Reconfiguration of Mechatronic UML Component Architectures

Bachelor’s Thesis
Submitted to the Software Engineering Research Group
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science

by
CHRISTIAN DREISING
Wibbeltstraße 17
59590 Geseke

Thesis Supervisor:
Jun.-Prof. Dr.-Ing. Steffen Becker
and
Prof. Dr. Johannes Blömer

Paderborn, November 4, 2011
Declaration
(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

______________________________  __________________________
City, Date                          Signature
Contents

1 Introduction ........................................ 1
  1.1 Motivation ....................................... 2
  1.2 Case Study ...................................... 3
  1.3 Problem Description and Objectives ............... 4
  1.4 Overview ....................................... 6

2 Related Work ....................................... 7

3 Fundamentals ....................................... 9
  3.1 MechatronicUML ................................. 9
    3.1.1 Real-Time Statechart ........................ 9
    3.1.2 Real-Time Coordination Pattern ............. 11
    3.1.3 Component Model ............................ 12
    3.1.4 Component Instance Configuration ........... 15
  3.2 Graphs and Graph Transformations ................. 16
    3.2.1 Graphs .................................... 16
    3.2.2 Graph Transformations ....................... 17
  3.3 Reconfiguration of MechatronicUML models .......... 18
    3.3.1 Story Patterns and Story Diagrams .......... 19
    3.3.2 Component Story Patterns and Component Story Diagrams 20

4 MechatronicUML Reconfiguration Concept ............ 23
  4.1 Reconfiguration Message Event Flow ............... 23
  4.2 Reconfiguration by Component Story Diagrams ....... 26
  4.3 Reconfiguration of Structured Components .......... 27
    4.3.1 Reconfiguration Execution Controller (REC) .... 28
    4.3.2 Reconfiguration Process Manager (RPM) ......... 29
    4.3.3 Configuration State Manager (CSM) ............ 32
  4.4 Reconfiguration of Atomic Components .............. 34
  4.5 Pattern Instantiation on System Level .............. 39
    4.5.1 Extension of Reconfiguration Process Manager .... 39
    4.5.2 Broadcast Link Pattern ....................... 40
    4.5.3 Negotiation Pattern ........................ 45

5 Reconfiguration Example ........................... 49
  5.1 BeBot MechatronicUML model ..................... 49
  5.2 Example Scenario Details ....................... 50
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>BeBot mini robot (fully equipped) [HWR09].</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Adding a new member to an existing convoy of BeBots.</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Three component instances of BeBots driving in a convoy.</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Concrete Syntax of an orthogonal state [BDG+11].</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Concrete Syntax example of a transition [BDG+11].</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Real-Time Coordination Pattern example.</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>Atomic Component type with one single-port and one multi-port.</td>
<td>13</td>
</tr>
<tr>
<td>3.5</td>
<td>Statechart of an Atomic Component with a single-port, a multi-port and a synchronization.</td>
<td>14</td>
</tr>
<tr>
<td>3.6</td>
<td>Structured Component with one part and one multi-part.</td>
<td>15</td>
</tr>
<tr>
<td>3.7</td>
<td>Component Instance Configuration with two connected component instances.</td>
<td>15</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of a directed graph.</td>
<td>16</td>
</tr>
<tr>
<td>3.9</td>
<td>Structured Component instance with an embedded Atomic Component instance.</td>
<td>17</td>
</tr>
<tr>
<td>3.10</td>
<td>Example of an object diagram of a Structured Component’s instance configuration with one embedded Atomic Component instance.</td>
<td>17</td>
</tr>
<tr>
<td>3.11</td>
<td>Structured Component instance with two embedded Atomic Component instances.</td>
<td>18</td>
</tr>
<tr>
<td>3.12</td>
<td>Example of a graph transformation rule to reconfigure a Structured Component instance.</td>
<td>19</td>
</tr>
<tr>
<td>3.13</td>
<td>Simple Story Pattern to reconfigure a Structured Component instance.</td>
<td>20</td>
</tr>
<tr>
<td>3.14</td>
<td>Simple Component Story Diagram to reconfigure a Structured Component instance.</td>
<td>21</td>
</tr>
<tr>
<td>3.15</td>
<td>Example of executing a Component Story Diagram on a component instance configuration.</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Fully reconfigurable Structured Component with two embedded Atomic Components</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Sequence Diagram of reconfiguration message event flow within a Structured Component instance with two embedded Atomic Component instances.</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Structured Component with Reconfiguration Execution Controller included.</td>
<td>29</td>
</tr>
</tbody>
</table>
List of Figures

