MechatronicUML – Syntax and Semantics[1] [2]

Technical report

tr-ri-11-325

Steffen Becker, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann,
Claudia Priesterjahn, Wilhelm Schäfer, Oliver Sudmann, Matthias Tichy
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[stbecker|xell|cytom|c.heinzemann|upohl|cpr|wilhelm|oliversu|mtt]@mail.uni-paderborn.de

Version: 0.1

Paderborn, August 4, 2011

[1]This 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.

[2]This 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**
1.1. Example . . . 2

2. Concepts **5**
2.1. Development Process . . . 5
2.1.1. Integration into the Development Process for Advanced Mechatronic Systems . . . 5
2.1.2. The MECHATRONICUML Process . . . 7
2.2. Real-Time Coordination Pattern . . . 19
2.2.1. Real-Time Coordination Pattern . . . 19
2.2.2. Role . . . 20
2.2.3. Communication Connector . . . 22
2.2.4. Pattern Instantiation . . . 22
2.2.5. Role Behavior . . . 23
2.2.6. Verification Property . . . 25
2.3. Message Interface Specification . . . 26
2.3.1. Message Interface . . . 26
2.3.2. Message Type . . . 27
2.4. Real-Time Statechart . . . 28
2.4.1. Real-Time Statechart . . . 28
2.4.2. State . . . 29
2.4.3. Region . . . 32
2.4.4. Transition . . . 32
2.4.5. Entry-/Exit-Point . . . 35
2.4.6. Shallow History . . . 36
2.4.7. Action . . . 36
2.4.8. Clock . . . 37
2.4.9. Synchronization . . . 37
2.4.10. Asynchronous Message-Event . . . 42
2.5. Component Model . . . 43
2.5.1. Atomic Component Type . . . 43
2.5.2. Structured Component Type . . . 51
2.6. Component Instance Configuration . . . 60
2.6.1. Component Instance . . . 60

2.6.2. Component Instance Configuration 60

3. Complete Example **65**
3.1. Real-Time Coordination Patterns 65
3.1.1. Navigation Pattern 65
3.1.2. Delegation Pattern 67
3.1.3. Distribution Pattern 68
3.1.4. PositionTransmission Pattern 71
3.1.5. DistancesTransmission Pattern 72
3.2. Message Interface Specification 74
3.2.1. Navigation Pattern 74
3.2.2. Delegation Pattern 74
3.2.3. Distribution Pattern 75
3.2.4. PositionTransmission Pattern 75
3.2.5. DistancesTransmission Pattern 76
3.3. System Structure 76
3.3.1. Structured Components 76
3.3.2. Atomic Components 79
3.4. Real-Time Statechart 82
3.4.1. Exploration 82
3.4.2. Navigation 84
3.4.3. BeBot Observer 86
3.4.4. Collision Control 88
3.5. Component Instance Configuration 91
3.5.1. Single BeBot 91
3.5.2. Networks of BeBots 92

4. Related Work **97**
4.1. Specification Languages for Systems Engineering 97
4.2. Process Models for System Engineering 98
4.3. Software Component Models 98
4.4. Specifications of reconfigurable systems 99
4.5. Formal Models for Modeling of Real-time Behavior 99

5. Conclusions and Future Work **101**

Bibliography **103**

A. Technical Reference **113**
A.1. Package `modelinstance` 113
A.1.1. Package Overview 113
A.1.2. Detailed Contents Documentation 113

A.2. Package **muml** 115
A.2.1. Package Overview 115
A.3. Package **muml::model** 115
A.3.1. Package Overview 115
A.4. Package **muml::model::component** 115
A.4.1. Package Overview 115
A.4.2. Detailed Contents Documentation 115
A.5. Package **muml::model::constraint** 125
A.5.1. Package Overview 125
A.5.2. Detailed Contents Documentation 125
A.6. Package **muml::model::core** 128
A.6.1. Package Overview 128
A.6.2. Detailed Contents Documentation 129
A.7. Package **muml::model::instance** 131
A.7.1. Package Overview 131
A.7.2. Detailed Contents Documentation 131
A.8. Package **muml::model::msgiface** 133
A.8.1. Package Overview 133
A.8.2. Detailed Contents Documentation 133
A.9. Package **muml::model::pattern** 135
A.9.1. Package Overview 135
A.9.2. Detailed Contents Documentation 135
A.10. Package **muml::model::realtimestatechart** 138
A.10.1. Package Overview 138
A.10.2. Detailed Contents Documentation 138
A.11. Package **modeling** 153
A.11.1. Package Overview 153
A.11.2. Detailed Contents Documentation 153
A.12. Package **modeling::activities** 156
A.12.1. Package Overview 156
A.12.2. Detailed Contents Documentation 156
A.13. Package **modeling::activities::expressions** 164
A.13.1. Package Overview 164
A.13.2. Detailed Contents Documentation 164
A.14. Package **modeling::calls** 165
A.14.1. Package Overview 165
A.14.2. Detailed Contents Documentation 165
A.15. Package **modeling::calls::expressions** 168
A.15.1. Package Overview 168
A.15.2. Detailed Contents Documentation 168
A.16. Package **modeling::expressions** 169
A.16.1. Package Overview 169

A.16.2. Detailed Contents Documentation 169
A.17. Package **modeling::patterns** 174
A.17.1. Package Overview 174
A.17.2. Detailed Contents Documentation 174
A.18. Package **modeling::patterns::expressions** 183
A.18.1. Package Overview 183
A.18.2. Detailed Contents Documentation 183
A.19. Package **modeling::templates** 184
A.19.1. Package Overview 184
A.19.2. Detailed Contents Documentation 184

Chapter 1.

Introduction

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

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

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

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

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

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

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

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

A major aim of 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, 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 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 MECHATRONICUML.

This technical report consolidates the older publications [Bur02, Gie03, GB03, GST+03, GTB+03, Hir04, BGT05, Bur06, GHH+08, Hir08, EHH+11, SSGR11] in a single document. We present the current version of MECHATRONICUML in detail and give formal specifications for abstract and concrete syntax as well as an informal description of its semantics.

Chapter 2 describes informally the syntax and the semantics of MECHATRONICUML based on the running example which is presented in the next section. The complete models of the running example are presented in Chapter 3. After a discussion of related approaches in Chapter 4, we conclude with an outlook on future work in Chapter 5. Appendix A contains a thorough definition of the abstract syntax.

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].

[1] In the remainder of this document, we will refer to component types simply as components for the sake of easier readability (cf. Section 2.5).

[2] http://wwwhni.uni-paderborn.de/en/priority-projects/intelligent-miniature-robot-bebot

[3] http://wwwhni.uni-paderborn.de/en/

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

Figure 1.1.: BeBot

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 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.

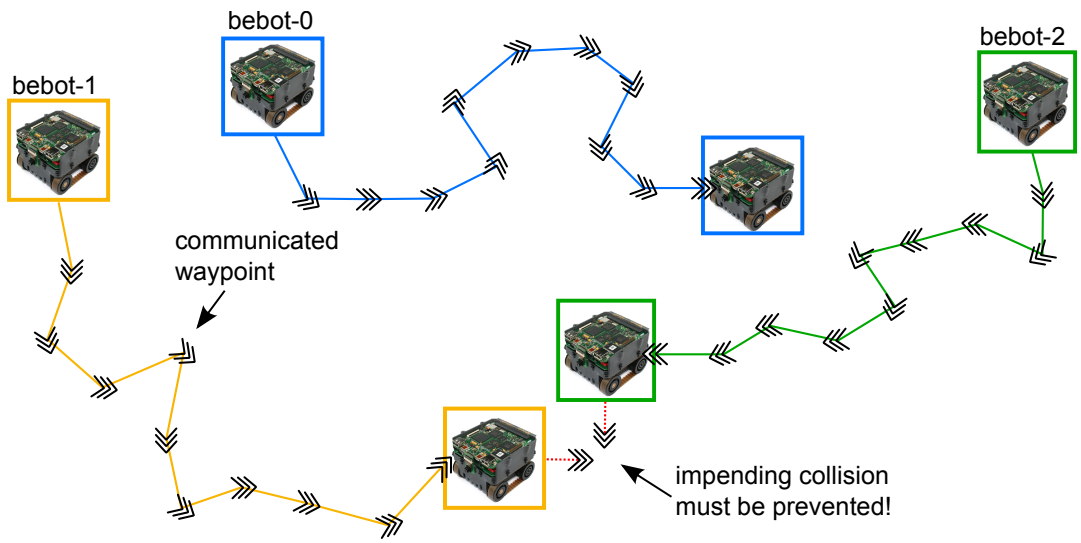


Figure 1.2.: Area of the Exploration scenario

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.

Concepts

In this chapter, we will introduce the main concepts of the MECHATRONICUML. We will start by introducing the development process of the MECHATRONICUML approach as an overview. Afterwards, we will introduce the different modeling formalisms of the MECHATRONICUML in more detail. The 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. The types of messages that may be exchanged between components are typed by means of message interfaces, the behavior of the communicating entities is specified by Real-Time Statechart, an extension of UML Statemachines [Obj09] by clocks as known from timed automata [AD94]. Finally, the MECHATRONICUML provides an instance model to specify concrete system configurations.

2.1. 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 2.1.1. Afterwards, the MECHATRONICUML process is described in detail in Section 2.1.2.

2.1.1. 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 *interdiciplinary conceptual design*, the parallel *discipline-specific development*, and the *system integration*. The first two phases are shown in Figure 2.1. 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.

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.

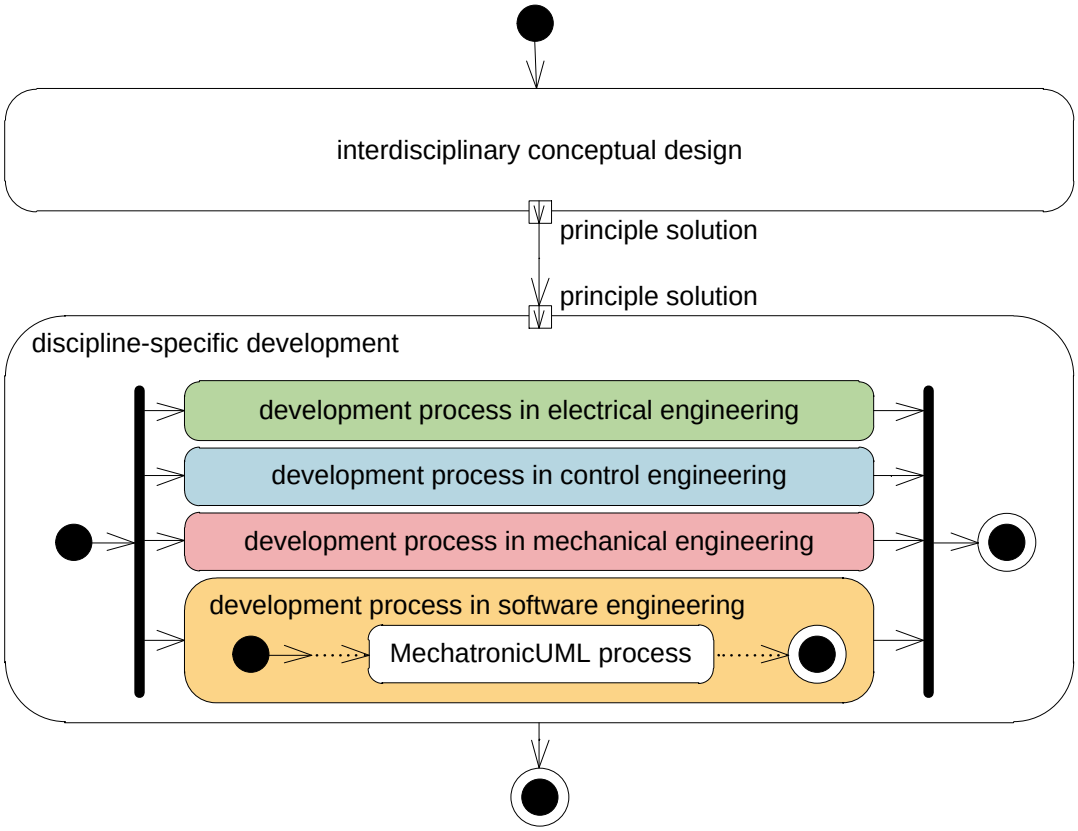


Figure 2.1.: The Integration of the MECHATRONICUML Process into the Development Process of Advanced Mechatronic Systems

2.1.2. The MECHATRONICUML Process

Figure 2.2 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 compo-

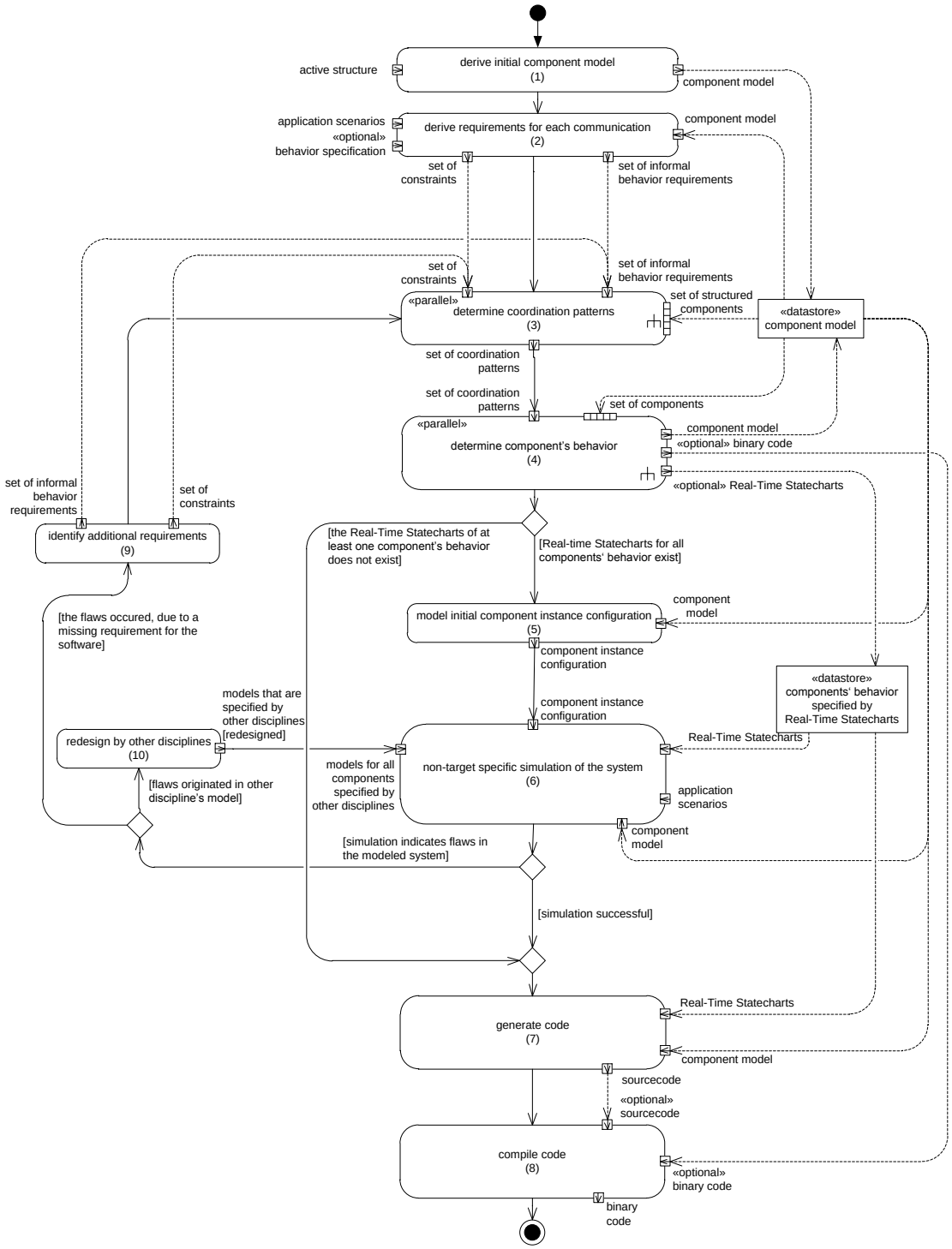


Figure 2.2.: Overall MECHATRONICUML Process

nent consist of parts that are typed by other components (cf. Section 2.5.2.1). An atomic component has a behavior specification and cannot embed any parts (cf. Section 2.5.1).

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 interdependent 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 lifeness properties of the protcol. For each particpant 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 2.2). In Step 3, the Real-Time Coordination Patterns for each connector of the structured components' parts are determined as described in detail in Section 2.1.2.1 and Section 2.1.2.2. 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 2.1.2.3: 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 2.1.2.3) [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 2.1.2.4). 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 2.1.2.4). 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 saftey 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 2.6.

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 relation (Step 9). The development process is repeated for all depending components starting with the determination of the coordination pattern (Step 3).

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

2.1.2.1. 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 2.3, 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 reuseable 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 saftety constraints are derived from the informal constraints, it is ensured that the communication behavior fullfills these safety constraints. Step 3.2 is described in more detail in Section 2.1.2.2.

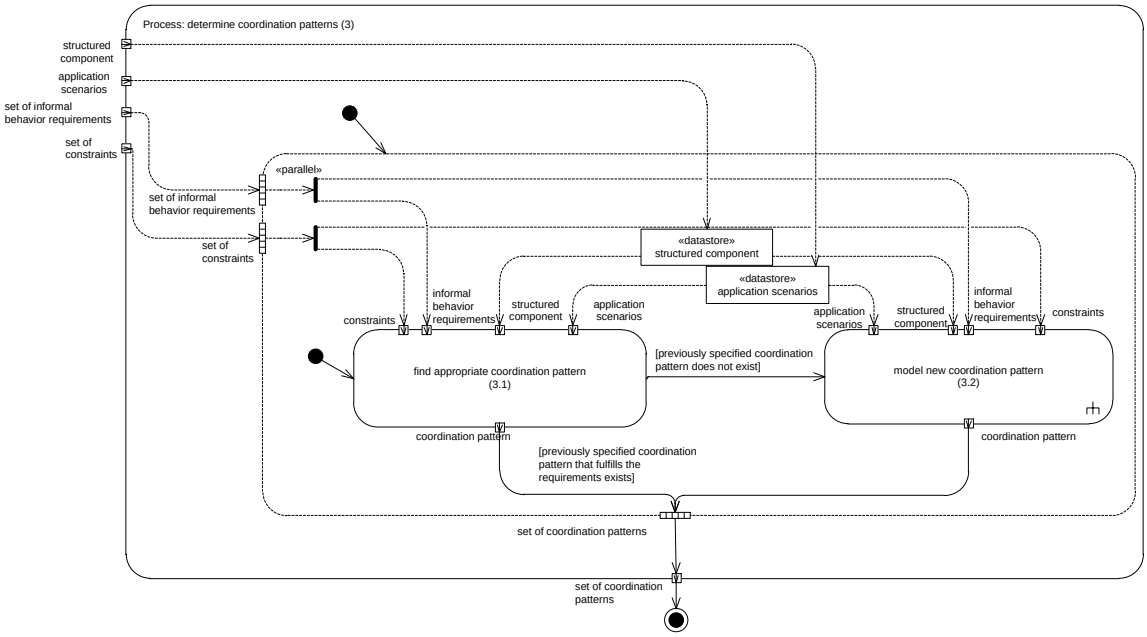


Figure 2.3.: The Subprocess to Reuse or Model a new Coordination Pattern for all Communications within one Structured Component

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].

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

The steps to model a Real-Time Coordination Pattern are shown in Figure 2.4 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 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.

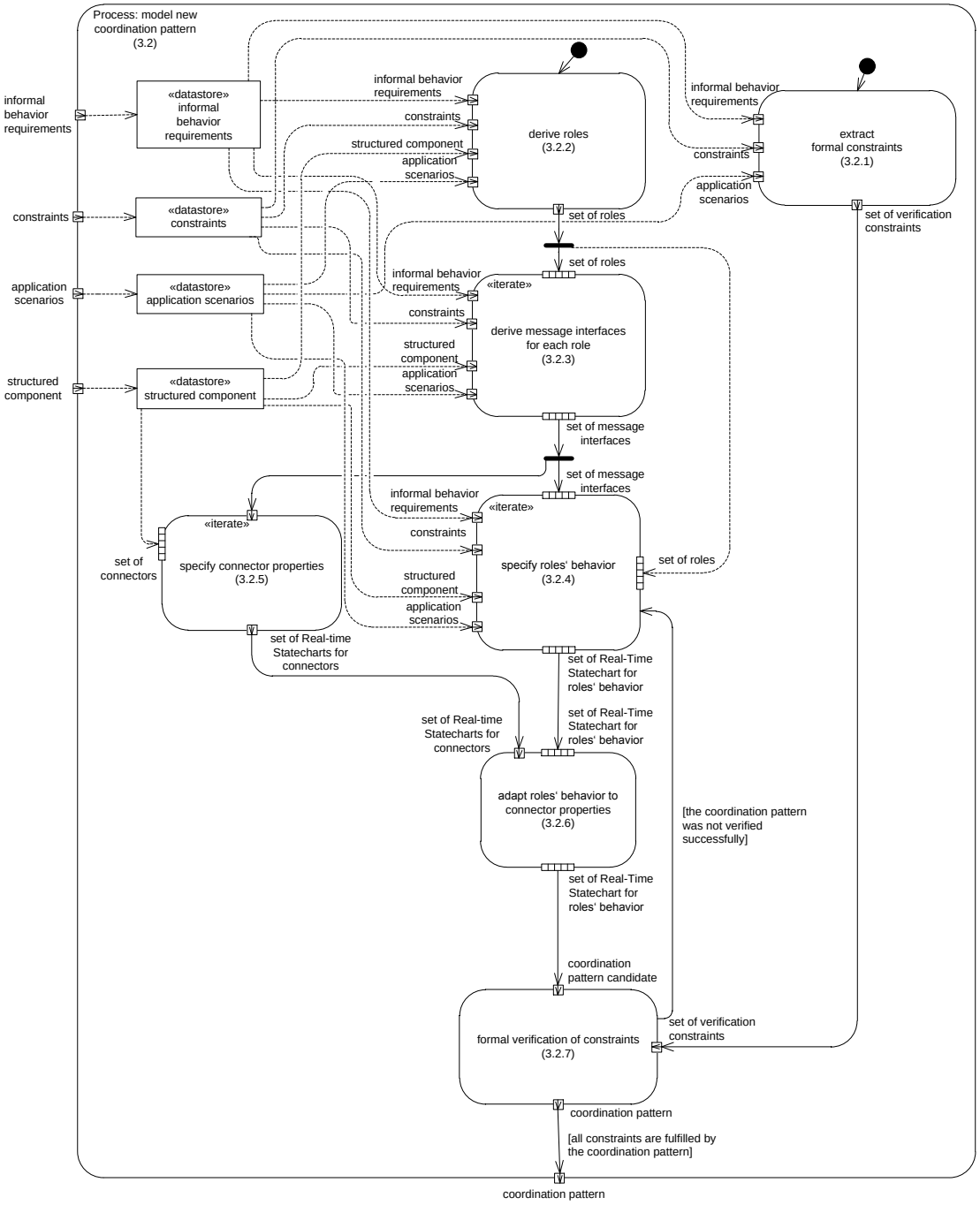


Figure 2.4.: The Subprocess to Model a new Coordination Pattern

For each role, the message interface (cf. Section 2.3) 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-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 communciation channel are specified. In mechatronic systems, the communication of two components may be unreliable. For instance, messages that are transfered through a wirless 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 verfied 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.

2.1.2.3. Determination of the Components' Behavior (Step 4)

The behavior of a component can be determined in three different ways as described in Section 2.1.2. The detailed steps for these alternative ways to determine the behavior of a component are highlighted by different colors in Figure 2.5. 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

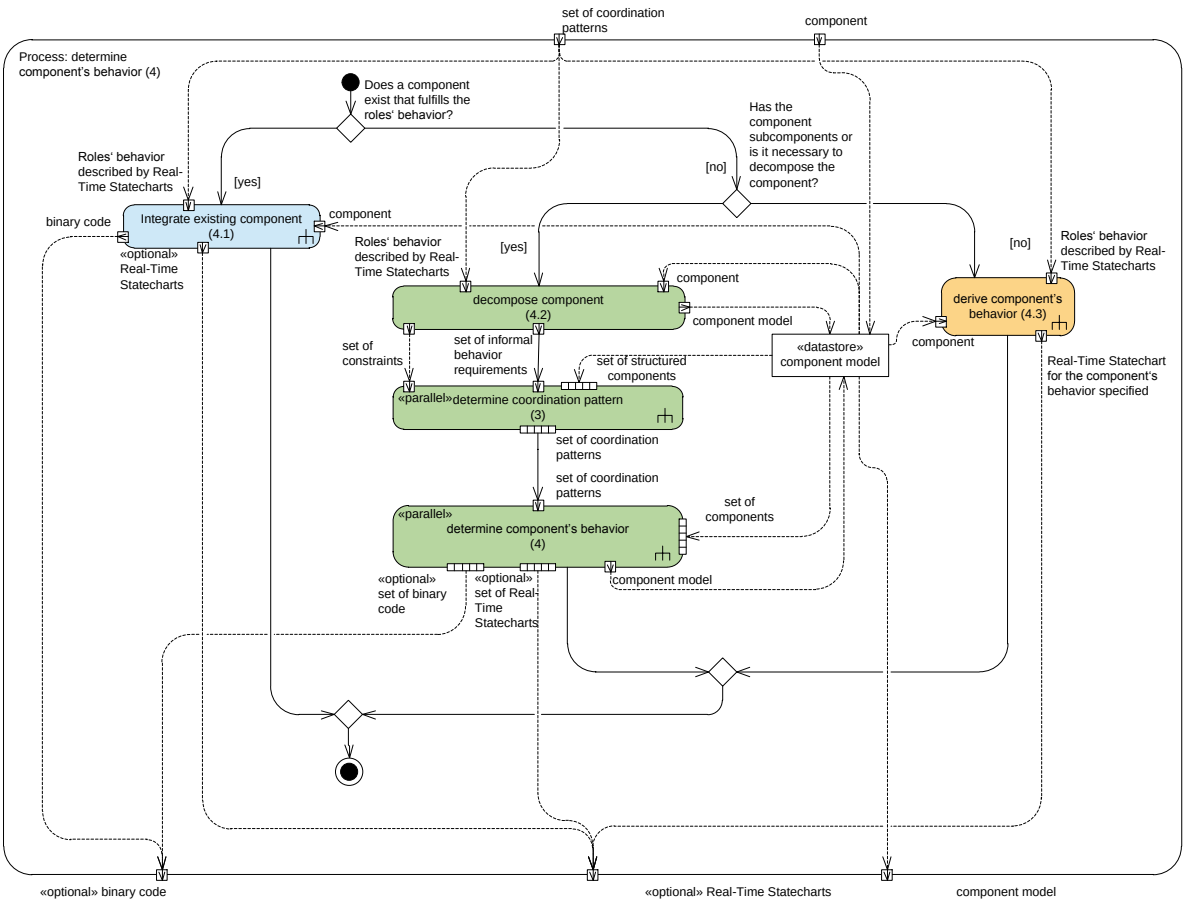


Figure 2.5.: The Subprocess to Determine the Internal Structure or Behavior of a Component

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

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

If the behavior is derived directly, Step 4.3 is performed. A detailed explanation is given in Section 2.1.2.4. 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 Coor-

dination Pattern are still fulfilled for the refined behavior of the component [GTB+03, HH11]. According to these refinement rules, additional behavior is added such as additional messages for the internal communication within the component. These additional messages are necessary to synchronize the behavior of 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 2.5.2). These parts represent the subcomponents that are added to the component (cf. Section 2.5.2.3). The ports of the component are delegated to ports of the parts (cf. Section 2.5.2.4). 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 MECHATRONICUML process (cf. Section 2.1.2). 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 MECHATRONICUML 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.

2.1.2.4. 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 2.5.1.3). The goal of Step 4.3 is to derive a Real-Time Statechart for the component's behavior (cf. Section 2.5.1.4). Figure 2.6 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 cor-

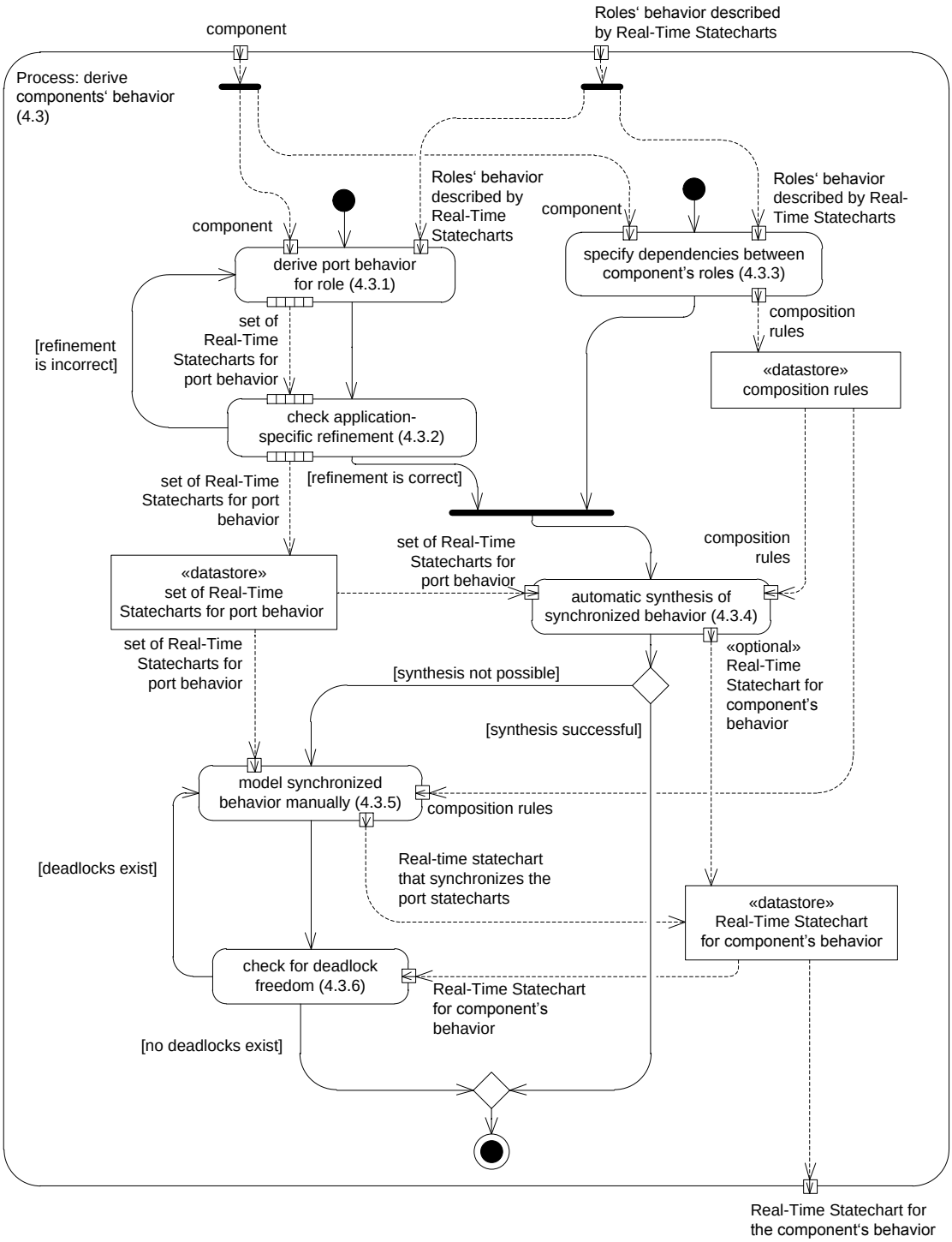


Figure 2.6.: The Subprocess to Model the Behavior of a Component

responding Real-Time Coordination Pattern is parameterized, it may be necessary to identify the appropriate parameters for the application. Furthermore, application specific refinements such as actions for side-effects, additional internal messages, or states must be added. These refinements can change the behavior of the roles and can, therefore, violate the safety constraints of the Real-Time Coordination Pattern. Thus, in Step 4.3.2 the correctness according to the approach proposed by HEINZEMANN et al. is ensured [HH11]. The result of these two steps is an independent parallel composition of Real-Time Statecharts.

