The MechatronicUML Method –
Process, Syntax, and Semantics

Technical Report
tr-ri-12-318

Steffen Becker, Christian Brenner, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann, Claudia Priesterjahn, Wilhelm Schäfer, Julian Suck, Oliver Sudmann, Matthias Tichy
Software Engineering Group, Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
[stbecker|cbr|xell|cytom|c.heinzemann|upohl|c.priesterjahn]
wilhelm|julian.suck|oliversu|mtt]@uni-paderborn.de

Version: 0.2

Paderborn, March 7, 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**  
   1.1. Example .................................................. 3

2. **MECHATRONIC-UML Overview**  

3. **Modeling Languages**  
   3.1. Real-Time Coordination Pattern  
       3.1.1. Application Example  
       3.1.2. Role  
       3.1.3. Role Connector  
       3.1.4. Kinds of Real-Time Coordination Patterns  
       3.1.5. Real-Time Coordination Instance  
       3.1.6. Behavior Specification  
       3.1.7. Verification of Properties  
   3.2. Message Interface Specification  
       3.2.1. Message Interface  
       3.2.2. Message Type  
   3.3. Real-Time Statechart  
       3.3.1. Clock  
       3.3.2. State  
       3.3.3. Region  
       3.3.4. Transition  
       3.3.5. Entry-/Exit-Point  
       3.3.6. Shallow History  
       3.3.7. Action  
       3.3.8. Synchronization  
       3.3.9. Asynchronous Message-Event  
   3.4. Component Model  
       3.4.1. Atomic Component Type  
       3.4.2. Structured Component Type  
   3.5. Component Instance Configuration  
       3.5.1. Component Instance  
       3.5.2. Component Instance Configuration
CONTENTS

3.6. Deployment ................................................. 65
  3.6.1. Hardware Nodes ...................................... 65
  3.6.2. Allocation ........................................... 65

4. Development Process ........................................ 67
  4.1. Integration into the Development Process for Advanced Mechatronic Systems 67
  4.2. The MECHATRONIC UML Process ......................... 69
    4.2.1. Determination of Coordination Patterns (Step 3) .......... 73
    4.2.2. Modeling of a Real-Time Coordination Pattern (Step 3.2) .... 73
    4.2.3. Determination of the Components’ Behavior (Step 4) ........ 76
    4.2.4. Modeling of the Components’ Behavior (Step 4.3) .......... 78

5. Complete Example ............................................ 81
  5.1. Real-Time Coordination Patterns ....................... 81
    5.1.1. Navigation Pattern ................................ 81
    5.1.2. Delegation Pattern ................................ 83
    5.1.3. Distribution Pattern .............................. 84
    5.1.4. PositionTransmission Pattern ...................... 87
    5.1.5. DistancesTransmission Pattern .................... 88
  5.2. Message Interface Specification ...................... 91
    5.2.1. Navigation Pattern ................................ 91
    5.2.2. Delegation Pattern ................................ 91
    5.2.3. Distribution Pattern .............................. 92
    5.2.4. PositionTransmission Pattern ...................... 92
    5.2.5. DistancesTransmission Pattern .................... 93
  5.3. System Structure ....................................... 93
    5.3.1. Structured Components ............................ 93
    5.3.2. Atomic Components ............................... 96
  5.4. Real-Time Statechart .................................. 99
    5.4.1. Exploration ....................................... 99
    5.4.2. Navigation ........................................ 101
    5.4.3. BeBot Observer .................................. 103
    5.4.4. Collision Control ................................ 105
  5.5. Component Instance Configuration .................... 108
    5.5.1. Single BeBot ..................................... 108
    5.5.2. Networks of BeBots ............................... 109
  5.6. Deployment ............................................. 114

6. Theoretical Background ...................................... 115
  6.1. Compositional Verification Theorem ................... 115
    6.1.1. Preliminaries .................................... 115
    6.1.2. Putting it all together ........................... 117
7. Related Work

7.1. Specification Languages for Systems Engineering

7.1.1. SysML

7.1.2. MATLAB/Simulink Stateflow

7.1.3. Modelica

7.2. Process Models for Systems Engineering

7.3. Software Component Models

7.4. Specifications of Self-Adaptive Systems

7.4.1. Graph Transformation based Approaches

7.4.2. Process Algebra based Approaches

7.4.3. Formal Logic based Approaches

7.4.4. Other Approaches

7.4.5. Run-Time Frameworks for Self-Adaptation

7.5. Formal Models for Modeling of Real-time Behavior

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. Technical Reference

A.1. Package modelinstance

A.1.1. Package Overview

A.1.2. Detailed Contents Documentation

A.2. Package muml

A.2.1. Package Overview

A.3. Package muml::model

A.3.1. Package Overview

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

A.4.1. Package Overview

A.4.2. Detailed Contents Documentation

A.5. Package muml::model::constraint

A.5.1. Package Overview

A.5.2. Detailed Contents Documentation

A.6. Package muml::model::core

A.6.1. Package Overview

A.6.2. Detailed Contents Documentation

A.7. Package muml::model::deployment

A.7.1. Package Overview

A.7.2. Detailed Contents Documentation
A.8. Package `muml::model::instance` ........................................ 170
   A.8.1. Package Overview ............................................. 170
   A.8.2. Detailed Contents Documentation ............................. 170
A.9. Package `muml::model::msgiface` .................................... 175
   A.9.1. Package Overview ............................................. 175
   A.9.2. Detailed Contents Documentation ............................. 175
A.10. Package `muml::model::pattern` ..................................... 177
   A.10.1. Package Overview ............................................. 177
   A.10.2. Detailed Contents Documentation ............................. 177
A.11. Package `muml::model::realtimestatechart` ....................... 181
   A.11.1. Package Overview ............................................. 181
   A.11.2. Detailed Contents Documentation ............................. 181
A.12. Package `modeling` .................................................. 198
   A.12.1. Package Overview ............................................. 198
   A.12.2. Detailed Contents Documentation ............................. 198
A.13. Package `modeling::activities` ..................................... 201
   A.13.1. Package Overview ............................................. 201
   A.13.2. Detailed Contents Documentation ............................. 201
A.14. Package `modeling::activities::expressions` ...................... 209
   A.14.1. Package Overview ............................................. 209
   A.14.2. Detailed Contents Documentation ............................. 209
A.15. Package `modeling::calls` .......................................... 210
   A.15.1. Package Overview ............................................. 210
   A.15.2. Detailed Contents Documentation ............................. 210
A.16. Package `modeling::calls::expressions` ........................... 213
   A.16.1. Package Overview ............................................. 213
   A.16.2. Detailed Contents Documentation ............................. 213
A.17. Package `modeling::expressions` ................................... 214
   A.17.1. Package Overview ............................................. 214
   A.17.2. Detailed Contents Documentation ............................. 214
A.18. Package `modeling::patterns` ....................................... 219
   A.18.1. Package Overview ............................................. 219
   A.18.2. Detailed Contents Documentation ............................. 219
A.19. Package `modeling::patterns::expressions` ....................... 228
   A.19.1. Package Overview ............................................. 228
   A.19.2. Detailed Contents Documentation ............................. 228
A.20. Package `modeling::templates` ..................................... 229
   A.20.1. Package Overview ............................................. 229
   A.20.2. Detailed Contents Documentation ............................. 229
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 [CdLG09], 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
distinguishes component types as well as their instances\textsuperscript{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 \textsc{MechatronicUML} 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, \textsc{MechatronicUML} 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 \textsc{MechatronicUML} 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 \textsc{MechatronicUML}.

This technical report extends the previous version [BDG\textsuperscript{+}11]. It consolidates the older publications [Bur02, Gie03, GB03, GST\textsuperscript{+}03, GTB\textsuperscript{+}03, BGT05, GHH\textsuperscript{+}08, EHH\textsuperscript{+}11, SSGR11] and theses [Hir04, Bur06, Hir08] in a single document. We present the current version of \textsc{MechatronicUML} 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 \textsc{MechatronicUML} 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 \textsc{MechatronicUML} based on the running example which is presented in the next section. In Chapter 4, we illustrate the development process of \textsc{MechatronicUML}. 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 \textsc{MechatronicUML} 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.

\textsuperscript{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 Paderborn\(^3\).

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.

![BeBot](image)

Figure 1.1.: BeBot

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/
CHAPTER 1. INTRODUCTION

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.

MECHATRONICUML Overview

The MECHATRONICUML method enables the model-driven design of discrete software of self-adaptive mechatronic systems. The key concepts of MECHATRONICUML 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, MECHATRONICUML 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 MECHATRONICUML which we will briefly illustrate in the following. A detailed description of the development process can be found in Section 4.

![Figure 2.1.: Overview of the MECHATRONICUML Process](image)

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

In Step 1, MECHATRONICUML 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 syn-
thesis or simulation based approach [Gre11]. A detailed description of the use of Modal Sequence Diagrams in MECHATRONICUML 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], MECHATRONICUML 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. MECHATRONICUML 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 MECHATRONICUML. 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. MECHATRONICUML itself provides no behavior specification for controllers. Instead, we assume that the behavior is specified by a control engineering tool like CamellView 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. MECHATRONICUML 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 Steps 3, 4a, and 5a.

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, 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, MECHATRONICUML 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
CHAPTER 2. MECHATRONICUML OVERVIEW

Figure 2.2.: Overview of the Modeling Languages used in MECHATRONICUML
continuous components, MECHATRONICUML 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 different 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 *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 can not communicate with each other; instances of role client can not 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 3.3.9 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="image1" alt="out-role" /></td>
<td><img src="image2" alt="out-role" /></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>in-role</td>
<td><img src="image3" alt="in-role" /></td>
<td><img src="image4" alt="in-role" /></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>in/out-role</td>
<td><img src="image5" alt="in/out-role" /></td>
<td><img src="image6" 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 \textit{out-role}. A role that may only receive messages is an \textit{in-role}. A role that may send and receive messages is an \textit{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 \textit{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 \textit{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 \textit{multi-role}.
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. If 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.

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 \(i, j \in \mathbb{N} \land i \geq 1 \land j \geq 2 \land i \leq j\).

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.

3.1.4. 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.

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.
3.1.4.2. Forms of Communication

In MechatronicUML, two possible forms of communication exist: one-to-one and one-to-many. These forms of communication define to how many other instances one role can communicate 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 3.1 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 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 3.3.9) to specify the interaction between the roles. These message events are
3.1. REAL-TIME COORDINATION PATTERN

typed over the message types declared in the message interfaces of the role. 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 can not 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 can not directly call operations and side effects of other role instances.

Figure 3.5.: A Multi-Role and a Single-Role Specified by Real-Time Statecharts

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. However, a multi-role is defined by a Real-Time Statechart which has several restrictions; these will be explained in the following: Due to the one-to-many communication, 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 in one common sub-role behavior. A multi-role further defines an adaptation behavior to define an order of the different communications to each instance of the other role [EHH+11]. Therefore, the Real-Time Statechart of a multi-role contains only one state in the top level statechart. This state contains exactly two regions: a region named sub-role and a region named adaptation (cf. Figure 3.5). The region sub-role defines the behavior for the sub-role-instances of an instance of the multi-role. This region is parametrized by the parameter k, which is the unique identifier (e.g., an integer value) for each instance of the single-role. The region adaptation defines the adaptation behavior of the multi-role. It enables to
send information to a specific instance of the single-role or to send a multi-cast to all instances of the single-role. Furthermore, the adaptation behavior may determine in which order the single-roles are informed.

The region adaptation does not communicate directly with the instances of the single-role, but with the sub-role regions which mediate between the adaptation region and the instances of the single-role. In future releases of this document, this region will be further responsible for reconfiguring the multi-role [EHH + 11].

### 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.: The Real-Time Coordination Pattern Distribution with Properties for the Role distributor and the whole Pattern](image)

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 $A[] \neg \text{deadlock}$, which expresses that there exists no deadlock within this pattern. The second property states $\text{distributor.adaptation.Error} \rightarrow (\text{client}_1.\text{receive.Error} \land \ldots \land \text{client}_8.\text{receive.Error})$, 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 can not 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 adaption 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 MECHATRONIC UML.

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.

![Figure 3.7. Concrete Syntax of a Message Interfaces and an Inheritance Relation](image)

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).

### 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:

\[
\text{< messagetype > ::= \#MessageType . name ' ( ' [<parameterlist>] ' ) '}
\]

\[
\text{< parameterlist > ::= < parameter > | < parameter > , < parameterlist >}
\]

\[
\text{< parameter > ::= \#EParameter . name ':' \#EParameter . eType . name}
\]

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.
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 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 3.3.8 and 3.3.9.

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 model element for defining attributes of a statechart is called rtscDefinitionLabel.
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 (\texttt{rtscNameLabel}) shown in the upper left corner and the \texttt{rtscDefinitionLabel} in the upper right corner. The remainder of the rectangle may contain state compartments, the visual representation of states.

![Concrete Syntax Template of a Statechart](image)

The following EBNF expression defines the default notation of a \texttt{rtscDefinitionLabel}. Elements which have the prefix "\#" are references to the meta model elements of a statechart:

\[
\texttt{rtscDefinitionLabel} ::= \lfloor \texttt{operationDefinition} \rfloor \\
\quad \lfloor \texttt{varDefinition} \rfloor \\
\quad \lfloor \texttt{clockDefinition} \rfloor \\
\texttt{operationDefinition} ::= \texttt{op:} \#eClass.eOperations.eType \texttt{op:} \#eClass.eOperations.name \lfloor \texttt{var:} \#eClass.eOperations.eParameters \rfloor \\
\texttt{varDefinition} ::= \texttt{var:} \#eClass.eAttribute.eAttributeType \#eClass.eAttribute.name \\
\texttt{clockDefinition} ::= \texttt{cl:} \#clocks.name
\]

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.

![Concrete Example of a Statechart](image)
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 MECHATRONICUML 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\(^1\). 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 always exactly one state is 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 until 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 can not 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 can not pass anymore, because this would violate the invariant. Obviously, this behavior can not be implemented in a real system, and therefore has to be prevented. Time stopping deadlocks can be identified by means of formal verification. One possible way to resolve them is adding a transition to a fail-safe state which 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, on the other

---

\(^1\)In our recent papers composite states are also called complex states.
hand, will 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 synchronizations which are typed by channels (cf. Section 3.3.8). These channels belong to the orthogonal 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 compartment is visually represented by a StateActionLabel. A state’s channels are defined 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:

\[
\text{<StateActionLabel> ::= [<entryAction>]
[<doAction>]
[<exitAction>]
\]
\[
\text{<entryAction> ::= 'entry{' ( #entryEvent.action.expressions | #entryEvent.action.name ) '}'
[ 'reset:' #entryEvent.clockResets[ ', ' #entryEvent.clockResets]* '}' ]
\]
\[
\text{<doAction> ::= 'do{' ( #doEvent.action.expressions | #doEvent.action.name ) '}'
[ 'periodLower': #doEvent.periodLower, 'periodUpper': #doEvent.periodUpper ]
\]
\[
\text{<exitAction> ::= 'exit{' ( #exitEvent.action.expressions | #exitEvent.action.name ) '}'
[ 'reset:' #exitEvent.clockResets[ ', ' #exitEvent.clockResets]* '}' ]
\]

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:

\[
\text{<channelDefinitionLabel> ::= 'ch: [ #channels.name[ '(' 'channels.inParameter.eType
[ ', ' channels.inParameter.eType* ') '] ' ' channels.name[ '(' 'channels.inParameter.eType* ') '] ]'}
\]

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 explaining 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-TIMESTAMP

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 which 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.