4.4 Structured Component with Reconfiguration Process Manager and Reconfiguration Execution Controller .......................... 30
4.5 Real-Time Coordination Pattern “Reconfiguration Execution” ... 31
4.6 Real-Time Statechart of role “Reconfiguration Commander” .... 32
4.7 Real-Time Statechart of role “Reconfiguration Executer” ....... 32
4.8 Structured Component with Reconfiguration Process Manager, Reconfiguration Execution Controller and Configuration State Manager. 33
4.9 Real-time Coordination Pattern “Configuration State” .......... 34
4.10 Atomic Component containing a reconfiguration port. .......... 34
4.11 Real-Time Statechart of an Atomic Component with simple reconfiguration behavior. .................................. 35
4.12 Realtime Coordination Pattern “Reconfiguration Propagation” ... 36
4.13 Role “Reconfiguration Master” of Pattern “Reconfiguration Propagation” .................................................. 37
4.14 Role “Reconfiguration Slave” of Pattern “Reconfiguration Propagation” .................................................. 37
4.15 Structured Component with RPM, extended by link and negotiation ports. ............................................. 38
4.16 Sequence Diagram for establishment of contacts between two autonomous systems. ........................................... 40
4.17 Broadcast Link Pattern for establishment of contacts between two autonomous systems. ........................................... 42
4.18 Real-Time Statechart of a RPM, whose link port implements the “Broadcast Link Pattern”. ................................. 43
4.19 Component Story Diagram “createNegoPortOnRPM”. ....... 45
4.20 Component Story Diagram “createNegoPortOnHost” ......... 45
4.21 Real-Time Coordination Pattern “Negotiation” .................. 46
4.22 Structured Component including new elements of the reconfiguration concept. ............................................ 46

5.1 Structured Component type of a BeBot. ............................ 50
5.2 BeBot instances reconfiguration scenario step 1. .................. 51
5.3 Role “StateRequester” of Real-Time Coordination Pattern “Configuration State” .............................................. 52
5.4 Role “StateManager” of Real-Time Coordination Pattern “Configuration State” .............................................. 53
5.5 Role “Negotiator” of Real-Time Coordination Pattern “Negotiation”. ...................................................... 54
5.6 BeBot instances reconfiguration scenario step 1. ................. 56
5.7 Component Story Diagram “createLeaderPort”. .................. 56
5.8 Component Story Diagram “createLeaderPortOnHost” ....... 57
5.9 BeBot instances reconfiguration scenario step 2. ................. 57
5.10 Component Story Diagram “createReconfPortOnRPM”. ...... 58
5.11 Component Story Diagram “createMemberControl” .......... 58
5.12 BeBot instances reconfiguration scenario step 3. ................. 59
6.1 Cutout of the meta model for MECHATRONICUML component type level, extended by new elements of the reconfiguration concept. . . 61
6.2 Cutout of the meta model for MECHATRONICUML component instance level, extended by new elements of reconfiguration concept. 62
1 Introduction

Over the past few years, the portion and significance of software in technical devices increased very strongly. Automotive, medical systems, process control and manufacturing industries are only some domains in which the use of software is very important [KM10]. Although computers become smaller and therefore less and less visible, they affect our daily life to an ever-increasing extent. For example, complex software elements are included in modern TVs, cell phones, game consoles and also in non exclusively electronic devices like cars and robots. Today, software can actually be considered as the heart of many industrial systems [KM10]. Especially safety-critical systems like aircrafts or nuclear systems depend on precise operating software elements, because failures could result in loss of life, significant property damage, or damage to the environment [Kni02].