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

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

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

At last, the refined roles' Real-Time Statecharts and the synchronization statechart are combined to one Real-Time Statechart (cf. Section 2.5.1.4). 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.

2.2. Real-Time Coordination Pattern

MECHATRONICUML partitions the component behavior into internal and communication behavior. Real-Time Coordination Patterns specify the communication behavior that software component instances have to fulfill for communicating with each other. Furthermore, they take real-time dependencies during the communication into account.

A Real-Time Coordination Pattern specifies the message- and state-based communication of exactly two kinds of communication partners, e. g., master and slave, which are referred to as roles (cf. Section 2.2.2). In a Real-Time Coordination Pattern, the role instances are communicating with each other using a communication connector (c.f. Section 2.2.3). The developer has to instantiate a Real-Time Coordination Pattern for specifying a concrete communication between two or more component instances (cf. Section 2.2.4) and link the communication behavior for this communication to the remaining behavior of the component (cf. Section 2.5). The behavior of each role may be described using Real-Time Statecharts (cf. Section 2.2.5). Furthermore, the developer may specify safety and liveness properties regarding the coordination of the pattern 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 2.2.6).

2.2.1. Real-Time Coordination Pattern

Figure 2.7 shows an example for a Real-Time Coordination Pattern. A dashed ellipse contains the name of the pattern (here: Delegation). Each solid squares represents a role. A dashed line connects the ellipse and the square. Each dashed line has a label that shows the name of the role. The solid line, which represents the communication connector, connects the two squares. The triangles within the two squares define that the two roles communicate bidirectionally with each other (this is further explained in Section 2.2.2).

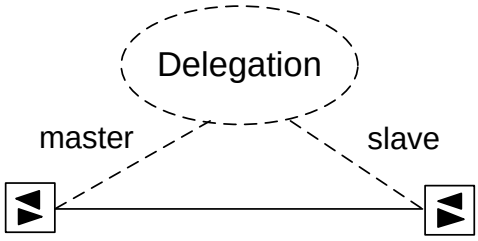


Figure 2.7.: The Real-Time Coordination Pattern Delegation

In our example, the pattern Delegation specifies the communication which is needed to delegate a specific task. The two roles are master and slave. The role master delegates the specific task to the slave. The slave executes this task and responds with a success- or failure-message to the master. Section 3.1.2 shows and describes the corresponding Real-Time Statecharts of both roles.

2.2.1.1. Constraints

- A pattern has exactly two roles.
- A pattern has exactly one communication connector.

2.2.2. Role

A role represents a communication partner of a Real-Time Coordination Pattern. Each of the two roles of a Real-Time Coordination Pattern has a unique name to differ between them within the Real-Time Coordination Pattern.

Instances of one role are communicating with instances of the other role via asynchronous messages. Therefore, a role specifies the asynchronous messages that an instance of this role may send or receive.

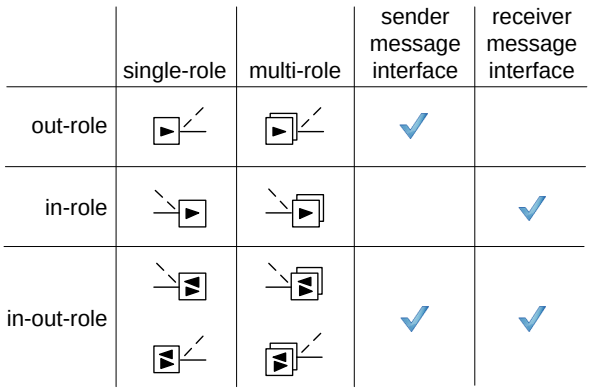


Figure 2.8.: Concrete Syntax of Roles and Messages Interfaces a Role Contains

Figure 2.8 shows the concrete syntax for roles. It depends on the fact whether the role may send or receive messages. A role of a Real-Time Coordination Pattern that may only send messages is an out-role, a role that may only receive messages is an in-role and a role that may send and receive messages is an in-out-role.

The concrete syntax of the out-role has a triangle within its square (the so-called out-triangle) that points to the center of the Real-Time Coordination Pattern. The in-role is a triangle (the so-called in-triangle) that points away from the center of the Real-Time Coordination Pattern. The in-out-role is visualized with two triangles. The upper one points to the left and the lower one points to the right.

Furthermore, the concrete syntax of a role depends on the number of instances this role may have. The cardinality of a role specifies a lower and an upper bound for this number of instances. Valid values for the lower bound are positive integers including zero. Valid values for the upper bound are positive integers (excluding zero) and infinity (represented with *). The upper bound must be equal or greater than the lower bound.

If the cardinality of a role has an upper bound of 1, then it is called a single-role. A square with a single border line visualizes such a role. A role which cardinality has an upper bound greater than 1 is a multi-role. A multi-role is visualized through a square with a cascaded double border line (cf. Figure 2.8).

Each role may reference a sender message interface, which contains all message types a role instance may send to instances of the other role, and a receiver message interface, which contains all message types a role instance may receive from instances from the other role (cf. Section 2.3).

Figure 2.8 shows which types of roles reference which type of message interface. The out-role references exactly one sender message interface. The in-role references exactly one receiver message interface. At last, the in-out-role references exactly one sender message interface and exactly one receiver message interface.

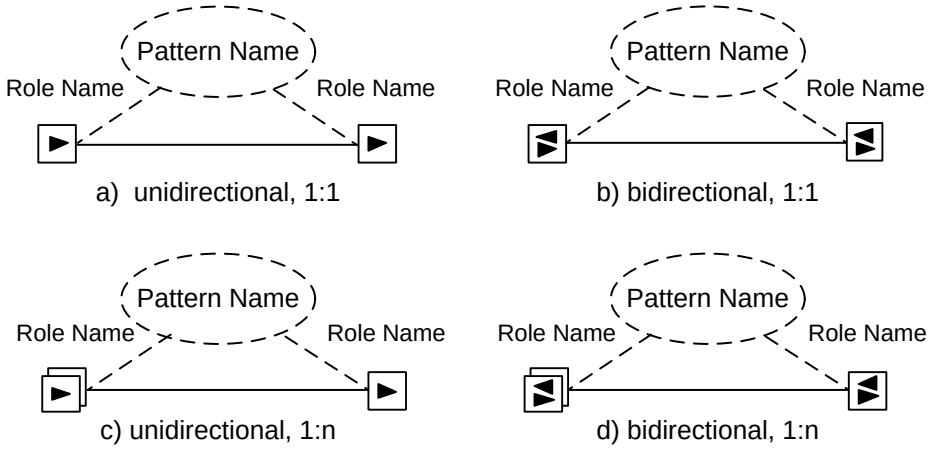


Figure 2.9.: The Four Kinds of a Real-Time Coordination Pattern

Figure 2.9 shows the four kinds of a Real-Time Coordination Pattern. They differ on the one side between the direction of communication (unidirectional and bidirectional) and on the other side between the form of communication (1:1 or 1:n).

A unidirectional communication means that messages may only be sent in one direction. Therefore, this communication consists of an out-role role and an in-role. Figures 2.9 a) + c) show the two possibilities for this communication. A bidirectional communication means that messages may be sent (and received) in both directions. Thus, both roles must be in-out-roles. Figures 2.9 b) + d) show the two possibilities for such a communication.

In MECHATRONICUML, two possible forms of communication exist: 1:1 and 1:n. 1:1 means that there is only one instance per role per Real-Time Coordination Pattern participating in the communication. Therefore, both roles must be single-roles. Figures 2.9 a) + b) show the two possibilities for an 1:1 communication. 1:n means that one role has only one instance, but the other role may have multiple instances per instantiated Real-Time Coordination Pattern. A pattern which specifies such a communication has one single-role and one multi-role (the

1 is the multi-role, the n is the single-role). Figures 2.9 c) + d) show the two possibilites for specifying a 1:n form of communication.

2.2.2.1. Constraints

- The names of the two roles in a Real-Time Coordination Pattern are unique.
- Only one role of a Real-Time Coordination Pattern may be a multi-role. Therefore, the form of communication m:n (m instances of the one role, n instances of the other) is in this version of the report not possible.
- The lower bound of the cardinality lb of a role is $lb \in \mathbb{N}_0$.
- The upper bound of the cardinality ub of a role is $ub \in \mathbb{N} \cup \{*\}$.
- The upper cardinality bound is equal or greater than the lower bound.
- A Real-Time Coordination Pattern has either one out-role and one in-role or two in-out-roles.
- The 1:n communication kind *broadcast* is not possible.

2.2.3. Communication Connector

Both roles of a Real-Time Coordination Pattern are connected by a communication connector. It represents a communication connection between the two roles and is visualized by a black line between the two squares for the roles (cf. Figure 2.7). For analysis purposes (e. g., simulation or verification) the developer may specify an own real-time behavior for a communication connector, e. g., to model propagation delays, buffering of messages, message loss or wrong messages. The developer may use a Real-Time Statechart to describe this behavior.

2.2.3.1. Constraints

- A communication connector connects exactly two roles.

2.2.4. Pattern Instantiation

A Real-Time Coordination Pattern specifies the communication between two or more component instances (cf. Section 2.5). Therefore, the developer applies each of the two roles of a Real-Time Coordination Pattern to exactly one discrete port of a discrete or hybrid component.

A single-role may only be applied to a single-port; a multi-role may be applied to a multi-port or to a single-port. If a role is applied to a port of an atomic component, the port has to fulfill the role behavior specification. If the developer applies a role to a port p1 of a

structured component, the behavior specification of the role must be fulfilled by the port the port p1 delegates to.

Usually, both roles of a Real-Time Coordination Pattern are assigned to different component types, but it is also possible to assign both roles of the pattern to different ports of the same component type. In that case, different component instances of the same component type communicate with each other.

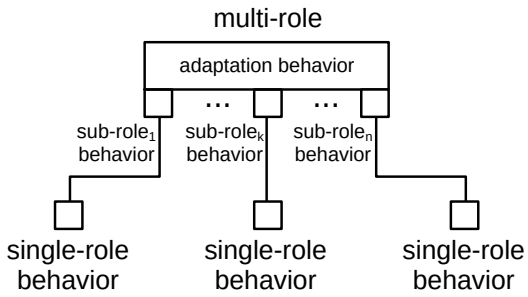


Figure 2.10.: Structure of a Multi-Role being Connected to Several Single-Roles

If a Real-Time Coordination Pattern defines a 1:1 communication, both single-roles and the communication connector will be instantiated once per instantiation of a Real-Time Coordination Pattern. If a Real-Time Coordination Pattern defines a 1:n communication, the multi-role has to communicate with n different instances of the single-role (cf. Figure 2.10). The multi-role does not communicate directly with the instances of the single-role, but uses a sub-role for this purpose, which encapsulates the communication. Therefore, the multi-role is instantiated once, but contains n instances of the sub-role, one for each instance of a single-role. Furthermore, the communication connector is instantiated n times and connects each instance of the sub-role with an instance of the sub-role.

2.2.5. Role Behavior

In the current version of MECHATRONICUML, 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 may specify the behavior of a role by a Real-Time Statechart (cf. Section 2.4). The concurrent execution of both roles determines the behavior of the Real-Time Coordination Pattern.

Within the Real-Time Statechart of a role, only asynchronous message events may be used (cf. Section 2.4.10). These events are typed over the message types declared in the message interfaces of the role. Asynchronous messages may only be used for interaction with the other role and may not be used for communication within the role (e. g., to communicate between several regions).

A role may define variables to store information (e. g., the current position of the system). The Real-Time Statechart of the role may access these variables and use them for transition

guards or may change them by side effects of the transitions. The Real-Time Statechart of the role may not access the variables that are defined by the other role. Furthermore, a role may define operations that implement the actions of the states and the side effects of the transitions (cf. Section 2.4.7). The Real-Time Statechart of the role may call these operations, but may not directly call operations of the other role.

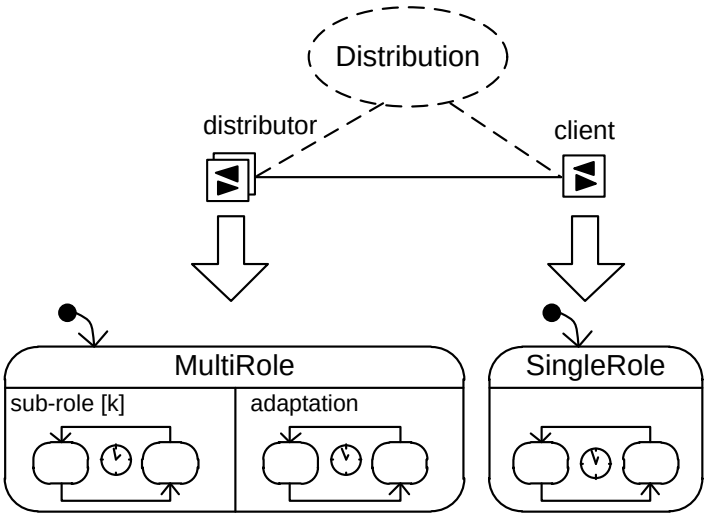


Figure 2.11.: A Multi-Role and a Single-Role Specified by Real-Time Statecharts

Figure 2.11 shows the Real-Time Coordination Pattern Distribution with its roles distributor and client. As depicted, a single-role like client is defined by a Real-Time Statechart without further restrictions. But a multi-role is defined by a Real-Time Statechart which has several restrictions. These will be explained in the following.

Due to the 1:n communication, the multi-role has to communicate with n instances of the single-role. In the current version of MECHATRONICUML, the behavior to communicate with each instance of the single-role has to be identical and is defined in one parameterized sub-role behavior. A multi-role further defines an adaptation behavior to define an order of the n communications, [EHH+11]. Therefore, the Real-Time Statechart of a multi-role contains only one state on the upper level of hierarchy. This state contains exactly two regions: a region named sub-role and a region named adaptation (cf. Figure 2.11).

The region sub-role defines the behavior for the sub-roles of the multi-role. This region is parameterized by the parameter k, which is the unique identifier (an integer value) for each instance of the single-role. A sub-role region is instantiated n times, once for each of the n instances of the single-role. The unique identifier thereby replaces the parameter k.

The region adaptation defines the adaptation behavior of the multi-role. It specifies the order of communication and 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. 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].

2.2.6. Verification Property

The Real-Time Coordination Pattern and both roles may have verification properties regarding the safety or the liveness of the system. "A safety property is one that something bad will not happen. [...] A liveness property is one which states that something good must happen." [Lam77]

Verification properties must be fulfilled by the behavior specification of the pattern or of a role (e. g., through their Real-Time Statecharts). The fulfillment may be checked using model checkers if the property is defined in a formal syntax like the Timed Computational Tree Logic (TCTL) [ACD93].

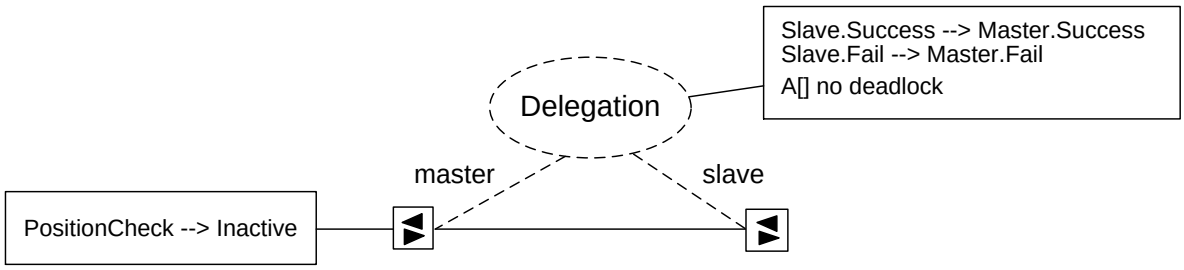


Figure 2.12.: The Delegation Pattern with Safety Properties for the Role master and the whole Pattern

Figure 2.12 visualizes the pattern Delegation and shows the verification properties for its role master 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 square of a role (here: to the role master) or to the black-dashed ellipse which displays the name of the pattern (here: to the pattern Delegation). If a role or a pattern has no verification properties (like the role slave), the rectangle and its link are not shown.

In our example, three TCTL-properties are specified for the pattern Delegation using the concrete syntax of the model checker UPPAAL [BDL04]: slave.Success - - > master.Success expresses, that if the role slave is in state Success, then the master will also eventually enter its Success-state on all executions. slave.Fail - - > master.Fail expresses, that if the role slave is in state Fail, then the master will also eventually enter its Fail-state. The third property states A[] no deadlock, which expresses that there exists no deadlock within this pattern.

The role master specifies the TCTL-property PositionCheck - - > Inactive, which informally expresses that always if state PositionCheck is reached, then eventually the state Inactive is reached.

2.3. Message Interface Specification

Message Interfaces define the interfaces of the roles of parameterized coordination patterns. Each message interface defines a set of signatures, called message types, of asynchronous events that may be exchanged between two roles of a Real-Time Coordination Pattern (cf. Section 2.2).

2.3.1. Message Interface

A message interface defines a set of signatures for asynchronous messages that may be exchanged between the roles of a Real-Time Coordination Pattern. We call these signatures message types (cf. Section 2.3.2). Each message interface specifies a name that must be unique within the modeled system. Figure 2.13 shows an example of two message interfaces with the names Delegation_Master and Extended_Delegation_Master.

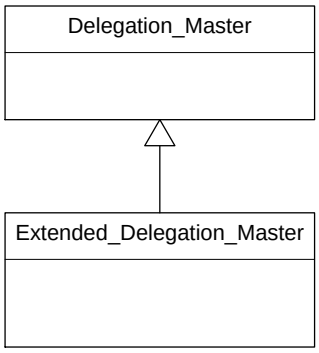


Figure 2.13.: Concrete Syntax of a Message Interface and a Generalization Connector

The 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 (cf. Figure 2.14).

Message interfaces support inheritance relations as they are known from class models like UML [Gro10b] or Ecore [SBPM08]. A message interface may inherit from many other message interfaces. In Figure 2.13, the message interface ExtendedDelegationSender inherits directly from DelegationSender. Then, we call DelegationSender the *super message interface* and ExtendedDelegationSender the *sub message interface*. The inheritance relation is transitive, i.e. sub message interfaces of ExtendedDelagationSender inherits indirectly from DelegationSender.

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.

2.3.1.1. Constraints

- A message interface must either specify at least one message type itself or it must inherit from at least two message interfaces.
- A message interface must not inherit from itself (directly or indirectly).

2.3.2. Message Type

A message type declares the signature for an asynchronous message of a Real-Time Statechart. A message type has a name and an ordered set of parameter declarations where each parameter declaration specifies a name and a concrete type.

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

```
<messagetype> ::= #MessageType.name '(' [<parameterlist>] ')'
<parameterlist> ::= <parameter> | <parameter> ',' <parameterlist>
<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.

Figure 2.14 shows the two message interfaces Delegation_Master and Delegation_Slave containing three message types. The message type check has two parameters x and y of type int. The message types accepted and declined specify no parameters.

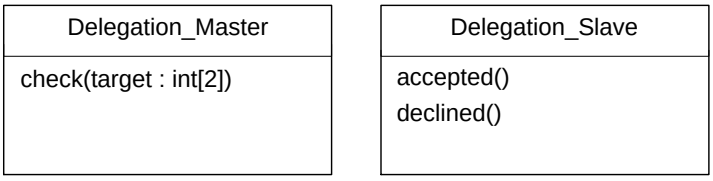


Figure 2.14.: Concrete Syntax of a Message Type

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.

2.4. Real-Time Statechart

Real-Time Statecharts are used to describe the behavior of a component (cf. Section 2.5.1.4), a port (cf. Section 2.5.1.3), or a role (cf. Section 2.2.5. An antecessor version of Real-Time Statecharts were defined by Giese and Burmester [GB03]. A Real-Time Statechart contains states, transitions, clocks, history elements, entry-points, and exit-points. Different Real-Time Statecharts can communicate with each other via their port, or their role. As a result, each statechart of a port, or a role has a message-event pool as suggested by the UML [Gro10b] which stores the incoming messages from the statechart of the connected role or port. We use a FIFO queueing for the message-event pool.

2.4.1. Real-Time Statechart

Real-Time Statecharts are a combination of Harels statecharts [Har87], timed automata [DM01, BDL04, AD94] and UML state machines[Gro10b]. The execution semantics is event-based and reactive. The timing behavior of Real-Time Statecharts are based on timed automata. For readers who are not familiar with the theory of timed automata we recommend the survey paper by Alur and Dill [AD94].

We add features such as deadlines for transitions [GB03] or priorities to get deterministic behavior. Another feature which we add is the possibility to reference a statechart in a region of another statechart. In this case we call them sub-(real-time) statechart. If a Real-Time Statechart is the top element in the hierarchy we call them root-(real-time) statechart. In contrast to the UML it is not possible to reference a Real-Time Statechart in a so called *submachine state* [p. 551][Gro10b].

The notion of a Real-Time Statechart is also a combination of the notion of Harels statecharts and the notion of timed automata. The following EBNF expression defines the default notation for a StatechartAttributeDefinitionLabel. Elements which have the prefix “#” are references to the meta model elements of a statechart:

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

The StatechartAttributeDefinitionLabel is shown above the states of the root-statechart. Figure 2.15 shows the position of the label and Figure 2.21 shows a concrete example. The following sections describe the syntax and semantics of the contained elements. We use in these sections the word statechart as a synonym for Real-Time Statechart.

2.4.2. State

A state represents a situation in which the system stays if the state is active. Each state has a name, which must be unique inside the same statechart. Directed transitions connect states (cf. Section 2.4.4). A state can have side-effects as entry-, do-, and exit-actions. We define the execution semantics of actions in Section 2.4.7.

In MECHATRONICUML we distinguish between simple states, composite states, and orthogonal composite states. A simple state has no hierarchy and has no embedded elements. A composite state[1] is a state which contains at least one region. The developer can add hierarchy to a statechart by adding a region to a state. Further, regions make it possible to model orthogonal behavior (cf. Section 2.4.3). By assigning a region to a simple state, the simple state becomes a composite state. In each region of an active composite state is always one state active. If a composite state contains more than one region it becomes an orthogonal composite state.

In real-time systems it is important that a system stays not to long in a certain state when it has to fulfill hard real-time constraints. Therefore, it is possible to specify this requirement as a time invariant for the state. The developer specifies the time invariant as a concrete upper time bound. An outgoing transition must fire before the state is longer active than the upper time bound. The statechart is in a *time stopping deadlock* if no transition fires before the state violates its invariant [DMY02a]. A statechart should switch to a fail-safe state to avoid an invariant violation and consequently a time stopping deadlock.

The different regions of a composite state can communicate with each other via synchronization which are typed by channels (cf. Section 2.4.9). This channels belong to the orthogonal composite state.

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.

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:

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

[1] in recent papers also called complex state

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:

```
<channelDefinitionLabel> ::=
'ch:␣' [#channels.name['('#channels.inParameter.eType
[',␣'#channels.inParameter.eType]*')'] [',␣'
#channels.name['('#channels.inParameter.eType [',
'#channels.inParameter.eType]*')']* ]
```

Figure 2.15 shows a template for the state and statechart syntax.

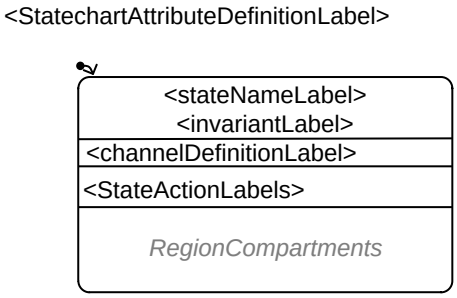


Figure 2.15.: Concrete Syntax Template of a State with StateActionLabel, ChannelDefinition-Label and the Parent Statechart with the StatechartAttributeDefinitionLabel

Figure 2.16 shows a simple state and Figure 2.17 shows a simple state with an entry, exit-, and do-action.

Figure 2.16.: Concrete Syntax of a Simple State

A composite state has an additional compartment for the embedded statechart. This shows all elements which are contained in this statechart. Figure 2.18 shows a composite state with has a statechart as child. This statechart contains two other states.

Constraints

- A state has at most one invariant per existing clock.

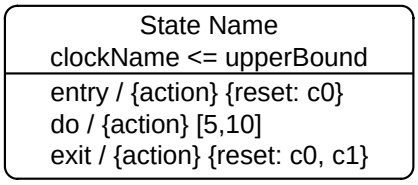


Figure 2.17.: Concrete Syntax of a Simple State with Actions

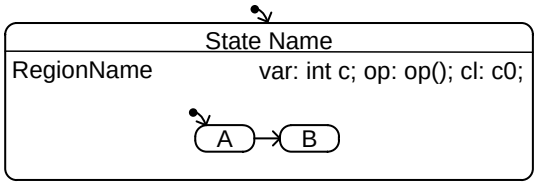


Figure 2.18.: Syntax of Complex State with One Region Compartment

2.4.2.1. Initial State

The developer can declare a state as the initial state. 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 2.19 shows an initial state.

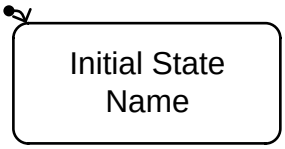


Figure 2.19.: Concrete Syntax of an Initial State

Constraint

- Only one initial state per statechart.

2.4.2.2. 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. A final state can have an entry-, do-, and exit-action. The deactivation of the parent composite state causes the execution of the exit-action of the final state. A final state is shown as a state which is marked with a doubled line around the state. Figure 2.20 shows a final state.

Figure 2.20.: Concrete Syntax of Final State

Constraints

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

2.4.3. 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 a region. Each region has a name. If a state contains more than one region, each name of the region must be unique. If the region is containing a Real-Time Statechart of a port, its typical name is the name of this port (cf. Section 2.5.1.3).

Real-Time Statechart have an deterministic behavior. Real-Time Statechart would have a nondeterministic behavior if they have several regions and in more than one region a transition is ready to fire at the same time instance. We make the behavior deterministic by defining a priority for each region to get a sequential semantics as Zündorf [Zü01, p. 159] defined. The priorities must be unique among all regions which have the same parent state. These priorities determine the order of processing. This means, the regions and respectively the outgoing transitions of the currently active state are evaluated and fired in the priority order of their parent regions. The region priorities also determine the execution order of the do-actions of the active states. The resulting structure of statecharts, containing states, and embedded statecharts must be an *acyclic* graph. It is forbidden to reference a statechart in a region which self stands in a parent relation with the same statechart.

A composite state with regions is visualized by tiling the state into regions by using dashed lines. It does not matter if they are tiled horizontal or vertical. A small circle in the upper right corner near the dashed line contains the priority value. A solid line separates the text compartment of the surrounding state from the orthogonal regions. Figure 2.21 shows an orthogonal state with two regions.

2.4.4. Transition

A transition represents a possible state change of a statechart. Therefore, it has a source state and a target state. Further constructs which we annotate at a transition are guards, clocks constraints, synchronizations, trigger and raise messages, transition-actions, and priorities. We define the priority of a transition as a local priority. This means that the priority order depends on all outgoing transitions of the source state of a transition.

A state change from the source state to the target state happens when a transitions *fires*. This means that the source state of the transition changes its state from active to inactive and that the target state of the transition changes its state from inactive to active. Further, the

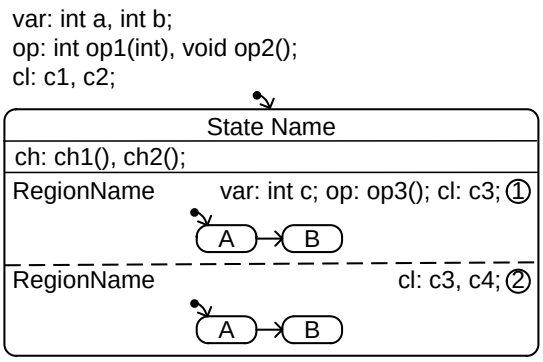


Figure 2.21.: Concrete Syntax of an Orthogonal State

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 the entry-action of the target state is executed. A transition fires only when it is *enabled*. As a prerequisite that a transition becomes enabled the source state must be active. Also, the guard and the clock constraint of a transition must be evaluated to true for enabling a transition. Further, if required, must the synchronization channel be enabled or the asynchronous trigger message-event must be available. Additionally, an enabled transition fires only if it has the highest priority regarding to other enabled transitions. If a transition has an asynchronous trigger message-event it must wait till the required message-event is available in the message-event pool (cf. Section 2.4.10). A transition can be synchronized with another transition. For a more detailed description of synchronization refer to Section 2.4.9. If a transition is real-time-critical it has a deadline. The developer specifies this deadline as a lower and upper bound. The lower bound specifies the minimum time which the transition needs to fire and the upper bound specifies the maximum time which the transition needs to fire.

We distinguish transitions if they are a *simple* or a *high level* transition. The source state of a simple transition is a simple state. If the source state is a composite state we call the transition high level transition. If several transitions are enabled, whose source states are located on different hierarchy levels, only the transition with the highest located source state fires. There is only one exception for the rule that higher located transitions overrule lower located transitions. When using synchronizations this rule is may be not correct (cf. Section 2.4.9).

It is possible to specify so called inter-level transitions which cross the boarder of a hierarchical state via entry-/exit-points. For a description refer to Section 2.4.5.

A transition is drawn as a line with an arrowhead. A transition leaves the border of a state or state exit-point and end at either a state or state entry-point. The priority value is placed at the source of the transition. Figure 2.22 shows a template for the transition syntax.

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

```
<transitionLabel>::=[<clConstraints>][<guard>][<triggerEvent>][<sync>]
```

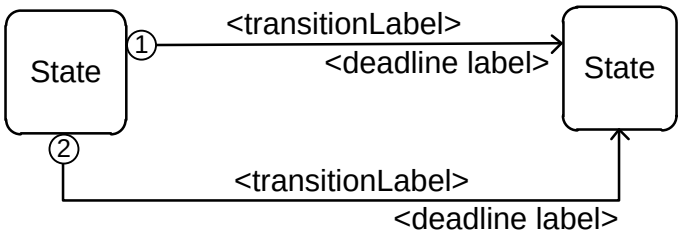


Figure 2.22.: Concrete Syntax Template of Transition with Transition Label and Deadline Label

```
    [ '/' [<action>][<raisedEvent>][<clockResets>] ]

<sync>          ::= <receivedSync> | <sendSync>
<triggerEvent>  ::= #triggerEvent.name '(' [ <parameterList> ] ')'
<clConstraints> ::= '[' #clockConstraints [ ',' #clockConstraints ]* ']'
<guard>         ::= '[' #guard ']'
<action>        ::= '{' (#transitionAction.expressions |
                     #transitionAction.name)
'}' <raisedEvent>   ::= #raisedEvent.name [ '(' <parameterList> ')' ]
<clockResets>   ::= '{reset:␣' <clockList> '}'

<receivedSync>  ::= syncExpr'?'
<sendSync>      ::= syncExpr'!'
syncExpr ::= #syncChannel.name [ '['
#integerExpression ']' ] [ '(' <paramterList> ')' ]

<clockList>     ::= #clock.name [ ',␣' #clock.name ]*

<parameterList> ::= #parameterValue [ ',' #parameterValue ]*
```

The following EBNF expression defines the default notation for adeadlineLabel. Elements which have the prefix "#" are references to the meta model elements of transition:

```
<deadlineLabel> ::= [
  ['[' #relativeDeadline.lowerBound ';' #relativeDeadline.upperBound']']
  ['\n' #absoluteDeadlines.clock.name '∈'
    '[' #absoluteDeadlines.lowerBound ';' #absoluteDeadlines.upperBound ']'
  ]*
]
```

Figure 2.23 shows an example of a concrete transition with nearly all allowed elements. The only missing element is the received synchronization, because we don't allow that the same transition have a sender and a receiver synchronization.

Constraints

- The source and the target of the transition must be set.

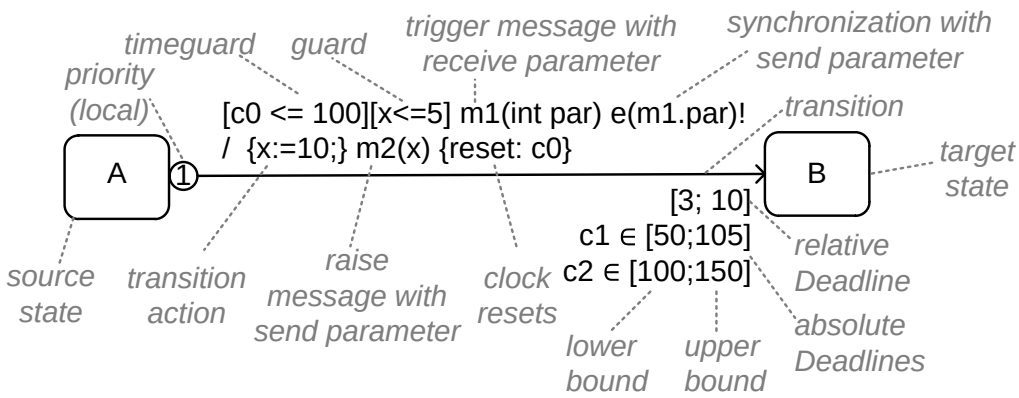


Figure 2.23.: Concrete Syntax Example of a Transition

- Transitions cannot cross region borders.
- The same transition cannot have a sender and a receiver synchronization.

2.4.5. Entry-/Exit-Point

Developers uses entry-/ exit-points to realize inter-level transition to and from a sub-statechart. A sub-statechart is a statechart which is contained in a region. This region is the parent region of that sub-statechart.

An exit-point makes it possible to deactivate the parent state of the parent region of the sub-statechart. Entering an exit-point of a sub-statechart implies the deactivation of the parent region and the firing of the transition that has the state exit-point as source.

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 2.24.

An entry-point makes it possible to activate an explicit state of a sub-statechart. Entering an entry-point of a state implies the firing of the transition that has the entry-point as source in the sub-statechart and the activation of the state which is the target of that transition.

An entry-point is drawn as a small circle on the border of a submachine, as shown in Figure 2.24.

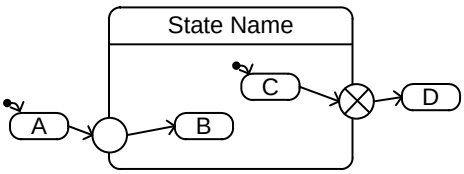


Figure 2.24.: Concrete Syntax of Entry-/Exit-Points

Constraints

- In the current version of MECHATRONICUML we don't allow to use entry-/ exit-points in states that have more than one region.

2.4.6. Shallow History

A Real-Time Statechart can have one history element which stores the most recent 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 and not the active states of an active composite state. 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]. MECHATRONICUML currently only supports shallow history.

Shallow history is drawn as a circle which encloses the letter *H* (cf. Figure 2.25).

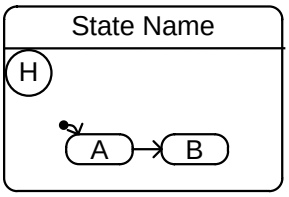


Figure 2.25.: Concrete Syntax of Shallow History

Constraints

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

2.4.7. 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 has no entry-action defined, 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 upper bound. The lower bound defines the earliest time when the do action is executed after the last execution. The upper bound defines the latest time when 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.

An action has a defined expression, which can be expressed in any language, such as Java, C, or Modelica. Currently, we do not read or parse the actions. So we define no semantics for it. We plan for the next version of MECHATRONICUML to define a MECHATRONICUML action language that is independent of a certain programming language and has a defined syntax and semantics. This should be transferable to certain concrete programming languages. Instead of defining actions in an action language, it is possible to describe actions via graph transformation rules or story diagrams [EHH+11].

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

2.4.8. Clock

A Real-Time Statechart has like timed automata a finite number of clocks. A clock stores a amount of elapsed time during the execution of a system. Time elapses continuously and not in discrete steps [AD94]. An entry-, exit-action can reset a clock (cf. Section 2.4.7) and a transition which fires can reset a clock (cf. Section 2.4.4). The stored time is relative to the last point in time when the clock has been reset.

2.4.9. 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 2.4.2) 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 2.26. It consists of the two parallel regions B and C inside the initial composite state

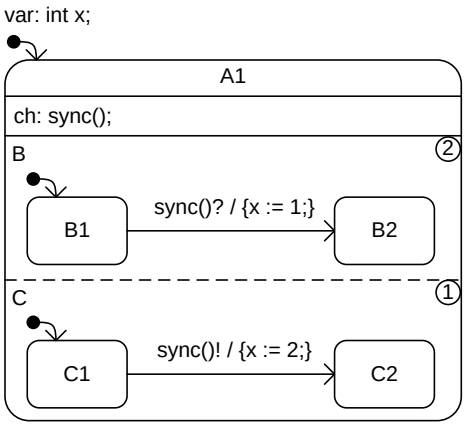


Figure 2.26.: Synchronizations May Overrule Execution Order Defined by Region Priorities

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

The region priority of the sender transition determines the priority of the synchronization. Figure 2.27 gives an example for this behavior. The initial composite state A1 declares the synchronization channels syncBE() and syncCD() and contains the four regions B, C, D and E with descending priorities. In each region there are two states with a transition between them. The transitions have no conditions except for one sending or receiving synchronization at each and thus are all enabled. The transition B1→B2 with the highest region priority of 4 receives and the transition E1→E2 with the lowest region priority of 1 sends synchronizations via syncBE. The transitions with region priorities in between – C1→C2 with region priority of 3 and D1→D2 with region priority of 2 – send respectively receive synchronizations via syncCD.

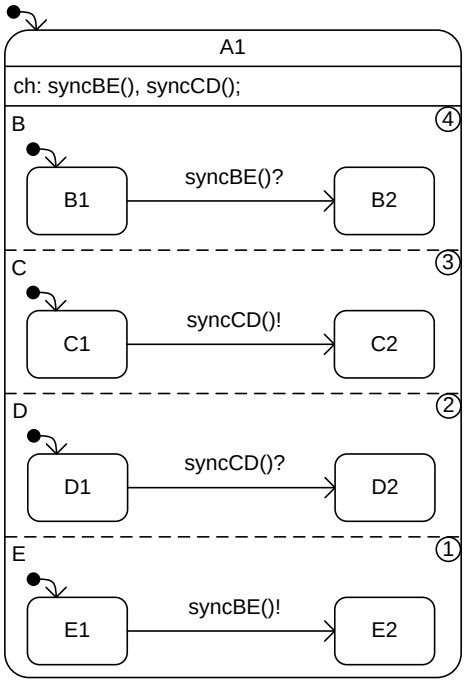


Figure 2.27.: The Sender Transition Determines the Priority of the Synchronization so Final State Configuration is (A1, B1, C2, D2, E1)

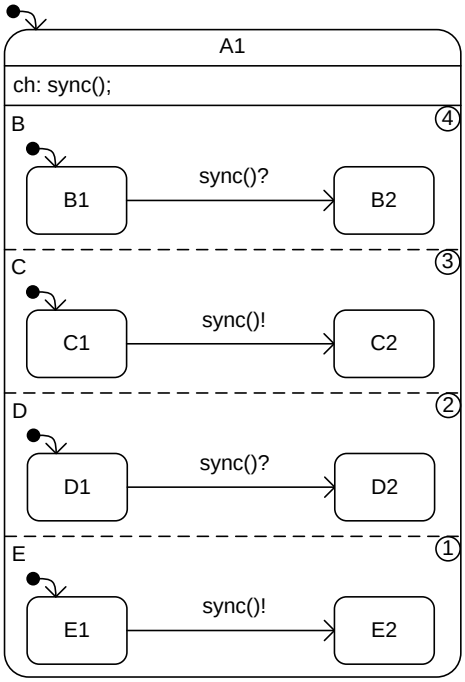


Figure 2.28.: Only One-to-One Synchronizations Are Allowed So Final State Configuration is (A1, B2, C2, D1, E1)

Because the sender transition determines the priority of the synchronization the transitions C1→C2 and D1→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 2.28 illustrates this. It differs from Figure 2.27 insofar that state A1 declares only the single synchronization channel sync() which all four transitions use. Because only one-to-one synchronizations are allowed this results in the four different synchronization combinations C1→C2 and B1→B2 or C1→C2 and D1→D2 or E1→E2 and B1→B2 or E1→E2 and D1→D2. Of these the sender transition with the highest region priority (C1→C2) synchronizes with the receiver transition with the highest region priority (B1→B2) which results in the final state configuration (A1,B2,C2,E1,D1).

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 2.29. The initial composite state A1 declares

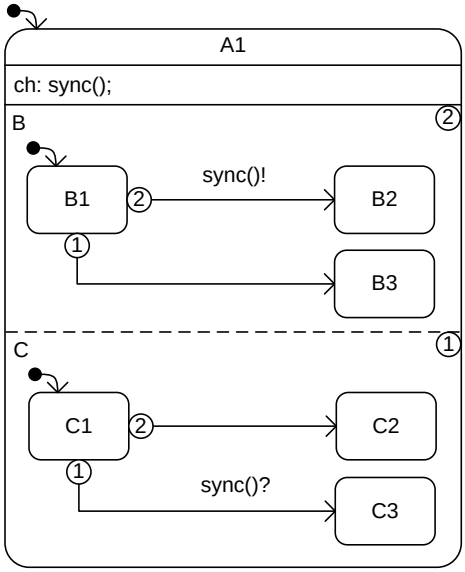


Figure 2.29.: Synchronization May Overrule Transition Priorities

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 2.27 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 2.4.4) 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 2.30. It differs from Figure 2.29 insofar that it defines

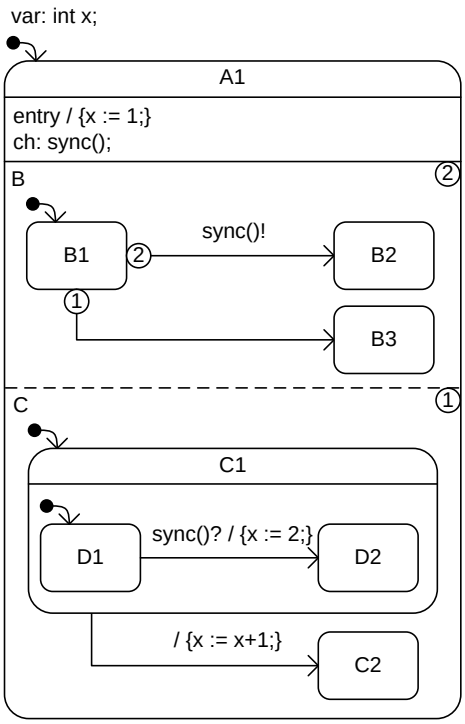


Figure 2.30.: Synchronization May Violate the Principle that Transitions with Higher Located Source State Overrule Transitions with Lower Located Source States

a variable x of type int which is set by the entry action of state A1 to 1 and contains a different region C. In region C there is only a transition from the initial state C1 to state C2 which increments the value of variable x. C1 is a composite state now which contains region D with the states D1 and D2 and the transition from D1 to D2 which receives a synchronization via channel sync and assigns variable x with 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.

2.4.10. Asynchronous Message-Event

We uses message-events to model asynchronous communication between Real-Time Statecharts. Sender and receiver message interfaces define asynchronous message-events (cf. Section 2.3). A message-event has parameters which transfer information from the sender to the receiver of that message-event. The signature of the message type of the message-event defines which parameter the message-event has (cf. Section 2.3). 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 the Real-Time Statechart 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 raised message-event is send via the associated port of the Real-Time Statechart. This port is connected to another port which has itself a Real-Time Statechart. Further, this port has Real-Time Statechart and a receiver message interface. We uses in Real-Time Statechart the defined message-events of the receiver message interface as trigger message-events. A trigger message-event is a message-event which can enable an transition when it is available and all other required conditions, for enabling a transition, are true (cf. Section 2.4.4). A 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. Per message-event only one transition can fire and dispatch the message-event. Message-events have currently no 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 planed for a future version. Figure 2.23 shows an example of asynchronous message-events which a transition uses.

2.5. 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., they 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 reason is that our component model allows for formal analysis of instances of components. 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 2.6.

In our component model, we distinguish between atomic components and structured components. Atomic components that will be introduced in Section 2.5.1 contain a stateful behavior specification. Depending on their implementation and purpose, we will distinguish between four kinds of atomic components. Structured components that will be introduced in Section 2.5.2 are assembled by embedding other components. Therefore, they carry no explicit behavior specification by themselves. In contrast to atomic components, we will only distinguish between two kinds of structured components.

2.5.1. Atomic Component Type

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

2.5.1.1. 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 is exposes via the ports of the components. 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 four kinds of atomic components based on their implementation and purpose. The four kinds are discrete components, continuous components, hardware components, and hybrid components. A discrete component is a software component whose behavior is specified by a discrete component behavior specification (cf. Section 2.5.1.4). As defined in Section 2.2, 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 continuous component is assumed to be specified in

a control engineering tool like CamelView[2] or Matlab/Simulink [3]. A hybrid component can interact with discrete components and continuous components at the same time. Therefore, the hybrid component contains a discrete component behavior specification including discrete ports and hybrid ports (cf. Section 2.5.1.2) through which it interacts with continuous components. A hardware component is a special kind of component that represents any kind of physical entity like a hard disk or a cable. Hardware components are no actual part of the system model itself, but, they are used to analyze failure propagations of hardware failures in the software model in course of a hazard analysis [PSWTH11].

Figure 2.31 shows the concrete syntax of all four kinds of atomic components. Discrete, hybrid, and continuous components 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 because they all represent software. In contrast, hardware components are visualized as 3D boxes. They use the same positioning of the component icon and the name label as the other kinds of components. The concrete syntax of ports will be explained in the subsequent Section 2.5.1.2.

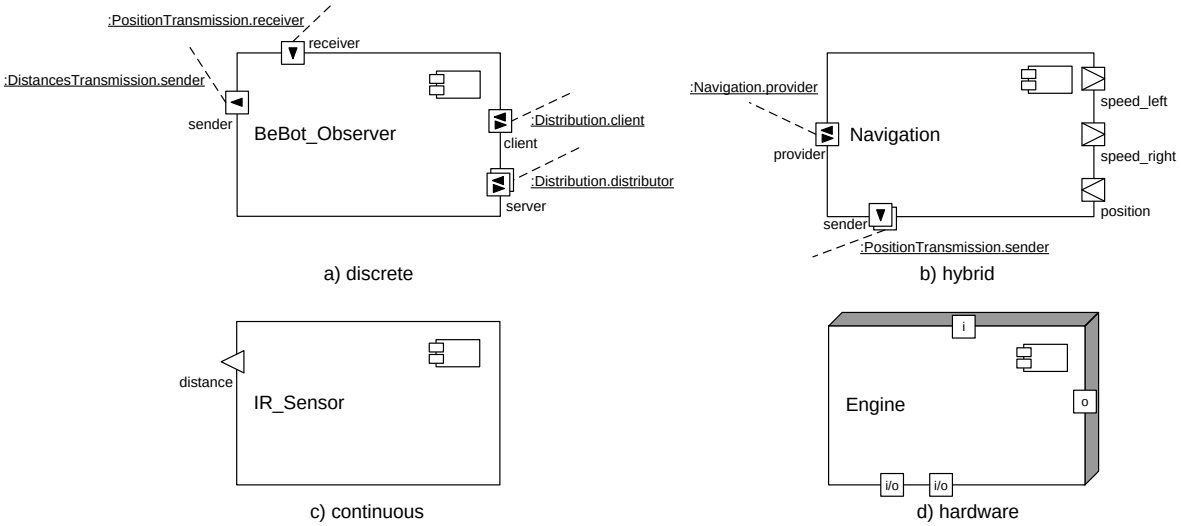


Figure 2.31.: Concrete Syntax of Different Kinds of Atomic Components

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 2.5.1.2.

[2] http://www.ixtronics.com/ix_hist/English/CAMeLView.htm
[3] http://www.mathworks.de/

2.5.1.2. 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 2.5.2.5) 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 2.6). 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. The behavior of a port is specified by a port behavior specification as described in Section 2.5.1.3.

Similar to the different kinds of components, we distinguish four kinds of ports in our component model based on the kind of information they process. These are discrete, continuous, hybrid, and hardware ports. A discrete port is used for sending or receiving discrete, asynchronous messages that are typed over a message type (cf. Sections 2.4.10 and 2.3.2). Discrete ports may be in-ports, out-ports and in/out-ports that means they may receive messages, send messages, or both. A continuous port sends or receives continuous values like, e.g., a current and, thus, it may not be an in/out-port. A hybrid port either operates as an A/D or as a D/A converter, i.e. it transforms a continuous input signal into a discrete value or a discrete output value into a continuous output signal according to its port behavior specification (cf. Section 2.5.1.3). A hardware port models a failure of a hardware entity that effects the software model. This port is connected to the port (discrete, continuous or hybrid) at which the hardware failure enters the software.

The concrete syntax of ports is depicted in Figure 2.32. 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 2.31. In addition, the name of discrete, continuous and hybrid ports 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 underlined name of the role is visualized at this line and is prefixed by a colon. In a complete MECHATRONICUML model, such a role of a Real-Time Coordination Pattern must be assigned to all discrete ports in the model.

Discrete, hybrid and hardware 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. Hardware ports embed the letters "i", "o", or "i/o" if the port is an in-port, out-port, or in/out-port respectively. For a continuous port, the triangle "points" into the component for an in-port and out of the component for an out-port.

Discrete ports may have a sender and a receiver interface by means of a message interface (cf. Section 2.3.1). 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.

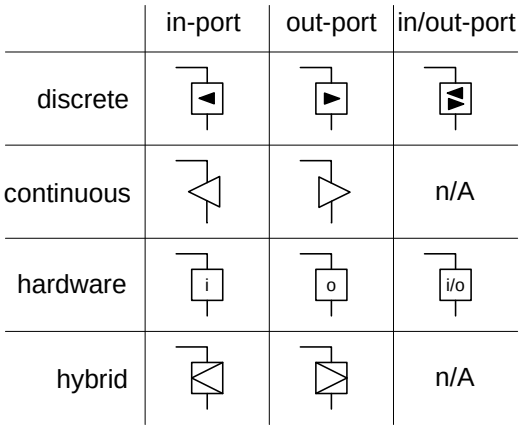


Figure 2.32.: Concrete Syntax of Ports

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 lower bound defines the minimum number of port instances that each instance of the component must have. Accordingly, the upper bound defines the maximum number of possible 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. For a hardware port, the lower bound and the upper bound must be 1.

The concrete syntax of multi-ports is shown in Figure 2.33. A multi-port has a cascaded double border line and it positioned like a single-port.

The structure and the behavior specification of a multi-port is analogous to multi-roles of a Real-Time Coordination Pattern (cf. Section 2.2.5). As shown in Figure 2.34, a multi-port consists of sub-port instances and an adaptation behavior. 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. All sub-port instances share the same port behavior specification as defined in Section 2.5.1.3 that defines their behavior [EHH+11]. 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.

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 in order to preserve the compositional verification theorem [GTB+03].

	in-port	out-port	in/out-port
discrete	◄	►	◄►
continuous	n/A	n/A	n/A
hardware	n/A	n/A	n/A
hybrid	n/A	n/A	n/A

Figure 2.33.: Concrete Syntax of Multi-Ports

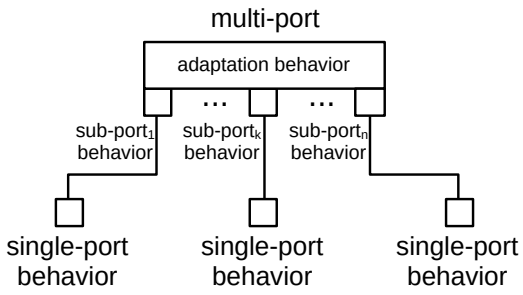


Figure 2.34.: Structure of a Multi-Port being Connected to Several Single-Ports

Constraints

- A discrete port may only be contained in a discrete component or in a hybrid component.
- A hybrid port may only be contained in a hybrid component.
- A continuous port may only be contained in a continuous component.
- A hardware port may only be contained in a hardware 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.

2.5.1.3. Port Behavior Specification

The port behavior specification specifies the run-time behavior of a port. In the current version of this document, we will only describe the behavior of discrete ports in terms of states and message passing. The behavior of hybrid and continuous ports will be defined in a future version of this document.

In the current version of MECHATRONICUML, the behavior of a discrete port is specified by means of a Real-Time Statechart. That Real-Time Statechart may be obtained in two 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 2.5.1.4). 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.

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 2.5.1.4). 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 2.4.10).

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 2.4.7). In the current version of MECHATRONICUML, 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 2.4.9 to interact with the internal behavior of the component (cf. Section 2.5.1.4). 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 2.4.9). Such synchronizations are used to exchange data with the internal component behavior and to resolve dependencies between different port Real-Time Statecharts.

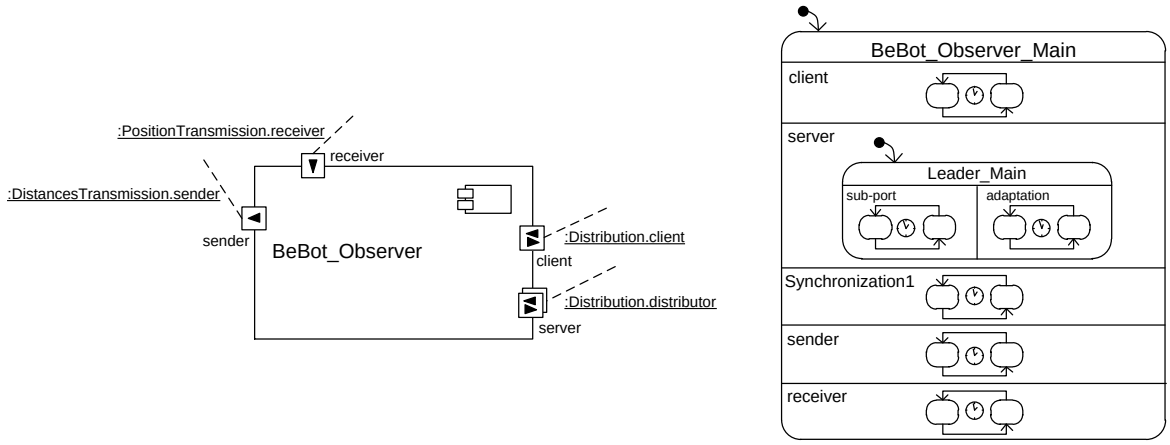
2.5.1.4. Discrete Component Behavior Specification

In the current version of MECHATRONICUML, the behavior specification of a discrete atomic component is given by means of a Real-Time Statechart. The MECHATRONICUML process offers two possibilities to obtain that component behavior specification. Both possibilities consider the port Real-Time Statechart as the implementation of the port behavior, i.e., the port behavior will be integrated into the component behavior rather than implemented in it.

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 component has no additional internal behavior 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 2.4.9). The synchronizations must not be used for communication between different component instances.

Figure 2.35 shows an example of the component behavior specification of the component BeBot_Observer containing one state BeBot_Observer_Main. The regions for the state of the component Real-Time Statechart are constructed as follows. For each discrete single-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 3.1.3). 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.



a) Atomic Component BeBot_Observer

b) Component Behavior Specification for BeBot_Observer

Figure 2.35.: Structure of the Statechart of the Component Behavior of an Atomic Component

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

For a single-port, synchronization is only allowed between the port statechart and the synchronization statecharts of the component behavior specification. For a multi-port, 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 2.4.7). The synchronization statechart may not call methods defined in the ports.

2.5.2. Structured Component Type

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

2.5.2.1. 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 components. 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 or structured hardware components are not supported. Whether a structured component is discrete or hybrid depends on the types of the embedded component parts as described in Section 2.5.2.3.

2.5.2.2. 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 2.5.2.4 on Delegations). In case of a discrete port, the port has an associated role of a Real-Time Coordination Pattern in order to connect it to another component by using a Real-Time Coordination Pattern.

Constraints

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

2.5.2.3. Component Part

A component part is a representation of a component that is embedded into a structured component. It describes the potentially multiple occurrence 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.

Since component parts are specified on the type level, a component part corresponds to importing the respective component type. Since importing the same type multiple times is superfluous, all component parts within one structured component must be typed over different component types. In cases where an instance of a structured component should use multiple instances of a part, a multi-part with a respective cardinality has to be used.

Figure 2.36 shows an example for the concrete syntax of a structured component with name BeBot_SW embedding two component parts which are typed over the atomic components BeBot_Observer and Navigation of Figure 2.31. 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 2.31).

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.

Analogously to multi-ports, a multi-part is visualized by a cascaded double border line as shown in Figure 2.37.

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.

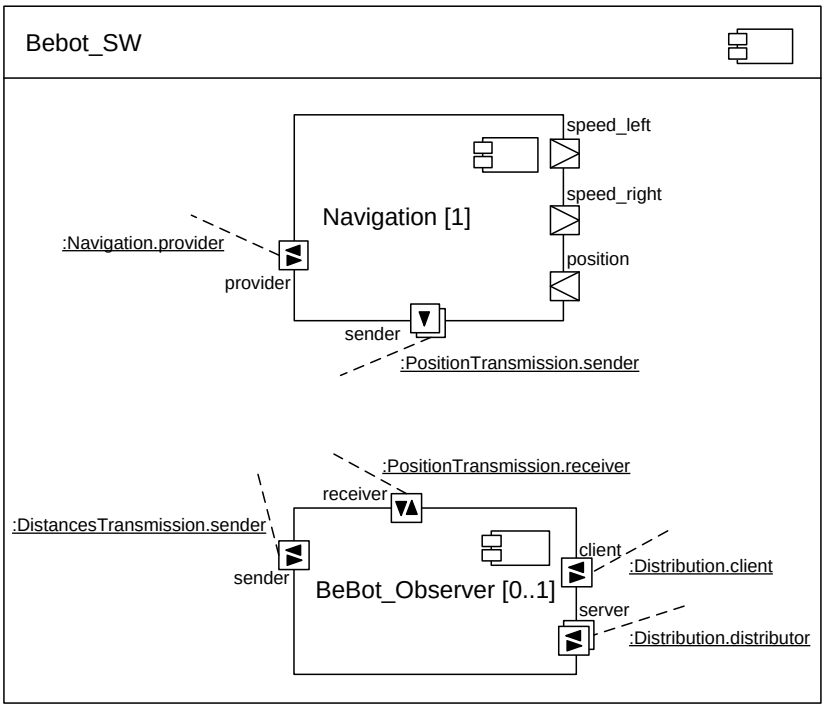


Figure 2.36.: Concrete Syntax of a Hybrid Structured Component

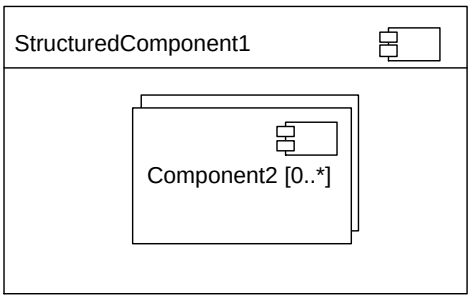


Figure 2.37.: Concrete Syntax of a Multi-Part

- A structured component may only embed component parts that are typed over discrete, hybrid, or continuous components.

2.5.2.4. Delegation

The ports of a structured component do not have a behavior specification (cf. Section 2.5.2.2). 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 2.38 shows the structured component BeBot_SW that has five ports that are delegated to two embedded component parts of types Navigation and 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.

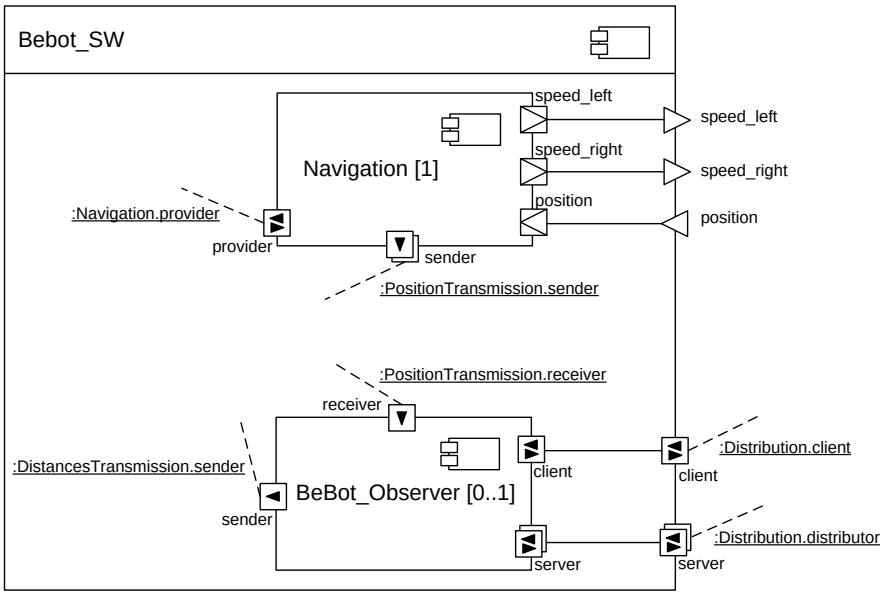


Figure 2.38.: Concrete Syntax of a Delegation

Whether a delegation link may be created depends on two conditions. First, they have to be structurally compatible according to Figure 2.39. Second, they must have compatible interfaces. If both conditions are fulfilled, the delegation link may be created. The two conditions will be explained in the following.

The structurally compatible combinations of single-ports of a structured component and a component part are summarized in Figure 2.39. If a combination is marked with a checkmark, it is possible, otherwise it is not possible.

Port of Component Part		Port of Structered Component: discrete			continuous			hardware			hybrid		
		in	out	in/ out	in	out	in/ out	in	out	in/ out	in	out	in/ out
discrete	in	✓											
	out		✓										
	in/out			✓									
continuous	in				✓								
	out					✓							
	in/out												
hardware	in												
	out												
	in/out												
hybrid	in	✓			✓						✓		
	out		✓			✓						✓	
	in/out												

Figure 2.39.: Structurally Possible Delegation Links

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 having the same cardinality. Second, it may be delegated to a single-port of a multi-part where the multi-part has the same cardinality a the multi-port. Third, a multi-port may be delegated to a multi-port of a multi-part. Figure 2.40 summarizes the possible combinations. The semantics of these combinations will be defined along with the reconfiguration operations in a future version of this document.

