz\(^\text{13}\).

In the preexisting version of the backward translation, only the active locations of an UPPAAL state were translated into active MECHATRONICUML states of an RTSC instance [1]. We enhanced the backward translation by the following other details about the current state of the system, in the course of CYBERTRON:

- Variable values
- Clock values
- Buffers and contained messages
- Connectors and traversing messages
- Message parameters
- Additional structural information, e.g., component instances, single-port instances, and the corresponding containment relation

The following simple example shows how a MECHATRONICUML snapshot is structured and how a MECHATRONICUML trace can be utilized when debugging. Please assume to remove the raise message event of the transition from Init to Requested in Figure 2.4 and checking the property \text{AG not deadlock} on the Overtaking RTCP. The verification will return that the property is not fulfilled. In the following, we use the created MECHATRONICUML trace to find out why this is the case. Figure 3.12, shows the last snapshot of this trace. We chose the style of hierarchical boxes to visualize the information of a snapshot, because it matches the hierarchical models of MECHATRONICUML.

The outer box represents the runtime version of the CIC to verify (OvertakingCIC). It contains boxes representing runtime versions of the contained component instances and the connectors. In our case we have two component instances (overtaker and overtakee) and one connector between their ports that does not contain a message at the moment.

In the first embedded box of the overtaker box, you see overtaker’s component RTSC instance. The corresponding RTSC was generated by the protocol to

\(^{11}\)http://www.eclipse.org/Xtext/
\(^{12}\)http://projects.eclipse.org/projects/modeling.mmt.qvt-oml
\(^{13}\)http://www.graphviz.org
3 Verification Concepts

Figure 3.12: Labeled MECHATRONICUML snapshot

CIC transformation and consists of only one state (Initial) containing the RTSC of overtake’s port. As already mentioned in Section 3.2.2, the behavior of this port is the same as the behavior of the role it refines.

In the second embedded box of the overtaker box you can see the runtime version of the overtakerPort. First, there is a box expressing the port’s RTSC instance (OvertakerRoleBehavior) which is in the Requested state. Additionally, the value of the velocity variable is 0. The box below shows the port’s buffer (overtakerBuffer) that does not contain any message at the moment. The box of the overtaker runtime component instance is structured analogously.

Now, the OvertakerRoleBehavior is in the Requested state and OvertakeeRoleBehavior is in the Init state. OvertakeeRoleBehavior waits for the requestOvertaking message, which is never sent by OvertakerRoleBehavior. This results in a deadlock and thereby the AG not deadlock property is violated.
4 Deployment Concepts

In this chapter we explain the developed deployment concepts. The deployment covers all steps that are necessary to place and finally execute a software system on a target platform.

In the context of our project group the software system is specified by the MechatronicUML PIM, which describes the architecture and behavior of the software system. The target platform refers to the platform which executes the software system. In our context, the target platform is a cyber-physical system including its hardware resources, sensors, actuators, and operating system.

The target platform is represented in MechatronicUML by the platform-description model (PDM). The PDM covers the description of the hardware of a cyber-physical system in terms of composed hardware resources, as described in the MechatronicUML PSM Techreport [3]. For the purpose of simplicity, we refer to a set of hardware resources, which are able to execute code, as electronic control units (ECUs). Since the PDM did not cover the specification of the software platform, we extended the PDM by an Operating System Language, which allows the description of the operating system and its API. For details about the Operating System Language please refer to Section 4.2.

In the following, we present our process for the deployment of MechatronicUML PIMs to a target platform. Afterwards, we present each process step in detail in a separate section. In Section 4.3, we present the logical placement of the PIM’s software component instances to ECUs (called allocation). The extension of the PIM with information for a specific target platform is explained in Section 4.4. The refinement of the PIM into a platform-specific model (PSM), is introduced in Section 4.4.1. Finally, we explain in Section 4.6.1 the generation of source code based on the PSM.

4.1 Deployment Process

The deployment process extends the MechatronicUML method [4] by the refinement of a platform-independent model for a specific target platform. Therefore, it provides a structured method to get from a MechatronicUML PIM to a deployment of the system. Figure 4.1 provides an overview of the deployment process, which we illustrate in the following.

The starting point of the deployment process are the artifacts produced in the former steps: First, the PIM, which specifies the software system. For instance, the Overtaking software system described in Section 2.2. Second, the PDM, which
In **Step 1**, an allocation of software component instances to ECUs of the target platform is specified. An allocation has to respect certain constraints, for example the limited amount of available memory for each ECU. Moreover, it is often required that an allocation is optimal with regard to some optimization goal. Thus, a manual allocation is usually not feasible within a reasonable amount of time. Therefore, we provide an Allocation Specification Language for MECHATRONICUML, which allows the deployment engineer to specify high level allocation constraints and optimization goals. For instance, the allocated component instances should not consume more than a certain amount of memory, or an allocation shall have minimum communication latency. After that, an allocation is computed automatically w.r.t. these constraints and optimization goals. The Allocation Specification Language and the computation of an allocation is described in Section 4.3.

After an allocation has been computed, it is fixed which component instances shall be deployed to which ECU. In order to integrate a component instance for a ECU, it has to specified how the component instance uses the hardware resources and API-Calls of this ECU, e.g., how to read a value from a sensor. The ECUs, their hardware resources and their provided API-Calls are represented by our PDM. Thus, a mapping between the resources of the PDM and the PIM has to be specified in **Step 2**. This mapping is called Platform-Mapping, as described in the MDA Guide [13]. For a description of the Platform-Mapping, please refer to Section 4.4. Based on the Platform-Mapping, the PIM is refined to a PSM, which is used as an input for the code generation. For a description of the PSM, please refer to Section 4.5.

In **Step 3**, based on the PSM, a Codegen model is automatically created. The Codegen model is used as the input for the code generation. The code generation is described in detail in Section 4.6.1.

In **Step 4**, the PSM and the generated source code can be analyzed. Based on
a Worst-Case-Execution-Time Analysis, a Schedulability Analysis for the target platform takes place. After the analyses have been done, the results are used to detect and fix possible errors in the PIM, the allocation, or the Platform-Mapping. Finally, the produced source code can be compiled, uploaded, executed, and tested on the target platform.

4.2 Extension of the Platform-Description: Operating System

This section introduces our extension of the PDM to describe the operating system of the target platform. One of the key points of a cyber-physical system is the interaction of the system with its environment. This interaction is done via different sensors and actuators, which are controlled by the system. To control different sensors and actuators, the target platform’s operating system usually provides an API. For instance, nxtOSEK\(^1\) provides the API-Call INT ecrobot_get_light_sensor (SHORT port_id) to read the light value from a connected LEGO Mindstorms light sensor. To specify the API of an operating system, we created the Operating System Language, which we introduce in the following.

4.2.1 Goal of the Operating System Language

The previous PDM [3] was restricted to the description of the hardware platform. Thus, there existed no possibility to specify which API is provided by the operating system of a target platform. However, in the PIM, the sensors and actuators are modeled by continuous components, which are considered as blackboxes. Therefore, it is not clear which code has to be triggered in the generated code to deal with the sensors and actuators, which are represented as continuous components.

Thus, the developer had to manually adapt the generated code and specify the API-Calls for each continuous component instance. Moreover, this manual adaption had to be repeated for each change of the PIM or PDM. Since the manual adaption for each change of the PIM or PDM is error prone for a large amount of different sensors and actuators, we developed a textual based language to describe the API, based on Rose’s bachelor thesis [14].

4.2.2 Language Elements & Semantics

The Operating System Language allows the specification of the operating system’s API. The EBNF Grammar of the Operating System Language is shown in Listing 4.1.

\(^1\)http://lejos-osek.sourceforge.net/
4 Deployment Concepts

OperatingSystem = 'OperatingSystem:' ID '{' APIRepository* '}' ';'

APIRepository = 'Device_API_Calls:' ID '{' APICommand* '}' ';'

APICommand = MUML::DataType ID '(' (Parameter || ',' ')')
             ['[TimeConstraint]'] ';'

Parameter = MUML::DataType ID

TimeConstraint = Number JAVA::TIMEUNIT

Listing 4.1: EBNF Grammar: Operating System Language

An OperatingSystem consists of several APIRepositories. An APIRepository describes the API-Calls (APICommands) for exactly one sensor or actuator, which are provided by the operating system. An APICommand specifies its signature, which is in particular its return value and its parameters.

Furthermore, an APICommand can optionally be annotated with a TimeConstraint. A TimeConstraint specifies the minimum time interval between the repeated execution of the API-Call. This annotation is necessary, because some sensors report corrupt data, and some actuators behave undefinedly if they are triggered without a certain delay. These time constraints have to be specified and respected in the generated code to ensure correct behavior of the target platform.

Listing 4.2 shows the API-Calls for the light sensor of our overtaking robots (see Section 2.3) of the operating system nxtOSEK².

OperatingSystem:nxtOSEK{

Device_API_Calls: LightSensor {
    VOID ecrobot_set_light_sensor_active (SHORT port_id);
    INT ecrobot_get_light_sensor (SHORT port_id)[10ms];
}

}

Listing 4.2: Example: Operating system nxtOSEK

The operating system nxtOSEK provides two API-Calls for triggering the light sensor. The first API-Call ecrobot_set_light_sensor_active activates the light sensor and enables power to the specified hardware port. The API-Call ecrobot_get_light_sensor returns the current light value as an integer. Furthermore, the light sensor can only be read with an interval of 10 ms, otherwise corrupt light values are returned.

²http://lejos-osek.sourceforge.net/ecrobot_c_api_frame.htm
4.3 Allocation

This section deals with the Allocate Software Components step, which is the first step of the deployment process. In order to execute the modeled and verified software, we have to specify which component instance is executed on which ECU. Such a mapping is called an allocation.

In general, an allocation has to satisfy various constraints. For instance, the component instances that are allocated to an ECU should not consume more main memory in total than the ECU provides. Apart from such general constraints, there are also more application-specific constraints. For instance, for our running example, we require that the structured component instances overtake and overtaker and all its embedded component instances are allocated to the Overtaker and Overtakee platform instance, respectively.

Often, an allocation should be optimal with regard to some measure function. For example, we are looking for an allocation that minimizes the communication latency between certain or all interacting components. This is useful, for instance, for multimedia components, because a high latency might decrease the sound or video quality from a user’s perspective.

With a growing number of component instances and ECUs, it is impracticable to manually come up with an allocation that satisfies all constraints and is optimal with regard to some measure function. Thus, we want to automatically compute an allocation. To achieve this, it is necessary to express the constraints and measure function in a formal way. For this, we developed the Allocation Specification Language. In order to automatically compute an allocation, we transform the formal constraint and measure function specification into a corresponding 0-1 Integer Linear Program (0-1 ILP). A solution for this 0-1 ILP corresponds to an allocation that satisfies all constraints and is optimal with regard to the measure function.

The remainder of this section is structured as follows. Section 4.3.1 describes the goal of the Allocation Specification Language and its design ideas. Then, we describe the language’s syntax and its informal semantics in Section 4.3.2. In Section 4.3.3, we describe the formal preliminaries that are needed for the following sections. In order to proof the correctness of the previously mentioned transformation, we specify the Allocation Specification Language’s formal semantics in Section 4.3.4. Based on these formal semantics, we describe the transformation to a 0-1 ILP and prove that this transformation is semantics-preserving in Section 4.3.5.

4.3.1 Goal of the Allocation Specification Language and its Design Idea

In this section, we describe the goals and the design idea of the Allocation Specification Language. The primary design goal of the Allocation Specification Language is to provide a means to the user for specifying allocation constraints and
4 Deployment Concepts

a measure function in an easy and expressive way.

In order to achieve this, the language embeds the Object Constraint Language (OCL) [15], which is widely used in the model-driven world. Thus, we assume that the potential users of the Allocation Specification Language are already familiar with OCL.

As already mentioned, the language should support the specification of constraints. More precisely, it should support the four different kinds of constraints that were identified in [16]. These four kinds of constraints and their informal description are summarized in Table 4.1. The general idea is that for each of

<table>
<thead>
<tr>
<th>Constraint kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sameLocation</td>
<td>Used to allocate two component instances to the same ECU.</td>
</tr>
<tr>
<td>differentLocation</td>
<td>Used to allocate two component instances to different ECUs.</td>
</tr>
<tr>
<td>requiredHardwareResourceInstance</td>
<td>Used to specify that a component instance can only be allocated to certain ECUs. Additionally, it is used to specify that a component instance can only be allocated to certain ECUs if and only if other component instances are allocated to certain ECUs as well.</td>
</tr>
<tr>
<td>resource</td>
<td>Used to express certain resource restrictions like the amount of available main memory, which are imposed by the ECUs.</td>
</tr>
</tbody>
</table>

Table 4.1: The four constraint kinds and their informal description

these constraints an OCL expression is specified by the user, which evaluates to a specific OCL type. The concrete OCL type is dependent on the constraint’s kind. For instance, to implement a constraint that ensures, for safety reasons, that the overtaker_driver and overtaker_communicator component instances are allocated to the same ECU, we would specify a constraint of the kind sameLocation. This constraint’s OCL expression evaluates to a set that consists of a 2-tuple. The elements of the 2-tuple refer to the overtaker_driver and overtaker_communicator component instances.

Another design goal of the Allocation Specification Language is modularity and reuse, in order to avoid, for instance, the duplication of frequently used OCL expressions. For this, the language provides a library concept. A library encapsulates operations that are frequently used when specifying constraints or a measure function. Instead of writing these operations from scratch repeatedly, it is possible to include a library and have access to all of its operations.
4.3 Allocation Specification Language Concrete Syntax

In this section, we describe the concrete syntax of the Allocation Specification Language and its informal semantics with the help of examples. We mainly focus on the language’s general structure instead of explaining every production rule in detail. The Allocation Specification Language’s Xtext grammar can be found in Appendix D.

The building blocks of an allocation specification (grammar rule Specification) are arbitrary many Constraint, Service, and an optional MeasureFunction. In order to build a modular language, there are technical language elements like ImportCS, IncludeCS, and ClassifierContextDeclCS.

**Constraint** As already mentioned in Section 4.3.1, the language supports four different kinds of constraints. Each Constraint consists of exactly one Model, which represents the OCL expression. In Section 4.3.4, we see that a constraint’s OCL expression evaluates to a certain OCL type, which is usually a set of tuples. Depending on the constraint’s kind, the tuples have a certain structure. To illustrate this, consider the following informal allocation constraint: Ensure that the structured component instances overtaker and overtakee and all its embedded component instances are allocated to the Overtaker and Overtakee platform instance, respectively. Using the Allocation Specification Language, this is expressed formally in Listing 4.3.

```plaintext
constraint requiredHardwareResourceInstance ComponentsToPlatform {
  descriptors (first, second);
  ocl Set{"overtaker", "overtakee"} -> collect(componentName : String |
    self.allocateEmbeddedToPlatformInstance(componentName, componentName.toUpperCaseFirst()))
} -> asSet();
```

Listing 4.3: A constraint of the kind requiredHardwareResourceInstance

Listing 4.3 shows a constraint, which is called ComponentsToPlatform, of the kind requiredHardwareResourceInstance (for details, refer to the RequiredHardwareResourceInstance grammar rule). A constraint of the kind requiredHardwareResourceInstance is used to express that a component instance can only be allocated to certain ECUs. Structurally, this constraint consists of so-called tuple descriptors (grammar rule ComponentResourceTupleDescriptor) and an OCL expression (grammar rule Model). In this example, the expected OCL type of the OCL expression is Set(Tuple(first : ComponentInstance, second : StructuredResourceInstance)). That is a set that consists of tuples that have two named parts [15, p. 27]. The named part first has the type ComponentInstance and the named part second has the type StructuredResourceInstance. To explain its semantics, consider the set Set(Tuple(first = overtaker_driver, second = overtaker_b1), Tuple(first =
overtaker\_driver, second = overtaker\_b2}\), which is a subset of the evaluation result of the OCL expression in Listing 4.3. This set consists of two tuples. The named part first of both tuples refers to the component instance overtaker\_driver. The semantics is that the overtaker\_driver component instance has to be allocated to the ECU overtaker\_b1 or to the ECU overtaker\_b2.

Now going back to the example, we have to provide an OCL expression that evaluates to such a set. To achieve this in an easy way, the Allocation Specification Language provides the allocate\_Embedded\_To\_Platform\_Instance operation, which is part of a set of standard operations. This operation takes the name of a structured component instance, the name of a platform instance and evaluates to a set of tuples. The named part first of these tuples refers to the passed structured component instance itself or to one of its embedded component instances. The named part second refers to an ECU, which is part of the passed platform instance.

Recall from Section 2.3 that the name of the platform instance can be obtained from the corresponding structured component instance by capitalizing the first letter of its name. So, what the OCL expression in Listing 4.3 basically does is calling self.allocate\_Embedded\_To\_Platform\_Instance(overtaker, Overtaker), self.allocate\_Embedded\_To\_Platform\_Instance(overtakee, Overtakee) and joining the resulting sets.

As mentioned earlier, a required\_Hardware\_Resource\_Instance constraint consists of several tuple descriptors. The example constraint in Listing 4.3 consists of exactly one tuple descriptor descriptors (first, second). It is used to describe the named parts of the tuples that are returned by the constraint’s OCL expression. This is necessary if we want to express more complex constraints like: (allocate overtaker\_driver and overtaker\_communicator to overtaker\_b1) or (allocate overtaker\_driver to overtaker\_b1 and allocate overtaker\_communicator to overtaker\_b2). For this, we have to write an OCL expression that returns a set like

\[
\text{Set}\{\text{Tuple}\{c1 = overtaker\_driver, e1 = overtaker\_b1, } \\
\text{c2 = overtaker\_communicator, e2 = overtaker\_b1}\}, \\
\text{Tuple}\{c1 = overtaker\_driver, e1 = overtaker\_b1, } \\
\text{c2 = overtaker\_communicator, e2 = overtaker\_b2}\}\}
\]

Moreover, we have to specify the tuple descriptors descriptors (c1, e1), (c2, e2) to group the corresponding component instance, ECU pairs.

It is important to note that if we change the tuple descriptors to descriptors (c1, e2), (c2, e1), the set has a different semantics: (allocate overtaker\_driver and overtaker\_communicator to overtaker\_b1) or (allocate overtaker\_driver to overtaker\_b2 and allocate overtaker\_communicator to overtaker\_b1).

**MeasureFunction** The optional MeasureFunction is used to specify the optimization goal. It refers of at least one Service. A service can be understood as a
certain entity of the software system to which several component instances belong. For instance, the communication between (certain or all) component instances can be viewed as service. Each Service consists of arbitrary many QoSDimension elements, which rate service specific quality goals. QoS dimensions for a communication service are, for example, reliability and latency.