A system consisting of mechanic-, electronic-, software- and control engineering units is denoted as a mechatronic system [SW07]. Cars, as an example of a mechatronic system, often have to cope with changing conditions of their environment. A driving car, for example, has to reduce its velocity, if another car appears in front it. Thus, cars provide complex functionalities like adaptive cruise control or dynamic headlight leveling to automate reactions to changes in their environment. To realize these functions, an increasing number of networked mechatronic subsystems is needed. Additionally, the subsystems must be able to communicate with each other in real-time. As Software is used to control communication within mechatronic systems or networks of them, the software part of such systems becomes more and more large-scale and complex [SW07]. This results in high development costs, low productivity and unmanageable software quality [CLfW00].

A viable solution to cope with these problems is the “Component-based Software Engineering” (CBSE) approach [BHP06], which is based on building software systems using component specifications in a component model [Szy02]. CBSE combines elements of software architecture and modular software design [KM10]. It handles the complexity of large systems by use of the two key principles abstraction and reuse [BDG+11]. The component model can be abstracted from technical implementation details [BDG+11] and is therefore a less complex representation of the system. Furthermore, the components are reusable development artifacts and thus, identical solutions have not to be redeveloped [BDG+11].

In general, a component can be regarded as a black box with well defined interfaces and behavior [BHP06]. A software system is simply built as an assembly of components already developed and prepared for integration [KM10].
1. Introduction

1.1 Motivation

Mechatronic systems are often subject to changing conditions in their environment during run-time. Therefore, they have to be able to adapt their behavior in relation to these changes. We consider such a system to be adaptive [Ham09]. Some aspects of adaptivity are fault tolerance (continue working, even if a failure occurs), self-tuning (optimize system performance) and hot code updates (integrating improved software) [Ham09]. A car, for example, could automatically turn on its headlights to increase the range of vision, if the light intensity of the environment falls below a certain limit. After this, the car could increase its velocity and it thereby improves the system performance.

The Software Engineering Group of the University of Paderborn is developing a language to model distributed, component based mechatronic systems - the MECHATRONICUML. MECHATRONICUML is based on the “Unified Modeling Language” (UML) [Gro10], a standardized, general-purpose modeling language which can be used to characterize component-based software systems. UML is as a standard graphical modeling language for specification, construction and documentation of software.

The MECHATRONICUML reuses the well-known notations and concepts of UML and adapts them to the domain of mechatronic systems [Tic09]. It follows the CBSE approach and supports the development of structural as well as behavioral aspects of mechatronic systems [BDG+11]. In contrast to many other academic systems, which are only design-oriented and do not provide any run-time environment [BHP06], the MECHATRONICUML provides a component type- as well as a component instance level. The component type level is used to specify a mechatronic system at design-time, whereas the component instance level represents a concrete instantiation of a mechatronic system at run-time. We call a set of connected component instances a component instance configuration [BDG+11].

The adaptivity of component systems is denoted by reconfiguration [Ham09]. We call the modification of a system’s structure at run-time a dynamic reconfiguration [BHP06]. This means, that adaptivity of a MECHATRONICUML system can be achieved by performing a dynamic reconfiguration. One practical possibility to reconfigure a MECHATRONICUML system is to modify the configuration of its component instances [Hir08, Tic09].
1.2 Case Study

We will consider the following scenario as an ongoing example in this document to demonstrate the existing-, as well as the new concepts of the MECHATRONICUML.

Several autonomous robots drive in a known environment. As robots, we will use the miniature robot platform BeBot [HWR09] shown in Figure 1.1. The BeBot has been developed at the Heinz Nixdorf Institute\(^1\) at the University of Paderborn. The BeBot’s actuation consists of a chain drive. Twelve infrared sensors, controlled by two microcontrollers, are mounted on the BeBot to sense its environment. Additionally, the BeBot contains a bluetooth- and a wireless LAN module to communicate with other BeBots. Two USB ports provide extensibility of the BeBot’s functionality [HWR09]. In our scenario, we will add an USB-GPS-Receiver to each BeBot as it needs to detect its current position.

\[\text{Figure 1.1: BeBot mini robot (fully equipped) [HWR09].} \]

The BeBots are able to build convoys for the purpose of energy saving. A convoy consists of several BeBots, which drive one after another. One of these BeBots is assigned to be the convoy leader, the other BeBots are called convoy members. The leader BeBot drives in front of the convoy and coordinates the member BeBots by ordering them to drive to certain positions.