		Single-Port	Multi-Port
Single-Part	Single-Port	✓	
	Multi-Port		✓
Multi-Part	Single-Port		✓
	Multi-Port		✓

Figure 2.40.: Structurally Possible Delegation Links for Multi-Parts

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, interface compatibility will be defined in future releases of this document.

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.

2.5.2.5. 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. We use assembly connectors in our component model to define allowed interactions between component parts.

Figure 2.41 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.

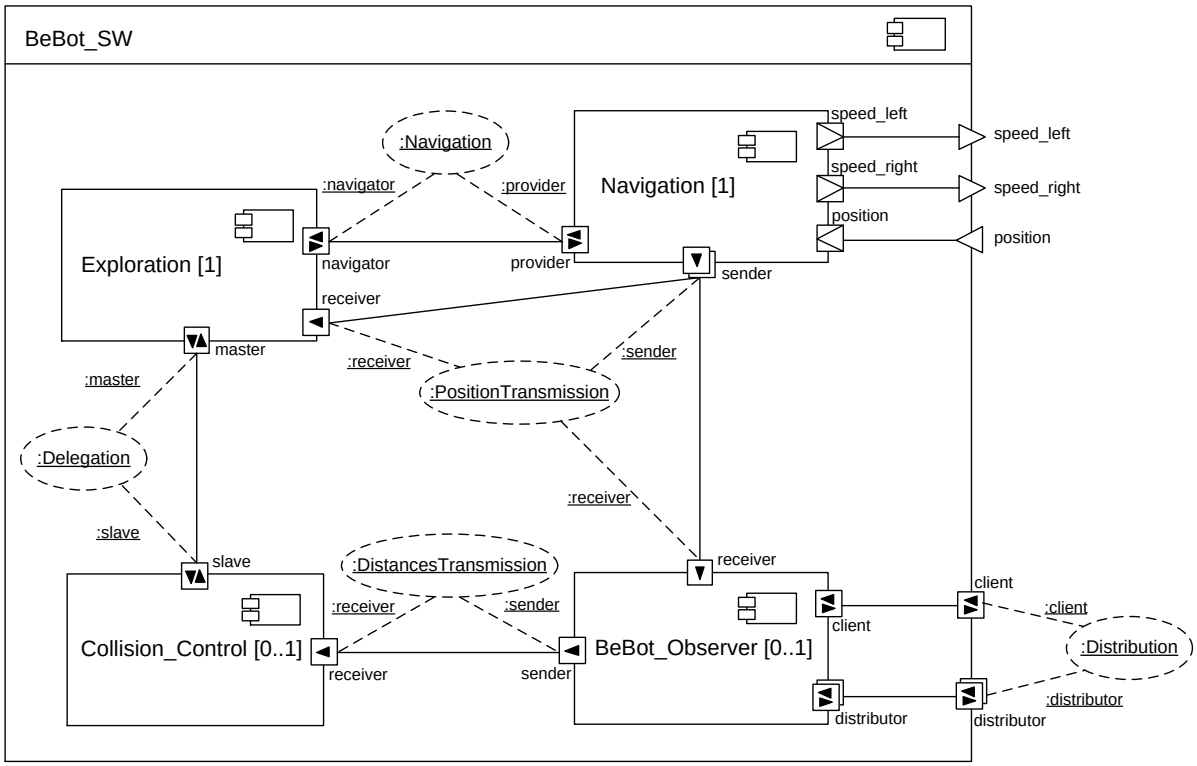


Figure 2.41.: Complete Example of a Structured Component including several Assemblies connecting the Component Parts

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 2.42. 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 2.42. 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. Hardware ports may be connected to any discrete, continuous, or hybrid port because they are only for analyzing the effects of hardware failures in the software model (cf. Section 2.5.1.2).

		discrete			continuous			hardware			hybrid		
		in	out	in/ out	in	out	in/ out	in	out	in/ out	in	out	in/ out
discrete	in		✓									✓	
	out	✓									✓		
	in/out			✓									
continuous	in					✓						✓	
	out				✓						✓		
	in/out												
hardware	in		✓			✓			✓			✓	
	out	✓			✓			✓			✓		
	in/out			✓						✓			
hybrid	in		✓			✓						✓	
	out	✓			✓						✓		
	in/out					✓							

Figure 2.42.: Structurally Possible Assembly Connectors

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.

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 NavigationPattern.

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 2.41 exists, no assembly instance may be created for an assembly ending at a port of Col-

lision_Control (cf. Section 3.5.1). The concrete assembly instances are created by reconfiguration operations during run-time. Such reconfiguration operations will be defined in future versions of this document.

Constraints

- An assembly may not connect two ports of the same simple component part.
- If two ports are connected by an assembly, they must implement different roles of the same Real-Time Coordination Pattern.

2.6. Component Instance Configuration

A component instance is a concrete occurrence of a component type (cf. Section 2.5) with concrete attribute values. It is used during design to create concrete assemblies of component instances, so-called component instance configurations. These component instance configurations are then applied during run-time.

2.6.1. Component Instance

A component instance can be derived whether from an atomic component type, a structured component type, or a hardware component type. During instantiation, the variable parts of the component definition are determined. The variable parts are the port instances that are derived from port types (cf. Section 2.5.1.2) and the embedded component instance that are derived from the component parts (cf. Section 2.5.2.3) of a structured component type.

During instantiation, the port instances of the set of port types of the component type are derived. 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. For each component part, a component instance is created. The type of the component instance is the component type referenced by the component part. There can also exist several instances of one part depending on the component part's cardinality. The initial instances of multi-parts and multi-ports are created with lowest cardinality. All attributes are set to system default values.

Figure 2.43 shows the concrete syntax of the component instance b1 of the component type BeBot_SW. A component instance basically looks like 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 syntax of port instances is the same as for port types. In the component instance configuration only the highest level of structured component instances is shown, i.e., all embedded component instances are hidden.

The component instance b1:BeBot_SW of Figure 2.43 has the three continuous port instances speed_left, speed_right, and position. The multi-role distributor of the component type BeBot_SW (e.g. Fig. 2.41) is split into the two discrete port instances distributor1 and distributor2 that implement this multi-role.

Hardware component instances are derived from hardware component types. Figure 2.44 shows the concrete syntax of a hardware component instance. As the hardware component type, it is visualized by a 3D-box. Additionally, it has a name that follows the syntax of a component instance as described above.

2.6.2. Component Instance Configuration

Component instances are connected by connector instances according to the specified Real-Time Coordination Patterns (cf. Section 2.2). Assembly connector instances connect port instances of component instances of the same hierarchy level. Delegation connector instances

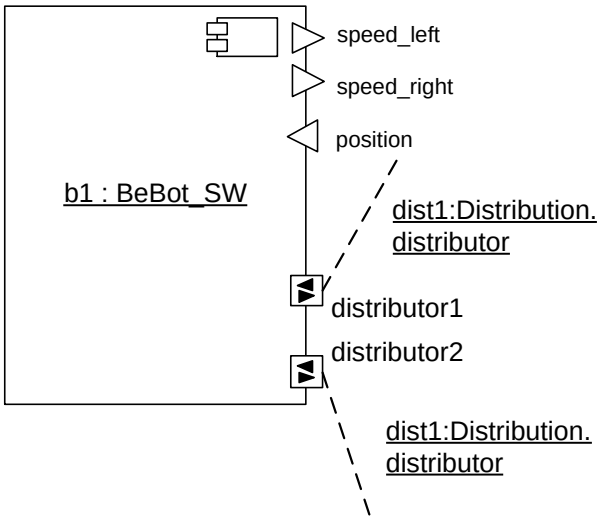


Figure 2.43.: Concrete Syntax of a Component Instance of the Component Type BeBot

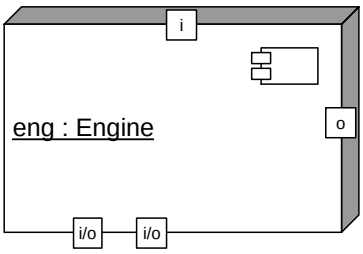


Figure 2.44.: Concrete Syntax of a Hardware Component Instance

connect port instances of adjacent hierarchy levels. This yields a structure of component instances which is called component instance configuration.

Figure 2.45 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 dist1 of the Real-Time Coordination Pattern Distribution (cf. Section 2.2 and Figure 3.1) 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.

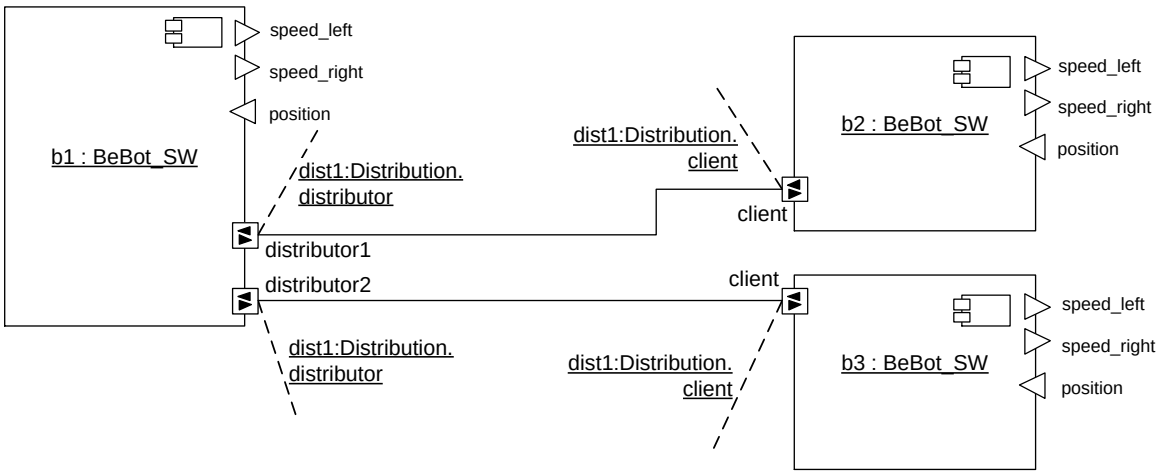


Figure 2.45.: Concrete Syntax of a Component Instance Configuration

The component instance configuration shown in Figure 2.45 shows the highest hierarchy level of the system. Figure 2.46 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 ports of the next higher level component instance b1:Bebot_SW by delegation connector instances.

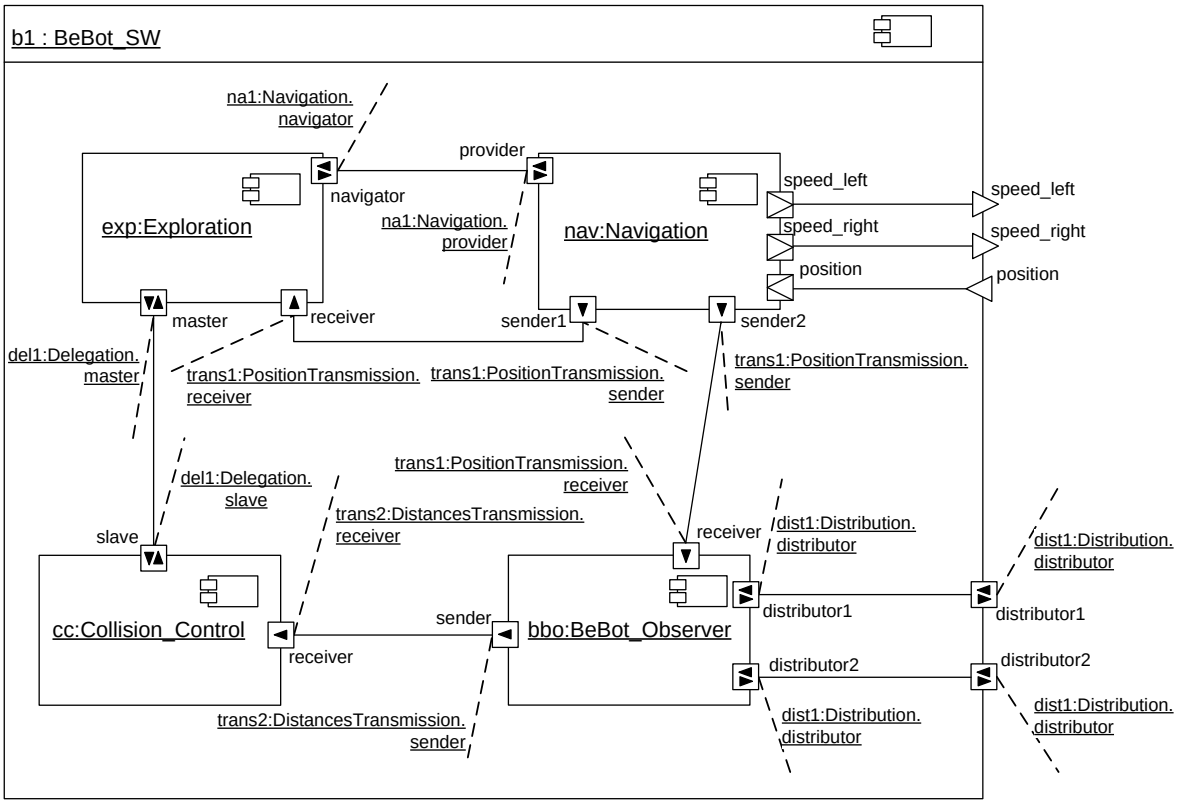


Figure 2.46.: Concrete Syntax of a Component Instance Configuration

Chapter 3.

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 3.1. Afterwards, we summarize the message interfaces used for the specification of the Real-Time Coordination Pattern in Section 3.2 and introduce the BeBot component architecture in Section 3.3. Then, the behavior of the components is described in Section 3.4 and several component instance configurations for the example scenario are introduced in Section 3.5.

3.1. Real-Time Coordination Patterns

In this section, we introduce the five Real-Time Coordination Patterns which we use in our example (cf. Figure 3.1).

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 2.2.2). 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 3.18 in Section 3.3. All of the five Real-Time Coordination Patterns of Figure 3.1 are instantiated within the BeBot_SW component.

3.1.1. 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 3.2.1.

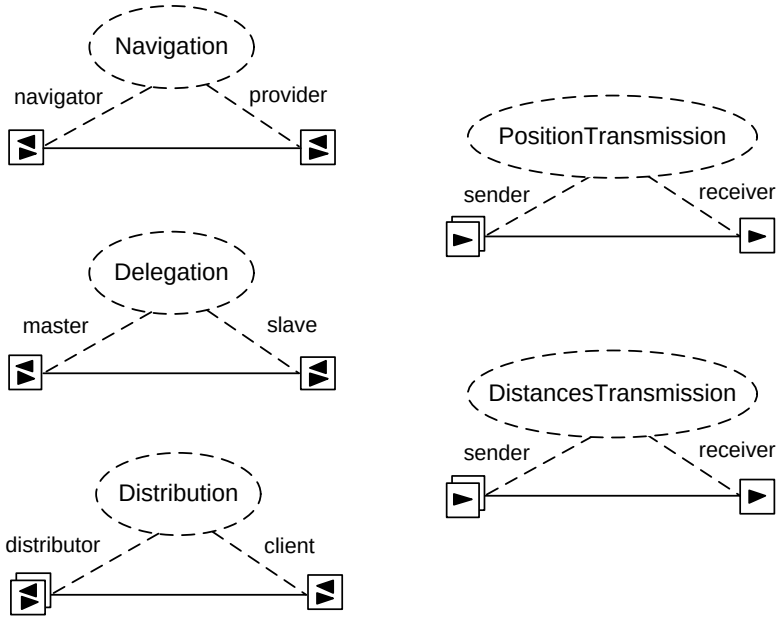


Figure 3.1.: The Five Real-Time Coordination Patterns used in the BeBot Scenario

3.1.1.1. Role Navigator

Figure 3.2 shows the Real-Time Statechart of the role navigator. It consists of two states, the

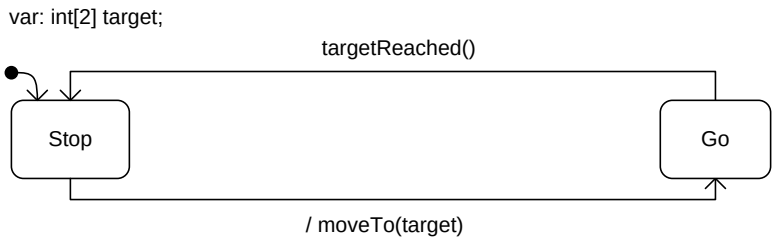


Figure 3.2.: Role Navigator of Pattern Navigation

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.

3.1.1.2. Role Provider

Figure 3.3 shows the Real-Time Statechart of the role provider. It represents the counter-part to the Real-Time Statechart in Figure 3.2 and consists of the two states Polling which is the initial state and Moving. The message moveTo triggers the transition from Polling to Moving.

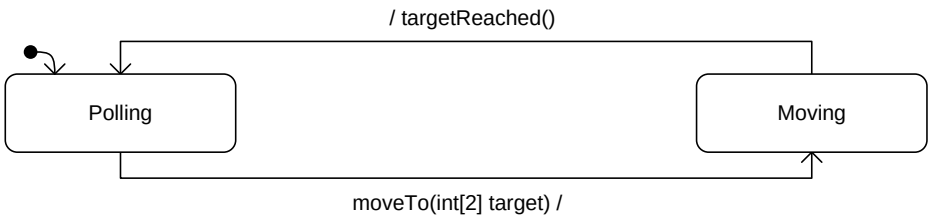


Figure 3.3.: Role Provider of Pattern Navigation

The message moveTo has a target position as a one-dimensional integer array with two entries as parameter. The transition from Moving back to Polling sends the message targetReached.

3.1.2. 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 3.2.2.

3.1.2.1. Role Master

Figure 3.4 shows the Real-Time Statechart of the role master. It consists of the initial state

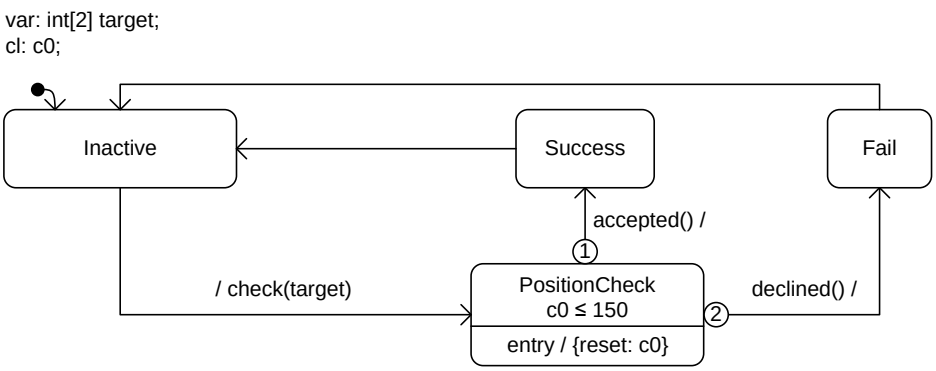


Figure 3.4.: Role Master of Pattern Delegation

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.

3.1.2.2. Role Slave

Figure 3.5 shows the Real-Time Statechart of the role slave. It represents the counter-part to

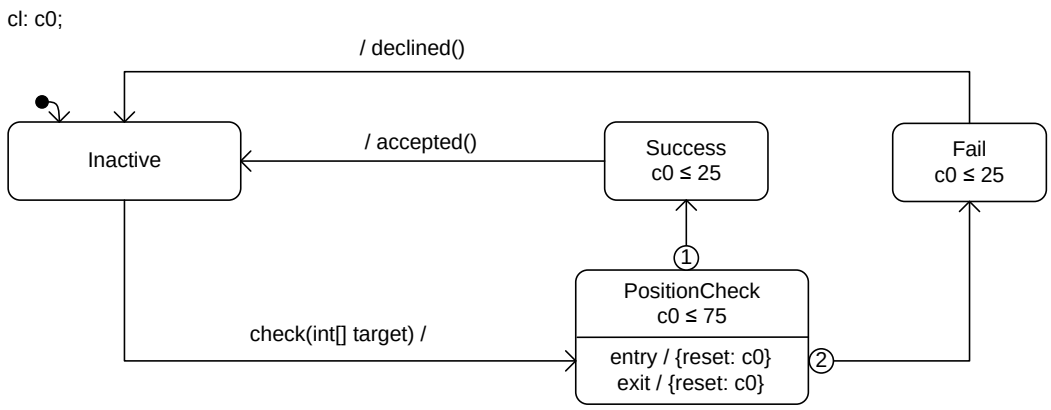


Figure 3.5.: Role Slave of Pattern Delegation

the Real-Time Statechart in Figure 3.4 and consists of the same states. The message check trigger 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.

3.1.3. 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. We describe the message interfaces which the roles use in Section 3.2.3.

3.1.3.1. Role Distributor

Figure 3.6 shows the Real-Time Statechart of the multi-role distributor. It specifies the behavior of a multi-role so it follows the convention that there is only one initial composite state, Distribution_distributor, with the two regions adaptation and sub-role (cf. Section 2.2.5).

The two regions are synchronized via the synchronization channel array next with n+1 entries where n is the number of instances of the single-role client. It specifies a synchronization

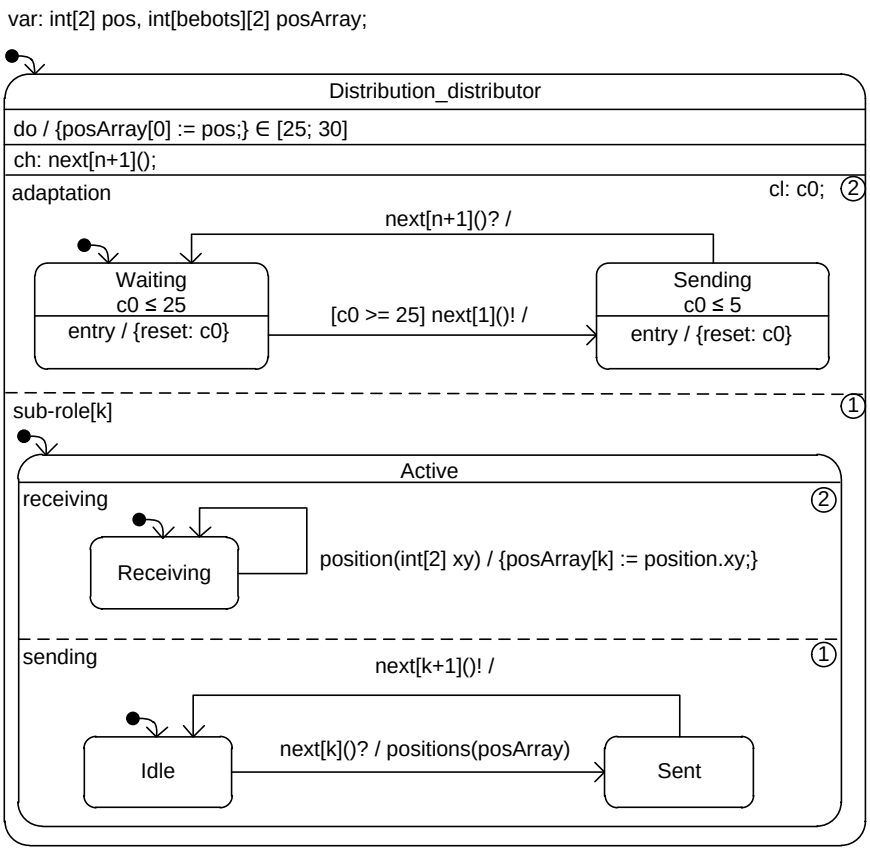


Figure 3.6.: Role Distributor of Pattern Distribution

chain between the regions adaptation and sub-role. The chain starts and stops in adaptation by sending a synchronization via next[1] and receiving one via next[n+1]. The synchronizations in region sub-role use the channels in between to receive synchronizations via next[k] and send them via next[k+1]. The variable k is the index of the current instance of this multi-region. The multi-region is instantiated for each of the instances of the multi-role.

The actual behavior is described in the following. A do-action of Distribution_distributor ensures that the one-dimensional integer array pos, which represents the own position with its two coordinate-entries, is set as the first entry of the two-dimensional integer array posArray each 25 to 30 units of time.

The region adaptation contains the initial state Waiting and the state Sending. Waiting has an entry action which resets the clock c0 and an invariant which uses c0 to ensure that it is left no later than 25 units of time after its activation. If the value of c0 is greater or equal to 25 units of time Waiting is left towards Sending and a synchronization is send via channel next[1]. Sending also resets c0 upon entry and has to be left no later than 5 units of time after its activation. A synchronization via next[n+1] triggers the transition back to Waiting.

The index k of the multi-role parameterize the multi-region sub-role. It contains only the initial composite state Active with the two regions receiving and sending. Inside receiving there is only the state Receiving with a self-transition which the message position triggers and sets the received parameter xy as the k-th entry of two two-dimensional integer array posArray. Inside sending there are the initial state Idle and the state Sent. The synchronization via the channel next[k] triggers the transition from Idle to Sent and sends the message positions with the two-dimensional integer array posArray, which contains the own and the collected positions, as parameter. The transition back to Idle sends a synchronization via channel next[k+1].

3.1.3.2. Role Client

Figure 3.7 shows the Real-Time Statechart of the role client. It represents the counter-part to

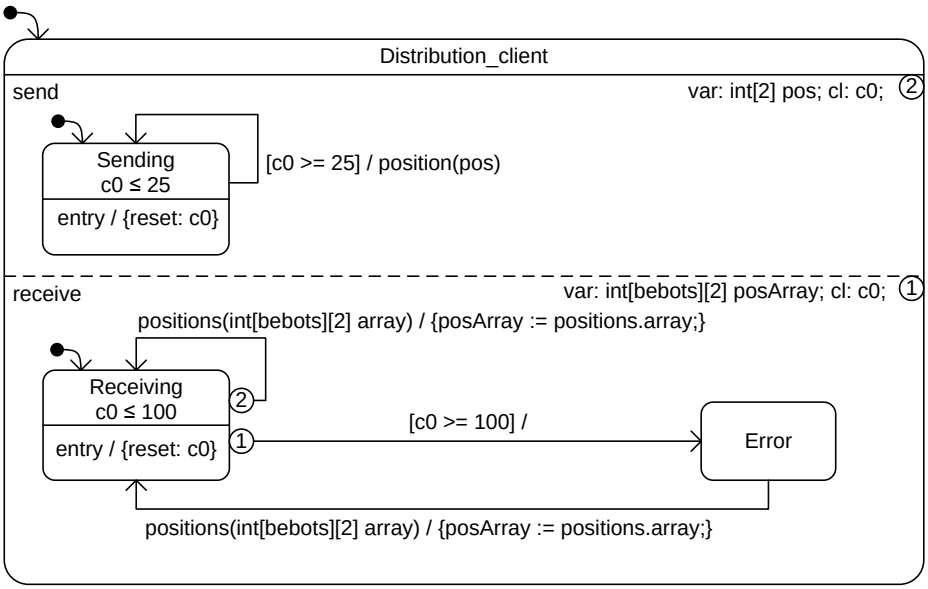


Figure 3.7.: Role Client of Pattern Distribution

the Real-Time Statechart in Figure 3.6, more precisely to the state Active in region sub-role, and consists of the initial composite state Distribution_client with its two regions send and receive.

The region send contains only the initial state Sending which resets the clock c0 via an entry-action and has to be left no later than 25 units of time after its entry by the use of an invariant. This happens after 25 or more units of time via a self-transition that sends the message position with the one-dimensional integer array pos as parameter.

The region receive contains the initial state Receiving and the state Error. A reset of clock c0 and an invariant of Receiving ensure that it is left no later than 100 units of time after

its activation via one of its two outgoing transitions. The higher prioritized one is a self-transition which is triggered by the message positions and sets the received two-dimensional integer array array as new value of the variable posArray. The other transition activates Error after 100 or more units of time after activation of Receiving. As soon as the message positions is received, posArray is updated to the received array parameter and Receiving is activated again.

3.1.4. 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 3.2.4.

3.1.4.1. Role Sender

Figure 3.8 shows the Real-Time Statechart of the multi-role sender. Like the Real-Time Stat-

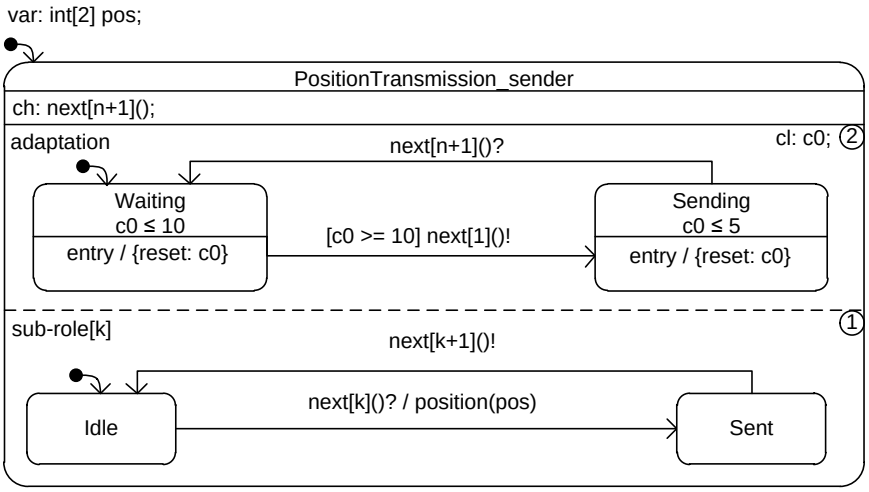


Figure 3.8.: Role Sender of Pattern PositionTransmission

echart in Figure 3.6 it follows the convention that there is only one initial composite state, PositionTransmission_sender, with the two regions adaptation and sub-role (cf. Section 2.2.5).

The region adaptation is nearly identical to the one in Figure 3.6, 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.

3.1.4.2. Role Receiver

Figure 3.9 shows the Real-Time Statechart of the role receiver. It represents the counter-part

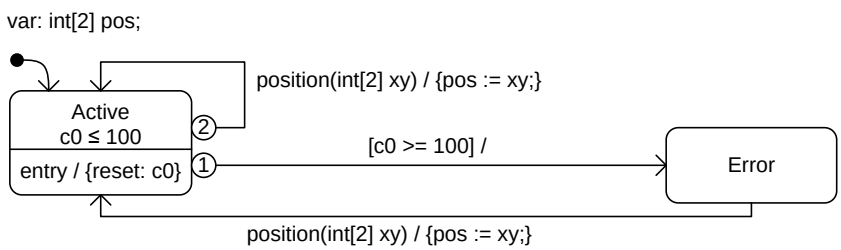


Figure 3.9.: Role Receiver of Pattern PositionTransmission

to the Real-Time Statechart in Figure 3.8, more precisely to the region sub-role, and is nearly identical to the region receive of the Real-Time Statechart in Figure 3.7. The only differences are the different message position which is received and the different parameter xy which is set to the variable pos.

3.1.5. 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 3.2.5.

3.1.5.1. Role Sender

Figure 3.10 shows the Real-Time Statechart of the multi-role sender. It is nearly identical to

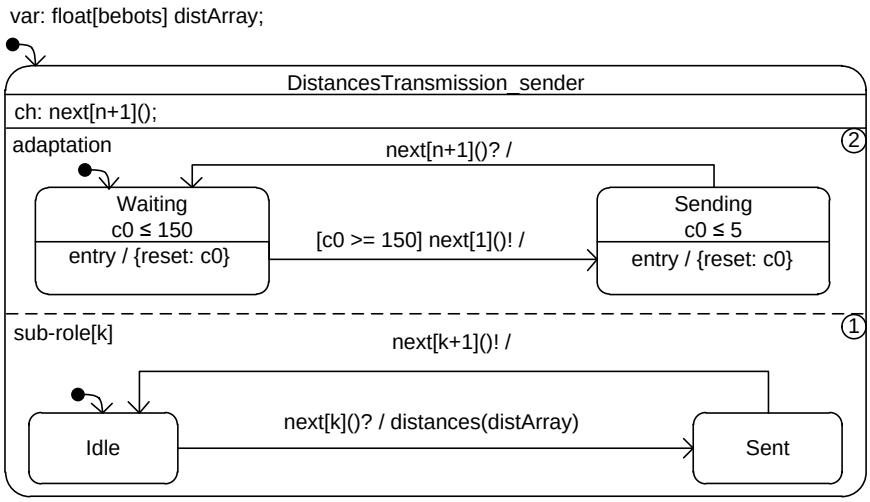


Figure 3.10.: Role Sender of Pattern DistancesTransmission

the Real-Time Statechart in Figure 3.8 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.

3.1.5.2. Role Receiver

Figure 3.11 shows the Real-Time Statechart of the role receiver. It represents the counter-part

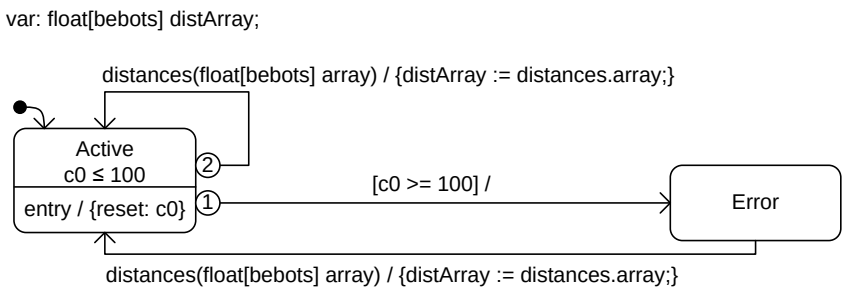


Figure 3.11.: Role Receiver of Pattern DistancesTransmission

to the Real-Time Statechart in Figure 3.10 and is nearly identical to the Real-Time Statechart in Figure 3.8. 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.

3.2. Message Interface Specification

In this section, we introduce the message interfaces which the Real-Time Coordination Patterns of Section 3.1 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.

3.2.1. 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 3.12 shows the two message interfaces needed for this pattern.

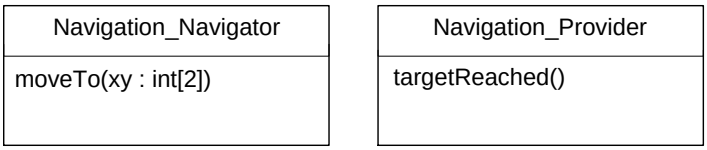


Figure 3.12.: Message Interfaces for the Navigation Pattern

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.

3.2.2. Delegation Pattern

We use the Delegation pattern to delegate a task to the slave which reports the success of the task execution. Figure 3.13 shows the two message interfaces which the pattern uses.

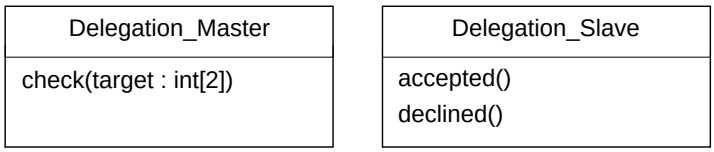


Figure 3.13.: Message Interfaces for the Delegation Pattern

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.

3.2.3. Distribution Pattern

The Distribution pattern transmits position data between the BeBots. Figure 3.14 shows the two message interfaces used by the pattern.

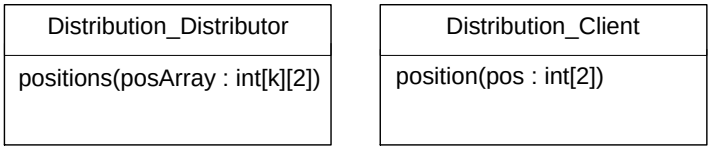


Figure 3.14.: Message Interfaces for the Distribution Pattern

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.

3.2.4. 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 3.15 shows.

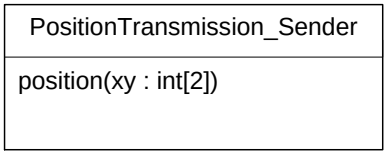


Figure 3.15.: Message Interfaces for the PositionTransmission Pattern

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.

3.2.5. DistancesTransmission Pattern

The DistanceTransmission 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 3.16 shows.

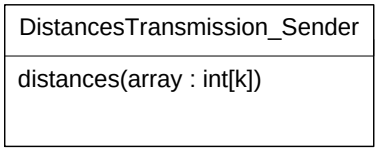


Figure 3.16.: Message Interfaces for the DistancesTransmission Pattern

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.

3.3. 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 3.3.1. The structured components are composed by a set of atomic components which we introduce in detail in Section 3.3.2. 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 3.1.

3.3.1. 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 3.17 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 velocity 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.

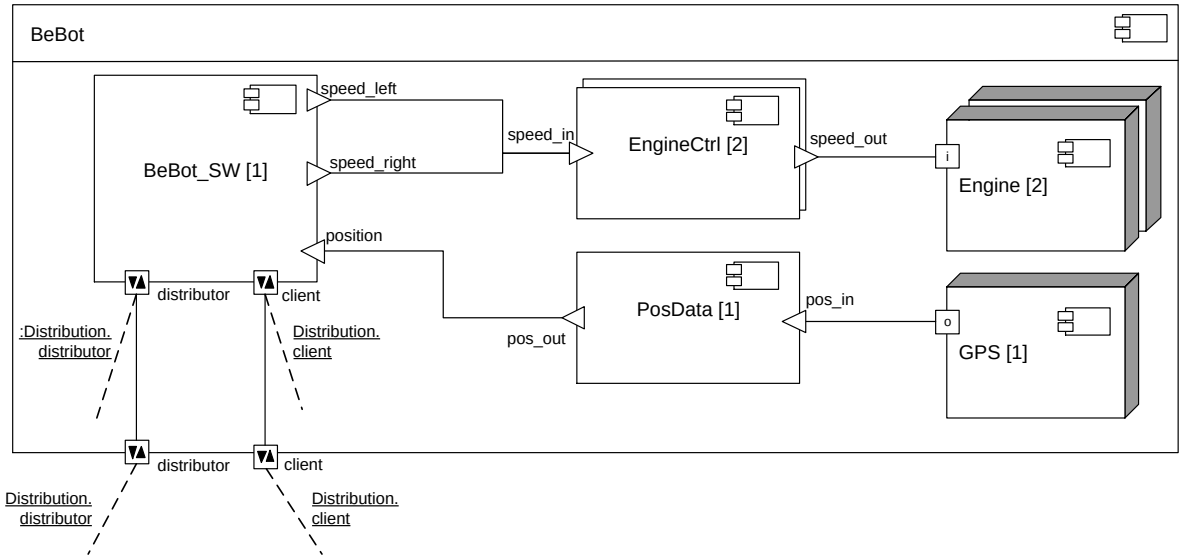


Figure 3.17.: Structured Component Defining the BeBot

Additionally, the BeBot contains two hardware component parts. The Engine component part represents the two chain drives of the BeBot while the GPS part represents the GPS hardware. These two parts are contained in the model in order to analyze the effects of hardware failures on the software models [PSWTH11].

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 2.5 on Page 15.

Figure 3.18 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 3.1. This corresponds to Step 4.3.2 of the development process as shown in Figure 2.6 on Page 2.6.

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

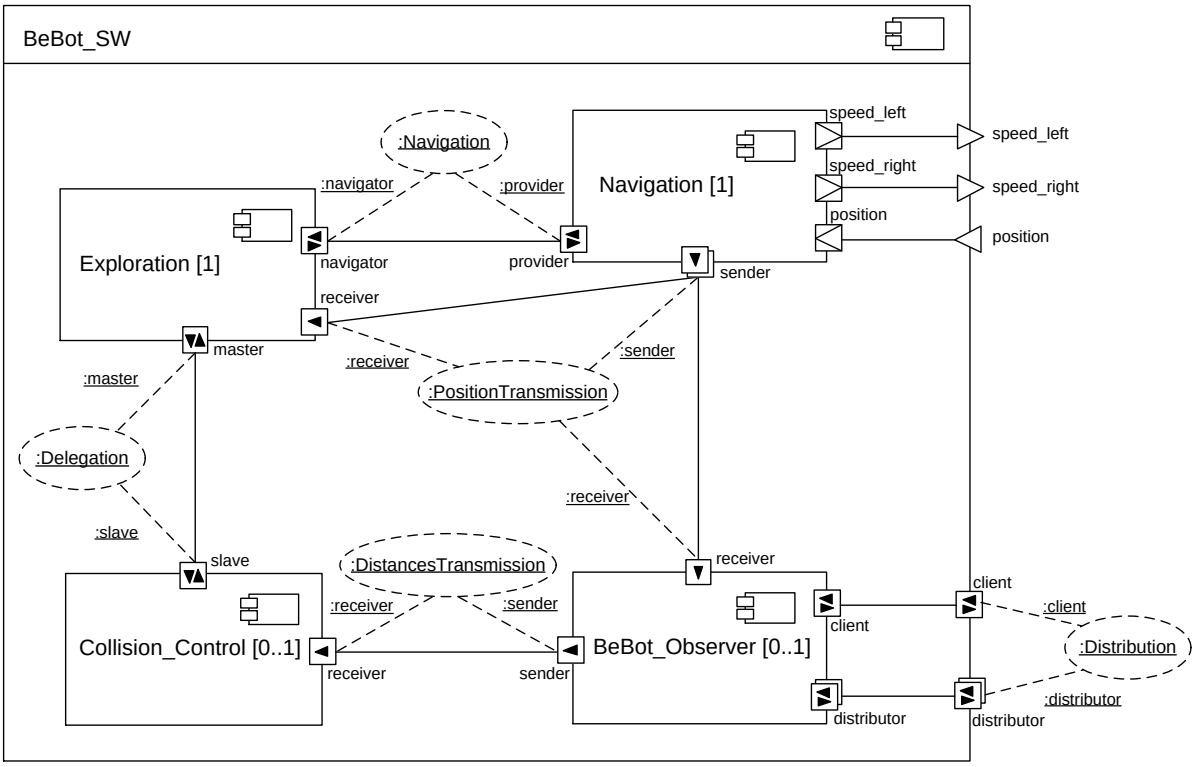


Figure 3.18.: Component Diagram of the BeBot Software

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.

3.3.2. Atomic Components

In our example, we use the eight atomic components shown in Figure 3.19 for modeling the BeBot. We use three discrete atomic components, one hybrid atomic component, two continuous atomic components, and two hardware atomic components. We introduce the general purpose of the continuous and hardware components in Section 3.3.1. Since we do not model any behavior for these kinds of atomic components, we do not introduce them in detail in this section. Instead, we focus on the three discrete atomic components Exploration, BeBot_Observer, and Collision_Control as well as the hybrid atomic component Navigation. We describe in the following sections these atomic components. We explain their behavior in the Section 3.4.

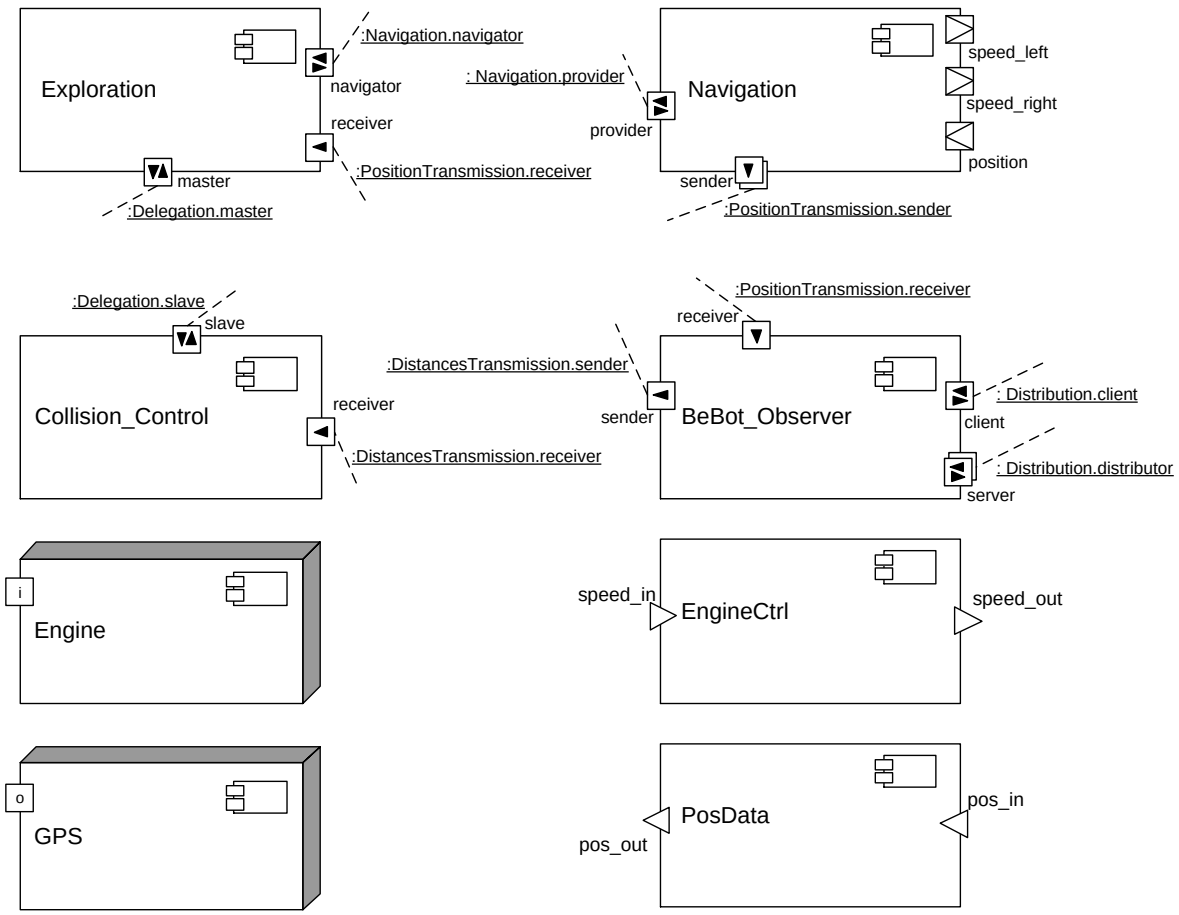


Figure 3.19.: The Atomic Components of the BeBot

3.3.2.1. 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 3.4.1.

3.3.2.2. 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 speeds 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 3.4.2.

3.3.2.3. 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 3.4.3.

3.3.2.4. 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 3.4.4.

3.4. 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 3.3.2. Real-Time Statecharts specifies the behavior of each component.

3.4.1. 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 3.20 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 3.3.1, 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 3.9). 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 3.2). 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 3.4). 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

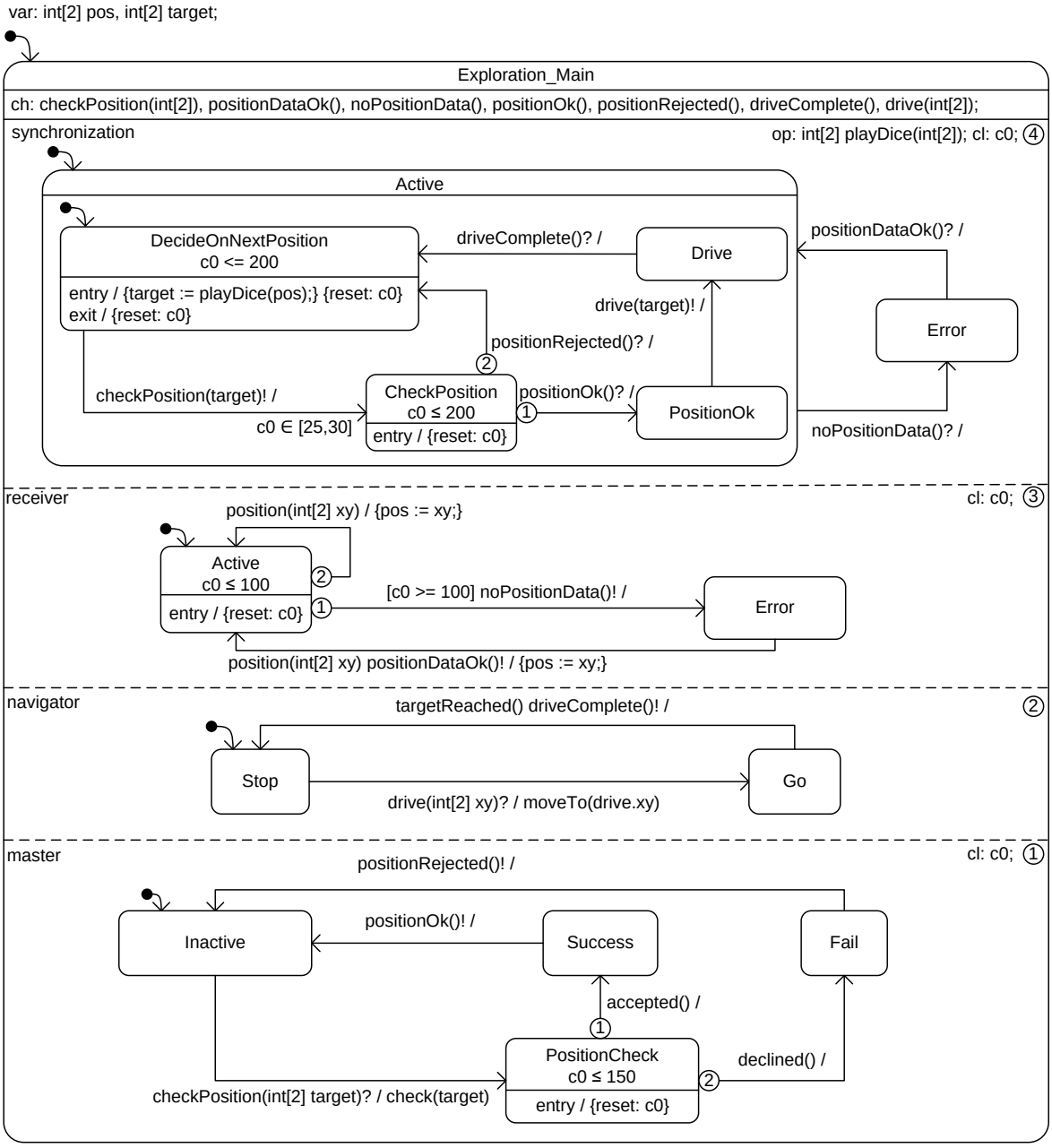


Figure 3.20.: Real-Time Statechart of Component Exploration

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.

3.4.2. 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 3.21 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 component ports and one Real-Time Statechart to synchronize them. Therefore Navigation_Main declares the channels finished, without parameters, and go, with an one-dimensional integer array with two entries, representing a new target position, as parameter.

As described in Section 3.3.1, 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 3.3). 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.

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 3.8). 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 does not have to be synchronized with the other regions, it is only refined by using the input position as parameter to the equally named message.

The region synchronization contains the three states Stop, Calculate and Move. Stop is the initial state which sets the outputs speed_left and speed_right to zero. 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 calcSpeed, with the current position and the target position as parameter. It calculates the needed movement to reach the target and the according speed values for

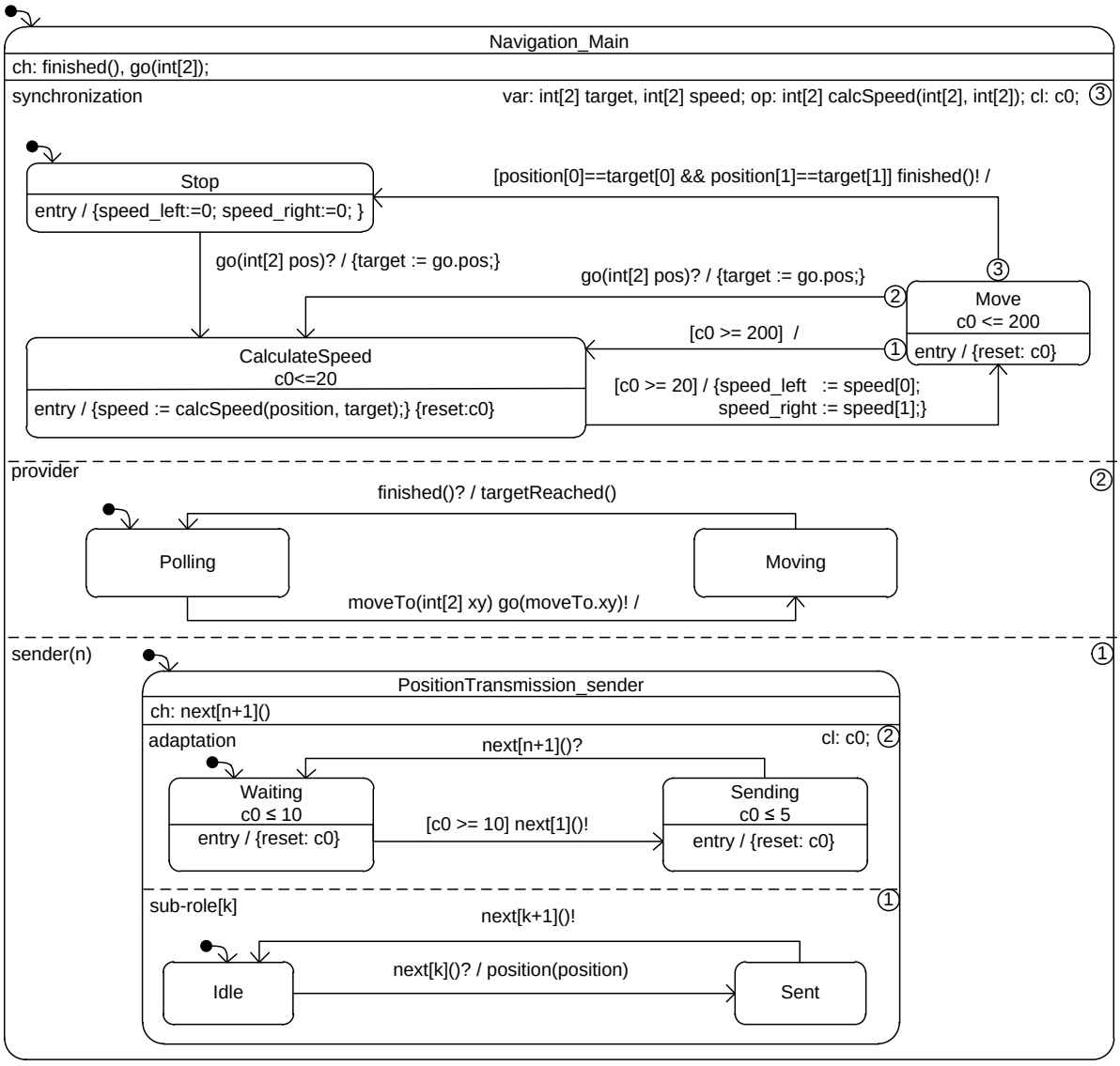


Figure 3.21.: Real-Time Statechart of Component Navigation

the left and the right wheel 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 Move sets the values in speed to the outputs speed_left and speed_right. 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 back to the initial state Stop has the highest priority, checks if the first two entries of position and target are identical and sends a synchronization via channel finished. The other two transitions 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.

3.4.3. 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 3.22 shows the Real-Time Statechart of BeBot_Observer. Since it follows the standard structure for a Real-Time Statechart of a discrete component, it only consists of an initial composite state, BeBot_Observer_Main, whose parallel regions contain the Real-Time Statecharts of the component ports and one Real-Time Statechart to synchronize them. Therefore BeBot_Observer_Main declares the channels commOk, commError, ok and error all without parameters.

As described in Section 3.3.1, 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 3.9). 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 3.10). 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 3.7). 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-

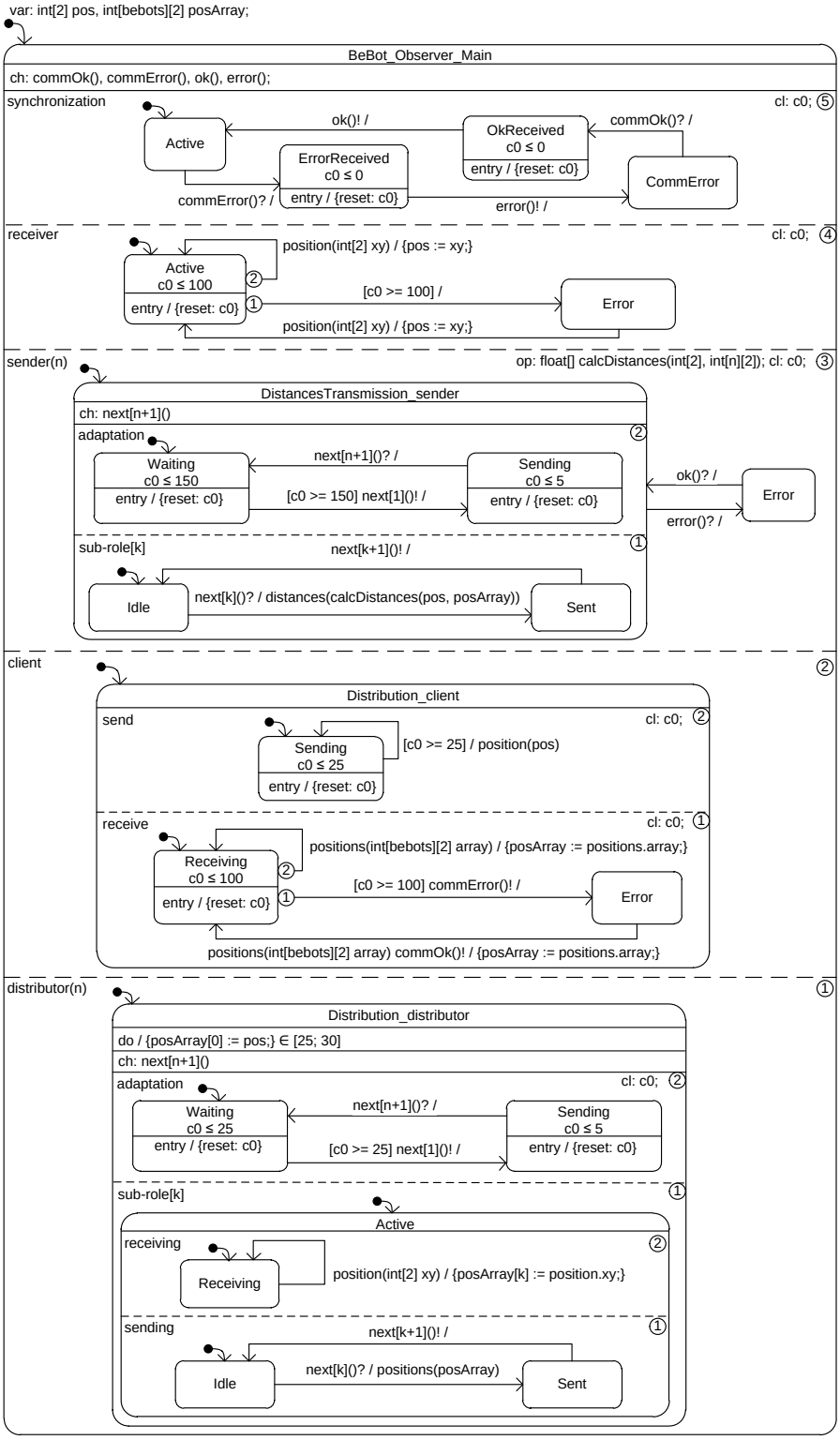


Figure 3.22.: Real-Time Statechart of Component BeBot Observer

Time Statechart of the role (cf. Figure 3.6). 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.

3.4.4. 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 3.23 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 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 3.3.1, 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 3.11). 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 3.5). 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

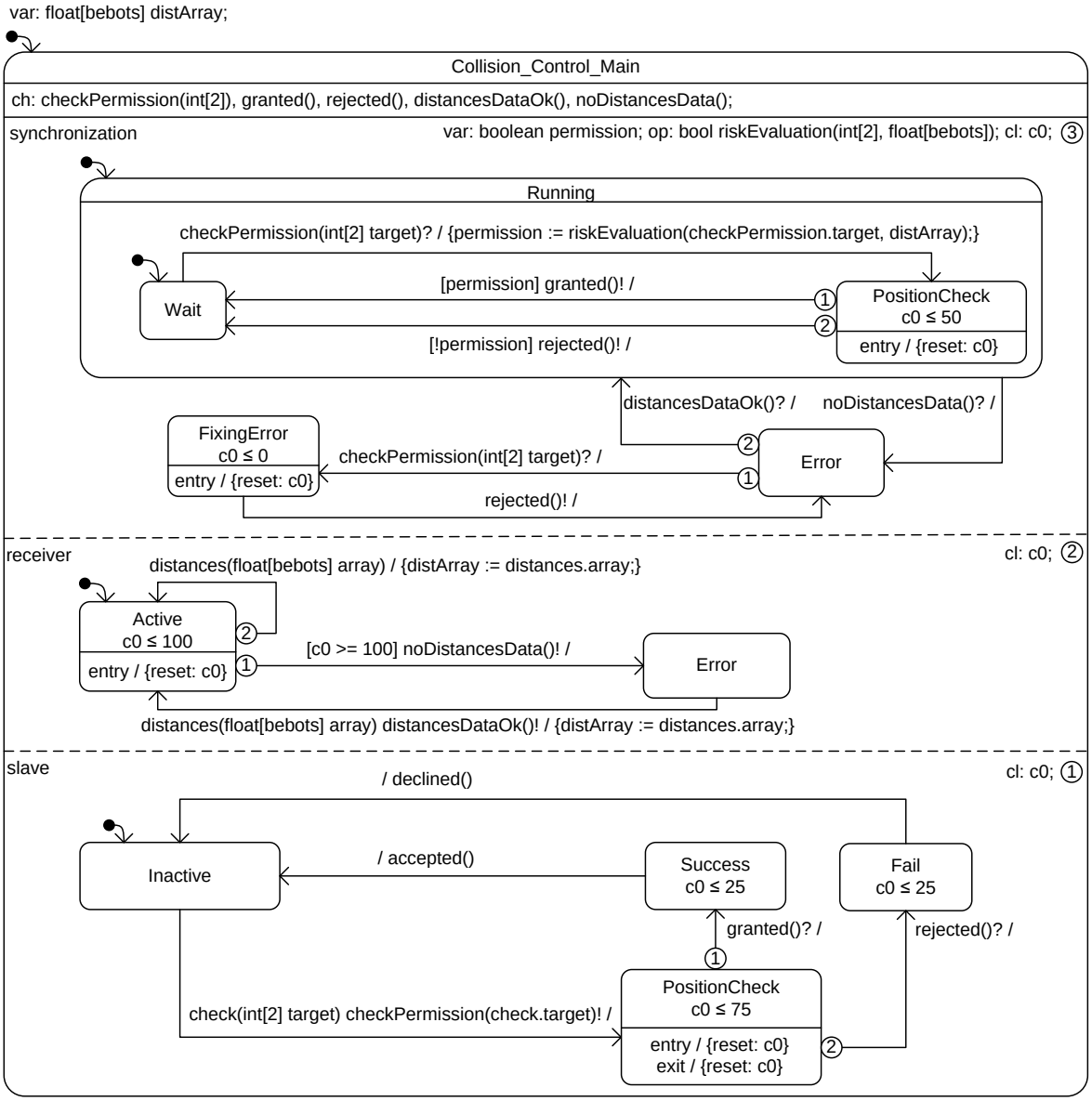


Figure 3.23.: Real-Time Statechart of Component Collision Control

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.

3.5. Component Instance Configuration