![Initial State](image)

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.

![Final State](image)

Figure 3.15.: Concrete Syntax of 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 nondeterministic 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.

![Figure 3.16.: Concrete Syntax of an Orthogonal State With Two Regions](image-url)
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 the source state of the transition changes its status from active to inactive and that the target state of the transition changes its status from inactive to active. Further, the exit-action of the source state is executed, the transition-action of the transition is executed, the raise message-event of the transition is released, and finally 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 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 is dependend on the target platform and might not be available before the deployment.

We distinguish among *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 3.3.8. 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 3.3.9).

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.

If several transitions are enabled whose 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 3.3.8).
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 defined within a transitionLabel. The only exception is the definition of a deadline for which a special deadlineLabel is used. Figure 3.17 shows a template for the transition syntax.

Figure 3.17.: 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:

\[
\text{transitionLabel} ::= [ \text{clConstraints} ] [ \text{guard} ] [ \text{triggerEvent} ] [ \text{sync} ] [ \text{clockResets} ]
\]

\[
\text{sync} ::= \text{receivedSync} \mid \text{sendSync}
\]

\[
\text{triggerEvent} ::= \#\text{triggerEvent.name} \'(\ [ \text{parameterList} ] \')
\]

\[
\text{clConstraints} ::= [\ ',\ ',\ ',\ ',\ ',\ ',\ ']
\]

\[
\text{guard} ::= [\ ']
\]

\[
\text{action} ::= \{\ (\text{transitionAction.expressions} \mid \text{transitionAction.name})\}
\]

\[
\text{clockResets} ::= \{\text{reset}::\text{clockList}\}
\]

\[
\text{receivedSync} ::= \text{syncExpr}\'?
\]

\[
\text{sendSync} ::= \text{syncExpr}\!'
\]

\[
\text{syncExpr} ::= \text{syncChannel.name} \['\ ']
\]

\[
\text{integerExpression} ::= \text{syncChannel.name} \['\ ']
\]

\[
\text{clConstraints} ::= [\ ',\ ',\ ',\ ',\ ',\ ']
\]
### 3.3. REAL-TIME STATECHART

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`:

\[
<\text{deadlineLabel}> ::= [ \\
[ ' [ ' #relativeDeadline.lowerBound ', #relativeDeadline.upperBound ' ] ' ] \\
[ ' \n' #absoluteDeadline.clock.name ' ∈ ' ] \\
[ ' [ ' #absoluteDeadline.lowerBound ', #absoluteDeadline.upperBound ' ] ' ] \\
]*
\]

Figure 3.18 shows an example of a concrete transition.

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

#### Constraints

- The source and the target of the transition must be set.
- Transitions cannot cross region borders.

### 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 3.19.

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 3.19.

![Figure 3.19: Concrete Syntax of Entry-/Exit-Points](image)

**Constraints**

- Outgoing transitions of entry-points and of state exit-points may not carry any additional elements.
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 3.20).

![Concrete Syntax of Shallow History](image)

**Figure 3.20.: Concrete Syntax of Shallow History**

**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 an arbitrary language, such as Java, C, or Modelica. Currently, we do not read or parse any actions. We, therefore, define no semantics for it.

We plan for the next version of **MECHATRONIC UML** to define a **MECHATRONIC UML** action language that is independent of any concrete programming language and has a defined syntax and semantics. This action language should be transferable to specific programming languages. Instead of defining actions in an action language, it is possible to describe them via graph transformation rules or story diagrams [EHH+11].

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. 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 synchronize 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 raised message-events of the receiver transition. We allow only one synchronization, receiving or sending, per transition.

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 3.21. 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 \(B1 \rightarrow B2\) would be executed before \(C1 \rightarrow C2\) 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 so the assignment order is \(x := 2; x := 1\); instead which results in the final value for \(x\) of 1.

The region priority of the sender transition determines the priority of the synchronization. Figure 3.22 gives an example for this behavior. The initial composite state \(A1\) declares the synchronization channels \(\text{syncBE}()\) and \(\text{syncCD}()\) 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 \(B1 \rightarrow B2\) with the highest region priority of 4 receives and the transition \(E1 \rightarrow E2\) with the lowest region priority of 1 sends synchronizations via \(\text{syncBE}\). The transitions with region priorities in between \(C1 \rightarrow C2\) with region priority of 3 and \(D1 \rightarrow D2\) with region priority of 2 – send respectively receive synchronizations via \(\text{syncCD}\). Because the sender transition determines the priority of the synchronization the transitions \(C1 \rightarrow C2\) and \(D1 \rightarrow D2\) fire which results in the final state configuration \((A1, B1, C2, D2, E1)\).

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 3.23 illustrates this. It differs from Figure 3.22 insofar that state \(A1\) declares only the single synchronization channel \(\text{sync}()\) which all four transitions use. Because only one-to-one synchronizations are allowed this results in the four different synchronization combinations \(C1 \rightarrow C2\) and \(B1 \rightarrow B2\) or \(C1 \rightarrow C2\) and \(D1 \rightarrow D2\) or \(E1 \rightarrow E2\) and \(B1 \rightarrow B2\) or \(E1 \rightarrow E2\) and \(D1 \rightarrow D2\). Of these the sender transition with the highest region priority \(C1 \rightarrow C2\) synchronizes with the receiver transition with the highest region priority \(B1 \rightarrow B2\) which results in the final state configuration \((A1, B2, C2, E1, D1)\).
Figure 3.22.: The Sender Transition Determines the Priority of the Synchronization so Final State Configuration is (A1, B1, C2, D2, E1)

Figure 3.23.: Only One-to-One Synchronizations Are Allowed So Final State Configuration is (A1, B2, C2, D1, E1)
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 3.24. The initial composite state A1 declares

![Figure 3.24: Synchronization May Overrule Transition Priorities](image)

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,B1,C1) the transitions B1→B2 and C1→C2 would be executed because of their respective higher priorities. But caused by the synchronization the transition C1→C3 is executed instead of C1→C2. As shown in the example of Figure 3.22 this would not be the case if transition B1→B2 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 3.25. It differs from Figure 3.24 insofar that 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
Figure 3.25.: Synchronization May Violate the Principle that Transitions with Higher Located Source State Overrule Transitions with Lower Located Source States
states D1 and D2 and the transition from D1 to D2 which receives a synchronization via channel sync and assigns variable x with the value 2. Without the sending and receiving of synchronizations via channel sync and starting with the active state configuration (A1,B1,C1,D1) the transitions B1→B2 and C1→C2 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 B1→B2 and D1→D2 are executed first – resulting in the value 2 of variable x – and afterwards transition C1→C2 – resulting in the final value 3 of x.

3.3.9. 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 3.18 shows an example of asynchronous message-events used by a transition.
3.4. Component Model

In MechatronicUML, 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 composed 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 MechatronicUML 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 MechatronicUML 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-
3.4. COMPONENT MODEL

A continuous component is assumed to be specified in a control engineering tool like CamelView\(^2\) or Matlab/Simulink\(^3\).

Figure 3.26 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.

![Concrete Syntax of Different Kinds of Atomic Components](image)

Figure 3.26.: Concrete Syntax of Different Kinds of Atomic Components

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.

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 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.

\(^2\)http://www.ixtronics.com/ix_hist/English/CAbEView.htm
\(^3\)http://www.mathworks.de/
A discrete port is used for sending or receiving discrete, asynchronous messages that are typed over a message type (cf. Sections 3.3.9 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 MECHATRONICUML 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". 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 3.27. 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 3.26. 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 3.26).

![Concrete Syntax of Ports](http://www.mathworks.de/help/toolbox/simulink/ug/f15-99132.html)

Discrete and hybrid ports are depicted by squares, whereas continuous ports are depicted by isosceles triangles. Discrete ports embed small filled 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.

\[\text{http://www.mathworks.de/help/toolbox/simulink/ug/f15-99132.html}\]
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. 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]. 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 3.28. A multi-port has a cascaded double border line and it positioned like a single-port.

<table>
<thead>
<tr>
<th></th>
<th>in-port</th>
<th>out-port</th>
<th>in/out-port</th>
</tr>
</thead>
<tbody>
<tr>
<td>discrete</td>
<td><img src="image" alt="in-port" /></td>
<td><img src="image" alt="out-port" /></td>
<td><img src="image" alt="in/out-port" /></td>
</tr>
<tr>
<td>continuous</td>
<td>n/A</td>
<td>n/A</td>
<td>n/A</td>
</tr>
<tr>
<td>hybrid</td>
<td>n/A</td>
<td>n/A</td>
<td>n/A</td>
</tr>
</tbody>
</table>

Figure 3.28.: Concrete Syntax of Multi-Ports

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 specification 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.
- 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.

### 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. That Real-Time Statechart 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 3.3.9).

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 methods are specified by means of story diagrams [FNTZ00] that combine UML Activity Diagrams [Gro10b] and graph rewrite rules [Roz97]. 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 3.3.8 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 3.3.8). 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 MECHATRONICUML specification (cf. Section 3.4.1.4).

\(^5\)[http://www.mathworks.de/help/toolbox/simulink/ug/f15-99132.html]
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 3.29 shows an example.

![Diagram of Hybrid Port Behavior Specification](image)

Figure 3.29.: Example of the Behavior Specification of Hybrid Ports

Figure 3.29 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 checking problem. Model checking such hybrid specifications is infeasible. Thus, we assign an update rate to the hybrid port. Then, this update rate specifies that the variable representing the signal value only changes each $x$ units of time which allows to perform standard timed model checking.

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 MECHATRONIC UML 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. Section 3.3.8). The synchronizations must not be used for communication between different component instances.

Figure 3.30 shows an example of the component behavior specification of the component Navigation containing one state Navigation_Main. 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.

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, synchronizations 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 methods 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 methods 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.
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. 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.

Figure 3.31 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 MotorCtrl 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 3.26).

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 like an atomic component. The component part is labeled with the name of the component it is typed over. Additionally, the label contains the cardinality of the component part in the following form:

`[ ' #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 3.31 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 3.32.

Analogously to multi-ports, a multi-part is visualized by a cascaded double border line as shown in Figure 3.32.
3.4. COMPONENT MODEL

Figure 3.31.: Concrete Syntax of a Hybrid Structured Component

Figure 3.32.: Concrete Syntax of a Multi-Part
CHAPTER 3. MODELING LANGUAGES

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.

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 3.33 shows the structured component \texttt{BeBot\_SW} that has five ports that are delegated to two embedded component parts of types \texttt{Navigation} and \texttt{BeBot\_Observer}. The delegation link is represented by a 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.

![Figure 3.33: Concrete Syntax of a Delegation](image)

Whether a delegation link may be created depends on two conditions. First, they have to be structurally compatible according to Figure 3.34. Second, they must have compatible inter-
faces. 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 3.34. If a combination is marked with a checkmark, it is possible, otherwise it is not possible.

![Figure 3.34.: Structurally Possible Delegation Links](image)

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 3.35 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 assigned, 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.

### 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 3.36 shows the complete definition of the structured component BeBot_SW. The assembly connectors are visualized as 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 Navigation is connected to the component parts Exploration and BeBot Observer which means that the component part Navigation may communicate with the component part Exploration or the component part BeBot Observer or both.

Figure 3.37 shows the complete definition of the structured component BeBot. It embeds the discrete structured component BeBot_SW of Figure 3.36 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 3.36.: Complete Example of a Discrete Structured Component including several Assemblies connecting the Component Parts
Figure 3.37.: Example of a Hybrid Structured Component
3.4. COMPONENT MODEL

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 3.38. 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 3.38. 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 3.36 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 3.36, 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 3.37, 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.
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 component instance is the component type referenced by the component part. There may also exist several instances of one part depending on the component part’s cardinality. Initial instances of multi-parts and multi-ports are created with lowest cardinality.

Figure 3.39 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:BeBot_SW of Figure 3.39 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. 3.36) has two sub-port instances distributor1 and distributor2 that implement this multi-role.

Figure 3.40 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.
CHAPTER 3. MODELING LANGUAGES

Figure 3.39.: Concrete Syntax of a Component Instance of the Component Type BeBot

Figure 3.40.: Structure of a Multi-Port Instance being Connected to Several Single-Port Instances
3.5. Component Instance Configuration

A component instance configuration is obtained by connecting component instances via connectors 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 to 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 3.41 shows the component instances b1:BeBot_SW, b2:BeBot_SW, and b3:BeBot_SW, b2:BeBot_SW and b3:BeBot_SW are connected to b1:BeBot_SW 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 b1:BeBot_SW implement the multi-role distributor and the two ports named client of b2:BeBot_SW and b3:BeBot_SW implement the role client. The component instances and assembly connector instances form a component instance configuration.

Figure 3.41.: Concrete Syntax of a Component Instance Configuration

The component instance configuration shown in Figure 3.41 shows the highest hierarchy level of the system. Figure 3.42 shows the component instance configuration that specifies the inherent part structure of b1:BeBot at the next lower level. 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 3.42.: Concrete Syntax of a Component Instance Configuration
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 components to hardware nodes.

3.6.1. Hardware Nodes

A deployment consists of a component instance configuration and a set of hardware nodes. A hardware node represents a physical entity, e.g. an electronic control unit where component instances may be deployed. Further, hardware nodes have hardware ports to communicate with other hardware nodes and with software components. Hardware ports only have one direction, either incoming or outgoing.

In this work, we omit the fact that the software is deployed on a computing hardware. We only show the information flow between the software and sensors and the software and actuators.

Figure 3.43 shows a hardware node that models the physical BeBot. It is visualized by a 3D-box. It has a name that follows the syntax of a component instance as described above. The ports are visualized by squares that contain either the letter “i” or “o”. “i” indicates that the port has incoming data, “o” stands for outgoing data.

![BeBot_HW]

Figure 3.43.: Concrete Syntax of a Hardware Node

3.6.2. Allocation

Component instances are allocated to hardware nodes by communication links. The communication link specifies on which hardware node a component instance is allocated and which port of the component instance is connected to which port of the hardware node. Figure 3.44 shows the deployment of the component instance b1:BeBot_SW on the hardware node BeBot_HW.

The above description of deployment is our first step into this direction and is planned to be further refined in subsequent versions of this document.
Figure 3.44.: Deployment of a Component Instance to a Hardware Node
Chapter 4.

Development Process

A first coarse-grained description of the MECHATRONICUML process was already published by Schäfer et al. [SSGR11]. The goal of this process definition is to explain the MECHATRONICUML process in more detail and 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 MECHATRONICUML process.

