The MechatronicUML Design Method – Process, Syntax, and Semantics

Technical Report
tr-ri-12-326

Steffen Becker, Christian Brenner, Christopher Brink, Stefan Dziwok,
Christian Heinzemann, Uwe Pohlmann,
Wilhelm Schäfer, Julian Suck, Oliver Sudmann
Software Engineering Group, Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
[stbecker|cbr|christopher.brink|stefan.dziwok|c.heinzemann]
upohl|wilhelm|julian.suck|oliversu]@uni-paderborn.de

Renate Löffler
Fraunhofer IPT, Project Group Mechatronic Systems Design
Zukunftsmeile 1
33102 Paderborn, Germany
renate.loeffler@ipt.fraunhofer.de

Version: 0.3
Paderborn, August 17, 2012

1This work was partially developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

2This work was partially developed in the project ‘ENTIME: Entwurfstechnik Intelligente Mechatronik’ (Design Methods for Intelligent Mechatronic Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, ‘Investing in your future’.
# Contents

1. **Introduction** ................................................. 3  
   1.1. Example .................................................. 5

2. **MECHATRONIC UML Overview** ............................. 7

3. **Modeling Languages** ...................................... 13  
   3.1. Real-Time Coordination Pattern ............................ 13  
       3.1.1. Application Example ................................. 14  
       3.1.2. Role ................................................. 15  
       3.1.3. Role Connector ...................................... 17  
       3.1.4. Kinds of Real-Time Coordination Patterns ............ 18  
       3.1.5. Real-Time Coordination Instance .................... 19  
       3.1.6. Behavior Specification ............................... 21  
       3.1.7. Verification of Properties ........................... 22  
   3.2. Message Interface Specification .......................... 24  
       3.2.1. Message Interface ................................... 24  
       3.2.2. Message Type ....................................... 25  
   3.3. Real-Time Statechart .................................... 26  
       3.3.1. Clock ............................................... 28  
       3.3.2. State ............................................... 28  
       3.3.3. Region .............................................. 32  
       3.3.4. Transition .......................................... 33  
       3.3.5. Entry-/Exit-Point .................................. 37  
       3.3.6. Shallow History .................................... 38  
       3.3.7. Action .............................................. 39  
       3.3.8. Variable ........................................... 40  
       3.3.9. Operation .......................................... 40  
       3.3.10. Action Language .................................... 40  
       3.3.11. Synchronization .................................... 47  
       3.3.12. Asynchronous Message Event ......................... 52  
       3.3.13. Real-Time Statecharts of Multi-Roles and Multi-Ports 52  
   3.4. Component Model ......................................... 55  
       3.4.1. Atomic Component Type ............................... 55  
       3.4.2. Structured Component Type ............................ 64
### 3.5. Component Instance Configuration
- 3.5.1. Component Instance ........................... 74
- 3.5.2. Component Instance Configuration .............. 75

### 3.6. Deployment ........................................ 78
- 3.6.1. Hardware Nodes .................................. 78
- 3.6.2. Allocation ........................................ 78

### 4. Development Process ................................. 81
- 4.1. Process for Self-Optimizing Mechatronic Systems . 81
- 4.2. The MECHATRONIC UML Process .................... 83
  - 4.2.1. Determination of Coordination Patterns (Step 3) . 87
  - 4.2.2. Modeling of a Real-Time Coordination Pattern (Step 3.2) . 88
  - 4.2.3. Determination of the Components’ Behavior (Step 4) . 89
  - 4.2.4. Modeling of the Components’ Behavior (Step 4.3) . 92

### 5. Complete Example .................................. 95
- 5.1. Real-Time Coordination Patterns .................. 95
  - 5.1.1. Navigation Pattern ............................. 95
  - 5.1.2. Delegation Pattern ............................. 97
  - 5.1.3. Distribution Pattern ............................ 98
  - 5.1.4. PositionTransmission Pattern .................. 101
  - 5.1.5. DistancesTransmission Pattern ................. 102
- 5.2. Message Interface Specification .................. 104
  - 5.2.1. Navigation ..................................... 104
  - 5.2.2. Delegation ..................................... 104
  - 5.2.3. Distribution .................................... 105
  - 5.2.4. PositionTransmission ........................... 105
  - 5.2.5. DistancesTransmission .......................... 106
- 5.3. System Structure .................................. 106
  - 5.3.1. Structured Components ........................ 106
  - 5.3.2. Atomic Components ............................ 109
- 5.4. Real-Time Statechart .............................. 112
  - 5.4.1. Exploration .................................... 112
  - 5.4.2. Navigation ..................................... 114
  - 5.4.3. BeBot Observer .................................. 116
  - 5.4.4. Collision Control ............................... 118
- 5.5. Component Instance Configuration ................ 121
  - 5.5.1. Single BeBot ................................... 121
  - 5.5.2. Networks of BeBots ............................ 122
- 5.6. Deployment ........................................ 127
## 6. Theoretical Background

### 6.1. Compositional Verification Approach
- **6.1.1. Preliminaries**
- **6.1.2. Putting it all together**

## 7. Related Work

### 7.1. Contract-Based Design
- **7.1.1. Contract Levels**
- **7.1.2. Automata Models for Supporting the Contract Levels**
- **7.1.3. SPEEDS**

### 7.2. Behavioral Connectors
- **7.2.1. Higher-Order Architectural Connectors**
- **7.2.2. Reo**

### 7.3. Multi-Agent-Systems

### 7.4. Specification Languages for Systems Engineering
- **7.4.1. SysML**
- **7.4.2. MATLAB/Simulink Stateflow**
- **7.4.3. Modelica**
- **7.4.4. CHARON**
- **7.4.5. Masaccio/Giotto**

### 7.5. Process Models for Systems Engineering

### 7.6. Software Component Models

### 7.7. Specifications of Self-Adaptive Systems
- **7.7.1. Reference Architectures for Self-Adaptive Systems**
- **7.7.2. Modeling of Self-Adaptive Systems**
- **7.7.3. Run-Time Frameworks for Self-Adaptation**

### 7.8. Formal Models for Modeling of Real-time Behavior
- **7.8.1. Discrete time models**
- **7.8.2. Dense time models**

## 8. Conclusions and Future Work

## 9. Bibliography

### 9.1. Own Publications

### 9.2. Bachelor Theses, Master Theses and Ph.D. Theses

### 9.3. Foreign Publications

## A. Meta-model Documentation

### A.1. Package `modelinstance`
- **A.1.1. Package Overview**
- **A.1.2. Detailed Contents Documentation**
## CONTENTS

<table>
<thead>
<tr>
<th>A.2. Package</th>
<th>muml</th>
<th>187</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2.1. Package Overview</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>A.3. Package</td>
<td>muml::model</td>
<td>187</td>
</tr>
<tr>
<td>A.3.1. Package Overview</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>A.4. Package</td>
<td>muml::model::component</td>
<td>187</td>
</tr>
<tr>
<td>A.4.1. Package Overview</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>A.4.2. Detailed Contents Documentation</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>A.5. Package</td>
<td>muml::model::constraint</td>
<td>210</td>
</tr>
<tr>
<td>A.5.1. Package Overview</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>A.5.2. Detailed Contents Documentation</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>A.6. Package</td>
<td>muml::model::core</td>
<td>213</td>
</tr>
<tr>
<td>A.6.1. Package Overview</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>A.6.2. Detailed Contents Documentation</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>A.7. Package</td>
<td>muml::model::deployment</td>
<td>220</td>
</tr>
<tr>
<td>A.7.1. Package Overview</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>A.7.2. Detailed Contents Documentation</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>A.8. Package</td>
<td>muml::model::instance</td>
<td>225</td>
</tr>
<tr>
<td>A.8.1. Package Overview</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>A.8.2. Detailed Contents Documentation</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>A.9. Package</td>
<td>muml::model::msgiface</td>
<td>234</td>
</tr>
<tr>
<td>A.9.1. Package Overview</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>A.9.2. Detailed Contents Documentation</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>A.10. Package</td>
<td>muml::model::pattern</td>
<td>237</td>
</tr>
<tr>
<td>A.10.1. Package Overview</td>
<td>237</td>
<td></td>
</tr>
<tr>
<td>A.10.2. Detailed Contents Documentation</td>
<td>237</td>
<td></td>
</tr>
<tr>
<td>A.11. Package</td>
<td>muml::model::realtimestatechart</td>
<td>242</td>
</tr>
<tr>
<td>A.11.1. Package Overview</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>A.11.2. Detailed Contents Documentation</td>
<td>242</td>
<td></td>
</tr>
<tr>
<td>A.12. Package</td>
<td>actionLanguage</td>
<td>262</td>
</tr>
<tr>
<td>A.12.1. Package Overview</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>A.12.2. Detailed Contents Documentation</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>A.13. Package</td>
<td>core</td>
<td>267</td>
</tr>
<tr>
<td>A.13.1. Package Overview</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>A.13.2. Detailed Contents Documentation</td>
<td>267</td>
<td></td>
</tr>
<tr>
<td>A.14. Package</td>
<td>core::expressions</td>
<td>270</td>
</tr>
<tr>
<td>A.14.1. Package Overview</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>A.14.2. Detailed Contents Documentation</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>A.15. Package</td>
<td>core::expressions::common</td>
<td>272</td>
</tr>
<tr>
<td>A.15.1. Package Overview</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>A.15.2. Detailed Contents Documentation</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>A.16. Package</td>
<td>storydiagrams</td>
<td>277</td>
</tr>
<tr>
<td>A.16.1. Package Overview</td>
<td>277</td>
<td></td>
</tr>
</tbody>
</table>
A.16.2. Detailed Contents Documentation ....... 277
A.17. Package storydiagrams::activities .......... 278
   A.17.1. Package Overview .................. 278
   A.17.2. Detailed Contents Documentation ........ 278
A.18. Package storydiagrams::activities::expressions .... 287
   A.18.1. Package Overview .................. 287
   A.18.2. Detailed Contents Documentation ........ 287
A.19. Package storydiagrams::calls ............... 289
   A.19.1. Package Overview .................. 289
   A.19.2. Detailed Contents Documentation ........ 289
A.20. Package storydiagrams::calls::expressions ... 293
   A.20.1. Package Overview .................. 293
   A.20.2. Detailed Contents Documentation ........ 293
A.21. Package storydiagrams::patterns ............ 296
   A.21.1. Package Overview .................. 296
   A.21.2. Detailed Contents Documentation ........ 296
A.22. Package storydiagrams::patterns::expressions ... 306
   A.22.1. Package Overview .................. 306
   A.22.2. Detailed Contents Documentation ........ 306
A.23. Package storydiagrams::templates ........... 309
   A.23.1. Package Overview .................. 309
   A.23.2. Detailed Contents Documentation ........ 309

B. Action Language XText Grammar ............... 311
Changelog

Version 0.3

Version 0.3 includes the following major changes compared to the preceding version 0.2.

General

- Added Changelog

Overview

- Change of process overview in Section 2: If the formal use case specification is changed (Step 5b), instead of repeating only Steps 3, 4a, and 5a, the whole process is repeated.

Modeling Languages

- Real-Time Coordination Pattern: Multi-roles have a property ordered which implies an order of the sub-role instances at run-time (cf. Section 3.1.2.2)
- Message Interfaces: Made concrete syntax of parameter definition consistent to the concrete syntax of parameter definitions in Real-Time Statechart operations
- Real-Time Statechart: Changed concrete syntax for trigger messages at a transition: Parameters and their types are no longer displayed.
- Added new Section 3.3.10 about Action Language for Real-Time Statechart.
- Revised and extended concept for specifying behaviors of multi-roles, added operations for exploiting the order of an ordered multi-role (cf. Section 3.3.13)
- Component parts of structured components have a name (cf. Section 3.4.2.3).
- Component instances are typed over a component type and a component part (cf. Section 3.5.1).
- Changed specification for deployment.
- Adapted new concrete syntax to all figures
Development Process

- Adapted MECHATRONICUML process in Section 4 to use Modal Sequence Diagrams as a formal basis for the application scenarios. Affected processes: integration into development process for advanced mechatronic systems, overall MECHATRONICUML process, determine coordination pattern, model new coordination pattern.

- Distinguished between the principle solution and the system model in Section 4

- Simplified the subprocess model new coordination pattern in Section 4.2.2

Complete Example

- Adapted new concrete syntax to all figures

- Removed errors in the Real-Time Statecharts

- Changed deployment example

Theoretical Background

- Updated description of the compositional verification approach

Related Work

- Added related work to Contract-Based Design

- Added related work to Behavioral Connectors Multi-Agent-Systems

- Extended related work to Specification Languages for Systems Engineering

- Extended related work to Software Component Models

- Extended related work to Specifications of Self-Adaptive Systems

- Extended related work to Formal Models for Modeling of Real-time Behavior

Meta-model Documentation

- Replaced use of EAttribute, EOperation, EParameter by own meta-classes for attributes, operations, and parameters

- Added type model for defining simple and array data types

- Added action language meta-model
Action Language XText Grammar

- Added this new appendix for the grammar of our action language

Version 0.2

Version 0.2 includes the following major changes compared to the preceding version 0.1.

General

- Changed title from "MechatronicUML – Syntax and Semantics" to "The MechatronicUML Design Method – Process Syntax and Semantics".

Overview

- Added Chapter MECHATRONICUML Overview

Modeling Languages

- Rewrote section Real-Time Coordination Patterns
- Improved images and explanations in section Real-Time Statechart
- Improved description for Component Model
- Extracted Hardware Components from section Component Instance Configuration and defined an own section for them

Development Process

- Defined an own chapter for the development process

Complete Example

- Corrected several errors

Theoretical Background

- Added chapter for describing the theoretical background
Related Work

- Extended related work to Specification Languages for Systems Engineering
- Extended related work to Software Component Models
- Extended related work to Specifications of Self-Adaptive Systems

Meta-model Documentation

- Improved package muml::model::component
- Added Package muml::model::deployment
- Deleted class Infinity from package muml::model::core
- Improved package muml::model::instance
- Improved package muml::model::pattern
- Improved package muml::model::realtimestatechart
Chapter 1.

Introduction

Innovation in today’s technical systems is largely driven by embedded software. For example, it has been estimated that the current generation of upper class cars will contain about one gigabyte of software [PBKS07]. Technical systems pose a challenge for software development as they are often employed in a safety-critical context and they operate under tight resource constraints.

The trend of software integration accelerates as more and more embedded devices are not working in isolation but heavily interact and coordinate with other parts of the technical system. This requires discrete state-based software in addition to the previously used continuous controllers [Kil05] for controlling the dynamic behavior of the physical part of the system. This leads to complex hybrid embedded software.

This is even more the case in systems of systems. There, autonomous systems coordinate and communicate in an ad-hoc fashion [SW07]. In this case, the network topology is not fixed at design time but rather adapts itself at run time.

Finally, the integration of self-X behavior [CdLG+09], like self-adaptation, self-optimization, self-organizing, and self-healing, is another trend in innovative systems. Again, software plays an important part in realizing this behavior.

All these trends lead to complex technical systems whose structure and behavior cannot be fully determined a priori. The key issue for the successful development of such systems is handling the inherent complexity. Therefore appropriate development methods and languages as well as supporting tools are required.

The key principles for handling the complexity are abstraction and reuse. Model-driven development approaches enable to abstract from technical implementation details and, thus, allow analyses of the quality of the system, e.g., concerning the safety and the availability of the system. Second, recurring solutions should not be redeveloped in an ad-hoc manner. Instead they have to be stored as reusable development artifacts.

MECHATRONICUML is a modeling language which uses concepts of the UML [Obj09] which specifically targets the software embedded in technical systems and addresses the aforementioned characteristics like self-X. Development of MECHATRONICUML has started at the Software Engineering Group at the University of Paderborn in 2001. MECHATRONICUML supports the development of structural as well as behavioral aspects of mechatronic software. It follows the component-based approach [Szy98] for software development. Specifically, it
CHAPTER 1. INTRODUCTION

distinguishes component types as well as their instances\(^1\). The behavior of components is specified using Real-Time Statecharts, which are a combination of UML state machines and Timed Automata.

A major aim of MECHATRONIC UML is the formal verification of safety critical properties of mechatronic systems which often operate in safety-critical contexts. A single, isolated mechatronic system can be formally verified with respect to safety properties by classical techniques in reasonable time. This is, unfortunately, not the case for modern mechatronic systems which, as mentioned before, coordinate and communicate with other systems in an ad-hoc fashion and/or integrate self-X behavior.

To counter this complexity, MECHATRONIC UML introduces reusable Real-Time Coordination Patterns which formalize the coordination between mechatronic systems. Patterns formalized in such a way enable reusing the verification results without repeatedly reverifying communication behaviors. Separating the verification of communication protocols and single component behaviors enables the compositional verification of complex systems by partitioning the system’s state space to verifiable chunks.

Although the main focus of MECHATRONIC UML is on the discrete state-based behavior of mechatronic systems, especially the coordination with other mechatronic systems, special support is provided for the integration of control software. Finally, a thoroughly defined development process links all development artifacts and development activities of MECHATRONIC UML.

This technical report extends the previous versions [BDG\(^+\)11, BBD\(^+\)12]. It consolidates the older publications [Bur02, Gie03, GB03, GST\(^+\)03, GTB05, GHH\(^+\)08, EHH\(^+\)11, SSGR11, HPB12] and theses [Hir04, Bur06, Hir08] in a single document. In [GS12], a consolidated version of MECHATRONIC UML up to 2007 is presented whose concepts are continuously merged into this technical report. We present the current version of MECHATRONIC UML in detail and give formal specifications for abstract and concrete syntax as well as an informal description of its semantics.

The report is structured as follows. In Chapter 2, we provide a brief overview of the MECHATRONIC UML method. That includes a short description of the development process as well as the modeling languages. Chapter 3 describes informally the syntax and the semantics of the modeling language used in MECHATRONIC UML based on the running example which is presented in the next section. In Chapter 4, we illustrate the development process of MECHATRONIC UML. The complete models of the running example are presented in Chapter 5. In Chapter 6, we provide more information on the theoretical foundations of the MECHATRONIC UML method. After a discussion of related approaches in Chapter 7, we conclude with an outlook on future work in Chapter 8. Appendix A contains a thorough definition of the abstract syntax.

\(^1\)In the remainder of this document, we will refer to component types simply as components for the sake of easier readability (cf. Section 3.4).
1.1. Example

In this document, we will use an environment exploration scenario as an ongoing example in which several autonomous robots have to explore an unknown environment. As robots, we will use the intelligent miniature robot *BeBot*² (see Figure 1.1). The BeBot is a test carrier for intelligent machines and cooperative networks developed at the Heinz Nixdorf Institute at the University of Paderborn³.

As shown in Figure 1.1, the BeBot uses two chain-drives with DC motors to move around. It has twelve infrared-sensors and a front camera to sense its environment. The BeBot may utilize a Bluetooth and a wireless LAN module for communicating with other BeBots. The functionality of the BeBot may be extended using the USB ports. In our example, we extend the functionality of the BeBot by a GPS-Receiver for detecting the current position of the BeBot.

![Figure 1.1.: BeBot](image)

In our scenario, several BeBots explore an unknown area as shown in Figure 1.2. For reasons of simplicity, we assume that the area is unbounded and contains no obstacles. We will enhance our example with obstacles in future version of this document in order to make the scenario a more realistic. At present, the BeBots only have the task to explore the area without colliding with each other.

The BeBot performs a step-wise movement instead of moving at a constant speed. In each step, the BeBot performs the following operations: it chooses randomly a target position within a fixed distance around it to move to. Then, the BeBot turns and moves to this position. After reaching the target position, the BeBot stops and performs another step as described before.

A BeBot may only move to its intended target position if it cannot come into collision with another BeBot while moving there. That decision requires knowledge about the positions of

³[http://wwwhni.uni-paderborn.de/en/](http://wwwhni.uni-paderborn.de/en/)
CHAPTER 1. INTRODUCTION

impending collision must be prevented!

bebot-0
bebot-1
bebot-2
communicated waypoint

Figure 1.2.: Area of the Exploration scenario

the other BeBots in the area. While a BeBot may use its GPS sensor for obtaining its current position, it cannot sense the position of the other BeBots by itself. Therefore, one of the BeBots acts as a position distributor. Each BeBot transmits regularly its own position to the position distributor. The position distributor stores the current positions of all BeBots and sends them regularly to all BeBots. That ensures that each BeBot receives regular updates of the positions of all BeBots in the area.

The BeBot uses the position data of the other BeBots to avoid collisions. In each step, the BeBot compares its calculated target position to the positions of the other BeBots and decides whether a collision may occur or not. If a collision may occur, the BeBot does not move to its target position, but remains at its current position.

In principle, the position distributor may be elected during run-time. In case of a failure in the position distributor, the position distributor may also be reelected during run-time. At present, we restrict ourselves to a preset position distributor and support no reelection at run-time. However, we plan to extend our example to capture such behavior in a future version of this document.
Chapter 2.

MECHATRONIC UML Overview

The MECHATRONIC UML method enables the model-driven design of discrete software of self-adaptive mechatronic systems. The key concepts of MECHATRONIC UML are a component-based system model which enables scalable compositional verification of safety-properties, the model-driven specification and verification of reconfiguration operations, and the integration of the discrete software with the controllers of the mechatronic system. Therefore, MECHATRONIC UML provides a set of domain specific visual languages (DSVL) as well as a defined development process.

Figure 2.1 provides an overview of the development process of MECHATRONIC UML which we will briefly illustrate in the following. A detailed description of the development process can be found in Section 4.

The starting point for the development of a system is typically a set of informal requirements in natural language. Such requirements are not formally analyzable with respect to contradictions and inconsistencies.

In Step 1, MECHATRONIC UML requires the translation of the informal requirements into formal use case specifications. The formal use case specifications are specified in terms of Modal Sequence Diagrams [HM07]. Modal Sequence Diagrams are a formalized variant of UML sequence diagrams [Obj09]. Using Modal Sequence Diagrams, the interactions between the system elements as well as the real-time constraints that need to hold for the interaction may be specified. Then, inconsistencies and contradictions may be analyzed by using a synthesis or simulation-based approach [Gre11]. A detailed description of the use of Modal Sequence Diagrams in MECHATRONIC UML will be added to future versions of this document.
In Step 2, the use case specifications are used to derive an initial set of components describing the structure of the system. A component is a software entity that encapsulates a part of the system behavior which implements a certain function. Each component defines a set of external interaction points, called ports, for accessing its functionality. In contrast to other component-based approaches [Szy98, LW07], MECHATRONIC UML employs active components that execute a behavior specification in a single thread. The component model is structured hierarchically, i.e., components are either implemented directly (atomic components) or they are decomposed into several other components (structured components). A description of the component model may be found in Section 3.4.

In advanced mechatronic systems, the components that constitute the software do not work in isolation, but they collaborate for realizing the intended system functionality. MECHATRONIC UML accounts for that by considering the communication protocols by which components interact as a first-class modeling entity. They are specified by means of Real-Time Coordination Patterns in Step 3. Real-Time Coordination Patterns define different roles for the interaction, e.g., a client and a server. We discuss Real-Time Coordination Patterns in Section 3.1.

The behavior of the roles of a Real-Time Coordination Pattern is specified by Real-Time Statecharts which are a combination of UML state machines [Obj09] and timed automata [DM01, BY03, AD94]. They specify the message exchange of the roles with respect to the real-time properties that the roles must obey. Using this behavior specification, each Real-Time Coordination Pattern is formally verified for safety and bounded liveness properties [GTB+03, BGHS04]. A detailed description of Real-Time Statecharts is given in Section 3.3.

In Step 4a, the Real-Time Coordination Patterns are used to derive the component behavior specification for the atomic components identified in Step 3. Firstly, roles of Real-Time Coordination Patterns are assigned to the ports of the components. I.e., the components implement the respective role of the Real-Time Coordination Pattern. Secondly, in case of an atomic component, the Real-Time Statechart for the component is derived from the Real-Time Statecharts of the roles. That Real-Time Statechart may include additional, component internal behavior which, e.g., resolves dependencies between the roles. A detailed description can be found in Section 3.4.1.4. The component is then formally verified for deadlock freedom to ensure that all roles safely cooperate with each other.

At this point, we apply the compositional verification approach of MECHATRONIC UML. Compositional verification enables to verify Real-Time Coordination Patterns and components isolated from each other. At first, each Real-Time Coordination Pattern is verified for safety and bounded liveness properties. Then, the verified correctness of the Real-Time Coordination Patterns with respect to the safety properties is used to verify each component separately. That avoids verifying a large system in a single step which is, in general, infeasible. Background information on the compositional verification theorem is provided in Section 6.1.

Instead of deriving a behavior specification for a component as an atomic component, it may also be decomposed into a set of components as a structured component as described before. This is realized in Step 4b. Then, Real-Time Coordination Patterns specifying the
interaction between the newly created subcomponents need to be modeled and Steps 3 and 4a are repeated.

In Step 5a, the model of the system containing only the discrete behavior of the system is integrated with the controllers of the mechatronic system. The controllers are integrated as continuous components into the model. MECHATRONIC UML itself provides no behavior specification for controllers. Instead, we assume that the behavior is specified by a control engineering tool like CamelView or MATLAB/Simulink. The correct integration of the controllers, however, cannot be verified formally, but only simulated in the respective control engineering tools. Such simulation requires a complete instance of the system including its controllers. MECHATRONIC UML supports such instances by means of component instance configurations which are discussed in detail in Section 3.5.

If the verification of the components or the simulation of the whole system fails, either the system model or the formal requirements need to be changed. This is subject to Step 5b of the process and requires a repetition of the whole process.

In Step 6, a deployment which assigns the components to a target hardware is created for the modeled system. The deployment defines the hardware on which the software operates. Then, the component instances contained in a component instance configuration are assigned to the hardware. That results in a platform-specific model of the system which is then used for code generation. Further information on deployments is provided in Section 3.6.

At present, the model-driven design and verification of reconfiguration is not explained in this document. For now, we refer to [EHH+11, HPB12, THHO08, BGO06] for more information on reconfiguration. A detailed description, however, will be added to future versions of this document.

Figure 2.2 summarizes the modeling languages that are used during the development and their relationships. Modal Sequence Diagrams define the formal requirements for Real-Time Coordination Patterns. Real-Time Coordination Patterns are used to define the communication behavior of the components of the system. We use Real-Time Statecharts to define the behavior of the roles of the Real-Time Coordination Pattern. The messages that are exchanged between the roles are formally declared in message interfaces (cf. Section 3.2). The components of the system instantiate the pattern and may refine it by adding internal computations. Components are distinguished into atomic components and structured components. Structured components are composed of a set of other components while atomic components have a behavior specification. That behavior specification is, again, specified by Real-Time Statecharts. The components are instantiated in a component instance configuration which may then be deployed on hardware in a deployment.

In conclusion, MECHATRONIC UML provides a component-based system model that separates components and their interactions in terms of Real-Time Coordination Patterns. This separation is the key enabler for the compositional verification theorem which permits the formal verification of large systems. By integrating the controllers of the system by means of continuous components, MECHATRONIC UML supports the integrated development of software for mechatronic systems and the integration of models of software engineering and control engineering. Furthermore, the correct timing behavior of reconfiguration between differ-
Figure 2.2.: Overview of the Modeling Languages used in MECHATRONICUML
ent controllers during run-time may analyzed formally using correct embeddings as discussed in [Bur06, Hir08].
Chapter 3.

Modeling Languages

In this chapter, we will introduce the different modeling formalisms that MECHATRONICUML offers. MECHATRONICUML uses a component model to specify types of architectural entities of the system under construction and Real-Time Coordination Patterns to model communication between those entities.

In Section 3.1, we introduce Real-Time Coordination Pattern in detail. The types of messages that may be exchanged between components are typed by means of message interfaces which we describe in Section 3.2. The behavior of the communicating entities is specified by Real-Time Statechart (cf. Section 3.3), an extension of UML Statemachines [Obj09] by clocks as known from timed automata [AD94]. In Section 3.4, we introduce the MECHATRONICUML component model which uses Real-Time Coordination Pattern and Real-Time Statechart. MECHATRONICUML provides an instance model to specify concrete system configurations which we explain in Section 3.5. Finally, it is possible to deploy component instances to hardware elements as described in Section 3.6.

3.1. Real-Time Coordination Pattern

MECHATRONICUML partitions the component behavior into internal and communication behavior. Real-Time Coordination Patterns specify the behavior that component instances (respectively their ports) have to fulfill for communicating with each other for the purpose of coordination. Furthermore, they take real-time requirements regarding the coordination and communication into account.

A Real-Time Coordination Pattern specifies the message- and state-based coordination and communication of coordination partners, e.g., server and client, which are referred to as roles (cf. Section 3.1.2). In a Real-Time Coordination Pattern, role instances are communicating with each other over communication connectors (cf. Section 3.1.3). Real-Time Coordination Patterns differ on the direction of communication and on the form of communication (cf. Section 3.1.4).

A developer has to instantiate a Real-Time Coordination Pattern for specifying a concrete coordination through communication among two or more role instances (cf. Section 3.1.5). For example, it is possible to define a coordination among one role instance that acts as the
server and multiple role instances that act as a client. The behavior of each role is described by a Real-Time Statechart (cf. Section 3.1.6).

Furthermore, the developer may specify safety and bounded liveness properties regarding the coordination between the roles or regarding a single role. For this, he has to assign them to the Real-Time Coordination Pattern or to one of its roles, respectively. The fulfillment of the properties may be formally verified (cf. Section 3.1.7).

### 3.1.1. Application Example

An example for a coordination is as follows: A system consists of one distributor on the one side that communicates with one to eight clients on the other side. The distributor periodically receives information of all clients and distributes the combined information among them using a multi-cast. If one of the clients does not send its information within a certain time, the distributor informs all clients, that a safety-critical situation occurred. As soon as all clients send their information again, the safety-critical situation ends.

To model this coordination, we define a Real-Time Coordination Pattern with the name 

\[
\text{Distribution}
\]

The two types of roles in this communication are distributor and client. Each instance of role distributor can communicate with up to eight instances of role client bidirectionally. Furthermore, each instance of role client can communicate with exactly one instance of role distributor. Instances of role distributor cannot communicate with each other; instances of role client cannot communicate with each other, too. The formal behavior of each role is defined by a Real-Time Statechart. Section 5.1.3 shows and describes them in detail.

![Figure 3.1: The Real-Time Coordination Pattern Distribution](image)

Figure 3.1 shows the Real-Time Coordination Pattern Distribution in concrete graphical syntax. All concrete syntax elements of the example are annotated in grey. The explanation for the concrete syntax elements is as follows: A dashed ellipse contains the name of the pattern. A dashed square, which may be cascaded, with a dashed line that is connected to the ellipse represents a role. A label that is next to the dashed line shows the name of the role. The solid line, which represents the role connector, connects the roles. The two triangles within both roles define that they communicate bidirectionally with each other. The label that is next to
the role under the role connector is the so-called role-cardinality, which defines the number of connections a role may have.

### 3.1.2. Role

A role represents the type of a coordination partner of a Real-Time Coordination Pattern. Each role of a Real-Time Coordination Pattern has a unique name within a Real-Time Coordination Pattern. A role instance is typed over a role and represents a specific coordination partner.

Instances of roles can communicate with each other via discrete, asynchronous messages. Therefore, a role specifies the set of discrete, asynchronous messages that are typed over a message type (cf. Sections 3.3.12 and 3.2.2) that an instance of this role may send or receive. A role instance may only send or receive one message at a particular point in time.

Figure 3.2 shows the concrete syntax for roles. In general, roles are illustrated by dotted squares and a dashed line. This line is connected to the pattern ellipse, which contains the name of the pattern (cf. Figure 3.1). In addition, the name of the role is shown next to the dashed line and positioned on the outside of the pattern.

<table>
<thead>
<tr>
<th></th>
<th>single-role</th>
<th>multi-role</th>
<th>sender message interface</th>
<th>receiver message interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>out-role</td>
<td><img src="image" alt="out-role" /></td>
<td><img src="image" alt="out-role" /></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>in-role</td>
<td><img src="image" alt="in-role" /></td>
<td><img src="image" alt="in-role" /></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>in/out-role</td>
<td><img src="image" alt="in-out-role" /></td>
<td><img src="image" alt="in-out-role" /></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 3.2.: Concrete Syntax of Roles and Messages Interfaces a Role Contains

We distinguish roles (i) on their directions which specifies if they may send or receive messages and (ii) on the role-cardinality which specifies with how many other role instances an instance of this role may communicate to. These distinctions are also depicted within the concrete syntax of the roles.
### 3.1.2.1. Direction of Roles

The direction of a role specifies if a role may send, receive or send and receive messages. A role that may only send messages is an *out-role*. A role that may only receive messages is an *in-role*. A role that may send and receive messages is an *in/out-role*.

Roles may reference a sender and a receiver message interface (cf. Section 3.2.1). A sender message interface defines which types of messages may be sent via this role. A receiver message interface defines which types of messages may be received via this role. Thus, the direction of a role can be derived from its message interfaces. If a role references only a sender message interface, it is an out-role. If it references only a receiver message interface, it is an in-role. If it references both, it is an in/out-role.

As depicted in Figure 3.2, the concrete syntax of the out-role has a filled isosceles triangle within its square (the so-called out-triangle) whose top points to the connector of the Real-Time Coordination Pattern. The in-role has a triangle (the so-called in-triangle) whose top points away from the connector of the Real-Time Coordination Pattern. The in/out-role is visualized with two triangles. The top of the upper one points to the left and the top of the lower one points to the right.

### 3.1.2.2. Cardinality of Roles

Each instance of a role has one or more connections to other role instances. For each connection a so-called *sub-role-instance* exists within the role-instance. Therefore, each connection consists of exactly two sub-role-instances of different role instances. The number of connections and therefore the number of sub-role-instances a role instance may have is limited. Therefore, for each role a cardinality exists, which describes how many sub-role-instances each role instance may have.

We describe the role-cardinality by the Min-Max-Notation [Abr74]. This means, a developer has to determine for each role how many sub-role-instances an instance of this role may have at minimum and at maximum and therefore the minimum and maximum number of connections of this role instance. The Min-Max-Notation is contrary to the multiplicity notation of the UML. The Min-Max-Notation defines for each entity of type x how many associations of type y (at minimum and at maximum) to entities of type z it may have. The multiplicity notation of the UML defines how many entities of type x may be (at minimum and at maximum) associated over associations of type y to one entity of type z.

The role-cardinality may be variable or fixed. If it is variable, then the minimum-cardinality and the maximum-cardinality are different. If it is fixed, then minimum and maximum are equal.

If the role-cardinality has a fixed value of 1, which means this role can communicate to only one other role-instance, then we call this role a *single-role*. If the role-cardinality has an upper bound greater than 1, which means this role can communicate to more than one other role-instance, then we call this role a *multi-role*.
3.1. REAL-TIME COORDINATION PATTERN

Within the concrete syntax a square with a dashed, single borderline visualizes the single-role. A square with a dashed, cascaded borderline visualizes the multi-role (cf. Figure 3.2). The role-cardinality is depicted as a label that is located next to the role under the role connector. The label consists of square brackets and one number or two numbers separated by two dots within. It the cardinality is fixed, then only one number is shown and shows the value of the fixed cardinality. If the cardinality is variable, then two numbers are shown; the lower bound is the first number, the upper bound is the second number.

A multi-role can be marked as ordered. In an instance of an ordered multi-role, a total order of the sub-role instances is defined. The order can be used within the behavior specification of the multi-role as explained in Section 3.3.13. In the concrete syntax, an ordered multi-role is marked by the annotation \{ordered\} which is placed below the role as shown in Figure 3.1.

3.1.2.3. Constraints

- A single-role has a fixed role-cardinality of 1.
- A multi-role has role-cardinality with minimum \( i \) and maximum \( j \) with \( i, j \in \mathbb{N} \land i \geq 1 \land j \geq 2 \land i \leq j \).
- Only multi-roles can be ordered.
- An ordered role requires a defined orderedVariable of type EInt.
- A role requires an incoming or outgoing role connector.
- A role requires a behavior.

3.1.3. Role Connector

A role connector connects the roles of a Real-Time Coordination Pattern. It represents a communication connection between the roles and is visualized by a black line between the two squares for the roles (cf. Figure 3.1).

For analysis purposes (e.g., simulation or verification) the developer may specify a dedicated real-time behavior for a communication connector, e.g., to model propagation delays, buffering of messages, message loss, corrupt messages or a wrong order of messages. The developer has to use a Real-Time Statechart to describe this behavior [Bur06].

As the default connector behavior, we assume that all messages of a connector have the same propagation delay (greater than zero), messages are buffered and can be lost. Though, corrupt messages or a wrong order are not considered.

3.1.3.1. Constraints

- Source and target role have to belong to the same Real-Time Coordination Pattern.
3.1.4. Kinds of Real-Time Coordination Patterns

Real-Time Coordination Patterns differ on the direction of communication (unidirectional or bidirectional) and on the form of communication (one-to-one or one-to-many). Figure 3.3 shows the five kinds of a Real-Time Coordination Pattern.

![Diagram of Real-Time Coordination Patterns]

Figure 3.3.: The Five Kinds of a Real-Time Coordination Pattern

3.1.4.1. Direction of Communication

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

A bidirectional communication means that all roles can send and receive messages. Thus, all roles must be in-out-roles. Figures 3.3 d) + e) show the two possibilities for such a communication.
3.1. REAL-TIME COORDINATION PATTERN

3.1.4.2. Forms of Communication

In MECHATRONIC UML, two possible forms of communication exist: one-to-one and one-to-many. These forms of communication define how many other instances one role can communicate with and how many instances participate in this communication.

One-to-one means that two roles communicate with each other and both roles have only one instance per instantiated Real-Time Coordination Pattern. Both roles have a role-cardinality of 1. Therefore, both roles must be single-roles. Figures 3.3 a) + d) show the two possibilities for a one-to-one communication.

One-to-many means that two roles communicate with each other and one role has only one instance and communicates with multiple instances of the other role. The role with one instance may have multiple sub-role-instances and can therefore communicate with multiple instances of the other role. The role, which may have multiple instances, has a role-cardinality of 1, because there is only one instance of the other role. A pattern, which specifies such a communication has one multi-role and one single-role. Figures 3.3 b), c) + e) show the three possibilities for specifying a one-to-many form of communication.

3.1.4.3. Constraints

- The names of the roles of one Real-Time Coordination Pattern are unique.
- A Real-Time Coordination Pattern has either one out-role and one in-role or two in-out-roles.

3.1.5. Real-Time Coordination Instance

An instance of a Real-Time Coordination Pattern consists of a set of role instances that are typed over the roles specified in the Real-Time Coordination Pattern and a set of role connector instances that are typed over the role connector of the Real-Time Coordination Pattern. We will illustrate this technique with the already introduced Real-Time Coordination Pattern Distribution (see Figure 3.1).

During instantiation, the variable parts of the Real-Time Coordination Pattern are determined. If the Real-Time Coordination Pattern has the form of communication one-to-one, there exist no variable parts, because the pattern consists of two single-roles that are both connected to an instance of the other role. Therefore, the only possible pattern instance consists of one instance per role and one connector instance that connects the two role instances.

If the Real-Time Coordination Pattern has the form of communication one-to-many the variable parts include the variable role-cardinality of the multi-role and the number of role instances of the single-role. A developer determines both parts at the same time, because they depend on each other. The explanation for this is as follows: A determined role-cardinality defines for a multi-role-instance how many sub-role-instances it consists of. Each sub-role-instance of a multi-role is connected to a different single-role instance. Therefore,
in a one-to-many communication, there exists only one multi-role-instance but several single-role-instances, where the number of single-role-instances depends on the determined role-cardinality of the multi-role-instance. Each pair of a sub-role-instance and a single-role-instance is connected by a different role connector instance.

An example for a Real-Time Coordination Pattern Instance is as follows: An instance of Real-Time Coordination Pattern Distribution specifies a communication between four role instances. One instance is typed over the role distributor and contains three sub-role-instances. Each of these three is connected via different role connector instances to one of the three instances which are typed over the role client. This instance of Real-Time Coordination Pattern Distribution is valid because the maximum role-cardinality of role distributor is eight and the current role-cardinality of the role instance which is typed over role distributor is three. To conclude, this instantiation defines a 1:3 communication with one distributor and three clients.

Figure 3.4.: Instance of Real-Time Coordination Pattern Delegation

Figure 3.1 shows the aforementioned instance of Real-Time Coordination Pattern Distribution in concrete graphical syntax. All concrete syntax elements of the example are annotated in grey. The explanation for the concrete syntax elements is as follows: A dashed ellipse contains the pattern instance label which consists of the name of the pattern that is underlined and prefixed by a colon. Each instance of a single-role has its own dotted square and its own dashed line, which points to the ellipse. A multi-port-instance contains a fixed number of sub-role-instances, which are framed by a dashed line. Each sub-role-instance has its own dotted square. A multi-port-instance points to the ellipse via a dashed line. A label is placed next to dashed line of each role-instance (single and multi) and consists of the name of the role that is underlined and prefixed with a colon. Single-role-instances and sub-role-instances contain triangles to show the direction of the role-instance. The solid line, which represents the role connector instance, connects a single-role instance and a sub-role-instance with each other. The same graphical syntax for the role connector instance is used if a pattern instance has a one-to-one form of communication where two single-role instances are connected.
3.1.6. Behavior Specification

The concurrent execution of the roles of a Real-Time Coordination Pattern specifies the execution behavior of the Real-Time Coordination Pattern. Therefore, the developer has to specify the communication behavior for each role. This role behavior specification can be seen as a contract that a port must fulfill, when a role is applied to it. If a role is applied to a port of an atomic component, the port has to fulfill the role behavior specification directly. If the developer applies a role to a port $p_1$ of a structured component, the behavior specification of the role must be fulfilled by the port the port $p_1$ delegates to.

The behavior of a role is state-based and is subject to real-time restrictions (e.g., the maximum time the system dwells within a certain state). The developer specifies the behavior of a role by a Real-Time Statechart (cf. Section 3.3). The concurrent execution of all roles instances of an instantiated Real-Time Coordination Pattern determines the behavior of this Real-Time Coordination Pattern.

Within the Real-Time Statechart of a role, only asynchronous message events can be used (cf. Section 3.3.12) to specify the interaction between the roles. These message events are typed over the message types declared in the message interfaces of the role (see also Section 3.2). Asynchronous messages may not be used for communication within the role (e.g., to communicate between several regions).

A role may define variables to store information regarding the communication (e.g., the current physical position of the system). The Real-Time Statechart of the role can access these variables and use them for transition guards or can change them in side effects of the transitions. The Real-Time Statechart of the role cannot access the variables that are defined by other role instances. Furthermore, a role can define operations (e.g., to calculate the current position of the system) and the side effects of transitions (cf. Section 3.3.7). The Real-Time Statechart of the role can call these operations and side effects, but cannot directly call operations and side effects of other role instances.

Figure 3.5 shows the Real-Time Coordination Pattern Distribution with its roles distributor and client. As depicted, the behavior of a single-role, like client, is defined by a Real-Time Statechart without further restrictions. The behavior of a multi-role, like distributor, is separated into an adaptation behavior and a sub-role behavior. That results from the one-to-many communication where the multi-role has to communicate with several instances of the single-role. The behavior to communicate with each instance of the single-role has to be identical and is defined by one common sub-role behavior. The adaptation behavior defines an order of the different communications to each instance of the other role and will be used for specifying run-time adaptations of the communication as introduced in [EHH+11] in future versions of this document. We introduce the different parts of a Real-Time Statechart for a multi-role in more detail in Section 3.3.13.
3.1.7. Verification of Properties

The behavior specification of a Real-Time Coordination Pattern must usually fulfill properties regarding the safety or the bounded liveness of the system. The definition for these two types of properties is as follows: “A safety property is one that something bad will not happen. […] A liveness property is one which states that something good must happen.” [Lam77] A common safety property is that there will never be a deadlock within the system. A common liveness property is that all states may be accessed.

Using MECHATRONICUML a developer can automatically verify these constraints using model checkers (e.g. UPPAAL [BDL04]), which explore the whole state space of the concrete coordination. This state space is defined by the Real-Time Statecharts, which are defined for each role and for the connector. To verify a property, it has to be defined in a formal logic like the Timed Computational Tree Logic (TCTL) [ACD93].

Figure 3.6 visualizes the Real-Time Coordination Pattern Distribution and shows properties for its role distributor and the pattern itself. A black-lined rectangle contains all verification properties of a role or a pattern. The rectangle links via a black line to the dashed square of a role (here: to the role distributor) or to the black-dashed ellipse, which displays the name of the pattern (here: to the pattern Distribution). If a role or a pattern has no verification properties (like the role client), the rectangle and its link are not shown.

In our example, two TCTL-properties are specified for the pattern Distribution using the concrete syntax of the model checker UPPAAL: The first property states $A[] \text{no deadlock}$, which expresses that there exists no deadlock within this pattern. The second property states $\text{distributor.adaptation.Error} \rightarrow (\text{client_1.receive.Error} \land \ldots \land \text{client_8.receive.Error})$, which expresses that if role distributor is in state Error (which is embedded in region adaptation), then all clients will eventually enter state Error of region receive. In other words, if the distributor
3.1. REAL-TIME COORDINATION PATTERN

Figure 3.6.: The Real-Time Coordination Pattern Distribution with Properties for the Role distributor and the whole Pattern

cannot collect the position data of all clients, then the clients will be informed so that they can react to this error. The role distributor specifies the TCTL-property $A[] \text{adaptation.c0} \leq 50$, which formally expresses that the clock $c0$ of region adaptation never exceeds 50 time units. If this constraint always holds, then the developer can ensure that the distributor can distribute its information every 50 time units.
3.2. Message Interface Specification

Message Interfaces define the interfaces of the roles of parameterized coordination patterns and of the ports of components (cf. Section 3.4). Each message interface defines a set of message types. Message types are used to type asynchronous messages that may be exchanged between two roles of a Real-Time Coordination Pattern (cf. Section 3.1) or sent via the respective port. Since they are always attached to a role or port, they are considered as second class objects in MechatronicUML.

3.2.1. Message Interface

A message interface defines a set of message types for asynchronous messages that may be exchanged between the roles of a Real-Time Coordination Pattern. Thus, they specify the interfaces of roles and discrete ports. Each message interface specifies a name that must be unique within the modeled system. Figure 3.7 shows an example of two message interfaces with the names Delegation_Master and Extended_Delegation_Master.

Message interfaces are visualized as as rectangles with two compartments that are separated by a horizontal line. The upper compartment contains the name of the message interface, the lower compartment contains a list of message types where each line contains one message type.

Message interfaces support inheritance relations as they are known from class models like UML [Gro10b] or Ecore [SBPM08]. A message interface may inherit from multiple other message interfaces. In Figure 3.7, the message interface Extended_Delegation_Master inherits directly from Delegation_Master. In this case Delegation_Master is the super message interface and Extended_Delegation_Master the sub message interface. The inheritance relation is transitive, i.e. sub message interfaces of Extended_Delegation_Master inherits indirectly from Delegation_Master.
In our concrete syntax, we denote an inheritance relation by an arrow leading from the sub message interface to the super message interface. The arrow head is an unfilled triangle. Such an arrow between two message interfaces always denotes a direct inheritance.

A message interface contains all message types that it defines itself and all message types of all direct and indirect super message types. In the concrete syntax, a message interface only displays the message types that it defines itself.

3.2.1.1. Constraints

- A message interface must either specify at least one message type itself or it must inherit from at least two message interfaces. Otherwise, no specialization takes place.
- A message interface must not inherit from itself (directly or indirectly).
- The names of all message interfaces are unique.

3.2.2. Message Type

A message type declares a name and an ordered parameter list for an asynchronous message of a Real-Time Statechart. Each parameter specifies a name and its concrete type.

In our concrete syntax, a message type is represented as a string adhering to the following BNF:

\[
\begin{align*}
\langle \text{messagetype} \rangle & \ ::= \ \# \text{MessageType} . \text{name} \ ( \ [ \langle \text{parameterlist} \rangle ] ) \\
\langle \text{parameterlist} \rangle & \ ::= \ \langle \text{parameter} \rangle \ | \ \langle \text{parameter} \rangle \ , \ \langle \text{parameterlist} \rangle \\
\langle \text{parameter} \rangle & \ ::= \ \# \text{Parameter} . \text{type} . \text{name} \ \# \text{Parameter} . \text{name}
\end{align*}
\]

Thus, the concrete syntax of a message type is similar to the concrete syntax of a method declaration in UML [Gro10b]. The parameter list is optional, i.e., a message type may also declare no parameters.

In Figure 3.7, the message interface Delegation_Master declares the message type check. This message type specifies one parameter named target and is of type integer array with length 2. The message interface Extended_Delegation_Master inherits this message type and additionally defines the message type checkEnv. This message type specifies two parameters. These are target of type integer array with length 2 and radius of type double.

The message types within a message interface may not be overridden or overloaded. Therefore, the name of a message type must be unique for a message interface. It is, however, allowed that message types of different message interfaces where neither inherits from the other have the same name.

3.2.2.1. Constraints

- The names of all message types within on message interface are unique.
- Each parameter of a message type has a unique name.
CHAPTER 3. MODELING LANGUAGES

3.3. Real-Time Statechart

Real-Time Statecharts are state-based models which are used for defining the behavior of components (cf. Section 3.4.1.4), ports (cf. Section 3.4.1.3), or roles (cf. Section 3.1.6).

Both syntactically and semantically, Real-Time Statecharts are a combination of Harel’s statecharts [Har87], UML state machines [Gro10b], and timed automata [DM01, BDL04, AD94]: Concepts and syntactical elements for supporting hierarchy are derived from statecharts and state machines, while notions and notations from timed automata allow for defining time-based behavior. For ensuring deterministic behavior, we also add additional features such as deadlines for transitions or priorities. Giese and Burmester [GB03] defined a previous version of Real-Time Statecharts.

For describing behavior, a Real-Time Statechart may contain several different types of modeling elements. As an extended variant of finite automata, Real-Time Statecharts define behavior by means of states and transitions. More advanced concepts for the specification of control flow are supported by history elements, entry-points, and exit-points. Like in timed automata, time-based behavior is modeled using clocks. Another important element of Real-Time Statecharts are actions, which describe behavior that can be referenced by states and transitions. All of these elements will be explained within the following subsections.

For keeping the behavior model compact, Real-Time Statecharts support hierarchy and orthogonal behavior [Gro10b]. A state can indirectly include (sub-) states by embedding regions, each of which in turn contains exactly one other Real-Time Statechart. Such a statechart contained in a region of another statechart is also called a sub-statechart. In contrast to the submachine states defined in the UML, we do not permit to reference the same Real-Time Statechart from more than one region [Gro10b, p. 551]. We make this restriction to avoid additional complexity concerning the scopes of clocks and variables. As we already offer explicit support for re-use both for components and for patterns, we regard an additional low-level way of re-use as unnecessary. If a Real-Time Statechart is the top element in the hierarchy, we call it a root-(real-time) statechart.

As Real-Time Statecharts are primarily intended for modeling the behavior of reactive, event-based systems, the ability to describe communication is essential. Real-Time Statecharts can communicate with each other within the same atomic component by using synchronizations. Communication with the ports of other components is possible using asynchronous message events. Each statechart of a port or a role has a message event pool assigned to it, as suggested in the UML specification [Gro10b]. A message event pool stores the incoming messages from the statechart of the connected role or port until they are handled by the Real-Time Statechart. Both variants of communication, synchronous and asynchronous, are explained in Sections 3.3.11 and 3.3.12.

A Real-Time Statechart may define variables and clocks as its attributes. These can be referred to by all elements contained in the Real-Time Statechart, including indirectly or even recursively contained elements in sub-statecharts. The rtscDefinitionLabel (cf. Figure 3.8) shows the defined variables and clocks of a statechart.
Figure 3.8 shows a template for the concrete syntax of a statechart: The visual representation of a Real-Time Statechart is a rectangle with the name of the statechart (rtscNameLabel) shown in the upper left corner and the rtscDefinitionLabel in the upper right corner. The remainder of the rectangle may contain state compartments, the visual representation of states.

The following EBNF expression defines the default notation of an rtscDefinitionLabel. Elements which have the prefix “#” are references to the meta model elements of a statechart:

```
<rtscDefinitionLabel> ::= [ <operationDefinition> ]
                     [ <varDefinition> ]
                     [ <clockDefinition> ]
<operationDefinition> ::= 'op:' '#eClass.eOperations.eType`
                      '#eClass.eOperations.name'( '#eClass.eOperations.eParameters [','
                      '#eClass.eOperations.eParameters]' )'
<varDefinition> ::= 'var:' '#eClass.eAttributes.eAttributeType`
                  '#eClass.eAttributes.eAttribute.name [','
                  '#eClass.eAttributes.eAttribute.name]'`
<clockDefinition> ::= 'cl:' '#clocks.name [','
                  '#clocks.name]'`
```

Figure 3.9 shows an example of the concrete syntax for Real-Time Statecharts, except for the state compartments. For a description of the concrete syntax for states, see Section 3.3.2.