In the following, we assume that the communication latency between the embedded component instances of the structured component instance overtaker should be minimized. This is formally expressed, using the Allocation Specification Language, in Listing 4.4.

```java
service overtakerCommunication {
    qos latency {
        value lat;
        descriptors (c1, e1), (c2, e2):
        ocl self.minimizeStructuredComponentLatency("overtaker");
    }
}

min measure overtakerCommunication;
```

Listing 4.4 defines a service called overtakerCommunication. This service consists of exactly one QoS dimension, which is called latency. In this example, the QoS dimension’s OCL expression is supposed to return a set of the type Set(Tuple(lat : Real, c1 : ComponentInstance, e1 : StructuredResourceInstance, c2 : ComponentInstance, e2 : StructuredResourceInstance)). To explain its semantics consider the set

\[
\text{Set}\{\text{Tuple}\{\text{lat} = 0, c1 = overtaker\_driver, e1 = overtaker\_b1, c2 = overtaker\_communicator, e2 = overtaker\_b1\}\}, \\
\text{Tuple}\{\text{lat} = 2, c1 = overtaker\_driver, e1 = overtaker\_b1, c2 = overtaker\_communicator, e2 = overtaker\_b2\}\}
\]

The semantics is that there is no communication latency (\(\text{lat} = 0\)), if the overtaker\_driver and overtaker\_communicator are allocated to overtaker\_b1. If the overtaker\_driver is allocated to overtaker\_b1 and overtaker\_communicator is allocated to overtaker\_b2, there is a communication latency of 2ms (\(\text{lat} = 2\)).

Again, the names of the tuple’s named parts are described using the tuple descriptors value lat and descriptors (c1, e1), (c2, e2). Also, we assume the existence of an minimizeStructuredComponentLatency operation, which takes the name of a structured component instance and returns the desired set. Currently, such an operation is not part of the Allocation Specification Language’s set of standard operations.
4 Deployment Concepts

**ImportCS** An ImportCS is used to import a meta-model. For instance, the statement `import 'http://example.com/extension'` imports the meta-model that is registered via the specified uri. Afterwards, it is possible to specify an OCL expression that refers to classes that are provided by this meta-model.

**ClassifierContextDeclCS** A ClassifierContextDeclCS is used to couple an operation to a classifier [15, p. 194]. Listing 4.5 shows how to extend OCL’s String class with a custom `toUpperCaseFirst` operation. To improve the readability, the implementation of this operation omits boundary checks, which are needed if the string is empty or has size 1.

```ocl
context ocl::String
def: toUpperCaseFirst() : String =
self.at(1).toUpperCase() + self.substring(2, self.size())
```

Listing 4.5: Extend OCL’s String class with a toUpperCaseFirst operation

**IncludeCS** An IncludeCS is used to include an external library. For instance, a statement `include 'platform:/plugin/de.uni_paderborn.fujaba.muml.allocation.language.xtext/operations/OCLContext.ocl'` includes the library that is located at the specified uri. The library `OCLContext.ocl` is a standard library that provides various operations, which should ease the constraint specification. Table 4.2 shows an excerpt of the available standard library operations. Technically, a library is represented by a Complete OCL Document ³.

### 4.3.3 Formal Preliminaries

In this section, we introduce the formal preliminaries that are needed to define the semantics of the Allocation Specification Language and its transformation to a 0-1 ILP.

We assume that all OCL expressions are specified in the syntactical context of the class `OCLContext`. As depicted in Figure 4.2, the class `OCLContext` has references to the classes `ComponentInstanceConfiguration` and `HWPlatformInstanceConfiguration`. Hence, it is possible to specify OCL expressions that refer to MECHATRONICUML language elements like component instances, structured resource instances etc. Let M be the object model [15, p. 209] that is represented by the UML class diagram in Figure 4.2. Moreover, let \( \sigma = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}}) \) be a system state [15, p. 210f.] for the object model M. Basically, such a system state corresponds to a UML object diagram that is typed over the UML class diagram in Figure 4.2. Let OCL be the set of all syntactically correct (in the context of the class `OCLContext`) OCL expressions. To evaluate an OCL expression \( \Psi \in OCL \), an evaluation environment is needed [15, p.

³CompleteOCL.xtext in the org.eclipse.ocl.examples.xtext.completeocl plugin
4.3 Allocation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Parameters</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocateToSameECUs</td>
<td>instance1 : String, instance2 : String</td>
<td>Set(Tuple(first : ComponentInstance, second : ComponentInstance))</td>
</tr>
<tr>
<td>Allocate instance1 and instance2 to the same ECU.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>allocateToDifferentECUs</td>
<td>instance1 : String, instance2 : String</td>
<td>Set(Tuple(first : ComponentInstance, second : ComponentInstance))</td>
</tr>
<tr>
<td>Allocate instance1 and instance2 to different ECUs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>allocateEmbeddedToPlatformInstance</td>
<td>swInstanceName : String, platformInstanceName : String</td>
<td>Set(Tuple(first : ComponentInstance, second : ResourceInstance))</td>
</tr>
<tr>
<td>If swInstanceName refers to an AtomicComponentInstance, allocate it to one of the ECUs of the platform instance whose name is platformInstanceName. If swInstanceName refers to a StructuredComponentInstance, allocate it and all its embedded component instances to one of the ECUs of the platform instance whose name is platformInstanceName.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>allocateToECU</td>
<td>instanceName : String, ecuName : String</td>
<td>Set(Tuple(first : ComponentInstance, second : ResourceInstance))</td>
</tr>
<tr>
<td>Allocate the instanceName component instance to the ecuName ECU.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAllSWInstances</td>
<td>None</td>
<td>Set(ComponentInstance)</td>
</tr>
<tr>
<td>Returns a set that consists of all component instances.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getSWInstance</td>
<td>instanceName : String</td>
<td>ComponentInstance</td>
</tr>
<tr>
<td>Returns a component instance whose name is instanceName.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Excerpt of the available standard library operations

230]. For reasons of simplification, we assume that $\text{eval}_\sigma(\Psi)$ denotes the evaluation result of the OCL expression $\Psi \in OCL$ and that the evaluation environment’s system state is $\sigma$. We usually omit the $\sigma$ and simply write $\text{eval}(\Psi)$.

Let $COMP = \sigma_{\text{CLASS}}(\text{ComponentInstance})$ be a set of MechatronicUML component instances. Analogously, let $ECU = \sigma_{\text{CLASS}}(\text{StructuredResourceInstance})$ be a set of MechatronicUML ECUs. A mapping $f \in ECU^{COMP} := \{f : COMP \rightarrow ECU\}$ is called an allocation. Let $c \in COMP$ and $e \in ECU$. Then, $f(c) = e$ means that the component instance $c$ is allocated to the ECU $e$.

In OCL, a tuple consists of several named parts, which are used to access its elements [15, p. 27]. Listing 4.6 demonstrates how to create and access a tuple.
4 Deployment Concepts

Figure 4.2: The class OCLContext is the syntactical context for OCL expressions

Listing 4.6: Creating and accessing a tuple in OCL

```ocaml
define t : Tuple(comp : ComponentInstance, ecu : StructuredResourceInstance)
define t = Tuple(ecu = e, comp = c) in t.comp
```

The first line defines a variable of the type Tuple(comp : ComponentInstance, ecu : StructuredResourceInstance). That is, the variable t is a tuple with the named parts comp and ecu. The third line constructs a new tuple object and binds the object e to the named part ecu and the object c to the named part comp (assumption: e has the type StructuredResourceInstance, c has the type ComponentInstance and both variables are available in the scope of the OCL expression). Unlike a mathematical tuple, an OCL tuple has no order. That is, the ordering of the named parts is unimportant [15, p. 27]. The fifth line is used to access the element that was bound to the named part comp. Thus, the whole OCL expression in Listing 4.6 evaluates to the object c.

We interpret an OCL tuple as a mathematical tuple by using the order of the specified tuple descriptors. For convenience, we omit tuple descriptors in the following sections and assume the implicit order of the named parts from the tuple’s type definition. Hence, the OCL tuple t in Listing 4.6 corresponds to the mathematical tuple \((c, e)\). More generally, an OCL tuple of the type Tuple\((n_1 : T_1, ..., n_k : T_k)\), where \(T_i\) is a type, is interpreted as the mathematical tuple \((n_1, ..., n_k)\).

Furthermore, an OCL set s of the type Set\((T)\), where \(T\) is a type, is considered as mathematical set. For instance, to select an arbitrary element e from the set s the notation \(e \in s\) is used.
4.3.4 Allocation Specification Language Semantics

In this section, we formally describe the semantics of the four constraints and the measure function.

First, we describe the semantics of the four constraints with the help of sets. For each constraint, we define a set that consists of feasible allocations. Formally, a constraint $\Phi$ is represented by 3-tuple $(\Phi_{kind}, \Phi_{OCL_T}, \Phi_{OCL_E})$. The first component $\Phi_{kind} \in \{\text{sameLocation, differentLocation, requiredHardwareResourceInstance, resource}\}$ represents the kind of the constraint. The second component $\Phi_{OCL_T}$ represents an OCL type. The third component $\Phi_{OCL_E} \in OCL$ represents an OCL expression, whose OCL type conforms to $\Phi_{OCL_T}$.

The sameLocation constraint

The sameLocation constraint is used to specify that two component instances have to be allocated to the same ECU. A sameLocation constraint $\Phi$ is defined as follows:

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

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

$$f \in F_\Phi \iff \forall (c_1, c_2) \in \text{eval}(\Phi_{OCL_E}) : f(c_1) = f(c_2)$$

The differentLocation constraint

The differentLocation constraint is used to specify that two component instances have to be allocated to different ECUs. A differentLocation constraint $\Phi$ is defined as follows:

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

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

$$f \in F_\Phi \iff \forall (c_1, c_2) \in \text{eval}(\Phi_{OCL_E}) : f(c_1) \neq f(c_2)$$

The requiredHardwareResourceInstance constraint

The requiredHardwareResourceInstance constraint is used to specify that a component instance can only be allocated to certain ECUs. Additionally, it is also possible to express more complex dependencies like allocate $c_1$ to $e_1$ and $c_2$ to $e_2$ or allocate $c_1$ to $e_2$ and $c_2$ to $e_3$ etc. A requiredHardwareResourceInstance constraint $\Phi$ is defined as follows:

- $\Phi_{kind} = \text{requiredHardwareResourceInstance}$
- $\Phi_{OCL_T} = \text{Set}(\text{Tuple}(c_1 : \text{ComponentInstance}, e_1 : \text{StructuredResourceInstance}, ..., c_n : \text{ComponentInstance}, e_n : \text{StructuredResourceInstance}))$
4 Deployment Concepts

- \( \Phi_{OCL E} \in OCL \)

Let \( \sim \) be a binary relation on the set \( \operatorname{eval}(\Phi_{OCL E}) \) such that \((c_1, e_1, c_2, e_2, ..., c_n, e_n) \sim (c'_1, e'_1, c'_2, e'_2, ..., c'_n, e'_n) \iff \{c_1, e_1, c_2, e_2, ..., c_n, e_n\} = \{c'_1, e'_1, c'_2, e'_2, ..., c'_n, e'_n\} \). Obviously, \( \sim \) is an equivalence relation. The set \( Q := \operatorname{eval}(\Phi_{OCL E}) / \sim \) denotes the quotient set, that is the set of equivalence classes. The set of feasible allocations \( F_\Phi \subseteq ECU^{COMP} \) is defined as follows:

\[
\exists f \in F_\Phi \iff \forall x \in Q : \bigvee_{(c_1, e_1, c_2, e_2, ..., c_n, e_n) \in x} f(c_1) = e_1 \land f(c_2) = e_2 \land ... \land f(c_n) = e_n
\]

**The resource constraint** The resource constraint is used to express certain resource restrictions like the amount of available main memory, which are imposed by the ECUs. A resource constraint \( \Phi \) is defined as follows:

- \( \Phi_{kind} = \text{resource} \)
- \( \Phi_{OCL T} = \text{Set}(\text{Tuple}(w : \text{Set}(\text{Tuple}(c_1 : \text{ComponentInstance}, e_1 : \text{StructuredResourceInstance}, ..., c_n : \text{ComponentInstance}, e_n : \text{StructuredResourceInstance}, w : \text{Real})), r : \text{Real})) \)
- \( \Phi_{OCL E} \in OCL \)

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

\[
f \in F_\Phi \iff \forall (w, r) \in \operatorname{eval}(\Phi_{OCL E}) : \sum_{(c_1, e_1, ..., c_n, e_n, w') \in w} w' x(f, c_1, e_1, ..., c_n, e_n) \leq r
\]

, where

\[
x(f, c_1, e_1, ..., c_n, e_n) := \begin{cases} 1 & \text{if } f(c_1) = e_1 \land ... \land f(c_n) = e_n, \\ 0 & \text{else} \end{cases}
\]

**Multiple constraints** So far, we defined for each constraint a set of feasible allocations. Since an allocation has to usually respect multiple constraints, we have to define the set of feasible allocations for a set of constraints. Let \( C \) be a set of constraints. The set of feasible allocations \( F_C \subseteq ECU^{COMP} \) is defined as follows:

\[
F_C := \bigcap_{\Phi \in C} F_\Phi
\]

**The measure function** The measure function is used to specify an optimization goal. Basically, the measure function is the sum of several services. As shown in Section 4.3.2, a service consists of multiple QoS dimensions.

Formally, a QoS dimension \( q \) is represented by a 2-tuple \((q_{OCL T}, q_{OCL E})\):
4.3 Allocation

- \( q_{\text{OCLT}} = \text{Set}(\text{Tuple}(\ c_1 : \text{ComponentInstance}, e_2 : \text{StructuredResourceInstance}, ..., \ c_n : \text{ComponentInstance}, e_n : \text{StructuredResourceInstance}, w : \text{Real}))) \)

- \( q_{\text{OCL}_E} \in \text{OCL} \)

Let \( f \) be an allocation. Next, we define

\[
\text{sum}(f, q) := \sum_{(c_1, e_1, ..., c_n, e_n, w) \in \text{eval}(q_{\text{OCL}_E})} wx(f, c_1, e_1, ..., c_n, e_n)
\]

, where \( x \) is defined as in Equation (4.2).

A service \( S \) is defined as a set of QoS dimensions. Let \( \text{SERVICES} \) be a set of services. The measure function \( m \) is defined as follows:

\[
m(f) := \sum_{S \in \text{SERVICES}} \sum_{q \in S} \text{sum}(f, q)
\]

**Allocation specification** An allocation specification is used to describe all constraints, the measure function, and the optimization goal. Formally, an allocation specification \( \alpha \) is a 3-tuple \((C_\alpha, m_\alpha, \theta_\alpha)\), where

- \( C_\alpha \) is a set of constraints
- \( m_\alpha \) is a measure function
- \( \theta_\alpha \in \{\leq, \geq\} \) is a comparison operator and represents the optimization goal.
  
  That is if \( \theta_\alpha = \leq \), then \( m_\alpha \) is minimized, else \( m_\alpha \) is maximized.

**Feasible and optimal allocation** Finally, we define a set that consists of feasible and optimal, with regard to an allocation specification, allocations. Let \( \alpha \) be an allocation specification. The set

\[
\text{OPT}_\alpha := \{f \in F_{C_\alpha} : \forall f' \in F_{C_\alpha} : m_\alpha(f) \theta_\alpha m_\alpha(f')\}
\]

is the set of feasible, with regard to the constraints in \( C_\alpha \), and optimal, with regard to the measure function \( m_\alpha \), allocations.

### 4.3.5 Transformation to 0-1 ILP

In this section, we specify a transformation that transforms an allocation specification to a 0-1 Integer Linear Program (0-1 ILP). A solution to the 0-1 ILP corresponds to a feasible and optimal allocation.

The starting point of the transformation is an allocation specification, which was defined in Paragraph 4.3.4. The idea is that each constraint can be mapped to a corresponding equation or inequality, respectively. Analogously, the measure function is mapped to a corresponding linear function. Also, we show that the transformation is semantics-preserving.
4 Deployment Concepts

**Binary decision variables** For all component instances \( c \in \text{COMP} \) and ECUs \( e \in \text{ECU} \), we introduce binary decision variables \( x_{c,e} \in \{0,1\} \). These binary decision variables are meant to represent an allocation. That is, \( x_{c,e} = 1 \) indicates that component instance \( c \) is allocated to ECU \( e \). In order to make sure that each component instance is allocated to exactly one ECU, we add the following constraint

\[
\sum_{e \in \text{ECU}} x_{c,e} = 1 \tag{4.3}
\]

for all \( c \in \text{COMP} \). Let \( f \subseteq \text{COMP} \times \text{ECU} \) be a binary relation such that \((c,e) \in f \overset{\text{def.}}{\iff} x_{c,e} = 1 \) for \( c \in \text{COMP} \) and \( e \in \text{ECU} \). Since the binary decision variables should represent an allocation, we have to show that the relation \( f \) corresponds to a mapping. For this, we show that \( f \) is left-total and right-unique.

**Lemma 1.** \( f \) is left-total and right-unique

*Proof.* To show that \( f \) is left-total, we have to verify the property: \( \forall c \in \text{COMP} \exists e \in \text{ECU} : (c,e) \in f \). Let \( c \in \text{COMP} \). Since Equation (4.3) holds for \( c \), there exists an \( e \in \text{ECU} \) such that \( x_{c,e} = 1 \). Thus, \((c,e) \in f \). Consequently, \( f \) is left-total.

Next, we show that \( f \) is right-unique. That is, we have to verify the property: \( \forall (c,e),(c,e') \in f : e = e' \). Let \((c,e),(c,e') \in f \). Hence, \( x_{c,e} = 1 \) and \( x_{c,e'} = 1 \). Due to the fact that Equation (4.3) holds for \( c \) and \( x_{c,e},x_{c,e'} \in \{0,1\} \), it follows that \( e = e' \). Consequently, \( f \) is right-unique. \( \square \)

From now on, for \( c \in \text{COMP} \) \( f(c) \) denotes the unique \( e \in \text{ECU} \) such that \((c,e) \in f \) holds.