Given a situation where two BeBots already drive in a convoy and a third BeBot tries to join them. In this case, the “New Convoy Member” BeBot first has to establish contacts to the “Convoy Leader” BeBot and negotiate, if it is feasible to join the convoy. Figure 1.2 shows this situation.

\(^1\)http://wwwhni.uni-paderborn.de/en/
1.3 Problem Description and Objectives

We are now going to represent the scenario given in the case study introduced in the previous section by a MECHATRONICUML model. MECHATRONICUML offers a possibility to build hierarchical component models by embedding components into each other. We distinguish between Atomic Components, which form the lowest level of the MECHATRONICUML component model, and Structured Components, which are able to embed other Structured- as well as Atomic Components [BDG+11]. Components contain ports, which are used to send messages to communicate with other components (cf. Chapter 3).

Figure 1.3 shows three MECHATRONICUML Structured Component instances representing the BeBots of the case study. To keep it simple so far, we only display cutouts of the BeBot’s component instances, which are relevant for our scenario. We will consider a more detailed MECHATRONICUML model of the BeBot in Section 5.1.

The “newMember” BeBot has to establish a connection to the “leader” BeBot using a Real-Time Coordination Pattern (cf. Section 3.1.2). To do this, the “member” BeBot has to instantiate a new embedded component instance and two new port instances. The “leader” BeBot, on his part, has to create two new port instances. The green colored elements shown in Figure 1.3 are the ones to be created to add a new member BeBot to an existing convoy.
1.3 Problem Description and Objectives

Thus, this scenario requires a dynamic reconfiguration of component instances at run-time. The goal of this work is to develop a reconfiguration concept for the MechatronicUML, which provides modeling of a reconfiguration behavior at design time (component type level) as well as executing dynamic reconfigurations at run-time (component instance level).

Current reconfiguration approaches for MechatronicUML use graph transformations to modify a component instance configuration [HT08]. A graph transformation can be executed on a certain component instance configuration of a Component instance at run-time. Thus, the graph transformation defines the reconfiguration behavior of a MechatronicUML component. By now, it is not possible to model such a reconfiguration behavior of a Structured Component on design time, as a Structured Component derives its complete behavior from its embedded Atomic Components and does not contain any behavior description itself [BDG+11]. Reconfiguration behavior of a Structured Component could only be modeled within an embedded Atomic Component. However, each component encapsulates its inner structure and behavior and thus, a modification of its component instance configuration can only be performed by the component instance itself and not by other component instances. This means, that it is not possible to trigger an execution of a graph transformation to reconfigure a Structured Component instance by one of its embedded Atomic Component instances. For this reason, we will extend the Structured Components by an element which holds their reconfiguration behavior and is allowed to execute graph transformations on the Structured Component’s instance configuration to reconfigure it.

Furthermore, it should be possible to trigger a reconfiguration from any component instance on any hierarchical level within a MechatronicUML component.
instance configuration. We will provide this by introducing a reconfiguration interaction of component instances via message events to control the reconfiguration process.

The autonomous BeBots in our case study need to establish contacts to communicate with each other. Until now, the MECHATRONICUML does not provide a solution for establishing contacts between two or more autonomous MECHATRONICUML systems. Thus, this work also gives an idea of how to solve this problem by using concepts of the Bluetooth Link Manager Protocol [BS02].

1.4 Overview

We will consider some related work, which took influence on the results of this work, in the subsequent Chapter 2. After this, we will present the basics of the MECHATRONICUML, including existing reconfiguration techniques, in Chapter 3. Afterwards, we will introduce a concept for reconfiguration of MECHATRONICUML models in Chapter 4. The reconfiguration concept will be evaluated by an example shown in Chapter 5. The implementation of the reconfiguration concept within the Fujaba4Eclipse Realtime Toolsuite [PTH+10] and the corresponding extension of the MECHATRONICUML meta model will be presented in Chapter 6. Finally, we will summarize the conclusions of this work and give some ideas of future work in Chapter 7.
2 Related Work

Several component-based models already provide reconfiguration support, as it is an essential feature to model a flexible system. ACME [GMW00], Darwin [MK96] and Wright [All97] are key representatives of design-oriented component models. All of them use an Architecture Description Language (ADL) to model and analyze component architectures [BHP06]. The main advantages of design-oriented models are their powerful functionalities to model a system at design-time. Unfortunately, they do not offer any sort of run-time environment. Insufficient run-time support prevents these component models from being accepted more widely [BHP06].