Constraints

- A statechart contains at least one state.
3.3.1. Clock

A Real-Time Statechart, like a timed automaton, has a finite number of clocks. A clock models the elapsing of time during the execution of a system. Time elapses continuously, not in discrete steps [AD94]. Entry- or exit-actions (cf. Section 3.3.7) and transitions (cf. Section 3.3.4) can reset clocks to zero. The time value represented by a clock is relative to the last point in time when the clock has been reset.

3.3.2. State

A state represents a situation in which the system resides while the state is active. Each state has a name, which must be unique within the same statechart. Possible state changes are defined by directed transitions (cf. Section 3.3.4), connecting source states with target states. A state can have side-effects as entry-, do-, and exit-actions. We define the execution semantics of actions in Section 3.3.7.

In MECHATRONIC UML we distinguish between simple states, composite states, and orthogonal composite states. A simple state has no hierarchy and has no embedded elements. The developer can add hierarchy to a statechart by adding a region to a state. A state which contains at least one region is called a composite state. Furthermore, regions allow to model orthogonal behavior [Gro10b] (cf. Section 3.3.3).

We call a state the system currently resides in, an active state. The initially active state of a Real-Time Statechart is defined by its initial state (cf. Section 3.3.2.1). In the root statechart and in each region of an active composite state exactly one state is always active. If a composite state contains more than one region, it is an orthogonal composite state.

Real-time systems usually have to fulfill hard real-time constraints. Therefore, the developer needs a way to express that such a system will have to leave a state in a specific point in time. In Real-Time Statecharts this requirement can be specified for each state as a time invariant. The developer specifies such a time invariant as a concrete upper time bound which cannot be surpassed in the corresponding state.

An invariant forces the system to leave the corresponding state via an outgoing transition at the latest when its upper time bound is reached. If this is not possible because no outgoing transition can currently fire, the statechart is in a time-stopping deadlock [DMY02a]: In such a situation time in the model cannot pass any more because this would violate the invariant. This behavior cannot be implemented in a real system and therefore has to be prevented. Time-stopping deadlocks can be identified by means of formal verification (e.g., using the uppaal model checker). One possible way to resolve time-stopping deadlocks is adding a transition to a fail-safe state that can be taken when reaching the upper time bound defined by the invariant.

Note that for a correct implementation of Real-Time Statecharts, all transformations to more concrete models, including the code-generation, and finally the deployment have to make sure that all invariants are actually obeyed by the real system. Formal verification, however, will

\[\text{In our recent papers composite states are also called complex states.}\]
only prove the system correct under the assumption that the concrete implementation derived
from the Real-Time Statechart-model adheres to these invariants.

The different regions of an orthogonal composite state can communicate via synchroniza-
tions which are typed by channels (cf. Section 3.3.11). These channels belong to the orthogo-
nal composite state containing the regions.

Figure 3.10 shows a template for the state and statechart syntax. We define the concrete
syntax of a state as follows. In general a state is shown as a rectangle with rounded corners,
with the state name shown inside the rectangle. Further, if an entry-, an exit-, or a do-action is
set, it has an internal action compartment which displays the state action label. This compart-
ment is visually represented by a StateActionLabel. A state’s channels are visualized within a
ChannelDefinitionLabel.

The following EBNF expression defines the default notation for a StateActionLabel. Elements
which have the prefix “#” are references to the meta model elements of State:

\[
\begin{align*}
<\text{StateActionLabel}> & ::= [<\text{entryAction}>] \\
& \quad [<\text{doAction}>] \\
& \quad [<\text{exitAction}>]
\end{align*}
\]

\[
\begin{align*}
<\text{entryAction}> & ::= 'entry' [ \{' (#entryEvent.action.expressions | \\
#entryEvent.action.name)\}'] \\
& \quad ['\{' reset:'#entryEvent.clockResets[', ' #entryEvent.clockResets[*']\}' ]
\end{align*}
\]

\[
\begin{align*}
<\text{doAction}> & ::= 'do' [ \{' (#doEvent.action.expressions | \\
#doEvent.action.name)\}'] \\
& \quad [\{' #doEvent.periodLower', ' #doEvent.periodUpper'\}]
\end{align*}
\]

\[
\begin{align*}
<\text{exitAction}> & ::= 'exit' [ \{' (#exitEvent.action.expressions | \\
#exitEvent.action.name)\}'] \\
& \quad ['\{' reset:'#exitEvent.clockResets[', ' #exitEvent.clockResets[*']\}' ]
\end{align*}
\]

The following EBNF expression defines the default notation for a ChannelDefinitionLabel. Elements
which have the prefix “#” are references to the meta model elements of state:

\[
\begin{align*}
<\text{ChannelDefinitionLabel}> & ::= 'ch' [ #channels.name[ ('#channels.inParameter.cType \\
[', '#channels.inParameter.cType[*']\}' ] \ ' , \\
#channels.name[ ('#channels.inParameter.cType[ ', \\
' #channels.inParameter.cType[*']\}' ] ] \\n\end{align*}
\]

Figure 3.11 shows a simple state, whereas Figure 3.12 shows a simple state with an entry,
exit-, and do-action.

A (non-orthogonal) composite state has an additional compartment for displaying the sub-
statechart referenced by its (single) region. Figure 3.13 shows a composite state. Its sub-
statechart also contains two other states.

The concrete syntax for orthogonal composite states is defined in Section 3.3.3, after ex-
plaining regions.

**Constraints**

- A state has at most one invariant per clock in its statechart.
CHAPTER 3. MODELING LANGUAGES

Figure 3.10.: Concrete Syntax Template of a State with StateActionLabel, ChannelDefinitionLabel and the Parent Statechart with the StatechartAttributeDefinitionLabel

Figure 3.11.: Concrete Syntax of a Simple State

Figure 3.12.: Concrete Syntax of a Simple State with Actions

Figure 3.13.: Syntax of a Complex State with One Region Compartment
3.3. REAL-TIME STATECHART

- A state has a unique name.
- The bound of clock invariant has to be greater or equal to zero.
- In a clock constraint only a lower-equal or strictly lower-than operator can be used.
- Each outgoing transition of a state has a unique priority.
- A DoEvent’s lower period has to be greater than or equal to 1 and lower than or equal to its upper period.

3.3.2.1. Initial State

The developer can declare a state as the initial state of its Real-Time Statechart. This state is the first state that becomes active after activating the parent region.

An initial state is shown as a state which is marked with a black-filled circle and a directed edge from the black-filled circle to the initial state. Figure 3.14 shows an initial state.

![InitialStateName](InitialStateName)

Figure 3.14.: Concrete Syntax of an Initial State

Constraint

- Only one initial state per statechart is permitted.

3.3.2.2. Final State

The developer can declare a state as a final state. This state denotes that a statechart is in a stable configuration. This means, the parent region of the state cannot change its state anymore. Unlike in the UML, reaching a final state in a Real-Time Statechart does not lead to the statechart’s termination. Due to this difference we allow a final state to define an entry-, do-, and exit-action. This is also in contrast to the UML, which does not allow any of these.

As a deactivation of the parent composite state is the only way to leave a final state, this is also the only case resulting in the execution of the final state’s exit action. A final state is shown as a state with a border drawn as a double-line. Figure 3.15 shows a final state.
Constraints

- A final state has no outgoing transition.
- A final state has no region.

3.3.3. Region

A region introduces hierarchy and enables orthogonal behavior in a statechart. We allow to add one or multiple regions to a state. Regions are active iff the parent composite state is active. A region contains exactly one Real-Time Statechart and cannot directly contain other elements such as states or transitions. The root-statechart is not contained by any region. Each region has a name. If a state contains more than one region, the names of all regions must be unique. If the region contains a Real-Time Statechart of a port or role, its name typically is the name of this port or role (cf. Section 3.4.1.3).

The structure of statecharts, containing states, and embedded statecharts must be an acyclic graph. It is forbidden to reference a statechart in a region (recursively) contained in that statechart.

Real-Time Statecharts have a deterministic behavior. Even when in more than one region a transition is ready to fire at the same time instance, a non-deterministic choice of the execution order is prevented: This is ensured by assigning a priority to each region, enforcing a sequential semantics as Zündorf [Zü01, p. 159] defined. These priorities must be unique among all regions with the same parent state. Higher numbers indicate higher priorities. When executing a Real-Time Statechart, transitions in regions with a lower priority can only fire if no transition in any region with a higher priority within the same parent state is enabled. Also, do-actions of active states are executed in order of their regions’ priorities, from the highest to the lowest.

Like composite states with just one region, orthogonal composite states have an additional compartment for sub-statecharts. However, as orthogonal composite states contain several regions, this compartment is tiled into smaller areas, each of them displaying the sub-statechart referenced by one region. This tiling is either vertical or horizontal. The borders between the regions’ areas are visualized by dashed lines. A small circle in the upper right corner of the region’s area contains the priority value. A solid line separates the state’s text compartment from the compartment for the orthogonal regions. Figure 3.16 shows an orthogonal state with two regions.
3.3. REAL-TIME STATECHART

Figure 3.16.: Concrete Syntax of an Orthogonal State With Two Regions

**Constraints**

- Each region has a unique priority.
- Each region has a unique name.

### 3.3.4. Transition

A transition represents a possible state change of a statechart from a source state to a target state. Transitions can be annotated with guards, clock constraints, clock resets, deadlines, synchronizations, trigger and raise messages, transition-actions, and priorities.

A state change from a source state to a target state happens when a transition *fires*. This means that (1) the source state of the transition changes its status from active to inactive and that (2) the target state of the transition changes its status from inactive to active. Further, (3) the exit-action of the source state is executed, (4) the transition-action of the transition is executed, (5) the raise message event of the transition is released, and finally (6) the entry-action of the target state is executed. All of this happens sequentially in the order of the previous enumeration.

During a state change an amount of time between the lowest and the highest relative deadline can pass. However, this can be restricted by absolute deadlines, which for each clock of the Real-Time Statechart may further restrict the possible time values of that clock allowed while firing the transition. The state change may not end before the lower bounds and not after the upper bounds of all absolute deadlines that have been reached by the corresponding clocks. If no deadline is defined for a transition, its firing takes no time at all. The actual time needed for firing a transition or executing an action is not modeled in Real-Time Statecharts, as this information depends on the target platform and might not be available before the deployment.

We distinguish between *simple* and *high-level* transitions. The source state of a simple transition is a simple state. If the source state is a composite state, we call the transition a high level transition.

A transition fires only when it is *enabled*. As a prerequisite for enabling a transition the source state must be active. Also, if guards or clock constraints are defined for a transition, all of them must evaluate to true. If synchronizations are annotated at the transition, all of the corresponding synchronization channels must be enabled. For a more detailed description of
synchronization refer to Section 3.3.11. Transitions with an asynchronous trigger message will only trigger if that message is available in the message event pool of the Real-Time Statechart (cf. Section 3.3.12).

An enabled transition only fires if it has the highest priority compared to all other enabled transitions with the same source state. Priorities are represented as natural numbers, with a higher number indicating a higher priority. The priorities of all transitions with the same source state are required to be (locally) unique.

A transition does not have to fire immediately when it is enabled. However, after it has been enabled it is required to fire before becoming disabled again. Therefore, if the transition has a time guard with an upper bound, the realtime statechart may only delay firing the transition at most until the corresponding clock reaches that upper bound. Note that other outgoing transitions of the same state with a higher priority can also disable the first transition by becoming enabled themselves, which is allowed. In addition, the current transition might require a synchronization with another transition in an orthogonal region which can be disabled by delaying. In all cases the enabled transition is required to fire before its deactivation, regardless of the reason.

Figure 3.17 shows an example of the situation when the firing of an enabled transition may be delayed: The transition $B_1 \rightarrow B_2$ can, due to its time guard, only be enabled for $c_0 \in [2, 5]$. However, the transition $B_1 \rightarrow B_3$ has a higher priority and will, therefore, disable the first transition at $c_0 = 4$. Additionally, the transition $B_1 \rightarrow B_2$ is only enabled when the synchronization $sync()$? is offered by another transition. In this case this can only be $C_1 \rightarrow C_2$, which is enabled for $c_1 \geq 3$. Hence, the transitions $B_1 \rightarrow B_2$ and $C_1 \rightarrow C_2$ can only fire as a pair (see Section 3.3.11), which is possible if $c_0$ is in $[2, 4]$ and $c_1$ is at least $3$. Since both clocks are initialized with $0$ and advance synchronously, the firing can effectively occur for $c_0, c_1 \in [3, 4)$.

Figure 3.17.: Example Illustrating When a Transition is Required to Fire.
If several transitions are enabled of which source states are located on different hierarchy levels, only those enabled transitions with the highest located source state may fire. Therefore, transitions directly contained in a statechart have precedence over transitions contained in any sub-statechart referenced in one of its states. Only among these highest-level enabled transitions the region priorities (see Section 3.3.3) and finally, in case of another tie within the same region, the transition priorities (see above) are taken into account. The use of synchronizations can result in exceptions to this rule (cf. Section 3.3.11).

It is possible to specify so-called inter-level transitions which cross the border of a hierarchical state via entry-/exit-points. For a description refer to Section 3.3.5. As usual, the source state’s level of hierarchy is relevant for determining which one among the enabled transitions to fire first. However, for determining the order of execution it does not matter, on which level of hierarchy the target state of a transition is located.

If a transition is real-time-critical the developer can associate a deadline with it, specified as a lower and an upper bound. The lower bound specifies the minimum time the transition may take to fire and the upper bound specifies the maximum time it may take.

A transition is drawn as a line with an arrowhead. This arrow originates at the border of a state or state exit-point and ends at the border of either a state or a state entry-point. The priority value is displayed at the source of the transition. Elements annotated at a transition are visualized within a `transitionLabel`. The only exception is the definition of a deadline for which a special `deadlineLabel` is used. Figure 3.18 shows a template for the transition syntax.

```
State
1
<transitionLabel> <deadline label> State
2 <transitionLabel> <deadline label>
```

Figure 3.18.: Concrete Syntax Template of Transition with Transition Label and Deadline Label

The following EBNF expression defines the default notation for a `transitionLabel`. Elements which have the prefix “#” are references to the meta model elements of transition:

```
<transitionLabel>:=[<cIConstraints>][<guard>][<triggerEvent>][<sync>]
    [ '/' [<action>][<raisedEvent>][<clockResets>]]

<sync> ::= <receivedSync> | <sendSync>
<triggerEvent> ::= '#triggerEvent.name
<cIConstraints> ::= '[' #clockConstraints [ ',' #clockConstraints ]* ']
<guard> ::= '[' #guard ']
[action] ::= '({ (#transitionAction.expressions | #transitionAction.name) '})'
```
The following EBNF expression defines the default notation for a deadlineLabel. Elements which have the prefix “#” are references to the meta model elements of class Transition:

```plaintext
<deadlineLabel> ::= ["'\[ ' #relativeDeadline.lowerBound '; ' #relativeDeadline.upperBound ' ] ' ]

\n
'\n' #absoluteDeadline.clock.name '∈'

'\[ ' #absoluteDeadline.lowerBound '; ' #absoluteDeadline.upperBound ' ] ' ]
```

Figure 3.19 shows an example of a concrete transition.

![Figure 3.19: Concrete Syntax Example of a Transition](image)

**Constraints**

- The source and the target of the transition must be set.
- Transitions cannot cross region borders.
- Trigger or raised message events must be defined in the corresponding message interface of the discrete port or role.
• Trigger message events message has no owned parameter bindings.

3.3.5. Entry-/Exit-Point

Developers can use entry-/exit-points to realize inter-level transitions to or from specific states of sub-statecharts.

Entry- and exit-points can either be associated with a complex state or directly be top-level elements of a sub-statechart. In the first case they are also called state entry-/exit-points. Like states, entry- and exit-points can be the source or target of a transition.

Entry-points of a sub-statechart have exactly one outgoing transition to one state of that statechart. Defining an entry-point within a sub-statechart makes it possible for the statechart containing the parent state to directly enter that state. For this the parent-state additionally has to define a state entry-point referencing the entry-point of the embedded sub-statechart. This state entry-point, in turn, can be the target of a transition. Such a transition targeting a state’s state entry-point, in most ways, fires like an ordinary transition to or from that state (cf. Section 3.3.4). However, the active state of a sub-statechart embedded in the parent state entered in this way is not necessarily the initial state defined for that statechart: Instead, the new active state is the one targeted by the transition from the corresponding entry-point of the statechart. This transition itself does not define any additional behavior and may not carry any additional model elements. Its target state is reached immediately when the statechart is entered through the entry-point.

State entry points may also be defined for orthogonal complex states. In this case they may reference (up to) one entry-point of each sub-statechart, but at least one in total. For the remaining sub-statecharts the active state will be the initial state, as usual.

An entry-point is drawn as a small circle on the border of a sub-statechart or state, as shown in Figure 3.20.

Exit-points of a sub-statechart have exactly one incoming transition originating at a state of that statechart. Defining an exit-point within a sub-statechart makes it possible for the statechart containing the parent state to leave that state using a separate transition for that particular exit-point. For this the parent-state additionally has to define a state exit-point referencing the exit-point of the embedded sub-statechart. The execution semantics of transitions involving exit-points is similar to the one for those involving entry-points (see above): The transition to the sub-statechart’s exit-point is fired as usual, while the transition from the state exit-point only indicates the new active state within the statechart containing the parent state.

State exit-points may also be defined for orthogonal complex states. In this case the parent state is left if any of the sub-statechart’s exit-points is reached.

An exit-point is drawn as a small circle with a cross on the border of a sub-statechart or state, as shown in Figure 3.20.
CHAPTER 3. MODELING LANGUAGES

Constraints

• Outgoing transitions of entry-points and of state exit-points may not carry any additional elements.

• An entry-point’s outgoing transition target is either a state or a state entry-point.

• An exit-point’s outgoing transition target is either a state or a state exit-point.

• An entry-point has exactly one outgoing transition.

• An entry-points’ outgoing transition target state belongs to the same statechart as the entry point.

• An exit-point has at most one outgoing transition.

• An exit-point has at most one outgoing transition.

• An exit-point as exactly one incoming transition.

• An exit-points’ incoming transition source state belongs to the same statechart as the exit point.

• A state entry-point has least one incoming transition.

• A state entry-point has least one outgoing transition.

3.3.6. Shallow History

A Real-Time Statechart can have one history element, which stores the most recently active state if the region of the state in which the Real-Time Statechart is embedded is deactivated. The history element stores only the active state itself and not the active states of an active composite state’s sub-statecharts. As soon as the region is entered again, the most recent active state is directly activated again. Its entry-action is executed and after that its do-action is executed.
The UML distinguishes between shallow and deep history. Deep history stores “the state configuration that was active when the composite state was last exited” [Gro10b, p. 542]. MECHATRONIC UML currently only supports shallow history.

Shallow history is drawn as a circle which encloses the letter \( H \) (cf. Figure 3.21).

![Figure 3.21.: Concrete Syntax of Shallow History](image)

### Constraints

- A history element must have no incoming or outgoing transitions.

### 3.3.7. Action

The developer uses actions as a side effect of a transition as well as within a state. We have four different kinds of actions:

- **Transition-Action:** The firing of a transition causes the execution of the transition action.

- **Entry-Action:** The entry-action belongs to a state and is executed as soon as the state is activated.

- **Do-Action:** The do-action belongs to a state and is executed as soon as the execution of the entry-action is finished.

  If the developer did not define any entry-action, the do-action is immediately executed if the state is activated. A do-action is executed periodically. The developer defines the period via a lower and an upper bound. The lower bound defines the earliest time at which the do action is executed after the last execution. The upper bound defines the latest time at which the do action is executed after the last execution.

- **Exit-Action:** The exit-action belongs to a state and is executed as soon as the state is deactivated.

The effect of an action can be defined as an expression using our action language as defined in Section 3.3.10. The action language is parsed and stored as model instances. The action
Another possibility to define actions are **textual expressions** which can be an arbitrary language, such as Java, C, or Modelica. **Textual expressions** are not not read or parsed. We define no semantics for them. Further, it is possible to describe side effects also via graph transformation rules or story diagrams [EHH11].

An action has a name that represents it. There are two possibilities to display an action: Either its expression is displayed as a whole or the name of the action is displayed.

### 3.3.8. Variable

A Real-Time Statechart can define an arbitrary number of variables. A Real-Time Statechart uses variables to store data that is, e.g., received by asynchronous messages and can be used by guards of a transition.

A variable has a type. Real-Time Statecharts support primitive variables and array variables. A primitive variable is typed by a primitive type as, e.g., integer or double. An array variable holds multiple values of the same type. An array is typed by a primitive type and a cardinality. The number of values is defined by the cardinality of the array. The values of the array are accessed by an index. For a cardinality of \( n \), the indices range from 0 to \( n - 1 \).

An array can define more than one dimension. Each dimension of an array may specify an own cardinality, but all dimensions need to use the same primitive type.

### 3.3.9. Operation

An operation is a behavioral feature of a Real-Time Statechart. It is defined by an expression using our action language (cf. Section 3.3.10.3). In future versions of this document, we will also enable to specify operations by means of story diagrams [FNTZ00, vDHP12] that combine UML Activity Diagrams [Gro10b] and graph rewrite rules [Roz97]. An operation defines an ordered set of input parameters and a return type. Each input parameter is defined by a type and a name which uniquely identifies the parameter in the set of parameters of an operation.

### 3.3.10. Action Language

In MechatronicUML, we define an action language for specifying executable behaviors and Boolean evaluations. Especially, we use our action language for defining guards and clock constraints at transitions (cf. Section 3.3.4) and to define any kind of actions as described in Section 3.3.7. The action language consists of a well-defined abstract syntax in form of an own meta-model and a textual concrete syntax in form of an EBNF grammar.

Our action language supports the specification of assignments, comparison expressions, arithmetic expressions, logic expressions, and unary expressions. Expressions can be combined into a block and blocks can be included into control structures like loops or conditional
3.3. REAL-TIME STATECHART

statements. The action language is based on OMG Alf [Gro11a] and Uppaal [BDL04], which both have a largely C-legacy (“Java like”) syntax.

3.3.10.1. Block

A block is used to group expressions. Therefore, it consists of a list of expressions in a linear order. The braces ‘{’ and ‘}’ are used to encapsulate a code block and a new scope. Such a list of expressions may be attached as a body to a loop or represent a path of a conditional statement (cf. [Gro11a, p.97]).

Example

```
{ // block begin
  varA := varB; // body
} // block end
```

3.3.10.2. Expression

An expression represents a behavior and can be evaluated to a collection of values, which can also be void. Expressions which have side effects are called assignments. They change values of e.g. variables or parameters. Expression is the abstract super type of all elements in the action language (cf. [Gro11a, p.25].)

To perform an action on a value of an attribute of a Real-Time Statechart, we use the attribute expression, which has a reference to an attribute of the Real-Time Statechart and optional indices in squared brackets (‘[indexExpression]’) if a concrete array element should be referenced. To refer to a concrete value, we use the literal expression, which represents a value of a primitive data type. The unary expression is an expression with only one operator and one operand. An operand is the expression on which result the operator performs its action. A binary expression is an expression with one operator and two operands. The so called left-hand-side and the right-hand-side. We distinguish between three types of binary expression: (1) The arithmetic expression has the AssignOperator as operator, (2) a comparison expression has the ComparingOperator as operator, and (3) the binary logic expression has the LogicOperator as operator.

Example

```
{ // right-hand-side
  varA[5][2] := varB + varC; // arithmetic expression
  varD := varB < varC; // comparison expression
  varD := varB \ varC; // logic expression
  varD := not varC; // unary expression
}```
3.3.10.3. Operation Implementation

An operation of a Real-Time Statechart (cf. 3.3.9) can be implemented by a block of this action language. In the action language the return value of an operation is specified by the keyword `return` followed by one return expression.

Example

```{ return varA ; }
```

3.3.10.4. Operation Call

The operation can be invoked from the action language by an operation call. The operation call is written by a reference on the unique operation name followed by rounded brackets `()` which enclose the parameter bindings for the operation call. The paramater binding is specified by `parameter_name := expression`. The operation call is closed by an whitespace followed by a semicolon e.g. `;`.

Example

```{ // assign expression with an operation call which gets two binded parameter
  varA := operation ( parameterName := varB , parameterNameB := varC ) ;
}
```

3.3.10.5. Assignment

An assignment is used to assign a value to an attribute. A simple assignment is made using the `<ASSIGN>` Operator `:=`. Further, we have four more assign operators, which are used as abbreviated syntax form. An assignment is a binary expression. Therefore, it has a left-hand-side and a right-hand-side. The left-hand-side of an assignment must be a single attribute and must not be another expression. The right-hand-side expression evaluates to a value which is assigned to the left-hand-side attribute (cf. [Gro11a, p.92]).

Example

```{ varA := varB ; // assignment expression
  varA := varB [ varA ] ; // assignment expression
}
```
3.3.10.6. Loop

A loop statement executes a block until the Boolean value of loop test expression is false. The action language supports three kinds of loops. The while-statement first evaluates the loop test expression and afterwards, if the expression evaluates to true, it executes the block. The do-while-statement first executes the block and afterwards it evaluates the loop test expression. If the expression evaluates to true, it executes the block again. The for-loop-statement firstly initializes a loop variable by the initialize expression and afterwards assigns on each loop run a loop variable by the counting expression to successive values of a sequence (cf. [Gro11a, p.117 ff.]).

Example

```java
{ 
    while (loopTest) 
    { 
        varA := varA + 1; 
    }

do 
{ 
    varA := varA + 1; 
}while (loopTest);

for (loopVar := 1; loopVar < 22; loopTest++) 
{ 
    varA := varA + loopVar; 
}
}
```

3.3.10.7. If-Statement

An if-statement is used when the referenced block should be executed only if the condition expression evaluates to true. An if-statement always has one if-condition and one corresponding if-block, any number of else-if-conditions and corresponding else-if-blocks, and at most one else-block (cf. [Gro11a, p.112]).

Example

```java
{ 
    if (if_condition) 
    { 
        varC := 1; 
    }
    elseif (elseif_condition) 
    { 
```
CHAPTER 3. MODELING LANGUAGES