MECHATRONICUML is designed to support the development of software for advanced mechatronic systems. The MECHATRONICUML process can, therefore, not be seen isolated from the development process of the whole mechatronic system that also consists of components from other disciplines such as mechanical engineering, electrical engineering, and control engineering. The integration of MECHATRONICUML within the overall development process of mechatronic systems is explained in Section 4.1. Afterwards, the MECHATRONICUML process is described in detail in Section 4.2.

4.1. Integration into the Development Process for Advanced 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 support 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 advanced 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 the principle solution, a first system model which captures all interdisciplinary
relevant aspects of the system. The principle solution 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]. The purpose of the principle solution is to form a common basis for the subsequent parallel development of the discipline-specific parts of the system within the discipline-specific development phase. In this second 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 parts of the system are integrated into an overall consistent system model.

Figure 4.1.: The Integration of the MECHATRONICUML Process into the Development Process of Advanced Mechatronic Systems
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.

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.

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 situations of the system are described by **application scenarios**. An application scenario consists of a textual description that explains the situation 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.

### 4.2. The **MECHATRONICUML Process**

Figure 4.2 shows a diagram of the overall MECHATRONICUML process. Based on the active structure, the application scenarios, and the behavior specification from the principle solution, the software components are developed using MECHATRONICUML in 8 major steps and two steps starting an iteration (Steps 9 and 10).

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.

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 system elements. System elements that consist of other system elements are represented by structured components in the component model. The 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 component model can be extended by further components and connectors. The result of this step is the initial version of the component model.

In the principle solution, the behavior is often specified coarse-grained and informally. Furthermore, the application scenarios and the behavior specifications typically describe many in-
CHAPTER 4. DEVELOPMENT PROCESS

Figure 4.2.: Overall MechatronicUML Process
terdependent interactions of many components and the environment. However, these behavior specifications can often be decomposed into smaller specification parts that span a smaller set of components. For these smaller specifications, that we call protocols, the behavior can be implemented more easily and verified more effectively.

Based on the derived component model, the behavior, that is specified in the principle solution, is decomposed into informal requirements for the protocol behavior of each connector in the component model (Step 2). The result are a set of constraints for the components’ communications and a set of requirements for the external visible behavior of all components. Both results are described informally in this step.

In Step 3, the informal requirements are used to specify the protocol behavior more precisely 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 liveness properties of the protocol. For each participant in the protocol, an abstract role is modeled by a Real-Time Statechart to allow a flexible reuse of the protocol in other contexts. These Real-Time Statechart are later (cf. Step 4) instantiated and refined in a component’s port. Additionally, temporal logic is used to define properties that hold for the protocol behavior. 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 participates in 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 external 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, a process that is similar to the overall MECHATRONICUML process is performed (cf. Section 4.2.4). 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 the subcomponents, and the behavior specification of all subcomponents. The behavior of the component is typically specified by a Real-Time Statechart. If the subcomponent is a reused component, only the binary code may exist.

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 MECHATRONICUML approach, the system is simulated to identify missing constraints and flaws of other disciplines’ models (Step 6). At the moment, the 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. This is necessary, because MECHATRONICUML 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. The component model is, therefore, not sufficient to create a simulation model of the system. For the simulation to start, an initial instance of the structured component can be specified by a component instance configuration as described in Section 3.5.

For different application 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 requirements for the protocol behavior and the constraints as specified in Step 2 are incomplete. The former case must be handled within the development of the other disciplines (Step 10). After the redesign of the other disciplines’ models, Step 6 is repeated.

If missing requirements for the software are identified, the set of informal behavior requirements and the set of constraints will be extended for the corresponding communication rela-
4.2. THE MECHATRONIC UML PROCESS

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.

In Figure 4.3, it is shown that, therefore, the coordination patterns of a structured component are determined in parallel for all communication relations. 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 informal requirements of the behavior, an appropriate coordination pattern is identified for the constraints of the communication, and the corresponding application scenarios described in the principle solution.

If it is not possible to find an existing coordination pattern, a new Real-Time Coordination Pattern is modeled in Step 3.2. Based on the informal behavior requirements and the application scenarios, 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 informal 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 behavior that is specified by Real-Time Statecharts, a connector that models the behavior of the roles’ communication channels by a Real-Time Statechart, and a set of safety constraints that hold for the communication protocol. First, in Step 3.2.1, a set of formal
Figure 4.3.: The Subprocess to Reuse or Model a new Coordination Pattern for all Communications within one Structured Component

safety constraints is derived from the informal behavior requirements and the constraints of the the communication. In parallel, the behavior of the coordination pattern is defined in the Steps 3.2.2 to 3.2.6.

During Step 3.2.2, the roles are derived from the informal behavior requirements and the application scenarios. Mostly, the participating components within the application scenarios and the informal behavior requirements are role candidates. An applicable set of roles is found if the informal behavior requirements and the application scenarios can be decomposed in such a way that the parts of the behavior can be associated with the roles easily.

For each role, the message interface (cf. Section 3.2) is derived from the informal behavior requirements (Step 3.2.3). As the roles in Step 3.2.1 implicate a decomposition of the behavior, the messages, that must be sent and received by a role, can be identified based on the informal behavior specification.

The roles’ behaviors are specified in Step 3.2.4. The informal behavior specification 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-
4.2. THE MECHATRONICUML PROCESS

Process: model new coordination pattern (3.2)

- informal behavior requirements
- structured component
- extract formal constraints (3.2.1)
- set of verification constraints
- application scenarios
- informal behavior requirements
- structured component
- derive roles (3.2.2)
- set of roles
- iterate
- specify roles' behavior (3.2.4)
- set of roles
- adapt roles' behavior to connector properties (3.2.6)
- set of Real-Time Statecharts for connectors
- set of Real-Time Statecharts for connectors
- coordination pattern candidate

- informal behavior requirements
- structured component
- derive message interfaces for each role (3.2.3)
- set of message interfaces
- iterate
- specify roles' behavior (3.2.4)
- set of roles
- adapt roles' behavior to connector properties (3.2.6)
- set of Real-Time Statecharts for connectors
- set of Real-Time Statecharts for connectors
- coordination pattern candidate

- formal verification of constraints (3.2.7)
- coordination pattern

Figure 4.4.: The Subprocess to Model a new Coordination Pattern
Time Statechart that describes the roles’ behavior such that formal analysis techniques can be used to verify the safety properties [HH11].

In parallel, the quality properties of the communication channel are specified. 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 3.2.5, 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 within the roles’ behavior. The Real-Time Statecharts of the roles must be adapted in such a way that the safety constraints for the communication still hold. In Step 3.2.6 the roles’ behavior is extended accordingly.

At last (Step 3.2.7), 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, the Steps 3.2.4 and 3.2.6 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 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.
4.2. THE MECHATRONIC UML PROCESS

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 construction 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 [GTB03, 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 dependend 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

Figure 4.5.: The Subprocess to Determine the Internal Structure or Behavior of a Component
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 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
4.2. THE MECHATRONIC UML PROCESS

**Process:** derive components’ behavior (4.3)

Roles’ behavior described by Real-Time Statecharts

specify dependencies between component’s roles (4.3.3)

automatic synthesis of synchronized behavior (4.3.4)

check for deadlock freedom (4.3.6)

refinement is incorrect

refinement is correct

set of Real-Time Statecharts for port behavior

set of Real-Time Statecharts for port behavior

composition rules

composition rules

Real-Time Statechart for the component’s behavior

Real-Time Statechart that synchronizes the port statecharts

model synchronized behavior manually (4.3.5)

set of Real-Time Statecharts for port behavior

automatic synthesis of synchronized behavior (4.3.4)

Real-Time Statechart for component’s behavior

<optional> Real-Time Statechart for component’s behavior

<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optional>
<optio
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 the whole example with MECHATRONICUML. As an example, we use the environment exploration scenario using BeBots as introduced in Section 1.1. In the following sections, we first describe the Real-Time Coordination Patterns including their role behaviors used for the scenario in Section 5.1. Afterwards, we summarize the message interfaces used for the specification of the Real-Time Coordination Pattern 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 several 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 which we use in our example (cf. Figure 5.1).

The patterns Navigation and Delegation have both the form of communication 1:1 and the communication direction bidirectional. The patterns Distribution, PositionTransmission and DistancesTransmission have the form of communication 1:n, but Distribution has the communication direction bidirectional and PositionTransmission and DistancesTransmission the communication direction unidirectional (cf. Section 3.1.4). In the following sections, we introduce the used Real-Time Coordination Patterns and the behavior of their roles.

Examples for pattern instantiations are shown in Figure 5.18 in Section 5.3. All of the five Real-Time Coordination Patterns of Figure 5.1 are instantiated within the BeBot_SW component.

5.1.1. Navigation Pattern

The 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 which the roles use in Section 5.2.1.
Figure 5.1.: The Five Real-Time Coordination Patterns used in the BeBot Scenario

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.

5.1.1.2. Role Provider

Figure 5.3 shows the Real-Time Statechart of the role provider. It represents the counter-part
5.1. REAL-TIME COORDINATION PATTERNS

Figure 5.3.: Role Provider of Pattern Navigation

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 \texttt{moveTo} triggers the transition from Polling to Moving. The message \texttt{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 \texttt{targetReached}.

5.1.2. Delegation Pattern

The 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

Figure 5.4.: Role Master of Pattern Delegation

Inactive and the states PositionCheck, Success and Fail. 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 c0 is reset via an entry-action. An invariant using c0 ensures that PositionCheck is left no later than 150 units of time after
its activation. There are two outgoing transitions from which the one with the higher priority is triggered by the message declined and leads to Fail. The message accepted triggers the other transition and leads to Success. From both, Success and Fail, a transition goes back to Inactive.

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. Furthermore PositionCheck has to be left no later than 75 units of time after its activation now and an exit-action resets c0, too. The outgoing transitions are trigger-less. The states Success and Fail have invariants on c0 which ensure their leaving no later than 25 units of time after PositionCheck is left. Finally, the transitions to Inactive send the messages declined, when leaving Fail, and accepted, when leaving Success.

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 which 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,
5.1. REAL-TIME COORDINATION PATTERNS

Distribution_distributor_main, with the two regions adaptation and sub-role (cf. Section 3.1.6). The sub-role region contains a Real-Time Statechart which only consists of the state Active. The state Active, in turn, contains 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 it the states Idle of receive and send. If a sub-role receive position data from a client in terms of a position message, the Real-Time Statechart in region receive switches from Idle to Done. Periodically, the adaptation region checks whether position data has been received from all clients. Therefore, it fires the transition from Waiting to Receiving if c0 is greater than 50 thereby synchronizing with the first sub-role. If the message position
is available, 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 an the sub-role switches to Done and sets the variable error to true. Afterwards, it triggers the next sub-role using the synchronization receive[k+1]. The sub-roles trigger each other until the last sub-role n reaches state Done. Then, the synchronization receive[k+1] synchronizes with the adaptation statechart. If error is true, then the adaptation statechart switches to state Error, otherwise it switches to state All_Received.

After the distributor received new position data from all of its clients, it starts 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 send[1]. Then, the region send of the first sub-role switches to Done and sends the message positions to the client. Thereafter, it synchronizes with the next sub-role using the synchronization send[k+1]. The sub-roles triggers each other again until the sub-role n reaches state Done. Then, this sub-role synchronizes with the adaptation statechart which switches to state Waiting. Then, 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 c0 becomes greater than 35 unit 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 c0 exceeds 45 units of time. The position data is send 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 c0 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).
The region adaptation is nearly identical to the one in Figure 5.6, only the invariants use in 10 and 5 different upper bounds. The region sub-role is only in such ways different, that it is reduced to the region sending which maintains the synchronization chain and that it sends the message position with the own position, as one-dimensional integer array pos, instead.

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, and is nearly identical to the region receive of the Real-Time Statechart in Figure 5.7. The only differences are the different message position which is received and the different parameter xy which is set to the variable pos.

5.1.5. DistancesTransmission Pattern

The pattern DistancesTransmission (1:n, unidirectional) is nearly identical to the pattern PositionTransmission and transmits an array of distance values 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 5.2.5.

5.1.5.1. Role Sender

Figure 5.10 shows the Real-Time Statechart of the multi-role sender. It is nearly identical to the Real-Time Statechart in Figure 5.8 and differs only in the upper bound 150 of the invariant of state Waiting and in the sending of the message distances with a one-dimensional floating point array as parameter.

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
5.1. REAL-TIME COORDINATION PATTERNS

Figure 5.10.: Role Sender of Pattern DistancesTransmission

DistancesTransmission_sender

DistancesTransmission_sender_main

adaptation

next[n+1]()!

next[n+1]()?

Waiting

c0 ≤ 150

c0 ≥ 150

entry / {reset: c0}

Sending

c0 ≤ 5

c0 ≥ 5

entry / {reset: c0}

sub-role[k]

next[k+1]()!

next[k]()?

distances(distArray)

Idle

Sent

var: float[bebots] distArray;

Figure 5.11.: Role Receiver of Pattern DistancesTransmission

DistancesTransmission_receiver

var: float[bebots] distArray;

distances(float[bebots] array) / \{distArray := distances.array;\}

Active

c0 ≤ 100

c0 ≥ 100

entry / {reset: c0}

Error

distances(float[bebots] array) / \{distArray := distances.array;\}
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.
5.2. Message Interface Specification

In this section, we introduce the message interfaces which the Real-Time Coordination Patterns of Section 5.1 uses. 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 Pattern

The Navigation pattern 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 pattern.

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

Figure 5.12.: Message Interfaces for the Navigation Pattern

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 Pattern

We use the Delegation pattern 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 pattern uses.

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

Figure 5.13.: Message Interfaces for the Delegation Pattern

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 Pattern

The Distribution pattern transmits position data between the BeBots. Figure 5.14 shows the two message interfaces used by the pattern.

![Distribution Pattern Message Interfaces](image)

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 \( k \) 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 Pattern

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

![Position Transmission Pattern Message Interface](image)

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.
5.3. SYSTEM STRUCTURE

5.2.5. DistancesTransmission Pattern

The DistancesTransmission pattern transmits the distances to the other BeBots inside the BeBot. Since it is a unidirectional 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(array : int[k])</td>
</tr>
</tbody>
</table>

Figure 5.16.: Message Interfaces for the DistancesTransmission Pattern

The DistancesTransmission_Sender message interface specifies one message type distances to transmit the distances. It contains a one-dimensional array of length $k$ where $k$ is the number of client 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 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, the BeBot contains two continuous component parts EngineCtrl and PosData. The 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 PosData component part provides the GPS positioning data to the software.
The 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 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 77.

Figure 5.18 shows the structured component BeBot_SW that represents the software of the BeBot.

The component specifies five ports: speed_left, speed_right, position, client, and distributor. The continuous ports speed_left and speed_right set the speed for the chain drives of the BeBot. The continuous port position obtains the current position from the GPS sensor.