Nevertheless, some advanced component models like OpenCOM [CBG+04] and K-Component [DC01] provide run-time environments. They are even able to modify an application’s architecture during execution, which enables them to perform dynamic reconfigurations. The disadvantage of the component models mentioned before is, that they do not provide a general modeling language to define these reconfigurations. Instead of that, they directly rely on the implementation programming language [DLLC08]. This limits the possibility to reflect run-time reconfigurations at design time, because reconfigurations have to be programmed instead of being modeled.

Another subject to be concerned are feasibility, validation and verification of a reconfiguration. This is often simply done in the context of static ADLs [DLLC08]. Sadly, this is not sufficient, because it is little flexible. Oreizy proposes in [Ore96] to additionally define an Architecture Modification Language (AML), which describes the modification operations of a system architecture. Furthermore, he demands to describe the constraints, which reconfigurations underlie, by an Architecture Constraint Language (ACL). A constraint, for example, could be, that a component has to be in a so called quiescent state [KM98], before executing a reconfiguration on it. The quiescent approach makes sure, that a component will not be reconfigured, if it currently performs an action which must not be aborted abruptly.

The Fractal component model [BCS04] provides a classic ADL (Fractal ADL). Moreover, it has been extended by two additional languages called FScript and FPath in [DLLC08]. FScript is a scripting language for the definition of reconfigurations, whereas FPath is a Domain Specific Language (DSL) to introspect Fractal architectures. Fractal is therefore able to satisfy the features of an Ar-
chitecture Modification Language as well as an Architecture Constraint Language and thus, reconfigurations may be modeled at design-time. As this is a good basis for developing reconfiguration solutions of component-based systems, Fractal has more and more been considered by current reconfiguration approaches.

In [BHR09], Bennour et al. introduced a “Reconfiguration Framework for Distributed Components”, which adapts from the Fractal framework and uses an extended version of FScript for reconfiguration issues. Their approach provides a non-centralized reconfiguration of a distributed, hierarchical component model at run-time. They propose to extend each component by a non-functional port, that is able to interpret reconfiguration orders.

Secondly, each component needs to implement a reconfiguration controller. This is quite easy to implement, as embedding of controller elements into components is a standard feature of the Fractal framework. The reconfiguration controller is allowed to execute reconfiguration scripts, defined by the FScript language, on a component at run-time. In case of nested components, the controller is furthermore able to invoke a reconfiguration of a sub-component by sending a re-configuration order to the sub-component’s reconfiguration port. This will cause the sub-component’s reconfiguration controller to reconfigure the sub-component, again by executing a FScript command.

The idea of adding a control part to a component, in order to handle a reconfiguration, has also been mentioned by Bures et al. in [BHP06]. They introduce reconfigurations in a hierarchical component model called “SOFA 2.0” and propose to model the control part as a composition of microcomponents. A microcomponent is defined as a minimalistic, flat component, which features whether connectors nor distribution. A microcomponent could just represent a class, embedded in a common component and implementing a specified interface.

We will consider these basic ideas and adapt parts of them to the reconfiguration concept for the MECHATRONICUML component model in Chapter 4. In particular, the MECHATRONICUML reconfiguration concept should provide a possibility to model reconfigurations on a high level at design-time and it should be independent of the implementation programming language. As the MECHATRONICUML component model allows to embed components into other components, it must also be possible to reconfigure components on each hierarchical level of a MECHATRONICUML model.
3 Fundamentals

This chapter describes the theoretical fundamentals and concepts, which are important for the understanding of this work. We will first consider the basics of the MECHATRONICUML in Section 3.1. As already existing reconfiguration techniques for MECHATRONICUML base on graph transformations, we will give a short introduction to graph theory in Section 3.2. In Section 3.3, we will finally introduce the concepts of Story Diagrams and Component Story Diagrams, which are currently used to model reconfigurations of MECHATRONICUML models.