```c
varC := 2;
}
else
{
    varC := 3;
}
}

3.3.10.8. Operator

This section defines the different kinds of operators and the symbols for all operators. The action language distinguishes between assign operators, increment decrement operators, logic operators, arithmetic operators, comparing operators, and unary operators. Each operator kind can only be used in its corresponding expression.

```c
<AssignOperator> ::= <ASSIGN> | <PLUS_EQUAL> | <EQUAL_PLUS> | <MINUS_EQUAL> | <EQUAL_MINUS>

<ASSIGN> ::= ':='
<PLUS_EQUAL> ::= '+= '
<EQUAL_PLUS> ::= '=+ '
<MINUS_EQUAL> ::= '-= '
<EQUAL_MINUS> ::= '='

<IncrementDecrementOperator> ::= <INCREMENT> | <DECREMENT>

<INCREMENT> ::= '++'
<DECREMENT> ::= '––'

<LogicOperator> ::= <AND> | <AND_AND> | <OR> | <OR_OR> | <XOR> | <IMPLY> | <EQUIVALENT>

<AND> ::= '&'
<AND_AND> ::= '&&'
<OR> ::= ' | '
<OR_OR> ::= ' || '
<XOR> ::= ' xor '
<IMPLY> ::= '=> '
<EQUIVALENT> ::= '<=>'

arithOperator> ::= <PLUS> | <MINUS> | <TIMES> | <DIVIDE> | <MODULO>

PLUS> ::= ' + '
MINUS> ::= ' − '
TIMES> ::= ' ∗ '
3.3. REAL-TIME STATECHART

<\texttt{DIVIDE}>='/'
<\texttt{MODULO}>'\%'

<\texttt{ComparingOperator}>::=<\texttt{LESS}> | <\texttt{LESS\_OR\_EQUAL}> | <\texttt{EQUAL}> | <\texttt{GREATER\_OR\_EQUAL}> | <\texttt{GREATER}> | <\texttt{UNEQUAL}>

<\texttt{LESS>}::='<'
<\texttt{LESS\_OR\_EQUAL}>::='\leq'
<\texttt{EQUAL}>::='\equiv'
<\texttt{GREATER\_OR\_EQUAL}>::='\geq'
<\texttt{GREATER}>::='>'
<\texttt{UNEQUAL}>::='<>'

<\texttt{UnaryOperator}>::=<\texttt{NOT}> | <\texttt{MINUS}> | <\texttt{PLUS}>

<\texttt{NOT}>::='not'
<\texttt{MINUS}>::='-' 
<\texttt{PLUS}>::='+';

3.3.10.9. Terminal

This section defines the possible terminal symbols.

<\texttt{NUMBER}>::=( '0'..'9')∗( '.' ( '0'..'9') + )?

<\texttt{BOOLEAN}>::=
'true' | 'false'

3.3.10.10. Grammar

This section defines the concrete textual syntax and grammar as EBNF expression.

<\texttt{Block}>::="{"(#expressions+=<ExpressionStartRule>*)}" 
\| #expressions+=<ArithmeticExpression>

<\texttt{ExpressionStartRule}>::= <Assignment> | <ForLoop> | <WhileLoop> | <DoWhileLoop> | <IfStatement> | <ReturnStatement>

<\texttt{ForLoop}>::="for"('(' initializeExpression=<Assignment> 
#loopTest=<Expression>':'
#countingExpression=<ForLoopCountingExpression> ')'
#block=<Block>
<ForLoopCountingExpression> ::=  
  (#lhs_attributeExpression=<AttributeLeafExpression>  
  (#incrementDecrementOperator=<IncrementDecrementOperator>  
  | (#assignOperator=<AssignOperator>  
    rhs_assignExpression=<Expression>))

<WhileLoop> ::= 'while' '(' #loopTest=<Expression> ')' 
  #block=<Block>  

<DoWhileLoop> ::= 'do'  
  #block=<Block>  
  'while' '(' #loopTest=<Expression> ')'  

<IfStatement> ::=  
  'if' '(' #ifCondition=<Expression> ')'  #ifBlock=<Block>  
  ('else' '(' #elseIfConditions+=<Expression> ')'  #elseIfBlocks+=<Block>)*  
  ('else' #elseBlock=<Block>)?

<ReturnStatement> ::=  
  'return' #expression=<Expression> ';'

<Assignment> ::=  
  (#lhs_attributeExpression=<AttributeLeafExpression>  
  ((#assignOperator=<AssignOperator>  
    rhs_assignExpression=<Expression>))  
  | (#incrementDecrementOperator=<IncrementDecrementOperator>) ';

<Expression> ::=  
  <ArithmeticExpression>

<ArithmeticExpression> ::=  
  <ComparisonExpression> ( (#leftExpression=current  
    #operator=<ArithmeticOperator>  
    rightExpression=<ComparisonExpression>)*

<ComparisonExpression> ::=  
  <LogicalExpression>  
  ( (#leftExpression=current  
    #operator=<ComparingOperator>  
    rightExpression=<LogicalExpression>)*

<LogicalExpression> ::=  
  (<UnaryExpression> | <AttributeExpression> )  
  ( (#leftExpression=current  
    #operator=<LogicOperator>  
    rightExpression=<$UnaryExpression> | <AttributeExpression>))*

<UnaryExpression> ::=  
  #unaryOperator=<UnaryOperator>  
  #unaryExpression=<AttributeExpression>

<AttributeExpression> ::=  
  <OperationCall> |
3.3.11. Synchronization

A common use case when modeling orthogonal regions is to allow two regions to change their state only in an atomic way. This means only both transitions or neither are allowed to fire.

Sending and receiving synchronizations via synchronization channels synchronizes the firing of transitions of parallel regions. A synchronization channel has to be specified at a common ancestor state of the parallel regions (cf. Section 3.3.2) and serves as the type for the synchronizations using it.

Sending a synchronization via the synchronization channel from one sender transition to a receiver transition performs a synchronization. The sender transition binds concrete values to the parameters which can be accessed by the action and the raise message events of the receiver transition. We allow only one synchronization, receiving or sending, per transition.

Like Uppaal, we also support synchronization channel arrays, which essentially are ordinary arrays containing synchronization channels [BDL04]. Syntax and semantics of this construct are defined in the same way as for Uppaal: Synchronization channel arrays are defined within a ChannelDefinitionLabel (cf. Sect. 3.3.2) like single channels, but with a number in square brackets appended to the names for specifying the array’s capacity. For example chan c1[10] specifies a synchronization channel array with the name c1 and containing ten synchronization channels. Individual channels of such an array can be addressed within the syncExpr of a transitionLabel (cf. Sect. 3.3.4) by appending in square brackets an expression evaluating to an integer number, indicating the referenced index.

In contrast to older publications, the synchronizing transitions do not get the minimum (absolute) priority of both (cf. [GB03, p. 18]). Instead a synchronization affects the prioritization and execution order of parallel transitions as described in the following.
The sender transition is executed before the receiver transition because a synchronization is directed from sender to receiver. This may violate region priorities when the sender transition is in a region with a lower priority than the region of the receiver transition because without the sending and the receiving of synchronizations between them the transition in the region with the higher priority would be executed first. This special case is shown in the example of Figure 3.22.

![Figure 3.22: Synchronizations May Overrule Execution Order Defined by Region Priorities](image)

It consists of the two parallel regions B and C inside the initial composite state A1, which are synchronized by the use of the synchronization channel sync(). In region B the transition from initial state B1 to state B2 receives a synchronization via the channel sync and assigns the value 1 to variable x. Similarly in region C the transition from initial state C1 to state C2 sends a synchronization via the same channel and assigns the value 2 to x. Without the synchronization and starting with the active state configuration (A1,B1,C1), the transition B1→B2 would be executed before C1→C2 because of the higher priority of region B. This would result in the assignment order x:=1; x:=2; but caused by the synchronization, the sender transition is executed first leading to the assignment order x:=2; x:=1; instead. This results in the final value 1 for x.

The region priority of the sender transition determines the priority of the synchronization. Figure 3.23 gives an example for this behavior. The initial composite state A1 declares the synchronization channels syncBE() and syncCD() and contains the four regions B, C, D and E with descending priorities. In each region there are two states with a transition between them. The transitions have no conditions except for one sending or receiving synchronization at each and thus are all enabled. The transition B1→B2 with the highest region priority of 4 receives and the transition E1→E2 with the lowest region priority of 1 sends synchronizations via syncBE. The transitions with region priorities in between – C1→C2 with region priority of 3 and D1→D2 with region priority of 2 – send respectively receive synchronizations via syncCD. Because the sender transition determines the priority of the synchronization, the
transitions \(C_1 \rightarrow C_2\) and \(D_1 \rightarrow D_2\) fire which results in the final state configuration \((A_1, B_1, C_2, D_2, E_1)\).

We allow only one-to-one synchronizations and in particular no broadcast synchronizations. This means in case of more than one sending and/or receiving transition only pairs of one sender and one receiver transition are executed. The example given in Figure 3.24 illustrates this. It differs from Figure 3.23 insofar as state A1 declares only the single synchronization channel `sync()` which all four transitions use. Because only one-to-one synchronizations are allowed this results in the four different synchronization combinations \(C_1 \rightarrow C_2\) and \(B_1 \rightarrow B_2\) or \(C_1 \rightarrow C_2\) and \(D_1 \rightarrow D_2\) or \(E_1 \rightarrow E_2\) and \(B_1 \rightarrow B_2\) or \(E_1 \rightarrow E_2\) and \(D_1 \rightarrow D_2\). The sender transition with the highest region priority \((C_1 \rightarrow C_2)\) synchronizes with the receiver transition with the highest region priority \((B_1 \rightarrow B_2)\) which results in the final state configuration \((A_1, B_2, C_2, E_1, D_1)\).

A synchronizing transition in a region with higher priority may overrule the outgoing transition priorities in a region with lower priority. This can happen by requiring a transition to be executed which has conflicting outgoing transitions from the same source state but with higher priority. An example for this is shown in Figure 3.25. The initial composite state A1 declares the synchronization channel `sync` and contains the two parallel regions B and C. In region B there exist two outgoing transitions from initial state B1. The one with the higher transition priority targets state B2 and sends a synchronization via the channel `sync`, and the other transition simply targets state B3. Similarly in region C there exist two outgoing transitions from initial state C1. Here the lower prioritized transition to state C3 receives a synchronization over channel `sync` and the other one simply targets state C2. Without the synchronization and starting with the active state configuration \((A, B_1, C_1)\) the transitions \(B_1 \rightarrow B_2\) and \(C_1 \rightarrow C_2\) would be executed because of their respective higher priorities. But caused by the synchronization, the transition \(C_1 \rightarrow C_3\) is executed instead of \(C_1 \rightarrow C_2\). As shown in the example of Figure 3.23, this would not be the case if transition \(B_1 \rightarrow B_2\) is the receiver transition.

The principle that transitions with the higher located source state have higher priority than transitions with a lower located source state (see Section 3.3.4) may be overruled when using synchronizations. A synchronizing transition in a region with higher priority may require a transition to be executed which is inside the source state of another transition, which would normally have the higher priority because of its higher located source state. This case is illustrated in the example shown in Figure 3.26. It differs from Figure 3.25 insofar as it defines a variable \(x\) of type int, which is set by the entry action of state A1 to 1 and contains a different region C. In region C there is only a transition from the initial state C1 to state C2, which increments the value of variable \(x\). C1 is a composite state now which contains region D with the states D1 and D2 and the transition from D1 to D2 which receives a synchronization via channel `sync` and assigns variable \(x\) with value 2. Without sending and receiving synchronizations via channel `sync` and starting with the active state configuration \((A_1, B_1, C_1, D_1)\) the transitions \(B_1 \rightarrow B_2\) and \(C_1 \rightarrow C_2\) would be executed because of their higher priority respectively higher located source states leading to the final value 2 of variable \(x\). But caused by the synchronization the transitions \(B_1 \rightarrow B_2\) and \(D_1 \rightarrow D_2\) are executed first – resulting in the value 2 of variable \(x\) – and afterwards transition \(C_1 \rightarrow C_2\) – resulting in the final value 3 of \(x\).
Figure 3.23.: The Sender Transition Determines the Priority of the Synchronization; so Final State Configuration is \((A_1, B_1, C_2, D_2, E_1)\)

Figure 3.24.: Only One-to-One Synchronizations are Allowed; so Final State Configuration is \((A_1, B_2, C_2, D_1, E_1)\)
3.3. REAL-TIME STATECHART

Figure 3.25.: Synchronization May Overrule Transition Priorities

Figure 3.26.: Synchronization May Violate the Principle that Transitions with Higher Located Source State Overrule Transitions with Lower Located Source States
Constraints

- Each channel has a unique name.

3.3.12. Asynchronous Message Event

We use message events to model asynchronous communication between Real-Time Statecharts. Sender and receiver message interfaces define asynchronous message events (cf. Section 3.2).

A message event has parameters which transfer information from its sender to its receiver. The signature of the message type of the message event defines which parameter the message event has (cf. Section 3.2). The parameters have a call-by-value semantics. The sender transition binds concrete values to the parameters which can be accessed by the action and the raised message event or a send synchronization of the receiver transition.

In Real-Time Statecharts the defined message events of the associated sender message interface can be used as raise message events. A raise message event is a message event which is raised when a transition fires. A raise message event is sent via the associated port of the Real-Time Statechart. This port is connected to another port, which itself has a Real-Time Statechart and a receiver message interface.

In Real-Time Statecharts we use the message events defined within the receiver message interface as trigger message events. A trigger message event is a message event which can enable a transition when it is available and all other required conditions for enabling a transition are true (cf. Section 3.3.4). The message event pool of the Real-Time Statechart stores incoming message events.

When a transition uses a message event to fire, then this message event is dispatched and deleted from the message event pool. The message event pool is a FIFO queue. For each message event only one transition can fire and dispatch the message event. Message events have no specified duration of life. This means, they remain in the event pool until they are dispatched or the event pool is full. The handling of a message event pool overflow is not part of this document and is planned for a future version.

Figure 3.19 shows an example of asynchronous message events used by a transition.

3.3.13. Real-Time Statecharts of Multi-Roles and Multi-Ports

The behavior of multi-roles and multi-ports is defined by Real-Time Statecharts that adhere to a strictly defined structure. The rules that apply to multi-roles and multi-ports are the same. Therefore, we will only define the structure of a Real-Time Statechart that defines the behavior of a multi-role.

As defined in Section 3.1.6, the behavior of a multi-role consists of an adaptation behavior and a sub-role behavior. The sub-role behavior specifies the communication behavior that is executed for interacting with an instance of a single-role. The adaptation behavior is used to
3.3. REAL-TIME STATECHART

resolve ordering constraints between the sub-roles and to specify an adaptation of the sub-roles instances during run-time. The adaptation is defined in [EHH+11] and will be added to future version of this document.

![Real-Time Statechart Diagram]

**Figure 3.27.: Example of a Real-Time Statechart of a Multi-Rol**

Figure 3.27 shows an example of a Real-Time Statechart of the multi-role sender of the Real-Time Coordination Pattern PositionTransmission. The multi-role is ordered and the specified behavior is that the sub-roles subsequently send the position data. Therefore, the adaptation behavior triggers the first sub-role using the synchronization channel send. After the first sub-role has sent the position message, it triggers the next sub-role using, again, the synchronization channel send. The last sub-role synchronizes with the adaptation behavior using the synchronization channel send_done to indicate that all positions have been transmitted.

The Real-Time Statechart in Figure 3.27 outlines the standard structure of a Real-Time Statechart of a multi-role. The Real-Time Statechart of the multi-role, PositionTransmission_sender in the example, contains only one hierarchical state, which is the initial state. That state contains exactly two regions, which are called adaptation and sub-role. The region adaptation contains the adaptation behavior while the region sub-role defines the behavior that is executed by all sub-role instances.

The Real-Time Statechart of the sub-role always defines a variable id of type integer that uniquely identifies the sub-role instance within the scope of the multi-role. The Real-Time Statechart of the multi-role always defines two variables. One variable size of type integer, which contains the currently instantiated number of sub-roles, and one variable subIDs of type integer array, which contains the ids of all instantiated sub-roles. All of these variables are not supposed to be modified by the adaptation Real-Time Statechart or the sub-role Real-Time Statechart. The variables are only to be modified by reconfiguration operations, which will be included in future versions of this document.
The hierarchical state of the multi-role Real-Time Statechart, PositionTransmission_sender in the example, can define an arbitrary number of synchronization channels and synchronization channel arrays. In the example of Figure 3.27, the hierarchical state PositionTransmission_sender defines a synchronization channel send_done and a synchronization channel array send whose size is defined by the number of instantiated sub-roles.

Synchronizations channels and synchronization channel arrays can be used to synchronize the adaptation behavior with the sub-role instances and to synchronize the sub-role instances with each other. The sub-role IDs can be used as indices of a synchronization typed by a synchronization channel array. An example is given by the transition from Idle to Sent in the sub-role Real-Time Statechart. The transition specifies a synchronization send and requires the index id.

In an ordered multi-role, the multi-role Real-Time Statechart additionally defines four operations which are typically used as indices for a synchronization channel array. The four operations are:

- **int first()**
  Returns the ID of the first sub-role instance of the multi-role. If no sub-role is instantiated, the operation returns an invalid ID.

- **int last()**
  Returns the ID of the last sub-role instance of the multi-role. If no sub-role is instantiated, the operation returns an invalid ID.

- **int next(int id)**
  Returns the ID of the sub-role which is located directly after the sub-role with the id given as a parameter. If the id given as a parameter belongs to the last sub-role, the operation returns an invalid ID.

- **int prev(int id)**
  Returns the ID of the sub-role which is located directly before the sub-role with the id given as a parameter. If the id given as a parameter belongs to the first sub-role, the operation returns an invalid ID.

In the example of Figure 3.27, the transition from Waiting to Sending uses the operation first as an index for the synchronization send. As a result, the transition only synchronizes with the first sub-role instance. The upper transition from Sent to Idle in the sub-role Real-Time Statechart uses the operation next with the own ID as a parameter as an index to synchronize via send with the next sub-role. In its guard condition, the transition compares its own ID to the ID returned by the operation last to ensure that it is not the last sub-role.
3.4. Component Model

In MECHATRONIC UML, the system structure is specified by a component model. A component represents a software entity with high cohesion and clearly defined interfaces that it exposes via its ports. The component encapsulates its inner structure and behavior, i.e., it may not be accessed directly by other components [Obj09].

In contrast to the definition of Szyperski [Szy98], we explicitly distinguish between component types and component instances. The component types are instantiated to component instances to specify a concrete system, e.g., for simulation. In the remainder of this document, we will refer to component types simply as components. Instances of a component to be used in a concrete system specification are denoted as component instances. In this section, we will focus on the definition of components while the instantiation of components will be described in the subsequent Section 3.5.

In our component model, we distinguish between atomic components and structured components. Atomic components that will be introduced in Section 3.4.1 contain a stateful behavior specification. Depending on their implementation and purpose, we will distinguish between two kinds of atomic components: discrete atomic components and continuous atomic components. Structured components that will be introduced in Section 3.4.2 are assembled by embedding other components. Therefore, they carry no explicit behavior specification by themselves. Again, we will distinguish between two kinds of structured components: discrete structured components and hybrid structured components.

3.4.1. Atomic Component Type

In this section, we will introduce atomic components which form the lowest level of the MECHATRONIC UML component model.

3.4.1.1. Atomic Component

An atomic component is a component that contains a behavior specification directly. The behavior specification includes a definition of the internal behavior of the component as well as the externally visible behavior that it exposes via its ports. The components of MECHATRONIC UML operate according to the Active Object Pattern [SSRB00], i.e., each atomic component instance is executed concurrently in its own thread.

As mentioned before, we distinguish two kinds of atomic components based on their implementation and purpose. The two kinds are discrete components and continuous components. A discrete component is a software component whose behavior is specified by a discrete component behavior specification (cf. Section 3.4.1.4). As defined in Section 3.1, discrete components interact by means of message passing only. A continuous component represents a feedback (closed-loop) or feed-forward (open-loop) controller [Kil05] of a mechatronic system. In our component model, we only specify the interface of the continuous component, i.e., its ports, but not its internal behavior [BGH+07]. The internal behavior of a con-
tinuous component is assumed to be specified in a control engineering tool like CamelView\textsuperscript{2} or Matlab/Simulink\textsuperscript{3}.

Figure 3.28 shows the concrete syntax of the two kinds of atomic components. They are visualized uniformly as a rectangle with a component icon in the upper right corner and a horizontally left aligned and vertically centered name label. The concrete syntax of ports will be explained in the subsequent Section 3.4.1.2.

An atomic component may define internal attributes that can be used to store data inside a component instance. The attributes are then used by the component behavior specification. It is not possible to access the attributes of a component instance directly from outside the component.

Additionally, a component contains a set of port types, simply denoted as ports, to interact with other components. The ports of a component are visualized on the border of the component and will be defined in the following Section 3.4.1.2.

**Constraints**

- An atomic component is either a discrete component or a continuous component.
- A discrete atomic component requires a discrete component behavior specification.

**3.4.1.2. Port**

A port is a directed, external interaction point of a component. A component may only interact with other components via its ports. On the type level, all possible connections between components are specified by assemblies (cf. Section 3.4.2.5) connecting the ports of the components. The ports are then instantiated to port instances of component instances and may be

\begin{itemize}
\item [\textsuperscript{2}]{http://www.ixtronics.com/ix_hist/English/CAMeLView.htm}
\item [\textsuperscript{3}]{http://www.mathworks.de/}\
\end{itemize}
connected during run-time complying to the assemblies (cf. Section 3.5). Based on the direction, we distinguish in-ports, out-ports and in/out-ports. Information enters the component at an in-port, at an out-port information leaves the component. At in/out-ports, both is possible. Each port of a component has a name which uniquely identifies the port within the respective component.

We distinguish three kinds of ports in our component model, based on the kind of information they process. These are discrete, continuous, and hybrid ports.

A discrete port is used for sending or receiving discrete, asynchronous messages that are typed over a message type (cf. Sections 3.3.12 and 3.2.2). Discrete ports may be in-ports, out-ports and in/out-ports that means they may receive messages, send messages, or both. In a complete MECHATRONIC/UML model, a role of a Real-Time Coordination Pattern must be assigned to all discrete ports in the model.

A continuous port sends or receives signal values. As defined for MATLAB/Simulink “a signal is a time varying quantity that has values at all points in time”\(^4\). A continuous port is either an in-port or an out-port, but never an in/out-port. The type of the signal being processed by a continuous port is defined by a data type. The data type is either Boolean, Short, Int, Long, Float, or Double.

A hybrid port combines the properties of a continuous port and a discrete port. Therefore, it has a data type which specifies which kind of signal it can read or write and it has a discrete port behavior specification (cf. Section 3.4.1.3) which reads or writes this value. Thus, hybrid ports serve as the layer between the continuous world and the discrete world. The behavior of a port is specified by a port behavior specification as described in Section 3.4.1.3.

The concrete syntax of ports is depicted in Figure 3.29. The visualization depends on the kind of the port and the direction of the port. In general, ports are placed on the border of the component such that the center of the port lies on the border line of the component as shown in Figure 3.28. In addition, the name of the port is shown next to the component and positioned outside of the component. For a discrete port, a dashed line represents the role of a Real-Time Coordination Pattern which is assigned to this port. The name of the Real-Time Coordination Pattern and the name of the assigned role are annotated at the dashed line (cf. Figure 3.28).

Discrete and hybrid ports are depicted by squares, whereas continuous ports are depicted by isosceles triangles. Discrete ports embed small isosceles triangles that denote the direction of the port. For an in-port, the top of the triangle points inwards, for an out-port it points outwards. For a continuous port, the triangle "points" into the component for an in-port and out of the component for an out-port.

In our component model, a discrete atomic component may only have discrete and hybrid ports. A continuous atomic component may only have continuous ports.

Discrete ports may have a sender and a receiver interface by means of a message interface (cf. Section 3.2.1). Thus, the direction of a discrete port can be derived from its message interfaces. A sender message interface defines which types of messages may be sent via this port. A receiver message interface defines which types of events may be received via this port.

If a port has only a sender message interface, it is an out-port. If it has only a receiver message interface, it is an in-port. If it has both, it is an in/out-port. A discrete port may only send or receive one message at a particular point in time.

Analogously to roles of a Real-Time Coordination Pattern, discrete ports specify a cardinality with a lower and upper bound. This is similar to the concept of replicated ports in ROOM [SGW94]. For an upper bound of 1, we call it a single-port. For an upper bound greater than 1, we call it a multi-port [EHH+11]. Additionally, we distinguish between mandatory and optional ports based on the lower bound. A port is an optional port if it has a lower bound of 0. A port is a mandatory port if it has a lower bound greater or equal to 1, because in this case at least one instance of the port is mandatory for each instance of the component.

In our concrete syntax (cf. Figure 3.29), we use filled triangles to denote mandatory ports and unfilled triangles to denote optional ports.

The cardinality is given in terms of the Min-Max-Notation [Abr74], i.e., it specifies the number of connections a port may have. For a discrete port, each connection is managed by a sub-port instance of the port during run-time. Then, the lower bound defines the minimum number of sub-port instances that each instance of the port must have. Accordingly, the upper bound defines the maximum number of sub-port instances. For continuous and hybrid ports, the lower bound may be 0 or 1, the upper bound must be 1. A lower bound of 0 specifies that an instance of the port is not always active during run-time, i.e. it does not always send or receive values. In contrast to discrete ports, they have no sub-ports.

The concrete syntax of multi-ports is shown in Figure 3.30. A multi-port has a cascaded double border line and it positioned like a single-port. Again, a filled triangle denotes a mandatory multi-port while an unfilled triangle denotes an optional multi-port.

The structure and the behavior specification of a multi-port is analogous to multi-roles of a Real-Time Coordination Pattern (cf. Section 3.1.6). The behavior specification of a multi-port consists of an adaptation behavior and a sub-port behavior [EHH+11]. The adaptation behavior controls the creation and deletion of the sub-port instances at run-time and is responsible for resolving dependencies between the sub-port instances. The sub-port behavior defines the behavior of the sub-port instances. All sub-port instances share the same behavior specifica-
tion as defined in Section 3.4.1.3. The reconfiguration, i.e. creation and deletion of sub-port instances, is defined in [EHH+11] and will be added in a future version of this document.

As mentioned before, a developer needs to assign a role of a Real-Time Coordination Pattern to each discrete port of a component. Such assignment is only allowed if both, the role and the port, have exactly the same message interfaces. In addition, a single-role of a Real-Time Coordination Pattern may only be assigned to a single-port while a multi-role is typically assigned to a multi-port. In case of a multi-port, the cardinality of the multi-port must either be equal to the cardinality of the multi-role or it must restrict the the cardinality. As a special case, the port may restrict the cardinality to an upper bound of 1 which, in essence, allows that multi-roles are assigned to single-ports. The lower bound of the cardinality may be relaxed from 1 to 0 if the port must not be present in all scenarios and if, thus, the corresponding Real-Time Coordination Pattern is not always instantiated.

Additionally, ports may define local attributes that are used to store data within a port instance. The attributes are disjunct from the attributes defined by the component and may only be accessed by the port behavior specification. They may not be accessed by the discrete component behavior specification directly.

**Constraints**

- A discrete port may only be contained in a discrete component.
- A hybrid port may only be contained in a discrete component.
- A continuous port may only be contained in a continuous component.
- A discrete port has at least one interface.
- A discrete port has a role of a Real-Time Coordination Pattern which is assigned to it. A single-role may only be assigned to a single-port.
- A discrete port and the role assigned to the discrete port have the same message interfaces.
• The lower bound of the cardinality is greater or equal to 0 and less or equal to the upper bound. The upper bound is greater or equal to 1.

• There do not exist two ports with the same name in a component.

• A discrete port and a hybrid port have a port behavior specification.

3.4.1.3. Port Behavior Specification

The port behavior specification specifies the run-time behavior of a port. In this section we describe the behavior specification for all three kinds of ports.

Discrete Port Behavior Specification

The behavior of a discrete port is specified by means of a Real-Time Statechart. In case of a discrete single-port, the behavior is defined by a single Real-Time Statechart. In case of a discrete multi-port, the behavior is defined by a Real-Time Statechart which follows the rules introduced in Section 3.3.13.

The Real-Time Statechart of a discrete port may be obtained in three ways. Firstly, a role of a Real-Time Coordination Pattern may be assigned to the port by a developer. Then, the port behavior must guarantee to behave according to the behavior of the assigned role. This can be achieved by copying the Real-Time Statechart of the role to the port and refine it afterwards [HH11]. Secondly, if no suitable Real-Time Coordination Pattern exists for the intended communication, a Real-Time Statechart for the port may be specified directly. Then, this Real-Time Statechart has to be abstracted by the developer to a role of a Real-Time Coordination Pattern before connecting the port to a port of another component in a structured component or in a component instance specification. The abstraction removes all elements from the Real-Time Statechart that are implementation specific for the port like, e.g., synchronizations with the internal component behavior (cf. Section 3.4.1.4). Thirdly, both, the Real-Time Statechart of the port and the role of a Real-Time Coordination Pattern may exist. In all three cases, the Real-Time Statechart of the port has to be checked for a correct refinement of the role behavior [HH11]. If the check is successful, the port is said to implement to role behavior.

Essentially, a port is a correct refinement of a role if it neither adds nor removes externally visible behavior. The externally visible behavior consists of the messages that are sent and received via the respective port and the time intervals in which they are sent and received. Thus, it is allowed to add states, transitions, and actions which specify internal behavior like implementation specific computations or synchronizations. Furthermore, the refinement definition introduced in [HH11] exploits the fact that ports may buffer incoming messages until they are processed by the Real-Time Statechart of the port. That enables to delay the reception of messages, but requires an alternating sequence of sent and received messages.

Within the Real-Time Statechart of the port, only asynchronous events that are typed over the message types declared in the message interfaces of the port may be used. Asynchronous events may only be used for interaction with another component and they may not be used for interaction with the internal behavior of the component (cf. Section 3.4.1.4). Then, messages
typed over message types declared in the receiver interface may only occur in trigger events of the Real-Time Statechart. Accordingly, messages typed over message types declared in the sender interface may only occur as raised events (cf. Section 3.3.12).

The Real-Time Statechart of the port may access the attributes that are defined within the port. The attributes may be used for transition guards or changed by side effects of the transitions. The port may not access the attributes that are defined by the component.

In addition, a port may define operations that implement the actions of the states and the side effects of the transitions (cf. Section 3.3.7). The operations (cf. Section 3.3.9) are specified by means of story diagrams [FNTZ00] that combine UML Activity Diagrams [Gro10b] and graph rewrite rules [Roz97] or our action language (cf. Section 3.3.10). The port may not directly call operations of the component.

In addition to asynchronous events, the port Real-Time Statechart may use synchronizations as defined in Section 3.3.11 to interact with the internal behavior of the component (cf. Section 3.4.1.4). The port Real-Time Statechart may only use synchronizations that are typed over synchronization channels which are defined by the developer within the component behavior specification of the containing component (cf. Section 3.3.11). Such synchronizations are used to exchange data with the internal component behavior and to resolve dependencies between different port Real-Time Statecharts.

Continuous Port Behavior Specification  A continuous port represents a signal value which “is a time varying quantity that has values at all points in time”\(^5\). As such, a continuous port only specifies a data type for the signal, but no behavior specification. The behavior which controls the signal is implemented as part of a continuous component whose behavior is not considered to be part of a MECHATRONIC UML specification (cf. Section 3.4.1.4).

Hybrid Port Behavior Specification  A hybrid port combines the properties of discrete and continuous ports. Its behavior is defined by a Real-Time Statechart, but in contrast to discrete ports, a hybrid port does not send or receive asynchronous messages. Instead, a hybrid port reads or writes a signal value in its Real-Time Statechart. Therefore, the Real-Time Statechart has an implicit additional variable that represents the signal value. The name of that variable is the name of the port. If the hybrid port is an in-port, the respective variable is read-only. If it is an out-port, the variable may also be written. Figure 3.31 shows an example.

Figure 3.31 shows the behavior specification of the hybrid ports speed_right and position of the atomic component Navigation. As for discrete ports, the behavior specification is given in terms of a Real-Time Statechart. The Real-Time Statechart of speed_right uses the variable speed_right to access the signal value and position uses the variable position accordingly.

In order to support model checking of hybrid ports, the variable may only change in predefined intervals. If we omitted this restriction, we would have to consider defining the changes of the signal value. These changes of a signal value are typically defined by differential equations. A verification of a model including differential equations is called a hybrid model check-

CHAPTER 3. MODELING LANGUAGES

Atomic Component Navigation

Port Behavior Specification
for speed_right and position

Figure 3.31.: Example of the Behavior Specification of Hybrid Ports

3.4.1.4. Discrete Component Behavior Specification

The behavior specification of a discrete atomic component is given by means of a Real-Time Statechart. The MECHATRONICUML process offers two possibilities to obtain that component behavior specification. Both possibilities consider the port Real-Time Statechart as the implementation of the port behavior, i.e., the port behavior is a part of the component behavior.

The first possibility for obtaining the component behavior is using the synthesis approach introduced in [EH10]. The synthesis composes all port Real-Time Statecharts into a single Real-Time Statechart for the component. Additionally, a set of state restrictions may be specified that forbids certain combinations of states of the synthesized Real-Time Statecharts. If the synthesis algorithm successfully generates a Real-Time Statechart for the component, this Real-Time Statechart is a correct refinement of all port Real-Time Statecharts by construction. If no such Real-Time Statechart exists, the synthesis fails and does not return a Real-Time Statechart for the component. This approach can only be applied if the whole behavior of the component is specified within its ports and if no data has to be exchanged between the ports of the component.

The second possibility for obtaining the component behavior is the manual, but structured creation of a Real-Time Statechart for the component. This approach can be applied in any situation. Then, the Real-Time Statechart of the component consists of one state only that contains a set of regions and declares a set of synchronization channels that may be used for communication inside the component. Especially, synchronizations typed over these synchronization channels have to be used to exchange data between the ports and the optional internal behavior of the component by using the parameters of the synchronizations (cf. Sec-
The synchronizations must not be used for communication between different component instances.

Figure 3.32 shows an example of the component behavior specification of the component Navigation. The regions for the state of the component Real-Time Statechart are constructed as follows. For each discrete single-port and each hybrid port, there exists one region containing the Real-Time Statechart of the port. For each discrete multi-port, there exist one region containing a Real-Time Statechart which in turn has one state with two regions. These two regions contain a type definition of the Real-Time Statechart of the sub-ports and the adaptation Real-Time Statechart. The Real-Time Statechart for the sub-port will be instantiated once for each sub-port instance thereby resolving the parameters of the Real-Time Statechart (cf. Section 5.1.3). This instantiation is performed by the reconfiguration which will be explained in a future version of this document. Additionally, the state may contain arbitrarily many so-called synchronization statecharts. These synchronization statecharts represent the internal behavior of the component.

![Diagram of component behavior specification](image)

Figure 3.32.: Structure of the Statechart of the Component Behavior of an Atomic Component

The compositional verification approach described in [GTB+03] requires to restrict synchronization within an atomic component. The compositional verification approach verifies the Real-Time Coordination Patterns and the atomic components separately from each other. In order to obtain valid verification results, the dependencies between the different Real-Time Coordination Patterns must be resolved by the synchronization statechart in a structured way.

For a single-port, synchronization is only allowed between the port statechart and the synchronization statecharts of the component behavior specification. For a multi-port, synchro-
nizations are allowed between the adaptation statechart and the sub-port statechart. Since a multi-port may be instantiated multiple times, there exist multiple instances of the sub-port Real-Time Statechart. Then, synchronization is also allowed between the sub-port instances. The adaptation statechart may synchronize with the synchronization statecharts and all synchronization statecharts may synchronize with each other. All other synchronization between the different regions are not allowed.

The synchronization statecharts may access the attributes that are defined within the component, but they must not access the attributes of the ports directly. The attributes may be used for storing data and for specifying guards of transitions. Additionally, the component may specify a set of operations that implement the actions of the states and the side effects of the transitions (cf. Section 3.3.7). The synchronization statechart may not call operations defined in the ports.

3.4.2. Structured Component Type

This section describes the specification of structured components that introduce the modeling of hierarchical components into MECHATRONICUML.

3.4.2.1. Structured Component

A structured component is assembled by embedding other components by means of so-called component parts. A component part is either typed over an atomic component or an other structured component. The behavior of the structured component is then defined by the component behavior specifications of the component parts. Therefore, the structured component does not contain a component behavior specification itself.

For a structured component, the MECHATRONICUML component model only supports two kinds. These are discrete structured components and hybrid structured components. Structured continuous components are not supported. Whether a structured component is discrete or hybrid depends on the types of the embedded component parts (cf. Section 3.4.2.3). A structured component is a discrete structured component if all of its component parts are typed over discrete atomic components or discrete structured components. A structured component is a hybrid structured component if at least one of its component parts is typed over a continuous atomic component or a hybrid structured component.

3.4.2.2. Port

A structured component specifies a set of ports as well, but in contrast to an atomic component, the ports of the structured component do not contain a behavior specification. The ports of the structured component are delegated to ports of embedded component parts where they are implemented (cf. Section 3.4.2.4 on Delegations). In case of a discrete port, the port has an assigned role of a Real-Time Coordination Pattern in order to connect it to another component by using a Real-Time Coordination Pattern. The rules for assigning a role of a Real-Time
Coordination Pattern to a discrete port of a structured component are the same as the rules for atomic components which are introduced in Section 3.4.1.2.

A structured component may only define discrete and continuous ports, but no hybrid ports. Hybrid ports are not allowed for structured components because a hybrid port defines that a signal provided by a continuous port is discretized at a fixed rate (cf. Section 3.4.1.3). For the interface of the structured component, however, it is only necessary to specify that a signal value is expected or provided. Whether this value is discretized or processed by a continuous atomic component is part of the inner behavior of the component and does not need to be specified in it’s interface.

**Constraints**

- A discrete structured component may only contain discrete ports.
- A hybrid structured component may contain discrete ports and continuous ports.

### 3.4.2.3. Component Part

A component part is a representation of a component that is embedded into a structured component. It describes the potentially multiple occurrences of a component in a structured component type. Thus, component parts are defined on the type level as well and must be typed over a component (either atomic or structured). The definition of structured components by using component parts corresponds to the definition of structured classifiers in the UML [Gro10b]. Accordingly, the component parts define a type system for the contents of the structured component.

In analogy to the UML, a component part specifies a cardinality with a lower bound and an upper bound. This is also similar to the concept of replicated actors in ROOM [SGW94]. The lower bound specifies the minimum number of instances of this part that must be present in any instance of the structured component. Accordingly, the upper bound specifies a maximum number that may be instantiated. For an upper bound equal to 1, we call it a *single-part*. For an upper bound greater than 1, we call it a *multi-part*. Thus, the cardinality restricts the number of instances of a component part that may occur in a structured component during run-time.

In a structured component, multiple component parts may be typed over the same component. They are distinguished by the name of the component part. The names of all component parts must be unique within a structured component.

Figure 3.33 shows an example for the concrete syntax of a structured component with name BeBot embedding three component parts which are typed over the atomic components EngineCtrl and PositionSensor and a structured component BeBot_SW. Thus, BeBot is a hybrid structured component. The ports of component parts, that are not yet connected by delegations or assemblies, use the same concrete syntax that is used for ports of an atomic component (cf. Figure 3.28).

The structured component is visualized as a rectangle with two horizontal compartments. The upper compartment contains the left-aligned name of the component and the right-aligned
component icon. The lower compartment contains the embedded component parts. An embedded component part is visualized similar to an atomic component. The component part is labeled with the name of the component part and the name of the component it is typed over. Additionally, the label contains the cardinality of the component part. The label of a component part is defined by the following grammar:

\[
\text{<partLabel> ::= #componentPart.name [componentName] <cardinality>}
\]

\[
\text{<componentName> ::= #componentPart.componentType.name}
\]

\[
\text{<cardinality> ::= ' [ ' #componentPart.lowerBound ' . . ' #componentPart.upperBound ' ] '}
\]

If the lower bound equals the upper bound, we allow for an abbreviation that only indicates the upper bound as shown in Figure 3.33 where for all three component parts the lower bound and the upper bound are equal to 1. If the upper bound is not specified, it is indicated by an asterisk as shown in Figure 3.34.

Analogously to multi-ports, a multi-part is visualized by a cascaded double border line as shown in Figure 3.4.

**Constraints**

- A structured component \(A\) must not embed a component part that is typed over a structured component that (directly or indirectly) embeds a component part typed over \(A\), i.e., embedding must not introduce cycles in the component hierarchy.

- The lower bound of the cardinality is greater or equal to 0 and less or equal to the upper bound. The upper bound is greater or equal to 1.

- The names of all component parts are unique within a structured component.
3.4. COMPONENT MODEL

3.4.2.4. Delegation

The ports of a structured component do not have a behavior specification (cf. Section 3.4.2.2). Instead, the port of the structured component is delegated to a port of a component part. Then, the component part implements the respective behavior of the port of the structured component.

Figure 3.35 shows the structured component BeBot_SW that has five ports that are delegated to two embedded component parts nav and bbo of types Navigation and BeBot_Observer, respectively. The delegation link is represented by a solid line between the port of the structured component and the port of the part. If a port of a component part is connected to a port of the structured component by a delegation, then it does no longer visualize the role it implements. Instead, the implemented role is visualized by the port of the structured component using the same concrete syntax.

Whether a delegation link may be created depends on two conditions. First, they have to be structurally compatible according to Figure 3.36. Second, they must have compatible interfaces. If both conditions are fulfilled, the delegation link may be created. The two conditions will be explained in the following.

The structurally compatible combinations of single-ports of a structured component and a component part are summarized in Figure 3.36. If a combination is marked with a checkmark, it is possible, otherwise it is not possible.

For discrete ports, a structurally compatible combination also has to consider the cardinalities. A single-port may only be delegated to a single-port of a single-part. Multi-ports may be delegated to three constructs. First, a multi-port may be delegated to a multi-port of a single-part. Second, it may be delegated to a single-port of a multi-part where the multi-part. Third, a multi-port may be delegated to a multi-port of a multi-part. Figure 3.37 summarizes the possible combinations. The semantics of these combinations will be defined along with the reconfiguration operations in a future version of this document.

As stated before, the second condition for creating a delegation link is interface compatibility. Since discrete ports are required to have a role of a Real-Time Coordination Pattern as-
CHAPTER 3. MODELING LANGUAGES

Figure 3.35.: Concrete Syntax of a Delegation

Figure 3.36.: Structurally Possible Delegation Links
signed, the interface compatibility is fulfilled for them if both ports are assigned the same role of the same Real-Time Coordination Pattern. For continuous and hybrid ports, the interface compatibility is fulfilled if both ports specify the same data type for the signal.

Since the component parts of the structured component define a type system for the inner structure of the component, a port of a structured component may be delegated to ports of several component parts. Then, the delegations define all embedded component parts which may implement the port at some point in time during run-time. The concrete component part implementing the port of the structured component may be changed by a reconfiguration operation during run-time, but a port instance of a structured component may only be delegated to one port instance of a component part instance at a time. Such reconfiguration operations will be defined in future versions of this document.

**Constraints**

- Ports that are connected by a delegation have the same direction (in or out or in/out).

- Continuous ports and hybrid ports that are connected by a delegation have the same type.

- Discrete ports that are connected by a delegation refine the same role of the same Real-Time Coordination Pattern.

- A discrete multi-port may not be connected by a delegation to a single-port of a single-part.
3.4.2.5. Assembly

In many cases, the component parts embedded in a structured component have to interact to implement the functionality of the structured component. They interact by communicating over their ports which have a role of a Real-Time Coordination Pattern assigned to it. Then, the communication behavior is defined by the corresponding Real-Time Coordination Pattern. The connection between the ports are specified by assembly connectors.

Figure 3.38 shows the complete definition of the structured component BeBot_SW. The assembly connectors are visualized as solid lines between the ports of the component parts. In case of an assembly between discrete ports, the Real-Time Coordination Pattern defining the communication behavior is visualized for the assembly. In the example, the multi-port of the component part nav : Navigation is connected to the component parts exp : Exploration and bbo : BeBot Observer which means that the component part nav : Navigation may communicate with the component part exp : Exploration or the component part bbo : BeBot Observer or both.

![Figure 3.38. Complete Example of a Discrete Structured Component including several Assemblies connecting the Component Parts](image)

Figure 3.39 shows the complete definition of the structured component BeBot. It embeds the discrete structured component BeBot_SW of Figure 3.38 as a component part. Additionally, it
embeds two component parts typed over the continuous components EngineCtrl and PosData. Inside the component BeBot, the continuous ports of BeBot_SW are connected by assemblies to the continuous component parts.

Figure 3.39.: Example of a Hybrid Structured Component

Whether an assembly connector may be created between two ports of embedded component parts depends on two conditions. First, the ports have to be structurally compatible, i.e., they must be of the same kind and have inverse directions as defined in Figure 3.40. Second, they must have compatible interfaces. If both conditions are fulfilled, the assembly link may be created. The two conditions will be explained in the following.

The structurally possible combinations of ports of component parts are summarized in Figure 3.40. If a combination is marked with a checkmark, it is allowed, otherwise it is not allowed. In general, an in-port may only be connected to an out-port and vice versa such that information may flow at run-time. In/out-ports may only be connected to other in/out-ports.

As stated before, the second condition for creating an assembly connector is interface compatibility. For discrete ports, the interface compatibility is fulfilled if both ports are assigned different roles of the same Real-Time Coordination Pattern. If an assembly is created according to a Real-Time Coordination Pattern, we call it a Real-Time Coordination Pattern occurrence. For continuous ports, the interface compatibility is fulfilled if both ports have the same data type.
In the example, an assembly from port navigator of the component part Exploration to the port provider of the component part Navigation is possible. This is because both ports are discrete in/out-ports and they are assigned different roles of the Real-Time Coordination Pattern Navigation. Then, we have an occurrence of the Real-Time Coordination Pattern Navigation which is visualized by connecting the pattern ellipse to the corresponding ports by dashed lines. The dashed lines are labeled with the name of the role. The assembly from port speed_left of the component part BeBot_SW to the port speed_in of the component part EngineCtrl is possible because both have the data type Integer.

Since the component parts of the structured component define a type system for the inner structure of the component, several assembly connectors may be specified for a port of a component part. Then, the assemblies define all possible assembly instances that may be created inside a structured component instance. Intuitively, assembly instances may only be created between component parts that are instantiated, i.e., if no instance for Collision_Control of Figure 3.38 exists, no assembly instance may be created for an assembly ending at a port of Collision_Control (cf. Section 5.5.1). The concrete assembly instances are created by reconfiguration operations during run-time. Such reconfiguration operations will be defined in future versions of this document.

If one role of a Real-Time Coordination Pattern is a multi-role, an occurrence of a Real-Time Coordination Pattern may involve several assembly connectors. In our example, the role sender of the Real-Time Coordination Pattern PositionTransmission is a multi-role. The role receiver is assigned to the ports receiver of BeBot_Observer and Exploration, respectively.
As visualized in Figure 3.38, an occurrence of PositionTransmission involves two assembly connectors.

In our example, the components of type BeBot represent the top-most level of the modeled component architecture. As shown in Figure 3.39, the BeBot offers the ports distributor and client which are used in our example to connect to other BeBots. On this system level, we do not specify assembly connectors because the usage of the modeled BeBot in a larger system is not part of the modeling in this case. Instead, components on the system level may be connected with respect to the coordination patterns that they offer on their ports and the respective cardinalities of the ports. In our example, a BeBot may be connected to any component that also implements the Distribution pattern.

**Constraints**

- An assembly may not connect two ports of the same single-part.

- If two discrete ports are connected by an assembly, they must implement different roles of the same Real-Time Coordination Pattern or, in case of a pattern with only one role, they must both implement the role of the same Real-Time Coordination Pattern. As a result, both ports need to specify compatible message interfaces.

- An assembly between two ports requires one port to be an in-port and one port to be an out-port or both ports to be in/out-ports.
3.5. Component Instance Configuration

A component instance configuration is a design-time specification of a concrete instantiated system under construction. It contains a set of component instances that are typed over the components specified in the MECHATRONICUML component model. Component instances are connected by assembly instances for assembling a concrete system. Modelers leverage component instance configuration specifications for specifying the deployment of the system (cf. Sect. 3.6).

3.5.1. Component Instance

A component instance is derived from an atomic component type or a structured component type. During instantiation, the variable parts of the component type are determined. The variable parts include the port instances derived from port types (cf. Section 3.4.1.2) and embedded component instances derived from component parts (cf. Section 3.4.2.3) of a structured component type.

During instantiation, port instances are derived from the set of port types of the component type. This means, the port types which are actually instantiated by the component instance and the number of sub-ports of the multi-ports are determined. The actual number of port instances, however, must comply to the cardinality of the respective port type.

In case of a structured component, a component instance is created for each component part. The type of the resulting embedded component instance is defined by both, the component part as well as the component type referenced by the component part. That allows to distinguish embedded component instances based on the component parts. We will exploit that information for reconfiguration of structured component instances in future versions of this document. A component instance which is not embedded in a structured component instance is only typed by a component type and does not refer to a component part.

In a structured component instance, several instances of one part depending on the component part’s cardinality may exist. By default, initial instances of multi-parts and multi-ports are created with lowest cardinality.

Figure 3.41 shows the concrete syntax of the component instance b1 of the component type BeBot_SW. A component instance basically has the same concrete syntax as a component type. In contrast to component types, component instances have a name that begins with a lower case letter. This name is followed by a colon and the name of the component type that this component instance is derived from. The concrete syntax of port instances is the same as for port types.

The component instance b1 of Figure 3.41 is typed by the component type BeBot_SW and the component part bsw of the structured component type BeBot of Figure 3.33. It has the three continuous port instances speed_left, speed_right, and position. The instance of the multi-port distributor of the component type BeBot_SW (e.g. Fig. 3.38) has two sub-port instances distributor1 and distributor2 that implement this multi-role.
3.5. COMPONENT INSTANCE CONFIGURATION

Figure 3.41.: Concrete Syntax of a Component Instance of the Component Type BeBot_SW

The concrete syntax of a component instance is derived from the concrete syntax of components. In contrast to a component, the label of a component instance is underlined and defined by the following grammar.

\[<\text{instanceLabel}> ::= \#\text{componentInstance}\text{.name} [\text{/} <\text{partName}>] \text{' : '}<\text{componentName}>\]
\[<\text{partName}> ::= \#\text{componentInstance}\text{.componentPart}\text{.name}\]
\[<\text{componentName}> ::= \#\text{componentPart}\text{.componentType}\text{.name}\]

The label displays the name of the component instance. If the component instance is embedded in a structured component instance, the name is succeeded by a slash and the name of the corresponding component part. The last part of the label always displays a colon and the name of the component type.

Figure 3.42 shows the structure of a multi-port instance. At run-time, there exists exactly one instance of the multi-port which executes the adaptation behavior. The multi-port instance has a varying number of sub-port instances, each of them has one connector instance and is responsible for communication over this connector instance. Since we only support 1:n communication in the current version of MECHATRONICUML, it follows that every sub-port instance is connected to a single-port instance by a connector instance.

3.5.2. Component Instance Configuration

A component instance configuration is obtained by connecting component instances via connector instances. Component instances of the same hierarchy level may be connected with each other by means of assembly connector instances. If an assembly connector instance is created between two discrete port instances, the corresponding Real-Time Coordination Pattern is instantiated as well which results in a Real-Time Coordination Pattern instance (cf. Section 3.1.5). Delegation connector instances connect port instances of adjacent hierarchy levels. In the component instance configuration only the current level of structured component instances is shown, i.e., all embedded component instances are hidden.
Figure 3.42.: Structure of a Multi-Port Instance being Connected to Several Single-Port Instances

Figure 3.43 shows the component instances bebot1:BeBot, bebot2:BeBot, and bebot3:BeBot that are typed over the component type BeBot of Figure 3.33. Since the component instances of type BeBot are not embedded in a structured component instance, their label only indicates the instance name and the name of the component type. bebot2 and bebot3 are connected to bebot1 by an assembly connector instance. They are connected by the instance dist of the Real-Time Coordination Pattern Distribution (cf. Section 3.1 and Figure 5.1) where the discrete port instances distributor1 and distributor2 of bebot1 implement the multi-role distributor and the two ports named client of bebot2 and bebot3 implement the role client. The component instances and assembly connector instances form a component instance configuration.

Figure 3.43.: Concrete Syntax of a Component Instance Configuration

The component instance configuration shown in Figure 3.43 shows the highest hierarchy level of the system. Figure 3.44 shows the component instance configuration that specifies the inherent part structure of the BeBot_SW component instance of bebot1. The component instances nav:Navigation, pos:Position, and bbo:Bebot_Observer are connected to the port
instances of the next higher level component instance b1:Bebot_SW by delegation connector instances.

Figure 3.44.: Concrete Syntax of a Component Instance Configuration

Constraints

- At most one DelegationInstance per PortInstance is allowed.
3.6. Deployment

The software specified by the models introduced so far need hardware for their execution. Software can be executed on the same hardware or it can be distributed over several hardware entities. A deployment specifies the allocation of software atomic components to hardware nodes. Further, software component ports are mapped to hardware ports of the hardware nodes where the corresponding software component is allocated.

3.6.1. Hardware Nodes

A deployment consists of a component instance configuration, a set of hardware nodes, communication links, and quality of service aspects. A hardware node represents a physical entity, e.g. a computational, sensor, actor, or communication device. On computational hardware nodes component instances may be deployed. Hardware nodes have hardware ports as interfaces. Communication links connect hardware ports to communicate with each other. A communication link fulfills a quality of service, which is specified by a latency and packet delay variation time.

Figure 3.45 shows a hardware node that models the physical BeBot_ECU_1. It is visualized by a 3D-box. It has a name that follows the syntax of a component instance as described above. The hardware ports are visualized by squares that contain either the letter “i”, “o”, or “i/o”. “i” indicates that the port gets incoming data, “o” stands for outgoing data, and “i/o” for bidirectional data.

Figure 3.45.: Concrete Syntax of a Hardware Node

3.6.2. Allocation

Hardware nodes have a reference to component instances, which should be allocated to the node. Hardware ports have also a references to port instances which should be mapped to the
hardware ports. The dashed line shows which component port is mapped to which hardware port. Figure 3.46 shows the deployment of the component instance b1:BeBot on the hardware node BeBot_ECU1.

![Deployment Diagram](image)

Figure 3.46.: Deployment of a Component Instance to a Hardware Node

The above deployment model is our first step into this direction and is planned to be further refined in subsequent versions of this document.
Chapter 4.
Development Process

This chapter introduces the MECHATRONIC UML software development process. The software of a mechatronic system is strongly interconnected with components of other involved disciplines such as mechanical engineering, electrical engineering, and control engineering. Therefore, MECHATRONIC UML is integrated into an overall development process that coordinates the development among the different disciplines.

Schäfer et al. present a first coarse-grained description of the MECHATRONIC UML process [SSGR11]. Becker et al. refine this coarse-grained process and introduce a first fine-grained variant of the MECHATRONIC UML process [BDG+11]. In this chapter we present a further improved variant of the fine-grained MECHATRONIC UML process. We derived most improvements from the evaluation that Sander and Bröker performed [San11, Brö11].

The goal of this process definition is to explain the MECHATRONIC UML process in a human readable form that is as unambiguous as possible. The goal is not, however, to define an automateable process. UML activity diagrams are, therefore, used to describe the MECHATRONIC UML process.

This chapter is structured as follows: First, the integration of MECHATRONIC UML within the overall development process is explained in Section 4.1. Second, the MECHATRONIC UML process is described in detail in Section 4.2.

4.1. Process for Self-Optimizing Mechatronic Systems

Most development processes for mechatronic systems follow a variant of the V-Model such as described by the VDI 2206 ”Design Methodology for Mechatronic” [VDI04]. The VDI 2206 defines a joint development process, a joint modeling formalism, and a joint use of tools across the different disciplines that are required for the development of mechatronic systems. However, the approach is only very coarse-grained and does, therefore, not sufficiently address the collaboration among the different disciplines.

In an ongoing effort within the Collaborative Research Center 614 (CRC 614), an interdisciplinary, large-scale research project at the University of Paderborn, we have refined the V-Model of the VDI 2206 to improve the collaboration throughout the development of self-optimizing mechatronic systems [GFDK09]. The macro structure of the development process of the CRC 614 consists of three phases: the interdisciplinary conceptual design, the parallel
**discipline-specific development**, and the **system integration**. The first two phases are shown in Figure 4.1. During the interdisciplinary conceptual design, a team of interdisciplinary experts specifies an initial system model with the **CONceptual design Specification technique for the ENgineering of complex Systems (CONSENS)** which captures all interdisciplinary relevant aspects of the system. It includes the system’s **requirements**, the **active structure** that describes the system’s logical and physical structure, its spatial properties (shape), and its behavior [GFDK09].

In detail, the active structure consists of **system elements** that are similar to components in a UML component diagram. But, in contrast to UML components, system elements can be connected to each other by three different kinds of flow relations, namely energy flow, material flow, and information flow.

![Figure 4.1: The Integration of the MECHATRONICUML Process into the Development Process of Self-Optimizing Mechatronic Systems](image)

Figure 4.1.: The Integration of the MECHATRONICUML Process into the Development Process of Self-Optimizing Mechatronic Systems
4.2. The MECHATRONICUML Process

The behavior can be modeled by behavior–activity, behavior–state and behavior–sequence models that are similar to UML state machines, UML activity diagrams, and UML sequence diagrams [Gro10b]. Additionally, characteristic scenarios of the system’s interaction with the environment are described by application scenarios. An application scenario consists of a textual description that explains the scenario and the system’s reaction, the system elements of the active structure that are relevant for the scenario, and the behavior for the communication of those system elements.

This informal description of the application scenarios are likely to contain contradictions and inconsistencies. During the step “formal use case specification”, the interaction of the components and the interactions’ real-time constraints are specified by Modal Sequence Diagrams [HM07], a formal variant of UML sequence diagrams [Obj09]. In the same step the set of Modal Sequence Diagrams is analysed to find the inconsistencies and contradictions. For the analysis, the synthesis or simulation approach of Greenyer is applied [Gre11]. These two steps are performed iteratively until the set of Modal Sequence Diagrams is free of contradictions and inconsistencies. The set of Modal Sequence Diagrams is then added to the overall system model.

The overall result of the conceptual design is the principle solution, a milestone of the overall system model that forms a common basis for the subsequent parallel development of the discipline-specific parts of the system within the discipline-specific development phase.

In the discipline-specific development phase all disciplines detail the system in a parallel development process by using their specific methods, formalisms and tools. Various dependencies between the processes and the models usually exist in this phase that might result in an inconsistent overall system model. Therefore, the parallel processes are coordinated and synchronization techniques are used such that model-inconsistencies can be prevented. In the last phase, the results from all disciplines are integrated into an overall consistent system model.

The MECHATRONICUML process is a vital part of the overall software development process during the discipline-specific development phase. As the principle solution is the result of the former conceptual design phase, the partial models of the principle solution form the basis of the MECHATRONICUML approach. In particular, the active structure and the behavior models of the principle solution are the relevant initial models for the software development.

4.2. The MECHATRONICUML Process

Figure 4.2 shows a diagram of the overall MECHATRONICUML process. Based on the active structure, the set of the application scenarios’ Modal Sequence Diagrams, and the behavior – state from the overall system model, the software of the system is developed using MECHATRONICUML in 8 major steps and three steps starting an iteration (Steps 9, 10 and 11).

MECHATRONICUML follows a top-down approach: the initial component model is derived from the active structure in the principle solution during Step 1. In a later step (Step 4) this component model may be refined.
CHAPTER 4. DEVELOPMENT PROCESS

1. Derive initial component model
2. Derive requirements for each communication
3. Determine coordination patterns
4. Set of modal sequence diagrams for application scenarios
5. Set of constraints
6. Set of coordination patterns
7. Generate code
8. Compile code
9. Redesign formal use case specification
10. Adapt component model
11. Redesign by other disciplines

Figure 4.2.: Overall MehATRONICUML Process
Step 1 starts with identifying the system elements, that are relevant for the software development. For each relevant system element a component is added to the component model. A component can be a **structured component** or an **atomic component**. A structured component consist of parts that are typed by other components (cf. Section 3.4.2.1). An atomic component has a behavior specification and cannot embed any parts (cf. Section 3.4.1).

In the active structure, system elements may form a hierarchical structure of further system elements. Each further decomposed system element is represented by a structured component in the component model. Inner system elements are transformed to parts. These parts are typed by those components that represent the component type for the corresponding inner system element. Afterwards, the information flow between all relevant system elements is transformed to connectors in the component model. This step can also be performed in a semi-automatic way [GSG+09]. If necessary, this model can be extended by further components and connectors. The result of this step is the initial version of the component model.

Based on the initial component model, the set of Modal Sequence Diagrams for the application scenarios, that are specified in the overall system model, are decomposed into smaller specification parts that span a smaller set of components (Step 2). For these smaller specifications, that we will later transform into protocols, the behavior can be implemented more easily and verified more effectively. Each interaction must fulfill certain real-time constraints. All constraints on the order of messages are included in the set of Modal Sequence Diagrams of the application scenarios. But, there may be also constraints on different operating states of the system that must be considered during the communication. These constraints are extracted from the set of behavior – state models. The result are the set of constraints for the components’ communications and a set of Modal Sequence Diagrams for the external visible behavior of all components.

In Step 3, for each structural component a communication protocol for each connector is derived from the according set of Modal Sequence Diagrams. The protocol behavior is specified by Real-Time Statecharts, a variant of UML state machines with a formal semantics based on Timed Automata [AD94]. This allows the application of formal analysis techniques such as model checking to ensure certain safety and lifeness properties of the protocol. For each participant in the protocol, the abstract role’s behavior is modeled by a Real-Time Statechart to allow a flexible reuse of the protocol in other contexts. These Real-Time Statecharts are later (cf. Step 4) instantiated and refined in component ports. Additionally, temporal logic is used to define properties that hold for the protocol behavior. The properties are derived from the set of constraints that are extracted in step 2. The combination of these properties and the Real-Time Statecharts for one reusable, application independent protocol behavior is called **Real-Time Coordination Pattern** (cf. Section 3.1). In Step 3, the Real-Time Coordination Patterns for each connector of the structured components’ parts are determined as described in detail in Section 4.2.1 and Section 4.2.2. This is performed for all structured components in parallel.

After Step 3, each component uses at least one Real-Time Coordination Pattern. For each port of the components, a role of a Real-Time Coordination Pattern and the corresponding Real-Time Statechart are associated. This associated behavior of the roles specifies the ex-
ternal visible behavior of the components. In the next step (Step 4), for each component, the component’s behavior is determined with respect to the external visible behavior. In particular, it must be ensured that the determined behavior is a valid refinement of all associated role behaviors.

Step 4 can be split into three alternatives as described in detail in Section 4.2.3: first, it must be decided if an appropriate component exists that can be reused. For existing components, only the binary code may be available (e.g., the component may be delivered by an external company that does not provide the source models). In such a case, the binary code is the only output of Step 4 and no Real-Time Statechart exists for the component’s behavior for the rest of the process. However, as described by Henkler et al. the Real-Time Statechart of the component’s external visible behavior can be derived with the help of a learning approach such that a correct integration of the component can be ensured (cf. Section 4.2.3) [HBB+09, HMS+10].

Second, if the component is an atomic component, the component’s behavior is derived directly from the parallel composition of the roles behavior (cf. Section 4.2.4). The result is a Real-Time Statechart for the component’s behavior.

Third, the component can be decomposed into further subcomponents to reduce the complexity. The component becomes a structured component and embeds parts which represent the subcomponents. The behavior of the structured component is defined by the interaction of the parts and the behavior of the subcomponents. For the development of the subcomponents, again communication protocols are defined and the determination of the behavior is repeated for each subcomponent (cf. Section 4.2.3). The subcomponents may, therefore, be decomposed until the complexity of the behavior is acceptable to derive the behavior directly or an existing component can be integrated. The result consists of the component model that is extended by subcomponents, and the behavior specification of all subcomponents.

After Step 4, the structure and the behavior of the system’s software is specified completely with respect to the safety properties specified for the Real-Time Coordination Patterns. But, it is not yet guaranteed that all relevant safety constraints are defined for the system. Furthermore, the models that are specified by other disciplines such as mechanical engineering may induce additional constraints for the behavior or contain flaws.

In the MECATRONICUML approach, the software model is simulated together with models from other disciplines to identify missing constraints and flaws that result from unforeseeable interaction of the results from the different disciplines (Step 6). At the moment, a simulation is only possible if the components’ behavior is specified by Real-Time Statecharts. If for at least one component only the binary code exist, it is not possible to simulate the system. In such a case the steps for the simulation (Steps 5 and 6) are skipped.

If a simulation is possible, an initial component instance configuration must be defined in Step 5. In MECATRONICUML the component model specifies various possible structures. This is necessary, because the MECATRONICUML component model enables the specification of reconfigurable components in the component model. A reconfigurable component can exchange, add, or delete parts, connectors and ports during run-time. For the simulation,
an initial instance of the structure must be chosen from the possible structures. This initial structure is specified with a component instance configuration as described in Section 3.5.

A simulation expert chooses from the set of application scenarios the relevant scenarios. For each of these relevant scenarios, a simulation of the system is performed in Step 6. First, the component model and the corresponding Real-Time Statecharts must be transformed to the modeling formalism of an appropriate simulation tool. Additionally, also the models that are developed in other disciplines such as the controller or the shape of the system must be integrated into the simulation model. During the simulation, the behavior of the simulated system is compared to the expected behavior as it is defined by the application scenarios. If the simulated behavior differs from the expected behavior, either the models from other disciplines contain flaws, or the application scenarios’ set of Modal Sequence Diagrams and the constraints, that are adapted from the overall system model in Step 2, are incomplete. The former case must be handled within the development of the other disciplines (Step 11). After the redesign of the other disciplines’ models, Step 6 is repeated.

If the application scenarios are incomplete, the set of Modal Sequence Diagrams are extended for the corresponding communication relation (Step 9). In the same step, the set of Modal Sequence Diagram is checked for contradictions and inconsistencies, until the specification is correct again. It may be necessary to adapt the component model according to the changed set of Modal Sequence Diagrams (Step 10). The changes to the application scenarios and to the component model are reflected in the overall system model. If the changes are relevant to other disciplines, they are synchronized with the models of these disciplines [RS12][RDS+12]. Finally, the development process is repeated for all depending components starting with step 2.

After a successful simulation, the code is generated (Step 7) for all components without existing binary code. The last step is to compile the code (Step 8). In this step also the binary code is linked with the rest of the sourcecode.

4.2.1. Determination of Coordination Patterns (Step 3)

For each structured component, the communication of its parts is precisely specified by Real-Time Coordination Patterns during Step 3. For Real-Time Coordination Patterns it is assumed that each communication can be described independently. This does not, however, mean that no dependencies between different communications exist. But, it must be possible to decompose the communication behavior in such a way that the dependencies of the communications can be modeled as a relationship between the roles of different Real-Time Coordination Patterns within a component. These dependencies are solved later in Step 4 where the components’ behaviors are to be defined.

Figure 4.3 shows the parallel determination of Real-Time Coordination Patterns of all connectors in a structured component. Real-Time Coordination Patterns abstract from a concrete implementation of the communication behavior to enable the reuse of the Real-Time Coordination Pattern in other contexts. In Step 3.1 it is decided, whether an earlier defined Real-Time Coordination Pattern is reusable in the current context. Depending on the set of Modal Se-
sequence Diagrams, an appropriate coordination pattern is identified for the constraints of the communication.

Figure 4.3.: The Subprocess to Reuse or Model a new Coordination Pattern for all Communications within one Structured Component

If it is not possible to find an existing coordination pattern, a new Real-Time Coordination Pattern is modeled and saved for later reuse in Step 3.2. Based on the set of Modal Sequence Diagrams, the communication behavior is described by Real-Time Statecharts for the roles of the Real-Time Coordination Pattern. After formal safety constraints are derived from the set of constraints, it is ensured that the communication behavior fulfills these safety constraints. Step 3.2 is described in more detail in Section 4.2.2.

The result of this step is a Real-Time Coordination Pattern that fulfills a set of safety constraints. For later reuse, the Real-Time Coordination Pattern is saved to a pattern database [BGT05].

4.2.2. Modeling of a Real-Time Coordination Pattern (Step 3.2)

The steps to model a Real-Time Coordination Pattern are shown in Figure 4.4 in detail. A Real-Time Coordination Pattern is composed of different elements that are defined in these steps. More specifically, a Real-Time Coordination Pattern consists of roles, their message interfaces and the roles’ behavior that is specified by Real-Time Statecharts, a connector that models the properties of the roles’ communication channels by a Real-Time Statechart, and a
4.2. THE MECHATRONIC UML PROCESS

set of safety constraints that hold for the communication protocol. First, in Step 3.2.1, a set of formal safety constraints is derived from the constraints of the communication. In parallel, the behavior of the coordination pattern is defined in the Steps 3.2.2 to 3.2.5.

During Step 3.2.2, the roles are derived from the set of Modal Sequence Diagrams. Mostly, the participating components correspond to a Lifeline in the Modal Sequence Diagrams. An applicable set of roles is found if all Lifelines can be associated with the roles easily.

For each role, the message interface (cf. Section 3.2) is derived from the set of Modal Sequence Diagrams (Step 3.2.3). If the roles correspond to a Lifeline, all messages, that are sent from the Lifeline are added to the role’s message interface.

In mechatronic systems, the communication of two components may be unreliable. For instance, messages that are transferred through a wireless connection may change their order or get lost accidentally. The communication protocol must, however, guarantee a safe behavior. In Step 3.2.4, the quality is modeled by a Real-Time Statechart for the roles’ connectors. The effects that are introduced by the connector properties must be considered for the roles’ behavior in Step 3.2.5. The set of Modal Sequence Diagrams is used to derive a Real-Time Statechart for each role. This must be performed iteratively, because the roles depend on each other. The behavior should be specified in such a way that the Real-Time Coordination Pattern can be reused in a wide variety of applications. This often requires additional design effort such as parameterizing time-intervals or adding foreseeable alternative flows of events in the form of non-determinism. The result of this step is a Real-Time Statechart that describes the roles’ behavior such that formal analysis techniques can be used to verify the safety properties [HH11].

At last (Step 3.2.6), the specified behavior is verified against the safety and liveness constraints that are specified in Step 3.2.2 [EHH+11]. If it is not possible to fulfill all constraints, Steps 3.2.5 must be repeated to modify the roles’ behavior. If all constraints hold, Step 3.2 is performed and the result is a new Real-Time Coordination Pattern.

4.2.3. Determination of the Components’ Behavior (Step 4)

The behavior of a component can be determined in three different ways as described in Section 4.2. The detailed steps for these alternative ways to determine the behavior of a component are highlighted by different colors in Figure 4.5. For all alternative ways to determine the component’s behavior, the Real-Time Coordination Pattern, that is defined in Step 3, forms the basis. In particular, each component has a couple of roles associated to the ports. The behavior of all roles must be refined by the component’s behavior [GTB+03, HH11]. During the blue step (step 4.1), an existing component is reused and integrated into the system. The three green steps (Steps 4.2, 3 and 4) are necessary to decompose the component into smaller subcomponents. A direct specification of the behavior is addressed in the orange step (Step 4.3).

At first, it must be decided if an existing component can be reused. If an appropriate component is identified, the component is integrated (Step 4.1). The reuse and integration of components is an ongoing research project. In particular, we are working on methods to integrate
CHAPTER 4. DEVELOPMENT PROCESS

Figure 4.4.: The Subprocess to Model a new Coordination Pattern
4.2. THE MECHATRONIC UML PROCESS

Figure 4.5.: The Subprocess to Determine the Internal Structure or Behavior of a Component

legacy components into the system [HBB+09, HMS+10]. Most legacy components only come with the binary code of the component. Although it is possible to learn the external visible behavior of the component, the internal behavior specification of the component is unknown [HBB+09, HMS+10]. The result of this step is, therefore, the binary code. A Real-Time Statechart for the components behavior is only produced, if it is available for the component.

If no reusable component is available, a new component must be modeled. First, it must be decided whether the component’s behavior can be defined directly, or the component must be decomposed. In particular, the decision is depending on the complexity of the roles’ behaviors and the dependencies between the roles. The roles’ behaviors and their interdependencies are considered to define the component’s behavior.

If the behavior is derived directly, Step 4.3 is performed. A detailed explanation is given in Section 4.2.4. Based on the roles’ Real-Time Statecharts, a parallel composition of Real-Time Statecharts for the component is defined. The component’s Real-Time Statechart refines the behavior of the roles. For instance, times, that are parameterized in the Real-Time Coordination Pattern, are specified according to the concrete application. Giese et al. defined con-
struction rules for the refinement to ensure that the safety constraints of the Real-Time Coordination Pattern are still fulfilled for the refined behavior of the component [GTB+03, HH11]. According to these refinement rules, additional behavior is added such as additional messages for the internal communication within the component. These additional messages are necessary to synchronize the behavior of dependent roles. The result of this step is a Real-Time Statechart for the component’s behavior that is a valid refinement of the roles behavior.

A component with a complex behavior, may be decomposed into smaller subcomponents. During Step 4.2, the component becomes a structured component that is composed of a set of parts (cf. Section 3.4.2). These parts represent the subcomponents that are added to the component (cf. Section 3.4.2.3). The ports of the component are delegated to ports of the parts (cf. Section 3.4.2.4). At the end of the step, all structured component’s ports must be delegated to a port of a part. The associated roles’ behaviors of each component’s ports is to be refined by the subcomponent, it is delegated to.

Due to the dependencies between roles, the behavior can often not be decomposed such that the parts are independent from each other. Instead, connectors that represent communication relations of the parts must be added to deal with the dependencies. These extensions to the component’s structure are added to the overall component model at the end of Step 4.2.

The communication protocols of interacting subcomponents are described in the same way as in Step 2 of the overall MECHATRONIC UML process (cf. Section 4.2). Requirements regarding the protocol behavior are usually specified informally. These requirements may be described by text, sequence diagrams or behavior–state diagrams (as used in the principle solution). Additionally, a set of safety constraints that must be fulfilled by the protocol is defined in an informal manner.

As in the overall MECHATRONIC UML process, the informal communication requirements are first specified by Real-Time Coordination Patterns (Step 3) and the subcomponent’s behavior is determined based on the Real-Time Coordination Pattern afterwards (Step 4). Note, that the last step is a recursion. This means that the subcomponents may be decomposed further, if it is necessary to tackle the complexity of the component’s behavior. The behavior of the subcomponents and its interaction define the behavior of the decomposed component. The result of this step, therefore, consists of the architectural extensions and a set of behavior specifications for the components on the lowest architectural level. The behavior for all atomic components is, thereby, specified by Real-Time Statecharts. For the behavior of an integrated components, only the binary code may be provided.

4.2.4. Modeling of the Components’ Behavior (Step 4.3)

Initially, the behavior of the component is only specified by its externally observable behavior. This is specified for the different roles of the component by Real-Time Statecharts (cf. Section 3.4.1.3). The goal of Step 4.3 is to derive a Real-Time Statechart for the component’s behavior (cf. Section 3.4.1.4). Figure 4.6 shows the detailed steps that are necessary to achieve this goal.

The first Steps 4.3.1 and 4.3.3 are performed in parallel. In Step 4.3.1, for each port a Real-Time Statechart is derived from the associated roles’ Real-Time Statechart. The Real-Time
4.2. THE MECHATRONIC UML PROCESS

Figure 4.6.: The Subprocess to Model the Behavior of a Component
Statechart of the roles can usually be copied or referenced in this step. However, if the corresponding Real-Time Coordination Pattern is parameterized, it may be necessary to identify the appropriate parameters for the application. Furthermore, application specific refinements such as actions for side-effects, additional internal messages, or states must be added. These refinements can change the behavior of the roles and can, therefore, violate the safety constraints of the Real-Time Coordination Pattern. Thus, in Step 4.3.2 the correctness according to the approach proposed by Heinzemann et al. is ensured [HH11]. The result of these two steps is an independent parallel composition of Real-Time Statecharts.

 Dependencies of the roles, that usually exist, are considered in Step 4.3.3. The dependencies are extracted from the roles’ behavior and formalized by composition rules. Composition rules are a combination of timed automata and Boolean logic formulae that are used to specify behavior that must happen in a component or states that are forbidden [EH10].

 The previously specified Real-Time Statecharts must be synchronized according to the composition rules. This is performed by an automatic synthesis technique in Step 4.3.4. The synthesis automatically generates a Real-Time Statechart that is correctly refined and synchronizes the roles’ behavior according to the composition rules. This technique fails, if the behavior specification or the composition rules are inconsistent or contain contradictions. The result of this step is, therefore, a consistent, correctly refined behavior of the component specified by a Real-Time Statechart.

 Due to the inherent complexity of the synthesis, this technique cannot be applied to components with a too complex behavior. In this situation, the synchronization behavior must be modeled manually (Step 4.3.5). The synchronization is realized by an additional Real-Time Statechart called synchronization statechart. The synchronization statechart is developed based on the composition rules. It allows only non-conflicting behavior of the Real-Time Statecharts that are derived from the roles’ behavior in Step 4.3.1. These Real-Time Statecharts are extended by messages that enable the communication with the synchronization statechart. It is the task of the developer to ensure the correctness of the refinement during this extension step. An appropriate approach to guarantee a valid refinement by construction has been proposed by Giese et al. [GTB+03].

 At last, the refined roles’ Real-Time Statecharts and the synchronization statechart are combined to one Real-Time Statechart (cf. Section 3.4.1.4). Each refined roles’ Real-Time Statechart is inserted into a parallel region of the component’s Real-Time Statechart. Additionally, a parallel region is added for the synchronization behavior.

 The last step (Step 4.3.6) ensures that the component’s behavior is free from deadlocks. If deadlocks exist, the specification of the synchronized behavior must be modified and Step 4.3.5 must be repeated.

 The result of the whole subprocess 4.3 is the discrete component’s behavior specified by a Real-Time Statechart.
Chapter 5.

Complete Example

In this chapter, we provide a complete, self-contained example. We model it with MECHATRONICUML. Our example is the environment exploration scenario using BeBots as introduced in Section 1.1. In the following, we first describe the Real-Time Coordination Patterns including their role behaviors used for the scenario in Section 5.1. Afterward, we summarize the message interfaces used for the specification of the Real-Time Coordination Patterns in Section 5.2 and introduce the BeBot component architecture in Section 5.3. Then, the behavior of the components is described in Section 5.4 and the component instance configurations for the example scenario are introduced in Section 5.5.

5.1. Real-Time Coordination Patterns

In this section, we introduce the five Real-Time Coordination Patterns that we use in our example (Figure 5.1).

The Real-Time Coordination Patterns Navigation, Delegation, and DistancesTransmission have the form of communication 1:1. The Real-Time Coordination Patterns Distribution and PositionTransmission have the form of communication 1:n. Navigation, Delegation, and Distribution have a bidirectional communication direction; PositionTransmission and DistancesTransmission are unidirectional (cf. Section 3.1.4). We assume that these five Real-Time Coordination Patterns have no message delay and no message loss. Furthermore, we assume one incoming message buffer of size 1 for each role that may receive messages. Moreover, a message within the buffer must be consumed and cannot be deleted.

The instantiations of these Real-Time Coordination Patterns are shown in Figure 5.18 in Section 5.3.

In the following sections, we introduce the function of the Real-Time Coordination Patterns and the behavior of their roles.

5.1.1. Navigation Pattern

The Real-Time Coordination Pattern Navigation (1:1, bidirectional) transmits an intended target position from a navigator role to a provider role that provides movement to the received
position. After reaching the target position, the success is reported back to the navigator. We describe the message interfaces the roles use in Section 5.2.1.

5.1.1.1. Role Navigator

Figure 5.2 shows the Real-Time Statechart of the role navigator. It consists of two states, the initial state Stop and the state Go. The transition from Stop to Go sends a target position as a one-dimensional array with two entries, representing the x and y coordinates of a target position, via the message moveTo. The message targetReached triggers the transition from Go back to Stop.
5.1. REAL-TIME COORDINATION PATTERNS

5.1.1.2. Role Provider

Figure 5.3 shows the Real-Time Statechart of the role provider. It represents the counter-part to the Real-Time Statechart in Figure 5.2 and consists of the two states Polling, which is the initial state, and Moving. The message moveTo triggers the transition from Polling to Moving. The message moveTo has a target position as a one-dimensional integer array with two entries as parameter. The transition from Moving back to Polling sends the message targetReached.

![Diagram of Role Provider](image)

Figure 5.3.: Role Provider of Pattern Navigation

5.1.2. Delegation Pattern

The Real-Time Coordination Pattern Delegation (1:1, bidirectional) delegates the task of checking the validity of a target position from a master to a slave role. We describe the message interfaces, which the roles use, in Section 5.2.2.

5.1.2.1. Role Master

Figure 5.4 shows the Real-Time Statechart of the role master. It consists of the initial state Inactive and the state PositionCheck. The transition from Inactive to PositionCheck sends a target position as a one-dimensional integer array with two entries via the message check. Upon the activation of PositionCheck the clock c_wait is reset via an entry-action. An invariant using c_wait ensures that PositionCheck is left no later than 150 units of time after its activation. There are two outgoing transitions leading back to state Inactive. The one with the higher priority is triggered by the message declined, the other is triggered by the message accepted.

5.1.2.2. Role Slave

Figure 5.5 shows the Real-Time Statechart of the role slave. It represents the counter-part to the Real-Time Statechart in Figure 5.4 and consists of the same states. The message check triggers the transition from Inactive to PositionCheck and the target position is received as parameter. Upon the activation of PositionCheck, the clock c_work is reset via an entry-action. An invariant using c_work ensures that PositionCheck is left no later than 100 units of time after its activation. There are two outgoing transitions leading back to state Inactive. The one with the higher priority sends the message declined, the other sends the message accepted.
5.1.3. Distribution Pattern

The pattern Distribution (1:n, bidirectional) transmits position data between the multi-role distributor and one or more instances of the role client. The distributor collects the positions of all clients and sends the collected positions and its own position back to each of them using a multi-cast. We describe the message interfaces the roles use in Section 5.2.3.

5.1.3.1. Role Distributor

Figure 5.6 shows the Real-Time Statechart of the multi-role distributor. Since it specifies the behavior of a multi-role, it follows the convention that there is only one initial composite state, Distribution_distributor_Main, with the two regions adaptation and sub-role-template (cf. Section 3.1.6). The sub-role-template region contains a Real-Time Statechart that only consists of the state Active. The state Active, in turn, consists of two regions receive and send which are responsible for receiving or sending data to the clients.

The behavior of the multi-role is as follows. The execution starts in state Waiting of the region adaptation and in the states Idle of receive and send. Periodically, the adaptation region checks whether position data has been received from all clients. Therefore, it fires the transition from Waiting to Receiving if c is greater than 50 thereby synchronizing with the first sub-role. If the message position is available upon the synchronisation receive[ id], it is processed and the position information is written to the variable posArray. If the sub-role has no message in its buffer, an error occurred and the sub-role switches to Received and sets the variable error to true. In state Received, it synchronizes with the next sub-role using channel receive or it synchronizes with the adaptation statechart if it is the last sub-role of the ordered
5.1. REAL-TIME COORDINATION PATTERNS

Figure 5.6.: Role Distributor of Pattern Distribution
sub-role-list. If error is true, then the adaptation statechart switches to state Waiting without sending the position data to the clients, otherwise it switches to state All_Received.

If the distributor receives new position data from all of its clients, it will start sending the collected position data back to the clients. Therefore, it fires the transition from All_Received to Sending thereby synchronizing with the first sub-role using the synchronization channel send. Then, the region send of the first sub-role switches to Sent and sends the message positions to the client. Thereafter, it synchronizes with the next sub-role using the synchronization channel send if a next sub-role exists. Otherwise, this is the last sub-role that has to synchronize with the adaptation. Then, the adaptation statechart switches to state Waiting and the behavior starts all over again.

5.1.3.2. Role Client

Figure 5.7 shows the Real-Time Statechart of the role client. It represents the counter-part to the Real-Time Statechart in Figure 5.6, more precisely to the state Active in region sub-role, and consists of the initial composite state Distribution_client_main with its two regions send and receive.

Both regions start their execution in state Blocked. If c becomes greater than 35 units of time, the Real-Time Statechart switches to state Ready. In this state, it expects to receive new position data from the distributor every 50 units of time. If a message positions is received, the self-transition at state Ready in region receive is fired. As a side effect, the received position data is stored in the variable posArray. If no such message is received within 50 units of time, the invariant of state Ready expires and the state is left by firing the transition leading to the state Error. The statechart remains in that state until new position data is received. Receiving the new position data triggers the transition which leads back to Ready.
The statechart in region `send` initially sends its own position to the distributor when the value of `c` exceeds 45 units of time. The position data is sent by means of the message `position`. In state `Ready`, the invariant enforces the state to be left every 50 units of time. Then, the self-transition is fired, which causes new position data to be sent to the distributor. When entering the state `Ready` again, the entry action sets the value of `c` back to 0.

### 5.1.4. PositionTransmission Pattern

The pattern `PositionTransmission` (1:n, unidirectional) transmits a position from the multi-role sender to one or more instances of the role `receiver`. The message interface used by the roles is described in Section 5.2.4.

#### 5.1.4.1. Role Sender

Figure 5.8 shows the Real-Time Statechart of the multi-role `sender`. Like the Real-Time Statechart in Figure 5.6 it follows the convention that there is only one initial composite state, `PositionTransmission_sender`, with the two regions adaptation and sub-role (cf. Section 3.1.6).

![PositionTransmission_sender Statechart](image)

**Figure 5.8.: Role Sender of Pattern PositionTransmission**

The execution starts in state `Waiting` of region `adaptation` and in state `Idle` of region `sub-role`. When the value of `c` exceeds 10 units of time the region `adaptation` synchronizes with its first sub-role with the channel `send`. In state `Sending` it resets the clock `c`. The sub-role sends the message `position` upon synchronization and synchronizes then with the next sub-role (transition from `Sent` to `Idle`). The last sub-role synchronizes with the adaption statechart with the channel `send_done`. Note that the adaptation statechart has to exit its state `Sending` within
5 units of time, meaning the sending of the position data of all sub-roles has to be performed in that time invariant.

5.1.4.2. Role Receiver

Figure 5.9 shows the Real-Time Statechart of the role receiver. It represents the counter-part to the Real-Time Statechart in Figure 5.8, more precisely to the region sub-role. In the initial state Active, a self-transition is fired upon receiving the message position, while the position data is stored in the variable pos. When entering the Active state, the clock c_period is reset. If the clock exceeds 100 units of time, the statecharts changes its state to Error. From this state, a transition leads back to state Active when position is received.

5.1.5. DistancesTransmission Pattern

The pattern DistancesTransmission (1:1, unidirectional) transmits an array of distance values from the role sender to the role receiver. The message interface used by the roles is described in Section 5.2.5.

5.1.5.1. Role Sender

Figure 5.10 shows the Real-Time Statechart of the multi-role sender. It consists of one state Active. In this state it can send the message distances with a one-dimensional floating point array, which contains the distances to the other BeBots, as parameter.

Figure 5.9.: Role Receiver of Pattern PositionTransmission

Figure 5.10.: Role Sender of Pattern DistancesTransmission
5.1.5.2. Role Receiver

Figure 5.11 shows the Real-Time Statechart of the role receiver. It represents the counter-part to the Real-Time Statechart in Figure 5.10 and is nearly identical to the Real-Time Statechart in Figure 5.8. It differs from the latter only in the received message distances with a one-dimensional floating point array as parameter, which is set to the variable distArray.