In order to transform language elements like the requiredHardwareResourceInstance constraint (see Section 4.3.4), we have to express more advanced dependencies such as component instance \( c_1 \) is allocated to ECU \( e_1 \) and component instance \( c_2 \) is allocated to ECU \( e_1 \). To support this, we introduce new binary decision variables and inequalities, which are based on concepts that are presented in [17, p. 81]. For \( c_1,...,c_l \in \text{COMP} \), \( e_1,...,e_l \in \text{ECU} (l \geq 2) \) we introduce a new binary decision variable \( x_{c_1,e_1,...,c_l,e_l} \in \{0,1\} \). The idea is that this variable takes the value 1 if and only if \( c_1 \) is allocated to \( e_1,..., \), and \( c_l \) is allocated to \( e_l \). To achieve this, we have to couple it with the previously introduced binary decision variables. Hence, we add the following inequalities and intermediate binary decision variables

\[
\begin{align*}
x_{c_1,e_1,...,c_l,e_l} & \leq x_{c_1,e_1,...,c_{l-1},e_{l-1}} \\
x_{c_1,e_1,...,c_l,e_l} & \leq x_{c_2,e_2,...,e_l,e_l} \\
x_{c_1,e_1,...,c_{l-1},e_{l-1}} + x_{c_2,e_2,...,e_l,e_l} & \leq 1 + x_{c_1,e_1,...,c_l,e_l}
\end{align*}
\]

If \( l > 2 \), this is repeated for the two intermediate binary decision variables \( x_{c_1,e_1,...,c_{l-1},e_{l-1}} \) and \( x_{c_2,e_2,...,e_l,e_l} \) as well. Accordingly, we have to introduce at most \( 2^{l-1} - 1 \) intermediate binary decision variables and \( 3 \cdot (2^{l-1} - 1) \) inequalities.

Lemma (2), which was proved in [16], guarantees that the newly added binary decision variables exhibit the desired semantics.
Lemma 2. \( x_{c_1,e_1,e_l} = 1 \iff f(c_1) = e_1 \land \ldots \land f(c_l) = e_l \)

**The sameLocation constraint** Let \( \Phi \) be a sameLocation constraint. \( \Phi \) is transformed as follows:

\[
\sum_{e \in ECU} x_{c_1,e,c_2,e} = 1 \quad (4.4)
\]

for all \((c_1, c_2) \in \text{eval}(\Phi_{OCL_E})\). That is, we require that for each tuple \((c_1, c_2)\) from the evaluation result set there exists exactly one ECU \( e \) such that the binary decision variable \( x_{c_1,e,c_2,e} \) takes the value 1, which implies that \( c_1 \) and \( c_2 \) are both allocated to \( e \). Proposition (1) shows the correctness of the sameLocation constraint transformation.

**Proposition 1.** Let \( \Phi \) be a sameLocation constraint.

Equation (4.4) for all \((c_1, c_2) \in \text{eval}(\Phi_E)\) \iff \( f \in F_\Phi \)

Proof.

\[
\forall (c_1, c_2) \in \text{eval}(\Phi_{OCL_E}) : \text{Equation (4.4)} \\
\iff \forall (c_1, c_2) \in \text{eval}(\Phi_{OCL_E}) \exists e \in ECU : x_{c_1,e,c_2,e} = 1 \\
\iff (\text{Lemma (2)}) \forall (c_1, c_2) \in \text{eval}(\Phi_{OCL_E}) \exists e \in ECU : f(c_1) = e \land f(c_2) = e \\
\iff \forall (c_1, c_2) \in \text{eval}(\Phi_{OCL_E}) : f(c_1) = f(c_2)
\]

\( \square \)

**The differentLocation constraint** Let \( \Phi \) be a differentLocation constraint. \( \Phi \) is transformed as follows:

\[
\sum_{e \in ECU} x_{c_1,e,c_2,e} = 0 \quad (4.5)
\]

for all \((c_1, c_2) \in \text{eval}(\Phi_{OCL_E})\). That is, we require that for each tuple \((c_1, c_2)\) from the evaluation result set there exists no ECU \( e \) such that the binary decision variable \( x_{c_1,e,c_2,e} \) takes the value 1, which implies that \( c_1 \) and \( c_2 \) are both allocated different ECUs. Proposition (2) shows the correctness of the differentLocation constraint transformation.

**Proposition 2.** Let \( \Phi \) be a sameLocation constraint.

Equation (4.5) for all \((c_1, c_2) \in \text{eval}(\Phi_{OCL_E})\) \iff \( f \in F_\Phi \)

The proof is analogous to the proof of Proposition (1).
The requiredHardwareResourceInstance constraint  Let $\Phi$ be a requiredHardwareResourceInstance constraint. $\Phi$ is transformed as follows:

$$\sum_{(c_1, e_1, c_2, e_2, \ldots, c_n, e_n) \in x} x_{c_1, e_1, c_2, e_2, \ldots, c_n, e_n} \geq 1$$

(4.6)

for all $x \in Q$, where $Q$ is defined as in Section 4.3.4. That is, we require that there exists at least one tuple $(c_1, e_1, c_2, e_2, \ldots, c_n, e_n)$ in the $x$ set such that the binary decision variable $x_{c_1, e_1, c_2, e_2, \ldots, c_n, e_n}$ takes the value 1, which implies that $c_i$ is allocated to $e_i$ for $i = 1, \ldots, n$.

Note that it is possible to rewrite Inequality (4.6) to an equation by removing redundant elements from the set $x$. Let $\sim_r$ be a binary relation on the set $x$ such that $(c_1, e_1, \ldots, c_n, e_n) \sim_r (c_1', e_1', \ldots, c_n', e_n') \iff \exists \pi \in S_n : (c_1, e_1, \ldots, c_n, e_n) = (c_{\pi(1)}', e_{\pi(1)}', \ldots, c_{\pi(n)}', e_{\pi(n)}')$. $S_n$ is the symmetric group and $\pi$ a permutation. The relation $\sim_r$ is an equivalence relation. Now, the $\geq$ operator in Inequality 4.6 can be replaced with a $=$ operator by just summing up representatives from the quotient set $x / \sim_r$.

$$\sum_{([c_1, e_1, c_2, e_2, \ldots, c_n, e_n]) \in x / \sim_r} x_{c_1, e_1, c_2, e_2, \ldots, c_n, e_n} = 1$$

The resource constraint  Let $\Phi$ be a resource constraint. $\Phi$ is transformed as follows:

$$\sum_{(c_1, e_1, \ldots, c_n, e_n, w') \in w} w' \cdot x_{c_1, e_1, \ldots, c_n, e_n} \leq r$$

for all $(w, r) \in eval(\Phi_{OCL_E})$. This transformation directly corresponds to the semantics definition of the resource constraint. Basically, we just replaced the $x$ mapping, which is defined by Equation (4.2), in Inequality (4.1) with the corresponding binary decision variable.

The measure function  In the following, we show a QoS dimension can be rewritten with the help of binary decision variables. Afterwards, the transformation of the measure function is immediate. Let $q$ be a QoS dimension. Basically, we translate $q$ to the following sum

$$sum(q) := \sum_{(c_1, e_1, \ldots, c_n, e_n, w) \in eval(q_{OCL_E})} w \cdot x_{c_1, e_1, \ldots, c_n, e_n}$$

. Let $m$ be a measure function. $m$ is transformed into the sum

$$\sum_{S \in SERVICES} \sum_{q \in S} sum(q)$$

and this sum represents the 0-1 ILP’s linear objective function.
4.4 Platform-Mapping

In order to specify how a component instance should be integrated for a concrete ECU, the developer specifies a Platform-Mapping, as stated in the MDA Guide [13]. In particular, the Platform-Mapping specifies how the component instance uses the hardware resources and an API of a ECU, e.g., trigger a sensor or an actuator. Thus, the Platform-Mapping specifies how the implementation of a PIM should be realized for a concrete target platform. Examples for a Platform-Mapping are:

- The mapping of MECHATRONICUML data types to data types of a target platform, e.g., MECHATRONICUML LONG to the C-type int64
- The mapping of component instances to tasks of the operating system, as stated in the MECHATRONICUML Techreport [4], where an atomic component instance is executed as a separate operating system’s task
- The mapping of “events” in the PIM to API-Calls, e.g., the access to a sensor or an actuator to a specific API-Call

In the course of CYBERTRON the Platform-Mapping approach was limited to the last point, the mapping of accesses to sensor and actuators to API-Calls, which we introduce in the following.

4.4.1 API-Mapping

Although the allocation specifies which component instances are executed on which ECU, there is still a semantic gap between the PIM and the target platform. The continuous components, which represent sensors and actuators, serve as black boxes in the PIM. They provide data propagated via continuous in-/out-ports to the discrete software components (for details, refer to the MECHATRONICUML Techreport [4]). There exists no concept how data, sent to an in-port, shall be handled, or how the data read via an out-port shall be propagated in the generated code. This section introduces the API-Mapping Language, which copes with this problem.

Goal of the API-Mapping Language

To access the sensors and actuators of a cyber-physical system, the generated code needs to execute the API-Calls provided by the operating system that runs on a concrete ECU. To specify the API of an operating system, we introduced in Section 4.2 the Operating System Language. Now, we need to define which continuous component instance, which represents a sensor or actuator, can be accessed via which API-Call. Therefore, we need a mapping between the continuous component instances and the API of the operating system.

To specify such a mapping, we implemented a textual API-Mapping Language based on Rose’s Bachelor Thesis [14]. The API-Mapping Language defines which
4 Deployment Concepts

API-Calls shall be executed when a particular continuous port instance is accessed. It was created with the following goals:

- Express a mapping of continuous component instances to hardware sensors or actuators
- Specify how the input/output values of the continuous component instance can be written/read in the generated code
- Provide an adapter between the values used in the platform-independent model and the “real” values used in API-Calls

Language Elements & Semantics

The API-Mapping Language allows the specification of concrete API-Calls for continuous component instances based on the specification of the Operating System (refer to Section 4.2). A continuous component instance can have multiple continuous port instances, which provide or receive different data, according to the MECHATRONICUML Techreport [4]. For instance, an in-port for setting the motor speed and an out-port for reading the current speed. Thus, it might be necessary that a continuous component instance executes two different API-Calls, e.g. one for setting the speed and another for reading the current speed. Therefore, instead of specifying one API-Call per continuous component instance, we specify one API-Call per continuous port instance.

Listing 4.7 shows the EBNF Grammar of the API-Mapping Language.

```
MappingRepository = 'MappingRepository:' ID '{'(PortApiMapping || ',')'}'

PortApiMapping = 'PortInstance:' MUML::ContinuousPortInstance '{'execCommand:' Actionlanguage::Entry ['initCommand:' Actionlanguage::Entry']'}

Actionlanguage::Expression = ( APICallExpression | Actionlanguage::LogicalExpression)

APICallExpression = 'API_Call:' OperatingSystemLanguage::APICommand '('/ (Actionlanguage::ParameterBinding || ','))'';
```

A MappingRepository consists of several PortAPIMappings. A PortAPIMapping references the continuous port instance for which this mapping is specified. A PortAPIMapping consists of an required execCommand and an optional initCommand. The execCommand specifies the API-Call which
4.4 Platform-Mapping

shall be executed for reading or writing the continuous port instance’s value. The `initCommand` specifies the API-Call that needs to be executed before further actions can take place, e.g., enable power to a sensor. The `execCommand` and the `initCommand` reference the API-Calls, which have been specified in the Operating System Language (see Section 4.2).

In particular, the Operating System Language (see Section 4.2) specifies all API-Calls which are available by an operating system. The API-Mapping Language finally defines which API-Call of the operating system has to be executed, when a particular continuous port instance is accessed. Moreover, the API-Mapping Language specifies the concrete parameter of an API-Call. Therefore, an `APICallExpression` has a certain number of parameter bindings, according to signature of the API-Call in the Operating System Language. For instance, the Operating System Language specifies the signature of the API-Call `ecrobot_read_sensor(int port)`. This API-Call is called in the API-Mapping Language with the concrete parameter binding `ecrobot_read_sensor(3)`, as shown in Listing 4.8.

For both commands general expressions from the action language (refer to MECHATRONICUML Techreport [4]) can be used. Thus, complex statements can be specified, e.g., values on platform-independent level can be converted to match the parameter of the API-Calls and vice versa.

An excerpt of the API-Mapping for our overtaker vehicle is shown in Listing 4.8.

```
MappingRepository: nxtOSEKMapping{

PortInstance: overtaker_light{
  execCommand:{
    INT value:=API_Call:ecrobot_get_light_sensor(port_id:=3) ;
    if(value<=10){
      value:=0;
    }
    else{
      value:=1;
    }
    return value;
  }
  initCommand:API_Call:ecrobot_set_light_sensor_active(port_id:=3)
}

Listing 4.8: Example: API-Mapping Overtaker
```

The software of the overtaker vehicle consists of a continuous component instance `overtaker_linefollower`, which has the continuous port instance `light`, as shown in Figure 2.2 in Section 2.2. The light sensor is connected to the Lego Mindstorms ECU, which runs the nxtOSEK operating system, as described in Section 2.3. Therefore, we use the API-Calls specified in the Operating System Language, as shown in Listing 4.2 in Section 4.2. The light sensor is connected to the port C of the Lego Mindstorms NXT ECU, with port id 3. Thus, the `init-
4 Deployment Concepts

Command is `ecrobot_set_light_sensor_active(3)`. The light sensor can be read via `ecrobot_get_light_sensor(3)`. Thus, this API-Call is used to provide the value for the continuous port instance `light`.

Moreover, the returned value of the API-Call is adapted to the expected value in the Real-Time Statechart of the component `overtaker_linefollower`. For instance, the Real-Time Statechart assumes that the vehicle follows the black line if the light sensor returns 0, and assumes the vehicle has lost the line when the light sensor returns 1, where the value 0 stands for black and 1 for white. However, the API-Call returns values between 0 and 10 for different shades of black, and values > 10 for different shades of white. Thus, within the API-Mapping the value is converted to match the encoding of the Real-Time Statechart.

4.5 Derivation of a PSM

This section introduces our PSM and how it is derived from of the PIM. Within our process, the platform-specific model combines the information contained in the PIM and the Platform-Mapping, such that they are available for the code generation. Furthermore, the PIM is enriched with additional information, which are required for the code generation. In particular, information regarding the communication between continuous and discrete components. The derivation of the PSM is done via a Model-to-Model transformation based on Rose’s Bachelor Thesis [14]. The transformation copes with the problem that it is not specified for continuous components, which represent sensors or actuators, how the value of the continuous port shall be propagated to the hybrid port and vice versa.

The transformation is based on the following ideas: A continuous component, which represents an actuator or a sensor, can be accessed via an API-Call, as defined in the Platform-Mapping (see Section 4.4). Since an API-Call returns or takes as parameters discrete values, the continuous component can be transformed into a discrete component. Then, this transformed continuous component just needs to execute the API-Call specified in the Platform-Mapping. Within the PIM, a discrete component accesses the values of the continuous ports via hybrid ports within a fixed sampling interval [4]. This behavior can be reduced to message-based communication as follows:

- The transformed continuous out-port periodically sends a message to its connected hybrid port, according to the sampling interval
- The transformed continuous in-port periodically receives a message from its connected hybrid port, according to the sampling interval

Thus, the communication between continuous components and discrete components can be transformed to message-based communication, based on the Real-Time Coordination Protocol Periodic Transmission [18]. As a consequence, the continuous ports and hybrid ports can be transformed to discrete ports. This is an advantage for the code generation, because it only has to consider discrete
components and ports.

An alternative to the message-based communication, discussed in our project group, would be a shared variable for each connected hybrid port instance and continuous port instance. Nevertheless, a shared variable has the following disadvantages: Firstly, to access a shared variable the connected continuous component instances and the discrete component instances had to be allocated to the same ECU to which the sensor or actuator is connected. This leads to a restriction of possible allocations and might not be feasible, e.g., due to limited amount of memory of a ECU.

Secondly, a shared variable requires that the code generation deals with two kinds of communication, the message-based communication between discrete components, and the communication between discrete and continuous components. This would have lead to a more complicated code generation, although there is currently no need. Therefore, we reduce the communication between continuous component instances and the connected discrete component instances to message based communication. As a result, they can be allocated to different ECUs, and the code generation can simpler deal with them.

The transformation of the continuous components is separated in the following steps, based on Rose’s Bachelor Thesis [14]

1. The continuous components and their ports are transformed into discrete components and discrete ports
2. The connected hybrid ports are transformed into discrete ports
3. For the transformed continuous components a new Real-Time Statechart is created
4. For the transformed hybrid ports a new region in the existing Real-Time Statechart is created
5. Step 1-2 are similar executed for the the component instances

4.5.1 Step 1-2: Transformation of Continuous Components

Since the continuous components are black boxes without a specified behavior in the PIM, the code generation can not deal with them directly. As stated above, the continuous components, which represent sensors or actuators, can be accessed by an API-Call that returns or takes discrete values. Thus, we can treat them like discrete components.

Firstly, the continuous components are transformed into discrete software components and their continuous ports are transformed into discrete ports, as shown in Figure 4.3 for the overtaker system. Furthermore, a message is created for each transformed continuous port, with the data type of the of the continuous port as a parameter. Secondly, the hybrid ports that are connected to the continuous ports are transformed to discrete ports.
4 Deployment Concepts

Figure 4.3 shows the transformation of the continuous components to discrete components for the **overtaker**, as described in Section 2.2. The dotted ellipses show the data that is exchanged by the ports. Before the transformation, the continuous port provides data as an INT. After the transformation, the continuous port is transformed into a discrete port, which sends the message **LightSensor_Message** that takes an INT as a parameter.

![Diagram showing transformation of components](image)

Figure 4.3: Transformation of continuous components

As a result, the former continuous components and discrete software components are communicating via the newly created messages. Furthermore, all continuous components, all continuous ports and hybrid ports are transformed into discrete components respectively into discrete ports. For these transformed continuous components, and the former continuous and hybrid ports their corresponding behavior is created in the following steps.

### 4.5.2 Step 3: Creation of Parametrized Real-Time Statecharts for the transformed Continuous Components

Next, for each continuous component a **Real-Time Statechart** is created that executes the API-Call and periodically sends or receives a message from the former hybrid port, according to the **Real-Time Coordination Protocol Periodic Transmission** [18]. Since a continuous component can be instantiated to multiple continuous component instances, which might be allocated to different ECUs that run different operating systems, they need to execute different API-Calls. Nevertheless, the multiple component instances share the same behavior specified by **Real-Time Statechart** of their component type. Therefore, the **Real-Time Statechart** should be able to take a specific API-Call, which has to be executed, as an parameter.
4.5 Derivation of a PSM

An example for such a scenario is our overtaking system, as shown in Figure 2.2. The continuous component lightSensor is instantiated twice in the overtaker and in the overtakee, see Figure 2.2. The overtaker and overtakee are executed on different vehicles and their light sensor might be connected to different ports of the Mindstorms NXT ECU. Moreover, both ECUs might run different operating systems. Thus, both component instances must execute different API-Calls.