In this section, we introduce the component instance configurations of our example. First, we show in Section 3.5.1 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 3.5.2 contains a description of component instance configurations for several BeBots exploring the environment.

3.5.1. 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 3.24 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.

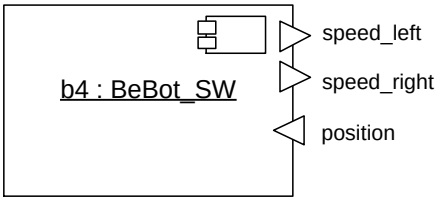


Figure 3.24.: Concrete Syntax of a Component Instance of the Component Type BeBot_SW that is not Connected to other BeBots

Figure 3.25 shows the structure of the embedded component instances of the BeBot of Figure 3.24. 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 3.18. 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 3.1). 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.

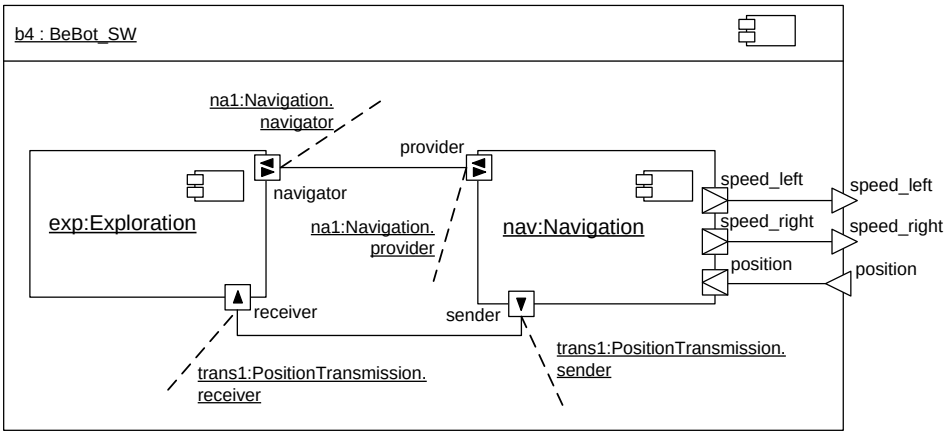


Figure 3.25.: Concrete Syntax of a Component Instance Configuration of a Single BeBot

3.5.2. 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 3.24, the component instances of the component BeBot_SW contain additional discrete ports for exchanging the position data. Figure 3.26 shows an instance of the discrete BeBot software for a BeBot operating as the position distributor for two client BeBots. Figure 3.27 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 3.1).

Component instance b1:BeBot_SW of Figure 3.26 has the two discrete port instances distributor1 and distributor2 that both implement the multi-role distributor of the Real-Time Coordination Pattern Distribution. Thus, this BeBot software executes the behavior of a position distributor BeBot. Each of the two port instances is connected to a client BeBot as shown in Figure 3.28.

Component instance b2:BeBot_SW of Figure 3.27 has a discrete port instance client that implements the single-role client of the Real-Time Coordination Pattern Distribution. Thus, this BeBot executes the behavior of a client to the position distributor.

The component instances shown in Figure 3.29 and 3.30 only represent the BeBot_SW. In order to obtain a component instance specification for a BeBot, the component type BeBot of Figure 3.17 must be instantiated and connected with other instances of type BeBot. Figure 3.28 shows a component instance configuration consisting of the three component instances of type BeBot, namely bebot1:BeBot, bebot2:BeBot and bebot3:BeBot. bebot1:BeBot

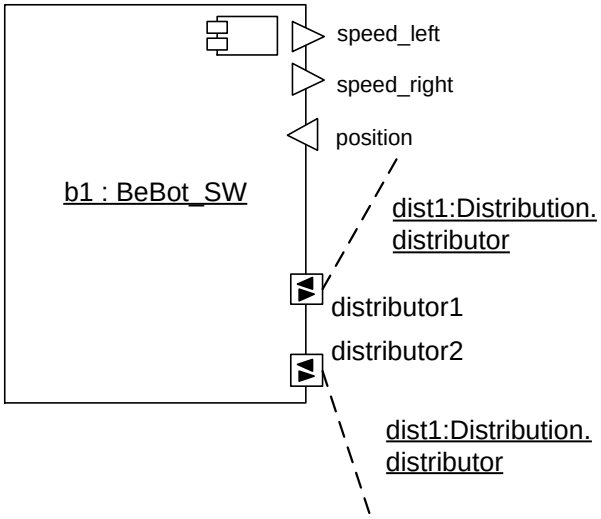


Figure 3.26.: Concrete Syntax of a Component Instance of the Component Type BeBot

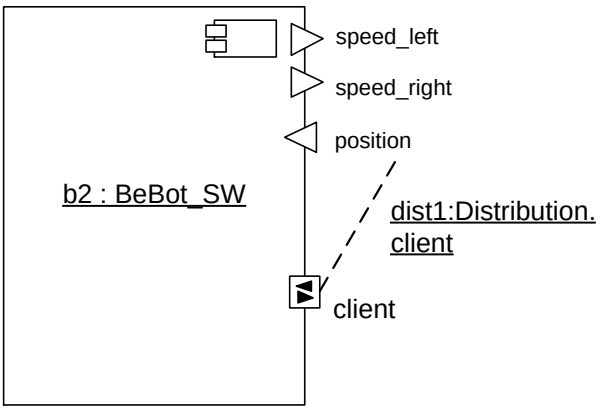


Figure 3.27.: Concrete Syntax of a Component Instance of the Component Type BeBot

is the position distributor while bebot2:BeBot and bebot3:BeBot operate as clients. Thus, the assembly connector instances which is derived from the Real-Time Coordination Pattern Distribution connects the bebot2:BeBot and the bebot3:BeBot to bebot1:BeBot. The discrete port instances distributor1 and distributor2 of bebot1:BeBot implement the multi-role distributor. The discrete port instances client of bebot2:BeBot and bebot3:BeBot implement the single-roles client.

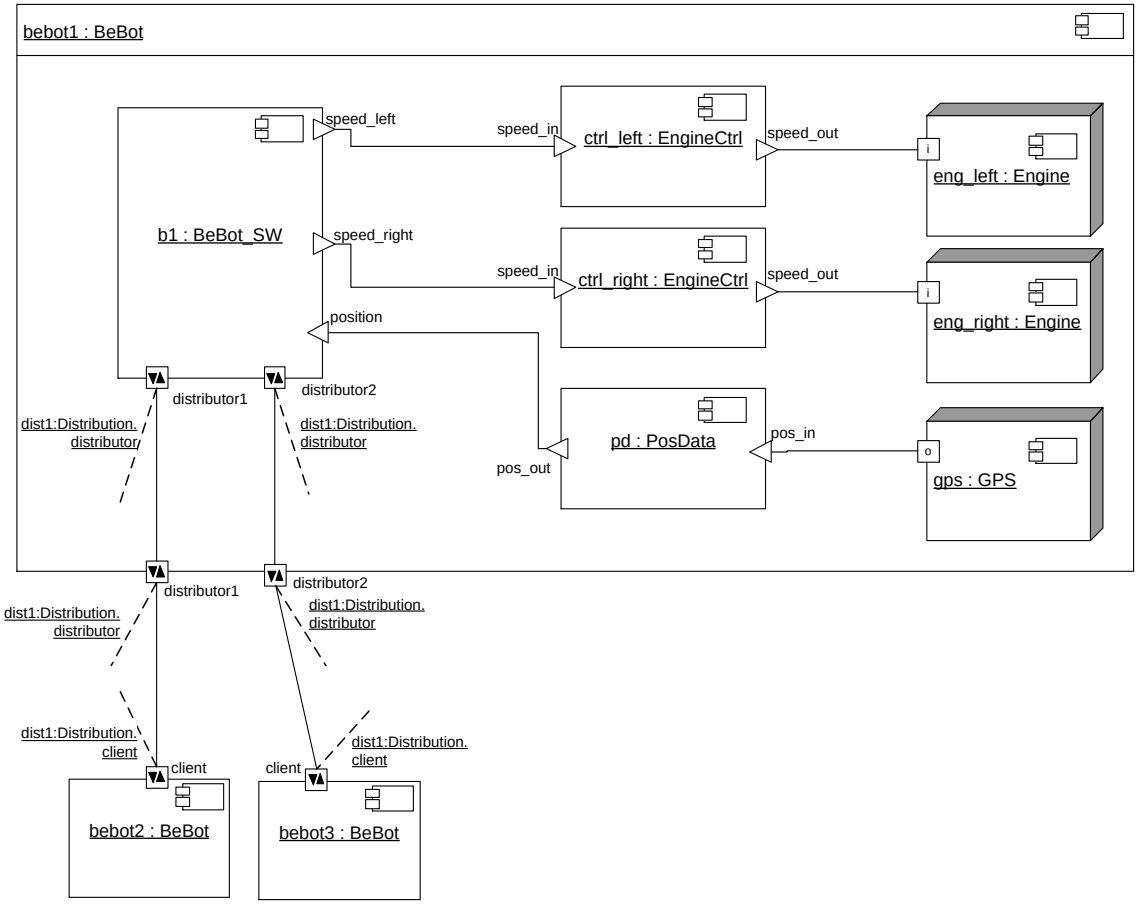


Figure 3.28.: Concrete Syntax of a Component Instance Configuration of Three Communicating BeBots

Figure 3.28 also shows the embedded component instances of bebot1:BeBot: b1:BeBot_SW, ctrl_left:EngineCtrl, ctrl_right:EngineCtrl, pd:PosData, eng_left:Engine, eng_right:Engine, and gps:GPS. The two discrete port instances distributor1 and distributor2 of the component instance b1:BeBot_SW, that implement the role distributor of the Real-Time Coordination Pattern Distribution, are delegated to the discrete port instances distributor1 and distributor2 of bebot1:BeBot. Component instance ctrl_left:EngineCtrl and component instance ctrl_right:EngineCtrl each have the two continuous port instances speed_in and speed_out. Component instance pd:PosData has

the two continuous port instances pos_in and pos_out. The hardware component instances eng_left:Engine and eng_right:Engine each have an incoming hardware port. The hardware component instance gps:GPS has an outgoing hardware port. The assembly 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 3.29 and Figure 3.30 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 3.25. 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 3.29 acts as a distributor. It communicates to other component instances of the type BeBot_SW via a multi-port that implements the multi-role distributor of the Real-Time Coordination Pattern Distribution. In our example, b1:BeBot_SW has two discrete port instances distributor1 and distributor2 that implement the multi-role distributor. This implies that b1:BeBot can distribute data to two client component instances of the type BeBot_SW. A delegation connector instance delegates these discrete port instances to the two discrete port instances distributor1 and distributor2 of the component instance bbo:BeBot_Observer. Further it implements the multi-role distributor of the Real-Time Coordination Pattern Distribution.

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

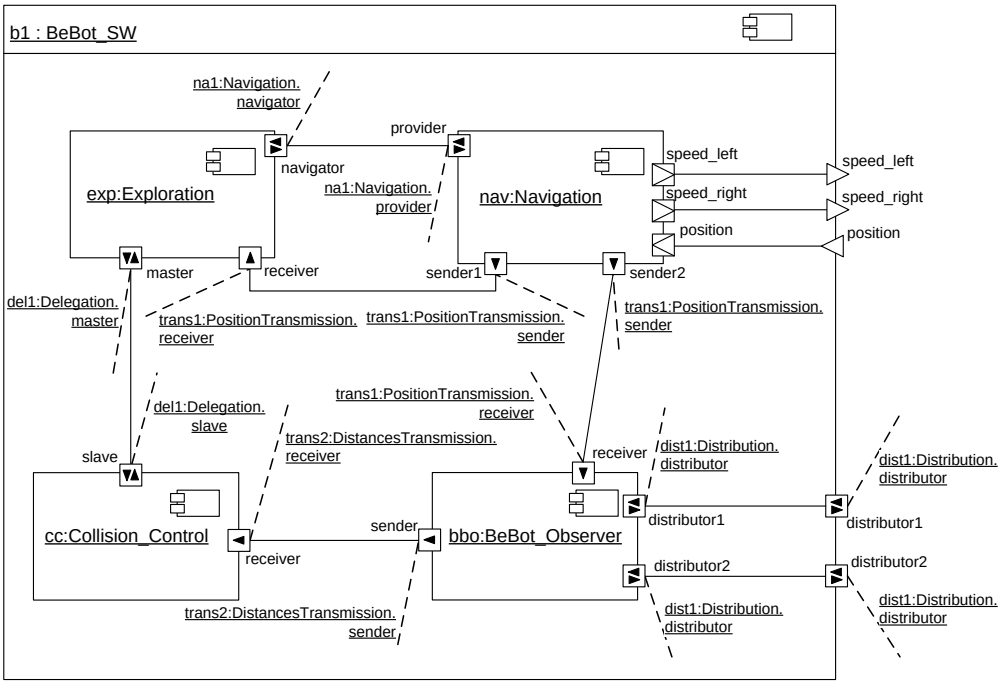


Figure 3.29.: Concrete Syntax of a Component Instance Distributor

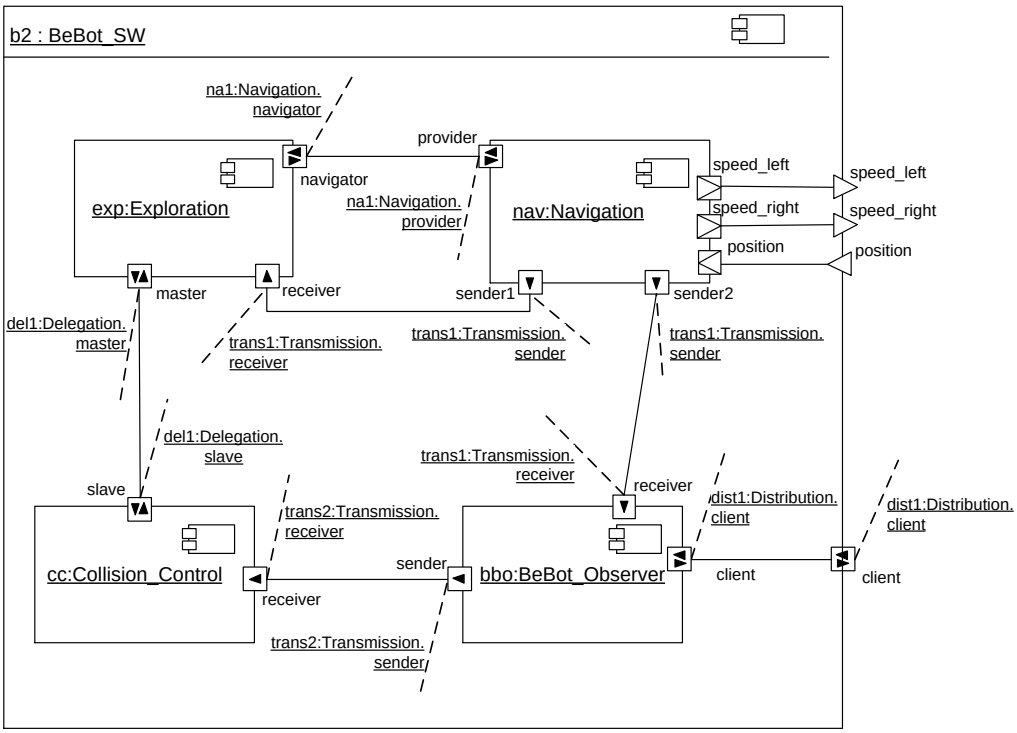


Figure 3.30.: Concrete Syntax of a Component Instance Client

Chapter 4.

Related Work

MECHATRONICUML 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 MECHATRONICUML is mainly focused on the software engineering aspects, it nevertheless reflects the relationships to the other engineering domains to some extend. Examples are continuos 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, EAST-ADL, or MATLAB/Simulink with Stateflow.
- Process models for system 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.

4.1. Specification Languages for Systems Engineering

There are several approaches for holistic and integrated modeling or specification languages for systems engineering approaches. 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]. MATLAB/Simulink and Modelica focus on an integrated analysis and simulation of a mechatronic system while MECHATRONICUML only focusses on specification of the discrete real-time software. Although we model the integration with the remaining system elements, they are not included in our models at that level of detail.

4.2. Process Models for System Engineering

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

4.3. Software Component Models

The surveys [LW07], [CCSV07], and [HPB+10] review different component models. In [LW07] and [CCSV07] general purpose component models like e.g. CORBA, EJB, or SOFA are reviewed as well as component models for the development of embedded real-time systems like e.g. Progress [VSC+09, BCC+08], Robocop [BdWCM05], or SOFA HI [PWT+08]. The survey [HPB+10] focuses on component models for the development of embedded real-time systems, only. General purpose component models (like CORBA, EJB, ...), which are typically intended to model non-real-time systems, do not fit in the considered class of systems as real-time aspects are not considered.

In the area of component models for embedded real-time systems, Robocop is probably closest to our approach. It focuses on the schedulability analysis of real-time systems but supports static architectures, only. SOFA HI is an extension of the SOFA component model for real-time systems. It provides extensive reuse capabilities and development support as well as the possibility to change the inner structure of a component at runtime. Additionally, SOFA HI supports checking compliance of behavior protocols and the component implementation. However, that does not include 1:n communications with dependencies or relaxation of time intervals. The Progress component model targets automotive applications and provides passive, signal based components which can be structured hierarchically. Thus, the concepts cannot be applied directly to our active, message based components.

[1] http://www.mathworks.de/

4.4. Specifications of reconfigurable systems

A general approach towards the development and specification of dynamic architectures is presented in [ZC06]. It uses reconfiguration rules to specify transitions between different reconfiguration states.

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 three main classes: graph-transformation based languages, process algebra based languages and formal logics based languages.

There are different 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]), which represents a system as molecules and its evolution via chemical reactions.

Approaches which utilize process algebras include Darwin [MK96], LEDA [CPT99] and Dynamic Wright [ADG98]. Dynamic Wright specifies the behavior of ports by CSP processes [Hoa85], while LEDA and Darwin use the π-calculus [MPW92].

Gerel [EW92] as an example for a formal logics based language uses first order predicate logic to define preconditions for rule applications. Rules itself are then specified as scripts. The approach by Aguirre et. al. [AM02] specifies component behaviors in temporal logic expressions.

4.5. 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. For example, there is a survey on different types of acceptance conditions in timed automata [AD94]. 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.

Chapter 5.

Conclusions and Future Work

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

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

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

Acknowledgments

MECHATRONICUML is the result of many people's work which we are greatly indebted to. MECHATRONICUML has been started by Holger Giese, now Professor at the Hasso Plattner Institut at the University of Potsdam, while he was with the Software Engineering Group at the University of Paderborn. He designed the MECHATRONICUML approach and developed the foundations. Thus, he greatly influenced the current state of MECHATRONICUML. Additional work has been done by the following doctoral students during their time at the 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.

Bibliography

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. 25

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science (LNCS)*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 99

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 5, 9, 28, 37, 99

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382:21–37, 1998. 99

[AM02] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. *Automated Software Engineering, International Conference on*, 0:271–274, 2002. 99

[BCC+08] Tomas Bures, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. Procom - the progress component model reference manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008. 98

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM. 99

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag, 2004. 25, 28

[BdWCM05] Egor Bondarev, Peter de With, Michel Chaudron, and Johan Muskens. Modelling of input-parameter dependency for performance predictions of component-based embedded systems. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '05, pages 36–43, Washington, DC, USA, 2005. IEEE Computer Society. 98

[BGH+07] S. Burmester, H. Giese, S. Henkler, M. Hirsch, M. Tichy, A. Gambuzza, E. Münch, and H. Vöcking. Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camel-view. In *Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA*, 2007. 43

[BGS05] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, Lecture Notes in Computer Science, pages 25–40. Springer Verlag, November 2005. 101

[BGT05] Sven Burmester, Holger Giese, and Matthias Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronic uml. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 47–61. Springer Berlin / Heidelberg, 2005. 2, 12

[Bur02] Sven Burmester. Generierung von Java Real-Time Code für zeitbehaftete UML Modelle. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, September 2002. 2

[Bur06] Sven Burmester. *Model-Driven Engineering of Reconfigurable Mechatronic Systems*. PhD thesis, Universität Paderborn, Berlin, Deutschland, 2006. 2, 102

[BW07] Lionel C. Briand and Alexander L. Wolf, editors. *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, 2007. 109, 110

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. 99

[CCSV07] Ivica Crnkovic, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. A classification framework for component models. In *Proceedings of the 7th*

Conference on Software Engineering and Practice in Sweden, October 2007. 98

[CdLG+09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009. 1

[CPT99] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 107–126, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V. 99

[DM01] Alexandre David and M. Oliver Möller. From huppaal to uppaal: A translation from hierarchical timed automata to flat timed automata. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, March 2001. 28

[DMY02a] Alexandre David, M. Möller, and Wang Yi. Formal verification of uml statecharts with real-time extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 208–241. Springer Berlin / Heidelberg, 2002. 29

[DMY02b] Alexandre David, M. Oliver Möller, and Wang Yi. Formal verification of uml statecharts with real-time extensions. In *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 208–241. Springer Berlin / Heidelberg, 2002. 99

[Dre11] Jannis Drewello. Musterbasierter Entwurf für das Verhalten von BeBot-Robotern mit MechatronicUML. Bachelor thesis, University of Paderborn, 2011. in German. 101

[EH10] Tobias Eckardt and Stefan Henkler. Component behavior synthesis for critical systems,. In *Architecting Critical Systems*, volume 6150 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin / Heidelberg, 2010. 18, 49

[EHH+11] Tobias Eckardt, Christian Heinzemann, Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, and Wilhelm Schäfer. Modeling and verifying dynamic communication structures based on graph transformations. *Computer Science - Research and Development*, pages 1–20, 2011. 10.1007/s00450-011-0184-y. 2, 14, 24, 25, 37, 46, 101

[EW92] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *International Workshop on Configurable Distributed Systems*, pages 68–79, Mar 1992. 99

[FMMR10] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. Modeling time in computing: A taxonomy and a comparative survey. *ACM Computing Surveys (CSUR)*, 42(4):1–59, February 2010. 99

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 2000. 49

[Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons, 2004. 98

[GB03] Holger Giese and Sven Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Software Engineering Group, University of Paderborn, Paderborn, Germany, June 2003. 2, 28, 38, 101

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pages 179–188. acmpress, November 2004. 101

[GFDK09] J. Gausemeier, U. Frank, J. Donoth, and S. Kahl. Specification technique for the description of self-optimizing mechatronic systems. *Research in Engineering Design*, 20:201–223, 2009. 10.1007/s00163-008-0058-x. 6, 98

[GH06] Holger Giese and Stefan Henkler. A survey of approaches for the visual model-driven development of next generation software-intensive systems. In *Journal of Visual Languages and Computing*, volume 17, pages 528–550, December 2006. 97

[GHH+08] Holger Giese, Stefan Henkler, Martin Hirsch, Vladimir Roubin, and Matthias Tichy. Modeling techniques for software-intensive systems. In Dr. Pierre F.

Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*. IGI Global, Hershey, PA, 2008. 2

[Gie03] Holger Giese. A formal calculus for the compositional pattern-based design of correct real-time systems. Technical Report tr-ri-03-240, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Deutschland, July 2003. 2

[Gro10a] Object Management Group. *OMG Systems Modeling Language (OMG SysML)*, June 2010. Document formal/10-06-02. 97

[Gro10b] Object Management Group. *Unified Modeling Language (UML) 2.3 Superstructure Specification*, May 2010. Document formal/2010-05-05. 6, 26, 27, 28, 36, 49, 52

[GSA+11] Jürgen Gausemeier, Wilhelm Schäfer, Harald Anacker, Frank Bauer, and Stefan Dziwok. Einsatz semantischer Technologien im Entwurf mechatronischer Systeme. In *8. Paderborner Workshop Entwurf mechatronischer Systeme (EMS), Paderborn, Germany*, volume 294 of *HNI-Verlagsschriftenreihe*, pages 7–35. Heinz Nixdorf Verlag, May 2011. in German. 101

[GSG+09] Jürgen Gausemeier, Wilhelm Schäfer, Joel Greenyer, Sascha Kahl, Sebastian Pook, and Jan Rieke. Management of cross-domain model consistency during the development of advanced mechatronic systems. In *Proceedings of the 17th International Conference on Engineering Design (ICED'09)*, 2009. 9

[GST+03] Holger Giese, Daniela Schilling, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. Technical Report tr-ri-03-241, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Deutschland, jul 2003. 2, 101

[GT06] Holger Giese and Matthias Tichy. Component-Based Hazard Analysis: Optimal Designs, Product Lines, and Online-Reconfiguration. In *Proc. of the 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP), Gdansk, Poland*, Lecture Notes in Computer Science. Springer Verlag, September 2006. 101

[GTB+03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the compositional verification of real-time uml designs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, pages 38–47. ACM Press, September 2003. 2, 14, 16, 18, 46, 50

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. 28

[HBB+09] S. Henkler, M. Breit, C. Brink, M. Böger, C. Brenner, K. Bröker, U. Pohlmann, M. Richtermeier, J. Suck, O. Travkin, and C. Priesterjahn. Fritscab: Fujaba re-engineering tool suite for mechatronic systems. In *Proceedings of the 7th International Fujaba Days*, 2009. 10, 14, 15

[HH11] Christian Heinzemann and Stefan Henkler. Reusing dynamic communication protocols in self-adaptive embedded component architectures. In *Proceedings of the 14th International Symposium on Component Based Software Engineering (CBSE-2011)*, June 2011. 14, 16, 18, 48

[HIM98] Dan Hirsch, Paolo Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 69–72, New York, NY, USA, 1998. ACM. 99

[Hir04] Martin Hirsch. Effizientes Model Checking von UML-RT Modellen und Real-time Statecharts mit UPPAAL. Master's thesis, University of Paderborn, June 2004. 2

[Hir08] Martin Hirsch. *Modell-basierte Verifikation von vernetzten mechatronischen Systemen*. PhD thesis, Universität Paderborn, Paderborn, Deutschland, 2008. 2, 102

[HMS+10] S. Henkler, J. Meyer, W. Schäfer, U. Nickel, and M. von Detten. Legacy component integration by the fujaba real-time tool suite. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010. 10, 14, 15

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985. 99

[HPB+10] Petr Hosek, Tomas Pop, Tomas Bures, Petr Hnetynka, and Michal Malohlava. Comparison of component frameworks for real-time embedded systems. In *Proceedings of CBSE 2010*, LNCS. Springer, 2010. 98

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21:373–386, 1995. 99

[Kil05] Christopher Kilian. *Modern Control Technology*. Delmar Cengage Learning, 3rd edition, 2005. 1, 43

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. 25

[LM98] D. Le Métayer. Describing software architecture styles using graph grammars. *Software Engineering, IEEE Transactions on*, 24(7):521–533, Jul 1998. 99

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709 –724, october 2007. 98

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996. 99

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992. 99

[Obj09] Object Management Group. *UML 2.2 Superstructure Specification*, 2009. Document – formal/09-02-02. 1, 5, 43

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Krüger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In Briand and Wolf [BW07], pages 55–71. 1

[PST11] C. Priesterjahn, D. Steenken, and M. Tichy. Component-based timed hazard analysis of self-healing systems. In *Proc. of the Workshop on Assurances for Self-Adaptive Systems at ESEC/FSE 2011, Szeged, Hungary*, September 2011. 101

[PSWTH11] Claudia Priesterjahn, Christoph Sondermann-Wölke, Matthias Tichy, and Christian Hölscher. Component-based hazard analysis for mechatronic systems. In *Proceedings of the 2nd IEEE International Workshop on Model-Based Engineering for Real-Time Embedded Systems Design, MoBE-RTES 2011*. IEEE Computer Society, USA, March 2011. 44, 77, 101

[PWT+08] Marek Prochazka, Roger Ward, Petr Tuma, Petr Hnetynka, and Jiri Adamek. A component-oriented framework for spacecraft on-board software. In *Proceedings of DASIA 2008, DAta Systems In Aerospace, Palma de Mallorca, European Space Agency Report Nr. SP-665*, May 2008. 98

[Rie11] Stefan Riepe. Zustandsbasierte Verhaltensmodellierung von Steuerungssystemen mit Entwurfsmustern. Bachelor thesis, University of Paderborn, June 2011. in German. 101

[Roz97] Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. 49

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2 edition, December 2008. 26

[Sch06] Daniela Schilling. *Kompositionale Softwareverifikation mechatronischer Systeme*. PhD thesis, Universität Paderborn, Paderborn, Deutschland, 2006. 102

[SSGR11] Wilhelm Schäfer, Oliver Sudmann, Joel Greenyer, and Jan Rieke. The mechatronic uml development process. In *Engineering of Software*. Springer Berlin Heidelberg, 2011. 2, 5

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000. 43

[Sta08] Florian Stallmann. *A Model-Driven Approach to Multi-Agent-Design*. PhD thesis, Universität Paderborn, Paderborn, Deutschland, 2008. 102

[SW07] Wilhelm Schäfer and Heike Wehrheim. The challenges of building advanced mechatronic systems. In Briand and Wolf [BW07], pages 72–84. 1

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998. 1, 43

[TGM00] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 179–193. Springer-Verlag, London, UK, 2000. 99

[THHO08] Matthias Tichy, Stefan Henkler, Jörg Holtmann, and Simon Oberthür. Component story diagrams: A transformation language for component structures in mechatronic systems. In *Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*, 2008. 101

[VDI04] VDI, Verein Deutscher Ingenieure. *VDI Guideline 2206, Design Methodology for Mechatronic Systems*. Berlin, 2004. 5, 98

[VSC+09] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the procom real-time component model. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:478–485, 2009. 98

[Zü01] Albert Zündorf. *Rigorous Object Oriented Software Development*. Habilitation, Software Engineering Group, University of Paderborn, 2001. 32

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM. 99

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.

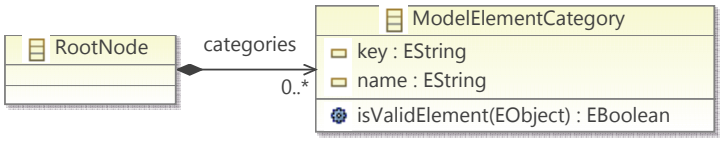


Figure A.1.: Meta-Model of the modelinstance Package

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.11.2.2 on Page 154

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:

```
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. Please see the contained sub-packages for 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. Please see the contained sub-packages for 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.

Parent Classes

- ConnectorType see Section A.4.2.7 on Page 120

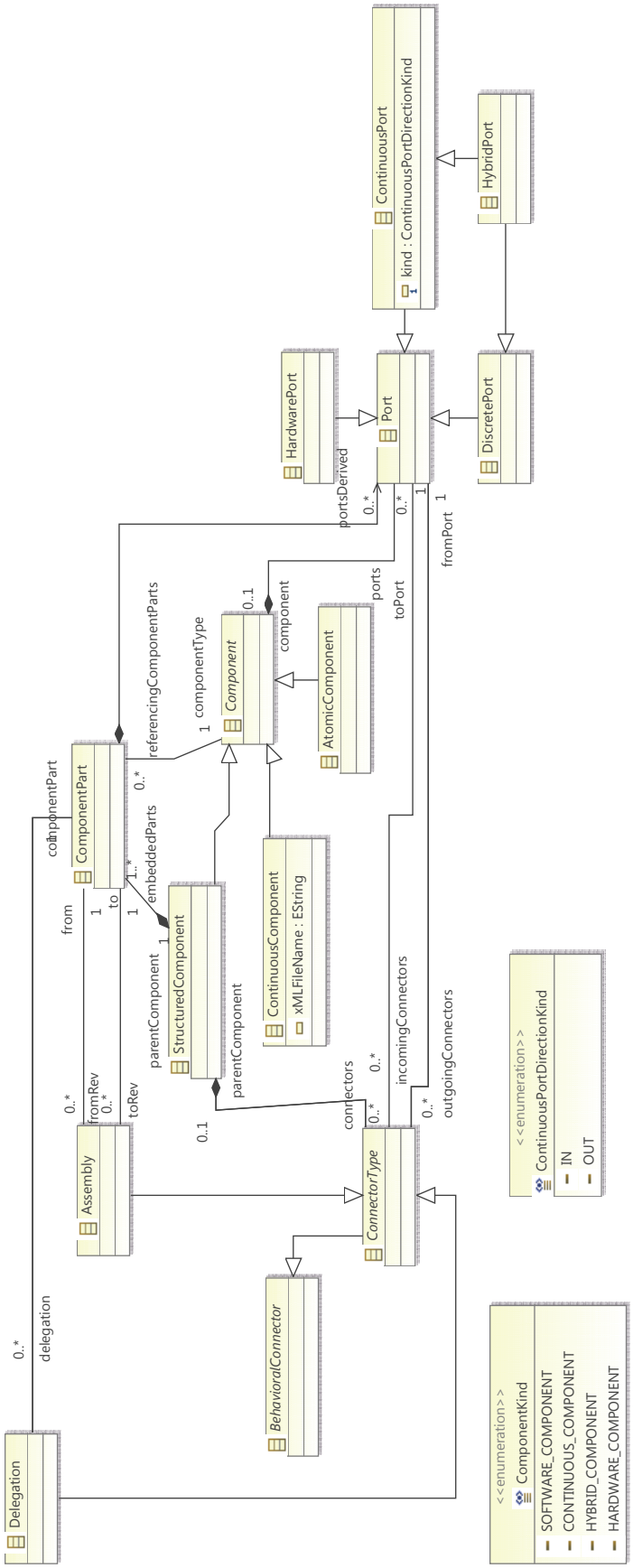


Figure A.2.: Meta-Model of the component Package

A.4.2.2. Class AtomicComponent

Overview This class represents an atomic component. Atomic components must not be further sub-divided into component parts. In contrast to structured components atomic components own a behavior in form of a realtime statechart.