```
DistancesTransmission_receiver
var: float[8] distArray;
clock: c_period;

Active

c_period ≤ 100
entry / {reset: c_period}

Error
[c_period ≥ 100]

distances / {distArray:=distances.array;}
```

Figure 5.11.: Role Receiver of Pattern DistancesTransmission
5.2. Message Interface Specification

In this section, we introduce the message interfaces the Real-Time Coordination Patterns of Section 5.1 use. A Real-Time Coordination Pattern uses each message interface twice: once as a sender message interface, once as a receiver message interface. We compose the names of all message interfaces using the name of the Real-Time Coordination Pattern followed by the name of the role carrying the message interface as a sender message interface.

5.2.1. Navigation

The Real-Time Coordination Pattern Navigation transmits an intended target position to a component that provides movement. After reaching the target position, the success is reported to the navigator. Figure 5.12 shows the two message interfaces needed for this Real-Time Coordination Pattern.

<table>
<thead>
<tr>
<th>Navigation_Navigator</th>
<th>Navigation_Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>moveTo(int[2] xy)</td>
<td>targetReached()</td>
</tr>
</tbody>
</table>

Figure 5.12.: Message Interfaces for Real-Time Coordination Pattern Navigation

The message interface Navigation_Navigator contains the message type moveTo. The navigator role uses the messages of this type to transmit a position to the provider role. The parameter xy is a one-dimensional array of length 2 that contains the coordinates of the position.

The message interface Navigation_Provider contains the message type targetReached. This message signals the navigator role that the BeBot reaches the intended target position.

5.2.2. Delegation

We use the Real-Time Coordination Pattern Delegation to delegate a task to the slave, which reports the success of the task execution. Figure 5.13 shows the two message interfaces, which the Real-Time Coordination Pattern uses.

<table>
<thead>
<tr>
<th>Delegation_Master</th>
<th>Delegation_Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>check(int[2] target)</td>
<td>accepted()</td>
</tr>
<tr>
<td></td>
<td>declined()</td>
</tr>
</tbody>
</table>

Figure 5.13.: Message Interfaces for Real-Time Coordination Pattern Delegation
The Delegation_Master message interface contains only the message type check which transmits a 2D position information encoded as a one-dimensional array of length 2 to the slave.

The message interface Delegation_Slave contains two message types, accepted and declined. The slave signals success by the message type accepted and signals that executing the task is not possible by the message declined.

### 5.2.3. Distribution

The Real-Time Coordination Pattern Distribution transmits position data between the BeBots. Figure 5.14 shows the two message interfaces used by the Real-Time Coordination Pattern.

- **Distribution_Distributor**
  positions(int[9][2] posArray)

- **Distribution_Client**
  position(int[2] pos)

Figure 5.14.: Message Interfaces for Real-Time Coordination Pattern Distribution

The distributor role uses the Distribution_Distributor message interface to transmit the positions of all BeBots to a client. Therefore, the message interface contains a message type positions which has a two-dimensional array as a parameter. There exists one entry for each of the at most 9 BeBots and for each entry it contains the x and y position.

The client uses the message interface Distribution_Client to transmit its own position to the distributor. Thus, the message interface specifies a message type position which contains a one-dimensional array of length two as a parameter. It contains the x as first entry and y position as second entry.

### 5.2.4. PositionTransmission

The Real-Time Coordination Pattern PositionTransmission transmits the own position inside the BeBot. Since it is a unidirectional Real-Time Coordination Pattern, it only uses one message interface, which Figure 5.15 shows.

- **PositionTransmission_Sender**
  position(int[2] xy)

Figure 5.15.: Message Interfaces for Real-Time Coordination Pattern PositionTransmission
The PositionTransmission_Sender message interface specifies one message type xy to transmit the position data. It contains a one-dimensional array of length two as a parameter containing the x and y position of the BeBot.

### 5.2.5. DistancesTransmission

The Real-Time Coordination Pattern DistancesTransmission transmits the distances to the other BeBots inside the BeBot. Since it is a unidirectional Real-Time Coordination Pattern, it only uses one message interface which Figure 5.16 shows.

<table>
<thead>
<tr>
<th>DistancesTransmission_Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>distances(int[8] array)</td>
</tr>
</tbody>
</table>

Figure 5.16.: Message Interfaces for Real-Time Coordination Pattern DistancesTransmission

The DistancesTransmission_Sender message interface specifies one message type distances to transmit the distances. It contains a one-dimensional array of length 8 for the at most 8 distances to the other BeBots.

### 5.3. System Structure

In this section, we introduce the components specifying the system structure for the BeBot example scenario. Firstly, we show the structured components representing the whole BeBot and the discrete BeBot software in Section 5.3.1. The structured components are composed by a set of atomic components which we introduce in detail in Section 5.3.2. Each of the atomic components implements a specific part of the overall BeBot behavior. They are embedded as component parts in the BeBot and interact with each other using the Real-Time Coordination Pattern introduced in Section 5.1.

#### 5.3.1. Structured Components

In our example, we use two structured components for the BeBot model: one to represent the BeBot as a whole including the continuous control software, the other one to model the discrete BeBot software. In the following, we introduce the structured components in that order.

Figure 5.17 shows the structured component BeBot that represents the BeBot as a whole. The component embeds a hybrid component part bsw of type BeBot_SW that contains the discrete BeBot software. This component part implements the ports distributor and client of the structured component BeBot. These ports exchange position data with other BeBots exploring
the environment. Additionally, it implements three continuous ports connected to the continuous component parts \texttt{en} and \texttt{pos} of types \texttt{EngineCtrl} and \texttt{PositionSensor}, respectively. The \texttt{en} component part of type \texttt{EngineCtrl} controls the speed of one of the chain drives of the BeBot. Since the BeBot has two chain drives, each BeBot instance requires two such controllers. The \texttt{pos} component part of type \texttt{PositionSensor} provides the GPS positioning data to the software.

The \texttt{BeBot_SW} component encapsulates the discrete BeBot software. The software needs to behave according to its role in the environment exploration scenario. More concrete, its behavior depends on whether it is the position distributor BeBot or a client BeBot. Additionally, the BeBot must implement behavior for navigation, collision control, and for deciding on the target position.

In our example, we decompose the overall behavior of the BeBot into four atomic components that are embedded into the \texttt{BeBot_SW} component by using component parts. This corresponds to Step 4.2 of the development process as shown in Figure 4.5 on Page 91.

Figure 5.18 shows the structured component \texttt{BeBot_SW} that represents the software of the BeBot.

The component specifies five ports: \texttt{speed\_left}, \texttt{speed\_right}, \texttt{position}, \texttt{client}, and \texttt{distributor}. The continuous ports \texttt{speed\_left} and \texttt{speed\_right} set the speed for the chain drives of the BeBot. The continuous port \texttt{position} obtains the current position from the GPS sensor.

We specify the behavior of the remaining two discrete ports of \texttt{BeBot_SW} and the discrete ports of the atomic components \texttt{Navigation}, \texttt{Exploration}, \texttt{BeBot\_Observer} and \texttt{Collision\_Control} by refining the roles of the Real-Time Coordination Patterns. We describe them
in Section 5.1. This corresponds to Step 4.3.2 of the development process as shown in Figure 4.6 on Page 4.6.

An instance of the pattern Distribution describes the communication between distributor and client. This is an example that communication between components of the same type is possible. Its multi-role distributor is assigned to the equally named multi-port distributor and its other role client to the equally named port client. The distributor port is only active when the BeBot operates as the position distributor. In this case, there is one instance of this port for each client BeBot such that they can receive position data from all other clients and can send them to all other clients. Accordingly, the client port is only used when operating as a client.

The pattern Navigation is instantiated between the ports navigator of the component part exp : Exploration and the port provider of the component part nav : Navigation. An instance of pattern Delegation is assigned to the port master of the component part exp : Exploration and the port slave of the component part cc : Collision_Control. The instantiated pattern DistancesTransmission describes the communication between the ports sender of the component part bbo : BeBot_Observer and receiver of the component part cc : Collision_Control. The cardinality of the multi-role receiver is reduced to an upper bound 1, such that it is consistent to a single-port. The instance of pattern PositionTransmission describes the communication between the multi-port sender of Navigation and the ports receiver of bbo : BeBot_Observer and exp : Exploration. The single-role receiver is instantiated two times here.
We describe the atomic components Navigation, Exploration, BeBot_Observer and Collision_Control in detail in the following section.

5.3.2. Atomic Components

In our example, we use the six atomic components shown in Figure 5.19 for modeling the BeBot. We use four discrete atomic components, and two continuous atomic components. We describe their purpose in the following sections and explain their behavior in Section 5.4.

![Diagram of Atomic Components](image)

Figure 5.19.: The Atomic Components of the BeBot

5.3.2.1. Exploration

The component Exploration controls the exploration of the environment. Therefore, it calculates a next position for the BeBot randomly based on the current position. The current position is received from the Navigation component using the PositionTransmission pattern. Before the new position for the BeBot is sent to the Navigation using the Real-Time Coordination Pattern Navigation, it is checked whether it is safe to move to the calculated position. Therefore, the Exploration sends the new position to the Collision_Control using the Delegation pattern to check for potential collisions. If no collision may occur, the position is actually sent to the Navigation in order to start moving there. The complete definition of the behavior of the Exploration component is explained in Section 5.4.1.
5.3.2.2. Navigation

The component Navigation provides two services. Firstly, it receives and processes the current position data from the GPS. Then, it transmits the position data regularly via the PositionTransmission pattern to the components Exploration and BeBot_Observer. The component Exploration uses the data for calculating the next position to move to and the component BeBot_Observer sends the own position to the other BeBots. Secondly, the Navigation provides an interface to the two chain drives of the BeBot. Given a target position, which is received via the Navigation pattern, the Navigation sends the left and right speed to the two engine controllers in order to move from the current position to the target position. After reaching the target position, the success is reported to the Exploration which then can compute the next position. The complete behavior definition of the Navigation component is explained in Section 5.4.2.

5.3.2.3. BeBot Observer

The component BeBot_Observer is responsible for maintaining the positions of all other BeBots in the environment. The BeBot_Observer may either operate as the position distributor or as a client of a position distributor via the Distribution pattern. As a client, the BeBot_Observer sends regularly its current position to the distributor. Then, the distributor answers with an array containing the current positions of all other BeBots. This information is then used to calculate the distances to the other BeBots which are sent to the Collision_Control via the DistancesTransmission pattern in order to avoid collisions. When operating as a position distributor, the BeBot_Observer waits for clients to report their position. If a new position of a client is received, the position data is updated internally and the updated position data is sent to the client. Like a client, the position distributor sends the calculated distances to the Collision_Control. In order to be able to communicate with a varying number of clients, the distributor port of the BeBot_Observer is a multi-port which is delegated from the BeBot_SW. It is delegated because the ports distributor and client are offered by the BeBot component to interact with other BeBots.

The complete behavior definition of the BeBot_Observer component is explained in Section 5.4.3.

5.3.2.4. Collision Control

The component Collision_Control is responsible for avoiding collisions with other BeBots exploring the environment. More specifically, the Collision_Control must decide for a given target position whether it is safe to move there. Therefore, it receives the intended target position from Exploration via the Delegation pattern. Additionally, the Collision_Control receives the distances to all other BeBots from the BeBot_Observer via the DistancesTransmission pattern. From these information, the Collision_Control can decide whether moving to the target position is safe or not. If it is safe, an accept is sent to Exploration. Otherwise, a decline
is sent. The complete behavior definition of the Collision_Control component is explained in Section 5.4.4.

5.3.2.5. PosData

The continuous component posData provides position data for the BeBot. It is connected to the GPS hardware and continuously evaluates the incoming sensor signals. As an output, it provides a continuously updated position signal which may be used by the Navigation component.

Since the behavior models for continuous components are not part of MECHATRONICUML, we do not further describe the behavior of PosData in this document.

5.3.2.6. EngineCtrl

The continuous component EngineCtrl controls one engine of the BeBots. It is assumed to be a closed-loop controller. The input speed_in is the current reference value for the engine, i.e., the speed that it should provide. Then, the controller ensures that the desired speed will be provided by the engine.

Since the behavior models for continuous components are not part of MECHATRONICUML, we do not further describe the behavior of EngineCtrl in this document.
5.4. Real-Time Statechart

In this section, we present in detail the complete behavior of the four atomic components Exploration, Navigation, BeBot_Observer, and Collision_Control introduced in Section 5.3.2. Real-Time Statecharts specify the behavior of each component.

5.4.1. Exploration

The component Exploration controls the exploration of the environment by calculating new positions for the BeBot, validating them via the component Collision_Control and sending them to the component Navigation.

Figure 5.20 shows the Real-Time Statechart of Exploration. Since it follows the standard structure for a Real-Time Statechart of a discrete component, it only consists of an initial composite state, Exploration_Main, whose parallel regions contain the Real-Time Statecharts of the component ports and one Real-Time Statechart named InternalBehavior to synchronize them. Therefore, Exploration_Main declares the channels checkPosition, with an one-dimensional integer array with two entries as parameter, positionDataOk, noPositionData, positionOk, positionRejected, driveComplete and drive, with an one-dimensional integer array with two entries as parameter.

As described in Section 5.3.1, the role receiver of pattern PositionTransmission is assigned to the equally named port of this component so the Real-Time Statechart of region receiver is a refinement of the Real-Time Statechart of the role (cf. Figure 5.9). It is refined by adding the sending of a synchronization via channel noPositionData to the transition from Active to Error and the sending of one channel positionDataOk to the transition from Error back to Active.

The role navigator of pattern Navigation is assigned to the equally named port of this component so the Real-Time Statechart of region navigator is a refinement of the Real-Time Statechart of the role (cf. Figure 5.2). We refine it by adding the receiving of a synchronization via channel drive, whose parameter is used as parameter for the sending of message moveTo, to the transition from Stop to Go and the sending of a synchronization via channel driveComplete to the transition from Go back to Stop.

The role master of pattern Delegation is assigned to the equally named port of this component so the Real-Time Statechart of region master is a refinement of the Real-Time Statechart of the role (cf. Figure 5.4). We refine it by adding the receiving of a synchronization via the channel checkPosition. The transition from Inactive to PositionCheck uses in its raise message check the parameter target of checkPosition. Further, the lower prioritized transition from PositionCheck back to Inactive synchronize via channel the positionRejected with the transition from the state CheckPosition to the state DecideOnNextPosition in region InternalBehavior. The higher prioritized transition synchronizes via the channel positionOk with the transition from CheckPosition to PositionOk.

The region InternalBehavior contains the two states Active and NoPositionData which are changed according to received synchronizations over the channels noPositionData, from Active to NoPositionData, and positionDataOk, back to Active. The initial state Active consists
Figure 5.20.: Real-Time Statechart of Component Exploration
of a single region with the four states DecideOnNextPosition, CheckPosition, PositionOk, and Drive. The initial state DecideOnNextPosition determines a new target position by calling the operation playDice with the current position pos as parameter and resets the clock c in its entry-action. Its invariant ensures that it is left no later than 200 units of time after its activation and its exit- action resets c again. The transition to CheckPosition sends a synchronization over channel checkPosition with target as parameter and has to satisfy an absolute deadline with a lower bound of 25 and an upper bound of 30 units of time over c. The invariant at CheckPosition ensures that a synchronization is received over channel positionRejected or positionOk within 200 units of time after its activation and clock reset of c per entry-action. Both result in the reactivation of state DecideOnNextPosition but the second requires the traversal of PositionOk and Drive by sending a synchronization over the channel drive, with the target position as parameter, and receiving over channel driveComplete.

5.4.2. Navigation

The component Navigation transmits the current position to the components Exploration and BeBot_Observer and receives a target position from component Exploration. Navigation needs the target position to calculate the speed which is needed to move to the target position. Figure 5.21 shows the Real-Time Statechart of Navigation.

Since it follows the standard structure for a Real-Time Statechart of a discrete component, it only consists of an initial composite state, Navigation_Main, whose parallel regions contain the Real-Time Statecharts of the discrete and hybrid component ports and one Real-Time Statechart named InernalBehavior to synchronize them. For the synchronization the state Navigation_Main declares the channels finished, without parameters, go, with an one-dimensional integer array with two entries as parameter, representing a new target position. Further, it declares the two channels syncLeft and syncRight to synchronize the new speed with the Real-Time Statecharts of the corresponding hybrid ports. Both channels have one integer parameter that represents the desired speed of the left and right chain drive.

The Real-Time Statecharts of the regions speed_left and speed_right are used to write the corresponding signal values of the hybrid outgoing ports as described in Section 3.4.2.2. The component Navigation receives via the hybrid port position the signal value which represents the current position of the BeBot. This position value is delegated to the Real-Time Statecharts of the regions synchronization and sender via the channels currentPosition, and currentPosDeleg. Each channel has an one-dimensional integer array with two entries, representing the current x- and y- position of the BeBot, as parameter. The Real-Time Statechart of the region position specifies how often the input signal position is read and processed.

As described in Section 5.3.1, the role provider of pattern Navigation is assigned to the equally named port of this component so the Real-Time Statechart of region provider is a refinement of the Real-Time Statechart of the role (cf. Figure 5.3). It is refined by adding the sending of a synchronization via channel go with the received parameter xy to the transition from Polling to Moving and the receiving of one channel finished to the transition from Moving back to Polling.
Figure 5.21.: Real-Time Statechart of Component Navigation
The multi-role sender of pattern PositionTransmission is assigned to the equally named multi-port of this component so the Real-Time Statechart of region provider is a refinement of the Real-Time Statechart of the role (cf. Figure 5.8). Because the behavior of this Real-Time Statechart wants to send its current position to all other BeBots via the message position, it must always know its current position. Therefore, it synchronizes repetitively every four ticks via the channel currentPosDeleg with the synchronization Real-Time Statechart to get the current position. The position is stored in the variable pos. This variable is used as the parameter of message position.

The region InternalBehavior contains the six states Stop, LeftStop, RightStop CalculateSpeed, RightMove, and Move. Stop is the initial state. The transition to CalculateSpeed receives a synchronization via channel go and sets the received parameter xy as the current target position. The entry-action of CalculateSpeed resets clock c and calls the operation calcSpeed, with the current position and the target position as parameter. It calculates the needed movement to reach the target and the according speed values for the left and the right chain drives which are stored in the first two entries of the one-dimensional integer array speed. An invariant over c ensures that CalculateSpeed is left no later than 20 units of time after its activation. If the value of c is greater or equal to 20, the transition to the urgent state RightMove sends the new speed value of the right chain drive via the channel syncRight to the self-transition of the state SetRightSpeed in the Real-Time Statechart of the region speed_right. As a result the output signal of the hybrid port speed_right gets the new desired speed value. Accordingly, the speed of the left chain drive is set via the channel syncLeft at the transition to state Move.

The invariant of Move and the reset of c in its entry-action ensure that it is left in no more than 200 units of time after its activation. It can be left via three transitions from which the transition to the urgent state RightStop has the highest priority. It checks if the first two entries of position and target are identical. The transition chain to the states RightStop and LeftStop sends synchronizations to both Real-Time Statecharts of the hybrid ports to set the speed of the chain drives to zero. The transition from LeftStop to the initial state Stop synchronizes via the channel finished with the region provider. The other two transitions from the state Move lead back to CalculateSpeed from which the one with the second highest priority receives a synchronization via channel go and sets the received position as the new target. The one with the lowest priority only checks if the value of c is greater or equal to 200.

### 5.4.3. BeBot Observer

The component BeBot_Observer is responsible for maintaining the positions of all BeBots in the environment and for calculating and transmitting the distances to the other BeBots to Collision_Control. Figure 5.22 shows the Real-Time Statechart of BeBot_Observer.

Since it follows the standard structure for a Real-Time Statechart of a discrete component, it only consists of an initial composite state, BeBot_Observer_Main, whose parallel regions contain the Real-Time Statecharts of the component ports and one Real-Time Statechart named InternalBehavior to synchronize them. For this BeBot_Observer_Main declares
Figure 5.22.: Real-Time Statechart of Component BeBot Observer
the channels commOk, commError, ok, and error all without parameters, and evalPos with a one-dimensional array of size 2 to store a position and evalPos with a two-dimensional array to store the position for every bebot.

As described in Section 5.3.1, the role receiver of pattern PositionTransmission is assigned to the equally named port of this component so the Real-Time Statechart of region receiver is a refinement of the Real-Time Statechart of the role (cf. Figure 5.9). The behavior is refined by adding the synchronization via channel newPos when a message position is received. With this channel it synchronizes with the client region.

The role sender of pattern DistancesTransmission is assigned to the equally named multi-port of this component so the Real-Time Statechart of region sender is a refinement of the Real-Time Statechart of the role (cf. Figure 5.10). It is refined by adding the state Error and a transition from DistancesTransmission_sender_main to Error with a synchronization via channel error, and a transition back with a synchronization via channel ok. Furthermore, the synchronization channel evalPos is added to the self-transition in state Active to synchronize with the client region.

The role client of pattern Distribution is assigned to the equally named port of this component so the Real-Time Statechart of region client is a refinement of the Real-Time Statechart of the role (cf. Figure 5.7). It is refined by the synchronization channel newPos added to region send as well as the synchronization channel evalPos added to region receive. Furthermore, in region receive the sending of a synchronization via channel commError to the transition from Receiving to Error and one via commOk to the reverse transition is added.

The multi-role distributor of pattern Distribution is assigned to the equally named multi-port of this component so the Real-Time Statechart of region distributor is a refinement of the Real-Time Statechart of the role (cf. Figure 5.6). Because the behavior of this Real-Time Statechart does not have to be synchronized with the other regions, it is only refined by using the globally defined variables pos and posArray.

The region InternalBehavior consists of the four states Active, which is the initial state, ErrorReceived, CommError, and OkReceived. The transition from Active to ErrorReceived receives a synchronization over channel commError and the transition to CommError sends one over channel error. The transition from Error to OkReceived receives a synchronization over channel commOk and the transition to Active sends one over channel ok. Invariants over clock c ensure that ErrorReceived and OkReceived, which entry-actions reset clock c, are left immediately after their activation.

### 5.4.4. Collision Control

The component Collision_Control is responsible for avoiding collisions with other BeBots by deciding whether a received target position from Exploration conflicts with the distances to all other BeBots as received from the BeBot_Observer.

Figure 5.23 shows the Real-Time Statechart of Collision_Control. Since it follows the standard structure for a Real-Time Statechart of a discrete component, it only consists of an initial
5.4. REAL-TIME STATECHART

Figure 5.23.: Real-Time Statechart of Component Collision Control
composite state, Collision_Control_Main, whose parallel regions contain the Real-Time State charts of the component ports and one Real-Time Statechart named InternalBehavior to synchronize them. Therefore, Collision_Control_Main declares the channels checkPermission, with a one-dimensional integer array with two entries, granted, rejected, distancesDataOk, and noDistancesData.

As described in Section 5.3.1, the role receiver of pattern DistancesTransmission is assigned to the equally named port of this component so the Real-Time Statechart of region receiver is a refinement of the Real-Time Statechart of the role (cf. Figure 5.11). It is refined by adding the sending of a synchronization via channel noDistancesData to the transition from Active to Error and the sending of one via distancesDataOk to the reversed transition.

The role slave of pattern Delegation is assigned to the equally named port of this component so the Real-Time Statechart of region slave is a refinement of the Real-Time Statechart of the role (cf. Figure 5.5). It is refined by adding the sending of a synchronization via channel checkPermission to the transition from Inactive to PositionCheck with check.target as parameter which is the parameter target received via message check. The receiving of synchronizations via channel rejected is added to the transition from PositionCheck to Inactive and via granted to the transition from PositionCheck to Inactive.

The region InternalBehavior contains the three states Running, which is the initial state, DistanceUnknown, and CheckNotPossible. A transition from Running to DistanceUnknown receives synchronizations via channel noDistancesData and the reverse transition via distancesDataOk. The other outgoing transition from DistanceUnknown to CheckNotPossible is lower prioritized and receives synchronizations via checkPermission. CheckNotPossible is left immediately after its activation by the use of an invariant over clock c and a clock reset of c by its entry-action. The transition back to DistanceUnknown sends a synchronization via channel rejected. The composite state Running contains a region with the two states Wait and PositionChecked. The transition from the initial state Wait to PositionChecked receives a synchronization over channel checkPermission and calls the operation getApproval with the received target position and the one-dimensional floating point array distArray as parameter. The result is set to the boolean variable permission. The earliest deadline for computing this result is left immediately after its activation by the use of an invariant over clock c and a clock reset of c by its entry-action. According to the evaluation result the state Wait is activated again by sending a synchronization via channel rejected, in the case that permission is false, or via granted, otherwise.
5.5. Component Instance Configuration

In this section, we introduce the component instance configurations of our example. First, we show in Section 5.5.1 the instance configuration for a single BeBot exploring the environment. In this case, the BeBot does not need to check for possible collisions with other BeBots. Afterwards, Section 5.5.2 contains a description of component instance configurations for several BeBots exploring the environment.

5.5.1. Single BeBot

In this section, we introduce a component instance configuration for a single BeBot exploring the environment in our example. That BeBot does not need to communicate with other BeBots to avoid collisions because there are no other BeBots in this case.

Figure 5.24 shows a component instance of the type BeBot_SW that is not connected to other component instances of the type BeBot_SW. Consequently, it only has the continuous port instances speed_left, speed_right and position that communicate with the engine.

Figure 5.24.: Concrete Syntax of a Component Instance of the Component Type BeBot_SW of a BeBot that is not Connected to other BeBots

Figure 5.25 shows the structure of the embedded component instances of the BeBot_SW of Figure 5.24. Since this BeBot does not communicate with other BeBots, it only contains the embedded component instances exp:Exploration and nav:Navigation that are specified for the component type BeBot_SW in Figure 5.18. The component instances exp:Exploration and nav:Navigation are connected by assembly connectors instances that are derived from the Real-Time Coordination Patterns Navigation and PositionTransmission (cf. Figure 5.1). The discrete port instances navigator and receiver of the component instance exp:Exploration implement the single-roles receiver of the Real-Time Coordination Pattern PositionTransmission and navigator of the Real-Time Coordination Pattern Navigation. The discrete port instances sender and provider of the component instance nav:Navigation implement the single-roles sender of the Real-Time Coordination Pattern PositionTransmission and provider of the Real-Time Coordination Pattern Navigation. The hybrid port instances speed_left, speed_right and position of nav:Navigation are delegated to the continuous port instances of the same names of the lop-level component instance b4:BeBot_SW.
5.5.2. Networks of BeBots

In this section, we introduce component instance configurations for a BeBot in case that more than one BeBot explores the environment. As described in Section 1.1, the BeBots now have to exchange their position data to avoid collisions. In the following, we show one component instance configuration for the position distributor BeBot and one component instance configuration that applies for all client BeBots. Afterwards, we show how these BeBot component instances have to be connected in our example.

In contrast to Figure 5.24, the component instances of the component BeBot_SW contain additional discrete ports for exchanging the position data. Figure 5.26 shows an instance of the discrete BeBot software for a BeBot operating as the position distributor for two client BeBots. Figure 5.27 shows a component instance of the type BeBot_SW for a client BeBot. Both component instances implement different roles of the Real-Time Coordination Pattern Distribution (cf. Figure 5.1).

---

Figure 5.25.: Concrete Syntax of a Component Instance Configuration of a Single BeBot

Figure 5.26.: Concrete Syntax of a Component Instance of the Component Type BeBot
Component instance b1:BeBot_SW of Figure 5.26 has the two discrete port instances distributor1 and distributor2 that both implement the multi-role distributor of the Real-Time Coordination Pattern Distribution. Thus, this BeBot software executes the behavior of a position distributor BeBot. Each of the two port instances is connected to a client BeBot as shown in Figure 5.28.

Component instance b2:BeBot_SW of Figure 5.27 has a discrete port instance client that implements the single-role client of the Real-Time Coordination Pattern Distribution. Thus, this BeBot executes the behavior of a client to the position distributor.

The component instances shown in Figure 5.29 and 5.30 only represent the BeBot_SW. In order to obtain a component instance specification for a BeBot, the component type BeBot of Figure 5.17 must be instantiated and connected with other instances of type BeBot. Figure 5.28 shows a component instance configuration consisting of the three component instances of type BeBot, namely bebot1:BeBot, bebot2:BeBot and bebot3:BeBot. bebot1:BeBot is the position distributor while bebot2:BeBot and bebot3:BeBot operate as clients. Thus, the assembly connector instances which is derived from the Real-Time Coordination Pattern Distribution connects the bebot2:BeBot and the bebot3:BeBot to bebot1:BeBot. The discrete port instances distributor1 and distributor2 of bebot1:BeBot implement the multi-role distributor. The discrete port instances client of bebot2:BeBot and bebot3:BeBot implement the single-roles client.

Figure 5.28 also shows the embedded component instances of bebot1:BeBot: b1:BeBot_SW, ctrl_left:EngineCtrl, ctrl_right:EngineCtrl, and pd:PosData. The two discrete port instances distributor1 and distributor2 of the component instance b1:BeBot_SW, that implement the role distributor of the Real-Time Coordination Pattern Distribution, are delegated to the discrete port instances distributor1 and distributor2 of bebot1:BeBot. Component instance ctrl_left:EngineCtrl and component instance ctrl_right:EngineCtrl each have the two continuous port instances speed_in and speed_out. Component instance pd:PosData has the two continuous port instances pos_in and pos_out. The assembly connector instances between b1:BeBot_SW and ctrl_left:EngineCtrl, b1:BeBot_SW and ctrl_right:EngineCtrl, b1:BeBot_SW...
Figure 5.28.: Concrete Syntax of a Component Instance Configuration of Three Communicating BeBots
and pd:PosData are not derived from any Real-Time Coordination Pattern as they only connect continuous port instances.

Figure 5.29 and Figure 5.30 show the embedded component instances of the component instances b1:BeBot_SW and b2:BeBot_SW. Both component instances embed the component instances exp:Exploration and nav:Navigation as already explained for the component instance b4:BeBot_SW of Figure 5.25. b1:BeBot_SW and BeBot_SW additionally contain the component instances cc:Collision_Control and bbo:BeBot_Observer. An assembly connector instance derived from the Real-Time Coordination Pattern Distribution connects exp:Exploration and cc:Collision_Control. The discrete port instance master of the component instance exp:Exploration implement the single-role master. The discrete port instance slave of the component instance cc:Collision_Control implements the single-role slave. An assembly connector instance derived from the Real-Time Coordination Pattern DistancesTransmission connects the cc:Collision_Control and the bbo:BeBot_Observer. The discrete port instance receiver of the component instance cc:Collision_Control implements the single-role receiver. The discrete port instance sender of the component instance bbo:BeBot_Observer implements the role sender. An assembly connector instance derived from the Real-Time Coordination Pattern PositionTransmission connects the bbo:BeBot_Observer and the nav:Navigation. The discrete port instance receiver of the component instance bbo:BeBot_Observer implements the single-role receiver. The discrete port instance sender2 implements a sub-role of the multi-role.

Figure 5.29.: Concrete Syntax of a Component Instance Distributor
The component instances b1:BeBot_SW and b2:BeBot_SW differ by the communication to other component instances of the type BeBot_SW. b1:BeBot_SW of Figure 5.29 acts as a distributor. It communicates to other component instances of the type BeBot_SW via a multi-port that implements the multi-role distributor of the Real-Time Coordination Pattern Distribution. In our example, b1:BeBot_SW has two discrete port instances distributor1 and distributor2 that implement the multi-role distributor. This implies that b1:BeBot can distribute data to two client component instances of the type BeBot_SW. A delegation connector instance delegates these discrete port instances to the two discrete port instances distributor1 and distributor2 of the component instance bbo:BeBot_Observer. Further, it implements the multi-role distributor of the Real-Time Coordination Pattern Distribution.

Component instance b2:BeBot_SW has the discrete single-port instance client to communicate with another component instance of the type BeBot_SW. This discrete port instance implements the single-role client of the Real-Time Coordination Pattern Distribution. A delegation connector instance delegates to the discrete port instance client of the component instance bbo:BeBot_Observer.

Figure 5.30.: Concrete Syntax of a Component Instance Client
5.6. Deployment

The software of the BeBot is allocated on the hardware of the BeBot. The hardware of one BeBot consists of the two actor hardware nodes left_Engine and right_Engine, the sensor hardware node GPS, the communication hardware node WLAN, and the computational resource BeBot_ECU. Figure 5.31 shows the hardware of two BeBots, distinguished by the numbers 1 and 2. The allocation of the component instances bebot1:BeBot, bebot2:BeBot on the computational hardware nodes BeBot_ECU1, BeBot_ECU2 are shown within the corresponding 3D-boxes.

The continuous port instances speed_out_left, speed_out_right, and pos_in are connected to the hardware ports of the corresponding hardware nodes by the dashed allocation links. The hardware ports are connected by solid communication links to the hardware ports of the corresponding Engine hardware nodes.

The GPS hardware nodes send position data to the corresponding BeBot software components.

![Deployment of two Component Instances to Hardware Nodes](image)

Figure 5.31.: Deployment of two Component Instances to Hardware Nodes
Chapter 6.

Theoretical Background

In this chapter, we explain the theoretical background of MECHATRONIC UML. The concepts described in this chapter have a high impact on the modeling formalisms introduced in Section 3. These concepts, however, are not required for modeling with MECHATRONIC UML, but provide a deeper insight into the concepts of MECHATRONIC UML.

In this version of the paper, we only explain the fundamentals of the compositional verification approach of MECHATRONIC UML in Section 6.1.

6.1. Compositional Verification Approach

This section will explain the compositional verification approach supported by the MECHATRONIC UML as defined in [Gie03]. The general idea of the approach is to verify each Real-Time Coordination Pattern (see section 3.1), simply called pattern in the following, and each component (see section 3.4) on its own and then compose the patterns and components to build the overall system while preserving the verification results. The advantage of verifying each pattern and component on its own is that the verification of the whole system is split up into several smaller verification steps that require less state space. Furthermore, due to performing several smaller verification steps the overall verification becomes scalable and, hence, also feasible for realistic systems. In more detail, the verification approach consists of the following successive steps: First, all patterns are verified in isolation. Second, the roles of the patterns are refined to derive the behavior of the ports of the components. Concerning the refinement the approach assures that the verification results of the roles are preserved. Third, the components are verified in isolation. Finally, the components are composed to build the overall system. Here, the approach ensures that the properties verified for the patterns as well as for the components still hold under the composition. The different steps are explained in more detail in the following sub-sections.

6.1.1. Preliminaries

The state-based behavior descriptions we consider here must provide some kind of paths that represent executions of the description. A path is usually associated with a sequence of states
whereas each state can be reached from the previous one by a valid transition of the description. Let $D$ be a description. Then the set of valid paths of $D$ is denoted by $\Pi(D)$. A path is said to be deadlock-free if it is infinite. A description $D$ is said to be deadlock-free if all of its paths are deadlock-free which is denoted by $D \models \neg \delta$. Next we need the possibility to express that two behavior descriptions $D$ and $D'$ are executed in parallel. This is usually done by means of a composition-operator denoted by $\parallel$. The composition of $D$ and $D'$ is denoted by $D \parallel D'$. It can be imagined as the cross-product construction known from finite automata. Concerning the composition we distinguish two cases. The first one is that $D$ and $D'$ do not interact with each other. If this is the case then the paths of $D$ resp. $D'$ are still the same under composition. Let $\Pi_{D \parallel D'}(D)$ denote the paths that $D$ is able to execute while composed with $D'$. Then it holds (provided $D$ and $D'$ are independent) that $\Pi_{D \parallel D'}(D) = \Pi(D)$. Of course, due to symmetry reasons, it also holds that $\Pi_{D \parallel D'}(D') = \Pi(D')$. The second case is that $D$ and $D'$ do interact. Interaction usually means that certain transitions of $D$ and $D'$ are only allowed to be executed together. There exist different approaches to realize that two distinct transitions of two different descriptions execute together. The before mentioned real-time automata for example use inputs and outputs, timed automata use synchronization channels. We stick to inputs and outputs as generally understandable terms in the following. The important thing to notice here is that if $D$ is composed with $D'$ and $D$ interacts with $D'$ then $D$ can be seen as a restriction on the paths of $D'$. In other words the paths of $D'$ are restricted to those ones that can be executed together with $D$. This view of restriction of paths is important concerning deadlock-freedom. If $D$ and $D'$ are deadlock-free then in contrast the composition $D \parallel D'$ might have a deadlock. This is the case if the composition allows that one of the two descriptions, w.l.o.g assume $D$, reaches a state from which all outgoing transitions require a transition in the other description $D'$ that is not enabled i.e. that can not execute. Note that a deadlock can also be interpreted as cutting away an infinite suffix of an originally infinite path resulting in a finite path. Here, it is important to note that the resulting finite path is no path of the original behavior description $D$. So if the composition $D \parallel D'$ causes a deadlock more formally it holds that $\Pi_{D \parallel D'}(D) \not\subseteq \Pi(D)$. On the other hand if the composition $D \parallel D'$ does not contain a deadlock then it holds that $\Pi_{D \parallel D'}(D) \subseteq \Pi(D)$. The before mentioned property is important concerning the preservation of safety properties under composition. In the literature a safety property is defined as “something bad will never happen”. If this statement is negated it sais that “something good happens all the time”. With respect to paths a safety property means that a certain property (“something good”) holds for all paths. These special properties are important in our context because if such a property $\varphi$ is satisfied for a single description $D$ and $D$ is composed with another description $D'$ and for the composition $D \parallel D'$ holds that it does not contain a deadlock then we know that $\Pi_{D \parallel D'}(D) \subseteq \Pi(D)$ holds and since $\varphi$ holds for all paths $\Pi(D)$ it will also hold for the subset $\Pi_{D \parallel D'}(D)$ and is hence preserved under composition. The only thing that has to be checked to ensure the preservation of $\varphi$ is deadlock-freedom of the composition. The property $\varphi$ itself does not have to be checked again. Besides composition we also want to express that one behavior description can partially mimic the behavior of another description. For this we introduce a refinement-relation. A description $D$ refines another description $D'$, denoted as $D \sqsubseteq D'$, if $\Pi(D) \subseteq \Pi(D')$. Note
6.1. COMPOSITIONAL VERIFICATION APPROACH

that refinement preserves deadlock-freedom as well as safety-properties, i.e. if \( D \sqsubseteq D' \) holds then \( D \models \varphi \land \neg \delta \Rightarrow D' \models \varphi \land \neg \delta \). Also note that a refining description might add arbitrary inputs and outputs. As long as these additional inputs and outputs do not cause a deadlock the description stays a valid refinement as only the set of possible paths is reduced.

The next important thing is to consider how composition behaves together with refinement. Assume we have a composition \( D_1 \parallel D_2 \) of two descriptions \( D_1 \) and \( D_2 \). Let \( D_1' \) be a third description which is a refinement of \( D_1 \), i.e. \( D_1' \sqsubseteq D_1 \). Now, suppose that \( D_1' \) does not add additional inputs or outputs that interact with \( D_2 \) but only those that are for interaction with other descriptions. Then the composition \( D_1' \parallel D_2 \) will still be able to perform the same paths as \( D_1 \parallel D_2 \) and it follows that \( D_1' \parallel D_2 \sqsubseteq D_1 \parallel D_2 \).

6.1.2. Putting it all together

Now we have all preliminaries to explain the compositional verification approach of the \textsc{mechatronicuml}. As we already know the structure of a \textsc{mechatronicuml} model is specified by patterns and components. A Real-Time Coordination Pattern consists of roles. These roles specify their communication behavior by Real-Time Statecharts. The roles are instantiated on components resulting in ports of the component. The Real-Time Statechart of the ports are refined versions of the roles statecharts with additional inputs and outputs in order to interact with each other. In the following we only speak of Real-Time Statecharts while keeping in mind that they represent the behavior descriptions we talked about before. The core idea of the approach is to verify each pattern and component of a \textsc{mechatronicuml} model on their own and then ensure the correctness of the system by a syntactically correct composition of pattern and components. Like this a verification step for the whole system can be avoided. The verification is devided into four steps.

**First step** In the first verification step each pattern is verified on its own. Let \( P_1, \ldots, P_n \) be all patterns that exist in the model. For a pattern \( P_i \) we denote its role statecharts by \( D_{P_i}^{1}, \ldots, D_{P_i}^{m_i} \). The additional Real-Time Statechart for the communication channel is denoted by \( D_{P_iC}^{P_i} \). Every Pattern \( P_i \) is assumed to have a constraint \( \phi_i \). Then for each pattern we verify that its constraint \( \phi_i \) and deadlock-freedom holds. More formally we verify \( P_i \models \phi_i \land \neg \delta \). The constraint \( \phi_i \) is only allowed to be a safety property in order to be preserved by composition and refinement.

**Second step** After verifying each pattern we switch to the instance view of the system. The instance view provides us with the information which pattern instances exist and on which component instances their port roles are instantiated. We assume that the inputs and outputs of each pattern instance (actually of their statecharts) have been consistently relabelled such that no pattern is able to interact with another pattern. Under this assumption the statecharts of all pattern instances can be composed while preserving the verified properties. Note that instantiation of patterns does not falsify the verified properties since a pattern instance is equal
to its corresponding pattern. We denote the instances of a pattern $P_i$ by $P_{[i,1]},\ldots,P_{[i,l_i]}$. The statecharts of the $k$-th instance of a pattern $P_i$ are denoted by $D_{[i,k],1}^P,\ldots,D_{[i,k],m_i}^P$, and its channel statechart as $D_{[i,k],C}^P$. Under the assumption that the inputs and outputs off all patterns are distinct the composition of all of their statecharts still satisfies deadlock-freedom and their respective constraints $\phi_i$, i.e.

$$D_{[1,1],1}^P\cdots D_{[1,1],m_1}^P\cdots D_{[1,1],C}^P\cdots D_{[n,1],1}^P\cdots D_{[n,1],m_n}^P\cdots D_{[n,1],C}^P = \phi_1 \land \ldots \land \phi_2 \land \neg \delta$$

**Third step** In the next step we re-order the role instances such that all role instances that belong to a component instance are grouped together. To be able to represent all role instances of a component in a consistent manner we introduce new names for them. Let $C_1,\ldots,C_n$ be the component instances of the system. Then the statecharts of the role instances of a component instance $C_i$ are denoted by $D_{[i,1],1}^C,\ldots,D_{[i,1],m_i}^C$. This does not change anything concerning the role instances. We are still speaking of the same role instances that we spoke of in the previous step. The re-ordering of the statecharts does not change anything concerning the satisfaction of the verified constraints, i.e.

$$D_{[1,1],1}^C\cdots D_{[1,1],m_1}^C\cdots D_{[1,1],C}^C\cdots D_{[n,1],1}^C\cdots D_{[n,1],m_n}^C\cdots D_{[n,1],C}^C = \phi_1 \land \ldots \land \phi_2 \land \neg \delta$$

Next we modify the role instance statecharts of each component such that they do interact with each other. We could also introduce a special *synchronization statechart* that intermediates between them. Regardless of how this interaction is realised the important thing is that the modified roles are still a valid refinement of the original ones which means that each of the role statecharts is still able to perform a subset of the paths it could perform before the modification. This refinement has to be checked explicitly. As long as we do not change anything of the statecharts except for additional inputs and outputs this check can be performed by checking for a deadlock. Note that this deadlock check does not have to be performed for the whole system but only locally for each of the component instances. A bit more formally for a component instance $C_i$ let $M_i^C$ be the over-all behavior of $C_i$ i.e. the modified role instances possibly together with a synchronization statechart. Then it has to be checked that $M_i^C \subseteq D_{[i,1],1}^C\cdots D_{[i,1],m_i}^C$ holds. If the refinement is fullfilled and we sticked to the rule to only add inputs and outputs to create interaction between the statecharts within the component instance we can replace the independent roles $D_{[i,1],1}^C\cdots D_{[i,1],m_i}^C$ by the refined roles $M_i^C$. Here we apply the fact from the preliminaries section that if $D_1 \subseteq D_1'$ and $D_1$ does not add any inputs and outputs that interact with another description $D_2$ then it follows that $D_1\|D_2 \subseteq D_1'\|D_2$. Here $M_i^C$ is $D_1$, $M_i^C \subseteq D_{[i,1],1}^C\cdots D_{[i,1],m_i}^C$ is $D_1'$ and $D_2$ is the whole rest of the system. If we exchange all port roles by the refined versions we get that the whole system which now includes interaction within the component instances still fullfills the verified properties, i.e.
Fourth step In the fourth step properties concerning the behavior of the component instances are verified. These properties are checked locally for each component instance on its own. They might express certain dependencies between the port roles for example if the statechart of port role $A$ is in state $foo$ then the statechart of port role $B$ has to be in state $bar$. As long as these properties are safety properties, i.e. must hold for all paths, they will be preserved by the composition because we already know from the previous step that our system is deadlock-free which again indicates that a subset of all possible paths of each component instance is preserved under composition and hence safety properties will also be preserved.
Chapter 7.

Related Work

**MECHATRONIC UML** is a language for modeling and analysis of component-based software designs of reconfigurable mechatronic systems. As mentioned earlier, mechatronic systems contain elements developed by different engineering experts, namely electrical, control, mechanical and software engineering. While **MECHATRONIC UML** is mainly focused on the software engineering aspects, it nevertheless reflects relationships to other engineering domains to some extend. Examples are continuous components to model controllers or hardware components for mechanical and electrical elements.

**MECHATRONIC UML** especially focuses on the specification of real-time behavior for components and ports, communication protocols adhering to real-time requirements, and run-time reconfigurations. As a consequence, related work stems from the following areas:

- Specification languages for communication protocols
- Integrated specification languages for systems engineering. Examples are SysML, Modelica, or MATLAB/Simulink with Stateflow.
- Process models for systems engineering
- Software Component Models for embedded real-time systems like ROBOCOP, SOFAHI, or Progress.
- Specifications of reconfigurable systems. Examples are Architecture Description Languages (ADLs) for self-* systems like Dynamic Wright.
- Formal models for specifying real-time behavior. Examples are Timed Automata or Time Process Algebras.

In the following, we will discuss each area in detail. However, please note that the discussion is subject to further extensions in future versions of this document.

### 7.1. Contract-Based Design

Among others, two goals of **MECHATRONIC UML** are (1) that components can be developed independently from each other and (2) that a component can be easily exchanged by another
component. Therefore, our ports must fulfill fixed contracts with their environment. As there exist several definitions for contracts, we follow the definition of Giese et al. [GV03] that states: “A contract unambiguously describes the assumed and guaranteed syntactical, behavioral, synchronization, and quality-of-service characteristics of an associated component interface.”

In contrast to Szyperski [Szy98], we do not distinguish between provided and used contracts, because in our application domain it is often the case that a component port provides and uses contracts. This leads to cyclic dependencies between components resp. their ports which easily lead to deadlocks and must therefore be synchronized [GV03].

7.1.1. Contract Levels

Beugnard et al. [BJPW99] define four levels to distinguish between the several kinds of a contract. In the following, we will briefly describe these levels and state how MECHATRONICUML supports these levels.

The most basic level is the syntactical level that specify the operations a component can perform, the parameters a component requires, and exceptions a component may raise. In MECHATRONICUML, we support this level by specifying at most one sender and receiver message interface per port. Moreover, MECHATRONICUML allows to define a message buffer size.

The second level is the behavioral level. It describes pre- and post-conditions as well as invariants. For example, Meyer [Mey92] defines in its programming language Eiffel preconditions and post-conditions for methods, and invariants for classes. Preconditions must hold before the execution of a method, post-conditions hold after the method execution, and invariants of a class hold in all states of the class. MECHATRONICUML supports this level by defining a Real-Time Coordination Pattern for specifying all legal sequences of message exchange between the two roles. This behavior is defined by one Real-Time Statechart per role. Furthermore, preconditions, post-conditions, and invariants can be further restricted by adding guards, clock constraints, and clock invariants to each Real-Time Statechart. Using formal verification techniques (e.g., the Uppaal model checker), additional conditions and invariants can be proven.

The third level is the synchronization level. It defines the dependencies between methods like sequences and parallelism. In MECHATRONICUML, this level is supported by using synchronization channels to synchronize the port-statecharts of an atomic component. Therefore, the order of consuming and sending messages can be synchronized. Moreover, a Real-Time Coordination Pattern between two ports of two components may send and receive messages in parallel. This can be synchronized using synchronization channels, too.

The forth level is the quality-of-service level. It covers aspects like availability, throughput, and latency. MECHATRONICUML currently allows to specify the connector-assumptions message latency and message loss. A Real-Time Coordination Pattern that is used at this connector has to be compliant to these assumptions.
7.1.2. Automata Models for Supporting the Contract Levels

In the following, we present automata models that support at least the syntactical and the behavioral level for specifying component contracts. First, we will present existing untimed automata models. A second section for presenting timed automata models will be provided in a future release of this document.

7.1.2.1. Untimed Automata Models

De Alfaro and Henzinger propose Interface Automata [dAH05] that are defined as formalized rich message-based interfaces for software and hardware components. These automata allow to check if two components are compatible with each other with respect to syntactical and behavioral characteristics. They further enriched these automata by synchronicity, fairness, resources [CdAHS03], permissiveness [HJM05], and real-time constraints. Therefore, they support the four levels of contracts. In contrast to MECHATRONIC UML, they do not allow the reconfiguration of the automaton at run-time.

I/O automata [LT87, LSV01] enable to specify the interfaces of concurrent and distributed event systems that communicate synchronously. The behavior of each system component is modeled by one flat I/O automation. An I/O automation is an automaton with action labels on each transition. They support three types of transition actions: inputs, outputs, and internal actions. Inputs and outputs are used to communicate with the other components; internal actions may only be executed within a component. The external communication happens instantaneously. Inputs can not be blocked, thus the actions of the environment can not be blocked nor ignored. Therefore, the communication is synchronously. An output actions is handled as a broadcast. An I/O automata may be non-deterministic. I/O automata enable to verify that certain trace properties are fulfilled or not fulfilled. They enable both, safety (nothing bad happens) and liveness (something good eventually happens) trace properties. To conclude, they only support the syntactical and the behavioral level.

Team Automata [tBEKR03] are an extension of I/O automata. They only support the syntactical and the behavioral level. They support formal verification regarding safety and liveness properties.

Component-Interaction Automata [BCVZ05] focus on the component-interaction, but also on the synchronization. Thus, this automata definition supports the syntactical, the behavioral, and synchronization level. They support formal verification regarding safety and liveness properties.

7.1.3. SPEEDS

Benveniste et al. [BCF+08] define mathematical foundations and the design methodology of a contract-based model development for their framework that was developed in the SPEEDS project. Their field of application is the embedded system design. They focus on consistency and compatibility. In contrast to MECHATRONIC UML that only focuses on the application
layer, they defined a contract layer that contains the functional layer, the ECU layer, and the hardware layer. Like MECHATRONIC UML, they use timed automata for describing the protocol behavior, too. Moreover, they use refinement relations to weaken the assumptions and strengthen the guarantees. Though, they do not focus on an automatic and scalable verification. Furthermore, they do not consider the reconfiguration of the protocol at run-time.

7.2. Behavioral Connectors

In MECHATRONIC UML connectors specify a communication behavior using Real-Time Coordination Patterns. This behavior must be implemented by the connected ports. The related work to this topic is presented in the following.

7.2.1. Higher-Order Architectural Connectors

Lau and Wang [LWF03] developed a notion of higher-order architectural connectors that take other connectors as parameters. Therefore, they allow to compose connectors by services like protocols for asynchronous buffered message communication and fault-tolerance mechanisms. Their goal is to improve the reuse and the incremental development of connectors and to ease creating complex protocols.

In MECHATRONIC UML, assemblies are linked with pre-defined Real-Time Coordination Patterns. These protocols specify the state-based behavior of each role and define quality of service parameters like delay and message loss. The reusability of a Real-Time Coordination Pattern is limited, because it assumes concrete timing-requirements. Though, we developed design pattern for our Real-Time Coordination Patterns [DHT12] to increase the reusability of recurring design problems regarding the protocol specification. Currently, there exists no approach to compose a Real-Time Coordination Pattern by other Real-Time Coordination Patterns. In contrast to Lau and Wang, MECHATRONIC UML supports the reconfiguration of the connector at run-time.

7.2.2. Reo

The exogenous coordination language Reo [ABRS04, Arb06] enables the compositional construction of component connectors. The authors have developed constraint automata to specify the behavior of Reo connectors. The specification of components that are using the connectors for cooperation and communication is not the focus of Reo. Therefore, Reo regards them as black-box-components. MECHATRONIC UML also supports the integration of black-box-components, but the standard case are white-box-components. As in MECHATRONIC UML, component instances of Reo are running in parallel and do not know the component instances they are communicating to, but only that they comply to the protocol.

In the following, we will compare the the connector definition of MECHATRONIC UML and Reo. An assembly instance in MECHATRONIC UML specifies the communication between ex-
7.3. MULTI-AGENT-SYSTEMS

Exactly two component instances. But in Reo, a connector specifies the communication between two or more component instances by composing atomic connectors (they are called channels and connect exactly two component instances) to hierarchical connectors. Among others, Reo channel support synchronization, asynchronous communication, buffering, ordering, computation, data retention, and data loss. Except of synchronization and computation, MECHATRONICUML also supports these services. Like MECHATRONICUML, Reo connectors do not inspect or change the communication content.

As in MECHATRONICUML, each connector of Reo is linked to a certain coordination pattern. Though, they have different semantics. In MECHATRONICUML, Real-Time Coordination Patterns have roles that specify the communication behavior a connected port has to implement. Therefore, the connector has no own coordination behavior but assumes certain QoS-characteristics like message loss and message delay. In Reo, the connector has an own executable behavior specification that is described using the aforementioned constraint automata.

7.3. Multi-Agent-Systems

In the area of multi-agent systems, there exists the FIPA-ACL standard for describing the agent communication [Fou02], which is based on speech-act theories and ontologies. A framework for this is Jade [BBCP05]. With AgentUML [BMO01], one can describe communication patterns. These are reusable protocols that can be described in protocol diagrams. Such diagrams extend UML sequence diagrams by agent roles, parametrized lifelines, and concurrent lifelines. However, a formal analysis is not possible. Multi-agent systems focus on autonomous problem-solving components. Though, they do not focus on the development of the network that is necessary for their communication or on the integration of other mechatronic disciplines like control engineering.

7.4. Specification Languages for Systems Engineering

There are several specification languages for systems engineering that allow holistic and integrated modeling of mechatronic systems. A recent survey on these approaches can be found in [GH06]. In the following the most related work to our approach is discussed.

7.4.1. SysML

SysML is an acronym which stands for Systems Modeling Language. It is developed by an informal association of tool vendors and industry leaders, which firm under the name SysML Partners\(^1\). The standard is currently available in SysML version 1.2 [Gro10a].

\(^1\)http://www.sysmlforum.com
SysML is defined as a UML 2.x Profile which extends, reuses, refines, and tailors UML. SysML extends UML by requirement diagrams and parametric diagrams. Parametric diagrams can show mathematical relationships between parameters or variables of a block. SysML reuses the UML concept of state machine, use case and sequence diagrams. The syntax and semantics of SysML activity diagrams is refined [JDB09].

SysML is developed to use model-based and driven systems engineering (MBSE, MDSE) [Fea98]. SysML addresses the holistic systems engineering development. SysML focuses on holistic modeling of mechatronic systems while MECHATRONIC UML focuses on specification and formal analysis of the discrete hard real-time software especially for safety critical applications in software intensive distributed systems. Although, we model the integration with the remaining system elements, they are not included in our models at that level of detail. A “block” in SysML can be compared with a component in MECHATRONIC UML. A main difference is that in SysML a “block” can be either a software or a hardware element and in MECHATRONIC UML a component is always a software element. Hardware elements are described by hardware nodes and are used to deploy software components to hardware (cf. Section 3.6). Since SysML is based on UML it inherits the problem of imprecise semantics from UML [HKRS05] in contrast to MECHATRONIC UML which has well defined syntax and semantics.

7.4.2. MATLAB/Simulink Stateflow

MATLAB is a tool suite for computing in systems engineering [Col07]. Simulink is a toolbox for graphical model-driven development of dynamic continuous and discrete systems, like an anti-lock brake system. The system which should be developed can be constructed, dimensioned, simulated, and analyzed with Simulink. The goal is a model of a system which represents its dynamic system behavior. A modeller models graphically block diagrams that “depicts time-dependent mathematical relationships among the system’s inputs, states, and outputs”.

Simulink block diagrams have a causal signal flow, which means that there is for each signal output a defined signal source and a time-dependent mathematical relationship. In MECHATRONIC UML an atomic software component is the corresponding element of a Simulink block. An atomic component is in contrast to a Simulink block an independent unit, which encapsulates its functionality and behavior.

Stateflow is a toolbox which extends Simulink by a finite state machine concept (FSM). The toolbox enables modeling of event-based reactive behavior specifications. The Stateflow finite state machines have concepts for hierarchical and parallel states and support the modeling of control flow as transitions between these Stateflow states. MATLAB functions can be invoked as actions when e.g. a state change is performed.

---

2http://www.sysmlforum.com/faq/relation-between-SysML-UML.html
3http://www.mathworks.de/products/simulink/examples.html
4http://www.mathworks.de/help/toolbox/simulink/ug/f7-5734.html