To cope with this problem, we introduced Parametrized Real-Time Statecharts, which are an extension of Real-Time Statecharts, whose behavior can be parametrized. In the case of a continuous component a Parametrized Real-Time Statechart has three parameters:

- An initialize function, which specifies the API-Call to initialize the sensor or actuator
- An execute function, which specifies the API-Call that shall be executed
- A time constraint, which specifies the minimum interval between two executions of the execute function

The parameters of the Parametrized Real-Time Statechart are created based on the continuous port. The signature of the execute function for a continuous in-port is ```void execFunction(DataTypeOfPort a)``` , and for a continuous out-port ```DataTypeOfPort execFunction()``` .

Moreover, the Parametrized Real-Time Statechart contains two regions. One region for the Real-Time Coordination Protocol Periodic Transmission [18], and another for executing the API-Calls. In case of an in-port, a region for periodic receiving is created. In case of an out-port, a region for periodic sending, based on the Real-Time Coordination Protocol Periodic Transmission [18], is created. The periodic sending, respective receiving is executed within the sampling time interval of the former connected hybrid port.

Figure 4.4 shows the Parametrized Real-Time Statechart for the component lightSensor in the overtaker vehicle. The Parametrized Real-Time Statechart of the component LightSensor consists of two regions. The region PeriodicTransmission_send periodically sends, according to the sampling interval of the former connected hybrid port, the read API value in form of the message LightSensor_Message to the former hybrid port. The region API_Execution first initializes the sensor by executing the initialize function and then periodically assigns the Real-Time Statechart variable apiValue the return value of the API-Call.

4.5.3 Step 4: Creation of Behavior for the Transformed Hybrid Ports

As already described in Section 4.5.1, the hybrid ports are transformed into discrete ports. For these newly created discrete ports a new behavior has to be
specified, with the following properties: Firstly, the behavior of the newly created discrete port has to correspond to the behavior of the connected transformed continuous port, according to the Real-Time Coordination Protocol Periodic Transmission [18]. In particular, for a hybrid in-port the behavior for periodic receiving, and for a hybrid out-port the behavior for periodic sending is created. This new behavior is embedded as a separate region in the Real-Time Statechart of the component of the port.

Secondly, the Real-Time Statechart of the port’s component accessed the hybrid port via a direct reference, as described in the MechatronicUML Techreport [4]. These reference are no longer valid, since the hybrid port has been replaced by the newly created discrete port. To cope with these references, a new Real-Time Statechart variable is created for each hybrid port. This Real-Time Statechart variable stores the data that shall be sent or received via the former hybrid port. Furthermore, all references to the former hybrid port are replaced with references to this Real-Time Statechart variable.

Figure 4.5 shows the modified Real-Time Statechart of the component overtakerDriver in the overtaker vehicle.

A new region PeriodicTransmission_receive is added to the Real-Time Statechart DriveControl and the original behavior is unchanged. This region is the counterpart of the region PeriodicTransmission_send as show in
4.5 Derivation of a PSM

Figure 4.5: Behavior of the component OvertakerDriver

Figure 4.4. The region **PeriodicTransmission_receive** periodically receives the message **LightSensor_Message**, within the sampling time of the former hybrid port, and assigns the **REAL-TIME STATECHART** variable **apiValue** the received value.

### 4.5.4 Step 5: Transformation of Continuous Component Instances

After the continuous components and their continuous ports have been transformed to discrete components and discrete ports, the continuous component instances are analogously transformed into discrete software component instances. These software component instances are typed over the discrete components, whose behavior is specified by a **PARAMETRIZED REAL-TIME STATECHART**. Their **PARAMETRIZED REAL-TIME STATECHARTS** take parameters for the initialize function, execute function, and the time interval between two calls of the execute function. To specify which concrete functions shall be executed by the **PARAMETRIZED REAL-TIME STATECHARTS**, when the component instances are executed, a parameter binding of these parameters is needed. Since different con-
Continuous port instances might have to execute different API-Calls, e.g., one port instance calls an API for setting a value, and another for reading a value, these parameter bindings are specified for each port instance.

To specify these parameter bindings the continuous port instances are transformed into parametrized discrete port instances. The parametrized discrete port instances contain the parameter bindings of the parameters of the PARAMETRIZED REAL-TIME STATECHARTS. In particular, a parametrized discrete port instance contains the concrete API-Call for the initialize function, the execute function, and a concrete time constraint, as specified in the Platform-Mapping in Section 4.4.1.

Figure 4.6 shows the big picture of the transformation with respect to our running example, as described in Section 2.2.

Figure 4.6: Parametrized discrete port instances and parameter bindings

The continuous component lightSensor is transformed to a discrete component and its two continuous component instances lineFollower are transformed to discrete component instances. On type-level the behavior of the component lightSensor is specified by the PARAMETRIZED REAL-TIME STATECHART LightSensorBehavior.

On instance-level the two component instances overtaker_lineFollower and overtakee_lineFollower belong to the structured component instance overtaker and overtakee, which represent the both vehicles. Consider the situation that the overtaker is executed on a Lego Mindstorms ECU, and the overtakee is executed on an Arduino Car. Since both vehicles are executed on different sys-
tems, the instances overtaker_lineFollower and overtakee_lineFollower have to execute different API-Calls to get the light value. The concrete API-Calls are stored for each parametrized discrete port instance, indicated by the black boxes next to the port instances. The port instances of the overtaker_lineFollower and overtakee_lineFollower define different execCommands. Then, the execCommands are used as bindings of the parameter executionFunction of the Parametrized Real-Time Statechart. Therefore, the Parametrized Real-Time Statechart can execute two different API-Calls for the two component instances overtaker_lineFollower and overtakee_lineFollower of the same component type LightSensor.

4.6 Middleware and Code Generation

The preceding sections explain how we prepare given MECHATRONICUML models for the code generation. To reduce the complexity of the code generator, we introduce the so called Codegen model. This model combines all information of the Allocation, the PSM, the Platform-Mapping, and the PDM in one single model file. To create the Codegen model, we created a Model-to-Model transformation. Within this transformation further transformations can be used to simplify the given input models for the code generator. Since a structured component instance does not have any behavior\footnote{This statement is only valid, if we do not consider reconfiguration. If reconfiguration shall be considered in the future, the reconfiguration behavior of a structured component shall be handled by the middleware. However, due to our limited amount of time, we didn’t implemented reconfiguration.}, there is no need to generate code for structured components. Therefore, we use a transformation to flatten the hierarchy of a component instance configuration. Additionally, in future work, it could be useful to store additional information for the code generation in this model. For example, it could be helpful to save, which part of a model has to be executed within one task/thread on the target ECU.

The MECHATRONICUML Tool Suite\cite{19} does provide a code generator, which is able to generate ANSI C99 code for a single ECU. Nevertheless, this code generator does not consider all concepts of MECHATRONICUML, e.g., support for message with parameters. Additionally, it does not provide a possibility to generate code for several ECUs or a distributed target platform respectively. To cope with these problems, we provide a MECHATRONICUML specific-middleware to enable code generation for distributed target platforms. Additionally, we extended the current code generation by a new message concept to provide messages with various parameters.

In Section 4.6.1, we first introduce the concepts for a MECHATRONICUML-specific middleware. After that, we explain the concrete implementation of our concepts in Section 4.6.2. We do not focus on implementation details, but give an overview of the general idea and design decisions.
4.6.1 Middleware

During design of the PIM, it is not defined, whether communicating component instances will be allocated on the same ECU or not. Therefore, sending and receiving of MECHATRONICUML messages in the generated code for the REAL-TIME STATECHART has to be independent of the chosen allocation of the component instances. This is especially important, if reconfiguration of component instances is allowed during runtime. Additionally, it is important that all port- and connector properties of the PIM model are respected. For example, if the connector defines a specific message latency, the middleware has to ensure that this latency is respected, e.g., by choosing a fast communication channel. These decisions can be also influenced by a certain set of hardware properties, e.g., the capacity of the bus, which is used for the communication between two ECUs. To support these functionalities, we provide a MECHATRONICUML-specific middleware.

As shown in Figure 4.7 the middleware layer (ML) is located between the application layer (AL), which contains the generated code for component instances respectively REAL-TIME STATECHARTS, and the platform layer (PL). We assume that the PL does not only contain the operating system of the platform, but provides an API to access the hardware, e.g., to read sensor values. The middleware again, uses the PL to communicate with the middleware of other ECUs. The middleware hides the distribution of the component instances, since every component instance only interacts with the middleware, but not directly with another component instance.

![Diagram of middleware layers](image)

**Figure 4.7:** Using a middleware, the application layer is independent from the platform layer.

Figure 4.8 shows the complete structure of the middleware. The Application Layer does not interact with the operating system or the PL directly, but only with the middleware. The middleware itself is structured in three main components to increase the maintainability: **Middleware Core**, which provides the core functionality of the middleware and internal functions, **Middleware Routing**, which
is responsible for determine the routing decisions, and the **Network-Interface** component, which delegates the messages to the correct network-interface implementation. If a component instance sends a message, the middleware ensures, that it will be delivered to the correct target component instance. The most important management component are the **Middleware Core**, the **Application Layer Interface**, the **Routing Logic**, and the **Operating System Layer Interface**.

These components have defined interfaces. Therefore, it is easy to extend or change a specific part of the middleware in the future, if needed.

Figure 4.8: Diagram of middleware components and interfaces

In the following, we describe the parts and interfaces of the middleware in more detail. First, we give an overview of the functions of the **Middleware Core**. After that, we describe the **Application Layer Interface**, which defines the interaction of the application layer and the middleware. Then, we explain the **Routing Logic** of the middleware. Finally, we describe the **Operating System**...
Interface, which defines the interaction of the middleware and the operating system.

Middleware Core

This part is the management component of the middleware. It provides the interfaces to the application layer, and manages and delegates all functions of the middleware. For this, the middleware should store information about the whole system, which are needed to provide all these functions.

Since the middleware has to deliver new received messages to the correct target ports, all components (and their ports) should be registered at the middleware. Therefore, the middleware can compute, if the message has to be sent internally or externally to the target component instance. This means, that for every port or component of the system, the containing ECU has to be stored. This information can be derived from the specified allocation. Additionally, a routing table is stored, which contains the information how the message has to be routed to reach to specific target ECU. For this, the Middleware Core interacts with the Routing-Logic of the Middleware, which is described later in this section. Further management or information providing functions should be added in future work. For example, the selection of the network-interface could be influenced by the information of the QoS assumptions, if message loss is possible or not.

Application Layer Interface

The interface, which is provided by the middleware to the application layer, defines a function to send MECHATRONICUML messages, which is called in the Real-Time Statechart code.

If this send-function is called, the specific MECHATRONICUML message is transferred to the middleware. After that, the REAL-TIME STATECHART can assume, that the message is sent and does not have to care about any logic for delivering this message.

The middleware delivers available messages to the port-buffers of the component instances. After a message is delivered by the middleware to a specific buffer of a port instance, the corresponding REAL-TIME STATECHART is able to consume this message.

Figure 4.9 shows a sequence diagram for an exemplary message exchange: Since some functions are called periodically, the calls of these functions without any effect to the system are grayed out. First, the REAL-TIME STATECHART sends a message, using the send-function of the middleware. The middleware determines the target ECU and sends this message through a network-interface to another ECU. At some point, the middleware receives a message from another ECU by executing periodically a receive function for all network-interfaces. In the Figure 4.9, there is only one used network-interface. Therefore, only one receive-function NI_I2C_receive() is called by the middleware. First, the middleware determines
the correct target ECU. After that, it delivers the message to the concrete port instance. The Real-Time Statechart checks periodically for a specific message, if needed. If the message is contained in the port, the Real-Time Statechart consumes this message.

Figure 4.9: Sequence diagram for message exchange using a MECHATRONICUML-specific middleware

Using this interface, the communication-specific code of the application layer only needs to interact with the middleware. Therefore, from the viewpoint of the Real-Time Statechart, the communication is independent of the allocation, the available network-interfaces, or the target platform, assuming that the middleware cares about delivering messages to the correct target port, no matter where the corresponding component instance is allocated.

Routing Logic

After receiving a message (from the application layer or from a network-interface), the middleware has to determine the target port for the message. This piece of information can directly be derived from the PIM. Since the middleware stores the ECU for every possible target port, the corresponding target ECU for the port can be determined by the middleware. If the target ECU is equal to the own ECU, the message can be delivered to the correct target port of the component instance. If the target ECU is a different ECU, the middleware has to send it to the target ECU via a communication channel. For this, two pieces of information have to be determined: The Next-Hop-ECU and the network-interface, which is used to send the message physically.
**Next-Hop-ECU:** Since not all ECUs are connected directly to all of the other ECUs, it is possible that the message has to be routed to the target ECU via other ECUs. Figure 4.10 shows a small example for this: If a message has to be sent from Overtaker.B1 to Overtakee.B1, it has to be routed over Overtaker.B2 and Overtakee.B2. In future versions of the middleware, the decision could be based on the properties of the connector (defined in the PIM), on the properties of the communication channel (defined in the PDM), and on online-decisions, e.g., availability of the communication channel. In this example, the middleware will determine Overtaker.B2 as Next-Hop-ECU. When Overtaker.B2 receives the message, the middleware of Overtaker.B2 will handle this message in the same way. For this, it determines a Next-Hop-ECU, in this case the ECU Overtakee.B2. This will be done, until the message is delivered to the target ECU Overtaker.B1.

**Network-Interface:** A network-interface is a concrete physical communication interface of an ECU, e.g., Bluetooth, I2C, or busses. If the Next-Hop-ECU is determined, the middleware selects a network-interface to send the message to this ECU. This information is again derived from the PDM. In the example in Figure 4.10, Overtaker.B1 and Overtaker.B2 are connected via an I2C bus. Therefore this specific network-interface is chosen.

![Figure 4.10: Routing message over several ECUs](image)

After executing these steps, the message is ready to be sent. For this, the middleware has to interact with the underlying operating system, which manages the access to the hardware.

**Operating System Layer Interface**

To send a message via a network-interface, the middleware calls functions of the operating system. Since this implementation is highly platform-, system-, and...
configuration specific, we propose that all concrete network-interfaces has to be implemented manually by the platform engineer. For this, the hardware architect has to provide a send-function for every network-interface, which can be called by the middleware to send a message via a specific network-interface. Additionally, a receive-function has to be implemented. This function will be called periodically by the middleware to receive messages from other ECUs. For the example of Figure 4.10, the middleware will call the implementation of the I2C-network-interface.

4.6.2 Code Generation

Goal of the code generation is to generate code from the created Codegen model for our target platform: nxtOSEK [20] on LEGO Mindstorm NXT 2.0. For this, we have to generate ANSI C99 code and some platform-specific code like a main-file and an .oil-file, which is needed by the operating system.

For the implementation, we use an existing C-code generator, which is able to generate source code for component instance configurations. We extended the current code generator by several features. The most important features are a new message implementation and the implementation of a MECHATRONICUML-specific middleware to enable generation for distributed systems with several ECUs.

Message Implementation

The current code generation is not able to handle messages with parameters. Since several message parameters of different types per MECHATRONICUML message are mandatory for sophisticated models, we replaced the old implementation for messages, message-buffers, and discrete single ports. The new implementation corresponds to the MECHATRONICUML specification [4], except for so called multiports, which are not implemented yet.

Since the messages might be sent from one ECU to another ECU, we use an existing approach for (de-)marshaling of messages in embedded systems: Protobuf Embedded-C [21], which is an implementation of Googles Protobuf [22] for embedded system. Therefore, we do not generate C-code for all used MECHATRONICUML messages, but a .proto-file. After the code generation, the code for the messages defined in the .proto-file is generated by the protobuf-script. This code includes the struct-code and all needed functions by Protobuf. Since there are implementations of Protobuf for several platforms, the middleware does not have to handle the encoding of the messages by itself, if heterogeneous platforms are used in the system.

Time Units

Until now, the code generation did support only milliseconds as time units for clock constraints in REAL-TIME STATECHART. We enabled the usage of all time units,
which are usable in **MechatronicUML Tool Suite**. In our code generation all time values are translated into milliseconds, since this is the smallest time unit which is supported by the target platforms we have used for testing. Since this information is very platform-specific, these information should be stored in the Codegen model in future.

**Parametrized Real-Time Statecharts**

To implement the Parametrized Real-Time Statechart (see Section 4.5.2), we extended the code generation for Real-Time Statecharts. Since the init- and exec-functions for the API-Calls cannot be defined in the type of the Real-Time Statechart, but have to be defined for every instance, we added for each such function a function-pointer to the Real-Time Statechart-struct.

After creating (instantiating) the Real-Time Statechart during runtime, the Real-Time Statechart gets initialized. At this point, we set the pointer to the correct instance-specific functions.

**Limitations within the Current Implementation**

In the scope of the project group, we implemented a prototype of the MechatronicUML-specific middleware. This prototype provides only the main functionality for a MechatronicUML-specific middleware: Routing of MechatronicUML messages. For this, we only consider static models and not models, which are not changed during runtime, e.g., by deactivating component instances. Additionally, the middleware does not consider QoS-assumptions of the connectors. Therefore, we assume that no network-interface will fail to send the message or will be interrupted completely. Since the middleware is well structured, adding additional features will be possible without changing the whole implementation.

For the MechatronicUML-specific middleware, we implemented three kinds of code: static code for the internal logic of the middleware, model-specific code like routing information, and functions-stubs that have to be implemented by the platform engineer. The static code contains functions for sending and receiving MechatronicUML messages. Additionally, the calls of these functions are generated into the implementation of the Real-Time Statechart and into the receiving-part of the middleware, which receives new messages via the implemented network-interfaces.

Since the middleware manages the message-handling, all needed information are generated for the middleware: We are only considering static distributed systems. Thus, the routing does not support changes in the routing table during runtime. Therefore, we determine the routing tables for all ECUs offline during the code generation. Online changes (during runtime) are not considered, e.g., if a network-interface is not available. Additionally, the middleware stores only one Next-Hop-ECU per target ECU and one network-interface. As a result, the middleware stores only one possible way for routing a message to a foreign ECU, although there are
other possible Next-Hop-ECUs in the PDM. To determine the routing tables, we mapped the hardware model to the JGraphT Framework [23] and used standard graph algorithms to solve the shortest path problem. Using this framework, it will be easy to extend this implementation by integrating more information of the model for routing decisions.