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

Parent Classes

- Component see Section A.4.2.4 on Page 117,
- BehavioralElement see Section A.6.2.3 on Page 129

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

- BehavioralElement see Section A.6.2.3 on Page 129

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 References Class `Component` has the following references:

eClass : EClass [0..1]

The class of a component serves to store all attributes of the component as well as all methods that might be executed in a local context. Manipulation of the attribute values or invocation of the methods is done within the realtime statechart of the component.

mustImplementReceiverInterfaces : MessageInterface [0..∗] see Section A.8.2.1 on Page 133

This reference points to all message interfaces that this component must implement as received message interfaces via one of its ports. This reference has been used by the PG Mauritius to enable creation of sequence diagrams without having modeled the complete component structure including all interfaces (see FDays 08 paper).

mustImplementSenderInterfaces : MessageInterface [0..∗] see Section A.8.2.1 on Page 133

This reference points to all message interfaces that this component must implement as sender message interfaces via one of its ports. This reference has been used by the PG Mauritius to enable creation of sequence diagrams without having modeled the complete component structure including all interfaces (see FDays 08 paper).

ports : Port [0..∗] see Section A.4.2.15 on Page 124

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 119

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)
```

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154,
- CommentableElement see Section A.11.2.1 on Page 153,
- ConstrainableElement see Section A.6.2.5 on Page 130

A.4.2.5. Enumeration `ComponentKind`

Overview The entries of the enumeration represent different kinds of components. These are discrete software components, continous components containing controller code, hybrid components that is a discrete software component which may have continuous input signals, and hardware components representing physical entities in the system.

Enum Properties Enumeration `ComponentKind` has the following literals:

SOFTWARE_COMPONENT = 0

CONTINUOUS_COMPONENT = 1

HYBRID_COMPONENT = 2

HARDWARE_COMPONENT = 3

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 References Class ComponentPart has the following references:

cardinality : Cardinality see Section A.6.2.4 on Page 130

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 117

The component type typing this component part.

delegation : Delegation [0..∗] see Section A.4.2.11 on Page 122

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 115

The assemblies originating in port parts of this component part.

parentComponent : StructuredComponent see Section A.4.2.16 on Page 124

The structured component type containing this component part.

/portsDerived : Port [0..∗] see Section A.4.2.15 on Page 124

derivation:

```
if componentType.oclIsUndefined() then
   OrderedSet { }
else
   componentType.ports
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 115

The assemblies leading to port parts of this component part.

Class Constraints Class ComponentPart has the following constraints:

CardinalityLowerBoundSet:

```
self.cardinality.lowerBound->notEmpty()
```

TypeNotEqualToParent:

```
self.componentType <> self.parentComponent
```

CardinalityUpperBoundSet:

```
self.cardinality.upperBound->notEmpty()
```

Parent Classes

- CommentableElement see Section A.11.2.1 on Page 153

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.11.2.2 on Page 154,
- BehavioralConnector see Section A.4.2.3 on Page 117

A.4.2.8. Class `ContinuousComponent`

Overview This class represents a continuous component on the modeling level. Its purpose is to store a XML file which is created by the CAMEL tool and describes the continuous behavior of the component.

Class Properties Class `ContinuousComponent` has the following properties:

xMLFileName : EString [0..1]

The XML file describing the continuous behavior of this component.

Parent Classes

- Component see Section A.4.2.4 on Page 117

A.4.2.9. Class `ContinuousPort`

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

Class Properties Class `ContinuousPort` has the following properties:

kind : ContinuousPortDirectionKind see Section A.4.2.10 on Page 121

Decides the direction of a continous 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
self.cardinality.oclIsUndefined() or (
    if self.cardinality.lowerBound.oclIsUndefined() 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

- Port see Section A.4.2.15 on Page 124

A.4.2.10. Enumeration `ContinuousPortDirectionKind`

Overview Decides the direction of a continous port.

Enum Properties Enumeration `ContinuousPortDirectionKind` has the following literals:

IN = 0

Represent an IN-Port of a continous port.

OUT = 1

Represent an OUT-Port of a continous port.

A.4.2.11. 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.

Parent Classes

- ConnectorType see Section A.4.2.7 on Page 120

A.4.2.12. Class `DiscretePort`

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

Parent Classes

- Port see Section A.4.2.15 on Page 124,
- BehavioralElement see Section A.6.2.3 on Page 129

A.4.2.13. Class `HardwarePort`

Overview This is concrete port specification which provides the functionality of a hardware port.

Class Constraints Class `HardwarePort` has the following constraints:

LowerBoundMustBeOne:

```
-- 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
self.cardinality.oclIsUndefined() or (
    if self.cardinality.lowerBound.oclIsUndefined() then
        false
    else
        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

- Port see Section A.4.2.15 on Page 124

A.4.2.14. Class HybridPort

Overview This class represents a hybrid port which acts as a A/D or an D/A converter.

Class Constraints Class `HybridPort` 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
self.cardinality.oclIsUndefined() or (
   if self.cardinality.lowerBound.oclIsUndefined() 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

- ContinuousPort see Section A.4.2.9 on Page 120,
- DiscretePort see Section A.4.2.12 on Page 122

A.4.2.15. Class `Port`

Overview Ports represent the interaction points between a component and the components environment.

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154,
- CommentableElement see Section A.11.2.1 on Page 153,
- ConstrainableElement see Section A.6.2.5 on Page 130

A.4.2.16. Class `StructuredComponent`

Overview This class represents a structured component which is capable of including arbitraryly 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.

Parent Classes

- Component see Section A.4.2.4 on Page 117

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.

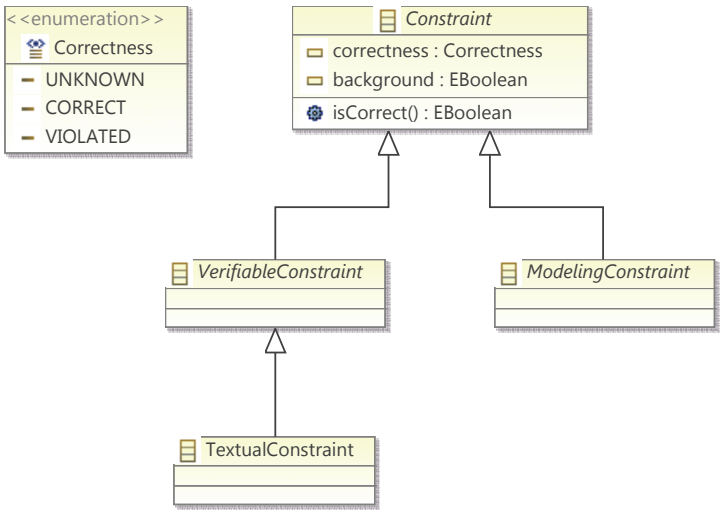


Figure A.3.: Meta-Model of the constraint Package

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 fullfill. 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 126

The correctness of this constraint encoded as a literal of the enum type "Correctness".

Class References Class `Constraint` has the following references:

constrainableElement : ConstrainableElement see Section A.6.2.5 on Page 130

The element this constraint applies to.

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154

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 125

A.5.2.4. Class `TextualConstraint`

Overview This class represents all verifable 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 127,
- ExtendableElement see Section A.11.2.2 on Page 154

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 125

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 cardinalites as natural numbers including infinity.

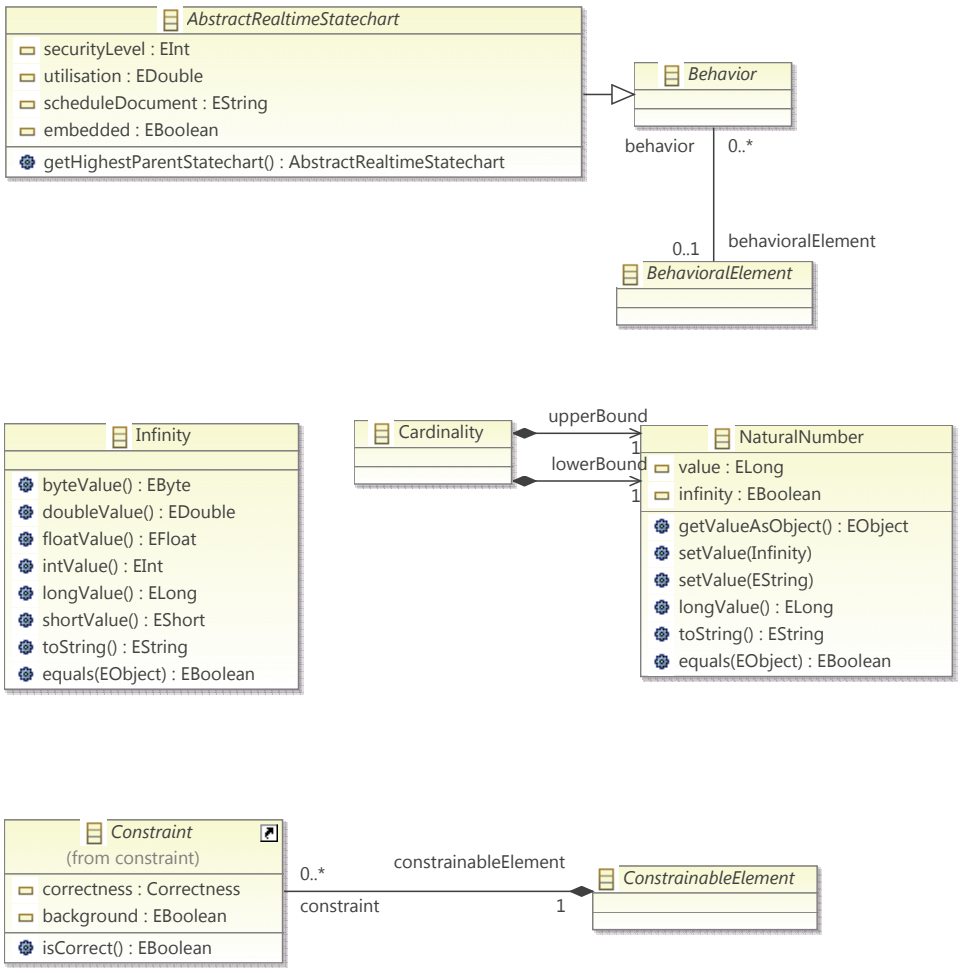


Figure A.4.: Meta-Model of the core Package

A.6.2. Detailed Contents Documentation

A.6.2.1. Class AbstractRealtimeStatechart

Overview This class represents a realtime statechart. TODO-SD: All references should moved to FujabaRealtimeStatechart, because an AbstractRealtimeStatechart abstracts from the concrete modeling.

Class Properties Class AbstractRealtimeStatechart has the following properties:

/embedded : EBoolean [0..1]

This attribute specifies whether this realtime statechart is embedded into a region or not.

scheduleDocument : EString [0..1]

needed for WCET-analysis

securityLevel : EInt [0..1]

needed for WCET-analysis

utilisation : EDouble [0..1]

needed for WCET-analysis

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154,
- CommentableElement see Section A.11.2.1 on Page 153,
- Behavior see Section A.6.2.2 on Page 129

A.6.2.2. Class Behavior

Overview Abstract super class for all elements that represent a behavior. Known subclasses: 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.6.2.7 on Page 130

The lower bound of this cardinality.

upperBound : NaturalNumber see Section A.6.2.7 on Page 130

The upper bound of this cardinality.

Class Constraints Class `Cardinality` has the following constraints:

LowerBoundMustBeLessOrEqualThanUpperBound:

```
((self.lowerBound.value <= self.upperBound.value) and self.
    lowerBound.infinity=false and self.upperBound.infinity=false)
    or (self.lowerBound.infinity=true and self.upperBound.infinity
    =true)
```

A.6.2.5. Class `ConstrainableElement`

Overview Abstract super class for all model elements that may carry a constraint.

A.6.2.6. Class `Infinity`

Overview This class extends java.lang.Number by the capability to represent infinity.

A.6.2.7. Class `NaturalNumber`

Overview This class represents either a natural number or infinity.

Class Properties Class `NaturalNumber` has the following properties:

infinity : EBoolean [0..1]

Determins whether this natural number represents infinity.

value : ELong [0..1]

The value of this natural number.

A.7. Package muml::model::instance

A.7.1. 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.

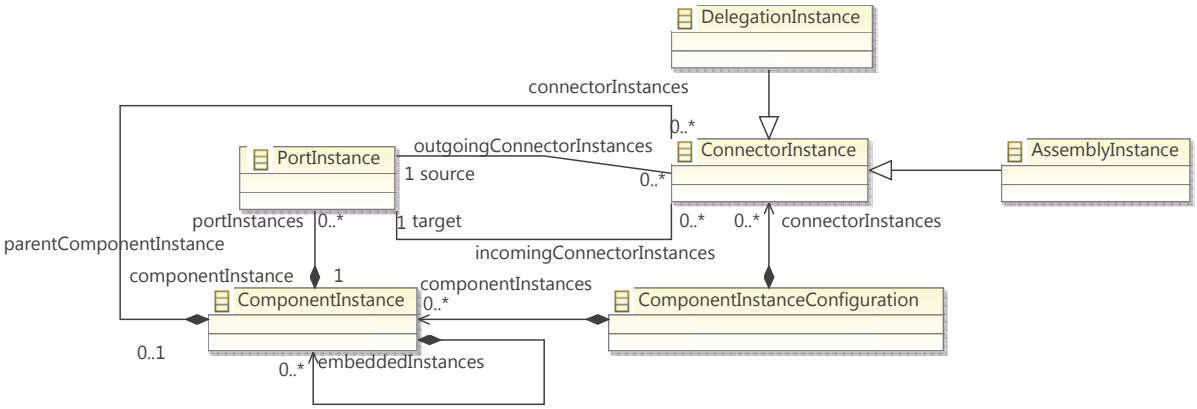


Figure A.5.: Meta-Model of the instance Package

A.7.2. Detailed Contents Documentation

A.7.2.1. Class AssemblyInstance

Overview This class represents an assembly connector at instance level.

Parent Classes

- ConnectorInstance see Section A.7.2.4 on Page 132

A.7.2.2. Class ComponentInstance

Overview This class represents a component instance. It is an instantiation of a component.

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154

A.7.2.3. Class `ComponentInstanceConfiguration`

Overview This class encapsules represents a configuration.

It contains all component instances and connector instances that belong to a concrete configuration.

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154

A.7.2.4. Class `ConnectorInstance`

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

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154,
- BehavioralConnector see Section A.4.2.3 on Page 117

A.7.2.5. Class `DelegationInstance`

Overview This class represents a delegation connector at instance level.

Parent Classes

- ConnectorInstance see Section A.7.2.4 on Page 132

A.7.2.6. Class `PortInstance`

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

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154,
- CommentableElement see Section A.11.2.1 on Page 153

A.8. Package muml::model::msgiface

A.8.1. 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.

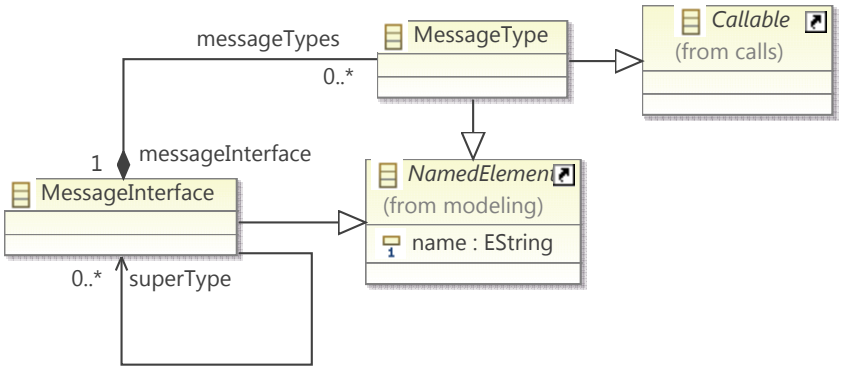


Figure A.6.: Meta-Model of the msgiface Package

A.8.2. Detailed Contents Documentation

A.8.2.1. 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:

messageTypes : MessageType [0..∗] see Section A.8.2.2 on Page 134

The message types being defined in this message interface.

superType : MessageInterface [0..∗] see Section A.8.2.1 on Page 133

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:

```
self.superType->forAll(x|x<>self)
```

NoBidirectionalGeneralization:

```
self.superType->forAll(x|x.superType->forAll(y|y<>self))
```

UniqueMessageTypeNames:

```
self.messageTypes->isUnique(name)
```

NoMessageTypeOrNotAtLeastTwoGeneralizations:

```
self.messageTypes->size()>=1 or self.superType->size()>=2
```

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154

A.8.2.2. 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.

Parent Classes

- Callable see Section A.14.2.1 on Page 165,
- NamedElement see Section A.11.2.4 on Page 154

A.9. Package muml::model::pattern

A.9.1. Package Overview

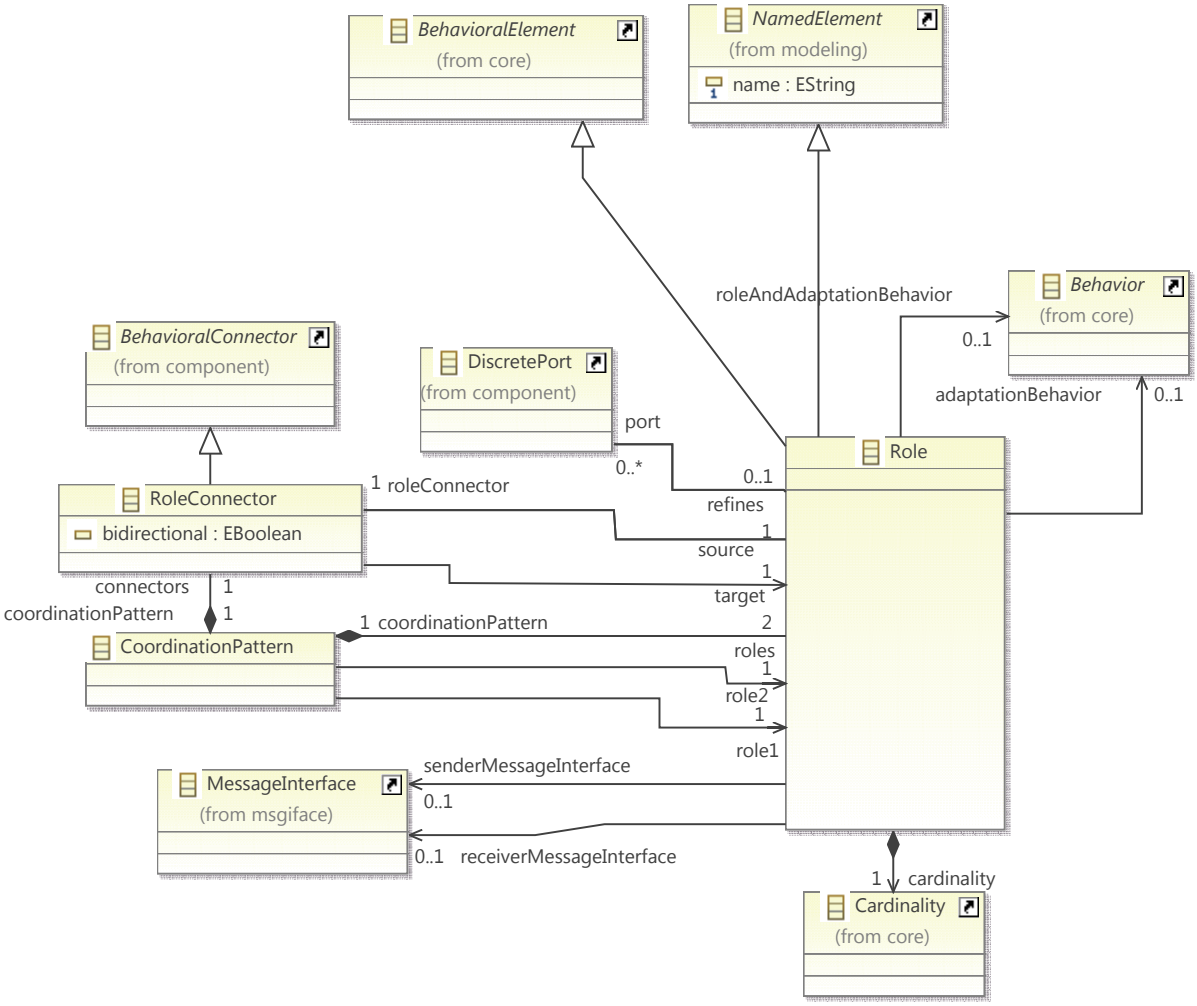


Figure A.7.: Meta-Model of the pattern Package

A.9.2. Detailed Contents Documentation

A.9.2.1. 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 whith 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 fullfill.

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154,
- ConstrainableElement see Section A.6.2.5 on Page 130

A.9.2.2. Class `Role`

Overview This class represents a role of a coordination pattern.

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154,
- ConstrainableElement see Section A.6.2.5 on Page 130,
- BehavioralElement see Section A.6.2.3 on Page 129

A.9.2.3. 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-direcational. This attribute should probably be renamed to bidirectional.

Class References Class `RoleConnector` has the following references:

coordinationPattern : CoordinationPattern see Section A.9.2.1 on Page 135

The coordination pattern this role connector is part of.

source : Role see Section A.9.2.2 on Page 136

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.9.2.2 on Page 136

The roles connected by this channel. At the moment an arbitrary number of roles are allow. This probably should be discussed.

Parent Classes

- BehavioralConnector see Section A.4.2.3 on Page 117

A.10. Package `muml::model::realtimestatechart`

A.10.1. Package Overview

A.10.2. Detailed Contents Documentation

A.10.2.1. 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.10.2.6 on Page 140

A.10.2.2. 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.11.2.4 on Page 154

A.10.2.3. Class `AsynchronousMessageEvent`

Overview An AsynchronousMessageEvent is a TransitionEvent that corresponds to receiving or sending a message. They are used to model asynchronous communication between realtime statecharts. A trigger events specifies that the corresponding message has to be received for the transition to be enabled, a raised event specifies that the corresponding message will be sent upon execution of the transition.

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154,
- TransitionEvent see Section A.10.2.30 on Page 152

A.10.2.4. Class `Clock`

Overview This class represents clocks of a realtime statechart.

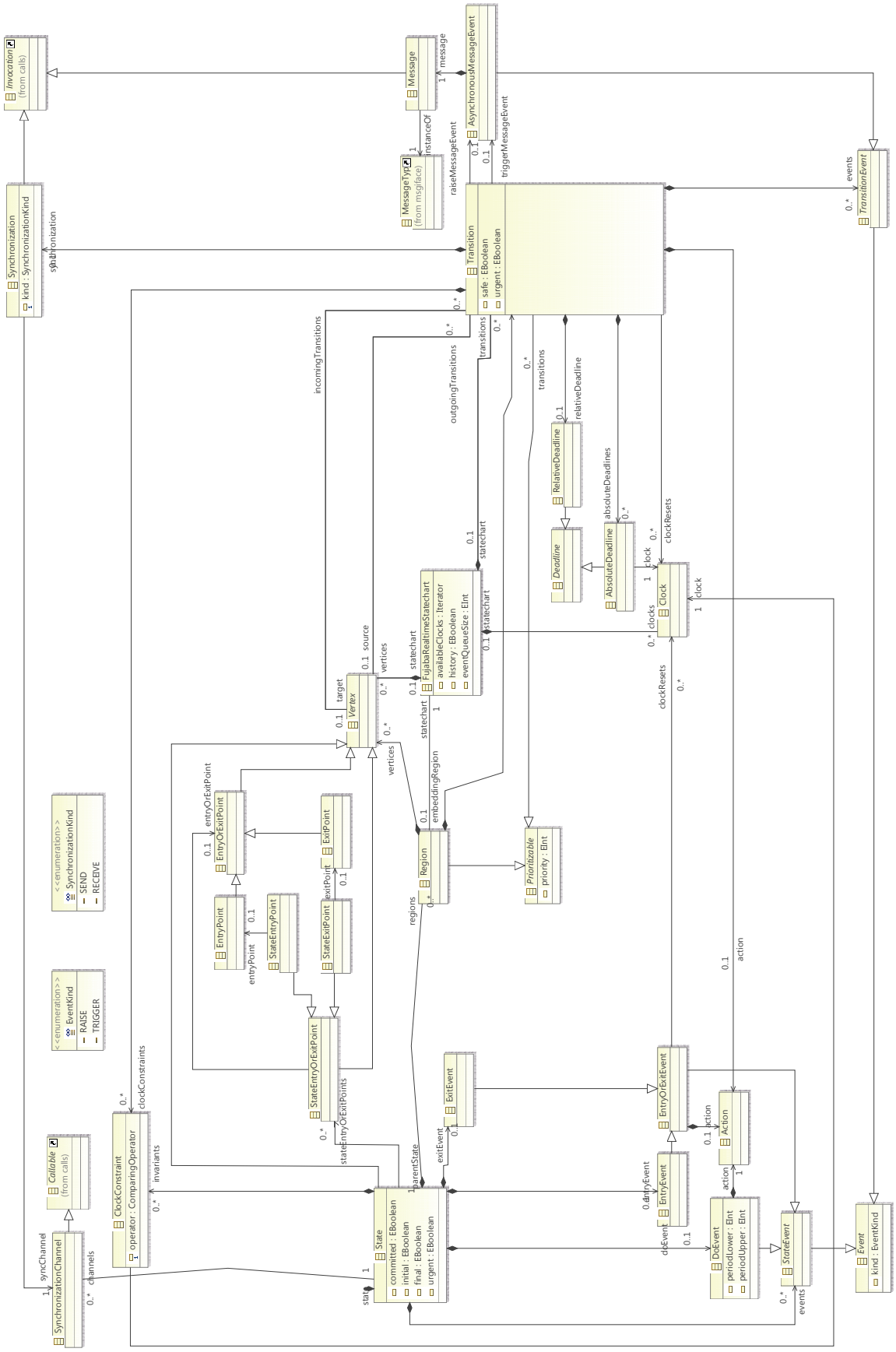


Figure A.8.: Meta-Model of the realtimestatechart Package

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154

A.10.2.5. 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.16.2.5 on Page 171

The operator that is used in this clock constraint.

Class References Class `ClockConstraint` has the following references:

bound : NaturalNumber see Section A.6.2.7 on Page 130

The bound of a deadline (upper or lower) is a natural number.

clock : Clock see Section A.10.2.4 on Page 138

The clock references in this clock constraint.

A.10.2.6. Class `Deadline`

Overview This class represents a deadline consisting of an upper and a lower bound.

A.10.2.7. 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.10.2.2 on Page 138

Each entry or exit action has one or more actions.

Parent Classes

- StateEvent see Section A.10.2.24 on Page 148

A.10.2.8. 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.10.2.9 on Page 141

A.10.2.9. 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.10.2.24 on Page 148

A.10.2.10. Class EntryOrExitPoint

Overview Entry points are defined points on a state that can be used to reach inner states.

Parent Classes

- Vertex see Section A.10.2.31 on Page 152

A.10.2.11. Class EntryPoint

Overview Entry points are defined points on a state that can be used to reach inner states.

Note We need this subclass, because GMF forbids using the same semantic element for different notational elements within the same container.

Class Constraints Class `EntryPoint` has the following constraints:

OneOutgoingTransition:

```
self.outgoingTransitions->size() = 1
```

Parent Classes

- EntryOrExitPoint see Section A.10.2.10 on Page 141

A.10.2.12. 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.10.2.13 on Page 142

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.10.2.13. 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.10.2.14. 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.10.2.9 on Page 141

A.10.2.15. Class ExitPoint

Overview

Note We need this subclass, because GMF forbids using the same semantic element for different notational elements within the same container.

Class Constraints Class `ExitPoint` has the following constraints:

AtMostOneOutgoingTransition:

```
self.outgoingTransitions->size() <= 1
```

Parent Classes

- EntryOrExitPoint see Section A.10.2.10 on Page 141

A.10.2.16. Class FujabaRealtimeStatechart

Overview This class is a concrete statechart implementation of a real-time statechart.

Class Properties Class `FujabaRealtimeStatechart` has the following properties:

/availableClocks : Iterator [0..1]

Available clocks are all clocks that were defined in this statechart or in ancestor statecharts.

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.

history : EBoolean [0..1]

If this attribute is true, it acts as a shallow history on the top hierarchy of this statechart.

Class References Class `FujabaRealtimeStatechart` has the following references:

clocks : Clock [0..∗] see Section A.10.2.4 on Page 138

The clocks of this realtime statechart.

/eClass : EClass [0..1]

The class belonging to this realtime statechart. Variables declared in the class might be manipulated by the statechart. Methods declared in the class might be executed by the statechart as side effects of the transition.

embeddingRegion : Region [0..1] see Section A.10.2.19 on Page 145

transitions : Transition [0..∗] see Section A.10.2.29 on Page 150

The transitions of the realtime statechart.

vertices : Vertex [0..∗] see Section A.10.2.31 on Page 152

The states of this realtime statechart.

Class Constraints Class `FujabaRealtimeStatechart` has the following constraints:

UniqueNameOfStates:

```
self.vertices.oclAsType(State)->isUnique(name)
```

MinOneState:

```
self.vertices.oclAsType(State)->notEmpty()
```

Parent Classes

- AbstractRealtimeStatechart see Section A.6.2.1 on Page 129

A.10.2.17. 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.14.2.2 on Page 166

A.10.2.18. 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.10.2.19. Class Region

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

Parent Classes

- Prioritizable see Section A.10.2.18 on Page 145,
- NamedElement see Section A.11.2.4 on Page 154

A.10.2.20. Class RelativeDeadline

Overview This class represents a relative deadline. It is always associated with a transition of the statechart. The deadlin is relative to the point in time when the execution of the transition starts.

Parent Classes

- Deadline see Section A.10.2.6 on Page 140

A.10.2.21. Class State

Overview This class represents a complex state of a realtime statechart. Complex states may again contain realtime statecharts hence enabling the creation of hierarchical statecharts. Further more complex states have do, entry and exit actions. Also complex states define which synchronization channels are allowed to be used by embedded statecharts.

Class Properties Class State has the following properties:

committed : EBoolean [0..1]

If a state is active and commited, no time is allowed to pass and no other transition is allowed to fire until the state is leaved. The meaning of being committed is the same as it is with committed states in Uppaal.

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.

urgent : EBoolean [0..1]

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

Class References Class `State` has the following references:

channels : SynchronizationChannel [0..∗] see Section A.10.2.27 on Page 149

The synchronization channels provided by this state.

/doEvent : DoEvent [0..1] see Section A.10.2.7 on Page 140

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

/entryEvent : EntryEvent [0..1] see Section A.10.2.8 on Page 141

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

events : StateEvent [0..∗] see Section A.10.2.24 on Page 148

/exitEvent : ExitEvent [0..1] see Section A.10.2.14 on Page 142

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

invariants : ClockConstraint [0..∗] see Section A.10.2.5 on Page 140

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

regions : Region [0..∗] see Section A.10.2.19 on Page 145

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.

stateEntryOrExitPoints : StateEntryOrExitPoint [0..∗] see Section A.10.2.22 on Page 147

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

Class Constraints Class `State` has the following constraints:

OneInvarianPerClock:

```
self.invariants->isUnique(clock)
```

OneInitialState:

```
self.statechart.vertices.oclAsType(State)->one(s|  s.initial)
```

NoOutgoingTransitionOfFinalState:

```
self.final implies self.outgoingTransitions->isEmpty()
```

NoRegionsOfFinalState:

```
self.final implies self.regions->isEmpty()
```

UniquePrioritiesOfOutgoingTransitions:

```
self.outgoingTransitions->isUnique(priority)
```

UniquePrioritiesOfRegions:

```
self.regions->isUnique(priority)
```

UniqueChannelNames:

```
self.channels->isUnique(name)
```

UniqueRegionNames:

```
self.regions->isUnique(name)
```

Parent Classes

- Vertex see Section A.10.2.31 on Page 152

A.10.2.22. Class StateEntryOrExitPoint

Overview The entry and exit points that are assigned to a state. They are derived from the entry and exit points of the statechart, the state embeds.

Parent Classes

- Vertex see Section A.10.2.31 on Page 152

A.10.2.23. Class StateEntryPoint

Overview The entry point that is assigned to a state. It is derived from the entry point of the statechart, the state embeds.

Note We need this subclass, because GMF forbids using the same semantic element for different notational elements within the same container.

Class References Class `StateEntryPoint` has the following references:

/entryPoint : EntryPoint [0..1] see Section A.10.2.11 on Page 141

derivation:

```
entryOrExitPoint.oclAsType(realtimestatechart::EntryPoint)
```

This is the inherited entryOrExitPoint casted as EntryPoint.

Class Constraints Class `StateEntryPoint` has the following constraints:

AtLeastOneIncomingTransition:

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

Parent Classes

- StateEntryOrExitPoint see Section A.10.2.22 on Page 147

A.10.2.24. 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.10.2.12 on Page 142

A.10.2.25. Class `StateExitPoint`

Overview The exit point that is assigned to a state. It is derived from the exit point of the statechart, the state embeds.

Note We need this subclass, because GMF forbids using the same semantic element for different notational elements within the same container.

Class References Class `StateExitPoint` has the following references:

/exitPoint : ExitPoint [0..1] see Section A.10.2.15 on Page 143

derivation:

```
entryOrExitPoint.oclAsType(realtimestatechart::ExitPoint)
```

This is the inherited entryOrExitPoint casted as ExitPoint.

Class Constraints Class `StateExitPoint` has the following constraints:

OneOutgoingTransition:

```
self.outgoingTransitions->size() = 1
```

Parent Classes

- StateEntryOrExitPoint see Section A.10.2.22 on Page 147

A.10.2.26. 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.10.2.28 on Page 150

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

Class References Class `Synchronization` has the following references:

/syncChannel : SynchronizationChannel see Section A.10.2.27 on Page 149

the channel that is used by the synchronization

Parent Classes

- Invocation see Section A.14.2.2 on Page 166

A.10.2.27. 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.14.2.1 on Page 165,
- NamedElement see Section A.11.2.4 on Page 154

A.10.2.28. 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.10.2.29. 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:

safe : EBoolean [0..1]

Needed for failure propagation.

urgent : EBoolean [0..1]

If a transition is urgent it has the fire immediately after its execution (no time is allowed to pass).

Class References Class `Transition` has the following references:

absoluteDeadlines : AbsoluteDeadline [0..∗] see Section A.10.2.1 on Page 138

a transition can has one or more absolute deadlines

action : Action [0..1] see Section A.10.2.2 on Page 138

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.10.2.5 on Page 140

A clock constraint restricts when the transition can be activeted in dependency of the values of the clock.

clockResets : Clock [0..∗] see Section A.10.2.4 on Page 138

The clock resets of this transition.

events : TransitionEvent [0..∗] see Section A.10.2.30 on Page 152

guard : Expression [0..1] see Section A.16.2.7 on Page 171

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.10.2.3 on Page 138

derivation:

```
self.events->select(e | e.oclIsKindOf(
    AsynchronousMessageEvent) and e.kind=EventKind::RAISE).
    oclAsType(AsynchronousMessageEvent)->first()
```

The event which is raised upon activiation of this transition.

relativeDeadline : RelativeDeadline [0..1] see Section A.10.2.20 on Page 145

a transition can have one relative deadline

source : Vertex [0..1] see Section A.10.2.31 on Page 152

The state which is the source of this transition.

statechart : FujabaRealtimeStatechart [0..1] see Section A.10.2.16 on Page 143

The realtime statechart this transition belongs to.

synchronization : Synchronization [0..1] see Section A.10.2.26 on Page 149

The synchronisation which is sent upon activation of this transition.

target : Vertex [0..1] see Section A.10.2.31 on Page 152

The state which is the target of this transition.

/triggerMessageEvent : AsynchronousMessageEvent [0..1] see Section A.10.2.3 on Page 138

derivation:

```
self.events->select(e | e.oclIsKindOf(
    AsynchronousMessageEvent) and e.kind=EventKind::TRIGGER).
    oclAsType(AsynchronousMessageEvent)->first()
```

The trigger event of this transition.

Class Constraints Class `Transition` has the following constraints:

SetTargetAndSource:

```
self.target->notEmpty() and self.source->notEmpty()
```

NoCrossingOfRegionBorders:

```
self.source.statechart.embeddingRegion=self.target.statechart.
    embeddingRegion or
self.source.oclAsType(StateEntryPoint).statechart.embeddingRegion
    =
self.target.statechart.embeddingRegion.parentState.statechart.
    embeddingRegion or
```

```
self.source.statechart.embeddingRegion.parentState.statechart.
    embeddingRegion=
self.target.oclAsType(StateExitPoint).statechart.embeddingRegion
```

Parent Classes

- Prioritizable see Section A.10.2.18 on Page 145,
- ExtendableElement see Section A.11.2.2 on Page 154

A.10.2.30. 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.10.2.12 on Page 142

A.10.2.31. 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.11.2.4 on Page 154

A.11. Package modeling

A.11.1. 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 reoccuring structural features like, e.g., names of elements. The classes in this package are intended to be sub-classed by any meta-model element.

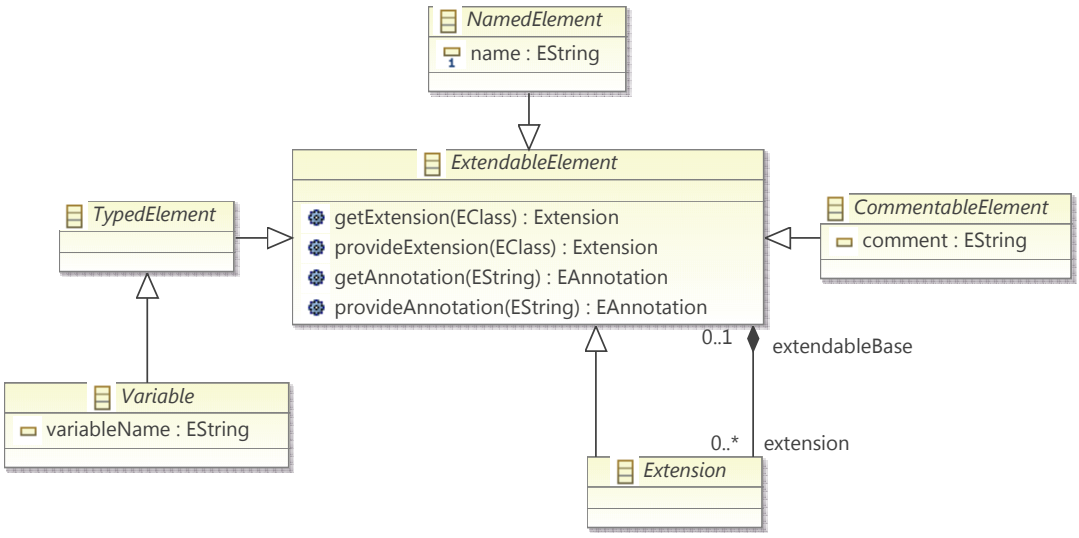


Figure A.9.: Meta-Model of the modeling Package

A.11.2. Detailed Contents Documentation

A.11.2.1. 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.11.2.2 on Page 154

A.11.2.2. 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.11.2.3. Class `Extension`

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

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154

A.11.2.4. 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.11.2.2 on Page 154

A.11.2.5. 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.11.2.2 on Page 154

A.11.2.6. 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:

/variableName : EString [0..1]

Parent Classes

- TypedElement see Section A.11.2.5 on Page 154

A.12. Package `modeling::activities`

A.12.1. Package Overview

A.12.2. Detailed Contents Documentation

A.12.2.1. Class `Activity`

Overview The diagram that describes the control flow of an operation. It is used to structure a number story patterns into a stroy diagram. Story patterns are contained in activity nodes which are connected by activity edges. In addition, there are special nodes like start, stop, and juction nodes.

Parent Classes

- CommentableElement see Section A.11.2.1 on Page 153,
- Callable see Section A.14.2.1 on Page 165

A.12.2.2. 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 Responsibilty).

Parent Classes

- ActivityNode see Section A.12.2.4 on Page 158,
- Invocation see Section A.14.2.2 on Page 166

A.12.2.3. Class `ActivityEdge`

Overview The ActivityEdge represents the control flow in an activity. It is a dericted 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.12.2.5 on Page 158

The guard defines the kind of the activity edge. The possible kinds of guards are specified by the EdgeGuard enum.

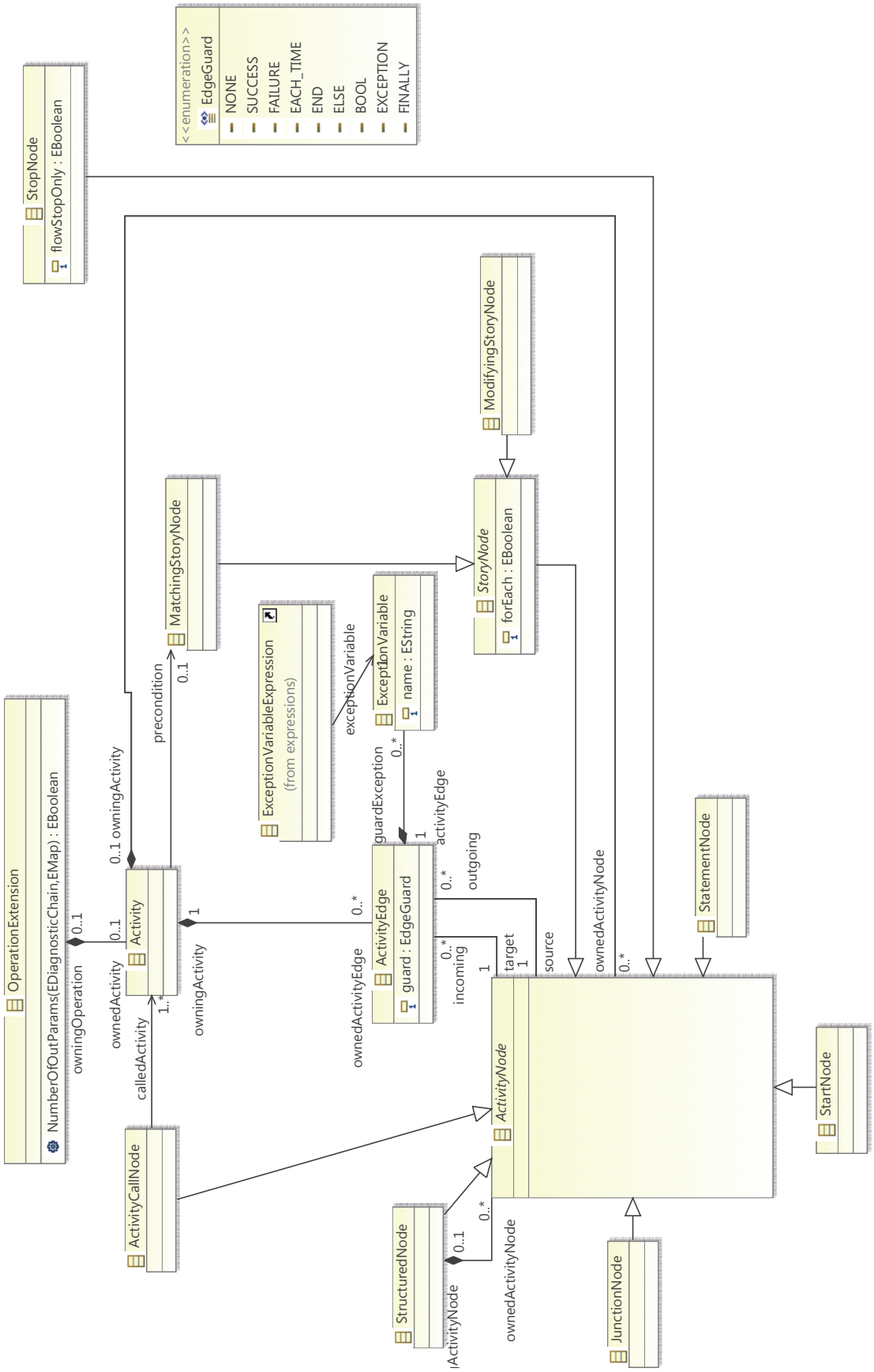


Figure A.10.: Meta-Model of the activities Package

Class References Class `ActivityEdge` has the following references:

guardException : ExceptionVariable [0..∗] see Section A.12.2.6 on Page 160

Declares variables representing the Exceptions that lead to firing this transition.

guardExpression : Expression [0..1] see Section A.16.2.7 on Page 171

Points to an expression in case the transition guard is BOOL. The expression has to evaulate to a boolean value.

owningActivity : Activity see Section A.12.2.1 on Page 156

Points to the activity this ActivityEdge is contained in.

source : ActivityNode see Section A.12.2.4 on Page 158

The source node of this ActivityEdge.

target : ActivityNode see Section A.12.2.4 on Page 158

The target node of this ActivityEdge.

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154

A.12.2.4. 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.11.2.4 on Page 154,
- CommentableElement see Section A.11.2.1 on Page 153

A.12.2.5. 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 souce 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 occured 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.12.2.6. 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.12.2.3 on Page 156

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.11.2.6 on Page 155

A.12.2.7. 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.12.2.4 on Page 158

A.12.2.8. 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.12.2.14 on Page 162

A.12.2.9. Class ModifyingStoryNode

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

Parent Classes

- StoryNode see Section A.12.2.14 on Page 162

A.12.2.10. 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.11.2.3 on Page 154,
- Callable see Section A.14.2.1 on Page 165

A.12.2.11. 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.12.2.4 on Page 158

A.12.2.12. Class StatementNode

Overview A statement node is a node that just contains an expression defining its behavior. In combination with a textual expression, arbitrary souce code might be added by using StatementNodes.

Parent Classes

- ActivityNode see Section A.12.2.4 on Page 158

A.12.2.13. 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.16.2.7 on Page 171

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.16.2.7 on Page 171

Defines the return values of the activity. These return values will be assigned to the out-parameters.

Parent Classes

- ActivityNode see Section A.12.2.4 on Page 158

A.12.2.14. 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.17.2.18 on Page 182

Parent Classes

- ActivityNode see Section A.12.2.4 on Page 158

A.12.2.15. Class `StructuredNode`

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

Parent Classes

- ActivityNode see Section A.12.2.4 on Page 158

A.13. Package `modeling::activities::expressions`

A.13.1. Package Overview

A.13.2. Detailed Contents Documentation

A.13.2.1. 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.16.2.7 on Page 171

A.14. Package modeling::calls

A.14.1. Package Overview

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

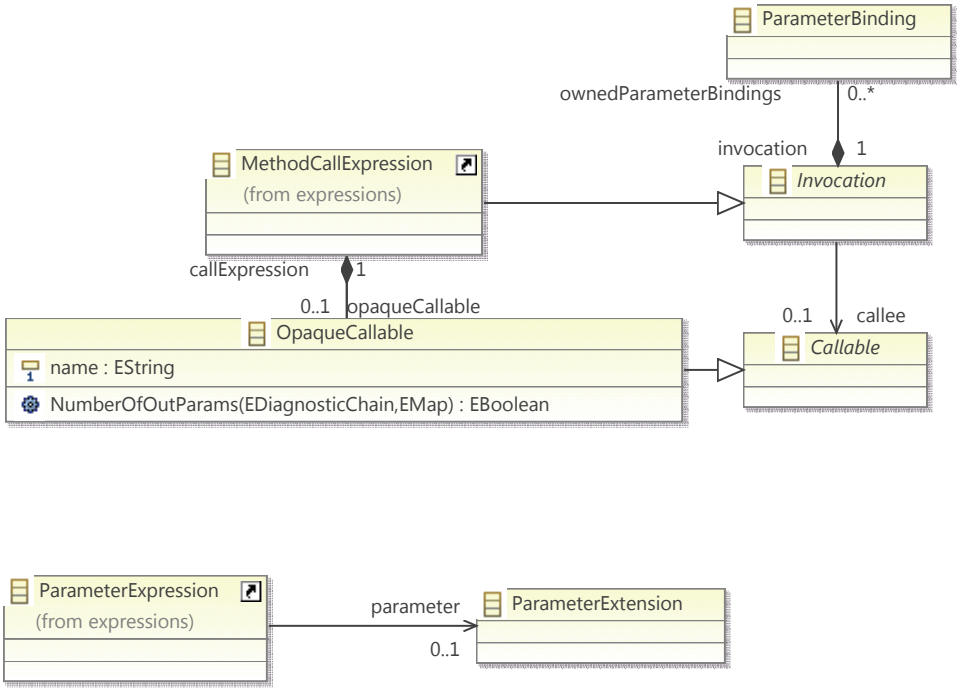


Figure A.11.: Meta-Model of the calls Package

A.14.2. Detailed Contents Documentation

A.14.2.1. 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.11.2.1 on Page 153

A.14.2.2. 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.11.2.1 on Page 153

A.14.2.3. 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.15.2.1 on Page 168

Parent Classes

- Callable see Section A.14.2.1 on Page 165

A.14.2.4. 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.11.2.1 on Page 153

A.14.2.5. Class `ParameterExtension`

Overview Represents an EParameter and adds functionality to it, especially beiing subtype of Variable.

Parent Classes

- Variable see Section A.11.2.6 on Page 155,
- Extension see Section A.11.2.3 on Page 154

A.15. Package `modeling::calls::expressions`

A.15.1. Package Overview

A.15.2. Detailed Contents Documentation

A.15.2.1. Class `MethodCallExpression`

Overview A MethodCallEpression 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.16.2.7 on Page 171,
- Invocation see Section A.14.2.2 on Page 166

A.15.2.2. 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.16.2.7 on Page 171

A.16. Package modeling::expressions

A.16.1. Package Overview

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

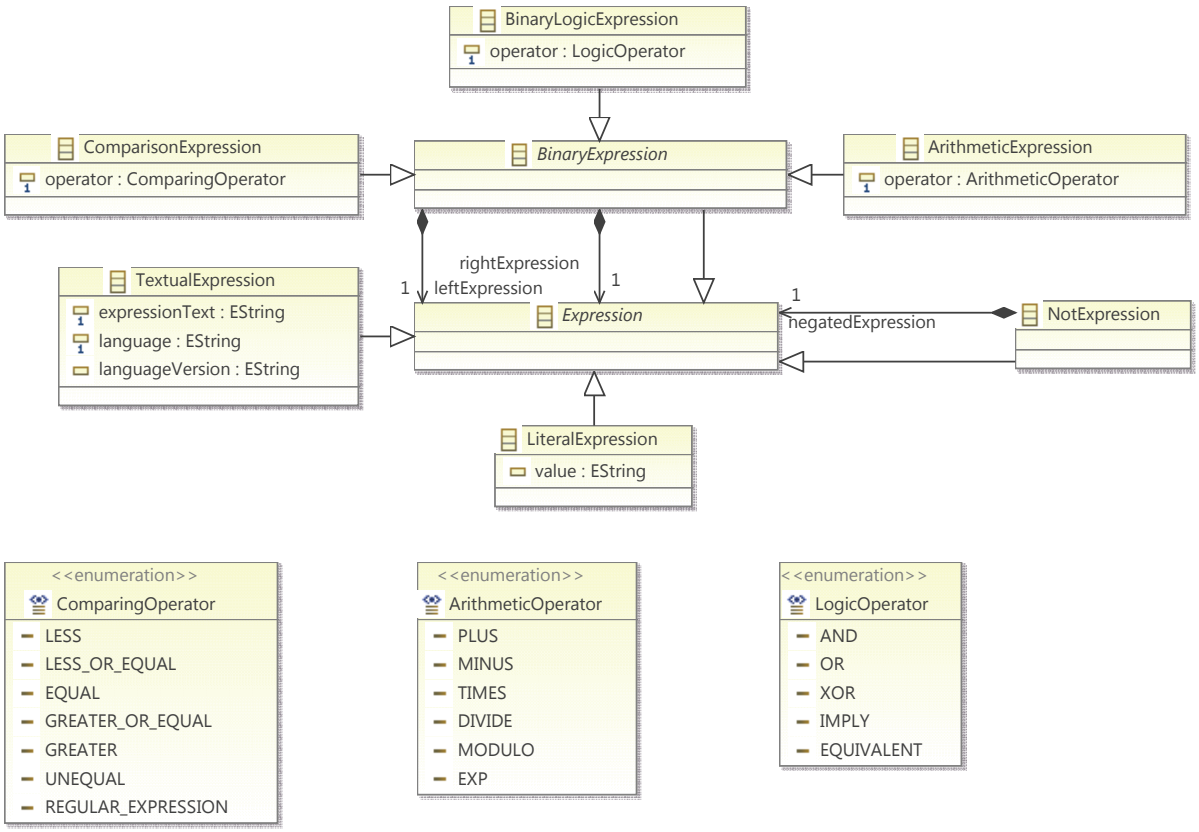


Figure A.12.: Meta-Model of the expressions Package

A.16.2. Detailed Contents Documentation

A.16.2.1. 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.16.2.2 on Page 170

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

Parent Classes

- BinaryExpression see Section A.16.2.3 on Page 170

A.16.2.2. 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 aˆb.

A.16.2.3. Class `BinaryExpression`

Overview Represents any binary expression like v < 5 or x + 7.

Parent Classes

- Expression see Section A.16.2.7 on Page 171

A.16.2.4. 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.16.2.9 on Page 172

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

Parent Classes

- BinaryExpression see Section A.16.2.3 on Page 170

A.16.2.5. 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.16.2.6. 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.16.2.5 on Page 171

Specifies the expression's comparing operator, e.g. <, >=, !=.

Parent Classes

- BinaryExpression see Section A.16.2.3 on Page 170

A.16.2.7. 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.11.2.5 on Page 154,
- CommentableElement see Section A.11.2.1 on Page 153

A.16.2.8. 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.16.2.7 on Page 171

A.16.2.9. 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.16.2.10. Class `NotExpression`

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

Parent Classes

- Expression see Section A.16.2.7 on Page 171

A.16.2.11. 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.16.2.7 on Page 171

A.17. Package `modeling::patterns`

A.17.1. Package Overview

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

A.17.2. Detailed Contents Documentation

A.17.2.1. 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.17.2.4 on Page 176

The binding operator defines whether this link will be matched, created or destroyed by the story pattern. The default value ist "check_only", i.e., the link will be matched.

bindingSemantics : BindingSemantics see Section A.17.2.5 on Page 177

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.17.2.6 on Page 177

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.17.2.10 on Page 179

pattern : StoryPattern see Section A.17.2.18 on Page 182

secondLinkConstraint : LinkConstraint [0..∗] see Section A.17.2.10 on Page 179

source : ObjectVariable see Section A.17.2.15 on Page 181

target : AbstractVariable see Section A.17.2.2 on Page 176

Parent Classes

- NamedElement see Section A.11.2.4 on Page 154

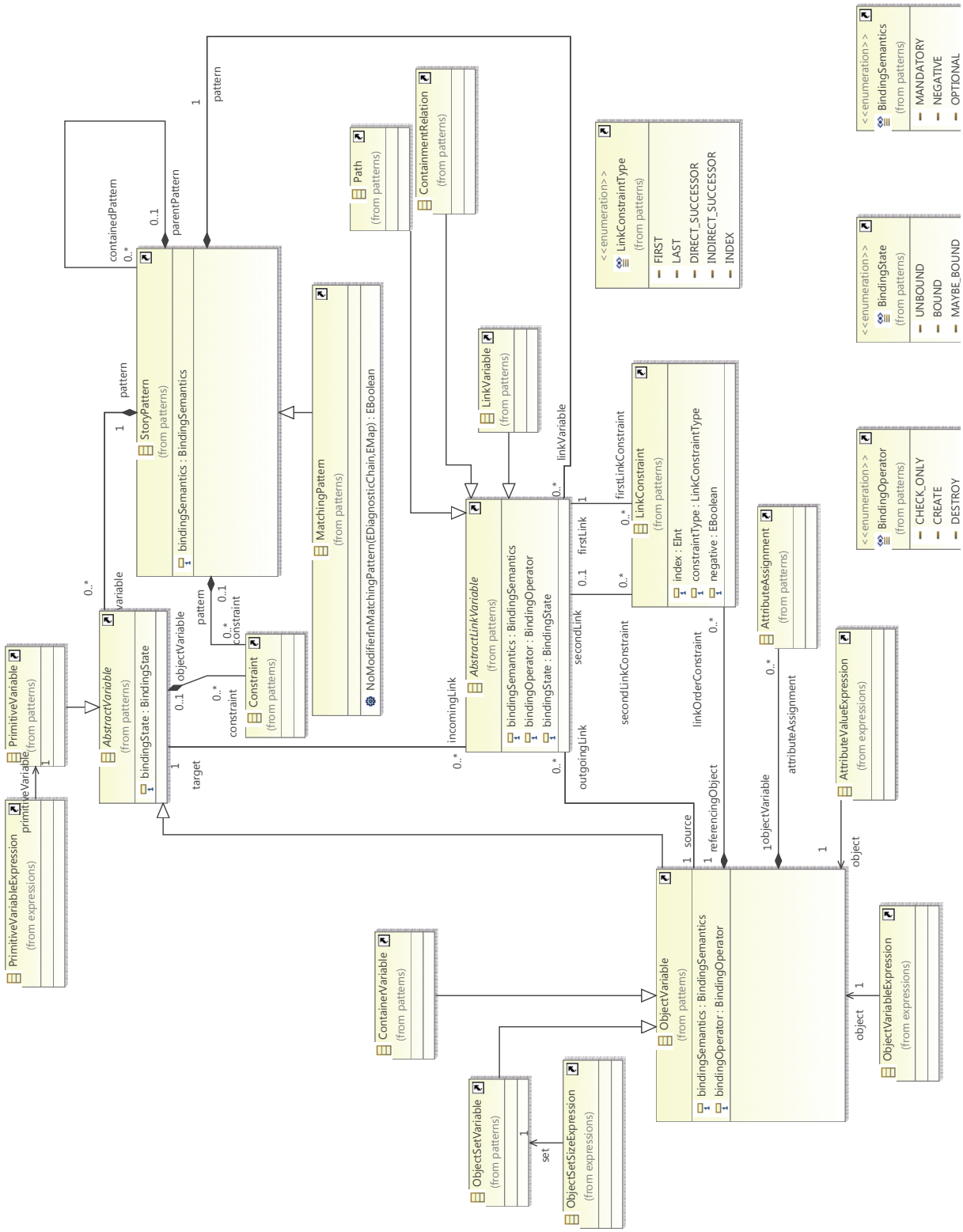


Figure A.13.: Meta-Model of the patterns Package

A.17.2.2. Class `AbstractVariable`

Overview Abstract super class for object and primitive variables.

Class Properties Class `AbstractVariable` has the following properties:

bindingState : BindingState see Section A.17.2.6 on Page 177

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.16.2.7 on Page 171

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.17.2.7 on Page 178

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.17.2.1 on Page 174

pattern : StoryPattern see Section A.17.2.18 on Page 182

Parent Classes

- Variable see Section A.11.2.6 on Page 155,
- NamedElement see Section A.11.2.4 on Page 154

A.17.2.3. 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.17.2.4. 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 be the story pattern.

A.17.2.5. 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.17.2.6. 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.17.2.7. 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 sucessful (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.17.2.8. 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.17.2.15 on Page 181

A.17.2.9. 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.

Parent Classes

- AbstractLinkVariable see Section A.17.2.1 on Page 174

A.17.2.10. 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.17.2.11 on Page 179

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.17.2.1 on Page 174

referencingObject : ObjectVariable see Section A.17.2.15 on Page 181

secondLink : AbstractLinkVariable [0..1] see Section A.17.2.1 on Page 174

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154

A.17.2.11. Enumeration LinkConstraintType

Overview The LinkConstraintType represents the different uses of LinkConstraints. Objects 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, INDIRECT_SUCCESSOR).

Enum Properties Enumeration `LinkConstraintType` has the following literals:

FIRST = 0

LAST = 1

DIRECT_SUCCESSOR = 2

INDIRECT_SUCCESSOR = 3

INDEX = 4

A.17.2.12. 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.17.2.1 on Page 174

A.17.2.13. 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.17.2.18 on Page 182

A.17.2.14. 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.17.2.15 on Page 181

A.17.2.15. 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.17.2.4 on Page 176

The binding operator defines whether this object will be matched, created or destroyed by the story pattern.

bindingSemantics : BindingSemantics see Section A.17.2.5 on Page 177

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.17.2.3 on Page 176

classifier : EClass

The type of this ObjectVariable, given as an EClass.

linkOrderConstraint : LinkConstraint [0..∗] see Section A.17.2.10 on Page 179

outgoingLink : AbstractLinkVariable [0..∗] see Section A.17.2.1 on Page 174

Parent Classes

- AbstractVariable see Section A.17.2.2 on Page 176

A.17.2.16. 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.17.2.1 on Page 174

A.17.2.17. 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.17.2.2 on Page 176

A.17.2.18. 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.17.2.5 on Page 177

Class References Class `StoryPattern` has the following references:

constraint : Constraint [0..∗] see Section A.17.2.7 on Page 178

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.17.2.18 on Page 182

linkVariable : AbstractLinkVariable [0..∗] see Section A.17.2.1 on Page 174

parentPattern : StoryPattern [0..1] see Section A.17.2.18 on Page 182

templateSignature : TemplateSignature [0..1] see Section A.19.2.3 on Page 184

variable : AbstractVariable [0..∗] see Section A.17.2.2 on Page 176

Parent Classes

- CommentableElement see Section A.11.2.1 on Page 153

A.18. Package modeling::patterns::expressions

A.18.1. Package Overview

A.18.2. Detailed Contents Documentation

A.18.2.1. 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.16.2.7 on Page 171

A.18.2.2. 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 a apply the pattern when at least two objects can be matched for the set.

Parent Classes

- Expression see Section A.16.2.7 on Page 171

A.18.2.3. 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.16.2.7 on Page 171

A.18.2.4. Class PrimitiveVariableExpression

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

Parent Classes

- Expression see Section A.16.2.7 on Page 171

A.19. Package `modeling::templates`

A.19.1. Package Overview

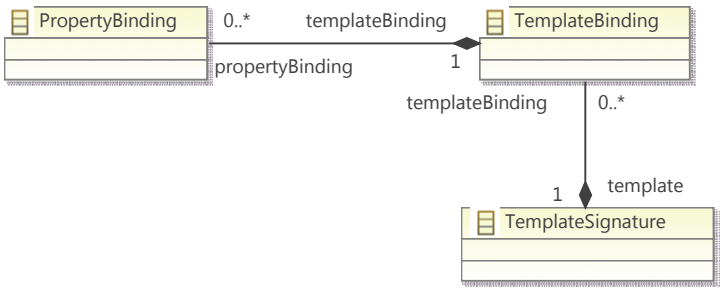


Figure A.14.: Meta-Model of the templates Package

A.19.2. Detailed Contents Documentation

A.19.2.1. Class `PropertyBinding`

Overview

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154

A.19.2.2. Class `TemplateBinding`

Overview

Parent Classes

- ExtendableElement see Section A.11.2.2 on Page 154

A.19.2.3. Class `TemplateSignature`

Overview