In contrast to MechatronicUML it is tedious to model asynchronous message-based communication protocols with real-time requirements [HPR+12]. Although, message-pools or buffers as described in Section 3.3.12 are not supported in Stateflow. The output events of Stateflow can only be used to exchange information between different Stateflow blocks without the buffering by the Stateflow receiver block. As a consequence the event is lost if the receiver doesn’t directly use the received events and cannot be used to synchronise systems. Message-buffering is very useful for the coordination of distributed mechatronic systems, as they cannot coordinate via shared variables, because they are physically separated. Nevertheless, it is possible to encode asynchronous message-based communication with a tedious combination of several linked Simulink and Stateflow blocks, but this is hard to maintain [HPR+12].

Further, Stateflow has only features for simple temporal logic operators, like the after() and before() operators. It is only possible to specify the time elapsed since activation of the associated state. Stateflow has no special constructs to specify clock variables which are independent of special states and make it possible to measure time intervals since their last reset like in Real-Time Statecharts.

Besides modeling features MATLAB/Simulink Stateflow provides simulators with solvers for ordinary differential equations (ODE). They support the discrete-time and continuous-time simulation of Simulink models. They offer to use variable time-step and fixed time-step integration methods. Further, code generators makes it possible to generate platform specific target code. For example TargetLink [KTS01] from the dSPACE company supports the generation of code for a couple of embedded real-time platforms. MechatronicUML can benefit from these features as they can be mapped to MATLAB/Simulink Stateflow. Currently, we are working on an automatic transformation from MechatronicUML to MATLAB/Simulink Stateflow.

### 7.4.3. Modelica

Modelica is a free object-oriented modeling and simulation language. The goal is to model physical phenomena with ordinary and differential algebraic equation, hierarchical system structures with a connection mechanism, and discrete system behavior with discrete events. The model can be automatically converted to sorted assignment statements as they are used in standard programming languages like C to simulate the system model [Elm78]. The language targets systems from the domain of “...mechatronics, including cars, aircrafts and industrial robots which typically consist of mechanical, electrical and hydraulic subsystems as well as control systems.” [EMO99] The language itself is independent of a concrete simulation environment. Modelica models consist of compositions of sub-models connected by connections that represent energy flow (undirected/ acausal) or signal flow (directed/ causal) [Fri04]. Modelica is inspired by mathematics and uses the concept of declarative programming by using equations.

Modelica does not contain a separated state-based modeling language like Stateflow or Real-Time Statecharts. It contains the library StateGraph2 which provides blocks to model
state-based behavior [OME+99]. This is similar to statechart elements which would be represented as UML classes. A concrete statechart is like a UML object diagram. As a consequence for each state and also for each transition a separate object exists which can be parametrised to encode e.g. guards. The resulting models have more objects than classical statecharts. Therefore, they are tedious to model and hard to maintain. Furthermore, like in Stateflow, it is very difficult to model asynchronous message-based communication and discrete behavior with real-time behavior.

Before Modelica code can be simulated it is compiled to some intermediate code, usually C code. This in turn will be compiled to machine code and executed together with a numerical ordinary differential equation solver or differential algebraic equation solver to produce a solution for variables as a function of time. Currently, we are also working on an automatic transformation to Modelica to simulate MECHATRONICUML models in interaction with other parts of a mechatronic system.

### 7.4.4. CHARON

CHARON is a high-level modeling language and design environment for embedded systems [ADE+01]. It allows for hybrid modeling, that is to model discrete as well as continuous software for the embedded systems. It supports hierarchical composition in architecture and in behavior. CHARON was developed at the Department of Computer and Information Science of the University of Pennsylvania under the direction of Rajeev Alur, with last activities in 2003.

For hierarchical architecture, components are described as agents. Composition, instantiation, and hiding is possible. The hierarchic behavior is described as modes, which are basically hierarchical state machines. Exceptions and history retention are supported. Each mode describes a continuous behavior, modelled with analog variables that flow continuously during continuous updates that model passing time [ADE+01]. The evolution of an analog variable can be constrained by differential constraints, algebraic constraints, and invariants. Switching modes is possible with discrete control over the modes.

CHARON provides a formally defined semantics and a refinement definition. The semantics is implementable because transition are non-urgent and in [AIK+03] time delays are assumed for sensing, computation, and execution.

Predicate abstraction is used to apply a finite state model checking approach to hybrid systems. Compositional model checking like in MECHATRONICUML is possible due to the formal refinement definition. There was also research in accurate and efficient simulation techniques.

In contrast to MECHATRONICUML reconfiguration as defined in [HPB12] is not possible.

### 7.4.5. Masaccio/Giotto

Masaccio [HMP01] is a high-level modeling language for hybrid system models (containing discrete as well as continuous parts) with the focus on formal verification against requirements.
Giotto [Hor03] is a high-level programming language with the focus on schedulability analysis against resources. Both languages were developed within the project FRESCO (Formal REal-time Software COmponents) at the EECS of the University of California at Berkeley.

Masaccio allows the parallel and serial composition of components with arbitrary nesting. An atomic component can be either discrete or continuous. Formal verification of the real-time hybrid models is supported due to the formal semantics. A refinement definition exists based on assume-guarantee reasoning. Thus, Masaccio supports compositional model checking like MECHATRONIC UML.

Pure Giotto programs can be synthesized from a Masaccio model. It is a time-triggered program with real-time semantics and consists of tasks, modes, and mode switches. The program is annotated with information about platform, scheduling, and communication. Giotto can be compiled to a given platform to be executed on a virtual machine. Given a partially annotated Giotto program, the compiler can determine the missing information based on schedulability analysis.

In contrast to MECHATRONIC UML reconfiguration as defined in [HPB12] is not possible.

7.5. Process Models for Systems Engineering

A process model for the development of mechatronic systems is defined by the VDI 2206 process which is a specialized adaptation of the V-model for mechatronic systems development [VDI04]. Based on the VDI 2206, the collaborative research centre SFB 614 created a new process model for mechatronic systems development [GFDK09].

7.6. Software Component Models

Lau and Wang [LW07] and Crnković et al. [CCSV07, CSVC11] survey software component models and classify them according to common characteristics. They do not restrict themselves to component models for a specific domain. However, Crnković et. al. [CSVC11] distinguish between general purpose component models, like e.g. CORBA or EJB, and specialized component models for particular domains. According to this survey, approaches in the second category usually aim at either business information systems or embedded real-time systems. Since MECHATRONIC UML targets the development of reconfigurable mechatronic systems that adhere to real-time constraints, we will focus on component models supporting real-time systems or run-time reconfiguration or both. The survey by Hošek et. al [HPB+10] exclusively focuses on component models of the latter kind.

A major influence during the development of MECHATRONIC UML was ROOM [SGW94], an object-oriented development methodology for real-time systems. The structure and behavior definition of ROOM are, as a result, very similar to MECHATRONIC UML. In ROOM, a system is composed of actors which are defined by actor classes. An actor specifies a port through which it may interact with other actors. The interaction is based on message exchange,
the messages are defined by a protocol class. Actors can be composed hierarchically of other actors. Then, the contained actors can interact by creating bindings between their ports, ports of a contained actor are delegated to the outside world by using relay ports. The behavior of each actor is defined by a ROOMChart which is a variant of Harel statemachines [Har87]. In contrast to MECHATRONIC UML, ROOMCharts only support simple timer events, but no clock concept. In ROOM, each actor defines a set a data classes which are used to store information. ROOM supports a reconfiguration concept by using replicated actors and replicated ports which are similar to multi-parts and multi-ports of MECHATRONIC UML. The behavior of a hierarchical actor is defined by a special behavior component which also executes the reconfiguration. In contrast to MECHATRONIC UML, ROOM does not provide a formal specification of replication actions, no controller integration, and no formal verification approach covering the real-time behavior and reconfiguration. Support for developing with ROOM was included in several CASE tools, most notably RationalRose Real-Time, implementing ROOM as a UML 1.4-profile called UML-RT [GS04]. Despite having had large influence on current approaches, ROOM was not explicitly aimed at supporting component-based development and is not included in any of the surveys mentioned above.

A second major influence for MECHATRONIC UML is UML2 [Gro10b] which is reflected by the name of the language. MECHATRONIC UML adopts concepts, terminology, and parts of its concrete syntax from the UML and enriches them with semantics and additional language features for the development of mechatronic systems. Since UML lacks formal semantics as well as features for specifying the real-time constraints of embedded systems, it cannot be used directly for developing the software of a mechatronic system. In literature, there is no consensus whether UML is a component model. While [LW07] regards UML 2 as a component model, [CSVC11] does not, because it does not directly support component-based development or components as "executable units".

One of the component models with the highest practical relevance in the domain of embedded real-time systems is the AUTomotive Open System Architecture (AUTOSAR) [GbR08]. AUTOSAR is an open industry standard for the development of automotive systems. The standard is in ongoing development by all major automotive OEMs and their suppliers as well as other partners. AUTOSAR based systems can be developed with a lot of different tools like Vector DaVinci, dSPACE SystemDesk, Etas ASCET or Artop. Artop is an implementation of the common base functionality of AUTOSAR which is based on Eclipse. It is available for free for all AUTOSAR members. Like MECHATRONIC UML, AUTOSAR defines a concrete development methodology and a component model. Components can be connected via different ports specified by AUTOSAR Interfaces. AUTOSAR distinguishes between client-server Interfaces for set of operations and sender-receiver Interface for data-oriented communication. Specifying component behavior is out of the scope of AUTOSAR. Therefore, external tools like Matlab Simulink are required. In MECHATRONIC UML, the behavior specification is integrated in the development methodology and can be done through the use of Real-Time Statecharts.

The reconfiguration approach of MECHATRONIC UML is inspired by the reconfiguration concepts of Fractal [BCL+06] which have been extended to distributed execution
in [BHR09]. In Fractal, components consist of a so-called membrane providing control interfaces and a content area containing embedded components. The membrane contains controllers that delegate interfaces to embedded components and perform reconfiguration. In [BHR09], the membrane is extended by a non-functional port to receive reconfiguration calls. Reconfiguration operations are specified using a textual language called FScript [DLLC09]. We adopted the concepts of a reconfiguration executor and of non-functional ports. We extended the remote reconfiguration invocation significantly. In contrast to their approach, we support a higher level modeling language for modeling reconfiguration rather than implementing it as a script. THINK [AHJ+09] provides a C-implementation of the Fractal component model which targets embedded real-time systems. THINK allows to annotate non-functional properties at architectural elements and defines a reconfiguration view for components. The necessary code for executing reconfiguration is provided by their code generation framework. They distinguish two conditions for quiescent states which are at a very low-level of abstraction and not directly applicable to MECHATRONIC UML.

SOFA-HI [PWT+08, PKH+11] is an extension of the SOFA 2.0 component model [HP06, HPB+05] for embedded real-time systems which is specified as a profile of SOFA 2.0. SOFA 2.0 is a component model for business information systems. Like MECHATRONIC UML, it supports the definition of hierarchical components as well as their reconfiguration at runtime. Runtime reconfiguration includes creating and deleting component instances and connector instances as well as interfaces of components. SOFA-HI adds the ability to model real-time behavior of components to SOFA 2.0, but it restricts the reconfiguration capabilities of SOFA 2.0. SOFA-HI supports run-time reconfiguration only by replacing components by other components. Component instances may also be replaced by a newer version of the component which still implements all necessary interfaces. In contrast to MECHATRONIC UML or SOFA 2.0, modifications of the architecture, i.e. creating and deleting components and connectors, are not supported.

SaveCCM is the component model of the SaveCCT component technology which provides a component-based development of embedded automotive software [ÅCF+07]. The SafeCCM component model consists of components, switches and assemblies. Components specify a set of ports. The approach distinguishes between data ports that provide data to be read or written and trigger ports that activate a component. As long as they are not activated, components remain passive. A component is implemented by a C-function or several interacting tasks implemented in C. Switches enable to change the interconnection of components. They specify guard conditions which cause data or control flow signals to be delivered to another component. Assemblies are a hierarchical composition of connected components and switches. "An assembly generally does not satisfy the requirements of a component" [ÅCF+07], but it provides a means for hierarchical structuring. SaveCCM supports two types of connections: immediate connections that are loss-less with no delay and complex connections that may have information loss and delays. The formal semantics of a SaveCCM model is defined by timed automata. Formal verification is achieved by a mapping to timed automata and finite state process models. An implementation of SaveCCT is given by the Safe-IDE [SPCH08]. In con-
trast to MECHATRONIC UML, SaveCCM does not support complete hierarchical components, message-based communication and structural reconfiguration.

The commercial Rubus component model [HMTN+08] for embedded real-time software for vehicles, which is developed by Arcticus Systems, is based on the same foundations as SaveCCM [ÅCF+07]. As in SafeCCM, components specify data and trigger ports with the same semantics as in SaveCCM. The component model distinguishes between assemblies and composites. Assemblies are un-dividable units of deployment whereas composites consist of parts that may be deployed to different ECUs. As in SafeCCM, neither assemblies nor composites are considered as components. Rubus uses so-called modes which define a mode of operation, e.g., initialization or shut-down. A mode applies to one ECU, only, and defines the executed behavior. The supported real-time annotations cover deadlines, offsets and period jitter for components. The approach enables to specify models in a graphical syntax. Formal verification and simulation are supported. In contrast to MECHATRONIC UML, the approach does not consider complete hierarchical components, message-based communication and structural reconfiguration.

CompoSE [KKTS09] is an approach for modeling embedded systems based using several modeling languages from different domains. It is based on a host language that is defined by a component model supporting basic and hierarchical components. The approach supports three kinds of unidirectional ports: flow ports (comparable to signals), event ports (asynchronous events), and control flow ports. By using special guest language components, it is possible to integrate components which a realized in other languages as, e.g., Simulink. In [ASTPH11], a reconfiguration approach based on CompoSE is introduced. There, each component has a set of configurations. In a basic component, a configuration consists of a combination of ports as well as a computation that defines its behavior. In a hierarchical component, a configuration consists of a combination of ports as well as a configuration of contained component instances. A reconfiguration switches between these predefined configurations. Reconfiguration is only triggered based on the quality of the flow signals and it does not consider a fading or flat switching [OMT+08] between to output ports. In contrast to MECHATRONIC UML, the approach does not consider message-based communication protocols or changing communication topologies.

MyCCM-HI [BFHP09] is a component model for embedded real-time systems that is based on the Lightweight CCM standard which is now part of the OMG Corba Component Model (CCM, [Gro11b]). In contrast to MECHATRONIC UML, models for MyCCM-HI can only be defined using a textual syntax, not in a graphical IDE. These models are, on the one hand, used for direct C code generation of the application code and, on the other hand, they are transformed into models of the architecture analysis and design language AADL [SAE09] for generating a middleware for the system. Special focus is put on the specification analysis of mode changes where the system switches between different configurations. The approach, however, does not consider reconfiguration of hierarchical component structures or a distributed execution of the reconfiguration.

Robocop [Maa05] is a robust component model for consumer electronic products designed by ITEA project. The project targets the development of “an open-component-based frame-
work for the middleware layer in high-volume embedded devices that enables robust and reliable operation, upgrading and extension, and component trading [Maa05].” MECHATRONICUML in contrast targets the development of the application layer in embedded devices. Robocop offers a infrastructure in from of frameworks for design, build, trade, and update of middleware. They provide a development-, a run-time-, a download-, and a resource management-framework. Robocop defines a component as a collection of *modell/view* and relations between these models. Anything which hold aspect information about a component is called a model. Robocop differs the development level from the executable level of components. At the development level the modeler defines the collection of models/views on a component and the executable level the modeler defines explicit dependencies, dynamic third party binding, and a lifecycle properties of a component. The executable component model is based on OMG CORBA and Microsoft COM. An application at run-time consists of a composition of several executable components and the Robocop run-time environment which takes care of the creation component lifecycle. Interfaces define required and provides services. In contrast to MECHATRONICUML the concrete development of components and support for a development language, a methodology, and tool support is out of scope in Robocop.

**PECOS** [GCW02] is a component model for embedded real-time software of field-devices. Field-devices are devices which control local actors, e.g., valves, or monitor their environment by collecting sensor values. A PECOS component can only be deployed on a single hardware node. PECOS distinguishes so-called *Active, Passive* and *Event Components*. Active components run in an own thread, passive components do not have an own thread, and event components reacts on external triggers, like time ticks or interrupts from hardware components. Components have ports to get input data and ports to write output data. In contrast to MECHATRONICUML, connected ports do not interact via messages. They have a shared memory with read and write access. A PECOS component is either directly implemented via C++/Java or it is composed of other components. PECOS does not support hierarchical component types directly, but it offers the possibility to specify templates which contains *abstract components* which can be later filled with concrete components. The modeller has to specify for each component a cycle time, which determines in which interval they have to be scheduled, and a worst case execution time (WCET), which specifies the maximum time that the execution of a component needs. As cycle time and WCETs are not platform independent, the PECOS component model is also not platform independent like MECHATRONICUML. PECOS uses Petri nets to represent the execution model of composite components and to reason about real-time constraints and to generate schedules which meet deadlines. The instance configuration of components, ports and connectors is fixed at compile time. Therefore, components, ports, or connectors cannot be created or destroyed at run-time to reconfigure the software. PECOS is not actively developed anymore and provides only incomplete tool support [HPB+10].

A component model for consumer electronic software developed by Philips is **Koala** [vOvdLKM00]. Koala supports a separation of components from the configuration development in one model. The component model differs between base components that realize functions and compound components, embedding other components. Both are described
by a component description language. Components are connected through requires and provides interfaces, whereas each requires interfaces must be connected with one provides interfaces. Furthermore, there are also option and diversity interfaces as well as switches, which are used for system configuration. The assemblies are fixed at compile time, so that run-time reconfiguration is not supported. In contrast to MECHATRONIC UML, the system configuration is done within the component model. Therefore, interfaces and switches are used to initialize components on the one hand and for component binding on the other hand.

BlueArX [KKHH08] is a component model developed by BOSCH for describing engine control systems in the automotive domain. The component model defines atomic components which have an implementation as well as structured components. The latter can consist of several other atomic or structured components. The communication between components is accomplished by using import and export interfaces which defines global variables and access properties. The software behavior of components is described by signal flows which can depend on modes. These flows can be visualized for whole systems to foster a better understanding of dependencies of interfaces. In MECHATRONIC UML, the behavior specification can be done through the use of Real-Time Statecharts which allow a specification of the internal component behavior and Real-Time Coordination Pattern for communication behavior. Modeling control flows are also supported by history elements, entry-points, and exit-points in the Real-Time Statecharts.

The ProCom [VSC+09, BCC+08] component model, developed within the Progress research center, targets the development of automotive applications. It defines two layers of components: on the lower-level layer passive components are defined, each offering a set of services that can be called through a "triggering port". The higher layer, on the other hand, consists of active components representing concurrently running subsystems, communicating with asynchronous messages over channels. Thus, this higher layer of ProCom is quite similar to our own notion of components, while we do not currently consider anything corresponding to the lower layer. For ProCom a prototypical Eclipse-based IDE is available, however [HPB+10] notes that analysis methods and deployment tools are not yet implemented. ProCom uses a behavior model called REMES [SVP09, VSCS10], which, like our Real-Time Statecharts, is a hierarchical state-based language semantically relying on timed automata. Contrary to MECHATRONIC UML, resource usage is explicitly modeled and analysed, using a mapping to "priced" timed automata and a variant of the tool UPPAAL [BDL04] for model checking them. However, explicit support for dynamic reconfiguration is not provided.

Pin [HIPW05] is a component model for embedded real-time systems. Components consist of a container and a custom code section. The container is prefabricated by the framework and provides the interfaces of a component. The custom code is the user-specified component behavior which is modeled by UML Statecharts [Obj09]. The custom code must only specify interfaces to the container. Components are connected by assemblies which employ either synchronous or asynchronous communication. The assemblies are fixed at design time, i.e., run-time reconfiguration is not supported. In contrast to MECHATRONIC UML, the real-time behavior is provided by an underlying, commercial run-time environment and not part of the modeling language.
7.7. Specifications of Self-Adaptive Systems

In this section, we discuss related from the area of self-adaptive systems. Related work in this area mainly originates from the research on architecture description languages (ADLs). First, we discuss the relationship of MechatronicUML to several reference architectures in Section 7.7.1. Second, we relate MechatronicUML to modeling approaches for self-adaptive systems in Section 7.7.2. Finally, we introduce several run-time framework for self-adaptive systems in Section 7.7.3.

7.7.1. Reference Architectures for Self-Adaptive Systems

In the domain of mechatronic systems, Hestermeyer et al. introduced the operator-controller-module (OCM) as a reference architecture [HOG04]. The OCM consists of three layers: the controller, the reflective operator, and the cognitive operator. The controller layer contains all controllers of the mechatronic system that react to sensor inputs and produce actuator outputs. The elements of this layer are captured by continuous atomic components in MechatronicUML. The reflective operator monitors and reconfigures the controller layer. It operates event-based and adheres to hard real-time constraints. On this level, interaction between components that mandatory for the controllers are exchanged. The components of the reflective operator can be specified by MechatronicUML. On the top-most level, the cognitive operator is responsible for planning and self-optimizing the system operations. This level operates in soft real-time, the communication between systems may be specified by MechatronicUML.

In [IBM06], IBM introduces the autonomic computing architecture also known as MAPE-K. It mainly targets the domain of business information systems although it was inspired by robotic system designs. The lowest level contains the running system which is then managed by one or more autonomic managers. The autonomic managers implements an "intelligent control loop" [IBM06, pg. 10]. It consists of the four steps monitor (M), analyze (A), plan (P), and execute (E). The autonomic manager monitors the running system, analyzes the monitored data, uses planning to determine whether a reconfiguration is needed to improve the systems operations, and finally executes the reconfiguration plan. In addition, the autonomic manager may use a knowledge source (K) which stores and provides domain knowledge which is gathered and used by the aforementioned steps. In this architecture, MechatronicUML would mainly be used for monitoring and executing the reconfiguration. Analysis and planning is supported in a simple way by the manager of a structured component. Long-term planning operations, however, are not specified by MechatronicUML. In [HM08], Huebscher and McCann survey different applications of and contributions to MAPE-K.

Kramer and Magee describe a three layer reference architecture for self-adaptive systems in [KM07]. The bottom layer is the component control layer where the functional/operational behavior of the system is located. The middle layer is the Change Management Layer where change operations are located that change the structure of the component control layer. The change management layer either reacts to events from the lower level or it reacts to goal
changes from the upper level. The upper level is the goal management layer where the goals of the system are managed and long-term planning and AI algorithms are located which compute a solution on how to reach the goals. The architecture is similar to the OCM architecture. In MECHATRONICUML, we only consider the two lower levels which we distinguish for each structured component.

Caporuscio et.al. provide a meta-architecture for self-adaptive systems and a three layer run-time architecture in [CFG10]. The meta-architecture is based on requirements which the application needs to fulfill during run-time. It contains monitors for sensing the environment and the application itself thereby producing run-time models of both. A decision maker ensures that the application meets its requirements and generates respective actions to manipulate the application which are executed by an actuator. This is very similar to MAPE-K. The run-time architecture consists of three layers: implementation layer, proxy layer, and description layer. The description layer contains the models of the application and the environment as well as reconfiguration operation definitions. The implementation layer contains the components that implement the systems functional behavior. These components have a virtuals representation at the description layer. The proxy layer connects the implemented components with their virtual counterparts on the description layer. A reconfiguration changes the application model on the proxy level, the proxy level then chooses and executes the specific operations which need to be executed on the implementation level. The implementation is based on OSGi [All11].

In [WMA10], Weyns et. al. introduce the FORMS reference model for self-adaptive systems. FORMS is a formal reference model which is formalized using Z. In the model, a self-adaptive system is embedded into an environment. It consists of several base-level subsystems and reflective subsystems. The base-level subsystems contain the functional behavior, the reflective subsystems manage (a set of) base-level subsystems and/or reflective subsystems. This is similar to atomic and structured components of MECHATRONICUML. Each base-level subsystem has a domain model representing the data, each reflective subsystem maintains a model@runtime using reflection for reasoning about and changing the underlying subsystems. The authors show that they can model the MAPE-K architecture using their approach.

In [DDF⁺06], Dobson et.al. survey approaches and research challenges for connecting self-x systems using a network topology. Their survey only treats the lower networking layers and discusses, e.g., the design and analysis of physical networks as well as trust and security issues.

### 7.7.2. Modeling of Self-Adaptive Systems

In this section, we discuss modeling languages and methods for developing self-adaptive systems that originated from the research area of architecture description languages (ADLs, [SG96]). An architecture description language defines the constituents of a software system and their interconnections. In case of MECHATRONICUML, the architecture is given by the component model.
A proposal for an ontology for ADLs which we also use in this section was given by Garlan et.al. [GMW00]. They defines the following terms: a *component* is the building block of the architecture (although the term is less strictly defined as for component models), a connector defines the interaction of components, a system is a configuration of components and connectors, a property defines extra-functional properties and corresponding analysis tools, a constraint defines restrictions that the architecture needs to fulfill while it evolves, and styles are families of related systems.

A general approach towards the development and specification of self-adaptive systems is presented in [ZC06]. There, an adaptive program consists of several state-based programs and an adaptation set that defines transitions between these programs. Using the adaptation set, the adaptive program may switch between the programs during run-time. In contrast to MECHATRONICUML, the approach does not explicitly use a component model, but MECHATRONICUML uses some of the presented ideas such as separation of the state-based programs and the adaptation behavior or the usage of quiescent states for adaptation. In [ZGC09], a modular verification approach for such systems is introduced which has a similar objective as the compositional verification theorem introduced in Section 6.1.

In [GH04, RC10, RC09], several design patterns for self-adaptive system are introduced. These design patterns assume a separation of functional and adaptation behavior which is also used by MECHATRONICUML. [RC10] classifies the patterns into monitoring, decision-making and reconfiguration patterns. Several patterns regarding decision-making and reconfiguration have been used for the design of reconfiguration in MECHATRONICUML.

Modeling languages for the specification of dynamic software architectures are surveyed in [BCDW04]. The survey investigates especially the modeling expressiveness of different modeling languages. The survey identifies four main classes depending on the formal semantics of the approach. The considered classes are: graph transformation based languages, process algebra based languages, formal logics based languages, and languages that define an own semantics without using one of the aforementioned formalisms. Based on this classification, MECHATRONICUML belongs to the graph transformation based languages.

In the following subsections, we will investigate related work using the same classes as used in [BCDW04]. Finally, we discuss run-time frameworks that enable adaptation of a software.

### 7.7.2.1. Graph Transformation based Approaches

There are several approaches for modeling dynamic software architectures via graph transformations. Since MECHATRONICUML is also based on a graphs and reconfigurations are specified by corresponding graph transformations, the approaches in this section are closest to MECHATRONICUML regarding the modeling of reconfiguration.

The approaches by Le Métayer [LM98] utilizes a graph grammar with terminal and non-terminal symbols whose production rules are specified as graph transformations. The grammar specifies an architectural style while the elements of the defined language are architectures. For runtime reconfiguration, the approach uses a special coordinator component which executes all system reconfigurations. The reconfiguration operations are specified as graph
transformations. A verification procedure ensures that a valid architecture may only be trans-
formed into another valid architecture w.r.t. the architectural style. The coordinator compo-
nent is comparable to the reconfiguration execution controller of MECHATRONIC-UML. In
contrast to Le Métayer, there does not exist a central instance executing all reconfigurations of
the system, but each structured component has its own reconfiguration controller.

The approach by Hirsch et. al. [HIM98] uses edge-labeled hypergraphs to represent software
architectures. They use a graph grammar with three sets of rules for modeling architectures
and their evolution. The first set of rules defines the creation of a valid architecture. The
second set of rules defines the runtime evolution of the architecture. The third set of rules is
used to coordinate the communication of components. The approach does not need a central
controller, but does not specify when to execute which reconfiguration.

The approach by Taentzer et. al. [TGM00, Tae02] differentiates between a network graph
containing the system nodes and local graphs for each of the nodes. A change to a local graph
represents a local change inside a node, a change of the network graph represents a change
of the system structure. The reconfiguration operations are specified by double pushout graph
transformation rules [EEPT06]. They consider quiescent states as proposed in [KM98], but
they do not specify a logic which determines when to execute which reconfiguration.

A special case of a graph transformation based approach is the Chemical Abstract Ma-
chine (CHAM, [IW95]). In CHAM, a system is called a solution which consists of several
molecules. A solution may be transformed into another solution by applying a chemical re-
action rule. As in MECHATRONIC-UML, there exists an initial solution and a set of reconfig-
uration rules specified a reactions. In contrast to MECHATRONIC-UML, there is no explicit
control on when to execute which reconfiguration.

CommUNITY [WF02, WLF01] uses components and connectors to define a system archi-
tecture based on an architectural style. The system consists of components and connectors,
each of them executes a CommUNITY program. Components may interact synchronously
and asynchronously. The architecture is represented by a graph, reconfiguration is modeled by
graph production rules that transform one architecture into another one. The approach supports
basic reconfiguration commands to add and remove components as well as connectors. Ba-
sic reconfiguration commands are combined into composite commands and scripts that extend
the basic commands by control flow including conditions (if-then-else) and loops [WLF01].
Scripts define callable units of commands that may be called by the system or by users. The
approach uses a textual syntax for programs and reconfigurations and does not consider real-
time constraints. The textual scripts are translated to formal graph production rule by on dou-
ble pushouts automatically. An implementation of CommUNITY not supporting reconfigura-
tion is introduced in [OW07].

The approach by Kacem et.al. [KKJD06] uses UML 2.0 components to specify a system
architecture and defines profile classes for modeling reconfiguration operations. The opera-
tions are guard by using OCL constraints. The reconfiguration operations are specified by
simple graph transformation rules in a notation which uses the concrete syntax of UML com-
ponent diagrams. That is similar to the concrete syntax of component story diagrams used by
MECHATRONIC UML, but the approach does not support control flow or real-time properties of the reconfiguration.

In [BG08], Becker and Giese identify several requirements to modeling approaches for self-adaptive systems. In MECHATRONIC UML, we consider all these except modeling reconfiguration on different levels of abstraction. In their approach, the system architecture follows the one proposed in [KM07] which distinguishes a component layer, a change management layer, and a goal management layer. They use UML Class diagrams [Obj09] to model the system structure and story diagrams [FNTZ00] to model behavior formally. On the component layer, story patterns are used. On the change management layer and goal management layer, story diagrams are used. This is similar to MECHATRONIC UML which uses component story diagrams for modeling reconfiguration. They use the inductive invariants approach [BBG*06] for verifying their approach. Since their approach is based on plain graphs, it does not necessarily preserve component encapsulation and it does not consider quiescent states.

Bruni et. al. [BBGM08] summarize concepts and terms concerning dynamic software architectures and map them to a unifying framework based on typed hypergraph grammars. The considered terms are programmed dynamism, self-repairing, self-adaptive, ad-hoc dynamism and constructible dynamism. Based on these terms, MECHATRONIC UML provides a mixture of programmed dynamism where reconfiguration are preprogrammed at design time and self-adaptive behavior where reconfigurations are triggered based on a changing environment. The considered system model is component-based, a graph defines a configuration of the system and rules define the evolution of the system architecture. The approach distinguishes between reachable configurations and acceptable configurations. Regarding the application of a reconfiguration, the authors distinguish between constrained and unconstrained reconfiguration as well as self initiation and external initiation of the reconfiguration. In a constrained reconfiguration, a reconfiguration may only be executed if architectural constraints are satisfied or the components are in a quiescent state. In that sense, MECHATRONIC UML provides constrained reconfiguration with self initiation.

All of the mentioned approaches share the problem that they do not consider real-time aspects of reconfiguration. That means, they neither restrict the execution of reconfiguration to a specific time interval nor consider the time needed for executing the reconfiguration.

### 7.7.2.2. Process Algebra based Approaches

Approaches which utilize process algebras include Darwin [MK96], LEDA [CPT99], Pi-Lar [CdlFBS01], Dynamic Wright [ADG98], and the approach by Bartels and Kleine [BK11].

Darwin [MK96] is based on the $\pi$-calculus [MPW92]. In Darwin, reconfiguration is reduced to lazy instantiation of components and switching between several pre-defined connections, called bindings. Lazy instantiation means that a component is not loaded before its first usage. Switching between predefined connections is achieved by flags which indicate the component whose output is to be used (cf. [KM98]). In contrast to MECHATRONIC UML, it is not possible to load or unload components during run-time. The specification of components
is also restricted to flat components. The specification of behavior and reconfiguration does not respect real-time properties.

In [GMK02], Darwin components are extended by a run-time architecture. A run-time component consists of the component implementation, a configuration view, and a component manager. The component implementation contains the functional behavior of the component and specifies the interfaces of the component. The configuration view maintained by each component contains a model@runtime of the whole system which is updated via broadcast events after each reconfiguration. The configurations managers of each component perform reconfiguration of the overall system (to avoid a single point of failure when using a central reconfiguration manager). Then managers ensure that architectural constraints hold. In contrast to MECHATRONICUML, the approach violates component encapsulation and does not support structured components.

LEDA [CPT99] is also based on the $\pi$-calculus and provides a hierarchical component model in which components may be assembled by embedding instances of other components. In LEDA, each component implements a set of roles which may be used to connect components with each other. During run-time, new components may be instantiated and existing connections, called attachments, may be changed. The definition of attachments can be conditional, i.e., the current attachment depends on an if-then-else conditions. By changing the flags in the condition, the attachment may be changed during run-time. Since composite components are assembled from instances of other components, LEDA provides no clear distinction between types and instances of components like MECHATRONICUML. The reconfiguration capabilities of LEDA are comparable to MECHATRONICUML, but neither behavior nor reconfiguration respect real-time properties.

PiLar [CdlFBS01, CR10] uses a component-based approach which distinguishes single components and composite components which is similar to MECHATRONICUML. A single component is a collection of interfaces while a composite component is a configuration of component instances. Like MECHATRONICUML, PiLar explicitly distinguishes component types and their instances. The external behavior of components is specified by constraints which are modeled by using the process algebra CCS (Calculus of Communicating Systems, [Mil82]). PiLar supports the creation and deletion of component instances and connectors during run-time [CdlFBS01]. In addition, PiLar supports replacing a component by a newer version during run-time thereby updating the type and all it’s instances [CR10]. The adaptation operations are specified by constraints in terms of CCS as well. Thus, PiLar provides similar reconfiguration capabilities as MECHATRONICUML. In contrast to MECHATRONICUML, PiLar mixes component types and instances in different layers of the system. A component instances may occur on different layers within the system while their respective type is always part of the next higher level. Thus, types and instances may be contained in the same layer.

Dynamic Wright [ADG98] specifies architectures by architectural styles that define a set of flat component types and connector types which connect the components. The behavior of components and their ports is specified by CSP (Communicating Sequential Processes, [Hoa85]). They provide a separation of port behavior and component internal com-
putations which is comparable to the MECHATRONICUML component behavior specification. The definition of connectors, that specify two roles, is comparable to Real-Time Coordination Pattern. Dynamic Wright allows for creation and deletion of components during run-time. In addition, connections between components may be changed by using attach and detach operations. Reconfigurations are specified in a CSP-based textual language. Like in MECHATRONICUML, each component has an initial configuration and a set of reconfiguration operations.

In [BK11], Bartels and Kleine introduce a CSP-based framework for the specification of adaptive systems. The approach suggests a three layered system specification. On the first level, adaptive behavior and functional behavior are specified separately in terms on CSP-processes. Then, they use the CSP-refinement definition for deriving a implementation of the adaptive behavior on Level 2 and a complete system implementation on Level 3. Each configuration of the system is represented by a process which is guarded by a boolean predicate. They implement adaptation by changing the variables used in the predicates. All processes may be verified on their own, but also in combination with other processes. In contrast to MECHATRONICUML, the framework does not allow to structure a system hierarchically. For implementing reconfiguration, MECHATRONICUML uses a different approach by modifying the existing configuration instead of switching to a different process executing a new configuration. MECHATRONICUML, however, shares the idea of separating functional and adaptation behavior.

Khakpour et.al. introduce PobSAM (Policy-based Self-Adaptive Model) for modeling self-adaptive systems in [KJT+11]. A system consists of managed actors implementing the functional behavior of the system and autonomous managers that perform run-time adaptation. Both behaviors are specified by policies that are defined using the process algebra CA. Managers and actors interact via events which trigger the application of a policy. The managers may add and remove actors as well as connections. In contrast to MECHATRONICUML, actors and managers cannot be composed hierarchically and the approach does not consider real-time properties.

All of the process algebra based approach share that they define the architecture as well as the reconfiguration operations in a textual syntax.

7.7.2.3. Formal Logic based Approaches

Aguirre et al. [AM02] define a declarative language for the specification and reasoning of reconfigurable component-based systems. Components are represented by classes, units of modularization that contain data and behavior. Connectors, called “associations”, are defined by participants and a set of synchronization actions. The definition of hierarchical subsystems is supported. Properties over the system are defined a temporal logic for reactive systems [MP92] with extensions, e.g., a starting point in time. A calculus enables the reasoning about each level of subsystem.

Gerel by Endler et al. [EW92] is a language for the specification of selection forumlas on reconfigurations and reconfiguration preconditions. The system consists of program and
configuration components. Program and configuration components are types in the sense of common types as used in MechatronicUML that can be nested. The program is specified using a programming language, e.g., C++, that is extended by a common set of embedded communication primitives and interaction interfaces to other components. An interface is a set of ports described in a common Interface Specification Language. Configuration components are constructed from interconnected instances of program and/or configuration components. The properties described in Gerel allow reasoning about the reliability of the reconfiguration.

These two approaches [AM02, EW92] use a textual language to describe their system.

The Restore Invariant Approach by Nafz et al. [NSS+11] provides a method to give guarantees for the behavior of self-organizing systems. The most important concepts of the system behavior are specified using the Safety Analysis and Modeling Language (SAML) [GO10]. In SAML the system is described by finite state automata. These automata are executed synchronously with discrete time steps. State variables are updated according to transition rules. Transitions contain non-determinism and probabilistic choice. Constraints in the form of predicate formula specify unwanted system states - the invariants. Thus a corridor for the correct behavior of the system is defined. In the Restore Invariant Approach the system starts a self-organization whenever an invariant is violated such that the behavior stays in the corridor defined by the invariants. Reconfiguration is applied in the form of role changes from a set of predefined roles that describe tasks in the system, e.g., tighten screw.

This approach uses a graphical model to specify the most relevant parts of the system behavior.

In all approaches, constraints on the reconfigurations are specified formally to allow reasoning about the correctness or reliability of the reconfigurations is enabled.

### 7.7.2.4. Approaches Defining an Own Semantics

Acme by Garlan et.al. [GMW00] is an ADL with the intention to serve as a unifying basis for previously developed ADLs like Darwin or Wright. It uses components and connectors to define systems and provides the possibility to annotate non-functional properties and design constraints for components. A design constraint defines structural properties for the architecture using the Armani constraint language [Mon01]. Acme provides no behavior definition on its own. Reconfiguration of Acme architectures is given by the Plastik extension [JBCG05]. It allows to specify conditions using Armani which enable the execution of a reconfiguration. Reconfigurations are specified on the architecture level using Acme expressions and are mapped to reconfiguration scripts on the run-time level. The reconfiguration operations are then executed by calls to the API of the OpenCOM component model which provides an execution environment. The specifications in Acme, Armani, and Plastik rely on a textual syntax and enable to change the whole system architecture. Acme and Plastik, however, provide no support for hierarchical reconfiguration and real-time properties.

The approach by Chen et. al. [CHS01] provides adaptation for layered architectures and has originally been designed for an adaptive network stack. The approach uses adaptive components whose adaptation is controlled by an component adaptor module (CAM). The CAM may
7.7. SPECIFICATIONS OF SELF-ADAPTIVE SYSTEMS

switch seamlessly between several adaptation-aware algorithm modules (AAM) that provide different algorithms for solving the same task. The adaptation decision is based on fitness-functions that rate for each AAM how suitable it is in the current situation. The overall coordination of the system is managed by an adaptation controller. In contrast to MECHATRONICUML, the approach by Chen et. al neither supports loading or unloading of components nor allows reconnecting components during run-time. The use of a central adaptation controller for the whole system is also not suitable for MECHATRONICUML because it violates the encapsulation of the components. The adaptation controller, however, is comparable to the reconfiguration module of structured components.

Rainbow by Garlan et al. [CGS06] is a framework for the development of systems that improve themselves during run-time based on monitored system properties. So-called strategies for self-adaptation are specified by the textual language Stitch [CG12]. Adaptation is specified by operators that represent basic configuration commands. These operators are used to construct adaptation strategies.

7.7.3. Run-Time Frameworks for Self-Adaptation

In [DAO+11], Derakhshanmanesh et. al. introduce a run-time adaptation framework based on typed attributed graph transformations. The framework consists of a middleware providing access to the adapted software, a run-time model layer maintaining a graph-based model of the software, and an adaptation management layer storing the reconfiguration rules and controlling their application. The proposed architectural elements of the framework are similar to manager and executor used in MECHATRONICUML. In contrast to MECHATRONICUML, the framework is not component-based and it is implementation driven.

The framework by Vogel and Giese [VG10] proposes to use several models of the adaptive system at different levels of abstraction at run-time. Each model addresses a specific concern, e.g., performance or system failures. For each concern, several models at different levels of abstraction may exist, e.g., a low-level one for performing the structural changes and a high-level one for reasoning about the system. They use triple graph grammars [Sch95] for synchronizing the models and define a refinement between the different models that ensures that changes on abstract models are properly reflected on the more concrete ones. Using such an approach would be a legal extension to the manager which only maintains a low-level model of the components.

In [MBG+11], Ma et.al. introduce a framework for run-time reconfiguration that supports a transactional semantics for distributed reconfigurations. If a reconfiguration affects several components in the system, the reconfiguration is either performed completely or all changes are rolled back. The framework only applies reconfiguration to a system element if it is in a quiescent state in which reconfiguration is safe.

This section covers formal models for the modeling of real-time behavior. Depending on their representation of time these models are sub-divided into two categories. The first category covers models that represent time as a sequence of discrete steps or ticks. This time model is called *discrete time model*. The second category contains models that are based on a notion of time which assumes that time is continuously increasing. Here, time is modeled as the positive real numbers. This time model is called *continuous* or *dense time model*. For a deeper insight in time models the survey paper by [FMMR10] presents alternative definitions of time and its specification in models. Following the provided classification of time, Real-Time Statecharts are based on a dense time domain, i.e., the values of the clock are elements of \( \mathbb{R} \). As in UPPAAL, the time model is a metric time model that supports quantitative properties explicitly referring to values of clocks.

### 7.8.1. Discrete time models

In [EMSS91] the authors use temporal structures to describe real-time behavior. Temporal structures consist of states, a transition relation and a labeling function that labels the states with atomic propositions. Time is added to these temporal structures by the assumption that every transition represents one time step. An obvious shortcoming of this model is that if more than one time step passes during the transition from one system state to another auxiliary states have to be introduced to mimic this passing of time.

In the paper [Lam05] with the provocating title *Real-time model checking is really simple* Leslie Lamport introduces *explicit-time descriptions*. These descriptions are no new formalism to describe real-time behavior but rather a generally applicable concept to extend existing formalisms and languages with time. The idea is to introduce a variable *now* that represents time and a *Tick* action that increments *now* to represent the passage of time. Furthermore, lower time bounds for certain actions are expressed by allowing these actions only if *now* has a value greater than the bound. Upper time bounds are expressed by not allowing an action if the action would increment *now* such that the upper bound would be violated. Lamport argues that explicit-time descriptions have the advantage that no new formalism has to be introduced for dealing with real-time. This, in turn, enables the re-use of existing theoretical frameworks, proofs and model checkers. The concept can deal with discrete as well as with dense time where in a discrete time model the *Tick* action would increment *now* by 1 and in the dense model by any positive real number.

### 7.8.2. Dense time models

A common modeling formalism for the modeling of real-time behaviors are timed automata [AD90] which have been applied successfully in the past. Concerning the semantics of timed automata there is a broad variety of different interpretations. A thorough and recent survey over different kinds of timed automata is given in [WDR11]. **MECHATRONICUML**
itself utilizes the semantics of timed automata as implemented by the UPPAAL tool \cite{Yu03}. In \cite{DMY02b}, the timed automata used by UPPAAL are extended to hierarchical timed automata that are closest to the Real-Time Statecharts used in \textsc{MechatronicUML}. In addition to the concepts of hierarchical timed automata, Real-Time Statecharts support time consuming transitions as well as periodic do-actions for states.

In \cite{HA05} a formalism called \textit{interface duration automata} is presented. Several of these automata might communicate by sending and receiving \textit{actions} when performing a transition. The time instants when a transition might be performed is restricted by attaching a time interval \([l, u]\) to each transition where \(l\) and \(u\) are positive real numbers. These bounds are evaluated with respect to a \textit{configuration} \((s, d)\) where \(s\) is a state and \(d\) a positive real number describing how long \(s\) has been active since the last time it was activated. A transition can be performed if \(l \leq d \leq u\). For these automata the authors present an algorithm for deciding the emptiness problem to which the reachability and safety problem can be reduced. The authors argue that the most important advantage of the formalism is that it can be checked for emptiness with a complexity nearly the same as for the untimed case. Furthermore, they claim that although their formalism is simpler than timed automata a lot of real-time systems can be modeled and verified with it.

In \cite{MFR06} an model checking approach is presented that uses \textit{CSP-OZ-DC}, a combination of concurrent sequential processes (CSP) \cite{Hoa85}, Object-Z (OZ) \cite{Smi00} and duration calculus (DC) \cite{ZH04} to specify real-time systems. The part consisting of CSP and OZ represents the system model and the DC part represents the specification. The semantics of the model and the specification is defined by a mapping to \textit{phase event automata} (\textit{PEA}), a variation of timed automata that supports communication over events and variables with infinite data domains.

Other approaches for modeling real-time behavior include timed process algebras like, e.g., timed CSP \cite{Oxf92}.
Chapter 8.

Conclusions and Future Work

In this technical report, we presented a consolidated version of MECHATRONICUML. The current version of this technical report focuses on modeling the structural aspects of mechatronic systems using hierarchical components as well as modeling the state-based, discrete, real-time behavior of those components and their interaction using Real-Time Statecharts. Particularly, Real-Time Coordination Patterns were introduced for modeling the safety-critical coordination between mechatronic systems. These patterns enable the compositional verification of arbitrary large systems with respect to safety properties.

The BeBot running example (we refer to [Dre11] for a detailed presentation) as well as an industrial case study [Rie11] show that MECHATRONICUML is appropriate for the model-driven development of safety-critical mechatronic systems - in particular for systems of autonomous mechatronic systems. The compositional verification approach of MECHATRONICUML has been successfully evaluated in [GST+03]. Further evaluation activities will be performed in the course of the research project ENTIME [GSA+11].

Several parts of MECHATRONICUML are currently not included in this technical report. We refer the interested reader to the related publications until the next revision of this technical report. The semantics of a previous version of Real-Time Statecharts have been formally defined in [GB03]. Code generation from MECHATRONICUML models has been presented in [BGS05]. Furthermore, MECHATRONICUML supports the reconfiguration of component structures either by embedding component structures in states [GBSO04] or by specifying operational rules based on the graph transformation formalism [THHO08, EHH+11]. Finally, MECHATRONICUML supports a component-based hazard analysis approach [GT06, PST11, PSWTH11]. We will include these topics in upcoming versions of this technical report.

Acknowledgments

MECHATRONICUML is the result of many people’s work which we are greatly indebted to. MECHATRONICUML has been started by Holger Giese, now Professor at the Hasso Plattner Institut at the University of Potsdam, while he was with the Software Engineering Group at the University of Paderborn. He designed the MECHATRONICUML approach and developed the foundations. Thus, he greatly influenced the current state of MECHATRONICUML. Additional work has been done by the following doctoral students during their time at the Software Engi-
neering Group in Paderborn (in alphabetical order): Sven Burmester [Bur06], Tobias Eckardt, Stefan Henkler [Hen12], Martin Hirsch [Hir08], Claudia Priesterjahn, Julian Suck, Florian Stallmann [Sta08], Daniela Schilling [Sch06], and Matthias Tichy [Tic09].

We thank Jannis Drewello for specifying the application scenario and the structure and behavior models for the BeBot. We thank Boris Wolf for his comments on draft versions of the section on Real-Time Statecharts and Ingo Budde for generating the technical reference out of the meta model.
Chapter 9.
Bibliography

9.1. Own Publications


9.1. OWN PUBLICATIONS


9.1. OWN PUBLICATIONS


9.2. Bachelor Theses, Master Theses and Ph.D. Theses


9.3. Foreign Publications


**9.3. Foreign Publications**


9.3. FOREIGN PUBLICATIONS


9.3. FOREIGN PUBLICATIONS


[CdLG+09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan


9.3. FOREIGN PUBLICATIONS


9.3. FOREIGN PUBLICATIONS


[HP06] Petr Hnetyńka and Frantisek Plasil. Dynamic reconfiguration and access to services in hierarchical component models. In Ian Gorton, George Heineman, Ivica Crnkovic, Heinz Schmidt, Judith Stafford, Clemens Szyperski, and Kurt Wallnau, editors, Component-Based Software Engineering, volume 4063 of


9.3. FOREIGN PUBLICATIONS


9.3. FOREIGN PUBLICATIONS


9.3. FOREIGN PUBLICATIONS


9.3. FOREIGN PUBLICATIONS


9.3. FOREIGN PUBLICATIONS


Appendix A.

Meta-model Documentation

A.1. Package `modelinstance`

A.1.1. Package Overview

The package `modelinstance` defines the base classes for the FUJABA xmi format. In detail, it defines a root node and model element categories in order to serialize the model elements that may be contained in a FUJABA model.

Figure A.1.: Meta-Model of the `modelinstance` Package

A.1.2. Detailed Contents Documentation

A.1.2.1. Class `ModelElementCategory`

Overview

The `ModelElementCategory` contains all model elements of a FUJABA model that have the same type and will be opened by the same editor. A `ModelElementCategory` may only store subclasses of `NamedElement`.

Class Properties

Class `ModelElementCategory` has the following properties:

- `key : EString [0..1]`
  
  The uniquely identifying key of this category. The key of the category may be used by editors to register for the model elements contained in this section.

- `name : EString [0..1]`
  
  A human readable name for this category.
**Class References**  Class `ModelElementCategory` has the following references:

- `modelElements : ExtendableElement [0..+]` see Section A.13.2.2 on Page 267
  
  The ModelElements which are contained in this category. All model elements must be of the same type.

**Class Constraints**  Class `ModelElementCategory` has the following constraints:

- `ExclusivelyContainsValidElements`:
  
  ```plaintext
  self.modelElements->select (e | not isValidElement(e))->isEmpty()
  ```

### A.1.2.2. Class RootNode

**Overview**  The RootNode is the single root element of the XMI file which is generated for the FUJABA model.

**Class References**  Class `RootNode` has the following references:

- `categories : ModelElementCategory [0..+]` see Section A.1.2.1 on Page 185
  
  The model element categories which are contained in this RootNode.

- `ecoreDataTypes : EDataType [0..+]`
A.2. Package `muml`

A.2.1. Package Overview

This package is the base package for all MechatronicUML models, editors and algorithms. Plugins contributing to MechatronicUML should use `de.uni_paderborn.fujaba.muml` as a base package.

Note: This package does not contain any classes.

A.3. Package `muml::model`

A.3.1. Package Overview

The model package contains the core meta-model of MechatronicUML. The subpackages define the base classes for the component model, real-time statecharts, message interfaces and coordination pattern.

Note: This package does not contain any classes.

A.4. Package `muml::model::component`

A.4.1. Package Overview

The package components contains all classes for modeling atomic and structured components. Components are defined on the type level and may be instantiated in a component instance configuration.

A.4.2. Detailed Contents Documentation

A.4.2.1. Class `Assembly`

Overview This class represents an assembly connector. Assembly connectors connect the port parts of two component parts.

Class References Class `Assembly` has the following references:

- `coordinationPattern : CoordinationPattern [0..1]` see Section A.10.2.1 on Page 237
  The coordination pattern that defines the protocol of this assembly.

- `from : ComponentPart` see Section A.4.2.6 on Page 194
  The component part of the port part from which this assembly originates.
Figure A.2.: Meta-Model of the component Package
to : ComponentPart  see Section A.4.2.6 on Page 194

The component part of the port part to which this assembly leads.

Class Constraints  Class Assembly has the following constraints:

**NoSelfAssembliesForSinglePortsOfSingleParts:**

\[
\text{self}.\text{fromPort}.\text{cardinality}.\text{upperBound}.\text{value} \leq 1 \text{ and } \text{self}.\text{fromPort}.\text{cardinality}.\text{upperBound}.\text{value} \leq 1
\]

implies

\[
\text{self}.\text{fromPort} <> \text{self}.\text{toPort}
\]

**ValidContinuousPortDirections:**

\[
\text{not } \text{self}.\text{fromContinuousPort}.\text{oclIsUndefined()} \text{ and } \text{not } \text{self}.\text{toContinuousPort}.\text{oclIsUndefined()}
\]

implies

\[
\text{self}.\text{fromContinuousPort}.\text{kind} <> \text{self}.\text{toContinuousPort}.\text{kind}
\]

**AssemblyBetweenDiscretePortsRequiresCoordinationPattern:**

\[
\text{if } \text{not } \text{self}.\text{fromDiscretePort}.\text{oclIsUndefined()} \text{ and } \text{not } \text{self}.\text{toDiscretePort}.\text{oclIsUndefined()} \text{ then}
\]

assembly between two discrete ports requires a coordination pattern

\[
\text{not } \text{self}.\text{coordinationPattern}.\text{oclIsUndefined()}
\]

else

true

endif

**AssemblyBetweenDiscretePortsRequiresSameCoordinationPattern:**

\[
\text{if } \text{not } \text{self}.\text{fromDiscretePort}.\text{oclIsUndefined()} \text{ and } \text{not } \text{self}.\text{toDiscretePort}.\text{oclIsUndefined()} \text{ then}
\]

\[
\text{not } \text{self}.\text{fromDiscretePort}.\text{refines}.\text{oclIsUndefined()} \text{ and } \text{not } \text{self}.\text{toDiscretePort}.\text{refines}.\text{oclIsUndefined()}
\]

and

— both refinements must belong to the same pattern

\[
\text{self}.\text{fromDiscretePort}.\text{refines}.\text{coordinationPattern} = \text{self}.\text{toDiscretePort}.\text{refines}.\text{coordinationPattern}
\]

else

true

endif

**AssemblyBetweenDiscretePortsRequiresDifferentRoles:**

\[
\text{if } \text{not } \text{self}.\text{fromDiscretePort}.\text{oclIsUndefined()} \text{ and } \text{not } \text{self}.\text{toDiscretePort}.\text{oclIsUndefined()} \text{ then}
\]

\[
\text{not } \text{self}.\text{fromDiscretePort}.\text{refines}.\text{oclIsUndefined()} \text{ and } \text{not } \text{self}.\text{toDiscretePort}.\text{refines}.\text{oclIsUndefined()}
\]

and

— both ports should have different roles (unless the pattern has only one role)
(self.fromDiscretePort.refines.coordinationPattern.roles ->
size() = 2 implies (self.fromDiscretePort.refines.name
<> self.toDiscretePort.refines.name))

else
  true
endif

AssemblyBetweenDiscretePortsSameMessageInterfaces:

if not self.fromDiscretePort.oclIsUndefined() and not self.
toDiscretePort.oclIsUndefined() then
  --- message interfaces must be compatible
  self.fromDiscretePort.senderMessageInterface = self.
toDiscretePort.receiverMessageInterface
  and
  self.fromDiscretePort.receiverMessageInterface = self.
toDiscretePort.senderMessageInterface
else
  true
endif

ValidDiscreteInPortCombination:

not self.fromDiscretePort.oclIsUndefined() and self.
fromDiscretePort.isDiscreteInPort
implies (not self.toDiscretePort.oclIsUndefined() and self.
toDiscretePort.isDiscreteOutPort)

ValidDiscreteOutPortCombination:

not self.fromDiscretePort.oclIsUndefined() and self.
fromDiscretePort.isDiscreteOutPort
implies (not self.toDiscretePort.oclIsUndefined() and self.
toDiscretePort.isDiscreteInPort)

ValidDiscreteInOutPortCombination:

not self.fromDiscretePort.oclIsUndefined() and self.
fromDiscretePort.isDiscreteInOutPort
implies (not self.toDiscretePort.oclIsUndefined() and self.
toDiscretePort.isDiscreteInOutPort)

ValidContinuousInPortCombination:

not self.fromContinuousPort.oclIsUndefined() and self.
fromContinuousPort.isContinuousInPort