For every ECU there is platform-specific code, which cannot be derived from the models. For example, the implementation of network-interfaces is very platform-, operating system-, and network-specific. Therefore, we only generate function-stub for every modeled network-interface. Therefore, the hardware engineer has to implement for each network-interface an initialize, send, and receive-function. When the middleware uses a network-interface to send or receive a message to another ECU, the implementation of these functions is called.
5 Evaluation

This chapter presents our evaluation results and current limitations of our work.

We use the Goal/Question/Metric method [24, 25] for the evaluation of our project. The method helps evaluating projects defining goals, questions, and metrics for it. For details of our use of the GQM Method we refer to our requirements specification [2].

GQM helped us defining the evaluation goals used in this chapter. In Section 5.1, we present the evaluation of our implemented verification concepts, as well as current limitations of the verification process. Section 5.2 shows our evaluation results and limitations of the implemented deployment concepts.

5.1 Evaluation of Verification Concepts

In this section, we evaluate the verification process with respect to the goals that we initially defined using GQM. For our implemented verification concepts, we identified the following goals: Completeness of the translations (Section 5.1.1), transparency (Section 5.1.2), semantics preservation and correctness (Section 5.1.3), and efficiency (Section 5.1.4).

5.1.1 Complete Translation

One of the goals of the verification process was to ensure a complete translation of Real-Time Coordination Protocols to Uppaal, so that we consider all MechatronicUML syntax elements in the verification. Also the backward translation of the traces should be complete to ensure that the user receives all relevant information. In addition to that, we intended to completely translate all relevant MTCTL verification properties to Uppaal.

For the forward translation, we achieved our goals to a large extent and translate most MechatronicUML elements to Uppaal. However, there are still some elements that are not translated yet, such as transition- and region priorities (resulting in an over-approximation).

For the backward translation, we achieved total completeness and are now able to display all information reflected in the Uppaal model (such as values of variables and clocks, firing transitions, and messages in buffers).

The Verification Property Language MTCTL is able to express all relevant properties that are directly expressible in Uppaal. MTCTL is even more expressive than Uppaal, allowing to specify single MTCTL properties that take multiple
5 Evaluation

UPPAAL queries to verify. While we already provide many powerful constructs, the language is prepared to be extended by additional predicates and mappings. A more thorough evaluation should show what is needed in practice.

5.1.2 Transparency

We planned to create a transparent verification approach. This means that the user should not have to be concerned with the fact that UPPAAL is used for the model checking, and only operate on MECHATRONICUML.

In our approach, the information in the backward translation and property result evaluation are purely based on the MECHATRONICUML platform-independent model, independent of concrete UPPAAL representation. Moreover, MTCTL properties can be specified without requiring knowledge of the UPPAAL model or UPPAAL’s TCTL syntax.

Consequently, our current state completely fulfills this goal. It is not necessary that users know UPPAAL nor do they have to interact with it in any way.

5.1.3 Semantics Preservation and Correctness

We included the goal of semantics preservation and correctness to ensure that all information is correctly translated to and from UPPAAL.

This included that all MECHATRONICUML elements are correctly mapped to UPPAAL and that the results of the verification are correct. For the backward translation it means that the counterexample traces correctly represent the verification results.

Our current state fulfills this goal to a large extent. Due to the partially incomplete translation to UPPAAL (see Section 5.1.1), certain elements are not considered yet. This results in an over-approximation that might potentially lead to wrong verification results (false negatives for safety properties, false positives for reachability properties).

For the backward translation, the missing snapshot semantics of MECHATRONICUML make it impossible to evaluate correctness. However, currently, certain extraneous (duplicate) snapshots may appear in traces, where something happens internally in the UPPAAL model without having consequences on the MECHATRONICUML level.

5.1.4 Efficiency

To be feasible in practice, the verification process should be space and time efficient.

The verification for our running example Overtaking protocol using our default properties and a safety property currently takes about 40 seconds. Since bugs in already deployed software generally lead to high costs, this indicates that
5.2 Evaluation of Deployment Concepts

In this section, we evaluate the implemented deployment concepts with respect to the goals that we initially defined.

5.2.1 Maintainability of the Deployment Process

One of our goals was to define an understandable and maintainable deployment process that can be easily extended. The maintainability of the deployment process refers to an easy exchange of implementations, and adding further refinements and steps into the process.

In general, the first step towards the maintainability of the deployment process is this documentation. For the allocation, we achieved our goal to a large extent, since the underlying Integer Linear Program Solver can be exchanged in our implementation with an arbitrary solver, e.g., a “faster” commercial solver. Moreover, the ability to specify a library for sophisticated allocation constraints provides a good reusability.

For the code generation, we achieved this goal by providing the Codegen model as a general interface for any code generation. Thus, any future and further code generations can build up on this model, which provides a common structure of the platform-specific model, the platform-mapping, and the code generation.

However, concepts for the analysis of the PSM and the generated code are left for future work.

5.2.2 Appropriateness of the Deployment Process

We planned to develop an appropriate deployment process, which allows the deployment of the specified MECHATRONICUML platform-independent model to a target platform. This means, that the user should not be forced to executed unnecessary steps, which could be done automatically or do not prove any further advantages. Moreover, the specified deployment process should result in a running system.
Our current state fulfills this goal to a large extent. The deployment process has been extensively tested with our running example, that can be regarded as a running system. In our current implementation it is not mandatory to produce the Platform-Mapping and the operating system model. Nevertheless, it has to be evaluated if the Platform-Mapping, in particular the API-Mapping, can be automatically derived or be skipped in the process.

### 5.2.3 Semantics Preservation and Correctness

We included the goal of semantics preservation and correctness to ensure that all information specified in MECHATRONICUML’s PIM are correctly translated in source code. In particular, the component instance configuration and the REAL-TIME STATECHARTS have to be correctly transformed to source code.

Our current state fulfills this goal to some extent. However, the verification of the generated source code with respect to the PIM is outside of the scope of our project group. Although most MECHATRONICUML elements are transformed to source code, the following are left for future work:

- Multi-Ports of Components
- Optional-Ports of Components
- Quality of Service Assumptions of Connectors
- Do-Event of States
- Reconfiguration

To support these features in future work, the code generation needs to be adapted.

### 5.2.4 Transparency

We planned to create a transparent deployment process. This means, that the user should not be force to adapt the generated code manually or to interact with the used Integer Linear Program Solver, but only interact with the MECHATRONICUML TOOL SUITE.

The current state of our implementation partially fulfills this goal. For the allocation the user can use the high level Allocation Specification Language, without concerning about Integer Linear Programs and the used solver. However, for the code generation the user needs to implement the functions for the network-interfaces for each ECU manually. For instance, the user has to implement the method stubs for sending and reading a message via bluetooth or RS485 on the Lego Mindstorms ECUs. Nevertheless, for our running example we had the experience, that the implementation of the network-interfaces does not require more than a few lines of code per network-interface.
5.2.5 Efficiency of Code Generation

Our goal of efficiency of the code generation is strongly coupled to our running example and the Lego Mindstorms NXT ECUs. The purpose of this goal is to ensure that our generated code is time- and memory efficient to deal with the limited resources of the NXT ECU. In particular, the NXT ECU provides the following resources:

- Atmel-ARM processor 48 MHz
- 64 KB RAM
- 256 KB ROM

Our state, concerning the running example, fulfills this goal to a large extent, since the generated code can be executed on the Mindstorms NXT ECUs. Table 5.1 shows the size of the compiled code and its memory consumption for our running example (see Section 2.2), consisting of 13 component instance, 8 REAL-TIME STATECHARTS, with in total 37 states.

<table>
<thead>
<tr>
<th></th>
<th>Linux 32bit</th>
<th>Lego Mindstorms NXT (for each of the four ECUs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary size</td>
<td>≈100 KB</td>
<td>≈60 KB</td>
</tr>
<tr>
<td>memory consumption</td>
<td>≈2 KB</td>
<td>≈1.8 KB</td>
</tr>
</tbody>
</table>
6 Related Work

In this chapter, we present several related works we considered for our approach. In Section 6.1, we present two existing approaches in which external model checkers are used to verify certain verification properties of specific models. We also show a related domain specific language for specifying verification properties. Afterwards, in Section 6.2, we present three approaches for a software deployment and middleware concepts in the field of cyber-physical systems.

6.1 Model Checking

This section introduces related work in the context of our verification approach.

6.1.1 Theseus

Goldsby et al. present Theseus, a generic visualization framework that allows the display of verification results and counterexamples [26]. The framework takes as input UML models and the result of one of the supported model checkers. It parses and exports the model checker trace into an XML format, employing name matching to relate the trace elements to the input UML models. The resulting XML file is then visualized by the framework in UML syntax.

The display of counterexample traces in Theseus using the original syntax strongly facilitates understanding property violations. This is comparable to our backward translation. However, Theseus currently does not support timed model checking. Furthermore, our backward translation contains MECHATRONICUML-specific elements (e.g., clocks, message buffers) that are not part of usual UML diagrams.

6.1.2 TopCased

TOPCased is a toolkit for the model-driven development of safety-critical systems [27]. Like the verification approach in MECHATRONICUML, the user creates models which are transformed and put into a model checker. Then, the results are shown on the level of the domain model.

The authors present a general approach for model checking domain-specific languages. As an example, they show how they check business models for liveness and safety properties. The properties are formulated using their language TOCL, an OCL-based language which has been extended by temporal operators.
In the beginning of the project group, OCL-based approaches were analyzed with respect to applicability for MECHATRONICUML [8]. Our results were that these languages are not suitable for domain experts and come with an unnecessarily high complexity. Our Verification Property Language MTCTL offers a syntax similar to the well-known TCTL for specifying verification properties for MECHATRONICUML and offers MTCTL2English translations.

In general, their approach is quite similar to our verification approach, differing mostly in implementation details. For example, they strictly apply an MDA approach, meaning that their model-checker works on a specific meta-model that augments the static model with runtime information. In MECHATRONICUML’s verification process, a snapshot is not represented explicitly by a model (during model checking) but rather implicitly in an UPPAAL state (which the backward translation is then able to map to our runtime model).

### 6.1.3 First-Order TCTL

First-Order TCTL (FO-TCTL) is a language for specifying verification properties. It was introduced in [28, 29]. Like MTCTL, FO-TCTL is based on predicate logic and TCTL. It is defined for general (timed) systems where each snapshot can be described as a finite set of elements. The general idea is that the universe for quantifiers is the state set. Predicates allow to make statements about these elements in the set (e.g., the set may contain all active states of a statechart, and variable-value tuples).

In contrast to MTCTL, FO-TCTL exhibits a low degree of abstraction and is more oriented towards being easily analyzable by model checking experts. As a result, FO-TCTL properties are rather difficult to read and understand for domain experts [28]. One advantage of FO-TCTL’s approach is that reconfiguration is supported at its very core (for example, new elements resulting from reconfiguration can simply be added to the state set). While less generally applicable, MTCTL is easier to use and verification properties are much more succinct. One significant example of this is the fact that MTCTL properties are defined on the design model, which is familiar to the engineer, rather than an abstract set representing a snapshot. In MTCTL, elements like states can be directly referenced and do not have to be identified using a conjunction of predicates.

### 6.1.4 CurCuMA

The Culture and Community-based Modeling Approach (CurCuMA) is a conceptual framework for multi-agent systems [30]. Connected to CurCuMA is also a verification approach for the behavior of multi-agent systems.

The focus there is on verifying the structure of a system with respect to reconfiguration of groups of agents. In this compositional verification approach, it is assumed that the involved entities follow certain behavioral protocols and that
these protocols themselves are verified for safety. Our work allows verifying these protocols and therefore complements the verification approaches for CurCuMA.

6.2 Deployment

In the platform-specific part of our project group, we considered few related works: CHROMOSOME [31], AUTOSAR [32], EAST-ADL2 [33], C-Forge [34], and ArcheOpterix [35].

6.2.1 CHROMOSOME

CHROMOSOME (Cross-domain Modular Operating System or Middleware - in short XME) is a middleware and runtime system, directly targeted to Cyber-physical systems. Its focus is to satisfy requirements from the embedded domain, like predictability and safety, but also requirements from the Internet domain such as adaptivity and plug&play capability [36]. The current release comes with two main components [37]:

- CHROMOSOME (XME) runtime system. A runtime system with a hardware abstraction layer (HAL).
- XMT (CHROMOSOME Modeling Tool). An eclipse-based tool worth automatic code generation capabilities for static configuration of the target platform [36].

In contrast to MechatronicUML, it currently only supports Linux- and Windows-based systems as target devices and no micro-controller environments like nxtOSEK [37]. Additionally, CHROMOSOME uses a data-centric design, while we use a message-based design in our MechatronicUML-specific middleware. Also, we developed concepts to consider quality of service properties in our middleware, which CHROMOSOME currently lacks.

6.2.2 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a standardized open software architecture for automotive ECUs [32]. Like the MechatronicUML deployment process, AUTOSAR has similar defined process, called AUTOSAR Methodology [38]. AUTOSAR defines a standard, but there is no free implementation (Artop is only available for AUTOSAR-Member and not open-source). MechatronicUML on the other hand is a method, which also provides a free implementation in form of the MechatronicUML Tool Suite. Moreover, AUTOSAR provides a more complex layer structure and middleware (called Basic Software) than the MechatronicUML. MechatronicUML layers and middleware are currently in a quite simple state with less layers.
6.2.3 EAST-ADL2

EAST-ADL2 [33] (Electronics Architecture and Software Technology - Architecture Description Language 2) is a language for describing the architecture of embedded automotive systems. It enables the specification of ECU-Networks and (non-automatic) allocation of software components to ECUs [33]. In contrast to MechatronicUML, it does not allow the specification and verification of (communication) behavior [33]. Also, EAST-ADL2 does not support runtime reconfiguration [4].

6.2.4 C-Forge

C-Forge [34] is a model-driven tool-chain which supports a component-based development process. While this sounds very similar to what our MechatronicUML tool suite does, there are big differences in how this is done: C-Forge relies on a component model and a component framework. The component model is used to model components, ports, and behavior, and the framework is used to define the way in which the models are executed [34]. While in MechatronicUML we transform our PIM and refine it with additional models like a PDM, C-Forge transforms the component model to a framework component, which can then be refined with information about how these components will be executed (e.g., number of threads or processes) [34]. In contrast to MechatronicUML, C-Forge is not used to generate code for the target platform, because a so-called model loader is used to automatically interpret both, the normal component model, as well as the framework component, which was created out of the component model [34].

6.2.5 ArcheOpterix

ArcheOpterix is an Eclipse plug-in for architecture evaluation and optimization [35]. It can be used to solve the component deployment problem [35], which corresponds to our allocation problem.

For the constraint specification, ArcheOpterix provides an interface, which can be implemented to specify a certain constraint [35]. Analogously, it provides an interface that can be implemented to provide the objectives, which are used to evaluate an allocation. Since the implementations are written in Java, it is possible to provide quite complex constraints and evaluation metrics.

To solve the allocation problem, ArcheOpterix uses an evolutionary algorithm [35]. This algorithm supports two approaches to find an optimal allocation. In the first approach, all specified objectives are mapped to a single objective function, which is represented by a weighted sum [35]. In the second approach, the evolutionary algorithm searches a set of allocations near the Pareto front [35]. Then, the deployment engineer has to choose one of these nearly Pareto optimal allocations.

In contrast to ArcheOpterix, which uses Java to implement the constraints and objectives, the Allocation Specification Language uses OCL. The advantage of
OCL is that the deployment engineer can directly specify the OCL expressions on the MechatronicUML design model. This is more suited than writing low-level Java code. Nevertheless, Java allows to write more expressive constraints and objectives.

Similar to ArcheOpterix, the Allocation Specification Language supports the optimization of a single objective function. To compute an allocation, we transform the allocation specification to a corresponding 0-1 ILP. The advantage is that the 0-1 ILP approach is more predictable and reproducible, because usually no randomness is involved when solving the 0-1 ILP (although this depends on the employed ILP solver).
7 Lessons Learned

The project group offered the possibility to gain experiences in different fields. Unfortunately, not everything went well during the last year, but we were able to learn from problems and challenges. Since this might be important for future project groups, we share our lessons learned in the following sections. In Section 7.1, we describe our experiences using Third Party Tools. After that, in Section 7.2, we give an overview about the usage of GQM as an evaluation technique for the project group. In Section 7.3, we present some important decisions for the internal organization of our group, and finally, in Section 7.4, we explain our experiences in writing documents, such as this one.

7.1 Relying on Third Party Tools

For implementing our artifacts, we strongly relied on third-party tools, implementations, and frameworks. During the implementation, it turned out that some of these third-party tools caused some trouble, and thus delayed the original project plan. These problems occurred although some of us had experience with these third-party tools, and we studied the documentation and tutorials before we actually started. This lead to the inside that the implementation of third-party tools can be immature, despite promising tooldemos, and documentation.

To avoid these problems, the project plan should have much space for the evaluation of third-party tools. The evaluation of third-party tools should not only include reading the documentation, but also checking for an active community, checking for active support and if possible writing test cases. Moreover, the project planning should consider that the adaption of some third-party tools fails, and thus have time buffer for switching the tool or finding another solution. Furthermore, you should agree on a deadline for the integration of a third-party tool. After this deadline, you should evaluate, if it is worth to spend more time on that tool.

7.2 GQM

The "Goal/Question/Metric"-method was used with the intention to have a well structured plan for measuring and interpreting quality aspects of the products of our project group, as well as evaluating certain aspects, like shown in Chapter 5. For these products, we defined multiple goals, which were characterized with questions that help determining if these goals are fulfilled. To have an objective
7 Lessons Learned

measure of these, our questions are answered by measuring self-defined metrics and comparing these with hypotheses, which we also have to define.

We came up with goals quite fast. Defining the questions required more time, but it was also a task we did not struggle with. Although, we defined questions which characterize our goals in a proper way, defining objective metrics and hypotheses, which were able to answer these questions, was where we spent most of the time and had the most problems.

For example, one of our goals was the maintainability of the software allocation. One of the questions for this goal was, if the code is documented. But what is a good metric for this and what a good hypothesis? We ended with a very subjective metric, we called ”degree of commented code” and the hypothesis ”high”. In hindsight, we could have formalized this better, for example with questions like ”Is every method commented?” , but this also is not a good measure for the quality of our software allocation code. Additionally, measuring metrics like this is a rather tedious task, for which we would have had no time in the end-phase of our project group.

By using GQM, we had to define specific goals, which helped us to set a focus for future work, in the early phase of the project group. Apart from that, we now have a very rough approximation of the quality of our project. During the project group, GQM was not of much help. We think that it did not influenced our work as we expected.