We specify the behavior of the remaining two discrete ports of BeBot_SW and the discrete ports of the atomic components Navigation, Exploration, BeBot_Observer and 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 Exploration and the port provider of the component part Navigation. An instance of pattern Delegation is assigned to the port master of the component part Exploration and the port slave of the component part Collision_Control. The instantiated pattern DistancesTransmission describes the communication between the ports sender of the component part BeBot_Observer and receiver of the component part 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 BeBot_Observer and 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.

![Figure 5.19.: The Atomic Components of the BeBot](image)

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. SYSTEM STRUCTURE

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 the complete behavior in detail of the four atomic components Exploration, Navigation, BeBot_Observer and Collision_Control introduced in Section 5.3.2. Real-Time Statecharts specifies 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 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 via 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 transition from Fail back to Inactive synchronize via channel the positionRejected with the transition from the state CheckPosition to the state DecideOnNextPosition. The transition from Success back to Inactive synchronize via the channel positionOk with the transition from CheckPosition to PositionOk.

The region synchronization contains the two states Active and Error which are changed according to received synchronizations over the channels noPositionData, from Active to Error, and positionDataOk, back to Active. The initial state Active consists of a single region with the four states DecideOnNextPosition, CheckPosition, PositionOk and Drive. The initial state
Figure 5.20.: Real-Time Statechart of Component Exploration
5.4. REAL-TIME STATECHART

DecideOnNextPosition determines a new target position by calling the operation playDice with the current position pos as parameter and resets the clock c0 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 c0 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 c0. 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 c0 per entry-action. Both result in the reactivation of state DecideOnNextPosition but the second requires the sending of a synchronization over the channel drive, with the target position as parameter, and the receiving of one over channel driveComplete and the traversal of PositionOk and Drive in between and is lower prioritized than the first.

5.4.2. Navigation

The component Navigation transmits the current position to Exploration and BeBot_Observer and receives a target position from Exploration. We need 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 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, syncRight to synchronize the new speed with the Real-Time Statechart of the corresponding hybrid ports. Both channels have one integer parameter which represents the desired speed of the left and right chain drive. The Real-Time Statecharts of the regions speed_left, 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 gets 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 currentPos-Deleg. 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 specify 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 via 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). The parameter n is replaced upon instantiation of the component by the number of the instances of the multi-port. 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 knows its current position. Therefore, it synchronization repetitively every 4 ticks via the channel currentPosDeleg with the synchronization Real-Time Statechart to get the current position. The position is stored in the variable sendPosition. This variable is used as the parameter to the message position.

The region synchronization contains the six states Stop, LeftStop, RightStop Calculate-Speed, RightMove, and Move. Stop is the initial state. The transition to CalculateSpeed receives a synchronization via channel go and sets the received parameter pos as the current target position. The entry-action of CalculateSpeed resets clock c0 and calls the operation calc-Speed, 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 c0 ensures that CalculateSpeed is left no later than 20 units of time after its activation. If the value of c0 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 c0 in its entry-action ensure that it is left 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 c0 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 State-
BeBot_Observer

synchronization
receiver

Active

CommError

commOk()? /
commError()? /

sender(n)

client
distributor(n)

Figure 5.22.: Real-Time Statechart of Component BeBot Observer
charts of the component ports and one Real-Time Statechart to synchronize them. Therefore BeBot_Observer_Main declares the channels commOk, commError, ok and error all without parameters.

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). 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 variable pos.

The multi-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). The parameter n is replaced upon instantiation of the component by the number of the instances of the multi-port. It is refined by adding the state Error and a transition from DistancesTransmission_sender to Error, which receives a synchronization via channel error, and a transition back, which receives a synchronization via channel ok.

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 adding the sending of a synchronization via channel commError to the transition from Receiving to Error and one via commOk to the reverse transition.

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). The parameter n is replaced upon instantiation of the component by the number of the instances of the multi-port. 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 synchronization 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 c0 ensure that ErrorReceived and OkReceived, which entry-actions reset clock c0, 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
CHAPTER 5. COMPLETE EXAMPLE

CollisionControl

Collision_Control_Main

synchronization

receiver

Running

slave

Figure 5.23.: Real-Time Statechart of Component Collision Control
composite state, Collision_Control_Main, whose parallel regions contain the Real-Time Statecharts of the component ports and one Real-Time Statechart to synchronize them. Therefore Collision_Control_Main declares the channels checkPermission, with an 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 Fail and via granted to the transition from PositionCheck to Success.

The region synchronization contains the three states Running, which is the initial state, Error and FixingError. A transition from Running to Error receives synchronizations via channel noPositionData and the reverse transition via positionDataOk. The other outgoing transition from Error to FixingError is lower prioritized and receives synchronizations via checkPosition. FixingError is left immediately after its activation by the use of an invariant over clock c0 and a clock reset of c0 by its entry-action. The transition back to Error sends a synchronization via channel rejected. The composite state Running contains a region with the two states Wait and PositionCheck. The transition from the initial state Wait to PositionCheck receives a synchronization over channel checkPermission and calls the operation riskEvaluation with the received target position and the one-dimensional floating point array distArray as parameter. The result is set to the boolean variable permission. PositionCheck has to be left no more than 50 units of time after its activation which is ensured by its invariant over clock c0 which is reset by the 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 that is not Connected to other BeBots](image)

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

Figure 5.25 shows the structure of the embedded component instances of the BeBot 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. COMPONENT INSTANCE CONFIGURATION

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 \( b1: \text{BeBot}_\text{SW} \) of Figure 5.26 has the two discrete port instances \( \text{distributor1} \) and \( \text{distributor2} \) that both implement the multi-role distributor of the Real-Time Coordination Pattern \( \text{Distribution} \). 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 \( b2: \text{BeBot}_\text{SW} \) of Figure 5.27 has a discrete port instance client that implements the single-role client of the Real-Time Coordination Pattern \( \text{Distribution} \). 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 \( \text{BeBot}_\text{SW} \). In order to obtain a component instance specification for a BeBot, the component type \( \text{BeBot} \) 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 \( \text{bebot1:BeBot} \), \( \text{bebot2:BeBot} \) and \( \text{bebot3:BeBot} \). \( \text{bebot1:BeBot} \) is the position distributor while \( \text{bebot2:BeBot} \) and \( \text{bebot3:BeBot} \) operate as clients. Thus, the assembly connector instances which is derived from the Real-Time Coordination Pattern \( \text{Distribution} \) connects the \( \text{bebot2:BeBot} \) and the \( \text{bebot3:BeBot} \) to \( \text{bebot1:BeBot} \). The discrete port instances \( \text{distributor1} \) and \( \text{distributor2} \) of \( \text{bebot1:BeBot} \) implement the multi-role distributor. The discrete port instances client of \( \text{bebot2:BeBot} \) and \( \text{bebot3:BeBot} \) implement the single-roles client. Figure 5.28 also shows the embedded component instances of \( \text{bebot1:BeBot} \): \( b1: \text{BeBot}_\text{SW}, \text{ctrl}_\text{left}: \text{EngineCtrl}, \text{ctrl}_\text{right}: \text{EngineCtrl}, \text{pd}: \text{PosData}, \text{eng}_\text{left}: \text{Engine}, \text{eng}_\text{right}: \text{Engine}, \text{and gps}: \text{GPS} \). The two discrete port instances \( \text{distributor1} \) and \( \text{distributor2} \) of the component instance \( b1: \text{BeBot}_\text{SW} \), that implement the role distributor of the Real-Time Coordination Pattern \( \text{Distribution} \), are delegated to the discrete port instances \( \text{distributor1} \) and \( \text{distributor2} \) of \( \text{bebot1:BeBot} \). Component instance \( \text{ctrl}_\text{left}: \text{EngineCtrl} \) and component instance \( \text{ctrl}_\text{right}: \text{EngineCtrl} \) each have the two continuous port instances \( \text{speed}_\text{in} \) and \( \text{speed}_\text{out} \). Component instance \( \text{pd}: \text{PosData} \) has the two continuous port instances \( \text{pos}_\text{in} \) and \( \text{pos}_\text{out} \). The hardware component instances \( \text{eng}_\text{left}: \text{Engine} \) and \( \text{eng}_\text{right}: \text{Engine} \) each have an incoming hardware port. The hardware component instance \( \text{gps}: \text{GPS} \) has an outgoing hardware port. The assembly
Figure 5.28.: Concrete Syntax of a Component Instance Configuration of Three Communicating BeBots
connector instances between b1:BeBot_SW and ctrl_left:EngineCtrl, fectrl_left:EngineCtrl and eng_left:Engine, b1:BeBot_SW and ctrl_right:EngineCtrl, ctrl_right:EngineCtrl and eng_right:Engine, b1:BeBot_SW and pd:PosData, and pd:PosData and gps:GPS are not derived from any Real-Time Coordination Pattern as they only connect continuous port instances or continuous port instances with hardware 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 instances 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.

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.
5.5. COMPONENT INSTANCE CONFIGURATION

**Figure 5.29.: Concrete Syntax of a Component Instance Distributor**

**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. Figure 5.31 shows the allocation of the component instance b1:BeBot_SW on the hardware node BeBot_HW. The continuous port instances speed_left, speed_right, and position are connected to the hardware ports of the hardware node BeBot_HW by communication links. speed_left and speed_right send data to the incoming hardware ports of BeBot_HW. position receives data from the outgoing port of BeBot_HW.

Figure 5.31.: Deployment of a Component Instance to a Hardware Node
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.

For now, we only explain the fundamentals of the compositional verification theorem of MECHATRONIC UML in Section 6.1.

6.1. Compositional Verification Theorem

This section will explain the compositional verification approach of the MECHATRONIC UML as defined in [Gie03]. In contrast to the original definition which uses real-time automata that are based on a discrete time model we abstract from these details here. Instead we extract the core idea and describe it in general terms such as paths, composition and refinement that are usually defined for state-based behavior descriptions. Like this the approach can easily be adopted to work with other state-based formalisms and time domains.

In the following two sub-sections we first explain some preliminaries followed by the actual approach.

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 $D \parallel D'(D)$ denote the paths that $D$ is able to execute while composed with $D'$. Then it holds (provided $D$ and $D'$ are independent) that $D \parallel D'(D) = \Pi(D)$. Of course, due to symmetry reasons, it also holds that $D \parallel D'(D') = \Pi(D')$. 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 $\Pi(D) \not\subseteq \Pi(D')$. On the other hand if the composition $D \parallel D'$ does not contain a deadlock then it holds that $\Pi(D) \subseteq \Pi(D')$. 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 $\Pi(D) \subseteq \Pi(D')$ and since $\varphi$ holds for all paths $\Pi(D)$ it will also hold for the subset $\Pi(D)\parallel\Pi(D')$ 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 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 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 $D'_1 \parallel D_2 \subseteq D_1 \parallel D_2$.

6.1.2. Putting it all together

Now we have all preliminaries to explain the compositional verification approach of the MECHATRONIC UML. As we already know the structure of a MECHATRONIC UML model is specified by Real-Time Coordination 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 MECHATRONIC UML model on their own and then ensure the correctness of the system by a syntactically correct composition of Real-Time Coordination Pattern and components. Like this a verification step for the whole system can be avoided. The verification is divided 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 $D_{P_i,1}, \ldots, D_{P_i,m}$. The additional Real-Time Statechart for the communication channel is denoted by $D_{P_i,C}$. 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 $P_{i,1}, \ldots, P_{i,k}$. The statecharts of the $k$-th instance of a pattern $P_i$ are denoted by $D_{P_{i,k},1}, \ldots, D_{P_{i,k},m}$ and its channel statechart as $D_{P_{i,k},C}$. 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.
Third step  In the next step we re-order the role instances such that all roll 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 $C_1, \ldots, C_h$ be the component instances of the system. Then the statecharts of the role instances of a component instance $C_i$ are denoted by $D^C_{i,1} \ldots D^C_{i,m_i}$. 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.

$$D^P_{[1,1],[1,1]} \parallel \ldots \parallel D^P_{[1,1],[m]} \parallel D^P_{[1,1],[i]} \parallel \ldots \parallel D^P_{[n,1],[m_n]} \parallel D^P_{[k,1],[C_i]} = \phi_1 \land \ldots \land \phi_2 \land \neg \delta$$

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 intermedi-ates 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 $M_i^C \subseteq D^C_{i,1} \ldots \parallel D^C_{i,m_i}$ 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 $D^C_{i,1} \ldots \parallel D^C_{i,m_i}$ 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 $D_1 \parallel D_2 \subseteq D'_1 \parallel D_2$. Here $M_i^C$ is $D_i$, $M_i^C \subseteq D^C_{i,1} \ldots \parallel D^C_{i,m_i}$ is $D'_i$ 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 fulfills the verified properties, i.e.

$$D^P_{[1,1],[C_i]} \parallel \ldots \parallel D^P_{[n,1],[m_n]} \parallel M_1^C \parallel \ldots \parallel M_h^C = \phi_1 \land \ldots \land \phi_2 \land \neg \delta$$

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 the component-based software design 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 the relationships to the other engineering domains to some extend. Examples are continuous components for controllers or hardware components for mechanical and electrical elements.

As a consequence, related work stems from the following areas:

- 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, SOFA HI, 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. 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]. The authors discuss among other approaches the use of the SysML [Gro10a] approach by the OMG or the use of MATLAB/Simulink in combination with Stateflow\(^1\). Another approach supporting an integrated modeling and analysis is Modelica [Fri04].

\(^1\)http://www.mathworks.de/
7.1.1. 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\(^2\). The standard is currently available in SysML version 1.2 [Gro10a].

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. SysML reuses the UML concept of state machine, use case and sequence diagrams\(^3\). The syntax and semantics of SysML activity diagrams is refined [JDB09].

SysML is developed to use model-based systems engineering (MBSE) [Fea98]. SysML targets the holistic systems engineering development. SysML focus on holistic modeling of mechatronic systems while MECHATRONICUML 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 MECHATRONICUML. A main difference is that in SysML a “block” can be either a software or a hardware element and in MECHATRONICUML 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 MECHATRONICUML which has a well defined syntax and semantics.

7.1.2. MATLAB/Simulink Stateflow

MATLAB is a tool suite for computing in systems engineering [Col07]. Simulink is a toolbox for graphical model based development. A modeller models graphically block diagrams that “depicts time-dependent mathematical relationships among the system’s inputs, states, and outputs”\(^4\).

Simulink block diagrams have a causal signal flow which means that their is for each signal output a defined signal source and a time-dependent mathematical relationship. In contrast to MECHATRONICUML where an atomic software component is 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 makes it possible to model an event-based reactive behavior specification. The Stateflow finite state machines have concepts for hierarchical and parallel states and supports the modeling of complex control flow as transitions between these Stateflow states. MATLAB functions can be invoked as actions when e.g. a state change is performed.

\(^2\)http://www.sysmlforum.com
\(^3\)http://www.sysmlforum.com/faq/relation-between-SysML-UML.html
\(^4\)http://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. Although, message-pools or buffers as described in Section 3.3.9 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 coordinate 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.