```
implies (                          
    not self.toContinuousPort.oclIsUndefined() and self. 
    toContinuousPort.isContinuousOutPort  
    or                                       
    not self.toHybridPort.oclIsUndefined() and self. 
    toHybridPort.isHybridOutPort           
)

ValidContinuousOutPortCombination:
    not self.fromContinuousPort.oclIsUndefined() and self. 
    fromContinuousPort.isContinuousOutPort   
implies (                                 
    not self.toContinuousPort.oclIsUndefined() and self. 
    toContinuousPort.isContinuousInPort     
    or                                       
    not self.toHybridPort.oclIsUndefined() and self. 
    toHybridPort.isHybridInPort             
)

ValidHybridInPortCombination:
    not self.fromHybridPort.oclIsUndefined() and self.fromHybridPort. 
    isHybridInPort                          
implies (                                 
    not self.toContinuousPort.oclIsUndefined() and self. 
    toContinuousPort.isContinuousOutPort   
)

ValidHybridOutPortCombination:
    not self.fromHybridPort.oclIsUndefined() and self.fromHybridPort. 
    isHybridOutPort                         
implies (                                 
    not self.toContinuousPort.oclIsUndefined() and self. 
    toContinuousPort.isContinuousInPort   
)
```

**Parent Classes**

- BehavioralConnector see Section A.4.3.2 on Page 192

### A.4.2.2. Class AtomicComponent

**Overview**  This class represents an atomic component. Atomic components must not be further sub-divided into component parts. In contrast to structured components atomic components own a behavior in form of a realtime statechart.

The different component types are implemented as a variation of the composite design pattern. Concerning the composite pattern this class represents the role "leaf".
Class Constraints  Class AtomicComponent has the following constraints:

SoftwareComponentRequiresBehavior:
  \[ self\cdot componentType = component :: \text{ComponentKind} :: \text{SOFTWARE\_COMPONENT} \]
  \[ \implies (\text{not } self\cdot behavior\cdot oclIsUndefined()) \]

ValidComponentType:
  \[ self\cdot componentType = component :: \text{ComponentKind} :: \text{SOFTWARE\_COMPONENT} \]
  or \[ self\cdot componentType = component :: \text{ComponentKind} :: \text{CONTINUOUS\_COMPONENT} \]

SoftwareComponentValidPorts:
  \[ self\cdot componentType = component :: \text{ComponentKind} :: \text{SOFTWARE\_COMPONENT} \]
  \[ \implies (self\cdot ports \rightarrow \forall (p \mid p\cdot oclIsTypeOf(component :: \text{DiscretePort}) \text{ or } p\cdot oclIsTypeOf(component :: \text{HybridPort})) \]

ContinuousComponentValidPorts:
  \[ self\cdot componentType = component :: \text{ComponentKind} :: \text{CONTINUOUS\_COMPONENT} \]
  \[ \implies (self\cdot ports \rightarrow \forall (p \mid p\cdot oclIsTypeOf(component :: \text{ContinuousPort})) \]

AtomicComponentsNamesMustBeUnique:
  \[ \text{AtomicComponent\cdot allInstances()}\cdot name\rightarrow count(self\cdot name) = 1 \]

Parent Classes
- Component see Section A.4.2.4 on Page 193,
- BehavioralElement see Section A.6.2.5 on Page 215

A.4.2.3. Class BehavioralConnector

Overview  Abstract super class for all connectors that have an associated behavior. The behavior is specified as a real-time statechart.

Parent Classes
- ConnectorType see Section A.4.2.7 on Page 196,
- BehavioralElement see Section A.6.2.5 on Page 215
A.4.2.4. Class Component

Overview This abstract class is the super class of all classes representing a concrete component type such as a structured, atomic or a continuous component.

Component types are implemented as a variation of the composite design pattern. Concerning the composite pattern this class represents the role "component".

Class Properties Class Component has the following properties:

- **componentType : ComponentKind [0..1]** see Section A.4.2.5 on Page 193
  This attribute specifies the kind of the component. A component may be either discrete software component, a continuous component, a hybrid component or a hardware component.

Class References Class Component has the following references:

- **ports : Port [0..*]** see Section A.4.2.14 on Page 205
  The ports of a component represent the interaction points between the component and its environment.

- **referencingComponentParts : ComponentPart [0..*]** see Section A.4.2.6 on Page 194
  This association contains all component parts which have this component as their type.

Class Constraints Class Component has the following constraints:

- **UniquePortNames:**
  ```python
  self.ports ->isUnique(name)
  ```

Parent Classes

- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267,
- ConstrainableElement see Section A.6.2.7 on Page 216

A.4.2.5. Enumeration ComponentKind

Overview The entries of the enumeration represent different kinds of components. These are discrete software components, continuous components containing controller code, and hybrid components that is a discrete software component which may have continuous input signals.
Enum Properties

Enumeration ComponentKind has the following literals:

SOFTWARE_COMPONENT = 0

A component of this kind represent discrete software components. A discrete software component has a behavior specification which is given by means of a real-time statechart.

CONTINUOUS_COMPONENT = 1

A continuous component represents a continuous controller. Such components do not carry a behavior specification in MechatronicUML. Instead, we assume that the behavior of such components is modeled by using a control engineering tool like Matlab/Simulink, Dymola/Modelica or CamelView. In MechatronicUML, only the interface of these components is modeled. The interface is given by their ports.

HYBRID_COMPONENT = 2

A hybrid component bridges the gap between discrete software components and continuous control components. A hybrid component may be considered as a discrete software component which has special ports for reading and writing continuous signals from and to continuous components, e.g., for setting a new reference value to a controller.

A.4.2.6. Class ComponentPart

Overview

This class represents a component part. Component parts are used to specify the inner structure of a structured component. A component part represents another component that is embedded in a structured component. It is specified on the model level and is always typed over a component (either structured or atomic).

Class Properties

Class ComponentPart has the following properties:

/isMultiPart : EBoolean [0..1]

derivation:

self.cardinality.upperBound.value > 1 or self.cardinality.
upperBound.infinity

This derived attribute indicates if the part is a multi part. It is only used to simplify OCL constraints.

Class References

Class ComponentPart has the following references:

cardinality : Cardinality  see Section A.6.2.6 on Page 216

The cardinality of a ComponentPart specifies how many instances of a ComponentPart are allowed to exist at runtime.
**componentType** : Component  see Section A.4.2.4 on Page 193

The component type typing this component part.

**delegation** : Delegation [0..*]  see Section A.4.2.10 on Page 199

The delegations connecting a port part of this component part with a port of the parent component type.

**fromRev** : Assembly [0..*]  see Section A.4.2.1 on Page 187

The assemblies originating in port parts of this component part.

**parentComponent** : StructuredComponent  see Section A.4.2.16 on Page 207

The structured component type containing this component part.

**/portsDerived** : Port [0..*]  see Section A.4.2.14 on Page 205

\[\text{derivation:} \]
\[
\text{if } \text{componentType}.oclIsUndefined() \text{ then } \text{OrderedSet} [ ] \]
\[
\text{else} \]
\[
\text{componentType}.\text{ports} \]
\[
\text{endif} \]

The ports of this part. They are derived from the ports of the componentType of this component part. It is a containment reference, so that GMF is able to let them flow around the component. Because this feature is derived, transient, volatile the model file will not store the ports in this feature.

**toRev** : Assembly [0..*]  see Section A.4.2.1 on Page 187

The assemblies leading to port parts of this component part.

**Class Constraints**  Class ComponentPart has the following constraints:

**CardinalityLowerBoundSet:**

\[\text{self}.\text{cardinality}.\text{lowerBound} \rightarrow \text{notEmpty}()\]

**TypeNotEqualToParent:**

\[\text{self}.\text{componentType} \neq \text{self}.\text{parentComponent}\]

**CardinalityUpperBoundSet:**

\[\text{self}.\text{cardinality}.\text{upperBound} \rightarrow \text{notEmpty}()\]

**Parent Classes**

- CommentableElement see Section A.13.2.1 on Page 267,
- NamedElement see Section A.13.2.4 on Page 269
A.4.2.7. **Class** ConnectorType

**Overview**  This abstract class is the common super class of delegations and assemblies.

**Class References**  Class ConnectorType has the following references:

1. **/fromContinuousPort : ContinuousPort [0..1]**  see Section A.4.2.8 on Page 197
   
   **derivation:**
   
   if not self.fromPort.oclIsUndefined() and self.fromPort.
   oclIsTypeOf(component::ContinuousPort) then
   self.fromPort.oclAsType(component::ContinuousPort)
   else
   null
   endif

   Return the fromPort as a ContinuousPort if possible

2. **/fromDiscretePort : DiscretePort [0..1]**  see Section A.4.2.11 on Page 202
   
   **derivation:**
   
   if not self.fromPort.oclIsUndefined() and self.fromPort.
   oclIsTypeOf(component::DiscretePort) then
   self.fromPort.oclAsType(component::DiscretePort)
   else
   null
   endif

   Return the fromPort as a DiscretePort if possible

3. **/fromHybridPort : HybridPort [0..1]**  see Section A.4.2.12 on Page 204
   
   **derivation:**
   
   if not self.fromPort.oclIsUndefined() and self.fromPort.
   oclIsTypeOf(component::HybridPort) then
   self.fromPort.oclAsType(component::HybridPort)
   else
   null
   endif

   Return the fromPort as a HybridPort if possible

4. **fromPort : Port**  see Section A.4.2.14 on Page 205
   
   The port this connector originates from.

5. **/toContinuousPort : ContinuousPort [0..1]**  see Section A.4.2.8 on Page 197
   
   The structured component this connector belongs to.

6. **parentComponent : StructuredComponent [0..1]**  see Section A.4.2.16 on Page 207
   
   The structured component this connector belongs to.
derivation:

if not self.toPort.oclIsUndefined() and self.toPortoclIsTypeOf(component::ContinuousPort) then
    self.toPort.oclAsType(component::ContinuousPort)
elsereturn nullendif

Return the toPort as a ContinuesPort if possible

/toDiscretePort : DiscretePort [0..1] see Section A.4.2.11 on Page 202
derivation:

if not self.toPort.oclIsUndefined() and self.toPortoclIsTypeOf(component::DiscretePort) then
    self.toPort.oclAsType(component::DiscretePort)
elsereturn nullendif

Return the toPort as a DiscretePort if possible

/toHybridPort : HybridPort [0..1] see Section A.4.2.12 on Page 204
derivation:

if not self.toPort.oclIsUndefined() and self.toPortoclIsTypeOf(component::HybridPort) then
    self.toPort.oclAsType(component::HybridPort)
elsereturn nullendif

Return the toPort as a HybridPort if possible

toPort : Port see Section A.4.2.14 on Page 205

The port this connector leads to.

Parent Classes

- ExtendableElement see Section A.13.2.2 on Page 267

A.4.2.8. Class ContinuousPort

Overview This class represents a concrete port specification which provides the continuous functionality of a port. A continuous port emits a signal value. A signal value has a data type and it has concrete values at all points in time.
Class Properties  Class ContinuousPort has the following properties:

\texttt{/isContinuousInPort : EBoolean [0..1]}

\texttt{derivation:}

\texttt{self.kind = component::ContinuousPortDirectionKind::IN}

This derived attribute indicates if the continuous port is an IN port

\texttt{/isContinuousOutPort : EBoolean [0..1]}

\texttt{derivation:}

\texttt{self.kind = component::ContinuousPortDirectionKind::OUT}

This derived attribute indicates if the continuous port is an OUT port

\texttt{kind : ContinuousPortDirectionKind  see Section A.4.2.9 on Page 199}

Decides the direction of a continuous port.

Class References  Class ContinuousPort has the following references:

\texttt{type : DataType  see Section A.6.2.8 on Page 216}

Defines the data type of the signal value which is emitted or received by

Class Constraints  Class ContinuousPort has the following constraints:

\texttt{LowerBoundMustBeZeroOrOne:}

\texttt{-- This Constraint is fulfilled, if no Cardinality exists.}
\texttt{-- But that is okay, as then another Problem-Marker is shown.}
\texttt{-- because Cardinality.lowerBound is 1..1}
\texttt{self.cardinality.oclIsUndefined() or (}
\texttt{  if self.cardinality.lowerBound.oclIsUndefined() then}
\texttt{    false}
\texttt{  else}
\texttt{    self.cardinality.lowerBound.value = 0 or self.cardinality.}
\texttt{    lowerBound.value = 1}
\texttt{  endif}
\texttt{)}

\texttt{UpperBoundMustBeOne:}

\texttt{-- This Constraint is fulfilled, if no Cardinality exists.}
\texttt{-- But that is okay, as then another Problem-Marker is shown.}
\texttt{-- because Cardinality.upperBound is 1..1}
\texttt{self.cardinality.oclIsUndefined() or (}
\texttt{  if self.cardinality.upperBound.oclIsUndefined() then}
\texttt{    false}
\texttt{  else}
\texttt{    self.cardinality.upperBound.value = 1}
\texttt{  endif}
\texttt{)}
A.4.2.9. Enumeration ContinuousPortDirectionKind

**Overview**  Decides the direction of a continous port.

**Enum Properties**  Enumeration ContinuousPortDirectionKind has the following literals:

- **IN = 0**
  - Represent an IN-Port of a continous port.

- **OUT = 1**
  - Represent an OUT-Port of a continous port.

A.4.2.10. Class Delegation

**Overview**  This class represents a delegation connector. A delegation connector connects a port of a structured component type and a port part of component part the structured component contains. The delegation has no behavior. In a running system, the port of the structured component and the port of the component part will be the same object like interfaces of classes where interface and class are the same object at runtime.

**Class References**  Class Delegation has the following references:

- **componentPart : ComponentPart**  see Section A.4.2.6 on Page 194
  - The component part of the port part which is connected by this delegation.

**Class Constraints**  Class Delegation has the following constraints:

- **ValidContinuousPortDirections:**
  ```java
  not self.fromContinuousPortoclIsUndefined() and not self.toContinuousPortoclIsUndefined()
  implies
  self.fromContinuousPort.kind = self.toContinuousPort.kind
  ```

- **DelegationBetweenContinuousPortsRequiresSameDataType:**
  ```java
  not self.fromContinuousPortoclIsUndefined() and not self.toContinuousPortoclIsUndefined()
  implies
  self.fromContinuousPort.type = self.toContinuousPort.type
  ```
DelegationBetweenDiscretePortsRequiresSameCoordinationPattern:

if not self.fromDiscretePort.oclIsUndefined() and not self.toDiscretePort.oclIsUndefined() then
not self.fromDiscretePort.refines.oclIsUndefined() and not self.toDiscretePort.refines.oclIsUndefined() and
— both refinements must belong to the same pattern
self.fromDiscretePort.refines.coordinationPattern = self.toDiscretePort.refines.coordinationPattern
else
true
endif

DelegationBetweenDiscretePortsRequiresSameRoles:

if not self.fromDiscretePort.oclIsUndefined() and not self.toDiscretePort.oclIsUndefined() then
not self.fromDiscretePort.refines.oclIsUndefined() and not self.toDiscretePort.refines.oclIsUndefined() and
— both ports should have the same roles
self.fromDiscretePort.refines.name = self.toDiscretePort.refines.name
else
true
endif

DiscreteMultiPortDelegationRequiresMultiPortOrSinglePortAndMultiPart:

not self.fromDiscretePort.oclIsUndefined() and not self.toDiscretePort.oclIsUndefined() and self.fromPort.isMultiPort
implies (  
— the target port is a multi port
self.toPort.isMultiPort
or
— the target part is a multi part
self.componentPart.isMultiPart
)

ValidDiscreteInPortCombination:

not self.fromDiscretePort.oclIsUndefined() and self.fromDiscretePort.isDiscreteInPort
implies (  
not self.toDiscretePort.oclIsUndefined() and self.toDiscretePort.isDiscreteInPort
)

ValidDiscreteOutPortCombination:
not self.fromDiscretePort.oclIsUndefined() and self.
fromDiscretePort.isDiscreteOutPort
implies (not self.toDiscretePort.oclIsUndefined() and self.
toDiscretePort.isDiscreteOutPort)

ValidDiscreteInOutPortCombination:
not self.fromDiscretePort.oclIsUndefined() and self.
fromDiscretePort.isDiscreteInOutPort
implies (not self.toDiscretePort.oclIsUndefined() and self.
toDiscretePort.isDiscreteInOutPort)

ValidContinuousInPortCombination:
not self.fromContinuousPort.oclIsUndefined() and self.
fromContinuousPort.isContinuousInPort
implies (not self.toContinuousPort.oclIsUndefined() and self.
toContinuousPort.isContinuousInPort)

ValidContinuousOutPortCombination:
not self.fromContinuousPort.oclIsUndefined() and self.
fromContinuousPort.isContinuousOutPort
implies (not self.toContinuousPort.oclIsUndefined() and self.
toContinuousPort.isContinuousOutPort
or
(not self.toHybridPort.oclIsUndefined() and self.
toHybridPort.isHybridOutPort)

ValidHybridInPortCombination:
not self.fromHybridPort.oclIsUndefined() and self.fromHybridPort.
isHybridInPort
implies (not self.toContinuousPort.oclIsUndefined() and self.
toContinuousPort.isContinuousInPort)

ValidHybridOutPortCombination:
Parent Classes

- ConnectorType see Section A.4.2.7 on Page 196

A.4.2.11. Class DiscretePort

Overview This class represents a concrete port specification which provides the discrete functionality of a port.

Class Properties Class DiscretePort has the following properties:

/\isDiscreteInoutPort : EBoolean [0..1]

derivation:
\[
\text{not self.receiverMessageInterface.ocIsUndefined() and not self.senderMessageInterface.ocIsUndefined()}
\]

This derived attribute indicates if the discrete port is an IN OUT port

/\isDiscreteInPort : EBoolean [0..1]

derivation:
\[
\text{not self.receiverMessageInterface.ocIsUndefined() and self.senderMessageInterface.ocIsUndefined()}
\]

This derived attribute indicates if the discrete port is an IN port

/\isDiscreteOutPort : EBoolean [0..1]

derivation:
\[
\text{not self.senderMessageInterface.ocIsUndefined() and self.receiverMessageInterface.ocIsUndefined()}
\]

This derived attribute indicates if the discrete port is an OUT port

Class References Class DiscretePort has the following references:

adaptationBehavior : Behavior [0..1] see Section A.6.2.4 on Page 215

If this port is a multi-port, this reference points to the real-time statechart that contains the adaptation behavior of the multi-port. Then, this real-time statechart is contained in the only state of the real-time statechart we is obtained by the reference roleAndAdaptationBehavior. If this port is a single-port, this reference will be undefined.
receiverMessageInterface : MessageInterface [0..1] see Section A.9.2.1 on Page 234

The receiver message interface defines which messages this discrete port specification receives.

refines : Role [0..1] see Section A.10.2.2 on Page 238

The role of a coordination pattern that this port refines.

roleAndAdaptationBehavior : Behavior [0..1] see Section A.6.2.4 on Page 215

If this port is a multi-port, this reference points to the real-time statechart that contains the adaptation behavior and the sub-port behavior. Thus, this real-time statechart only contains one state which embeds the real-time statecharts specifying the adaptation behavior and the sub-port behavior.

senderMessageInterface : MessageInterface [0..1] see Section A.9.2.1 on Page 234

The sender message interface defines which messages this discrete port specification sends.

Class Constraints  Class DiscretePort has the following constraints:

AtLeastOneMessageInterface:

\[
\text{self.oclIsTypeOf(component::DiscretePort)} \implies (\neg (\text{self.}\\text{senderMessageInterface.oclIsUndefined()} \\text{and} \text{self.}\\text{receiverMessageInterface.oclIsUndefined()}))
\]

DiscretePortRequiresBehavior:

\[\text{--- this also holds for a hybrid port, ports of structured components do not require a behavior}
(\neg \text{self.component.oclIsUndefined()} \\text{and} \text{self.component.oclIsTypeOf(component::AtomicComponent)}) \implies \neg \text{self.behavior.oclIsUndefined()}
\]

DiscretePortAtStructuredComponentHasNoBehavior:

\[\neg \text{self.component.oclIsUndefined()} \\text{and} \text{self.component.oclIsTypeOf(component::StructuredComponent)} \implies \text{self.behavior.oclIsUndefined()}
\]

DiscretePortRequiresRole:

\[
\text{self.oclIsTypeOf(component::DiscretePort)} \implies \neg \text{self.}\\text{refines.oclIsUndefined()}
\]

DiscretePortAndRoleSameMessageInterface:
not self.refines.oclIsUndefined() implies
(self.senderMessageInterface = self.refines.
  senderMessageInterface
and
self.receiverMessageInterface = self.refines.
  receiverMessageInterface
)

MultiPortMustRefineMultiRole:
if not (self.refines.oclIsUndefined() and self.refines.
  cardinality.oclIsUndefined() and self.refines.cardinality.
  upperBound.oclIsUndefined()) then
  self.isMultiPort implies self.refines.isMultiRole
else
  true
endif

Parent Classes

- Port see Section A.4.2.14 on Page 205,
- BehavioralElement see Section A.6.2.5 on Page 215

A.4.2.12. Class HybridPort

Overview  This class represents a hybrid port which acts as a bridge between continuous
collectors and discrete software. A hybrid port emits or receives a signal value which has a
data type and a concrete value at all points in time. Then, the hybrid port discretizes the signal
value in given time intervals and provides the value as variable to its Real-Time Statechart.
The hybrid port does not define message interfaces.

Class Properties  Class HybridPort has the following properties:

/ isHybridInPort : EBoolean [0..1]
  derivation:
    — derive from superclass ContinuousPort
    self.isContinuousInPort

This derived attribute indicates if the hybrid port is an IN port

/ isHybridOutPort : EBoolean [0..1]
  derivation:
    — derive from superclass ContinuousPort
    self.isContinuousOutPort
This derived attribute indicates if the hybrid port is an OUT port

**Class References** Class `HybridPort` has the following references:

- `samplingInterval : NaturalNumber` see Section A.6.2.9 on Page 217
  The sampling interval defines the time between two updates of the continuous signal which is received or send by this hybrid port. If the port is an IN-port, the sampling interval defines how often the continuous signal is read and stored internally. If the hybrid port in an OUT-port, the sampling interval defines how often a new value is send via this port.

**Parent Classes**
- `DiscretePort` see Section A.4.2.11 on Page 202,
- `ContinuousPort` see Section A.4.2.8 on Page 197

### A.4.2.13. Class PatternOccurrence

**Overview** A coordination protocol (pattern) can occur within a structured component. It defines the communication behavior of an assembly connection that connects discrete ports with each other.

**Class References** Class `PatternOccurrence` has the following references:

- `pattern : CoordinationPattern` see Section A.10.2.1 on Page 237
  The coordination protocol (pattern) of this PatternOccurrence.
- `ports : Port [1..*]` see Section A.4.2.14 on Page 205
  The ports that take part in this protocol.

**Parent Classes**
- `ExtendableElement` see Section A.13.2.2 on Page 267

### A.4.2.14. Class Port

**Overview** Ports represent the interaction points between a component and the components environment.

**Class Properties** Class `Port` has the following properties:

- `/isMultiPort : EBoolean [0..1]`
  derivation:
if not (self.cardinality.oclIsUndefined()) then
  (self.cardinality.upperBound.value > 1) or self.cardinality.upperBound.infinity
else
  false
endif

This derived attribute indicates if the port is a multi port.

Class References  Class Port has the following references:

  cardinality : Cardinality  see Section A.6.2.6 on Page 216
  
  The cardinality of a port specifies how many instances of a port are allowed to
  exist at runtime.

  component : Component [0..1]  see Section A.4.2.4 on Page 193
  
  The component, this port belongs to. Theoretically the bounds should be 1..1, but
  that would prevent the possibility for ComponentPart.portsDerived to be a contain-
  ment reference (see ComponentPart.portsDerived)

  /connectors : ConnectorType [0..*]  see Section A.4.2.7 on Page 196
    derivation:
      self.incomingConnectors -> union(self.outgoingConnectors)

  incomingConnectors : ConnectorType [0..*]  see Section A.4.2.7 on Page 196
  
  The connectors which lead into this port.

  outgoingConnectors : ConnectorType [0..*]  see Section A.4.2.7 on Page 196
  
  The connectors which originate from this port.

Parent Classes

  • NamedElement see Section A.13.2.4 on Page 269,

  • CommentableElement see Section A.13.2.1 on Page 267,

  • ConstrainableElement see Section A.6.2.7 on Page 216

A.4.2.15. Class StaticStructuredComponent

Overview  A static structured component is a structured component whose internal structure
consisting of component part, ports, delegation, and assemblies cannot be changed during run-


A.4. Parent Classes  

- StructuredComponent see Section A.4.2.16 on Page 207

A.4.2.16. Class StructuredComponent

Overview  This class represents a structured component which is capable of including arbitrarily many component parts.

Class References  Class StructuredComponent has the following references:

/allAtomicComponents : AtomicComponent [0..∗] see Section A.4.2.2 on Page 191

derivation:

    self.allStructuredComponents->collect(
        embeddedParts->select(
            componentTypeoclIsTypeOf(component::AtomicComponent)
        )->collect(componentType.oclAsType(component::AtomicComponent))
    )->asOrderedSet()

Transitive closure of all AtomicComponents which are used (specified as a componentType for a ComponentPart) in the component hierarchy. Note: AtomicComponents from directly embeddedParts are NOT included

/allStructuredComponents : StructuredComponent [0..∗] see Section A.4.2.16 on Page 207

derivation:

    self->closure(
        embeddedParts->select(
            componentTypeoclIsTypeOf(component::StructuredComponent)
        )->collect(componentType.oclAsType(component::StructuredComponent))
    )

Transitive closure of all StructuredComponents which are used (specified as a componentType for a ComponentPart) in the component hierarchy.

collectors : ConnectorType [0..∗] see Section A.4.2.7 on Page 196

The connectors this structured component contains. These can either be delegations or assemblies.

embeddedParts : ComponentPart [1..∗] see Section A.4.2.6 on Page 194

The component parts this structured component contains.
Class Constraints  Class StructuredComponent has the following constraints:

**StructuredComponentNoHybridPort:**
```
self.ports ->forall ( p | not poclIsOfType(component::HybridPort))
```

**ValidComponentType:**
```
self.componentType = component::ComponentKind::SOFTWARE_COMPONENT
or self.componentType = component::ComponentKind::HYBRID_COMPONENT
```

**NoCyclicComponentPartHierarchy:**
```
not self.allStructuredComponents ->includes ( self)
```

**DiscreteStructuredComponentValidParts:**
```
sel.componentType = component::ComponentKind::SOFTWARE_COMPONENT implies
  collect all atomic components from parent parts and union them
  with own atomic components
self.allAtomicComponents ->union (self.embeddedParts ->select (componentoclIsOfType(component::AtomicComponent)) ->collect(componentoclAsType(component::AtomicComponent)) ->asOrderedSet () ) ->forall (componentType = component::ComponentKind::SOFTWARE_COMPONENT)
```

**HybridStructuredComponentValidParts:**
```
sel.componentType = component::ComponentKind::HYBRID_COMPONENT implies
  collect all atomic components from parent parts and union them
  with own atomic components
self.allAtomicComponents ->union (self.embeddedParts ->select (componentoclIsOfType(component::AtomicComponent)) ->collect(componentoclAsType(component::AtomicComponent)) ->asOrderedSet () ) ->exists (componentType = component::ComponentKind::CONTINUOUS_COMPONENT)
```

**DiscreteStructuredComponentValidPorts:**
```
sel.componentType = component::ComponentKind::SOFTWARE_COMPONENT implies (sel.ports ->forall ( p | poclIsOfType(component::DiscretePort))
)
HybridStructuredComponentValidPorts:

\[
\text{self.componentType} = \text{component::ComponentKind::HYBRID_COMPONENT}
\implies
\left( \text{self.ports} \rightarrow \forall p \mid p.oclIsTypeOf(\text{component::DiscretePort}) \text{ or } p.oclIsTypeOf(\text{component::ContinuousPort}) \right)
\]

ComponentPartsHaveUniqueName:

\[
\text{self.embeddedParts} \rightarrow \text{isUnique(name)}
\]

Parent Classes

- Component see Section A.4.2.4 on Page 193
A.5. Package `muml::model::constraint`

A.5.1. Package Overview

The package constraint provides abstract super classes for modeling different kinds of constraints that may be attached to ConstrainableElements of the MechatronicUML meta-model.

![Meta-Model of the constraint Package](image)

Figure A.3.: Meta-Model of the constraint Package

A.5.2. Detailed Contents Documentation

A.5.2.1. Class Constraint

**Overview**  This class represents a constraint. A constraint defines certain properties a system has to fulfill. In terms of model checking a constraint represents the specification of the system.

**Class Properties**  Class Constraint has the following properties:

- **background : EBoolean [0..1]**
  
  This attribute decides whether background checking is activated for this constraint. If it is activated the correctness of the constraint is checked whenever the model changes. These checks are performed in the background such that user interaction is not interrupted.

- **correctness : Correctness [0..1]**  see Section A.5.2.2 on Page 211
  
  The correctness of this constraint encoded as a literal of the enum type "Correctness".
A.5. PACKAGE `Muml::Model::Constraint`

Class References Class `Constraint` has the following references:

- `constrainableElement : ConstrainableElement` see Section A.6.2.7 on Page 216
  The element this constraint applies to.

Parent Classes
- ExtendableElement see Section A.13.2.2 on Page 267

A.5.2.2. Enumeration `Correctness`

**Overview** This enumeration encodes the correctness result of a constraint. The correctness is UNKNOWN if the constraint has not yet been verified or if the verification failed for some reason. The constraint is CORRECT, if the verification returned true. Otherwise the constraint is VIOLATED.

**Enum Properties** Enumeration `Correctness` has the following literals:

- `UNKNOWN = 0`
- `CORRECT = 1`
- `VIOLATED = 2`

A.5.2.3. Class `ModelingConstraint`

**Overview** A modeling constraint is a static semantics constraint that restricts the model. It can be checked statically and will not be used for verification.

Parent Classes
- Constraint see Section A.5.2.1 on Page 210

A.5.2.4. Class `TextualConstraint`

**Overview** This class represents all verifiable constraints that can be entered as a string in a predefined constraint language like, e.g., CTL or TCTL. Therefore, it contains a textual expression which is used to store the constraint text and the language.

**Class References** Class `TextualConstraint` has the following references:

- `textualExpression : TextualExpression [0..1]` see Section A.14.2.2 on Page 270
  A textual expression which stores the constraint text and the language in which the constraint is specified.
Parent Classes

- VerifiableConstraint see Section A.5.2.5 on Page 212,
- ExtendableElement see Section A.13.2.2 on Page 267

A.5.2.5. Class VerifiableConstraint

Overview A verifiable constraint is a dynamic semantics constraint that will be used for verification of the model. This class serves as a super class for all types of verifiable constraints.

Parent Classes

- Constraint see Section A.5.2.1 on Page 210
A.6. Package `muml::model::core`

A.6.1. Package Overview

This package contains several core classes that are used by classes from several other packages. It provides abstract base classes for Statecharts, meta-model elements that use a state-chart to define their behavior and meta-model elements that may carry a constraint. Additionally, the package provides classes for modeling cardinalities as natural numbers including infinity.

A.6.2. Detailed Contents Documentation

A.6.2.1. Class `ActivityCallExpression`

Overview  An Expression that represents an activity.

Class References  Class `ActivityCallExpression` has the following references:

- `activity : Activity`  see Section A.17.2.1 on Page 278

  Specifies the exception variable that this expression refers to. If you have an activity edge that catches an exception e, then this expression can represent the reference e.

Parent Classes

- Invocation see Section A.19.2.2 on Page 289,
- Expression see Section A.14.2.1 on Page 270

A.6.2.2. Class `ArrayDataType`

Overview  This data type represents an array data type and specifies the maximum cardinality of inner data types.

Class References  Class `ArrayDataType` has the following references:

- `cardinality : NaturalNumber`  see Section A.6.2.9 on Page 217
- `type : DataType`  see Section A.6.2.8 on Page 216

  This reference points to the definition of the data type.

Parent Classes

- DataType see Section A.6.2.8 on Page 216
Figure A.4.: Meta-Model of the core Package
A.6.2.3. **Class Attribute**

**Overview** Implementation of an Attribute of a behavior which has a certain type.

**Class References** Class `Attribute` has the following references:

- `type : DataType` see Section A.6.2.8 on Page 216
  The type of this attribute.

**Parent Classes**
- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267

A.6.2.4. **Class Behavior**

**Overview** Abstract super class for all elements that represent a behavior. Known subclasses: AbstractRealtimeStatechart

**Class References** Class `Behavior` has the following references:

- `attributes : Attribute [0..*]` see Section A.6.2.3 on Page 215
  A behavior may define a set of Attributes in order to store data. The attributes may be used by the operations of the behavior and the behavior specification itself. The attributes are contained in the behavior.
- `behavioralElement : BehavioralElement [0..1]` see Section A.6.2.5 on Page 215
  The behavioral element this statechart belongs to.
- `operations : Operation [0..*]` see Section A.6.2.10 on Page 217
  A behavior may define a set of Operations as signatures of helper functions. These operations may be called by the behavior specification and may access the attributes of the behavior specification. The Operations are contained in the behavior.

A.6.2.5. **Class BehavioralElement**

**Overview** Abstract super class for all elements that have a behavior.

**Class References** Class `BehavioralElement` has the following references:

- `behavior : Behavior [0..1]` see Section A.6.2.4 on Page 215
  The real-time statechart that defines the behavior of this behavioral element.
A.6.2.6. **Class Cardinality**

**Overview**  This class represents the cardinality of an arbitrary model object. It consists of a lower and an upper bound.

**Class References**  Class `Cardinality` has the following references:

- `lowerBound : NaturalNumber`  see Section A.6.2.9 on Page 217
  
  The lower bound of this cardinality.

- `upperBound : NaturalNumber`  see Section A.6.2.9 on Page 217
  
  The upper bound of this cardinality.

**Class Constraints**  Class `Cardinality` has the following constraints:

- `LowerBoundMustBeLessOrEqualThanUpperBound`:

  ```
  ( (not self.lowerBound.infinity and not self.upperBound.infinity) 
  implies (self.lowerBound.value <= self.upperBound.value) ) 
  and (self.lowerBound.infinity implies self.upperBound.infinity) 
  ```

A.6.2.7. **Class ConstrainableElement**

**Overview**  Abstract super class for all model elements that may carry a constraint.

**Class References**  Class `ConstrainableElement` has the following references:

- `constraint : Constraint [0..*]`  see Section A.5.2.1 on Page 210
  
  The constraint for this element.

A.6.2.8. **Class DataType**

**Overview**  Abstract super class for all types that may be used for attributes, parameters, and operations.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269,

- CommentableElement see Section A.13.2.1 on Page 267
A.6.2.9. **Class NaturalNumber**

**Overview**  This class represents either a natural number or infinity.

**Class Properties**  Class NaturalNumber has the following properties:

- **infinity** : EBoolean [0..1]
  
  Determines whether this natural number represents infinity.

- **value** : ELong [0..1]
  
  The value of this natural number.

**Class Constraints**  Class NaturalNumber has the following constraints:

- **ValueGreaterOrEqualZero**:

  ```
  self.value >= 0
  ```

A.6.2.10. **Class Operation**

**Overview**  An operation specifies a behavior that can be called with a list of concrete parameters and may return a return value.

**Class References**  Class Operation has the following references:

- **implementations** : Expression [0..*]  see Section A.14.2.1 on Page 270
  
  The implementation for this operation. MechatronicUML supports the annotation of multiple implementations for an operation to support different target languages.

- **parameters** : Parameter [0..*]  see Section A.6.2.11 on Page 218

- **returnType** : DataType  see Section A.6.2.8 on Page 216

  The type of the return value of this operation.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269,

- CommentableElement see Section A.13.2.1 on Page 267
A.6.2.11. Class Parameter

Overview  This is a general representation of a Parameter which is by all model elements that receive parameters. Examples include operations, message types, and synchronization channels. A parameter defines a data type.

Class References  Class Parameter has the following references:
  
  type : DataType  see Section A.6.2.8 on Page 216
  
  The type of this parameter.