7.3 Internal Organization

In this section, we give an overview of (in our opinion) relevant aspects of the internal organization of the project group.

Building Subgroups Based on the main topics of the project group, we divided the group into subgroups: one for the Verification-part, and one for the Deployment-part. Additionally, we built a third group for the Running Example, whose members came from both other subgroups. This led to a low coupling across the subgroups and a “high” cohesion within the subgroups. In our weekly meetings, we discussed important questions, problems and results of the subgroups with the whole project group, as described below. Introducing this hierarchical structuring of the project group, we were able to work more efficiently.

Responsibilities For every task or topic, we assigned a responsible person. This person was responsible that this specific task will be executed (successfully). This covers general responsibilities like group management, test management, or other organizational tasks as well as responsibilities for specific tasks like maintaining the Gantt chart, or writing this final documentation. The responsible person had not to complete the whole task by her/himself, but was in charge to assign subtasks to other persons and to merge
the outcome of all subtasks. For example, for writing this document, one person was responsible to manage the whole writing process. For this, he assigned every section to one specific person. These assigned persons again were able to delegate the writing of subsections to several other persons, who are experts in the topic of the subsection, or (e.g. for short sections) to write it by himself. The advantage of this clear task assignment is, that there is no “someone will do it”-task, which usually become “no-one will do it”-tasks. Thus, we were able to meet our deadlines and to distribute the work (mostly) equally to all members of the project group.

Meetings During the whole year, we had regular weekly meetings. In this meetings, we discussed the most important topics and tasks for the next week and assigned them to specific persons. In every meeting, the subgroups presented their results of the last week and sometimes questions and problems which had to be discussed by the whole project group. Certainly, not every question or problem of a subgroup had to be discussed in the regular PG-meeting. To save meeting time, every subgroup specific topics where discussed in additional subgroup meetings. The results of these meetings (decisions and still open questions) were then presented in the PG-meeting. Additionally, we defined three milestone meetings (corresponding to the milestones, defined in the Gantt-Chart). These meetings were useful to get an overview of the overall state of the whole project and were used to discuss general design decisions, next deadlines, or changes in the Gantt-Chart.

Using SVN and Trac For managing all of our files and documents, we used SVN. For the Verification-group it was more useful to work on the current version of MechatronicUML instead of forking the whole MechatronicUML project. Therefore, the implementation files of the Verification-group were stored into the MechatronicUML-Verification-SVN. Since the Deployment-group was not affected by the implementation of the Verification-group, this split had no negative consequences for the implementation process. On the other hand, we used a Trac-System. Since we used two different SVN and the Trac-Systems were coupled to the corresponding SVN, we had to maintain the tickets of two Trac-Systems. This made it more complicated to get an overview of the implementation of all subgroups. In the beginning, we decided to use the Trac-System for all tasks of the project and for bug-reporting. Since we defined and assigned all tasks in the regular weekly meetings, writing and maintaining a ticket for this got superfluous. Therefore, we decided to use the Trac-Systems for bug-reporting and features, which are not defined in the Gantt-Chart or in the meeting explicitly only.

Preparation Phase Before we started the project, we had a preparation- and –as usual– seminar-phase for the project group. In this phase, everyone of the group prepared workshop for a specific topic, which would be important for the project phase, e.g. for Xtext, QVTo, or UPPAAL. The goal of these
workshops was to ensure that all group members know the basics for tools, languages, and technologies, which are used in the project phase. This was useful in the weekly meetings during the project phase, since all of us were able to understand problems without introducing everything from scratch. In the seminar phase, everyone had to write a seminar thesis to a specific topic, which was related to the planned project phase. Additionally, the results of every thesis were presented. Usually, every group member presents her/his topic separately. This would have resulted in nine different looking, non-coherent presentations. Therefore, we extended the seminar-phase by two weeks to coordinate all presentations. Thus, we were able to present all seminar-topics using one big story and using consistent presentation styles. Unfortunately, two weeks are a lot of time, which could have been used for project preparation. Nevertheless, this consistency-phase strengthened the team and the collaborative working in the group in a very early point of time, which again saved time during the project phase and eased the planning phase of the project phase a lot.

7.4 Write Meaningful Documents

In this project group, we had to write jointly large documents like the Requirements Specification, or this document. During writing the Requirements Specification, we faced several problems. Most of us had different ideas of what this document should be about. Although, we assigned the organization of all sections to specific persons, this led to inconsistency initial version of the document. We think, the main problem was, that the purpose of the document was not defined precisely enough. Even when we faced first inconsistency, we did not stop writing, but changed just the specific parts of the document. Thus, we had to spend a considerable amount of time for unification. In the end, we had a (more or less) consistent document, but its content was not of great value to us.

To avoid this in the next documents, we added an additional planning step, before we actually started writing. After assigning persons to each section, we wrote a planning text for every section and subsection. This text described the purpose and the (summarized) content of the section. After that, we discussed the whole document until the structure and purpose was clear and accepted by everyone. This small amount of additional planning time saved a lot of writing- and discussion-time during the writing-phase. However, it is advisable to stick exactly to the defined aspects in the planning discussions. Otherwise, the planning phase will indeed fulfill its purpose, but will take a lot more time than needed.

We think, this technique improved the content and meaning of our documents. We also recommend this not only for writing documents, but also for creating presentations.
8 Future Work

This section gives for both, the verification and deployment of MechatronicUML models, an overview about all open questions, unfulfilled goals, and problems that require further research.

8.1 Verification

Formal MechatronicUML Dynamic Semantics Definition  A general problem that we faced during Cybertron, was the fact that there does not exist a formal definition of MechatronicUML’s dynamic semantics. Our implementation of the forward translation is based on the description in [4], which is given in natural language and can therefore be ambiguous.

When such a formal definition is created, the forward translation has to be checked against this definition. Furthermore, as pointed out in Section 3.3.2, we currently map one Uppaal state to one MechatronicUML snapshot. The problem here is that a state change in the MechatronicUML model can result in a sequence of state changes in the resulting Uppaal model. This raises the need of mapping several Uppaal states to one MechatronicUML state in the backward translation of traces. Based on a formal dynamic semantics definition this can be improved.

MTCTL Evaluation  For the Verification Property Language MTCTL, we need more substantial use case examples to ascertain whether the currently supplied predicates and mappings are sufficient for applications in practice. If they are not, further predicates and mappings should be added. Furthermore, MTCTL together with the forward translation could provide more advanced means to check certain properties. For example, an operator timeInterval could be introduced to express properties like “between event a and event b, (at least x and) at most y time units pass” [8]. Such properties can be verified by (automatically) adding additional clocks, locations, and edges to Uppaal and running a reachability check for unwanted situations.

Region and Transition Priorities  Region and transition priorities are not considered during the forward translation. This leads to an over-approximation of the system’s behavior when verifying properties with Uppaal [1]. Namely, the resulting NTA contains more paths than the original RTCP. Checking safety properties (properties with an A path quantifier) this can lead to false negatives. This
means, if there exists a path in the NTA on which the property does not hold, it is possible that this path does not exist in the original RTSCs.

In the case of reachability properties (E path quantifier), the neglect of priorities can even result in a false verification. This means, if there exists a path in the NTA on which the property holds it is possible that this path does not exist in the original RTSCs.

A possible solution to translate transition priorities is stated by Hirsch [39]. This solution is based on the negation of transition guards, disabling transitions with a lower priority when a transition with a higher priority is enabled. Additionally, a certain modeling construct is added to support transitions that are enabled by events (synchronizations or trigger message events). A solution to translate region priorities is still subject of research.

However, we think that an extension of the forward translation by transitions priorities should be considered as not trivial. Moreover, currently this seems not to be realizable in the form of a normalization. MechatronicUML does not support any syntax element for checking whether a message is in a buffer and therefore ready to consume. This has to be supported in order to realize the concept of [39] on the MechatronicUML level. Concepts for the translation of region priorities are still subject of research.

**Urgent Transitions with Clock Constraints** Furthermore, MechatronicUML supports guards on urgent transitions. Such transitions are currently not supported by the forward translation, because UPPAAL does not allows guards on edges that synchronize via an urgent synchronization channel. A possible solution is to forbid clock constraints on urgent transitions in MechatronicUML. Therefore, it has to be evaluated if this semantics is needed in MechatronicUML.

Another solution could be the translation of the concept within the translation of priorities based on [39]. Here, urgent transitions could be translated to nonurgent ones. In addition, invariants could then be used to force the firing of these transitions. Furthermore, the precedence of urgent transitions has to be reflected by this translation.

**Verification of Components and Component Instance Configurations** Currently, we perform the verification only on RTCPs. However, as mentioned in Section 3.2.2 we create a certain kind of CIC for these RTCPs in the first transformation of the forward translation. The rest of the verification operates on these CICs. MTCTL already supports the definition of properties on CICs and on AtomicComponents. Adapting the migration to fully support the translation of CICs to UPPAAL’s NTAs, would therefore enable the verification of CICs.

Furthermore, our approach of translating RTCPs to CICs can be used in the context of verifying atomic components or a structure of flat component parts that are embedded in a structured component. Creating a CIC for an atomic component would enable verification on this component. Here, it has to be decided whether to
8.2 Deployment

Reconfiguration Our approach currently does not support reconfiguration. Furthermore, we translate RTSCs into CICs (with concrete multiplicities) and perform the verification on the CIC’s UPPAAL equivalent (see Section 3.2.2).

Therefore, the adaptation of the approach to reconfiguration would not only require to translate the reconfiguration rules to UPPAAL, but also to rethink (and most likely to restructure) the whole approach with respect to variable multiplicities. We assume this to be difficult and heavily time consuming. Furthermore, in order to consider reconfiguration with MTCTL, new predicates and mappings have to be introduced. Also, the existing MTCTL transformations will have to be conceptually adapted. For example, the quantifier normalization (Section 3.2.3) assumes that all states, transitions, etc. at runtime are known and statically fixed from the beginning.

Debug View As we currently utilize Graphviz for the visualization of MECHATRONICUML traces (see Section 3.3.2), it is not (or only with an disproportional amount of work) possible to use the concrete MECHATRONICUML syntax in this visualization. Therefore, the user still has to look up the MECHATRONICUML model when interpreting the MECHATRONICUML trace. Otherwise, it is sometimes hard to find the cause of a violation, e.g., to find out why a certain transition cannot fire it is required to look up the guards of this transition.

Therefore, a Debug View integrated into the eclipse Debug View and using MECHATRONICUML’s concrete syntax would be beneficial. First concepts for this are pointed out in [40].

Backward Translation to RTCPs As explained in Section 3.3.2, UPPAAL traces are currently translated back to traces of CICs. This is possible, because the forward translation creates special CICs that express the same behavior as the original RTCP (Section 3.2.2). However, as the user wants to verify an RTCP, a very natural assumption would be to get a trace of this RTCP. In this respect, the current backward translation of traces can lead to confusion. Thereby, we assume an adaption of the backward translation to RTCPs as a valid task for future work.

8.2 Deployment

Intelligent Routing - Dynamic Routing Tables and QoS-metrics Currently, our routing-tables are generated during the code generation and stay static, no matter whether connections fail or ECUs drop out of our network. We wanted
to implement routing-tables, which change dynamically during runtime. With this, we could react to failing connections by updating the routing-tables. Unfortunately, due to time restrictions, we had to skip this feature. Additionally, a consideration of QoS-metrics for a more intelligent routing would be beneficial. For example, choosing from multiple connections in order to consider (formerly defined requirements) for QoS-Properties like delay or bandwidth. Having dynamic routing-tables is also a requirement for reconfiguration of our systems, since we route, based on ports, which change when removing/changing/adding components.

**Improve code generation** Although our code generation currently is efficient and reliable, there are still parts which can be improved. This includes:

- **Supporting additional MechatronicUML language elements**: Currently, the code generation is not able to handle all language elements of MechatronicUML. Missing elements are: do-events, multi-ports, optional ports, and selector expressions of sync-channels. There is no need to implement a new transformation template for do-events, since there is already an existing normalization for do-events used for the Verification part. This normalization could easily be reused. In contrast to that, adding support for multi/optional-ports is time-consuming, because the current implementation design of Real-Time Statechart code is not able to handle these concepts. Therefore, a restructuring for Real-Time Statechart generation is needed. When these elements are added, reconfiguration should be considered as well. In parallel to the project group, there were some changes in the original branch of the code generation. These changes contain, among others, a new implementation for sync-channels. Therefore, after merging both code generators, sync-channel will be supported completely.

- **Improvement of protobuf-message types**: Currently, we have to define a static limit of elements a protobuf message-parameter can have. This property must be once defined for all parameters of every message. Further work on this topic could be to analyze if a dynamically derived maximum size, based on our MechatronicUML message definitions for example, would be a feasible solution.

- **Broader analysis of generated code**: Although, we have tested our code generation multiple times and analyzed it with memory leak detection tools like Valgrind, further testing could still improve the quality. Especially since we did not have the opportunity to do sophisticated debugging on our LEGO mindstorms NXT bricks. This could lead to additional improvements in stability, performance, and memory consumption.

**Marshalling of Messages** This is currently done by hand in the network-interface implementations. It is only needed if the size of a middleware message is bigger
than the maximum size the used network interface supports for one message. So, this Marshalling depends on the used network interface and can currently not be considered within our code generation, since we do not have this information about the network interface saved somewhere. Also, the network interface could have different configurations, which lead to a different maximum size of bytes that can be send with one message (e.g., additional bits for CRC).

**Improve Support for Different Operating Systems** As it is, we only support code generation for x86-based platforms and nxtOSEK. Apart from supporting additional operating systems, the usability can also be improved. Currently, we have to change a string in the transformation file in order to change the target operating systems. In the future, this information can be added to one of the models we use for our code generation. For example, this could be stored in the platform description model.

**Schedulability and Analysis** After the code generation is done, it would be beneficial to do a worst case execution time (WCET) analysis. With the results of the WCET analysis, we can do a schedulability analysis, and by this guarantee that timing requirements, which were defined in the PIM, still hold for our deployed PSM.

**Evaluation of the API-Mapping** Currently, for each **Real-Time Statechart** which executes an API-Call a function is generated which implements the API-Call specified in the API-Mapping. Another approach, we discussed in our project group was the generation of function stubs for the API-Calls which have to be manually implemented by the user, such that the description of the Operating System and the API-Mapping can be skipped.

However, for our running example, we faced the problem that the left and the right motor were modeled as two separate continuous component instances in the PIM, although they are, on hardware level, controlled via a multiplexer. The API-Call for controlling the multiplexer expected the speed of the right and left motor as arguments. Thus, it was required to safe the right and left speed in a global variable while executing the function for setting the motor speed for the other motor. Currently, the API-Mapping does not support global variables which shall be adapted in the future.

**Improvements for Allocation of Software Components** The current allocation approach does not support reconfiguration. Before we can extend the allocation approach, we have to clarify whether components can be reallocated to another ECU at runtime. Depending on the answer, new constraints may arise that have to be supported by the Allocation Specification Language.

Apart from that, we should investigate different approaches how to solve the allocation problem. For example, we should consider evolutionary algorithms
8 Future Work

like in [35] and compare them to 0-1 ILP. OPT4J [41] might be an interesting framework for this.

Additionally, we have to evaluate our allocation approach with more complex examples to check if it scales. That is, examples with more components, ECUs, constraints etc. Also, we have to evaluate the Allocation Specification Language with regard to its usability and understandability. For example, it could be possible to omit the tuple descriptors in some cases, which would make the language more concise.
9 Conclusion

The development of cyber-physical systems (CPS) is a non-trivial task. Considering the overtaking scenario, challenges lie in the safety-critical and distributed nature of CPS. Hence, regular methods for software development in the context of CPS do not only tend to be tedious and error prone, but also lead to high expenses. Consequently, modern research strives to ease the development of CPS, introducing model-driven approaches like MECHATRONICUML [4].

In its original version, MECHATRONICUML provided a language, process, and tooling for modeling and verifying CPS including a specification of the target platform. However, the semi-transparent verification with Uppaal was restricted to rather simple models and a deployment process was missing entirely [2]. Thus, we extended MECHATRONICUML, providing not only an improved verification, but also concepts and tooling for deploying models of CPS onto real hardware. Figure 9.1 shows our extended MECHATRONICUML process.

Figure 9.1: Extended MECHATRONICUML process

For the Verify Software Step (2), we introduce MTCTL as a new way to formalize requirements of Real-Time Coordination Protocols (RTCP). MTCTLs independence from Uppaal led to a verification approach where Uppaal is fully transparent. Hence, we expect MTCTL reduces training costs for developers, releasing them from manually handling Uppaal. Furthermore, we expanded the set of MECHATRONICUML language constructs that are supported by the Verify Software Step. This enables developers to enhance confidence in more complex RTCPs, making the development of CPS with MECHATRONICUML more efficient. In addition, our extended translation from Uppaal to MECHATRONICUML provides sophisticated MECHATRONICUML counterexamples. We expect that this will decrease the effort for debugging RTCPs and, consequently, accelerates the
development process of CPS using MECHATRONICUML. Finally, we integrated all our concepts for the Verify Software Step into MECHATRONICUML.

Furthermore, we newly added the Deployment Step (3) to the MECHATRONICUML process. For this, we developed a deployment process, which defines a structured way to refine platform-independent models (PIM) for a target platform. This process consists of steps for allocation, platform-mapping, and code generation. These steps not only guide developers when deploying PIMs onto real hardware, but also ease the development of CPS by automating crucial tasks like the writing of code, which represents the behavior of the models. Moreover, this is the first deployment approach for MECHATRONICUML.

Furthermore, we implemented our concepts in form of several languages (e.g. for allocation and platform-mapping), a middleware, and a code generation for the nxtOSEK platform.

For both the Verification and Deployment Step, we evaluated our concepts on the overtaking scenario using the goal, question, metric approach (GQM). Our results enhance confidence in our concepts with respect to usability, completeness, efficiency, maintainability, and portability. Hence, we improved MECHATRONICUML and, consequently, ease the development of CPS. Nonetheless, MECHATRONICUML still provides many opportunities for future work.
A Running Example: Diagrams

In this section, we show all diagrams, which we created for our running example.