Further, Stateflow has only features for simple temporal logic operators, like the \texttt{after()} and \texttt{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 bunch of embedded real-time platforms. MechatronicUML can benefit from these features as the can be mapped to MATLAB/Simulink Stateflow. Currently, we are working on an automatic transformation from MechatronicUML to MATLAB/Simulink Stateflow.

7.1.3. Modelica

Modelica is a free object-oriented modeling and simulation language and is being developed by the non-profit Modelica Association since 1996. The language itself is independent of a concrete simulation environment. It is suitable for multi-domain modeling involving but not limited to, mechanical, electrical, hydraulic and control systems, process oriented applications and system dynamics. The model behavior is based on ordinary and differential algebraic equation (ODE and DAE) systems combined with discrete events, so called hybrid DAEs.

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+09]. 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 are more complex than classical statecharts. There-
for they are tedious to model and hard to maintain. Further, 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 (ODE) solver or differential algebraic equation (DAE) 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.

7.2. 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 develop-
ment [VDI04]. Based on the VDI 2206, the collaborative research centre SFB 614 created a
new process model for mechatronic systems development [GFDK09].

7.3. Software Component Models

The surveys [LW07] and [CSVC11] consider different component models and classify them
according to common characteristics. Both papers do not restrict themselves to component
models for a specific domain. However, one observation noted in [CSVC11] is that there are
on the one hand general purpose component models, like e.g. CORBA, EJB, or SOFA, and on
the other hand specialized component models for particular domains exist. According to this
survey, approaches in the second category usually aim at either business information systems
or embedded systems. As embedded systems are also the target domain of MECHATRONICUML, the approaches closest to ours are in this group of component models. The survey
[HPB+10] exclusively focuses on component models of this kind.

In several parts of this document we are referring to UML 2 [Gro10b], from which MECHA-
TRONICUML borrows especially syntactical elements related to components and statecharts.
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". A
major obstacle for using UML 2 directly for component-based development, and even more so
that of safety-critical systems, is its lack of a formal semantics. Also, as UML 2 is not specif-
ically designed for developing embedded systems, it does not provide any special support for
specifying time-dependent behavior.

A major influence to early versions of MECHATRONICUML was ROOM [SGW94], an
object-oriented development methodology for embedded systems. Although it does not di-
rectly support component-based development, it allows for defining a hierarchical system
structure consisting of "Actors", which essentially are components. Also the behavior model
7.3. SOFTWARE COMPONENT MODELS

is a hierarchical statechart variant similar to our Real-Time Statecharts (cf. Section 3.3). Like ROOM, earlier versions of MECHATRONIC UML also emphasized object orientation. In the current version of MECHATRONIC UML described in this document, this is not the case anymore. 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.

One of the component models with the highest practical relevance in the domain of embedded real-time systems is AUTOSAR [GbR08], an industry standard for the development of automotive systems. Like MECHATRONIC UML, it defines a concrete development methodology. However, according to [HPB+10] it is the only component model among those considered that does not define any formal behavior model. Also, as it is purely a standard, it offers no tool support. In MECHATRONIC UML these two latter aspects are pivotal.

The ProCom [VSC+09, BCC+08] component model, developed within the Progress research center, also targets 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.

Major characteristics of MECHATRONIC UML are the focus on the development of distributed systems and the ability of dynamic reconfiguration. The latter is not included in this release, but concepts for this exist and their integration is planned. Among the component models covered in [HPB+10], only two offer support for both. These are MyCCM-HI [BFHP09] and SOFA-HI [PWT+08]. MyCCM-HI is based on the Lightweight CCM standard, developed by the OMG. Unlike in MECHATRONIC UML models for MyCCM-HI can currently only be defined using a textual syntax, not in a graphical IDE. These models are then, on the one hand, used for a direct C code generation and, on the other hand, they are transformed into lower-level models for the middleware AADL [BFHP09] that is used.

While SOFA (2.0) is in [CSVC11] classified as a general purpose component model, SOFA-HI [PWT+08] is its extension targeting real-time embedded systems. One target application is the development of software for spacecrafts. It provides extensive reuse capabilities and development support as well as the possibility to change the inner structure of a component at runtime. Although it is a profile of SOFA 2 which supports dynamic reconfiguration exten-
sively, this ability has been restricted in SOFA-HI to make system behavior more predictable. Generating components or new connectors between them is disallowed in SOFA-HI.

7.4. Specifications of Self-Adaptive 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 [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 systems are introduced. These design patterns assume a separation of functional and adaptation behavior which is also used by 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: graph transformation based languages, process algebra based languages, formal logics based languages, and languages that do not fall into one of the three other classes.

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.

7.4.1. Graph Transformation based Approaches

There are several approaches for modeling dynamic software architectures via graph transformations.

The approaches by Le Métayer [LM98] and Hirsch et. al. [HIM98] utilize a context free grammar whose production rules are specified as graph transformations. Le Métayer’s approach requires an additional special coordinator component which executes all system reconfigurations.

The approach by Taentzer et. al. [TGM00] models reconfigurations via general graph transformations.

A special case of a graph transformation based approach is the Chemical Abstract Machine (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 reaction rule.

7.4.2. Process Algebra based Approaches

Approaches which utilize process algebras include Darwin [MK96], LEDA [CPT99], PiLar [CdlFBS01], Dynamic Wright [ADG98], and the approach by Bartels and Kleine [BK11].
Darwin [MK96] is based on the π-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]).

LEDA [CPT99] is also based on the π-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 condition. 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.

PiLar [CdIFBS01, 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 [CdIFBS01]. In addition, PiLar supports replacing a component by a newer version during run-time thereby updating the type and all its instances [CR10]. The adaptation operations are specified by constraints in terms of CCS as well. In contrast to MECHATRONICUML, PiLar mixes component types and instances in different layers of the system. A component instance 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 computations 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.

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 con-
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 MECHATRONIC UML, the framework does not allow to structure a system hierarchically.

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

### 7.4.3. 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 formulas 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 MECHATRONIC UML 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.
7.4.4. Other Approaches

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 a component adaptor module (CAM). The CAM may 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.

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.

7.4.5. 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. In contrast to MECHATRONIC UML, 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.
CHAPTER 7. RELATED WORK

7.5. Formal Models for Modeling of Real-time Behavior

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 [BY03]. In [DMY02b], the timed automata used by UPPAAL are extended to hierarchical timed automata that are closest to the Real-Time Statecharts used in 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.

Furthermore, 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.

Other approaches for modeling real-time behavior include timed process algebras like, e.g., timed CSP [Oxf92].
Chapter 8.

Conclusions and Future Work

In this technical report, we presented a consolidated version of MECHATRONIC UML. 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 MECHATRONIC UML 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 MECHATRONIC UML 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 MECHATRONIC UML 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 MECHATRONIC UML models has been presented in [BGS05]. Furthermore, MECHATRONIC UML 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, MECHATRONIC UML supports a component-based hazard analysis approach [GT06, PST11, PSWTH11]. We will include these topics in upcoming versions of this technical report.

Acknowledgments

MECHATRONIC UML is the result of many people’s work which we are greatly indebted to. MECHATRONIC UML 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 MECHATRONIC UML approach and developed the foundations. Thus, he greatly influenced the current state of MECHATRONIC UML. Additional work has been done by the following doctoral students during their time at the Soft-
ware Engineering Group in Paderborn (in alphabetical order): Sven Burmester [Bur06], Tobias Eckardt, Stefan Henkler, Martin Hirsch [Hir08], Florian Stallmann [Sta08], and Daniela Schilling [Sch06].

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


[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with...
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


9.3. FOREIGN PUBLICATIONS


9.3. FOREIGN PUBLICATIONS


[TGM00] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed


Appendix A.

Technical Reference

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.

![Meta-Model of the modelinstance Package](image)

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.12.2.2 on Page 199

  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:`
  
  ```java
  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.
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 177
  
  The coordination pattern that defines the protocol of this assembly.

- `from : ComponentPart` see Section A.4.2.6 on Page 153
  
  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 153

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

Class Constraints  Class Assembly has the following constraints:

NoSelfAssembliesForSinglePorts:

\[
\text{self.fromPort.cardinality.upperBound.value} \leq 1 \Rightarrow \text{self.fromPort} \neq \text{self.toPort}
\]

ValidContinuousPortDirections:

\[
\text{not self.fromContinuousPort.oclIsUndefined()} \land \text{not self.toContinuousPort.oclIsUndefined()} \Rightarrow \text{self.fromContinuousPort.kind} \neq \text{self.toContinuousPort.kind}
\]

AssemblyBetweenDiscretePortsRequiresCoordinationPattern:

\[
\text{if not self.fromDiscretePort.oclIsUndefined()} \land \text{not self.toDiscretePort.oclIsUndefined()} \text{ then}
\]
\[
\text{not self.coordinationPattern.oclIsUndefined()} \Rightarrow \text{true}
\]
\[
\text{else}
\]
\[
\text{true}
\]
\[
\text{endif}
\]

AssemblyBetweenDiscretePortsRequiresSameCoordinationPattern:

\[
\text{if not self.fromDiscretePort.oclIsUndefined()} \land \text{not self.toDiscretePort.oclIsUndefined()} \text{ then}
\]
\[
\text{not self.fromDiscretePort.refines.oclIsUndefined()} \land \text{not self.toDiscretePort.refines.oclIsUndefined()} \land
\]
\[
\text{self.fromDiscretePort.refines.coordinationPattern} = \text{self.toDiscretePort.refines.coordinationPattern} \Rightarrow \text{true}
\]
\[
\text{else}
\]
\[
\text{true}
\]
\[
\text{endif}
\]

AssemblyBetweenDiscretePortsRequiresDifferentRoles:

\[
\text{if not self.fromDiscretePort.oclIsUndefined()} \land \text{not self.toDiscretePort.oclIsUndefined()} \text{ then}
\]
\[
\text{not self.fromDiscretePort.refines.oclIsUndefined()} \land \text{not self.toDiscretePort.refines.oclIsUndefined()} \land
\]
\[
\text{both ports should have different roles (unless the pattern has only one role)}
\]
(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

Parent Classes

• BehavioralConnector see Section A.4.2.3 on Page 151

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 com-
ponents own a behavior in form of a realtime statechart.

The different component types are implemented as a variation of the composite design pat-
tern. Concerning the composite pattern this class represents the role "leaf".

Class Constraints  Class AtomicComponent has the following constraints:

SoftwareComponentRequiresBehavior:
  self.componentType = component::ComponentKind::SOFTWARE_COMPONENT
  implies (not self.behavior.oclIsUndefined())

Parent Classes

• Component see Section A.4.2.4 on Page 151,

• BehavioralElement see Section A.6.2.3 on Page 166
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 154,
- BehavioralElement see Section A.6.2.3 on Page 166

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 152

  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.4.2.13 on Page 159

  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 153

  This association contains all component parts which have this component as their type.

Class Constraints  Class Component has the following constraints:

- **UniquePortNames**:

  self.ports ->isUnique(name)

- **SoftwareComponentHasOnlyDiscretePorts**:

  use typeOf otherwise hybrid ports are also allowed

  self.componentType = component::ComponentKind::SOFTWARE_COMPONENT

  implies self.ports ->forall(port | port.oclIsTypeOf(component::DiscretePort))
ContinuousComponentHasOnlyContinuousPorts:

−− use typeOf otherwise hybrid ports are also allowed

self.componentType = component::ComponentKind::CONTINUOUS_COMPONENT
implies self.ports ->forall (port | port.typeOf(component::ContinuousPort))

HybridComponentHasOnlyHybridPorts:

self.componentType = component::ComponentKind::HYBRID_COMPONENT
implies self.ports ->forall (port | port.kindOf(component::HybridPort))

Parent Classes

• NamedElement see Section A.12.2.4 on Page 199,

• CommentableElement see Section A.12.2.1 on Page 198,

• ConstrainableElement see Section A.6.2.5 on Page 166

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:
  
  \[
  \text{self.cardinality.upperBound.value} > 1 \text{ or self.cardinality.upperBound.infinity}
  \]

  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.6.2.4 on Page 166

  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 151

  The component type typing this component part.

- `delegation : Delegation [0..*]`  see Section A.4.2.10 on Page 156

  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 147

  The assemblies originating in port parts of this component part.

- `parentComponent : StructuredComponent`  see Section A.4.2.14 on Page 160

  The structured component type containing this component part.

- `/portsDerived : Port [0..*]`  see Section A.4.2.13 on Page 159

  derivation:
  
  \[
  \text{if componentType.oclIsUndefined() then}
  \text{OrderedSet [ ]}
  \text{else}
  \text{componentType.ports}
  \text{endif}
  \]
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 147

The assemblies leading to port parts of this component part.

Class Constraints Class ComponentPart has the following constraints:

CardinalityLowerBoundSet: 
\[ \text{self.cardinality.lowerBound} \rightarrow \text{notEmpty}() \]

TypeNotEqualToParent: 
\[ \text{self.componentType} \not= \text{self.parentComponent} \]

CardinalityUpperBoundSet: 
\[ \text{self.cardinality.upperBound} \rightarrow \text{notEmpty}() \]

Parent Classes

- CommentableElement see Section A.12.2.1 on Page 198

A.4.2.7. Class ConnectorType

Overview This abstract class is the common super class of delegations and assemblies.

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

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:

kind : ContinuousPortDirectionKind see Section A.4.2.9 on Page 155

Decides the direction of a continuous port.

Class References Class ContinuousPort has the following references:
type : EDataType

Defines the data type of the signal value which is emitted or received by the continuous port.

Class Constraints Class ContinuousPort has the following constraints:

LowerBoundMustBeZeroOrOne:

— This Constraint is fulfilled, if no Cardinality exists.
— But that is okay, as then another Problem-Marker is shown,
— because Cardinality.lowerBound is 1..1

\[
\text{self.cardinality.oclIsUndefined()} \text{ or (}
\begin{align*}
\text{false} \\
\text{self.cardinality.lowerBound.oclIsUndefined()} \text{ then (}
\begin{align*}
\text{false} \\
\text{self.cardinality.lowerBound.value} = 0 \text{ or self.cardinality.}
\end{align*}
\text{lowerBound.value} = 1
\end{align*}
\text{endif}
\)\]

UpperBoundMustBeOne:

— This Constraint is fulfilled, if no Cardinality exists.
— But that is okay, as then another Problem-Marker is shown,
— because Cardinality.upperBound is 1..1

\[
\text{self.cardinality.oclIsUndefined()} \text{ or (}
\begin{align*}
\text{false} \\
\text{self.cardinality.upperBound.oclIsUndefined()} \text{ then (}
\begin{align*}
\text{false} \\
\text{self.cardinality.upperBound.value} = 1
\end{align*}
\text{endif}
\)\]

Parent Classes

• Port see Section A.4.2.13 on Page 159

A.4.2.9. Enumeration ContinuousPortDirectionKind

Overview Decides the direction of a continuous port.

Enum Properties Enumeration ContinuousPortDirectionKind has the following literals:

IN = 0

Represent an IN-Port of a continuous port.

OUT = 1

Represent an OUT-Port of a continuous 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 153

The component part of the port part which is connected by this delegation.

Class Constraints Class Delegation has the following constraints:

ValidContinuousPortDirections:

validContinuousPortDirections: not self.fromContinuousPort.oclIsUndefined() and not self.toContinuousPort.oclIsUndefined() implies self.fromContinuousPort.kind = self.toContinuousPort.kind

DelegationBetweenContinuousPortsRequiresSameDataType:

delegationBetweenContinuousPortsRequiresSameDataType: not self.fromContinuousPort.oclIsUndefined() and not self.toContinuousPort.oclIsUndefined() implies self.fromContinuousPort.type = self.toContinuousPort.type

DelegationBetweenDiscretePortsRequiresSameCoordinationPattern:

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:

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
DiscreteMultiPortDelegationRequiresMultiPortOrSinglePortAndMultiPart:

\[
\text{not self.fromDiscretePort.oclIsUndefined()} \text{ and not self.toDiscretePort.oclIsUndefined()} \\text{ and self.fromPort.isMultiPort implies (}
\begin{align*}
\text{--- the target port is a multi port} \\
\text{self.toPort.isMultiPort} \\
\text{or} \\
\text{--- the target part is a multi part} \\
\text{self.componentPart.isMultiPart}
\end{align*}
\]

Parent Classes

- **ConnectorType** see Section A.4.2.7 on Page 154

A.4.2.11. **Class DiscretePort**

**Overview**  This class represents a concrete port specification which provides the discrete functionality of a port.

**Class References**  Class **DiscretePort** has the following references:

- **adaptationBehavior : Behavior [0..1]** see Section A.6.2.2 on Page 166

  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 175

  The receiver message interface defines which messages this discrete port specification receives.

- **refines : Role [0..1]** see Section A.10.2.2 on Page 178

  The role of a coordination pattern that this port refines.
roleAndAdaptationBehavior : Behavior [0..1] see Section A.6.2.2 on Page 166

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 175

The sender message interface defines which messages this discrete port specification sends.

Class Constraints

Class DiscretePort has the following constraints:

AtLeastOneMessageInterface:

\[ \text{selfoclIsTypeOf(component::DiscretePort) implies (not (self.senderMessageInterface.oclIsUndefined() and self.receiverMessageInterface.oclIsUndefined()))} \]

DiscretePortRequiresBehavior:

\[ \text{clarify if this also holds for hybrid ports} \]
\[ \text{not self.behavior.oclIsUndefined()} \]

Parent Classes

- Port see Section A.4.2.13 on Page 159,
- BehavioralElement see Section A.6.2.3 on Page 166

A.4.2.12. Class HybridPort

Overview

This class represents a hybrid port which acts as a bridge between continuous controllers 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 Constraints

Class HybridPort has the following constraints:

LowerBoundMustBeZeroOrOne:

\[ \text{This Constraint is fulfilled, if no Cardinality exists.} \]
\[ \text{But that is okay, as then another Problem–Marker is shown,} \]
\[ \text{because Cardinality.lowerBound is 1..1} \]
\[ \text{self.cardinality.oclIsUndefined()} \text{ or (} \]
\[ \text{if self.cardinality.lowerBound.oclIsUndefined()} \text{ then} \]
false 
else 
  self.cardinality.lowerBound.value = 0 or self.cardinality. 
  lowerBound.value = 1 
endif 
)

**UpperBoundMustBeOne:**

--- This Constraint is fulfilled, if no Cardinality exists.
--- But that is okay, as then another Problem-Marker is shown,
--- because Cardinality.upperBound is 1..1
self.cardinality.oclIsUndefined() or ( 
  if self.cardinality.upperBound.oclIsUndefined() then 
    false 
  else 
    self.cardinality.upperBound.value = 1 
  endif 
)

**Parent Classes**

- DiscretePort see Section A.4.2.11 on Page 157,
- ContinuousPort see Section A.4.2.8 on Page 154

**A.4.2.13. 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:

  self.cardinality.upperBound.value > 1 or self.cardinality. 
  upperBound.infinity

This derived attribute indicates if the port is a multi port (it is only used to simplify OCL constraints).

**Class References** Class Port has the following references:

cardinality : Cardinality see Section A.6.2.4 on Page 166

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 151
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 containment reference (see ComponentPart.portsDerived)

/connectors : ConnectorType [0..∗] see Section A.4.2.7 on Page 154
derivation:
  self.incomingConnectors → union (self.outgoingConnectors)

incomingConnectors : ConnectorType [0..∗] see Section A.4.2.7 on Page 154
The connectors which lead into this port.

outgoingConnectors : ConnectorType [0..∗] see Section A.4.2.7 on Page 154
The connectors which originate from this port.

Parent Classes
• NamedElement see Section A.12.2.4 on Page 199,
• CommentableElement see Section A.12.2.1 on Page 198,
• ConstrainableElement see Section A.6.2.5 on Page 166

A.4.2.14. Class StructuredComponent

Overview This class represents a structured component which is capable of including arbitrarily many component parts.
Component types are implemented as a variation of the composite design pattern. Concerning the composite pattern this class represents the role "composite". However structured components do not contain component types directly like in the original composite pattern. Instead they contain component parts which are typed by component types. The reason for this is to get a clear distinction between the component type level and the component instance level.

Class References Class StructuredComponent has the following references:
connectors : ConnectorType [0..∗] see Section A.4.2.7 on Page 154
The connectors this structured component contains. These can either be delegations or assemblies.
embeddedParts : ComponentPart [1..∗] see Section A.4.2.6 on Page 153
The component parts this structured component contains.
Class Constraints  Class StructuredComponent has the following constraints:

**UniqueComponentPartsWithinStructuredComponent:**