Parent Classes
  
  • NamedElement see Section A.13.2.4 on Page 269,
  
  • CommentableElement see Section A.13.2.1 on Page 267

A.6.2.12. Class ParameterBinding

Overview  A parameter binding associates a parameter with a concrete value which is bound to this parameter by an invocation. As an example, an operation defines a set of parameters. A call of this operation needs to provide concrete values for the parameters which are defined by a parameter binding. The value is represented by an expression.

Class References  Class ParameterBinding has the following references:
  
  parameter : Parameter  see Section A.6.2.11 on Page 218
  
  The parameter to which the value needs to be associated.

  value : Expression  see Section A.14.2.1 on Page 270
  
  The value which is associated with the parameter. The value is defined by an expression.

Parent Classes
  
  • ExtendableElement see Section A.13.2.2 on Page 267

A.6.2.13. Class PrimitiveDataType

Overview  This data type represents a primitive data type and refers to the PrimitiveDataType enumeration for specifying the concrete primitive type.

Class Properties  Class PrimitiveDataType has the following properties:
  
  primitiveType : PrimitiveTypes  see Section A.6.2.14 on Page 219
  
  Refers to the primitive data type as defined by the PrimitiveDataType enumeration. It defines the actual type.
A.6. PACKAGE `MUML::MODEL::CORE`

Parent Classes

- DataType see Section A.6.2.8 on Page 216

A.6.2.14. Enumeration `PrimitiveTypes`

Overview  Defines all primitive types that may be used in MechatronicUML.

Enum Properties  Enumeration `PrimitiveTypes` has the following literals:

- VOID = 0
- BOOLEAN = 1
- BYTE = 2
- SHORT = 3
- INT = 4
- LONG = 5
- DOUBLE = 6
- STRING = 7
A.7. **Package muml::model::deployment**

A.7.1. **Package Overview**

![Diagram of deployment package](image)

Figure A.5.: Meta-Model of the deployment Package

A.7.2. **Detailed Contents Documentation**

A.7.2.1. **Class CommunicationLink**

**Overview**  A communication link between hardware nodes and a target used for the deployment of connectors between component instances.

**Class References**  Class CommunicationLink has the following references:

- **deployedAssemblyInstances**: AssemblyInstance [0..*] see Section A.8.2.1 on Page 225
  Used for the deployment of connectors between component instances.
- **deployment**: Deployment see Section A.7.2.2 on Page 221
  The deployment to which the link belongs to.
- **qualityOfService**: QualityOfLinkService [0..1] see Section A.7.2.7 on Page 224
  The quality of service which the link fulfills.
- **source**: HardwarePort see Section A.7.2.5 on Page 223
  Source HardwarePort of the link.
target : HardwarePort  see Section A.7.2.5 on Page 223

Target HardwarePort of the link.

**Class Constraints**  Class CommunicationLink has the following constraints:

**SameConfiguration:**

\[
\text{self.deployedAssemblyInstances.componentInstanceConfiguration} = \\
\text{self.deployment.componentInstanceConfiguration}
\]

**Parent Classes**

- ExtendableElement see Section A.13.2.2 on Page 267

### A.7.2.2. Class Deployment

**Overview**  Deployment exists in the software lifecycle to bridge the gap between what a software developer could know about the execution environment and what the environment’s developer could know about the deployable software.\(^1\)

**Class References**  Class Deployment has the following references:

- **communicationLinks : CommunicationLink [0..\*]** see Section A.7.2.1 on Page 220

A Deployment contains CommunicationLinks which are used to connect HardwarePorts.

- **componentInstanceConfiguration : ComponentInstanceConfiguration [0..1]** see Section A.8.2.4 on Page 227

The componentInstanceConfiguration which contains the component instances which should be deployed to hardware nodes.

- **hardwareNodes : HardwareNode [1..\*]** see Section A.7.2.3 on Page 222

A Deployment contains HardwareNodes which represents hardware resources like computational, communication, sensor, or actor.

- **qualityOfLinkServices : QualityOfLinkService [0..1]** see Section A.7.2.7 on Page 224

Reference to all contained quality of services.

\(^1\)C. Szyperski, Foreword to Proceedings of Component Deployment, IFIP/ACM Working Conference, Berlin 2002
A.7.2.3. Class HardwareNode

Overview A run-time computational resource which generally has at least memory and processing capabilities. Component instances may reside on hardware nodes if they are not a sensor or an actor.

Class Properties Class HardwareNode has the following properties:

- `hardwareNodeKind : HardwareNodeKind [0..1]` see Section A.7.2.4 on Page 223
  A HardwareNode is kind of a computational resource, a sensor, an actor, or a communication device like a router.

Class References Class HardwareNode has the following references:

- `deployedInstances : ComponentInstance [0..*]` see Section A.8.2.3 on Page 227
  ComponentInstances which are deployed to the HardwareNode.

- `deployment : Deployment` see Section A.7.2.2 on Page 221
  Deployment which contains the HardwareNode.

- `hardwarePorts : HardwarePort [0..*]` see Section A.7.2.5 on Page 223
  A HardwareNode contains HardwarePorts to communicate with other HardwarePorts via a CommunicationLink.

Class Constraints Class HardwareNode has the following constraints:

- `NoDeploymentOnActorOrSensor:
  (self.hardwareNodeKind <> deployment::HardwareNodeKind::COMPUTATIONALRESOURCE) implies self.deployedInstances -> isEmpty()`
A.7.2.4. Enumeration HardwareNodeKind

**Overview**  Different kinds of HardwareNodes.

**Enum Properties**  Enumeration HardwareNodeKind has the following literals:

- COMPUTATIONAL_RESOURCE = 0
- SENSOR = 2
- ACTOR = 1
- COMMUNICATION = 3

A.7.2.5. Class HardwarePort

**Overview**  Hardware ports are used to communicate with other hardware nodes and with component instances.

**Class Properties**  Class HardwarePort has the following properties:

- **kind :** HardwarePortDirectionKind [0..1]  see Section A.7.2.6 on Page 224
  A HardwarePort is of kind in, out, or in/out. Depending on its kind it can receive, send, or receive and send messages.

**Class References**  Class HardwarePort has the following references:

- deployedPortInstance : PortInstance [0..*]  see Section A.8.2.13 on Page 232
  PortInstances which are deployed to the HardwarePort.

- **hardwareNode :** HardwareNode  see Section A.7.2.3 on Page 222
  Hardware nodes are hardware elements on which port instances could be deployed if they are not a sensor or an actor.

- incomingCommunicationLink : CommunicationLink [0..*]  see Section A.7.2.1 on Page 220
  CommunicationLinks which has the HardwarePort as sink.

- outgoingCommunicationLink : CommunicationLink [0..*]  see Section A.7.2.1 on Page 220
  CommunicationLinks which has the HardwarePort as source.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267
A.7.2.6. **Enumeration HardwarePortDirectionKind**

**Overview**  A HardwarePort is of kind in, out, or in/out. Depending on its kind it can receive, send, or receive and send messages.

**Enum Properties**  Enumeration HardwarePortDirectionKind has the following literals:

- \text{IN} = 0
- \text{OUT} = 1
- \text{IN\_OUT} = 2

A.7.2.7. **Class QualityOfLinkService**

**Overview**  Quality of Service of a communication link.

**Class References**  Class QualityOfLinkService has the following references:

- deployment : Deployment [0..1]  see Section A.7.2.2 on Page 221 
  The deployment to which the service belongs.

- latency : NaturalNumber [0..1]  see Section A.6.2.9 on Page 217 
  The time from the source port sending a message to the destination port receiving it.

- packetDelayVariation : NaturalNumber [0..1]  see Section A.6.2.9 on Page 217 
  Difference in end-to-end one-way delay between selected messages without lost messages.

**Parent Classes**

- CommentableElement see Section A.13.2.1 on Page 267,
- NamedElement see Section A.13.2.4 on Page 269
A.8. Package `muml::model::instance`

A.8.1. Package Overview

The package instance contains all classes for building configurations of component instances. Component instances are built from component types and connected by connectors. The resulting structure is a component instance configuration.

A.8.2. Detailed Contents Documentation

A.8.2.1. Class `AssemblyInstance`

Overview  This class represents an assembly connector at instance level.

Class Properties  Class `AssemblyInstance` has the following properties:

- `propagationDelayLowerBound : EInt [0..1]`
  The lower bound of the propagation delay of this assembly instance. The propagation delay defines how long a message needs from its sender to its receiver port instance.

- `propagationDelayUpperBound : EInt [0..1]`
  The upper bound of the propagation delay of this assembly instance. The propagation delay defines how long a message needs from its sender to its receiver port instance.

Class References  Class `AssemblyInstance` has the following references:

- `/assemblyType : Assembly [0..1]`  see Section A.4.2.1 on Page 187

  derivation:
  
  ```oclAsType(component::Assembly)```

  The assembly that this assembly instance is built from.

Parent Classes

- ConnectorInstance see Section A.8.5 on Page 228

A.8.2.2. Class `AtomicComponentInstance`

Overview  An atomic component instance is a component instance which has been derived from an atomic component. An atomic component instance has no embedded component instance configuration and executes the behavior specification which has been defined by its type.
Figure A.6.: Meta-Model of the instance Package
A.8.3. Class ComponentInstance

Overview  This class represents a component instance. It is an instantiation of a component.

Class References  Class ComponentInstance has the following references:

- componentPart : ComponentPart [0..1]  see Section A.4.2.6 on Page 194
- componentType : Component  see Section A.4.2.4 on Page 193
  The component type of which this instance is derived.
- portInstances : PortInstance [0..*]  see Section A.8.2.13 on Page 232
  The port instances that belong to this component instance.

Parent Classes

- NamedElement see Section A.13.2.4 on Page 269

A.8.4. Class ComponentInstanceConfiguration

Overview  This class encapsulates represents a configuration. It contains all component instances and connector instances that belong to a concrete configuration.

Class References  Class ComponentInstanceConfiguration has the following references:

- componentInstances : ComponentInstance [0..*]  see Section A.8.2.3 on Page 227
  The set of component instances of a component instance configuration.
- connectorInstances : ConnectorInstance [0..*]  see Section A.8.2.5 on Page 228
  The set of connector instances of a component instance configuration.
- /parentPortInstances : PortInstance [0..*]  see Section A.8.2.13 on Page 232
  derivation:
if (self.eContainer().oclIsKindOf(ComponentInstance)) then
  self.eContainer().oclAsType(ComponentInstance).portInstances
else
  OrderedSet {}
endif

The port instances of the containing component instance.

Parent Classes

- CommentableElement see Section A.13.2.1 on Page 267,
- NamedElement see Section A.13.2.4 on Page 269

A.8.2.5. Class ConnectorInstance

Overview This class is the common super class of delegation instances and assembly instances.

Class References Class ConnectorInstance has the following references:

  connectorType : ConnectorType [0..1] see Section A.4.2.7 on Page 196
  The connector type of this connector instance.

  source : PortInstance see Section A.8.2.13 on Page 232
  The port instance this connector instance originates from.

  target : PortInstance see Section A.8.2.13 on Page 232
  The port instance this connector instance leads to.

Parent Classes

- ExtendableElement see Section A.13.2.2 on Page 267

A.8.2.6. Class ContinuousPortInstance

Overview This class represents a continuous port at instance level. The port type of a continuous port instance must be a continuous port.

Parent Classes

- PortInstance see Section A.8.2.13 on Page 232
A.8.2.7. Class DelegationInstance

Overview  This class represents a delegation connector at instance level.

Class References  Class DelegationInstance has the following references:

\#/delegationType : Delegation [0..1]  see Section A.4.2.10 on Page 199

derivation:

\n  connectorType.oclAsType(component::Delegation)

The delegation type of this delegation instance.

Class Constraints  Class DelegationInstance has the following constraints:

OneDelegationInstancePerPortInstance:

\nnot self.source.oclIsUndefined() implies self.source.outgoingConnectorInstances->select(x | x.oclIsKindOf(DelegationInstance))->size() = 1

Parent Classes

- ConnectorInstance see Section A.8.2.5 on Page 228

A.8.2.8. Class DiscreteMultiPortInstance

Overview  This class represents a multi-port at instance level. For each multi-port of a component, there exists exactly one multi-port instance in the respective component instance at all times. That instance references an instance of the statechart of the multi-port as well as an instance of the adaptation behavior. The DiscreteMultiPortInstance also references all sub-port instances of the multi-port instance. The DiscreteMultiPortInstance has no visual representation in the concrete syntax. It is represented by its sub-roles.

Class References  Class DiscreteMultiPortInstance has the following references:

\#/gmfSubPortInstances : DiscreteSinglePortInstance [0..*]  see Section A.8.2.10 on Page 231

derivation:

\n  self.subPortInstances

This reference just derives the values of "subPortInstances" and specifies a containment. This containment reference is needed by the GMF tooling.

\nsubPortInstances : DiscreteSinglePortInstance [0..*]  see Section A.8.2.10 on Page 231

These are all sub-port instances of the multi-port. The sub-port instances are represented by DiscreteSinglePortsInstances.
Parent Classes

- DiscretePortInstance see Section A.8.2.9 on Page 230

A.8.2.9. Class DiscretePortInstance

Overview This class represents a discrete port at instance level. At instance level, we distinguish between single-port instances and multi-port instances by using two subclasses of this abstract class.

Class References Class DiscretePortInstance has the following references:

/receiverMessageInterface : MessageInterface [0..1] see Section A.9.2.1 on Page 234

derivation:

  if portType.oclIsUndefined() or not portType.oclIsKindOf(component::DiscretePort) then
    null
  else
    portType.oclAsType(component::DiscretePort).
    receiverMessageInterface
  endif

The receiver message interface defines which messages this port instance receives. It is derived from the receiver message interface of its port.

/senderMessageInterface : MessageInterface [0..1] see Section A.9.2.1 on Page 234

derivation:

  if portType.oclIsUndefined() or not portType.oclIsKindOf(component::DiscretePort) then
    null
  else
    portType.oclAsType(component::DiscretePort).
    senderMessageInterface
  endif

The sender message interface defines which messages this port instance sends. It is derived from the sender message interface of its port.

Parent Classes

- PortInstance see Section A.8.2.13 on Page 232
A.8.2.10. **Class DiscreteSinglePortInstance**

**Overview**  This class represents a discrete single port at instance level as well as a sub-port instance of a multi-port instance. Each single port instance references its behavior instance. When used as a sub-port instance, the instance references its role behavior instance.

**Class References**  Class **DiscreteSinglePortInstance** has the following references:

- `multiPortInstance : DiscreteMultiPortInstance [0..1]`  see Section A.8.2.8 on Page 229

**Parent Classes**

- DiscretePortInstance see Section A.8.2.9 on Page 230

A.8.2.11. **Class HybridPortInstance**

**Overview**  This class represents a hybrid port at instance level. The port type of a hybrid port instance must be a hybrid port.

**Parent Classes**

- DiscretePortInstance see Section A.8.2.9 on Page 230,
- ContinuousPortInstance see Section A.8.2.6 on Page 228

A.8.2.12. **Class PatternInstance**

**Overview**  An instance of a pattern occurrence. It specifies the behavior of discrete (single/multi) port instances that are connected with each other.

**Class References**  Class **PatternInstance** has the following references:

- `patternOccurrence : PatternOccurrence`  see Section A.4.2.13 on Page 205
  The PatternOccurrence of that instance.
- `portInstances : PortInstance [1..*]`  see Section A.8.2.13 on Page 232
  The port instance that uses this pattern instance.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269
### A.8.2.13. Class PortInstance

**Overview**  A port instance is a port of a component at instance level.

**Class References**  Class PortInstance has the following references:

- **componentInstance : ComponentInstance [0..1]**  see Section A.8.2.3 on Page 227
  The component instance this port instance belongs to.

- **/connectorInstances : ConnectorInstance [0..*]**  see Section A.8.2.5 on Page 228
  The connector instances that lead into this port instance.

- **outgoingConnectorInstances : ConnectorInstance [0..*]**  see Section A.8.2.5 on Page 228
  The connector instances that origin from this port instance.

- **portType : Port**  see Section A.4.2.14 on Page 205
  The port type of which this port instance is built from.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267

### A.8.2.14. Class StructuredComponentInstance

**Overview**  A structured component instance is a component instance that has been derived from a structured component type. A structured component instance specifies an embedded component instance configuration defining its inner structure.

**Class References**  Class StructuredComponentInstance has the following references:

- **embeddedCIC : ComponentInstanceConfiguration**  see Section A.8.2.4 on Page 227
  The component instances and connector instances that are embedded in this component instance are contained by the component instance configuration.
Parent Classes

- ComponentInstance see Section A.8.2.3 on Page 227
A.9. Package `muml::model::msgiface`

A.9.1. Package Overview

This package defines the message interfaces. A MessageInterface defines a set of event signatures using the class MessageType. These message types are used to type the events within a realtime statechart.

![Diagram of the msgiface Package](image)

Figure A.7.: Meta-Model of the msgiface Package

A.9.2. Detailed Contents Documentation

A.9.2.1. Class MessageInterface

Overview  This class represents a message interface. A message interface specifies which messages are allowed to be sent or received by a port or role.

Class References  Class MessageInterface has the following references:

/allAvailableMessageTypes : MessageType [0..*] see Section A.9.2.2 on Page 235

 derivation:

```plaintext
self -> closure (if superType -> isEmpty() then self -> asSet() else superType endif).messageTypes -> asSet()
```
messageTypes : MessageType [0..∗]  see Section A.9.2.2 on Page 235
The message types being defined in this message interface.

superType : MessageInterface [0..∗]  see Section A.9.2.1 on Page 234
The set of message interfaces this message interface inherits from. This message interface contains all message types that are defined by the super types and their super types.

**Class Constraints**  
Class MessageInterface has the following constraints:

**NoCyclicGeneralization:**
\[ \text{not } \text{self} \rightarrow \text{closure} (\text{superType}) \rightarrow \text{includes} (\text{self}) \]

**UniqueMessageTypeNames:**
\[ \text{self}. \text{messageTypes} \rightarrow \text{isUnique}(\text{name}) \]

**NoMessageTypeOrNotAtLeastTwoGeneralizations:**
\[ \text{self}. \text{messageTypes} \rightarrow \text{size()} \geq 1 \text{ or } \text{self}. \text{superType} \rightarrow \text{size()} \geq 2 \]

**UniqueMessageInterfaceNames:**
\[ \text{MessageInterface}. \text{allInstances()}. \text{name} \rightarrow \text{count}(\text{self}. \text{name}) = 1 \]

**Parent Classes**
- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267

**A.9.2.2. Class MessageType**

**Overview**  
A message type defines the signature of one event. That includes the name of the event as well as the list of parameters. The message type inherits from callable because concrete events in a real-time statechart must provide a parameter mapping for the parameters of the message type as it is defined for method invocations.

**Class References**  
Class MessageType has the following references:

**messageInterface : MessageInterface**  see Section A.9.2.1 on Page 234
This is the message interface where this message type is defined in.

**parameters : Parameter [0..∗]**  see Section A.6.2.11 on Page 218
This reference defines the set of parameters of this message type. A parameter defines a unique name and a DataType.
Class Constraints  Class MessageType has the following constraints:

UniqueParameterNames:

\[
\text{self.parameters} \rightarrow \text{isUnique(name)}
\]

Parent Classes

- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267
A.10. Package `muml::model::pattern`

A.10.1. Package Overview

![Diagram of the pattern Package]

Figure A.8.: Meta-Model of the pattern Package

A.10.2. Detailed Contents Documentation

A.10.2.1. Class `CoordinationPattern`

**Overview**  A coordination protocol specifies the coordination between a certain number of communication members. The communication members are represented by roles. To specify which roles communicate with each other they are connected by channels. The communication protocol used by the roles is specified by realtime statecharts. Each role has its own realtime statechart describing the roles communication behavior. Furthermore channels own a realtime statechart which enables specifying properties of certain real communication channels e.g. propagation delay or buffering of messages. Furthermore constraints can be assigned
to coordination patterns. Constraints specify certain properties the coordination specified by the pattern has to fulfill.

**Class References**  Class CoordinationPattern has the following references:

- **connector : RoleConnector**  see Section A.10.2.3 on Page 241
  
  Each coordination pattern has exactly one role connector. Cardinality is 1 because there exists no useful pattern wir more than two roles. If a useful pattern exists with more than 2 roles, than change cardinality to 1..*

- **/pattern : CoordinationPattern [0..1]**  see Section A.10.2.1 on Page 237
  
  derivation:
  
  self

  This derived reference only exists because GMF needs it to visualize the inner ellipse of a Real-Time Coordination Pattern.

- **roles : Role [1..2]**  see Section A.10.2.2 on Page 238
  
  The roles belonging to this pattern.

**Class Constraints**  Class CoordinationPattern has the following constraints:

- **UniqueRoleNames:**
  
  self.roles ->isUnique(name)

- **CoordinationPatternNamesMustBeUnique:**
  
  CoordinationPattern.allInstances().name->count(self.name) = 1

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269,

- ConstrainableElement see Section A.6.2.7 on Page 216

### A.10.2.2. Class Role

**Overview**  This class represents a role of a coordination pattern.

**Class Properties**  Class Role has the following properties:

- **/isMultiRole : EBoolean [0..1]**
  
  derivation:
if not (self.cardinality.oclIsUndefined()) then
  (self.cardinality.upperBound.value > 1) or self.
cardinality.upperBound.infinity
else
  false
endif

This derived attribute indicates if the role is a multi role.

ordered : EBoolean [0..1]

This attribute marks a multi-role as being ordered. In an ordered multi-role, one of
the contained integer attributes is used to define the order. Then, the instances of
the multi-role are numbered from 1 to n for n instances.

Class References

Class Role has the following references:

adaptionBehavior : Behavior [0..1] see Section A.6.2.4 on Page 215

The adaptation behavior of this role. Note that only multi-ports have an adaptation
behavior.

cardinality : Cardinality see Section A.6.2.6 on Page 216

A role has a cardinality.

coordinationPattern : CoordinationPattern see Section A.10.2.1 on Page 237

The coordination pattern this role belongs to.

incomingRoleConnector : RoleConnector [0..1] see Section A.10.2.3 on Page 241

The incoming RoleConnector, which connects this role with another role. Either
incomingRoleConnector or outgoingRoleConnector (or both) must be set.

outgoingRoleConnector : RoleConnector [0..1] see Section A.10.2.3 on Page 241

The outgoing RoleConnector, which connects this role with another role. Either
incomingRoleConnector or outgoingRoleConnector (or both) must be set.

port : DiscretePort [0..*] see Section A.4.2.11 on Page 202

The ports this role is assigned to.

receiverMessageInterface : MessageInterface [0..1] see Section A.9.2.1 on Page 234

The receiver message interface defines which messages this port receives.

roleAndAdaptationBehavior : Behavior [0..1] see Section A.6.2.4 on Page 215

/roleConnector : RoleConnector [0..1] see Section A.10.2.3 on Page 241

derivation:
\[
\text{if } \text{self.incomingConnector} \rightarrow \text{notEmpty()} \text{ then } \\
\quad \text{self.incomingConnector} \\
\text{else} \quad \text{self.outgoingConnector} \\
\text{endif}
\]

\textit{senderMessageInterface : MessageInterface [0..1]} see Section \textit{A.9.2.1} on Page 234

The sender message interface defines which messages this port sends.

\textbf{Class Constraints} Class \texttt{Role} has the following constraints:

\textbf{MultiPortRequiresDefinedOrder:}

\[
\text{self.ordered implies (self.cardinality.upperBound.value} > 1 \text{ or } \\
\quad \text{self.cardinality.upperBound.infinity)}
\]

\textbf{OrderedRequiresIntegerOrderVariable:}

\[
\text{self.ordered implies (self.orderVariable} \rightarrow \text{notEmpty()} \text{ implies self} \\
\quad \text{.orderVariable.eAttributeType} = \text{'EInt'}}
\]

\textbf{RoleHasConnector:}

\[
\text{self.incomingRoleConnector} \rightarrow \text{notEmpty()} \text{ or self} \\
\quad \text{.outgoingRoleConnector} \rightarrow \text{notEmpty()}
\]

\textbf{RoleRequiresBehavior:}

\[
\text{not self.behavior.oclIsUndefined()}
\]

\textbf{RoleRequiresInterface:}

\[
\text{not (self.senderMessageInterface.oclIsUndefined() and self} \\
\quad \text{.receiverMessageInterface.oclIsUndefined())}
\]

\textbf{Parent Classes}

- NamedElement see Section \textit{A.13.2.4} on Page 269,
- ConstrainableElement see Section \textit{A.6.2.7} on Page 216,
- BehavioralElement see Section \textit{A.6.2.5} on Page 215
A.10.2.3. **Class** RoleConnector

**Overview**  This class represents a communication channel connecting two roles of a coordination pattern.

**Class Properties**  Class RoleConnector has the following properties:

- **bidirectional : EBoolean [0..1]**  
  This attribute stores the direction of the channel. The direction can either be uni- or bi-directional. This attribute should probably be renamed to bidirectional.

**Class References**  Class RoleConnector has the following references:

- **coordinationPattern : CoordinationPattern**  see Section A.10.2.1 on Page 237  
  The coordination pattern this role connector is part of.

- **source : Role**  see Section A.10.2.2 on Page 238  
  The roles connected by this channel. At the moment an arbitrary number of roles are allow. This probably should be discussed.

- **target : Role**  see Section A.10.2.2 on Page 238  
  The roles connected by this channel. At the moment an arbitrary number of roles are allow. This probably should be discussed.

**Class Constraints**  Class RoleConnector has the following constraints:

- **OnlyRolesOfSameCoordinationPattern:**  
  
  \[
  (\text{not } \text{source} \text{.oclIsUndefined()} \text{ and not } \text{target} \text{.oclIsUndefined()} ) \implies \text{source} \text{.coordinationPattern = target} \text{.coordinationPattern}
  \]

**Parent Classes**

- BehavioralElement see Section A.6.2.5 on Page 215
A.11. Package `muml::model::realtimestatechart`

A.11.1. Package Overview

A.11.2. Detailed Contents Documentation

A.11.2.1. Class `AbsoluteDeadline`

**Overview**  This class represents an absolute deadline. It is always associated with a transition of the statechart. The deadline depends on the value of a certain clock.

**Class References**  Class `AbsoluteDeadline` has the following references:

- **clock : Clock**  see Section A.11.2.4 on Page 244
  
  the references clock of the absolute deadline.

**Parent Classes**

- Deadline see Section A.11.2.6 on Page 245

A.11.2.2. Class `Action`

**Overview**  An action is used as a side effect of a transition as well as within a state. Each transition can only define one action. A state can define up to three actions (one for state entry, one for state exit, one while dwelling within the state).

**Class References**  Class `Action` has the following references:

- **expressions : Expression [1..*]**  see Section A.14.2.1 on Page 270
  
  An action has a defined expression, which can be expressed in different languages.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269

A.11.2.3. Class `AsynchronousMessageEvent`

**Overview**  An AsynchronousMessageEvent is a TransitionEvent that corresponds to receiving or sending a message. They are used to model asynchronous communication between realtime statecharts. A trigger events specifies that the corresponding message has to be received for the transition to be enabled, a raised event specifies that the corresponding message will be sent upon execution of the transition.
Figure A.9.: Meta-Model of the realtimestatechart Package
**Class References**  Class `AsynchronousMessageEvent` has the following references:

*message : Message*  see Section A.11.2.15 on Page 248

The message associated with this event. The message is either requested to be received (trigger event) or it will be sent (raise event).

**Parent Classes**

- TransitionEvent see Section A.11.2.28 on Page 261

### A.11.2.4. Class Clock

**Overview**  This class represents clocks of a realtime statechart.

**Class References**  Class `Clock` has the following references:

*statechart : RealtimeStatechart [0..1]*  see Section A.11.2.17 on Page 249

The realtime statechart this clock belongs to.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269

### A.11.2.5. Class ClockConstraint

**Overview**  This class represents an arbitrary time constraint that can either be used as an invariant constraint of a state or as a transition guard.

**Class Properties**  Class `ClockConstraint` has the following properties:

*operator : ComparingOperator*  see Section A.15.2.4 on Page 274

The operator that is used in this clock constraint.

**Class References**  Class `ClockConstraint` has the following references:

*bound : NaturalNumber*  see Section A.6.2.9 on Page 217

The bound of a deadline (upper or lower) is a natural number.

*clock : Clock*  see Section A.11.2.4 on Page 244

The clock references in this clock constraint.
A.11.2.6. **Class Deadline**

**Overview**  This class represents a deadline consisting of an upper and a lower bound.

**Class References**  Class Deadline has the following references:

- **lowerBound** : NaturalNumber [0..1] see Section A.6.2.9 on Page 217
  
  The lower bound of a deadline is a natural number.

- **upperBound** : NaturalNumber [0..1] see Section A.6.2.9 on Page 217
  
  The upper bound of a deadline is a natural number.

A.11.2.7. **Class DoEvent**

**Overview**  The action of a state that is executed periodically as long as this state is active. The first period starts after the execution of the entry-action.

**Class Properties**  Class DoEvent has the following properties:

- **periodLower** : Elnt [0..1]
  
  the lower bound of the period

- **periodUpper** : Elnt [0..1]
  
  the upper bound of the period

**Class References**  Class DoEvent has the following references:

- **action** : Action  see Section A.11.2.2 on Page 242
  
  Each entry or exit action has one or more actions.

**Class Constraints**  Class DoEvent has the following constraints:

- **ValidLowerUpperPeriod:**
  
  self.periodLower >= 1 and self.periodLower <= self.periodUpper

**Parent Classes**

- StateEvent see Section A.11.2.22 on Page 255

A.11.2.8. **Class EntryEvent**

**Overview**  This class represents an entry event. The action associated with this event will be executed when the state is entered.
Note  We need this subclass, because GMF forbids using the same semantic element for different notational elements within the same container.

Parent Classes

- EntryOrExitEvent see Section A.11.2.9 on Page 246

A.11.2.9. Class EntryOrExitEvent

Overview  This class represents an entry or an exit event. The actions associated with this event will be executed when the state is entered or left respectively.

Class References  Class EntryOrExitEvent has the following references:

  action : Action [0..1]  see Section A.11.2.2 on Page 242
  Each entry or exit event can have one or more actions.

  clockResets : Clock [0..*]  see Section A.11.2.4 on Page 244
  The clock resets of this action

Parent Classes

- StateEvent see Section A.11.2.22 on Page 255

A.11.2.10. ClassEntryPoint

Overview  An EntryPoint is an intermediate pseudostate which makes it possible to chain transitions between different hierarchy levels. An EntryPoint is used to activate a dedicated inner state of an embedded statechart.

Class Constraints  Class EntryPoint has the following constraints:

  OneOutgoingTransition:
  self.outgoingTransitions->size() = 1

  EntryPointAndTargetSameStatechart:
  self.outgoingTransitions->size() = 1 implies
  (not self.outgoingTransitions->at(1).target.oclIsUndefined()
   and self.outgoingTransitions->at(1).target.statechart = self.statechart
  )
Parent Classes

- Vertex see Section A.11.2.29 on Page 261

A.11.2.11. **Class Event**

**Overview**  This abstract class represents all kinds of events that may occur in a statechart. A event can either be a trigger event or a raise event.

**Class Properties**  Class Event has the following properties:

  kind : EventKind [0..1]  see Section A.11.2.12 on Page 247

Decides the kind: Is this a raise event or a trigger event?

A event may either be a trigger event or a raise event. A trigger event triggers some action within the statechart, a raise event is generated by the statechart and will be processed by another statechart.

A.11.2.12. **Enumeration EventKind**

**Overview**  An event has two kinds: raise and trigger.

**Enum Properties**  Enumeration EventKind has the following literals:

  RAISE = 0

  Represents a raise event.

  TRIGGER = 1

  Represents a trigger event.

A.11.2.13. **Class ExitEvent**

**Overview**  This class represents an exit event. The action associated with this event will be executed when the state is left.

**Note**  We need this subclass, because GMF forbids using the same semantic element for different notational elements within the same container.

Parent Classes

- EntryOrExitEvent see Section A.11.2.9 on Page 246
A.11.2.14. Class ExitPoint

Overview An ExitPoint is an intermediate pseudostate which makes it possible to chain transitions between different hierarchy levels. An ExitPoint is used to deactivate a dedicated inner state of an embedded statechart.

Class Constraints Class ExitPoint has the following constraints:

- **AtMostOneOutgoingTransition:**
  \[
  \text{self.outgoingTransitions} \rightarrow \text{size}() \leq 1
  \]

- **OneIncomingTransition:**
  \[
  \text{self.incomingTransitions} \rightarrow \text{size}() = 1
  \]

- **ExitPointAndSourceSameStatechart:**
  \[
  \text{self.incomingTransitions} \rightarrow \text{size}() = 1 \text{ implies (not self.incomingTransitions} \rightarrow \text{at}(1).\text{source.oclIsUndefined ()} \\
  \text{and self.incomingTransitions} \rightarrow \text{at}(1).\text{source.statechart = self.statechart)}
  \]

Parent Classes

- Vertex see Section A.11.2.29 on Page 261

A.11.2.15. Class Message

Overview The messages are exchanged between components in order to communicate asynchronously. A message is typed over a message type and provides a binding of all parameters defined by the message type to concrete values.

Class References Class Message has the following references:

- **instanceOf : MessageType** see Section A.9.2.2 on Page 235
  Retrieves the message type this message is typed over.

- **parameterBinding : ParameterBinding [0..*]** see Section A.6.2.12 on Page 218

Parent Classes

- ExtendableElement see Section A.13.2.2 on Page 267
**A.11.2.16. Class Prioritizable**

**Overview**  Enables the priorization of elements.

**Class Properties**  Class Prioritizable has the following properties:

- **priority**: EInt [0..1]
  The priority of the element

---

**A.11.2.17. Class RealtimeStatechart**

**Overview**  This class is a concrete statechart implementation of a real-time statechart.

**Class Properties**  Class RealtimeStatechart has the following properties:

- **/embedded**: EBoolean [0..1]
  
  derivation:
  
  ```
  not self.embeddingRegion.oclIsUndefined()
  ```

  This attribute specifies whether this realtime statechart is embedded into a region or not.

- **eventQueueSize**: EInt [0..1]
  
  The size of the event queue of this port. It defines the maximum number of events that may be temporarily buffered by the port.

- **/flat**: EBoolean [0..1]
  
  derivation:
  
  ```
  not (vertices -> exists ( v : Vertex | v.oclIsTypeOf(State) implies v.oclAsType(State).regions -> notEmpty())
  ```

  This derived attribute allows to checks whether a statechart is flat or not. In a flat statechart, none of the contained states contains a regions with an embedded substatechart.

- **history**: EBoolean [0..1]
  
  If this attribute is true, it acts as a shallow history on the top hierarchy of this statechart.

**Class References**  Class RealtimeStatechart has the following references:

- **/allAvailableAttributes**: Attribute [0..*]  see Section A.6.2.3 on Page 215
  
  derivation:
\texttt{self} \rightarrow \texttt{closure} (\texttt{idf} \texttt{embeddingRegion.oclIsUndefined()} \texttt{then} \texttt{self}
\texttt{else} \texttt{embeddingRegion.parentState.statechart endif}).
\texttt{attributes} \rightarrow \texttt{asOrderedSet()}

\texttt{/allAvailableOperations : Operation [0..*]} see Section A.6.2.10 on Page 217
derivation:
\texttt{self} \rightarrow \texttt{closure} (\texttt{idf} \texttt{embeddingRegion.oclIsUndefined()} \texttt{then} \texttt{self}
\texttt{else} \texttt{embeddingRegion.parentState.statechart endif}).
\texttt{operations} \rightarrow \texttt{asOrderedSet()}

\texttt{/availableClocks : Clock [0..*]} see Section A.11.2.4 on Page 244
derivation:
\texttt{self} \rightarrow \texttt{closure} (\texttt{idf} \texttt{embeddingRegion.oclIsUndefined()} \texttt{then} \texttt{self}
\texttt{else} \texttt{embeddingRegion.parentState.statechart endif}).\texttt{clocks}
\rightarrow \texttt{asSet()}

Available clocks are all clocks that were defined in this statechart or in ancestor statecharts.

\texttt{clocks : Clock [0..*]} see Section A.11.2.4 on Page 244
The clocks of this realtime statechart.

\texttt{embeddingRegion : Region [0..1]} see Section A.11.2.18 on Page 251
If the real-time statechart is embedded into a region of a composite state, than
this reference returns the region of this state. If the real-time statechart is not
embedded, this reference will be undefined.

\texttt{transitions : Transition [0..*]} see Section A.11.2.27 on Page 257
The transitions of the realtime statechart.

\texttt{vertices : Vertex [0..*]} see Section A.11.2.29 on Page 261
The states of this realtime statechart.

Class Constraints Class \texttt{RealtimeStatechart} has the following constraints:

\textbf{UniqueNameOfStates:}
\texttt{self.\texttt{vertices} \rightarrow \texttt{select}(\texttt{oclIsTypeOf(State)}).\texttt{oclAsType(State)} \rightarrow
\texttt{isUnique(name)}}

\textbf{MinOneState:}
\texttt{self.\texttt{vertices} \rightarrow \texttt{select}(\texttt{oclIsTypeOf(State)}).\texttt{oclAsType(State)} \rightarrow
\texttt{notEmpty()}}

\textbf{NoCycles:}
If we are contained within a statechart...
\[
\text{not self.embeddingRegion.parentState.statechart.oclIsUndefined()}
\]
implies

... then we must not be a super statechart of it.
\[
\text{not self.isSuperStatechartOf(self.embeddingRegion.parentState.statechart)}
\]

Parent Classes
- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267,
- Behavior see Section A.6.2.4 on Page 215

A.11.2.18. Class Region

Overview Regions enables hierarchy and parallelism. Each state can have zero, one or more regions.

Class References Class Region has the following references:
- parentState : State see Section A.11.2.20 on Page 252
  The state this region is embedded.
- statechart : RealtimeStatechart see Section A.11.2.17 on Page 249
  The realtime statechart this region embeds.

Parent Classes
- NamedElement see Section A.13.2.4 on Page 269,
- Prioritizable see Section A.11.2.16 on Page 249

A.11.2.19. Class RelativeDeadline

Overview This class represents a relative deadline. It is always associated with a transition of the statechart. The deadline is relative to the point in time when the execution of the transition starts.
**Parent Classes**

- Deadline see Section A.11.2.6 on Page 245

### A.11.2.20. Class State

**Overview** This class represents a complex state of a realtime statechart. Complex states may again contain realtime statecharts hence enabling the creation of hierarchical statecharts. Further more complex states have do, entry and exit actions. Also complex states define which synchronization channels are allowed to be used by embedded statecharts.

**Class Properties** Class State has the following properties:

- **final : EBoolean [0..1]**
  
a final state is not allowed to have outgoing transitions.

- **initial : EBoolean [0..1]**
  
An initial state is the first one to active if the statechart is activated. There is only one initial state allowed at the top hierarchy of a statechart.

- **/simple : EBoolean [0..1]**
  
derivation:
  
  `regions -> isEmpty()`

  A state is simple if it does not contain a region with an embedded substatechart.

- **urgent : EBoolean [0..1]**
  
If a state is active and urgent, no time is allowed to pass until the state is leaved.

**Class References** Class State has the following references:

- **channels : SynchronizationChannel [0..*]** see Section A.11.2.25 on Page 256
  
The synchronization channels provided by this state.

- **doEvent : DoEvent [0..1]** see Section A.11.2.7 on Page 245
  
The do event. It is executed periodically while the corresponding state is active.

- **entryEvent : EntryEvent [0..1]** see Section A.11.2.8 on Page 245
  
The entry action is executed once when the corresponding state is entered.

- **/events : StateEvent [0..*]** see Section A.11.2.22 on Page 255
  
derivation:
  
  `Set{entryEvent, exitEvent, doEvent} -> select (x | not x. oclIsUndefined())`
This derived reference returns all StateEvents of this state. The StateEvents of this state are all entry-, do- and exit-Events.

**exitEvent** : `ExitEvent [0..1]` see Section A.11.2.13 on Page 247

The exit action is executed once when the corresponding state is left.

**invariants** : `ClockConstraint [0..*]` see Section A.11.2.5 on Page 244

The invariant belonging to this complex state. It describes how long it is allowed to reside in this complex state depending on the values of the clocks.

**regions** : `Region [0..*]` see Section A.11.2.18 on Page 251

The regions of this state. Regions are used to model composite states. In case of one region, we have an xor superstate, in case of multiple regions, we have an AND-superstate.

**stateEntryPoints** : `StateEntryPoint [0..*]` see Section A.11.2.21 on Page 254

A state references its entry points. They can only exist, if a state embeds one or more statecharts.

**stateExitPoints** : `StateExitPoint [0..*]` see Section A.11.2.23 on Page 255

A state references its exit points. They can only exist, if a state embeds one or more statecharts.

**Class Constraints**

Class `State` has the following constraints:

**OneInvariantPerClock:**

\[
\text{self.invariants} \rightarrow \text{isUnique}(\text{clock})
\]

**OneInitialState:**

\[
\begin{align*}
\text{self.statemachine.vertices} & \rightarrow \text{select}(x \mid x.\text{oclIsKindOf(State)}) . \\
\text{oclAsType(State)} & \rightarrow \text{select}(s \mid s.\text{initial}) \rightarrow \text{size}(s) = 1
\end{align*}
\]

**NoOutgoingTransitionOfFinalState:**

\[
\text{self.final} \implies \text{self.outgoingTransitions} \rightarrow \text{isEmpty}()
\]

**NoRegionsOfFinalState:**

\[
\text{self.final} \implies \text{self.regions} \rightarrow \text{isEmpty}() 
\]

**UniquePrioritiesOfOutgoingTransitions:**

\[
\text{self.outgoingTransitions} \rightarrow \text{isUnique(\text{priority})}
\]

**UniquePrioritiesOfRegions:**

\[
\text{self.regions} \rightarrow \text{isUnique(\text{priority})}
\]
UniqueChannelNames:
    self.channels -> isUnique(name)

UniqueRegionNames:
    self.regions -> isUnique(name)

BoundOfInvariantGreaterOrEqualZero:
    self.invariants -> forall(bound.value >= 0)

InvalidClockConstraintOperator:
    self.invariants -> forall(invariant | Set{core::expressions::common::ComparingOperator::LESS, core::expressions::common::ComparingOperator::LESS_OR_EQUAL | includes(invariant.operator))

Parent Classes

- Vertex see Section A.11.2.29 on Page 261

A.11.2.21. Class StateEntryPoint

Overview The StateEntryPoint is assigned to a state. An EntryPoint is an intermediate pseudostate which reference EntryPoints of embedded statecharts. The incoming transitions are chained with the outgoing transition of the referenced EntryPoints on the lower hierarchy level.

Class References Class StateEntryPoint has the following references:

- entryPoint : EntryPoint [1..*] see Section A.11.2.10 on Page 246
  Referenced EntryPoints of embedded statecharts.

- state : State see Section A.11.2.20 on Page 252
  The StateEntryPoint is assigned to a state.

Class Constraints Class StateEntryPoint has the following constraints:

- AtLeastOneIncomingTransition:
  self.incomingTransitions -> size() > 0

- AtLeastOneOutgoingTransition:
  self.outgoingTransitions -> size() > 0
Parent Classes

- Vertex see Section A.11.2.29 on Page 261

A.11.2.22. Class StateEvent

Overview  A StateEvent is an event that occurs within a state of a real-time statechart. StateEvents may only be trigger events.

Parent Classes

- Event see Section A.11.2.11 on Page 247

A.11.2.23. Class StateExitPoint

Overview  The StateExitPoint is assigned to a state. An ExitPoint is an intermediate pseudostate which reference ExitPoints of embedded statecharts. The incoming transition of the referenced ExitPoints on the lower hierarchy level are chained with the outgoing transition of the StateExitPoint.

Class References  Class StateExitPoint has the following references:
    
    - exitPoint : ExitPoint [1..*] see Section A.11.2.14 on Page 248
      
      Referred ExitPoints of embedded statecharts.
    
    - state : State see Section A.11.2.20 on Page 252
      
      The StateExitPoint is assigned to a state.

Class Constraints  Class StateExitPoint has the following constraints:

    OneOutgoingTransition: self.outgoingTransitions ->size () = 1

Parent Classes

- Vertex see Section A.11.2.29 on Page 261
A.11.2.24. **Class Synchronization**

**Overview** Two transitions can synchronize fire. One transition is the sender, the other the receiver. This means that both transitions (exactly one sender and one receiver) must be activated and has to fire at the same time.

**Class Properties** Class `Synchronization` has the following properties:

- `kind : SynchronizationKind` see Section A.11.2.26 on Page 257
  
  Decides the kind: Is this a send or a receive synchronization?

**Class References** Class `Synchronization` has the following references:

- `parameterBinding : ParameterBinding [0..*]` see Section A.6.2.12 on Page 218
  
  ParameterBindings that belong to a synchronization.

- `syncChannel : SynchronizationChannel [0..1]` see Section A.11.2.25 on Page 256
  
  the channel that is used by the synchronization

**Parent Classes**

- ExtendableElement see Section A.13.2.2 on Page 267

A.11.2.25. **Class SynchronizationChannel**

**Overview** Defines a type of a synchronization channel that can be used to synchronize between statecharts contained as substatecharts in the same state. Serves as a type for Synchronizations.

**Class References** Class `SynchronizationChannel` has the following references:

- `parameters : Parameter [0..*]` see Section A.6.2.11 on Page 218
  
  Parameters of a SynchronizationChannel.

- `state : State` see Section A.11.2.20 on Page 252
  
  The state in which this synchronization channel is defined.

**Parent Classes**

- NamedElement see Section A.13.2.4 on Page 269,

- CommentableElement see Section A.13.2.1 on Page 267
A.11.2.26. **Enumeration SynchronizationKind**

**Overview**  A synchronization has two kinds: send and receive.

**Enum Properties**  Enumeration *SynchronizationKind* has the following literals:

- **SEND = 0**  
  Represents a send synchronization.

- **RECEIVE = 1**  
  Represents a receive synchronization.

A.11.2.27. **Class Transition**

**Overview**  A transition connects different vertices. If the vertex is a state a self-transition is also possible.

**Class Properties**  Class *Transition* has the following properties:

- **blockable : EBoolean [0..1]**  
  Needed for failure propagation.

**Class References**  Class *Transition* has the following references:

- **absoluteDeadlines : AbsoluteDeadline [0..*]**  [See Section A.11.2.1 on Page 242]
  
  A transition can have one or more absolute deadlines

- **action : Action [0..1]**  [See Section A.11.2.2 on Page 242]
  
  The side effect of this transition. A side effect might be a variable assignment as well as a method invocation.

- **clockConstraints : ClockConstraint [0..*]**  [See Section A.11.2.5 on Page 244]
  
  A clock constraint restricts when the transition can be activated in dependency of the values of the clock.

- **clockResets : Clock [0..*]**  [See Section A.11.2.4 on Page 244]
  
  The clock resets of this transition.

- **events : TransitionEvent [0..*]**  [See Section A.11.2.28 on Page 261]
  
  All events which belong to this transition.

- **guard : Expression [0..1]**  [See Section A.14.2.1 on Page 270]
  
  The guard of a transition is defined by an expression which should have return type boolean. Comparing clock values is not allowed (use clock constraints instead).
raiseMessageEvent : AsynchronousMessageEvent [0..1]  see Section A.11.2.3 on Page 242

derivation:

let eventSet : Sequence(AsynchronousMessageEvent) = self.
events -> select(e | e.oclIsKindOf(AsynchronousMessageEvent)
and e.kind=EventKind::RAISE).oclAsType(AsynchronousMessageEvent) in
if eventSet -> size() = 0 then null else eventSet -> first()
endif

The event which is raised upon activation of this transition.

relativeDeadline : RelativeDeadline [0..1]  see Section A.11.2.19 on Page 251

a transition can have one relative deadline

source : Vertex  see Section A.11.2.29 on Page 261

The state which is the source of this transition.

statechart : RealtimeStatechart [0..1]  see Section A.11.2.17 on Page 249

The realtime statechart this transition belongs to.

synchronization : Synchronization [0..1]  see Section A.11.2.24 on Page 256

The synchronisation which is sent upon activation of this transition.

target : Vertex  see Section A.11.2.29 on Page 261

The state which is the target of this transition.

triggerMessageEvent : AsynchronousMessageEvent [0..1]  see Section A.11.2.3 on Page 242

derivation:

let eventSet : Sequence(AsynchronousMessageEvent) = self.
events -> select(e | e.oclIsKindOf(AsynchronousMessageEvent)
and e.kind=EventKind::TRIGGER).oclAsType(AsynchronousMessageEvent) in
if eventSet -> size() = 0 then null else eventSet -> first()
endif

The trigger event of this transition.

Class Constraints  Class Transition has the following constraints:

SetTargetAndSource:

self.target ->notEmpty() and self.source ->notEmpty()

NoCrossingOfRegionBorders:
self.source.statechart.embeddingRegion = self.target.statechart.embeddingRegion
  or
self.source.oclAsType(StateEntryPoint).statechart.embeddingRegion =
self.target.statechart.embeddingRegion.parentState.statechart.
  embeddingRegion
  or
self.source.statechart.embeddingRegion.parentState.statechart.
  embeddingRegion =
self.target.oclAsType(StateExitPoint).statechart.embeddingRegion

EntryPointMustOnlyPointToStatesOrStateEntryPoints:

  self.source.oclIsKindOf(EntryPoint) implies (
    self.target.oclIsKindOf(State)
    or
    self.target.oclIsKindOf(StateEntryPoint)
  )

ExitPointMustOnlyPointToStatesOrStateExitPoints:

  self.source.oclIsKindOf(ExitPoint) implies (
    self.target.oclIsKindOf(State)
    or
    self.target.oclIsKindOf(StateExitPoint)
  )

TriggerMessageEventsMustNotHaveAnOwnedParameterBinding:

  not self.triggerMessageEvent.message.oclIsUndefined() implies
  self.triggerMessageEvent.message.parameterBinding->notEmpty()

ValidTriggerMessageEvents:

  let a : msgiface::MessageInterface =
    (          
      if statechart.getPortOrRoleStatechart().behavioralElement.oclIsKindOf(component::DiscretePort) then
        statechart.getPortOrRoleStatechart().behavioralElement.oclAsType(component::DiscretePort).receiverMessageInterface
      else
        if statechart.getPortOrRoleStatechart().behavioralElement.oclIsKindOf(pattern::Role) then
          statechart.getPortOrRoleStatechart().behavioralElement.oclAsType(pattern::Role).receiverMessageInterface
        else
          null
        endif
      endif
    ) in
    (not triggerMessageEvent.message.instanceOf.oclIsUndefined()) implies
    (not a.oclIsUndefined() and a.messageTypes->includes(triggerMessageEvent.message.instanceOf))
ValidRaiseMessageEvents:

```
let a : msgiface :: MessageInterface =
  (  
    if statechart . getPortOrRoleStatechart() . behavioralElement .
        oclIsKindOf(component :: DiscretePort) then
      statechart . getPortOrRoleStatechart() . behavioralElement .
        oclAsType(component :: DiscretePort) .
        senderMessageInterface
    else
      if statechart . getPortOrRoleStatechart() . behavioralElement .
        oclIsKindOf(pattern :: Role) then
        statechart . getPortOrRoleStatechart() . behavioralElement .
        oclAsType(pattern :: Role) . senderMessageInterface
      else
        null
    endif
  endif
)
  in
  (not raiseMessageEvent . message . instanceOf . oclIsUndefined())
  implies (not a . oclIsUndefined() and a . messageTypes -> includes(
    raiseMessageEvent . message . instanceOf))
```

EntryPointOutgoingTransitionNoAdditionalElements:

```
( not self . source . oclIsUndefined() and self . source . oclIsTypeOf(
    realtime statechart :: EntryPoint))
implies (  
  self . synchronization . oclIsUndefined()  
  and self . clockResets -* isEmpty ()
  and self . triggerMessageEvent . oclIsUndefined()  
  and self . raiseMessageEvent . oclIsUndefined()  
  and self . clockConstraints -* isEmpty ()
  and self . absoluteDeadlines -* isEmpty ()
  and self . relativeDeadline . oclIsUndefined()  
  and self . guard . oclIsUndefined()  
  and self . events -* isEmpty ()
  and self . action . oclIsUndefined() )
```

StateExitPointOutgoingTransitionNoAdditionalElements:

```
( not self . source . oclIsUndefined() and self . source . oclIsTypeOf(
    realtime statechart :: StateExitPoint))
implies (  
  self . synchronization . oclIsUndefined()  
  and self . clockResets -* isEmpty ()
  and self . triggerMessageEvent . oclIsUndefined()  
  and self . raiseMessageEvent . oclIsUndefined()  
  and self . clockConstraints -* isEmpty ()
  and self . absoluteDeadlines -* isEmpty ()
  and self . relativeDeadline . oclIsUndefined()  
)```
A.11. PACKAGE Muml::Model::RealtimeStatechart

\[
\text{and self.guard.oclIsUndefined ( )}
\text{and self.events \rightarrow isEmpty ( )}
\text{and self.action.oclIsUndefined ( )}
\]

Parent Classes

- ExtendableElement see Section A.13.2.2 on Page 267,
- Prioritizable see Section A.11.2.16 on Page 249

A.11.2.28. Class TransitionEvent

Overview A TransitionEvent is an event that occurs at a transition of a real-time statechart. Trigger Events are part of the precondition for activating the transition, raise events are generated as a result of firing the transition.