Figure A.1: Real-Time Coordination Protocol Overtaking

Figure A.2: Real-Time Coordination Protocol VehicleDetection
A Running Example: Diagrams

Figure A.3: Components
Figure A.4: Component instance configuration
A Running Example: Diagrams

Figure A.5: Real-Time Statechart OvertakerRole from the Overtaking Real-Time Coordination Protocol

Figure A.6: Real-Time Statechart OvertakeeRole from the Overtaking Real-Time Coordination Protocol
DetectorRoleBehavior

variable: INT velocity, INT distance, INT distanceLimit;

Init

Initiated

Executing

[distance<distanceLimit] initiateOvertaking()

/executeOvertaking()

executeOvertaking()

Figure A.7: Real-Time Statechart DetectorRole from the VehicleDetection Real-Time Coordination Protocol

DelegatorRoleBehavior

variable: INT velocity;

Init

InitiationReceived

Executing

initiateOvertaking()
n/executeOvertaking()

Figure A.8: Real-Time Statechart DelegatorRole from the VehicleDetection Real-Time Coordination Protocol
variable: INT velocity

OvertakerCommunicatorMain

channel: initiated, accepted, executed;

OvertakerPortBehavior

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>initiated? requestOvertaking()</td>
</tr>
<tr>
<td>Requested</td>
<td>accepted! acceptOvertaking()</td>
</tr>
<tr>
<td>Overtaking</td>
<td>executed?/finishedOvertaking()</td>
</tr>
</tbody>
</table>

DelegatorPortBehavior

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>executedOvertaking executed!</td>
</tr>
<tr>
<td>InitiationReceived</td>
<td>initiatedOvertaking initiated!</td>
</tr>
<tr>
<td>Executing</td>
<td>accepted? executeOvertaking()</td>
</tr>
</tbody>
</table>

Figure A.9: REAL-TIME STATECHART OvertakerCommunicator
Figure A.10: **Real-Time Statechart OvertakeeCommunicator**

![Real-Time Statechart OvertakeeCommunicator](image-url)
Figure A.11: **REAL-TIME STATECHART OvertakerDriveControl**
Figure A.12: **REAL-TIME STATECHART** OvertakeDriveControl

Figure A.13: Resource diagram
Figure A.14: Resource instance diagram
Figure A.15: Platform diagram
A Running Example: Diagrams

Figure A.16: Platform instance diagram (simplified)
## B GQM-Goals

This section includes our final goals. These goals were refined during the project phase. First we list the goals for our verification, then the goals for our Deployment related work. At last we list the results.

### B.1 Verification Goals

#### B.1.1 Forward-Translation

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Forward-Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Completeness</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

**Q1: Are all state events translated?**

| M1.1.1.1.1 | Number of state events contained in the examples / Number of corresponding methods in declaration |
| 1           |

**Q2: Are all messages translated?**

| M1.1.2.1    | Number of message types contained in the examples / Number of corresponding variables in declaration |
| 1           |

| M1.1.2.2    | Number of message types assigned to one buffer in the examples / Max number of different variables storable in one array expressing the buffer |
| 1           |

**Q3: Is 1:n communication translated?**

| M1.1.3.1    | Multiplicity of Multi-Role (given by user)+1 / Number of processes of templates representing top-level behavior of roles |
| 1           |

**Q4: Are urgent transitions translated?**

| M1.1.4.1    | Number of synchronization channels over which urgent and non-urgent transitions synchronize in examples / Number of synchronization channels for which there exists a urgent and a non-urgent version |
| 1           |
### B GQM-Goals

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Forward-Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Transparency</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

#### Q1: Is it necessary to interact with Uppaal?

<table>
<thead>
<tr>
<th>M1.1.2.1.1</th>
<th>Number of clicks in Uppaal</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.1.2.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Q2: Are the Coordination Protocols changed by the transformation?

<table>
<thead>
<tr>
<th>M1.1.2.2.1</th>
<th>Number of features in examples before transformation / Number of features in examples after transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.1.2.2.1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Q3: Are there Uppaal-specific errors visible to the user?

<table>
<thead>
<tr>
<th>M1.1.2.3.1</th>
<th>Number of examples that raise Uppaal-specific errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.1.2.3.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### B GQM-Goals

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Forward-Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Semantics Preservation/Correctness</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

#### Q1: Does the transformation preserve satisfied safety properties?

<table>
<thead>
<tr>
<th>M1.1.3.1.1</th>
<th>Number of safety properties that hold on example, but are not correctly verified by Uppaal</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.1.3.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Q2: Does the transformation preserve satisfied liveness properties?

<table>
<thead>
<tr>
<th>M1.1.3.2.1</th>
<th>Number of liveness properties that hold on example, but are not correctly verified by Uppaal</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.1.3.2.1</td>
<td>0</td>
</tr>
</tbody>
</table>
### B.1 Verification Goals

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Forward-Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Efficiency for Verification</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

#### Q1: Do the translated Uppaal models have an appropriate state space size?

<table>
<thead>
<tr>
<th>M1.1.4.1.1</th>
<th>Number of examples that cannot be checked via Uppaal, because of state explosion</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.1.4.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### B.1.2 Back-Translation

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Back-Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Completeness</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

#### Q1: Is all relevant information part of the MechatronicUML trace?

<table>
<thead>
<tr>
<th>M1.2.1.1.1</th>
<th>Number of features contained in the snapshot / Number of used MECHATRONICUML Coordination Protocol features</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.1.1.1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Q2: Are clock values part of the MechatronicUML trace?

<table>
<thead>
<tr>
<th>M1.2.1.2.1</th>
<th>Number of clocks contained in the snapshot / Number of declared clocks within the given MECHATRONICUML Coordination Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.1.2.1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Q3: Are data variable values part of the MechatronicUML trace?

<table>
<thead>
<tr>
<th>M1.2.1.3.1</th>
<th>Number of data variables contained in the snapshot / Number of declared variables within the given MECHATRONICUML Coordination Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.1.3.1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Q4: Does the MechatronicUML trace provide information about firing transitions?

<table>
<thead>
<tr>
<th>M1.2.1.4.1</th>
<th>Number of firing transitions that are not contained in the MECHATRONICUML trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.1.4.1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Q5: Does the MechatronicUML trace provide information about time-consuming transitions?

<table>
<thead>
<tr>
<th>M1.2.1.5.1</th>
<th>Number of time-consuming transitions not displayed in the MECHATRONICUML trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.1.5.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Q6: Are all message buffers part of the MechatronicUML trace?

<table>
<thead>
<tr>
<th>M1.2.1.6.1</th>
<th>Number of message buffers contained in the snapshot / Number of declared message buffers within the given MECHATRONICUML Coordination Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.1.6.1</td>
<td>1 (Assuming the value is 1 for the sake of this example)</td>
</tr>
</tbody>
</table>

### Q7: Are all connectors part of the MechatronicUML trace?

<table>
<thead>
<tr>
<th>M1.2.1.7.1</th>
<th>Number of connectors contained in the snapshot / Number of declared connectors within the given MECHATRONICUML Coordination Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.1.7.1</td>
<td>1 (Assuming the value is 1 for the sake of this example)</td>
</tr>
</tbody>
</table>

---

### Analyze

<table>
<thead>
<tr>
<th>For the purpose of</th>
<th>Ensuring</th>
</tr>
</thead>
<tbody>
<tr>
<td>With respect to</td>
<td>Transparency</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

### Q1: Is the naming of the elements in the MechatronicUML trace model consistent with the elements from the MechatronicUML design model?

<table>
<thead>
<tr>
<th>M1.2.2.1.1</th>
<th>Number of element names in the MECHATRONICUML trace that differ from the design model</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.2.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Q2: Does the user have to interact with the Uppaal trace file?

<table>
<thead>
<tr>
<th>M1.2.2.2.1</th>
<th>Number of mouse clicks for translating the UPPAAL trace back to the MECHATRONICUML level (after the Forward Translation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.2.2.2.1</td>
<td>0 (Assuming the value is 0 for the sake of this example)</td>
</tr>
</tbody>
</table>

---

### Analyze

<table>
<thead>
<tr>
<th>For the purpose of</th>
<th>Back-Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>With respect to</td>
<td>Semantics Preservation/Correctness</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>
### B.1 Verification Goals

<table>
<thead>
<tr>
<th>Question</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: Are all MechatronicUML snapshots mapped to their corresponding Uppaal states?</td>
<td>Number of snapshots, which are not mapped to the correct UPPAAL state</td>
</tr>
<tr>
<td>M1.2.3.1.1</td>
<td>H1.2.3.1.1</td>
</tr>
<tr>
<td>Q2: Are snapshots left out?</td>
<td>Number of snapshots / Number of Uppaal states</td>
</tr>
<tr>
<td>M1.2.3.2.1</td>
<td>H1.2.3.2.1</td>
</tr>
</tbody>
</table>

### B.1.3 Verification Property Language

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Verification Property Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Completeness</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: Are all relevant liveness/reachability/safety properties expressible with our Verification Property Language?</td>
<td>Number of properties P1–Pn expressible with our language / Number of defined relevant safety properties n</td>
</tr>
<tr>
<td>M1.3.1.1.1</td>
<td>H1.3.1.1.1</td>
</tr>
<tr>
<td>Q2: Is the deadlock property expressible with our Verification Property Language?</td>
<td>Expressible deadlock property</td>
</tr>
<tr>
<td>M1.3.1.2.1</td>
<td>H1.3.1.2.1</td>
</tr>
<tr>
<td>Q3: Are all elements of the MechatronicUML Coordination Protocol referencable?</td>
<td>Number of referencable MECHATRONICUML design model elements / Number of MECHATRONICUML design model elements</td>
</tr>
<tr>
<td>M1.3.1.3.1</td>
<td>H1.3.1.3.1</td>
</tr>
<tr>
<td>Q4: Are real-time constraints expressible?</td>
<td>Number of properties including real-time constraints expressible with our Verification Property Language / Number of properties P1–Pn including real-time constraints</td>
</tr>
<tr>
<td>M1.3.1.4.1</td>
<td>H1.3.1.4.1</td>
</tr>
</tbody>
</table>
B GQM-Goals

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Verification Property Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Transparency</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

Q1: Does the user have to know Uppaal?

<table>
<thead>
<tr>
<th>M1.3.2.1.1</th>
<th>Number of properties P1–Pn that are expressible with Uppaal, but not with our Verification Property Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.3.2.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q2: Does the user have to add elements to the Coordination Protocol because Uppaal cannot handle the specific query directly?

<table>
<thead>
<tr>
<th>M1.3.2.2.1</th>
<th>Number of elements the user has to add to the Coordination Protocol to express our verification properties P1–Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.3.2.2.1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Verification Property Language Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Semantics Preservation/Correctness</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Verification Process</td>
</tr>
</tbody>
</table>

Q1: Are all verification properties correctly translated into Uppaal’s TCTL?

<table>
<thead>
<tr>
<th>M1.3.3.1.1</th>
<th>Number of properties P1–Pn not correctly translated into Uppaal</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1.3.3.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

B.2 Deployment Goals

B.2.1 Definition of a Deployment Process

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Deployment Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Completeness of the process</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Allocation Engineer, Deployment Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Method</td>
</tr>
</tbody>
</table>
## B.2 Deployment Goals

### Q1: Does the output of each step match as input for the next step?

<table>
<thead>
<tr>
<th>M2.1.1.1.1</th>
<th>Not matching interfaces of the in/out specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.1.1.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Q2: Is every step defined?

<table>
<thead>
<tr>
<th>M2.1.1.2.1</th>
<th>Number of steps which do not have a defined input and output</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.1.1.2.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Analyze

<table>
<thead>
<tr>
<th>For the purpose of</th>
<th>Deployment Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>With respect to</td>
<td>Ensuring</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Maintainability of the process</td>
</tr>
<tr>
<td>In the context of</td>
<td>Allocation Engineer, Deployment Engineer, MechatronicUML Engineer</td>
</tr>
</tbody>
</table>

### Q1: Can further steps be added?

<table>
<thead>
<tr>
<th>M2.1.2.1.1</th>
<th>Number of interfaces and artifacts which have to be changed, after a step is exchanged</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.1.2.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Q2: Is the process well documented?

<table>
<thead>
<tr>
<th>M2.1.2.2.1</th>
<th>Number of missing chapters for the Deployment documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.1.2.2.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Q3: Has every step a clearly defined input/output?

<table>
<thead>
<tr>
<th>M2.1.2.3.1</th>
<th>Number of steps which do not have a defined input and output</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.1.2.3.1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Analyze

<table>
<thead>
<tr>
<th>For the purpose of</th>
<th>Deployment Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>With respect to</td>
<td>Appropriateness of the process</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Allocation Engineer, Deployment Engineer, MechatronicUML Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MechatronicUML Method</td>
</tr>
</tbody>
</table>

### Q1: Does every step lead to the expected result?

<table>
<thead>
<tr>
<th>M2.1.3.1.1</th>
<th>Number of artifacts that do not match the expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.1.3.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>
Q2: Has every step a purpose?
M2.1.3.2.1 Number of steps that can be skipped
H2.1.3.2.1 0

Q3: Is the process deterministic?
M2.1.3.3.1 Number of mutually exclusive threads to the process
H2.1.3.3.1 0

Q4: Are all steps on the same abstraction level?
M2.1.3.4.1 Steps with different level of abstraction
H2.1.3.4.1 0

### B.2.2 Platform-Mapping and PSM Transformation

<table>
<thead>
<tr>
<th>Analyze</th>
<th>PSM Transformation and Platform-Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Semantic preservation of the transformation of continuous components/hybrid ports to discrete ports/discrete components</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Allocation Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>PSM Transformation</td>
</tr>
</tbody>
</table>

Q1: Does the transformation create one discrete atomic component instance for every passive continuous component instance?
M2.2.1.1.1 Number of discrete components + number of continuous component / number of total components after transformation
H2.2.1.1.1 1

Q2: Are the former connectors that connected a passive CCI and a SCI, correctly rearranged between the SCI and the new created discrete atomic component instance?
M2.2.1.2.1 Number of connector before transformation - number of connectors after transformation
H2.2.1.2.1 0

Q3: Are correct constrains for the Allocation Algorithm generated?
M2.2.1.3.1 Number of generated constraints per passive CCI
H2.2.1.3.1 1
### B.2 Deployment Goals

#### B.2.2 Deployment Goals

<table>
<thead>
<tr>
<th>Analyze</th>
<th>PSM Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Usability of the process step</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Allocation Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Deployment Process</td>
</tr>
</tbody>
</table>

**Q1: Is the transformation transparent?**

<table>
<thead>
<tr>
<th>M2.2.2.1.1</th>
<th>Number of clicks/keystrokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.2.2.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Q2: Is much manual user interaction required for allocation?**

<table>
<thead>
<tr>
<th>M2.2.2.2.1</th>
<th>Number of clicks/keystrokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.2.2.2.1</td>
<td>( n ) per continuous component</td>
</tr>
</tbody>
</table>

### B.2.3 Allocation

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Software Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Transparency of the Allocation Algorithm</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Allocation Engineer</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Deployment Process</td>
</tr>
</tbody>
</table>

**Q1: Is there GUI support for the allocation?**

<table>
<thead>
<tr>
<th>M2.3.1.1.1</th>
<th>Number command line interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.3.1.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Q2: Is user-interaction with the ILP-Solver required?**

<table>
<thead>
<tr>
<th>M2.3.1.2.1</th>
<th>Number of clicks/keystrokes within the ILP-Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.3.1.2.1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Q3: Is the result visualized in an appropriate way?**

<table>
<thead>
<tr>
<th>M2.3.1.3.1</th>
<th>Conformity with the MECHATRONICUML syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.3.1.3.1</td>
<td>100%</td>
</tr>
<tr>
<td>M2.3.1.3.2</td>
<td>Needed number of model files to represent the result</td>
</tr>
<tr>
<td>H2.3.1.3.2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Q4: Significant delay for computing an Allocation?**

<table>
<thead>
<tr>
<th>M2.3.1.4.1</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.3.1.4.1</td>
<td>low</td>
</tr>
</tbody>
</table>

129
### Analyze Allocation
#### For the purpose of Ensuring
#### With respect to Functionality of the Allocation
#### From the viewpoint of Allocation Engineer
#### In the context of MECHATRONICUML Deployment Process

<table>
<thead>
<tr>
<th>Q1: Is it possible to express all needed constraints in the DSL?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.3.2.1.1 Number of expressible constraints / number of constraints</td>
</tr>
<tr>
<td>H2.3.2.1.1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q2: Is the output a valid allocation? (Are all constrains fulfilled?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.3.2.2.1 Number of fulfilled constraints / number of constraints</td>
</tr>
<tr>
<td>H2.3.2.2.1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q3: Is it possible to express different optimization goals?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.3.2.3.1 Number of expressible goals / number of goals</td>
</tr>
<tr>
<td>H2.3.2.3.1 1</td>
</tr>
</tbody>
</table>

### Analyze Allocation
#### For the purpose of Ensuring
#### With respect to Maintainability of Process Step
#### From the viewpoint of MechatronicUML developer
#### In the context of MECHATRONICUML Deployment Process

<table>
<thead>
<tr>
<th>Q1: Is the code documented?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.3.3.1.1 Degree of commented code</td>
</tr>
<tr>
<td>H2.3.3.1.1 high</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q2: Can the allocation step be executed via different ILP solvers?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.3.3.2.1 Number of classes and transformations, which have to be changed when using a different ILP solver</td>
</tr>
<tr>
<td>H2.3.3.2.1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q3: Effort for Installation of ILP-Solver?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.3.3.3.1 Number of clicks and keystrokes</td>
</tr>
<tr>
<td>H2.3.3.3.1 low</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q4: Is the ILP-Solver OS independent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2.3.3.4.1 Number of supported OS</td>
</tr>
<tr>
<td>H2.3.3.4.1 2 or more</td>
</tr>
</tbody>
</table>
B.3 Results

B.2.4 Code Generation

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>Ensuring</td>
</tr>
<tr>
<td>With respect to</td>
<td>Efficiency of the Code Generation</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>Target Platform Expert</td>
</tr>
<tr>
<td>In the context of</td>
<td>MECHATRONICUML Deployment Process</td>
</tr>
</tbody>
</table>

Q1: Is the size of generated and compiled Code appropriate for the target platform?

<table>
<thead>
<tr>
<th>M2.4.1.1.1</th>
<th>Size (in Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.4.1.1.1</td>
<td>below 224 Kbytes</td>
</tr>
</tbody>
</table>

Q1: Does the Code compile for the target platform?

<table>
<thead>
<tr>
<th>M2.4.2.1.1</th>
<th>Number of failed compiles for valid input models</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.4.2.1.1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q2: Does the middleware work?