```
self.embeddedParts ->isUnique(p | p.componentType)
```

Parent Classes

- Component see Section A.4.2.4 on Page 151
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 163  
  The correctness of this constraint encoded as a literal of the enum type "Correctness".
Class References  Class Constraint has the following references:

constraintableElement : ConstrainableElement  see Section A.6.2.5 on Page 166

The element this constraint applies to.

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

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 162

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.

Parent Classes

- VerifiableConstraint see Section A.5.2.5 on Page 164.
- ExtendableElement see Section A.12.2.2 on Page 199
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 162
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 statechart 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.

![Meta-Model of the core Package]

A.6.2. Detailed Contents Documentation

A.6.2.1. Class `ActivityCallExpression`

**Overview**  An Expression that represents an activity.
Parent Classes

- Invocation see Section A.15.2.2 on Page 211,
- Expression see Section A.17.2.7 on Page 216

A.6.2.2. **Class Behavior**

**Overview**  Abstract super class for all elements that represent a behavior. Known sub-classes: AbstractRealtimeStatechart

A.6.2.3. **Class BehavioralElement**

**Overview**  Abstract super class for all elements that have a behavior.

A.6.2.4. **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.6.2.6 on Page 167
  The lower bound of this cardinality.

- **upperBound : NaturalNumber**  see Section A.6.2.6 on Page 167
  The upper bound of this cardinality.

**Class Constraints** Class Cardinality has the following constraints:

- **LowerBoundMustBeLessOrEqualThanUpperBound:**
  
  \[
  ((\text{self}.\text{lowerBound}.\text{value} \leq \text{self}.\text{upperBound}.\text{value}) \text{ and } \text{self}.\text{lowerBound}.\text{infinity}!=\text{false} \text{ and } \text{self}.\text{upperBound}.\text{infinity}!=\text{false})
  \text{ or } (\text{self}.\text{lowerBound}.\text{infinity}!=\text{true} \text{ and } \text{self}.\text{upperBound}.\text{infinity}!=\text{true})
  \]

A.6.2.5. **Class ConstrainingElement**

**Overview**  Abstract super class for all model elements that may carry a constraint.
A.6.2.6. **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:**
  
  \[
  \text{self.value} \geq 0
  \]
A.7. Package `muml::model::deployment`

A.7.1. Package Overview

![Diagram of HardwareNode, CommunicationLink, Deployment, and HardwarePort relationships.]

Figure A.5.: Meta-Model of the deployment Package

A.7.2. Detailed Contents Documentation

A.7.2.1. Class CommunicationLink

**Overview**  A target used for the deployment of connectors between component instances.

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

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 nodes.

**Class References**  Class HardwareNode has the following references:

- `deployedInstances : ComponentInstance [0..*]` see Section A.8.2.2 on Page 170

---

¹C. Szyperski, Foreword to Proceedings of Component Deployment, IFIP/ACM Working Conference, Berlin 2002
deployment : Deployment  see Section A.7.2.2 on Page 168
hardwarePorts : HardwarePort [0..*]  see Section A.7.2.4 on Page 169

Class Constraints  Class HardwareNode has the following constraints:

SameConfiguration:

\[
\text{self}.\text{deployedInstance}.\text{componentInstanceConfiguration}=\text{self}.\text{deployment}.\text{componentInstanceConfiguration}
\]

A.7.2.4. Class HardwarePort

Overview  Hardware ports are used to communicate with other hardware nodes and with component instances.
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 147

  derivation:

  ```
  connectorType.oclAsType(component::Assembly)
  ```

  The assembly that this assembly instance is built from.

Parent Classes

- ConnectorInstance see Section A.8.2.4 on Page 172

A.8.2.2. Class `ComponentInstance`

Overview  This class represents a component instance. It is an instantiation of a component.
Figure A.6.: Meta-Model of the instance Package
Parent Classes

- NamedElement see Section A.12.2.4 on Page 199

A.8.2.3. Class ComponentInstanceConfiguration

Overview This class encapsulates represents a configuration. It contains all component instances and connector instances that belong to a concrete configuration.

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

A.8.2.4. Class ConnectorInstance

Overview This class is the common super class of delegation instances and assembly instances.

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

A.8.2.5. 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.8.2.11 on Page 174

A.8.2.6. 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 156

derivation:

connectorType .oclAsType ( component :: Delegation )
The delegation type of this delegation instance.

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

**OneDelegationInstancePerPortInstance:**

\[
\text{not } \text{self.source.oclIsUndefined()} \implies \text{self.source.outgoingConnectorInstances} \rightarrow \text{select}(x \mid x.oclIsKindOf(DelegationInstance)) \rightarrow \text{size}() = 1
\]

**Parent Classes**

- ConnectorInstance see Section A.8.2.4 on Page 172

### A.8.2.7. 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.

**Parent Classes**

- `DiscretePortInstance` see Section A.8.2.8 on Page 173

### A.8.2.8. 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.

**Parent Classes**

- `PortInstance` see Section A.8.2.11 on Page 174

### A.8.2.9. 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.
Parent Classes

- DiscretePortInstance see Section A.8.2.8 on Page 173

A.8.2.10. **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.8.2.8 on Page 173,
- ContinuousPortInstance see Section A.8.2.5 on Page 172

A.8.2.11. **Class PortInstance**

**Overview**  A port instance is a port of a component at instance level.

Parent Classes

- NamedElement see Section A.12.2.4 on Page 199,
- CommentableElement see Section A.12.2.1 on Page 198
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.

![Meta-Model 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 176

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 176

The message types being defined in this message interface.
**superType** : MessageInterface \([0..*]\) see Section A.9.2.1 on Page 175

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:

**NoSelfGeneralization:**  
\[ \text{self.superType} \rightarrow \forall (x | x \not= \text{self}) \]

**NoBidirectionalGeneralization:**  
\[ \text{self.superType} \rightarrow \forall (x | x . \text{superType} \rightarrow \forall (y | y \not= \text{self})) \]

**UniqueMessageTypeNames:**  
\[ \text{self.messageTypes} \rightarrow \text{isUnique}(\text{name}) \]

**NoMessageTypeOrNotAtLeastTwoGeneralizations:**  
\[ \text{self.messageTypes} \rightarrow \text{size}() \geq 1 \text{ or } \text{self.superType} \rightarrow \text{size}() \geq 2 \]

**Parent Classes**

- NamedElement see Section A.12.2.4 on Page 199

**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 175

  This is the message interface where this message type is defined in.

**Class Constraints**  Class MessageType has the following constraints:

**UniqueParameterNames:**  
\[ \text{self.containedParameters} \rightarrow \text{isUnique}(\text{name}) \]

**Parent Classes**

- Callable see Section A.15.2.1 on Page 210,

- NamedElement see Section A.12.2.4 on Page 199
A.10. Package `muml::model::pattern`

A.10.1. Package 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.

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.

![Figure A.8.: Meta-Model of the pattern Package](image-url)
Parent Classes

- NamedElement see Section A.12.2.4 on Page 199,
- ConstrainableElement see Section A.6.2.5 on Page 166

A.10.2.2. Class Role

Overview  This class represents a role of a coordination pattern.

Class Properties  Class Role has the following properties:

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:

adaptationBehavior : Behavior [0..1]  see Section A.6.2.2 on Page 166
The adaptation behavior of this role. Note that only multi-ports have an adaptation
behavior.

cardinality : Cardinality  see Section A.6.2.4 on Page 166
A role has a cardinality.

coordinationPattern : CoordinationPattern  see Section A.10.2.1 on Page 177
The coordination pattern this role belongs to.

incomingRoleConnector : RoleConnector [0..1]  see Section A.10.2.3 on Page 180
The incoming RoleConnector, which connects this role with another role. Either
incomingRoleConnector or outgoingRoleConnector (or both) must be set.

orderVariable : EAttribute [0..1]
This attribute defines the order on the instances of the multi-role. It must be defined
in the real-time statechart of this role and it must be of type integer.

outgoingRoleConnector : RoleConnector [0..1]  see Section A.10.2.3 on Page 180
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 157
The ports this role is assigned to.
receiverMessageInterface : MessageInterface [0..1] see Section A.9.2.1 on Page 175

The receiver message interface defines which messages this port receives.

roleAndAdaptationBehavior : Behavior [0..1] see Section A.6.2.2 on Page 166

/roleConnector : RoleConnector [0..1] see Section A.10.2.3 on Page 180

derivation:
if self.incomingConnector -> notEmpty() then
    self.incomingConnector
else
    self.outgoingConnector
Endif

senderMessageInterface : MessageInterface [0..1] see Section A.9.2.1 on Page 175

The sender message interface defines which messages this port sends.

Class Constraints Class Role has the following constraints:

OrderOnlyForMultiPort:

self.ordered implies (self.cardinality.upperBound.value > 1 or
self.cardinality.upperBound.infinity)

OrderedRequiresIntegerOrderVariable:

self.ordered implies (self.orderVariable->notEmpty() implies self.orderVariable.eAttributeType = 'EInt')

RoleHasConnector:

self.incomingRoleConnector->notEmpty() or self.outgoingRoleConnector->notEmpty()

RoleRequiresBehavior:

not self.behavior.oclIsUndefined()
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 177
  The coordination pattern this role connector is part of.

- `source : Role` see Section A.10.2.2 on Page 178
  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 178
  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`:

  ```
  (not source.oclIsUndefined() and not target.oclIsUndefined())
  implies source.coordinationPattern = target.coordinationPattern
  ```

**Parent Classes**

- BehavioralElement see Section A.6.2.3 on Page 166
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.

**Parent Classes**

- Deadline see Section A.11.2.6 on Page 183

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).

**Parent Classes**

- NamedElement see Section A.12.2.4 on Page 199

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 event 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.

**Parent Classes**

- TransitionEvent see Section A.11.2.28 on Page 197

A.11.2.4. Class `Clock`

**Overview**  This class represents clocks of a realtime statechart.
APPENDIX A. TECHNICAL REFERENCE

Figure A.9.: Meta-Model of the realtimestatechart Package
Parent Classes

- NamedElement see Section A.12.2.4 on Page 199

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.17.2.5 on Page 216
  
  The operator that is used in this clock constraint.

Class References Class ClockConstraint has the following references:

- **bound : NaturalNumber** see Section A.6.2.6 on Page 167
  
  The bound of a deadline (upper or lower) is a natural number.

- **clock : Clock** see Section A.11.2.4 on Page 181
  
  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.

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 : EInt [0..1]**
  
  the lower bound of the period

- **periodUpper : EInt [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 181
  
  Each entry or exit action has one or more actions.
Class Constraints  Class DoEvent has the following constraints:

ValidLowerUpperPeriod:  

\[ \text{self.periodLower} \geq 1 \text{ and } \text{self.periodLower} \leq \text{self.periodUpper} \]

Parent Classes

- StateEvent see Section A.11.2.22 on Page 192

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 184

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.

Parent Classes

- StateEvent see Section A.11.2.22 on Page 192

A.11.2.10. Class EntryPoint

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:  

\[ \text{self.outgoingTransitions} \rightarrow \text{size}() = 1 \]
A.11. PACKAGE Muml::Model::RealtimeStatechart

Parent Classes

- Vertex see Section A.11.2.29 on Page 197

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 185

  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 184
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:**
  
  $$\text{size(self.outgoingTransitions)} \leq 1$$

**Parent Classes**

- Vertex see Section A.11.2.29 on Page 197

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.

**Parent Classes**

- Invocation see Section A.15.2.2 on Page 211

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 | voclIsTypeOf(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.

scheduleDocument : EString [0..1]

needed for WCET-analysis

securityLevel : EInt [0..1]

needed for WCET-analysis

utilisation : EDouble [0..1]

needed for WCET-analysis

Class References Class RealtimeStatechart has the following references:

/allAvailableAttributes : EAttribute [0..*]

derivation:

self -> closure (if embeddingRegion.oclIsUndefined() then self 
else embeddingRegion.parentState.statemachine endif).

attributes ->asSet()

/allAvailableOperations : EOperation [0..*]

derivation:

self -> closure (if embeddingRegion.oclIsUndefined() then self 
else embeddingRegion.parentState.statemachine endif).

operations ->asSet()
Available clocks are all clocks that were defined in this statechart or in ancestor
statecharts.

clocks : Clock [0..*] see Section A.11.2.4 on Page 181

The clocks of this realtime statechart.

embeddingRegion : Region [0..1] see Section A.11.2.18 on Page 189

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.

transitions : Transition [0..*] see Section A.11.2.27 on Page 194

The transitions of the realtime statechart.

vertices : Vertex [0..*] see Section A.11.2.29 on Page 197

The states of this realtime statechart.

Class Constraints Class RealtimeStatechart has the following constraints:

UniqueNameOfStates:

self.vertices -> select (oclIsOfType(State)).oclAsType(State) ->
isUnique(name)

MinOneState:

self.vertices -> select (oclIsOfType(State)).oclAsType(State) ->
notEmpty()

NoCycles:

-- If we are contained within a statechart...
(not self.embeddingRegion.parentState.statechart.oclIsUndefined())

implies

-- ... then we must not be a super statechart of it.
(not self.isSuperStatechartOf(self.embeddingRegion.parentState.
statechart))
A.11. Class Region

Overview Regions enables hierarchy and parallelism. Each state can have zero, one or more regions.

Parent Classes
- NamedElement see Section A.12.2.4 on Page 199,
- Prioritizable see Section A.11.2.16 on Page 186

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 183

A.11.2.20. Class State

Overview This class represents a complex state of a real time statechart. Complex states may again contain real time 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.
\textbf{simple : EBoolean [0..1]}

derivation:
\[
\text{regions} \rightarrow \text{isEmpty ()}
\]

A state is simple if it does not contain a region with an embedded substatechart.

\textbf{urgent : EBoolean [0..1]}

If a state is active and urgent, no time is allowed to pass until the state is leaved.

\textbf{Class References}

\textbf{Class State} has the following references:

\textbf{channels : SynchronizationChannel [0..\ast]} see Section A.11.2.25 on Page 193

The synchronization channels provided by this state.

\textbf{doEvent : DoEvent [0..1]} see Section A.11.2.7 on Page 183

The do event. It is executed periodically while the corresponding state is active.

\textbf{entryEvent : EntryEvent [0..1]} see Section A.11.2.8 on Page 184

The entry action is executed once when the corresponding state is entered.

\textbf{events : StateEvent [0..\ast]} see Section A.11.2.22 on Page 192

derivation:
\[
\text{Set} \{ \text{entryEvent, exitEvent, doEvent} \}
\]

This derived reference returns all StateEvents of this state. The StateEvents of this state are all entry-, do- and exit-Events.

\textbf{exitEvent : ExitEvent [0..1]} see Section A.11.2.13 on Page 185

The exit action is executed once when the corresponding state is left.

\textbf{invariants : ClockConstraint [0..\ast]} see Section A.11.2.5 on Page 183

The invariant belonging to this complex state. It describes how long it is allowed to resides in this complex state depending on the values of the clocks.

\textbf{regions : Region [0..\ast]} see Section A.11.2.18 on Page 189

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.

\textbf{stateEntryPoints : StateEntryPoint [0..\ast]} see Section A.11.2.21 on Page 192

A state references its entry and exit points. They can only exist, if a state embeds one or more statecharts.

\textbf{stateExitPoints : StateExitPoint [0..\ast]} see Section A.11.2.23 on Page 192

\textbf{Class Constraints}

\textbf{Class State} has the following constraints:
OneInvariantPerClock:

\[ \text{self.invariants} \rightarrow \text{isUnique} \left( \text{clock} \right) \]

OneInitialState:

\[ \text{not} \ \text{self.statechart.vertices} \rightarrow \text{select} \left( x \ | \ x.oclIsKindOf \left( \text{State} \right) \right).\]

\[ \text{oclAsType} \left( \text{State} \right) \rightarrow \text{select} \left( s \ | \ s.\text{initial} \rightarrow \text{isEmpty} \right) \]

NoOutgoingTransitionOfFinalState:

\[ \text{self.final} \ \text{implies} \ \text{self.outgoingTransitions} \rightarrow \text{isEmpty} \left( \right) \]

NoRegionsOfFinalState:

\[ \text{self.final} \ \text{implies} \ \text{self.regions} \rightarrow \text{isEmpty} \left( \right) \]

UniquePrioritiesOfOutgoingTransitions:

\[ \text{self.outgoingTransitions} \rightarrow \text{isUnique} \left( \text{priority} \right) \]

UniquePrioritiesOfRegions:

\[ \text{self.regions} \rightarrow \text{isUnique} \left( \text{priority} \right) \]

UniqueChannelNames:

\[ \text{self.channels} \rightarrow \text{isUnique} \left( \text{name} \right) \]

UniqueRegionNames:

\[ \text{self.regions} \rightarrow \text{isUnique} \left( \text{name} \right) \]

BoundOfInvariantGreaterOrEqualZero:

\[ \text{self.invariants} \rightarrow \text{forall} \left( \text{bound.value} \geq 0 \right) \]

InvalidClockConstraintOperator:

\[ \text{self.invariants} \rightarrow \text{forall} \left( \text{invariant} \mid \text{Set} \right) \left( \text{modeling} :: \text{expressions} :: \right) \left( \text{ComparingOperator} :: \text{LESS} , \text{modeling} :: \text{expressions} :: \right) \left( \text{ComparingOperator} :: \text{LESS OR_EQUAL} \right) \rightarrow \text{includes} \left( \text{invariant. operator} \right) \]

Parent Classes

- Vertex see Section A.11.2.29 on Page 197
A.11.2.21. **Class StateEntryPoint**

**Overview**  The StateEntryPoint is assigned to a state. An EntyPoint 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 184
- `state : State` see Section A.11.2.20 on Page 189

**Class Constraints**  Class StateEntryPoint has the following constraints:

- `AtLeastOneIncomingTransition: self.incomingTransitions -> size() > 0`

**Parent Classes**
- Vertex see Section A.11.2.29 on Page 197

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 185

A.11.2.23. **Class StateExitPoint**

**Overview**  The StateExitPoint is assigned to a state. An EntyPoint 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 186
- `state : State` see Section A.11.2.20 on Page 189

**Class Constraints**  Class StateExitPoint has the following constraints:

- `OneOutgoingTransition: self.outgoingTransitions -> size() = 1`
A.11. Parent Classes

- Vertex see Section A.11.2.29 on Page 197

A.11.2.24. Class Synchronization

Overview  Two transitions can synchron 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 194

  Decides the kind: Is this a send or a receive synchronization?

Class References  Class Synchronization has the following references:

  /syncChannel : SynchronizationChannel  see Section A.11.2.25 on Page 193

  derivation:

  callee.oclAsType(SynchronizationChannel)

  the channel that is used by the synchronization

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.

Parent Classes

- Invocation see Section A.15.2.2 on Page 211

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.

Parent Classes

- Callable see Section A.15.2.1 on Page 210,

- NamedElement see Section A.12.2.4 on Page 199
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 181
  - a transition can has one or more absolute deadlines
- **action** : Action [0..1] see Section A.11.2.2 on Page 181
  - 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 183
  - 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 181
  - The clock resets of this transition.
- **events** : TransitionEvent [0..*] see Section A.11.2.28 on Page 197
- **guard** : Expression [0..1] see Section A.17.2.7 on Page 216
  - 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 181
derivation:

\[
\text{self.events} \rightarrow \text{select (e | e.oclIsKindOf(AsynchronousMessageEvent) and e.kind=EventKind::RAISE).oclAsType(AsynchronousMessageEvent)} \rightarrow \text{first ()}
\]

The event which is raised upon activation of this transition.

relativeDeadline : RelativeDeadline [0..1] see Section A.11.2.19 on Page 189

A transition can have one relative deadline

source : Vertex see Section A.11.2.29 on Page 197

The state which is the source of this transition.

statechart : RealtimeStatechart [0..1] see Section A.11.2.17 on Page 186

The realtime statechart this transition belongs to.

synchronization : Synchronization [0..1] see Section A.11.2.24 on Page 193

The synchronisation which is sent upon activation of this transition.

target : Vertex see Section A.11.2.29 on Page 197

The state which is the target of this transition.

/triggerMessageEvent : AsynchronousMessageEvent [0..1] see Section A.11.2.3 on Page 181

derivation:

\[
\text{self.events} \rightarrow \text{select (e | e.oclIsKindOf(AsynchronousMessageEvent) and e.kind=EventKind::TRIGGER).oclAsType(AsynchronousMessageEvent)} \rightarrow \text{first ()}
\]

The trigger event of this transition.

Class Constraints

Class Transition has the following constraints:

SetTargetAndSource:

\[
\text{self.target} \rightarrow \text{notEmpty () and self.source} \rightarrow \text{notEmpty ()}
\]

NoCrossingOfRegionBorders:

\[
\text{self.source.statechart.embeddingRegion= self.target.statechart.embeddingRegion or self.source.oclAsType(StateEntryPoint).statechart.embeddingRegion = self.target.statechart.embeddingRegion.parentState.statechart.embeddingRegion or self.source.statechart.embeddingRegion.parentState.statechart.embeddingRegion = self.target.oclAsType(StateExitPoint).statechart.embeddingRegion}
\]
EntryPointMustOnlyPointToStatesOrStateEntryPoints:

\[
\text{not (self.sourceoclIsKindOf(EntryPoint) and (not self.target.oclIsKindOf(State) and not self.target.oclIsKindOf(StateEntryPoint))})}
\]

ExitPointMustOnlyPointToStatesOrStateExitPoints:

\[
\text{not (self.sourceoclIsKindOf(ExitPoint) and (not self.target.oclIsKindOf(State) and not self.target.oclIsKindOf(StateExitPoint))})}
\]

TriggerMessageEventsMustNotHaveAnOwnedParameterBinding:

\[
\text{not self.triggerMessageEvent.message.oclIsUndefined() implies self.triggerMessageEvent.message.ownedParameterBindings->isEmpty()}
\]

ValidTriggerMessageEvents:

\[
\text{let a : msgface::MessageInterface =}
\]

\[
\text{if statechart.behavioralElement.oclIsKindOf(component::DiscretePort) then}
\]

\[
\text{statechart.behavioralElement.oclAsType(component::DiscretePort).receiverMessageInterface}
\]

\[
\text{else}
\]

\[
\text{if statechart.behavioralElement.oclIsKindOf(pattern::Role) then}
\]

\[
\text{statechart.behavioralElement.oclAsType(pattern::Role).receiverMessageInterface}
\]

\[
\text{else}
\]

\[
\text{null}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{in}
\]

\[
\text{not triggerMessageEvent.message.instanceOf.oclIsUndefined() implies not a.oclIsUndefined() and a.messageTypes->includes(triggerMessageEvent.message.instanceOf)}
\]

ValidRaiseMessageEvents:

\[
\text{let a : msgface::MessageInterface =}
\]

\[
\text{if statechart.behavioralElement.oclIsKindOf(component::DiscretePort) then}
\]

\[
\text{statechart.behavioralElement.oclAsType(component::DiscretePort).senderMessageInterface}
\]

\[
\text{else}
\]

\[
\text{if statechart.behavioralElement.oclIsKindOf(pattern::Role) then}
\]

\[
\text{statechart.behavioralElement.oclAsType(pattern::Role).senderMessageInterface}
\]
else
    null
endif
endif
)
in
not raiseMessageEvent.message.instanceOf.oclIsUndefined() implies
not a.oclIsUndefined() and a.messageTypes->includes(
    raiseMessageEvent.message.instanceOf)

Parent Classes
- ExtendableElement see Section A.12.2.2 on Page 199,
- Prioritizable see Section A.11.2.16 on Page 186

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 185

A.11.2.29. Class Vertex

Overview  This class represents a node in a realtime statechart that is connected with other nodes via transitions.

Parent Classes
- NamedElement see Section A.12.2.4 on Page 199
A.12. Package modeling

A.12.1. Package Overview

The modeling package is the root package for the SDM meta-model. It defines several abstract super classes which implement an extension mechanism as well as reoccurring structural features like, e.g., names of elements. The classes in this package are intended to be sub-classed by any meta-model element.

![Meta-Model of the modeling Package](image)

Figure A.10.: Meta-Model of the modeling Package

A.12.2. Detailed Contents Documentation

A.12.2.1. 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]

**Parent Classes**

- ExtendableElement see Section A.12.2.2 on Page 199
A.12.2.2. Class ExtendableElement

**Overview**  Abstract base class for the whole SDM model. The ExtendableElement specifies the extension mechanism that can be used to extend an object by an Extension containing additional attributes and references.

**Parent Classes**
- EObject

A.12.2.3. Class Extension

**Overview**  Abstract super class for an Extension that can be defined for an object.

**Parent Classes**
- ExtendableElement see Section A.12.2.2 on Page 199

A.12.2.4. 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.12.2.2 on Page 199

A.12.2.5. Class TypedElement