Parent Classes

- Event see Section A.11.2.11 on Page 247

A.11.2.29. Class Vertex

Overview This class represents a node in a realtime statechart that is connected with other nodes via transitions.

Class References Class Vertex has the following references:

- incomingTransitions : Transition [0..*] see Section A.11.2.27 on Page 257
  The incoming transitions of this vertex
- outgoingTransitions : Transition [0..*] see Section A.11.2.27 on Page 257
  The outgoing transitions of this vertex
- statechart : RealtimeStatechart [0..1] see Section A.11.2.17 on Page 249
  The realtime statechart this state belongs to.

Parent Classes

- NamedElement see Section A.13.2.4 on Page 269
A.12. Package actionLanguage

A.12.1. Package Overview

The base package for all expressions which can be used for modeling actions, guards, invariants...

A.12.2. Detailed Contents Documentation

A.12.2.1. Enumeration AssignOperator

Overview

Enum Properties  Enumeration AssignOperator has the following literals:

- UNSET = 0
- ASSIGN = 1
- PLUS_EQUAL = 2
- EQUAL_PLUS = 3
- MINUS_EQUAL = 4
- EQUAL_MINUS = 5

A.12.2.2. Class Assignment

Overview

Class Properties  Class Assignment has the following properties:

- assignOperator : AssignOperator [0..1] see Section A.12.2.1 on Page 262
- incrementDecrementOperator : IncrementDecrementOperator [0..1] see Section A.12.2.8 on Page 265

Class References  Class Assignment has the following references:

- assignExpression : Expression [0..1] see Section A.14.2.1 on Page 270
- attribute : Attribute [0..1] see Section A.6.2.3 on Page 215

Parent Classes

- Expression see Section A.14.2.1 on Page 270
Figure A.10.: Meta-Model of the actionlanguage Package
A.12.2.3. Class AttributeExpression

Overview

Class References  Class AttributeExpression has the following references:

attribute : Attribute [0..1] see Section A.6.2.3 on Page 215

Parent Classes

• Expression see Section A.14.2.1 on Page 270

A.12.2.4. Class Block

Overview

Class References  Class Block has the following references:

expressions : Expression [0..∗] see Section A.14.2.1 on Page 270

Parent Classes

• Expression see Section A.14.2.1 on Page 270

A.12.2.5. Class DoWhileLoop

Overview

Parent Classes

• Loop see Section A.12.2.9 on Page 265

A.12.2.6. Class ForLoop

Overview

Class References  Class ForLoop has the following references:

countingExpression : Assignment [0..1] see Section A.12.2.2 on Page 262

initializeExpression : Assignment [0..1] see Section A.12.2.2 on Page 262

Parent Classes

• Loop see Section A.12.2.9 on Page 265
A.12.2.7. **Class IfStatement**

**Overview**

**Class References** Class IfStatement has the following references:
- `elseBlock : Block [0..1]` see Section A.12.2.4 on Page 264
- `elseIfBlocks : Block [0..*]` see Section A.12.2.4 on Page 264
- `elseIfConditions : Expression [0..*]` see Section A.14.2.1 on Page 270
- `ifBlock : Block [0..1]` see Section A.12.2.4 on Page 264
- `ifCondition : Expression [0..1]` see Section A.14.2.1 on Page 270

**Parent Classes**
- Expression see Section A.14.2.1 on Page 270

A.12.2.8. **Enumeration IncrementDecrementOperator**

**Overview**

**Enum Properties** Enumeration IncrementDecrementOperator has the following literals:

- UNSET = 0
- INCREMENT = 2
- DECREMENT = 1

A.12.2.9. **Class Loop**

**Overview**

**Class References** Class Loop has the following references:
- `block : Block [0..1]` see Section A.12.2.4 on Page 264
- `loopTest : Expression [0..1]` see Section A.14.2.1 on Page 270

**Parent Classes**
- Expression see Section A.14.2.1 on Page 270
A.12.2.10. **Class OperationCall**

**Overview**

**Class References**  Class OperationCall has the following references:

- operation : Operation  see Section A.6.2.10 on Page 217
- parameterBinding : ParameterBinding [0..*]  see Section A.6.2.12 on Page 218

**Parent Classes**

- Expression see Section A.14.2.1 on Page 270

A.12.2.11. **Class WhileLoop**

**Overview**

**Parent Classes**

- Loop see Section A.12.2.9 on Page 265
A.13. Package core

A.13.1. Package Overview

The core package is the root package for the storydriven core meta-model. It defines several abstract super classes which implement an extension mechanism as well as recurring structural features like, e.g., names of elements. The classes in this package are intended to be subclassed by any meta-model element.

A.13.2. Detailed Contents Documentation

A.13.2.1. Class CommentableElement

Overview  Abstract super class for all meta-model elements that may carry a comment in form of a string.

Class Properties  Class CommentableElement has the following properties:

   - `comment : EString [0..1]`
     The comment string that can be used to attach arbitrary information to CommentableElements.

Parent Classes

- ExtendableElement see Section A.13.2.2 on Page 267

A.13.2.2. Class ExtendableElement

Overview  Abstract base class for the whole story diagram model. The ExtendableElement specifies the extension mechanism that can be used to extend an object by an Extension containing additional attributes and references.

Class References  Class ExtendableElement has the following references:

   - `annotation : EAnnotation [0..*]`
   - `extension : Extension [0..*]` see Section A.13.2.3 on Page 269

Parent Classes

- EObject
Figure A.11.: Meta-Model of the core Package
A.13.2.3. **Class Extension**

**Overview**  
Abstract super class for an Extension that can be defined for an object.

**Class References**  
Class `Extension` has the following references:

- `/base : EObject`
- `extendableBase : ExtendableElement [0..1]`  
  see Section A.13.2.2 on Page 267
- `/modelBase : EModelElement [0..1]`
- `/owningAnnotation : EAnnotation [0..1]`

**Parent Classes**

- ExtendableElement see Section A.13.2.2 on Page 267

A.13.2.4. **Class NamedElement**

**Overview**  
Abstract super class for all meta-model elements that carry a name.

**Class Properties**  
Class `NamedElement` has the following properties:

- `name : EString`
  The name attribute of a meta-model element.

**Parent Classes**

- ExtendableElement see Section A.13.2.2 on Page 267

A.13.2.5. **Class TypedElement**

**Overview**  
Abstract super class for all meta-model elements that are typed by means of an EClassifier or an EGenericType.

**Class References**  
Class `TypedElement` has the following references:

- `genericType : EGenericType [0..1]`
- `/type : EClassifier [0..1]`

**Parent Classes**

- ExtendableElement see Section A.13.2.2 on Page 267
A.14. Package `core::expressions`

A.14.1. Package Overview

The base package for all expressions which can be used for modeling activities and patterns.

![Diagram of the meta-model of the expressions package](image)

Figure A.12.: Meta-Model of the expressions Package

A.14.2. Detailed Contents Documentation

A.14.2.1. Class `Expression`

**Overview** Represents any expression in an embedded textual language, e.g. OCL or Java. An expression’s type is dynamically derived by an external mechanism (see `TypedElement`).

**Parent Classes**
- `CommentableElement` see Section A.13.2.1 on Page 267

A.14.2.2. Class `TextualExpression`

**Overview** Represents any expression in a textual language embedded into Story Diagrams, e.g. OCL or Java.

**Class Properties** Class `TextualExpression` has the following properties:
expressionText : EString
   Holds the expression, e.g. in OCL or Java.

language : EString
   String representation of the used language which has to be unique. Examples are OCL and Java.

languageVersion : EString [0..1]
   String representation of the used language’s version. The format is <Major>.<Minor>[.<Revision>[.<Build>]] Examples: 1.4 or 3.0.1 or 1.0.2.20101208.

Parent Classes

- Expression see Section A.14.2.1 on Page 270
A.15. Package `core::expressions::common`

A.15.1. Package Overview

Represents arithmetic expressions like a + 5 or a * 7.

Class Properties

A.15.2. Detailed Contents Documentation

A.15.2.1. Class `ArithmeticExpression`

Overview

Class `ArithmeticExpression` has the following properties:
operator : ArithmeticOperator  see Section A.15.2.2 on Page 273

Specifies the expression’s arithmetic operator, e.g. +, -, *, /, or MODULO.

Parent Classes

- BinaryExpression see Section A.15.2.3 on Page 273

A.15.2.2. Enumeration ArithmeticOperator

Overview  Defines the operators for arithmetic expressions.

Enum Properties  Enumeration ArithmeticOperator has the following literals:

- PLUS = 0
- MINUS = 1
- TIMES = 2
- DIVIDE = 3
- MODULO = 4

A.15.2.3. Class BinaryExpression

Overview  Represents any binary expression like v < 5 or x + 7.

Class References  Class BinaryExpression has the following references:

- leftExpression : Expression  see Section A.14.2.1 on Page 270

  Represents the first operand of a binary expression, e.g. x in the expression x < 5.

- rightExpression : Expression  see Section A.14.2.1 on Page 270

  Represents the second operand of a binary expression, e.g. 5 in the expression x < 5.

Parent Classes

- Expression see Section A.14.2.1 on Page 270
A.15.2.4. **Enumeration ComparingOperator**  
**Overview**  Defines the operators for comparing expressions.

**Enum Properties**  Enumeration ComparingOperator has the following literals:

- LESS = 0
- LESS_OR_EQUAL = 1
- EQUAL = 2
- GREATER_OR_EQUAL = 3
- GREATER = 4
- UNEQUAL = 5
- REGULAR_EXPRESSION = 6

For comparison of a String with a regular expression.

A.15.2.5. **Class ComparisonExpression**  
**Overview**  Represents comparing expressions like a < 5 or a >= 7.

**Class Properties**  Class ComparisonExpression has the following properties:

- **operator : ComparingOperator**  see Section A.15.2.4 on Page 274
  
  Specifies the expression's comparing operator, e.g. <, >=, !=.

**Parent Classes**  
- BinaryExpression see Section A.15.2.3 on Page 273

A.15.2.6. **Class LiteralExpression**  
**Overview**  Represents any literal, i.e. a value whose type is an EDataType. Literals are, for example, 5, 3.14, 'c', "text", true.

**Class Properties**  Class LiteralExpression has the following properties:

- **value : EString [0..1]**

  String representation of the value, e.g. "5", "3.14", "c", "text", or "true".

**Parent Classes**  
- Expression see Section A.14.2.1 on Page 270
A.15.2.7. Enumeration **LogicOperator**

**Overview**  Defines the operators for binary logic expressions. The unary logic expression representing negated expressions is reflected by the NotExpression.

**Enum Properties**  Enumeration `LogicOperator` has the following literals:

- `AND = 0`
- `OR = 1`
- `XOR = 2`
- `IMPLY = 3`
- `EQUIVALENT = 4`

A.15.2.8. Class **LogicalExpression**

**Overview**  Represents binary, logic expressions like a AND b and a OR b.

**Class Properties**  Class `LogicalExpression` has the following properties:

- `operator : LogicOperator`  see Section A.15.2.7 on Page 275

  Specifies the expression’s logic operator, e.g. AND, OR, or XOR.

**Parent Classes**

- BinaryExpression see Section A.15.2.3 on Page 273

A.15.2.9. Class **UnaryExpression**

**Overview**  Represents a negated expression, e.g. NOT (a < 5).

**Class Properties**  Class `UnaryExpression` has the following properties:

- `operator : UnaryOperator`  see Section A.15.2.10 on Page 276

**Class References**  Class `UnaryExpression` has the following references:

- `enclosedExpression : Expression`  see Section A.14.2.1 on Page 270

  Represents the operand of a NotExpression, e.g. a < 5 in NOT (a < 5).

**Parent Classes**

- Expression see Section A.14.2.1 on Page 270
A.15.2.10. Enumeration UnaryOperator

Overview

Enum Properties  Enumeration UnaryOperator has the following literals:

- NOT = 0
- PLUS = 1
- MINUS = 2
A.16. Package storydiagrams

A.16.1. Package Overview

The storydiagram package is the root package for the story diagram meta-model. It defines the type Variable and otherwise is only used to contain more specific sub-packages.

Figure A.14.: Metamodel of the storydiagrams Package

A.16.2. Detailed Contents Documentation

A.16.2.1. Class Variable

Overview  Represents a variable which can be, for example, an object variable, an attribute, or any other kind of variable.

Class Properties  Class Variable has the following properties:

/variableName : EString [0..1]

Parent Classes

• TypedElement see Section A.13.2.5 on Page 269
A.17. Package storydiagrams::activities

A.17.1. Package Overview

A.17.2. Detailed Contents Documentation

A.17.2.1. Class Activity

Overview The diagram that describes the control flow of an operation. It is used to structure a number story patterns into a story diagram. Story patterns are contained in activity nodes which are connected by activity edges. In addition, there are special nodes like start, stop, and junction nodes.

Class References Class Activity has the following references:

- ownedActivityEdge : ActivityEdge [0..*] see Section A.17.2.3 on Page 280
  All ActivityEdges that are contained in this activity.

- ownedActivityNode : ActivityNode [0..*] see Section A.17.2.5 on Page 281
  The activity contains all activity nodes via this reference.

- owningOperation : OperationExtension [0..1] see Section A.17.2.13 on Page 285
  References a story node which represents the precondition for the execution of the activity. I.e., the activity is executed, iff the object structure in the story node can be matched. Obviously the referenced story node may only contain black (i.e., non-create and non-destroy) objects and links.

Parent Classes
- Callable see Section A.19.2.1 on Page 289,
- NamedElement see Section A.13.2.4 on Page 269

A.17.2.2. Class ActivityCallNode

Overview The ActivityCallNode is a special ActivityNode which represents the calling of another story diagram within an activity. To support polymorphic dispatching, multiple activities can be assigned to it (all of which must have the same call signature, i.e. matching in and out parameters). All assigned activities are then called in the given order and the first one whose precondition is fulfilled is executed (Chain of Responsibility).

Class References Class ActivityCallNode has the following references:
Figure A.15.: Metamodel of the activities Package
calledActivity : Activity [1..*]  see Section A.17.2.1 on Page 278

References all activities that are to be considered for the polymorphic dispatching of the call. All activities must have the same call signature.

Parent Classes

- ActivityNode see Section A.17.2.5 on Page 281,
- Invocation see Section A.19.2.2 on Page 289

A.17.2.3. Class ActivityEdge

Overview  The ActivityEdge represents the control flow in an activity. It is a directed connection from one activity to another one. There exist different kinds of activity edges which are differentiated by the guard attribute.

Class Properties  Class ActivityEdge has the following properties:

- guard : EdgeGuard  see Section A.17.2.6 on Page 282

  The guard defines the kind of the activity edge. The possible kinds of guards are specified by the EdgeGuard enum.

Class References  Class ActivityEdge has the following references:

- guardException : ExceptionVariable [0..*]  see Section A.17.2.7 on Page 283

  Declares variables representing the Exceptions that lead to firing this transition.

- guardExpression : Expression [0..1]  see Section A.14.2.1 on Page 270

  Points to an expression in case the transition guard is BOOL. The expression has to evaluate to a boolean value.

- owningActivity : Activity  see Section A.17.2.1 on Page 278

  Points to the activity this ActivityEdge is contained in.

- source : ActivityNode  see Section A.17.2.5 on Page 281

  The source node of this ActivityEdge.

- target : ActivityNode  see Section A.17.2.5 on Page 281

  The target node of this ActivityEdge.

Parent Classes

- ExtendableElement see Section A.13.2.2 on Page 267
A.17.2.4. Class ActivityFinalNode

Overview  At a StopNode, the execution of an activity terminates. If the activity specifies any out-parameters, they have to be bound to a return expression.

Class Properties  Class ActivityFinalNode has the following properties:

success : EBoolean

Class References  Class ActivityFinalNode has the following references:

/returnValue : Expression [0..1]  see Section A.14.2.1 on Page 270
    Convenience method when dealing with activities that implement an EOperation. In this case, only one out parameter is supported. This attributes then returns the first out parameter.

returnValues : Expression [0..*]  see Section A.14.2.1 on Page 270
    Defines the return values of the activity. These return values will be assigned to the out-parameters.

Parent Classes

- ActivityNode see Section A.17.2.5 on Page 281

A.17.2.5. Class ActivityNode

Overview  Abstract super class for all kinds of nodes that may be added to an activity. This class provides the basic functionality of connecting the activity nodes in the activity by ActivityEdges.

Class References  Class ActivityNode has the following references:

incoming : ActivityEdge [0..*]  see Section A.17.2.3 on Page 280
    All ActivityEdges entering this activity node.

outgoing : ActivityEdge [0..*]  see Section A.17.2.3 on Page 280
    All ActivityEdges that leave this activity node. The guards of the outgoing activity edges must be exclusive in order to obtain a well-defined activity.

owningActivity : Activity [0..1]  see Section A.17.2.1 on Page 278
    Points to the activity this ActivityNode is contained in.

owningActivityNode : StructuredNode [0..1]  see Section A.17.2.16 on Page 286
    The parent node if this node is contained in a StructuredNode.
Parent Classes

- NamedElement see Section A.13.2.4 on Page 269,
- CommentableElement see Section A.13.2.1 on Page 267

A.17.2.6. **Enumeration EdgeGuard**

**Overview**  This enum is used to model different kinds of activity edges.

**Enum Properties**  Enumeration EdgeGuard has the following literals:

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE = 0</td>
<td>No guard, only one outgoing activity edge of this kind is supported per activity node. If an edge with EdgeGuard NONE is used, it must be the only edge leaving a state.</td>
</tr>
<tr>
<td>SUCCESS = 1</td>
<td>Edge will be taken if execution of the source activity node was successful, e.g., a story pattern was matched successfully. There must be another edge leaving the same node which is of kind FAILURE.</td>
</tr>
<tr>
<td>FAILURE = 2</td>
<td>Edge will be taken if execution of the source activity node was not successful, e.g., a story pattern could not be matched. There must be another edge leaving the same node which is of kind SUCCESS.</td>
</tr>
<tr>
<td>EACH_TIME = 3</td>
<td>Edge may only leave a StoryNode whose forEach attribute is true. It will be taken for each match that can be identified for the story pattern in the foreach StoryNode. There must be another edge leaving the same node which is of kind END.</td>
</tr>
<tr>
<td>END = 4</td>
<td>Edge may only leave a StoryNode whose forEach attribute is true. It will be taken if no more fresh matches for the story pattern in the foreach node can be found.</td>
</tr>
<tr>
<td>ELSE = 5</td>
<td>Complement to the BOOL guard, ELSE may only be used if at least one BOOL activity edge leaves the same state. The edge will be taken if none of the BOOL guards can be evaluated to true.</td>
</tr>
<tr>
<td>BOOL = 6</td>
<td>An activity edge specifying a boolean guard using variables that have been previously used in the activity. Edge will be taken if the guardExpression of the activity edge evaluates to true. More than one BOOL edge is allowed to leave an activity node.</td>
</tr>
</tbody>
</table>
**EXCEPTION = 7**

An EXCEPTION edge will be taken if an exception of the type defined by the ExceptionVariable connected to the activity edge occurred while executing the source activity node of the edge. More than one edge of kind EXCEPTION is allowed to leave a node.

**FINALLY = 8**

An activity edge of kind FINALLY may only leave an activity node that has at least one other outgoing edge of kind EXCEPTION. The finally edge will be taken after the source node has been executed and after, possibly, the EXCEPTION edge has been taken.

**A.17.2.7. Class ExceptionVariable**

**Overview** Declares a variable representing an Exception that leads to firing a transition (ActivityEdge). Can only be applied to ActivityEdge whose guard is set to EXCEPTION.

**Class Properties** Class ExceptionVariable has the following properties:

- **name : EString**
  Specifies the name of the declared exception variable.

**Class References** Class ExceptionVariable has the following references:

- **activityEdge : ActivityEdge** see Section A.17.2.3 on Page 280
  Specifies the transition (activity edge) where the exception variable is declared.

- **exceptionType : EClassifier [0..*]**
  Specifies the type of the declared exception variable.

- **genericExceptionType : EGenericType [0..*]**

**Parent Classes**

- Variable see Section A.16.2.1 on Page 277

**A.17.2.8. Class FlowFinalNode**

**Overview**

**Parent Classes**

- ActivityFinalNode see Section A.17.2.4 on Page 281
A.17.2.9. Class InitialNode

Overview The start node of an activity defines the starting point for the execution of the activity.

Parent Classes
- ActivityNode see Section A.17.2.5 on Page 281

A.17.2.10. Class JunctionNode

Overview A JunctionNode represents a pseudo-activity which is used for branching and merging the control flow in an activity. It is visualized by a diamond shaped figure.

Parent Classes
- ActivityNode see Section A.17.2.5 on Page 281

A.17.2.11. Class MatchingStoryNode

Overview A MatchingStoryNode may only contain a MatchingPattern which does not change the graph. I.e., no element contained in this activity carries a create or destroy annotation. Thus, after executing a MatchingStoryNode, the underlying graph is guaranteed to be unchanged.

Class References Class MatchingStoryNode has the following references:

- ownedPattern : MatchingPattern see Section A.21.2.13 on Page 303
  This MatchingPattern contained in this activity.

Parent Classes
- StoryNode see Section A.17.2.15 on Page 286

A.17.2.12. Class ModifyingStoryNode

Overview A ModifyingStoryNode contains a story pattern which may change the underlying graph upon execution.

Class References Class ModifyingStoryNode has the following references:

- ownedRule : StoryPattern see Section A.21.2.18 on Page 305
  The story pattern contained in this ModifyingStoryNode.
A.17. PACKAGE STORYDIAGRAMS::ACTIVITIES

Parent Classes

- StoryNode see Section A.17.2.15 on Page 286

A.17.2.13. Class OperationExtension

Overview  An OperationExtension is a stand-in for an EOperation in our model. It is necessary because we cannot change the type EOperation. Thus, OperationExtension points to an EOperation but adds the reference to an Activity that describes the operations behavior.

Class References  Class OperationExtension has the following references:

- operation : EOperation [0..1]
  The EOperation whose behavior is defined by the Activity. The property is derived because the actual value is determined by the utility class OperationsExtensionOperation.

- ownedActivity : Activity [0..1] see Section A.17.2.1 on Page 278
  The Activity to which the operation belongs.

- returnValue : EParameter [0..1]
  The return value of the referenced operation.

Parent Classes

- Extension see Section A.13.2.3 on Page 269,
- Callable see Section A.19.2.1 on Page 289

A.17.2.14. Class StatementNode

Overview  A statement node is a node that just contains an expression defining its behavior. In combination with a textual expression, arbitrary source code might be added by using StatementNodes.

Class References  Class StatementNode has the following references:

- statementExpression : Expression  see Section A.14.2.1 on Page 270
  The expression which defines the behavior of this StatementNode.

Parent Classes

- ActivityNode see Section A.17.2.5 on Page 281
A.17.2.15. **Class StoryNode**  

**Overview** An activity node containing a story pattern.

**Class Properties** Class StoryNode has the following properties:

- **forEach : EBoolean**  
  Specifies whether just one match should be found for the contained pattern (forEach = false) or whether all matches should be found (forEach = true).

**Class References** Class StoryNode has the following references:

- **/storyPattern : StoryPattern** see Section A.21.2.18 on Page 305

**Parent Classes**

- ActivityNode see Section A.17.2.5 on Page 281

---

A.17.2.16. **Class StructuredNode**  

**Overview** A structured node is a node that contains several other activities.

**Class References** Class StructuredNode has the following references:

- **ownedActivityNode : ActivityNode [0..*]** see Section A.17.2.5 on Page 281
  
  All subnodes which are contained in this structured node.

**Parent Classes**

- ActivityNode see Section A.17.2.5 on Page 281
A.18. Package

storydiagrams::activities::expressions

A.18.1. Package Overview

Figure A.16.: Metamodel of the activities::expressions Package

A.18.2. Detailed Contents Documentation

A.18.2.1. Class ExceptionVariableExpression

Overview  Represents the value of an exception variable declared as a transition guard (the guard of an activity edge).

Class References  Class ExceptionVariableExpression has the following references:

exceptionVariable : ExceptionVariable  see Section A.17.2.7 on Page 283

Specifies the exception variable that this expression refers to. If you have an activity edge that catches an exception e, then this expression can represent the reference e.
Parent Classes

- Expression see Section A.14.2.1 on Page 270
A.19. Package storydiagrams::calls

A.19.1. Package Overview

This package contains all classes for modeling calls to activities and EOperations from within an activity.

A.19.2. Detailed Contents Documentation

A.19.2.1. Class Callable

Overview

An entity which can be called by an Invocation. A Callable can have a number of (ordered) parameters which are either in or out parameters. In the case of activities, the number of in and out parameters is unbounded, whereas OperationExtensions and OpaqueCallables can only have one out parameter (This is enforced by an OCL constraint).

Class References

Class Callable has the following references:

- containedParameters : EParameter [0..*]
  
  This reference is used to contain the parameters of a Callable if they are not already contained in another container. If the parameter is contained in another container as it is the case for parameters of a EOperation, they must not be added to this container!

- inParameter : EParameter [0..*]
  
  The ordered set of in parameters of this Callable. The parameters will not be contained in this reference, if parameters have to be contained in the callable, they also have to be added to the containedParameters reference.

- outParameter : EParameter [0..*]
  
  The ordered set of out parameters of this Callable. The parameters will not be contained in this reference, if parameters have to be contained in the callable, they also have to be added to the containedParameters reference.

Parent Classes

- CommentableElement see Section A.13.2.1 on Page 267

A.19.2.2. Class Invocation

Overview

Superclass for invocations of behavior which is specified elsewhere, e.g. in methods (MethodCallExpression) or activities (ActivityCallNode). An invocation has one parameter binding for each parameter (in or out) of the called method/activity. For Callables...
Figure A.17.: Metamodel of the calls Package
which are contained in the model (i.e. Activities and OperationExtensions) the Invocation di-
rectly points to the callee. OpaqueCallable are directly referenced by (and contained in) the
MethodCallExpressions.

**Class References** Class Invocation has the following references:

- callee : Callable [0..1] see Section A.19.2.1 on Page 289
- ownedParameterBindings : ParameterBinding [0..*] see Section A.19.2.4 on Page 291

**Parent Classes**

- CommentableElement see Section A.13.2.1 on Page 267

**A.19.2.3. Class OpaqueCallable**

**Overview** An OpaqueCallable represents an external method which is not explicitly mod-
eled (e.g. a method in an external library). Because it is not contained anywhere in the model
it is directly referenced by and contained in the MethodCallExpression.

**Class Properties** Class OpaqueCallable has the following properties:

- name : EString

**Class References** Class OpaqueCallable has the following references:

- callExpression : MethodCallExpression see Section A.20.2.1 on Page 293

**Parent Classes**

- Callable see Section A.19.2.1 on Page 289

**A.19.2.4. Class ParameterBinding**

**Overview** Binds a parameter to a certain value for a given invocation. The value of the
parameter is represented by an expression.

**Class References** Class ParameterBinding has the following references:

- invocation : Invocation see Section A.19.2.2 on Page 289
- parameter : EParameter [0..1]
- valueExpression : Expression see Section A.14.2.1 on Page 270
Parent Classes

- CommentableElement see Section A.13.2.1 on Page 267

A.19.2.5. Class ParameterExtension

Overview Represented an EParameter and adds functionality to it, especially being subtype of Variable.

Class References Class ParameterExtension has the following references:

/parameter : EParameter [0..1]

Parent Classes

- Variable see Section A.16.2.1 on Page 277,
- Extension see Section A.13.2.3 on Page 269
A.20. Package

storydiagrams::calls::expressions

A.20.1. Package Overview

A.20.2. Detailed Contents Documentation

A.20.2.1. Class MethodCallExpression

**Overview** A MethodCallExpression represents the direct invocation of a method. This can either be a method which is explicitly modeled as an EOperation in a class diagram (referenced by the OperationExtension) or an unmodeled method in an external library (referenced by an OpaqueCallable). Therefore, a MethodCallExpression references either an OperationExtension (indirectly via the callee role between Invocation and Callable) or an OpaqueCallable.

**Class References** Class MethodCallExpression has the following references:

- **opaqueCallable : OpaqueCallable [0..1]** see Section A.19.2.3 on Page 291
  
  This containment reference is a helper construct because the OpaqueCallable has to be contained somewhere. A MethodCallExpression (being an Invocation) could also reference an OpaqueCallable (being a Callable) via the callee reference but then the OpaqueCallable would not be contained anywhere in the model.

- **target : Expression [0..1]** see Section A.14.2.1 on Page 270
  
  A MethodCallExpression references an expression instead of a target object to allow the determination of the call target by an expression. This can be handy when a method should be called e.g. on the return value of another method (as in a.b().c() ). Then the method call of c() would be modeled by a MethodCallExpression with the callExpression a.b(), which again is a MethodCallExpression itself.

**Parent Classes**

- Expression see Section A.14.2.1 on Page 270,

- Invocation see Section A.19.2.2 on Page 289

A.20.2.2. Class ParameterExpression

**Overview** An Expressions that represents a parameter value, e.g. the value of an Activity’s parameter.

**Class References** Class ParameterExpression has the following references:

- **parameter : ParameterExtension [0..1]** see Section A.19.2.5 on Page 292
Figure A.18.: Metamodel of the calls::expressions Package
Parent Classes

- Expression see Section A.14.2.1 on Page 270
A.21. Package `storydiagrams::patterns`

A.21.1. Package Overview

This package contains all classes for modeling story patterns that may be embedded into StoryActivityNodes of an Activity.

A.21.2. Detailed Contents Documentation

A.21.2.1. Class `AbstractLinkVariable`

Overview Abstract super class for all kinds of link variables that represent links between two objects in a story pattern.

Class Properties Class `AbstractLinkVariable` has the following properties:

- `bindingOperator : BindingOperator` see Section A.21.2.4 on Page 299
  The binding operator defines whether this link will be matched, created or destroyed by the story pattern. The default value is "check_only", i.e., the link will be matched.

- `bindingSemantics : BindingSemantics` see Section A.21.2.5 on Page 299
  The binding semantics defines whether the link must be matched for a successful application of the containing story pattern, whether it must not be matched or whether it is optional, i.e., it will be bound if it can be bound but that does not affect the success of matching the story pattern. The default value is "mandatory" (i.e., it must be matched).

Class References Class `AbstractLinkVariable` has the following references:

- `firstLinkConstraint : LinkConstraint [0..+]` see Section A.21.2.10 on Page 301
- `pattern : StoryPattern` see Section A.21.2.18 on Page 305
- `secondLinkConstraint : LinkConstraint [0..+]` see Section A.21.2.10 on Page 301
- `source : ObjectVariable` see Section A.21.2.15 on Page 304
- `target : AbstractVariable` see Section A.21.2.2 on Page 298

Parent Classes

- NamedElement see Section A.13.2.4 on Page 269
Figure A.19.: Metamodel of the patterns Package
A.21.2.2. Class AbstractVariable

Overview  Abstract super class for object and primitive variables.

Class Properties  Class AbstractVariable has the following properties:

bindingState : BindingState  see Section A.21.2.6 on Page 300

The binding state defines whether the variable is already bound or whether a match
has to be obtained for it. The default value is "unbound".

Class References  Class AbstractVariable has the following references:

bindingExpression : Expression [0..1]  see Section A.14.2.1 on Page 270

A binding expression can be used to bind a variable in a different way than just by
pattern matching. This way, for example, the return value of a call can be bound
to a variable.

constraint : Constraint [0..*]  see Section A.21.2.8 on Page 301

All constraints which are defined for this variable. For a successful matching, all
constraints for this variable must evaluate to true.

incomingLink : AbstractLinkVariable [0..*]  see Section A.21.2.1 on Page 296

pattern : StoryPattern  see Section A.21.2.18 on Page 305

Parent Classes  
- Variable see Section A.16.2.1 on Page 277,
- NamedElement see Section A.13.2.4 on Page 269

A.21.2.3. Class AttributeAssignment

Overview  An AttributeAssignment is used to set the value of a certain attribute of an object.
It references the attribute that is to be set and the value. The value can be an expression to
allow for calculations or calls that determine the final value. AttributeAssignments are carried
out during the final phase of pattern application, i.e. after the matching and destruction are
completed.

Class References  Class AttributeAssignment has the following references:

attribute : EAttribute

The attribute whose value is set. It has to be an attribute of the objectVariable that
contains the AttributeAssignment.

objectVariable : ObjectVariable  see Section A.21.2.15 on Page 304

valueExpression : Expression  see Section A.14.2.1 on Page 270

The expression that determines the new value that is given to the attribute.
A.21.2.4. Enumeration BindingOperator

Overview The BindingOperator enum defines all possible operations for object and link variables. An object or link variable may be checked for existence by the story pattern (black object/link variable), it may be created (green object/link variable), or it may be destroyed (red object/link variable).

Enum Properties Enumeration BindingOperator has the following literals:

- **CHECK ONLY = 0**
  - CHECK ONLY is the default value of this enum. It requires an object or link variable just to be matched by the story pattern.
- **CREATE = 1**
  - An object or link variable marked as CREATE will be created by the story pattern.
- **DESTROY = 2**
  - An object or link variable marked as DESTROY will be destroyed by the story pattern.

A.21.2.5. Enumeration BindingSemantics

Overview The binding semantics defines which kind of match will be obtained for the object or link variable.

Enum Properties Enumeration BindingSemantics has the following literals:

- **MANDATORY = 0**
  - For a mandatory object or link variable, a match has to be found for a pattern to be successfully applied.
- **NEGATIVE = 1**
  - If an object or link variable is marked as NEGATIVE, no match may be found for that object or link variable. If a match can be found, the execution of the story pattern fails.
- **OPTIONAL = 2**
  - For an OPTIONAL object or link variable, the matching tries to find a match. If no match can be found, this does not affect the success of the pattern application. If a match can be found, the respective object or link is bound to the variable.
A.21.2.6. **Enumeration BindingState**

**Overview**  The BindingState defines whether an object or link variable is already bound to a concrete value or not.

**Enum Properties**  Enumeration BindingState has the following literals:

- **UNBOUND = 0**
  UNBOUND is the default value for this enum. If an object or link variable in a story pattern is unbound, a new match has to be obtained for that variable.

- **BOUND = 1**
  A bound variable has already been bound to a concrete value. The concrete value has to be passed either as a parameter or it has to be bound in a previous activity. If, during the execution of a story pattern, a bound variable has no value, the execution of the story pattern fails.

- **MAYBE_BOUND = 2**
  A variable marked with maybe_bound indicates that it is unknown (or unimportant) at design time whether the variable is bound or not. If, during the execution of the pattern, the variable is not bound, an object is matched and bound to the variable. If it is already bound, it is not altered. If the variable is still unbound after this process, the matching fails (except for OPTIONAL variables).

A.21.2.7. **Class CollectionVariable**

**Overview**  Represents a set of objects of the same type that are represented by a single node. The context for contained Constraints and AttributeAssignments is every single object in the set. E.g., if the constraint is "name = 'abc'", only objects with that name are matched and added to the set. The use of the binding operator "CREATE" is not defined for ObjectSetVariables, i.e., the sets can only be matched and deleted.

**Class Properties**  Class CollectionVariable has the following properties:

- atLeastOne : EBoolean
- unique : EBoolean

**Parent Classes**
- ObjectVariable see Section A.21.2.15 on Page 304
A.21.2.8. Class Constraint

Overview  A constraint represents a condition which must be fulfilled for a successful pattern matching. It can either be contained in the story pattern or in a variable. In the former case, the constraint is evaluated after the matching of the object structure is complete. It still has to be true for the pattern application to be successful (and therefore for creations and destructions to be carried out). If the constraint is contained in a variable, it constrains the matching of that variable, i.e., it is evaluated during the matching of the containing variable and has to be true for a successful matching. If the variable is an ObjectSetVariable, the constraint has to be true for every object in the set.

Class References  Class Constraint has the following references:

- constraintExpression : Expression  see Section A.14.2.1 on Page 270
  The constraintExpression defines the concrete condition of this constraint.
- objectVariable : AbstractVariable [0..1]  see Section A.21.2.2 on Page 298
  The object variable this constraint applies to.
- pattern : StoryPattern [0..1]  see Section A.21.2.18 on Page 305
  The story pattern this constraint applies to.

A.21.2.9. Class InclusionLink

Overview  Specifies the containment of an object in a set (represented by a ContainerVariable). Will be displayed by a line having a circle with a plus inside at the end of the container (the source end of the link). A create modifier specifies that the object will be added to the container, delete that it will be removed, and none that it will be checked to be contained.

Parent Classes

- AbstractLinkVariable see Section A.21.2.1 on Page 296

A.21.2.10. Class LinkConstraint

Overview  Link constraints (formerly known as MultiLinks in old meta-model) constrain the ordering of links of the referencingObject is a collection. This way objects can be required to have a certain position in the collection (FIRST, LAST, INDEX) or a certain ordering relative to each other (DIRECT_SUCCESSOR, INDIRECT_SUCCESSOR). While the first kind of LinkConstraint can be imposed upon a single link, the second kind requires two links that are related to each other (e.g., have the same referencingObject).

Class Properties  Class LinkConstraint has the following properties:
constraintType : LinkConstraintType  see Section A.21.2.11 on Page 302
    The constraint type of the LinkConstraint.

index : EInt
    The index of the linked object in the collection. The semantics of this attribute is
    only defined if the constraintType of the LinkConstraint is INDEX.

negative : EBoolean
    If the negative attribute is true, the link constraint may not be fulfilled for the
    complete pattern application to be successful.

Class References  Class LinkConstraint has the following references:
    firstLink : AbstractLinkVariable  see Section A.21.2.1 on Page 296
    referencingObject : ObjectVariable  see Section A.21.2.15 on Page 304
    secondLink : AbstractLinkVariable [0..1]  see Section A.21.2.1 on Page 296

Parent Classes
    • ExtendableElement see Section A.13.2.2 on Page 267

A.21.2.11. Enumeration LinkConstraintType

Overview  The LinkConstraintType represents the different uses of LinkConstraints. Ob-
    jects can be required to have a certain position in their containing collection (FIRST,
    LAST, INDEX) or a certain ordering relative to each other (DIRECT_SUCCESSOR, INDI-
    RECT_SUCCESSOR).

Enum Properties  Enumeration LinkConstraintType has the following literals:
    FIRST = 0
    LAST = 1
    NEXT = 2
    INDIRECT_SUCCESSOR = 3
    INDEX = 4

A.21.2.12. Class LinkVariable

Overview  A link variable represents one link between two object variables. It is typed over
    one of the associations between the classes of those objects. Because EMF only directly sup-
    ports references, the two link ends are typed over these references. In case of a uni-directional
    association, only the targetEnd is typed. In case of a bi-directional association, the reference
    that types the source end is automatically determined.
Class References  Class `LinkVariable` has the following references:

- **qualifierExpression : Expression [0..1]** see Section A.14.2.1 on Page 270
  
  If a link is typed over a qualified reference, a qualifier determines the key under which the object reachable via the link is stored. Because the qualifier can be set by an expression, it can either be a simple string or something more complex, e.g., a call like "object.getName()".

- **sourceEnd : EReference [0..1]**
  
  The source end of a link variable can only be determined when the link is typed over a bi-directional association. In this case, it points to the "reverse" direction of the association. If the reference is only uni-directional, the source end is null. The value of this attribute is derived automatically.

- **targetEnd : EReference**
  
  The target end points to the reference that types this direction of the link (the "normal" direction). This link end must be set always.

Parent Classes

- AbstractLinkVariable see Section A.21.2.1 on Page 296

A.21.2.13. Class `MatchingPattern`

Overview  A `MatchingPattern` is a special kind of story pattern that does not change the underlying graph. Thus, no contained object or link may carry an create or destroy BindingOperator.

Parent Classes

- StoryPattern see Section A.21.2.18 on Page 305

A.21.2.14. Class `MaybeLink`

Overview

Parent Classes

- AbstractLinkVariable see Section A.21.2.1 on Page 296
A.21.2.15. **Class ObjectVariable**

**Overview**  An ObjectVariable holds a value of a complex type which is defined by an EClass.

**Class Properties**  Class ObjectVariable has the following properties:

- **bindingOperator : BindingOperator**  see Section A.21.2.4 on Page 299
  
  The binding operator defines whether this object will be matched, created or destroyed by the story pattern.

- **bindingSemantics : BindingSemantics**  see Section A.21.2.5 on Page 299
  
  The binding semantics defines whether the object must be matched for a successful application of the containing story pattern, whether it must not be matched or whether it is optional, i.e., it will be bound if it can be bound but that does not affect the success of matching the story pattern.

**Class References**  Class ObjectVariable has the following references:

- **attributeAssignment : AttributeAssignment [0..*]**  see Section A.21.2.3 on Page 298
  
  classifier : EClass
  
  The type of this ObjectVariable, given as an EClass.

- **linkOrderConstraint : LinkConstraint [0..*]**  see Section A.21.2.10 on Page 301

- **outgoingLink : AbstractLinkVariable [0..*]**  see Section A.21.2.1 on Page 296

**Parent Classes**

- AbstractVariable see Section A.21.2.2 on Page 298

A.21.2.16. **Class Path**

**Overview**  A path is a special link variable that specifies an indirect connection between two objects. That means, the connected objects have other links and objects "between them". Exactly which types of links may be traversed during the matching of a path can be constrained by a path expression.

**Class References**  Class Path has the following references:

- **pathExpression : Expression**  see Section A.14.2.1 on Page 270
  
  The path expression constrains the matching of the path variable during pattern application. It can determine which links may be matched when and how many times to reach the target object of the path from the source object.
Parent Classes

- AbstractLinkVariable see Section A.21.2.1 on Page 296

A.21.2.17. Class PrimitiveVariable

Overview  Represents a variable that holds a value of a primitive type, e.g. integer, boolean, String.

Class References  Class PrimitiveVariable has the following references:

  classifier : EDataType
  The type of the primitive variable which must be an EDataType.

Parent Classes

- AbstractVariable see Section A.21.2.2 on Page 298

A.21.2.18. Class StoryPattern

Overview  A Story Pattern is a graph rewrite rule that may be embedded into a StoryActivityNode of an Activity.

Class Properties  Class StoryPattern has the following properties:

  bindingSemantics : BindingSemantics  see Section A.21.2.5 on Page 299

Class References  Class StoryPattern has the following references:

  constraint : Constraint [0..*]  see Section A.21.2.8 on Page 301
  All constraints which are defined for this story pattern. For a successful matching, all constraints for this story pattern must evaluate to true.

  containedPattern : StoryPattern [0..*]  see Section A.21.2.18 on Page 305

  linkVariable : AbstractLinkVariable [0..*]  see Section A.21.2.1 on Page 296

  parentPattern : StoryPattern [0..1]  see Section A.21.2.18 on Page 305

  templateSignature : TemplateSignature [0..1]  see Section A.23.2.3 on Page 310

  variable : AbstractVariable [0..*]  see Section A.21.2.2 on Page 298

Parent Classes

- CommentableElement see Section A.13.2.1 on Page 267
A.22. Package

storydiagrams::patterns::expressions

A.22.1. Package Overview

A.22.2. Detailed Contents Documentation

A.22.2.1. Class AttributeValueExpression

Overview  Represents the value of an object’s attribute, e.g. obj.attr for an object obj and an attribute attr.

Class References  Class AttributeValueExpression has the following references:

- attribute : EAttribute
  Specifies the object’s attribute whose attribute value is represented by this expression.

- object : ObjectVariable  see Section A.21.2.15 on Page 304
  Specifies the object variable whose attribute value is represented by this expression.

Parent Classes

- Expression see Section A.14.2.1 on Page 270

A.22.2.2. Class CollectionSizeExpression

Overview  Represents the number of elements in the set of objects that is represented by an object set variable. For example, if you have an object set variable mySet, then this expression would represent something like mySet.size(). The expression can be used to constrain the pattern application, e.g., to only a apply the pattern when at least two objects can be matched for the set.

Class References  Class CollectionSizeExpression has the following references:

- set : CollectionVariable  see Section A.21.2.7 on Page 300
  Specifies the object set variable whose number of set elements is to be represented by this expression.

Parent Classes

- Expression see Section A.14.2.1 on Page 270
Figure A.20.: Metamodel of the patterns::expressions Package
A.22.2.3. **Class ObjectVariableExpression**

**Overview** Represent the reference to an object in an expression, i.e. the value of an object variable.

**Class References** Class `ObjectVariableExpression` has the following references:

object : `ObjectVariable` see Section A.21.2.15 on Page 304

Specifies the object variable that holds the reference to be represented by this expression.

**Parent Classes**

- Expression see Section A.14.2.1 on Page 270

A.22.2.4. **Class PrimitiveVariableExpression**

**Overview** Represents the value of a primitive variable, e.g., 5 or "MyName".

**Class References** Class `PrimitiveVariableExpression` has the following references:

primitiveVariable : `PrimitiveVariable` see Section A.21.2.17 on Page 305

**Parent Classes**

- Expression see Section A.14.2.1 on Page 270
A.23. Package storydiagrams::templates

A.23.1. Package Overview

A.23.2. Detailed Contents Documentation

A.23.2.1. Class PropertyBinding

Overview

Class References Class PropertyBinding has the following references:

- bindingExpression : Expression  see Section A.14.2.1 on Page 270
- boundProperty : EStructuralFeature
- templateBinding : TemplateBinding  see Section A.23.2.2 on Page 310

Parent Classes

- ExtendableElement see Section A.13.2.2 on Page 267
A.23.2.2. **Class TemplateBinding**

**Overview**

**Class References** Class TemplateBinding has the following references:

- bindingExpression : Expression  see Section A.14.2.1 on Page 270
- boundParameter : EClassifier
- propertyBinding : PropertyBinding [0..*]  see Section A.23.2.1 on Page 309
- template : TemplateSignature  see Section A.23.2.3 on Page 310

**Parent Classes**

- ExtendableElement see Section A.13.2.2 on Page 267

A.23.2.3. **Class TemplateSignature**

**Overview**

**Class References** Class TemplateSignature has the following references:

- pattern : StoryPattern  see Section A.21.2.18 on Page 305
- templateBinding : TemplateBinding [0..*]  see Section A.23.2.2 on Page 310
- typeParameter : EClassifier [0..*]
Appendix B.

Action Language XText Grammar

grammar de.uni_paderborn.fujaba.muml.ActionLanguage with org.eclipse.xtext.common.Terminals

import "platform://resource/de.uni_paderborn.fujaba.muml.model/actionLanguage/model/actionLanguage.ecore" as actionLanguage
import "http://www.eclipse.org/emf/2002/Ecore" as.ecore
import "platform://resource/org.storydriven.core/model/core.ecore" as modeling
import "platform://resource/de.uni_paderborn.fujaba.muml.model/model/muml.ecore#/model/realtimestatechart" as rtsc
import "platform://resource/de.uni_paderborn.fujaba.muml.model/model/muml.ecore#/model/core.ecore" as core
import "platform://resource/org.storydriven.core/model/core.ecore#/expressions" as expressions
import "platform://resource/org.storydriven.core/model/core.ecore#/expressions/common" as commonExpressions

Block returns actionLanguage::Block hidden (WS, ML_COMMENT, SL_COMMENT):
    {actionLanguage::Block}
    "{" expressions+=ExpressionStartRule*)
    "}" expressions+=ArithmeticExpression ;

ForLoop returns actionLanguage::ForLoop:
    "for" "(" initializeExpression=Assignment loopTest=Expression ;" countingExpression=ForLoopCountingExpression ")" block=Block

WhileLoop returns actionLanguage::WhileLoop:
DoWhileLoop returns actionLanguage::DoWhileLoop:
  'do'
  block=Block
  'while' '(' loopTest=Expression ');

DoWhileLoop returns actionLanguage::DoWhileLoop:
  'do'
  block=Block
  'while' '(' loopTest=Expression ');

IfStatement returns actionLanguage::IfStatement:
  'if' '(' ifCondition=Expression ')
  ifBlock=Block
  ('elseif' '(' elseIfConditions += Expression ')' elseIfBlocks += Block) *
  ('else' elseBlock=Block) ?

ReturnStatement returns actionLanguage::ReturnStatement:
  { actionLanguage::ReturnStatement }
  'return' expression=Expression ';

ExpressionStartRule returns expressions::Expression:
  Assignment | ForLoop | WhileLoop | DoWhileLoop | IfStatement | ReturnStatement

Assignment returns actionLanguage::Assignment:
  ( lhs_attributeExpression=AttributeLeafExpression
  ( assignOperator=AssignOperator rhs_assignExpression=Expression ) | incrementDecrementOperator=IncrementDecrementOperator ) ';

ForLoopCountingExpression returns actionLanguage::Assignment:
  ( lhs_attributeExpression=AttributeLeafExpression
  ( incrementDecrementOperator=IncrementDecrementOperator | ( assignOperator =AssignOperator rhs_assignExpression=Expression ) ) )

Expression returns expressions::Expression:
  ArithmeticExpression

ArithmeticExpression returns expressions::Expression:
  ComparisonExpression ( ( ( commonExpressions::ArithmeticExpression .
    leftExpression=current ) ) operator=ArithmeticOperator
  rightExpression=ComparisonExpression ) *
ComparisonExpression returns expressions :: Expression:
   LogicalExpression
   ( ( { commonExpressions :: ComparisonExpression . leftExpression = current } 
       operator = ComparingOperator ) rightExpression = LogicalExpression ) * 
;
LogicalExpression returns expressions :: Expression:
   ( UnaryExpression | AttributeExpression ) 
   ( ( { commonExpressions :: LogicalExpression . leftExpression = current } 
       operator = LogicOperator ) rightExpression = ( UnaryExpression | 
       AttributeExpression ) ) * 
;
UnaryExpression returns expressions :: Expression:
   { commonExpressions :: UnaryExpression } 
   operator = UnaryOperator enclosedExpression = AttributeExpression 
;
AttributeExpression returns expressions :: Expression:
   OperationCall | AttributeLeafExpression 
;
AttributeLeafExpression returns actionLanguage :: AttributeExpression:
   { actionLanguage :: AttributeExpression } 
   attribute = [ core :: Attribute ] ( '\indices += Expression ' ) * 
;
OperationCall returns expressions :: Expression:
   { actionLanguage :: OperationCall } operation = [ core :: Operation ] ( '\( parameterBinding += ParameterBinding ( ' , ' parameterBinding += ParameterBinding ) * ) ' ) * 
;
ParameterBinding returns core :: ParameterBinding:
   { core :: ParameterBinding } parameter = [ core :: Parameter ] ( ': = ' value = Expression ) 
;
LiteralExpression returns expressions :: Expression:
   ( ' Expression ' ) | 
   { commonExpressions :: LiteralExpression } value = IdentifierOrValue 
;
IdentifierOrValue returns ecore :: EString:
   NUMBER | BOOLEAN | INT 
;
enum AssignOperator returns actionLanguage :: AssignOperator:

APPENDIX B. ACTION LANGUAGE XTEXT GRAMMAR

 ASSIGN=':= ' | PLUS_EQUAL='+= ' | EQUAL_PLUS='==+ ' | MINUS_EQUAL='==−' |
   EQUAL_MINUS='−== ' ;

 enum IncrementDecrementOperator returns actionLanguage ::
   IncrementDecrementOperator :
     INCREMENT='++ ' | DECREMENT='−− ' ;

 enum LogicOperator returns commonExpressions :: LogicOperator :
   AND='& ' | OR='| ' | XOR='xor ' | IMPLY='=> ' | EQUIVALENT='<=> ' ;

 enum ArithmeticOperator returns commonExpressions :: ArithmeticOperator :
   PLUS='+' | MINUS='− ' | TIMES='∗ ' | DIVIDE='/' | MODULO='% ' ;

 enum ComparingOperator returns commonExpressions :: ComparingOperator :
   LESS='<' | LESS_OR_EQUAL='<= ' | EQUAL='== ' | GREATER_OR_EQUAL='>= ' |
     GREATER='>' | UNEQUAL='<> ' ;

 enum UnaryOperator returns commonExpressions :: UnaryOperator :
   NOT='not ' | MINUS='− ' | PLUS='+' ;

terminal NUMBER returns ecore :: EBigDecimal :
   ' INT'. ' INT ;

terminal BOOLEAN returns ecore :: EBoolean :
   ' true ' | ' false ' ;