<table>
<thead>
<tr>
<th>M2.4.2.2.1</th>
<th>Number of lost messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.4.2.2.1</td>
<td>0</td>
</tr>
<tr>
<td>M2.4.2.2.2</td>
<td>Number of messages that are put into message buffers out of order</td>
</tr>
<tr>
<td>H2.4.2.2.2</td>
<td>0</td>
</tr>
</tbody>
</table>

Q3: Does the generated Code corresponds to the input?

<table>
<thead>
<tr>
<th>M2.4.2.3.1</th>
<th>Number of models not represented in code</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.4.2.3.1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q4: Is the Code Generation and gen. Code fully tested?

<table>
<thead>
<tr>
<th>M2.4.2.4.1</th>
<th>Degree of code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2.4.2.4.1</td>
<td>100 percent</td>
</tr>
</tbody>
</table>

B.3 Results

In this section we see the actual measured metrics.
<table>
<thead>
<tr>
<th>Goal/Metric</th>
<th>Hypothesis</th>
<th>Status</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward-Translation - Completeness (1.1.1)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.1.1.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.1.1.2.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.1.1.3.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.1.1.4.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td><strong>Forward-Translation - Transparency (1.1.2)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.2.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1.1.2.2.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.1.2.3.1</td>
<td>0</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td><strong>Forward-Translation - Correctness (1.1.3)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.3.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1.1.3.2.1</td>
<td>0</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td><strong>Forward-Translation - Efficiency (1.1.4)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1.4.1.1</td>
<td>0</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td><strong>Back-Translation - Completeness (1.2.1)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.1.1.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.2.1.2.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.2.1.3.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.2.1.4.1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1.2.1.5.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1.2.1.6.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.2.1.7.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td><strong>Back-Translation - Transparency (1.2.2)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.2.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1.2.2.2.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1.2.3.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td><strong>Back-Translation - Correctness (1.2.3)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.3.2.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td><strong>Verification Property Language - Completeness (1.3.1)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.1.1.1</td>
<td>1</td>
<td>0,8</td>
<td>80</td>
</tr>
<tr>
<td>1.3.1.2.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.3.1.3.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1.3.1.4.1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Verification Property Language - Transparency (1.3.2)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.2.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1.3.2.2.1</td>
<td>0</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td><strong>Verification Property Language - Correctness (1.3.3)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.3.1.1</td>
<td>0</td>
<td>7</td>
<td>0,78125</td>
</tr>
</tbody>
</table>

Figure B.1: Detailed overview of metrics for the verification part
<table>
<thead>
<tr>
<th>Goal/Metric</th>
<th>Hypothesis</th>
<th>Status</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment Process - Completeness (2.1.1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.1.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.1.1.2.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Deployment Process - Maintainability (2.1.2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.2.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.1.2.2.1</td>
<td>0</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>2.1.2.3.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Deployment Process - Appropriateness (2.1.3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1.3.1.1</td>
<td>0</td>
<td>2</td>
<td>80</td>
</tr>
<tr>
<td>2.1.3.2.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.1.3.3.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.1.3.4.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Platform Mapping - semantic preservation (2.2.1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.1.1.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2.2.1.2.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.2.1.3.1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Platform Mapping - Usability (2.2.2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2.2.1.1</td>
<td>2</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>2.2.2.2.1</td>
<td>n</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Software Allocation - Transparency (2.3.1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.1.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.3.1.2.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.3.1.3.1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2.3.1.3.2</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2.3.1.4.1</td>
<td>low</td>
<td>low</td>
<td>100</td>
</tr>
<tr>
<td>Software Allocation - Functionality (2.3.2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.2.1.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2.3.2.2.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2.3.2.3.1</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>Software Allocation - Maintainability (2.3.3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3.3.1.1</td>
<td>high</td>
<td>low-mid</td>
<td>40</td>
</tr>
<tr>
<td>2.3.3.2.1</td>
<td>0</td>
<td>1</td>
<td>90</td>
</tr>
<tr>
<td>2.3.3.3.1</td>
<td>low</td>
<td>low</td>
<td>100</td>
</tr>
<tr>
<td>2.3.3.4.1</td>
<td>2</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Code Generation - Efficiency (2.4.1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4.1.1.1</td>
<td>224</td>
<td>59</td>
<td>100</td>
</tr>
<tr>
<td>Code Generation - Correctness (2.4.2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4.2.1.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.4.2.2.1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.4.2.2.2</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2.4.2.3.1</td>
<td>0</td>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td>2.4.2.4.1</td>
<td>1</td>
<td>0.5</td>
<td>50</td>
</tr>
</tbody>
</table>

Figure B.2: Detailed overview of metrics for the Deployment part
### Verification

| 1.1 | Forward-Translation | 86.363 |
| 1.2 | Back-Translation    | 90.909 |
| 1.3 | Verification Property Language | 57.969 |

### Deployment

| 2.1 | Definition of a Deployment Process | 92.222 |
| 2.2 | Platform Mapping                  | 80.000 |
| 2.3 | Software Allocation               | 94.167 |
| 2.4 | Code Generation                   | 90.000 |

**All goals:** 86.1

Figure B.3: Overall compliance of measured values with our hypotheses
C MTCTL Syntax

This part of the appendix lists syntax of MTCTL, described by the concrete syntax of Xtext\(^1\), which is similar to EBNF.

**PropertyRepository** returns `mtctl::PropertyRepository`:

```
    (properties+=Property)*
```

**Property** returns `mtctl::Property hidden(WS, ML_COMMENT)`:

```
    expression=Expression ';' (comment=SL_COMMENT)?
```

**Expression** returns `mtctl::Expression`:

```
    LeadsToExpr
```

//Binary operators (increasing precedence)
**LeadsToExpr** returns `mtctl::Expression`:

```
    ImplyExpr ({mtctl::LeadsToExpr.leftOpd=current} 'leadsTo' rightOpd=
    ImplyExpr)*
```

**ImplyExpr** returns `mtctl::Expression`:

```
    AndExpr ({mtctl::ImplyExpr.leftOpd=current} 'implies' rightOpd=
    AndExpr)*
```

**AndExpr** returns `mtctl::Expression`:

```
    OrExpr ({mtctl::AndExpr.leftOpd=current} 'and' rightOpd=OrExpr)*
```

**OrExpr** returns `mtctl::Expression`:

```
    NotExpr ({mtctl::OrExpr.leftOpd=current} 'or' rightOpd=NotExpr)*
```

//Unary operators
**NotExpr** returns `mtctl::Expression`:

```
```

\(^1\)https://www.eclipse.org/Xtext/

135
**C MTCTL Syntax**

`'not' {mtctl::NotExpr} opd=NotExpr | QuantifierExpr

QuantifierExpr **returns** mtctl::Expression:

UniversalQuantExpr | ExistentialQuantExpr | TemporalQuantifierExpr | AtomExpr

UniversalQuantExpr **returns** mtctl::QuantifierExpr:

`'forall' '(' {mtctl::UniversalQuantExpr} var=VariableBinding ')' formula = (NotExpr)

ExistentialQuantExpr **returns** mtctl::QuantifierExpr:

`'exists' '(' {mtctl::ExistenceQuantExpr} var=VariableBinding ')' formula = (NotExpr)

VariableBinding **returns** mtctl::BoundVariable:

name=ID ' :' set=SetExpr

TemporalQuantifierExpr **returns** mtctl::Expression:

EFExpr | AFExpr | EGExpr | AGExpr

EFExpr **returns** mtctl::TemporalQuantifierExpr:

(`'EF' | 'E<>') {mtctl::EFExpr} expr=NotExpr

AFExpr **returns** mtctl::TemporalQuantifierExpr:

(`'AF' | 'A<>') {mtctl::AFExpr} expr=NotExpr

EGExpr **returns** mtctl::TemporalQuantifierExpr:

(`'EG' | 'E[]') {mtctl::EGExpr} expr=NotExpr

AGExpr **returns** mtctl::TemporalQuantifierExpr:

(`'AG' | 'A[]') {mtctl::AGExpr} expr=NotExpr

//Bottom of precedence chain.
AtomExpr **returns** mtctl::Expression:

136
'\(' Expression ')' | PredicateExpr | ComparisonExpr

//Predicates
PredicateExpr returns mtctl::Expression:
  TrueExpr | FalseExpr | DeadlockExpr | BufferOverflowExpr | StateExpr | MessageExpr | TransitionExpr

DeadlockExpr returns mtctl::DeadlockExpr:
  \{mtctl::DeadlockExpr\} 'deadlock'

BufferOverflowExpr returns mtctl::BufferOverflowExpr:
  \{mtctl::BufferOverflowExpr\} 'bufferOverflow'

TrueExpr returns mtctl::TrueExpr:
  \{mtctl::TrueExpr\} 'true'

FalseExpr returns mtctl::FalseExpr:
  \{mtctl::FalseExpr\} 'false'

StateExpr returns mtctl::PredicateExpr:
  StateActiveExpr | SubstateOfExpr | StateInStatechartExpr

StateActiveExpr returns mtctl::StateActiveExpr:
  'stateActive' '(', state=StateMapExpr ')'

SubstateOfExpr returns mtctl::SubstateOfExpr:
  'substateOf' '(', state=StateMapExpr ',', superstate=StateMapExpr ')''

StateInStatechartExpr returns mtctl::StateInStatechartExpr:
  'stateInStatechart' '(', state = StateMapExpr ',', statechart = StatechartMapExpr ')''

MessageExpr returns mtctl::PredicateExpr:
  MessageInBufferExpr | MessageInTransitExpr
C MTCTL Syntax

MessageInTransitExpr returns mtctl::MessageInTransitExpr:
    'messageInTransit' '(' message=MessageMapExpr ')'

MessageInBufferExpr returns mtctl::MessageInBufferExpr:
    'messageInBuffer' '(' message=MessageMapExpr ',' buffer=
        BufferMapExpr ')'

TransitionExpr returns mtctl::PredicateExpr:
    TransitionFiringExpr

TransitionFiringExpr returns mtctl::TransitionFiringExpr:
    'transitionFiring' '(' transition=TransitionMapExpr ')'

//Comparisons
ComparisonExpr returns mtctl::Expression:
    {mtctl::ComparisonExpr} lhs=MapExpr op=ComparisonOp rhs=
        MapExpr

enum ComparisonOp returns mtctl::ComparisonOp:
    EQUALS='==' | GREATER='>' | GREATER_OR_EQUAL='>=' | LESS='<' | LESS_OR_EQUAL='<=' | NOT_EQUAL='!='

//Expressions usable in comparisons. Starting with MapExpressions arranged by
//return type
MapExpr returns mtctl::MapExpr:
    MumlElemExpr | BufferMessageCountExpr | ConstExpr | SourceStateExpr | TargetStateExpr

IntegerMapExpr returns mtctl::MapExpr:
    MumlElemExpr | BufferMessageCountExpr | ConstExpr

TransitionMapExpr returns mtctl::MapExpr:
    MumlElemExpr
StateMapExpr returns mtctl::MapExpr:
    MumlElemExpr | SourceStateExpr | TargetStateExpr
;

StatechartMapExpr returns mtctl::MapExpr:
    MumlElemExpr
;

BufferMapExpr returns mtctl::MapExpr:
    MumlElemExpr
;

MessageMapExpr returns mtctl::MapExpr:
    MumlElemExpr
;

BufferMessageCountExpr returns mtctl::BufferMsgCountExpr:
    'bufferMessageCount' '(' buffer=BufferMapExpr ')' 
;

SourceStateExpr returns mtctl::SourceStateExpr:
    'sourceState' '(' transition=TransitionMapExpr ')' 
;

TargetStateExpr returns mtctl::TargetStateExpr:
    'targetState' '(' transition=TransitionMapExpr ')' 
;

MumlElemExpr returns mtctl::MumlElemExpr:
    elem=[ecore::EObject|QualifiedName] '[' instance=[ecore::EObject|QualifiedName] ']'?
;

ConstExpr returns mtctl::ConstExpr:
    val=EInt (timeUnit=TimeUnitExpr)?
;

TimeUnitExpr returns valuetype::TimeUnit:
    'days' | 'hrs' | 'mins' | 'secs' | 'msecs' | 'µsecs' | 'nsecs'
;

//Sets
SetExpr returns mtctl::SetExpr:
InstanceSetExpr | SubinstanceSetExpr | IntervalSetExpr | \{mtctl::StateSetExpr\} 'States' | \{mtctl::TransitionSetExpr\} 'Transitions' | \{mtctl::MessageSetExpr\} 'MessageTypes' | \{mtctl::ClockSetExpr\} 'Clocks' | \{mtctl::BufferSetExpr\} 'Buffers'

IntervalSetExpr \textbf{returns} mtctl::SetExpr:
\{mtctl::IntervalSetExpr\} \texttt{[} lowerVal=INT \texttt{,} upperVal=INT \texttt{]} \texttt{]}

InstanceSetExpr \textbf{returns} mtctl::SetExpr:
\{mtctl::InstanceSetExpr\} 'Instances' \texttt{<} type=MumlElemExpr \texttt{>}'

SubinstanceSetExpr \textbf{returns} mtctl::SetExpr:
\{mtctl::SubinstanceSetExpr\} 'Subinstances' \texttt{<} type=MumlElemExpr \texttt{>}'

//Other necessary definitions
QualifiedName:
    ID (\texttt{.'} ID)*
D Allocation Specification Language Syntax

This part of the appendix lists the syntax of the Allocation Specification Language, described by the concrete syntax of Xtext\(^1\), which is similar to EBNF.

\[\text{import } "http://www.fujaba.de/muml/allocation/language/0.1.0/cs" \text{ as cs}\]

\[\text{import } "platform:/resource/de.uni_paderborn.fujaba.muml.allocation.language/model/LanguageSpecificationCS.ecore#/cs" \text{ as cs}\]

Specification \textbf{returns} cs::SpecificationCS:
\[
\{\text{cs::SpecificationCS}\}
\begin{align*}
\text{name=ID '}' \\
(\text{ownedImport += ImportCS | ownedInclude += IncludeCS} \\
\text{ | contexts += ClassifierContextDeclCS | services += Service |} \\
\text{ constraints += Constraint})^* \\
(\text{goal=Goal measure=MeasureFunction})? \\
'\}'
\end{align*}
\]

Service \textbf{returns} cs::ServiceCS:
\[
'\text{service' name=ID '}{'}
\begin{align*}
\text{dimensions += QosDimension*} \\
'\}'
\end{align*}
\]

QosDimension \textbf{returns} cs::QoSDimensionCS:
\[
'\text{qos' name=ID '}{'}
\begin{align*}
'\text{value' weighting=ValueTupleDescriptor'}';' \\
'\text{descriptors' tupleDescriptors+=} \\
\text{ComponentResourceTupleDescriptor (',') tupleDescriptors+=}
\end{align*}
\]

\(^1\)https://www.eclipse.org/Xtext/
ComponentResourceTupleDescriptor)*';'
    'ocl' expression=Model ';' 
};

Constraint returns cs::ConstraintCS:
    'constraint' (LocationConstraint | ResourceConstraint | RequiredHardwareResourceInstanceConstraint)
    ;

LocationConstraint returns cs::LocationConstraintCS:
    type=LocationConstraintType (name=ID)? '{
    'descriptors' tupleDescriptor=LocationTupleDescriptor ';'
    'ocl' expression=Model ';' 
    '}
    ;

ResourceConstraint returns cs::ResourceConstraintCS:
    'resource' (name=ID)? '{
    'lhs' weighting=ValueTupleDescriptor';'
    'rhs' rhs=ValueTupleDescriptor';'
    'descriptors' tupleDescriptors+=
    ComponentResourceTupleDescriptor (',', tupleDescriptors+=
    ComponentResourceTupleDescriptor)*';'
    'ocl' expression=Model ';' 
    '}
    ;

RequiredHardwareResourceInstanceConstraint returns cs::
    RequiredHardwareResourceInstanceConstraintCS:
    'requiredHardwareResourceInstance' (name=ID)? '{
    'descriptors' tupleDescriptors+=
    ComponentResourceTupleDescriptor (',', tupleDescriptors+=
    ComponentResourceTupleDescriptor)*';'
    'ocl' expression=Model ';' 
    '}
    ;

enum LocationConstraintType returns cs::LocationConstraintTypes:
    SAME_LOCATION='sameLocation' | DIFFERENT_LOCATION='
    differentLocation'
    ;

LocationTupleDescriptor returns cs::LocationTupleDescriptorCS:
ValueTupleDescriptor returns cs::ValueTupleDescriptorCS:
    value=ID
;
ComponentResourceTupleDescriptor returns cs::ComponentResourceTupleDescriptorCS:
    '(' instance=ID ',' hwresinstance=ID ')' 
;
MeasureFunction returns cs::MeasureFunctionCS:
    'measure' services+=[cs::ServiceCS] ('+' services+=[cs::ServiceCS])*';'
;
enum Goal returns cs::Goal:
    MIN='min' | MAX='max'
;
E Implemented Accomplishments

This chapter is intended to list things that we spent a lot of time on, but are not represented in the concepts above (e.g., because they are implementation details).

E.1 Verification

- Revamped the complete translation chain to be based on verifying CICs (ready to semi-easily add verification for CICs, AtomicComponentInstances, etc.)
- Revamped the complete tooling for the verification process, adding a wizard, e.g., for choosing a set of properties to verify. (With MTCTL syntax highlighting!)
- Added verification options (based on APE)
- Formatted UPPAAL-Code (indentation and everything) by migrating the M2T to Xtend
- Layouting UPPAAL NTA
- Automatic generation of MUML2MUML QVTo library

E.2 Deployment

- Making the WiFi Block work on nxtOSEK
- Making the RS485 communication work on nxtOSEK, based on SDLC Protocol (CRC, Framing, etc.)
- Allocation specification language build workflow (the whole integrating OCL, scoping, typing, ... into the language)
- Allocation language test cases, because of several Xtext-OCL bugs
- Integrating Doxygen Comments in the code generation
- Sane re-implementation of the API-Mapping

E.3 Running Example

- It should be noted how much work it was to create this running example (multiple versions), debug it, discover flaws in MECHATRONICUML tooling,
get the vehicles running

E.4 Misc

- Several meta-model change discussions (constraint package, buffer names, portOrRoleBehavior/adaptationBehavior/behavior, ...)
- Creating patches for the OCL-Xtext Framework, which had serious bugs within the framework
- Implementing a QualifiedNameProvider for MECHATRONICUML, that generates good qualified names
- Fixing the QVTo compiler, which had serious performance issues, before we fixed it
Bibliography


[31] Sommer, S., Geisinger, M., Buckl, C., Bauer, G., Knoll, A.: Reconfigurable industrial process monitoring using the chromosome middleware. SIGBED Rev. 10(4) (December 2013) 61–64


Bibliography