**Overview**  Abstract super class for all meta-model elements that are typed by means of an EClassifier or an EGenericType.

**Parent Classes**
- ExtendableElement see Section A.12.2.2 on Page 199
A.12.2.6. **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:

\[
/\text{variableName} : \text{EString} \ [0..1]
\]

**Parent Classes**

- TypedElement see Section A.12.2.5 on Page 199
A.13. Package modeling::activities

A.13.1. Package Overview

A.13.2. Detailed Contents Documentation

A.13.2.1. 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.

Parent Classes

- Callable see Section A.15.2.1 on Page 210,
- NamedElement see Section A.12.2.4 on Page 199

A.13.2.2. 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).

Parent Classes

- ActivityNode see Section A.13.2.4 on Page 203,
- Invocation see Section A.15.2.2 on Page 211

A.13.2.3. 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.13.2.5 on Page 203

    The guard defines the kind of the activity edge. The possible kinds of guards are specified by the EdgeGuard enum.
Figure A.11.: Meta-Model of the activities Package
Class References  Class ActivityEdge has the following references:

guardException : ExceptionVariable [0..*] see Section A.13.2.6 on Page 205
Declares variables representing the Exceptions that lead to firing this transition.

guardExpression : Expression [0..1] see Section A.17.2.7 on Page 216
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.13.2.1 on Page 201
Points to the activity this ActivityEdge is contained in.

source : ActivityNode see Section A.13.2.4 on Page 203
The source node of this ActivityEdge.

target : ActivityNode see Section A.13.2.4 on Page 203
The target node of this ActivityEdge.

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

A.13.2.4. 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.

Parent Classes

- NamedElement see Section A.12.2.4 on Page 199,
- CommentableElement see Section A.12.2.1 on Page 198

A.13.2.5. Enumeration EdgeGuard

Overview  This enum is used to model different kinds of activity edges.

Enum Properties  Enumeration EdgeGuard has the following literals:

NONE = 0

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.
SUCCESS = 1

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.

FAILURE = 2

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.

EACH_TIME = 3

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.

END = 4

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.

ELSE = 5

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.

BOOL = 6

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.

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.13.2.6. **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.13.2.3 on Page 201
  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.12.2.6 on Page 200

A.13.2.7. **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.13.2.4 on Page 203

A.13.2.8. **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.

**Parent Classes**

- StoryNode see Section A.13.2.14 on Page 207
A.13.2.9. **Class ModifyingStoryNode**

**Overview** A ModifyingStoryNode contains a story pattern which may change the underlying graph upon execution.

**Parent Classes**
- StoryNode see Section A.13.2.14 on Page 207

A.13.2.10. **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.

**Parent Classes**
- Extension see Section A.12.2.3 on Page 199,
- Callable see Section A.15.2.1 on Page 210

A.13.2.11. **Class StartNode**

**Overview** The start node of an activity defines the starting point for the execution of the activity.

**Parent Classes**
- ActivityNode see Section A.13.2.4 on Page 203

A.13.2.12. **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.

**Parent Classes**
- ActivityNode see Section A.13.2.4 on Page 203
A.13.2.13. **Class StopNode**

**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 StopNode has the following properties:

- **flowStopOnly : EBoolean**
  
  true if subactivity is stopped, but not the whole control flow

**Class References**  Class StopNode has the following references:

- **/returnValue : Expression [0..1]**  see Section A.17.2.7 on Page 216
  
  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.17.2.7 on Page 216
  
  Defines the return values of the activity. These return values will be assigned to the out-parameters.

**Parent Classes**

- ActivityNode see Section A.13.2.4 on Page 203

A.13.2.14. **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.18.2.18 on Page 227

**Parent Classes**

- ActivityNode see Section A.13.2.4 on Page 203

A.13.2.15. **Class StructuredNode**

**Overview**  A structured node is a node that contains several other activities.
Parent Classes

- ActivityNode see Section A.13.2.4 on Page 203
A.14. Package
  modeling::activities::expressions

A.14.1. Package Overview

A.14.2. Detailed Contents Documentation

A.14.2.1. Class ExceptionVariableExpression

Overview  Represents the value of an exception variable declared as a transition guard (the guard of an activity edge).

Parent Classes

  • Expression see Section A.17.2.7 on Page 216
## A.15. Package `modeling::calls`

### A.15.1. Package Overview

This package contains all classes for modeling calls to activities and EOperations from within an activity.

![Figure A.12.: Meta-Model of the calls Package](image)

### A.15.2. Detailed Contents Documentation

#### A.15.2.1. 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).

**Parent Classes**

- CommentableElement see Section A.12.2.1 on Page 198
A.15.2.2. **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 which are contained in the model (i.e. Activities and OperationExtensions) the Invocation directly points to the callee. OpaqueCallables are directly referenced by (and contained in) the MethodCallExpressions.

**Parent Classes**
- CommentableElement see Section A.12.2.1 on Page 198

A.15.2.3. **Class OpaqueCallable**

**Overview**  An OpaqueCallable represents an external method which is not explicitly modeled (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.16.2.1 on Page 213

**Parent Classes**
- Callable see Section A.15.2.1 on Page 210

A.15.2.4. **Class ParameterBinding**

**Overview**  Binds a parameter to a certain value for a given invocation. The value of the parameter is represented by an expression.

**Parent Classes**
- CommentableElement see Section A.12.2.1 on Page 198

A.15.2.5. **Class ParameterExtension**

**Overview**  Represents an EParameter and adds functionality to it, especially being subtype of Variable.
Parent Classes

- Variable see Section A.12.2.6 on Page 200,
- Extension see Section A.12.2.3 on Page 199
A.16. Package modeling::calls::expressions

A.16.1. Package Overview

A.16.2. Detailed Contents Documentation

A.16.2.1. 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.

Parent Classes

- Expression see Section A.17.2.7 on Page 216,
- Invocation see Section A.15.2.2 on Page 211

A.16.2.2. Class ParameterExpression

Overview An Expressions that represents a parameter value, e.g. the value of an Activity’s parameter.

Parent Classes

- Expression see Section A.17.2.7 on Page 216
A.17. Package `modeling::expressions`

A.17.1. Package Overview

The base package for all expressions which can be used for modeling activities and patterns.

![Meta-Model of the expressions Package](image)

Figure A.13.: Meta-Model of the expressions Package

A.17.2. Detailed Contents Documentation

A.17.2.1. Class `ArithmeticExpression`

Overview    Represents arithmetic expressions like a + 5 or a * 7.

Class Properties    Class `ArithmeticExpression` has the following properties:

- **operator**: `ArithmeticOperator`    see Section A.17.2.2 on Page 215

  Specifies the expression’s arithmetic operator, e.g. +, -, *, /, or MODULO.
Parent Classes

- BinaryExpression see Section A.17.2.3 on Page 215

### A.17.2.2. 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
- EXP = 5

For formulas like a^b.

### A.17.2.3. Class BinaryExpression

**Overview** Represents any binary expression like \( v < 5 \) or \( x + 7 \).

**Parent Classes**

- Expression see Section A.17.2.7 on Page 216

### A.17.2.4. Class BinaryLogicExpression

**Overview** Represents binary, logic expressions like \( a \ AND \ b \) and \( a \ OR \ b \).

**Class Properties** Class BinaryLogicExpression has the following properties:

- `operator : LogicOperator` see Section A.17.2.9 on Page 217

  Specifies the expression’s logic operator, e.g. AND, OR, or XOR.

**Parent Classes**

- BinaryExpression see Section A.17.2.3 on Page 215
A.17.2.5. **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.17.2.6. **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.17.2.5 on Page 216
  Specifies the expression’s comparing operator, e.g. <, >=, !=.

**Parent Classes**

- BinaryExpression see Section A.17.2.3 on Page 215

A.17.2.7. **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**

- TypedElement see Section A.12.2.5 on Page 199.
- CommentableElement see Section A.12.2.1 on Page 198
A.17.2.8. **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".

**Class References**  Class LiteralExpression has the following references:

- **valueType : EDataType**
  
  The literal’s type, e.g. EInt, EString, etc.

**Parent Classes**

- Expression see Section A.17.2.7 on Page 216

A.17.2.9. **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.17.2.10. **Class NotExpression**

**Overview**  Represents a negated expression, e.g. NOT (a < 5).

**Parent Classes**

- Expression see Section A.17.2.7 on Page 216
A.17.2.11. **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.17.2.7 on Page 216
A.18. Package modeling::patterns

A.18.1. Package Overview

This package contains all classes for modeling story patterns that may be embedded into StoryActivityNodes of an Activity.

A.18.2. Detailed Contents Documentation

A.18.2.1. 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.18.2.4 on Page 221

  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.18.2.5 on Page 222

  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).

  bindingState : BindingState  see Section A.18.2.6 on Page 222

  The binding state defines whether the link is already bound or whether a match has to be obtained for it.

Class References  Class AbstractLinkVariable has the following references:

  firstLinkConstraint : LinkConstraint [0..*]  see Section A.18.2.10 on Page 224

  pattern : StoryPattern  see Section A.18.2.18 on Page 227

  secondLinkConstraint : LinkConstraint [0..*]  see Section A.18.2.10 on Page 224

  source : ObjectVariable  see Section A.18.2.15 on Page 226

  target : AbstractVariable  see Section A.18.2.2 on Page 221

Parent Classes

  • NamedElement see Section A.12.2.4 on Page 199
Figure A.14.: Meta-Model of the patterns Package
A.18.2.2. Class AbstractVariable

Overview  Abstract super class for object and primitive variables.

Class Properties  Class AbstractVariable has the following properties:
  
  bindingState : BindingState  see Section A.18.2.6 on Page 222  
  
  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.17.2.7 on Page 216  
  
  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.18.2.7 on Page 223  
  
  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.18.2.1 on Page 219  

  pattern : StoryPattern  see Section A.18.2.18 on Page 227

Parent Classes

- Variable see Section A.12.2.6 on Page 200,
- NamedElement see Section A.12.2.4 on Page 199

A.18.2.3. 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.

A.18.2.4. 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 be 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.18.2.5. 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.18.2.6. 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.18.2.7. 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.

A.18.2.8. Class ContainerVariable

Overview  Represents a single container, e.g. a Set or List. ContainmentRelations can be used to add or remove objects to or from this container. Every Constraint or AttributeAssignment can use the variable as a container (e.g., "set->size() > 5").

Parent Classes

- ObjectVariable see Section A.18.2.15 on Page 226

A.18.2.9. Class ContainmentRelation

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.
APPENDIX A. TECHNICAL REFERENCE

Parent Classes

- AbstractLinkVariable see Section A.18.2.1 on Page 219

A.18.2.10. 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.18.2.11 on Page 224
  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.18.2.1 on Page 219
- referencingObject : ObjectVariable see Section A.18.2.15 on Page 226
- secondLink : AbstractLinkVariable [0..1] see Section A.18.2.1 on Page 219

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

A.18.2.11. 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`
- `DIRECT_SUCCESSOR = 2`
- `INDIRECT_SUCCESSOR = 3`
- `INDEX = 4`

### A.18.2.12. 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 supports 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.

**Parent Classes**
- `AbstractLinkVariable` see Section A.18.2.1 on Page 219

### A.18.2.13. 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.18.2.18 on Page 227

### A.18.2.14. Class `ObjectSetVariable`

**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.

**Parent Classes**
- `ObjectVariable` see Section A.18.2.15 on Page 226
A.18.2.15. **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.18.2.4 on Page 221
  
  The binding operator defines whether this object will be matched, created or destroyed by the story pattern.

- **bindingSemantics** : BindingSemantics  see Section A.18.2.5 on Page 222
  
  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.18.2.3 on Page 221

- **classifier** : EClass
  
  The type of this ObjectVariable, given as an EClass.

- **linkOrderConstraint** : LinkConstraint [0..*]  see Section A.18.2.10 on Page 224

- **outgoingLink** : AbstractLinkVariable [0..*]  see Section A.18.2.1 on Page 219

**Parent Classes**

- AbstractVariable see Section A.18.2.2 on Page 221

A.18.2.16. **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.

**Parent Classes**

- AbstractLinkVariable see Section A.18.2.1 on Page 219
A.18.2.17. **Class** *PrimitiveVariable*

**Overview**  Represents a variable that holds a value of a primitive type, e.g. integer, boolean, String.

**Parent Classes**
- AbstractVariable see Section A.18.2.2 on Page 221

A.18.2.18. **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.18.2.5 on Page 222

**Class References**  Class *StoryPattern* has the following references:
- **constraint : Constraint [0..*]**  see Section A.18.2.7 on Page 223
  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.18.2.18 on Page 227
- **linkVariable : AbstractLinkVariable [0..*]**  see Section A.18.2.1 on Page 219
- **parentPattern : StoryPattern [0..1]**  see Section A.18.2.18 on Page 227
- **templateSignature : TemplateSignature [0..1]**  see Section A.20.2.3 on Page 229
- **variable : AbstractVariable [0..*]**  see Section A.18.2.2 on Page 221

**Parent Classes**
- CommentableElement see Section A.12.2.1 on Page 198
A.19. Package `modeling::patterns::expressions`

A.19.1. Package Overview

A.19.2. Detailed Contents Documentation

A.19.2.1. Class `AttributeValueExpression`

**Overview**  Represents the value of an object’s attribute, e.g. `obj.attr` for an object `obj` and an attribute `attr`.

**Parent Classes**
- Expression see Section A.17.2.7 on Page 216

A.19.2.2. Class `ObjectSetSizeExpression`

**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 apply the pattern when at least two objects can be matched for the set.

**Parent Classes**
- Expression see Section A.17.2.7 on Page 216

A.19.2.3. Class `ObjectVariableExpression`

**Overview**  Represents the reference to an object in an expression, i.e. the value of an object variable.

**Parent Classes**
- Expression see Section A.17.2.7 on Page 216

A.19.2.4. Class `PrimitiveVariableExpression`

**Overview**  Represents the value of a primitive variable, e.g., 5 or "MyName".

**Parent Classes**
- Expression see Section A.17.2.7 on Page 216
A.20. Package modeling::templates

A.20.1. Package Overview

Figure A.15.: Meta-Model of the templates Package

A.20.2. Detailed Contents Documentation

A.20.2.1. Class PropertyBinding

Overview

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

A.20.2.2. Class TemplateBinding

Overview

Parent Classes

- ExtendableElement see Section A.12.2.2 on Page 199

A.20.2.3. Class TemplateSignature

Overview