3.1 MechatronicUML

The MECHATRONICUML is designed to support the development of software for advanced mechatronic systems [BDG+11]. It uses a hierarchical component model to specify these systems under construction and a component instance configuration to represent them during execution. Thus, we distinguish between component types and component instances. Component types are used to specify a MECHATRONICUML system at design-time, whereas a set of instances of these component types forms a MECHATRONICUML system at run-time. In general, component types are simply referred as components for the sake of easier readability [BDG+11]. Component instances communicate with each other by sending asynchronous messages, which are specified by message types. Real-Time Coordination Patterns are used to model the communication. The behavior of a MECHATRONICUML system results from the behavior of its components, which is specified by so called Real-Time Statecharts, an extension of UML state machines [Gro10] to model hard real-time features [BDG+11]. We will summarize some important basics of MECHATRONICUML in the following sections. For a more detailed description, we refer to [BDG+11].

3.1.1 Real-Time Statechart

Real-Time Statecharts [GB03] base on UML state machines [Gro10], combined with the timing behavior of timed automata [AD94]. They are used to describe the behavior of a role (cf. Section 3.1.2), a component or a port (cf. Section 3.1.3). A Real-Time Statechart consists of states, transitions, clocks, actions and variables [BDG+11].
A state represents a current situation of a system during run-time. The MechatronicUML supports simple states without hierarchy and embedded elements, as well as composite states, which offer to model hierarchy and orthogonal behavior by the use of regions. Regions provide to embed further statecharts into a composite state. In case of a composite state is active, each of its regions must have at least one active state. Different regions of a composite state can be synchronized by so called synchronization channels, which enable communication between these regions. We call a composite state with more than one region an orthogonal state. Furthermore, a state may contain three kinds of actions. Entry-Actions are executed each time the state becomes active, Exit-Actions accordingly as soon as the state is deactivated. Do-Actions are performed while the state is active. An action has a defined expression, which can be expressed in any programming language. Besides, it is possible to describe actions via graph transformation rules (cf. Section 3.2.2) or story diagrams (cf. Section 3.3.1) [BG11]. Moreover, it is possible to define variables and clocks within a state. Clocks are used to model real-time-critical behavior like deadlines.

Figure 3.1 shows an orthogonal state with two regions and two synchronization channels ("ch1()", "ch2()"). The state contains two variables “a” and “b” of datatype integer and two actions “op1” and “op2”. It furthermore includes two clocks “c1” and “c2”. The regions themselves contain their own variables and clocks.

![Concrete Syntax of an orthogonal state](image)

State changes of a statechart are modeled by transitions. The upper region of the orthogonal state shown in Figure 3.1, for example, contains a transition which enables to switch from state “A” to state “B”. Transitions fire, if their source state is active and the required constraints are fulfilled. Constraints can be in form of guards, requiring a certain value of a variable, or timeguards, which depend on certain values of clocks. Real-Time Statecharts are able to communicate with other Real-Time Statecharts by sending asynchronous messages. These messages can be raised by transitions.
Furthermore, constraints of transitions may depend on receiving a certain message. Moreover, a firing transition may send a synchronization or perform a side effect in form of executing a certain action. Additionally, deadlines and priorities can be defined for a transition to get deterministic behavior of the Real-Time Statechart.

Figure 3.2 shows an example of a concrete transition with nearly all allowed elements. We will consider examples of complete Real-Time Statecharts in Chapter 4 and 5.

![Figure 3.2: Concrete Syntax example of a transition [BDG+11].](image)

### 3.1.2 Real-Time Coordination Pattern

Real-Time Coordination Patterns specify the communication behavior of MECHATRONICUML components. This behavior is defined by a message- and state based communication of exactly two communication partners, which are named as roles [BDG+11]. Role instances communicate with each other using a communication connector to exchange asynchronous messages. These messages are specified by signatures, so called message types. A message interface defines a set of message types that may be exchanged between two roles of a Real-Time Coordination Pattern. Therefore, each role may specify which messages it is able to send or receive by implementing a sender message interface as well as a receiver message interface. A role which simply implements a sender message interface is only able to send messages, thus, we call it an out-role. Accordingly, an in-role denotes a role that simply implements a receiver message interface and is therefore only able to receive messages. Thereby, it is possible to model unidirectional- as well as bidirectional communications. MECHATRONICUML furthermore provides to model a 1:n communication between role instances by using a multi-role. A multi-role can be instantiated several times up to the upper bound of its cardinality. The behavior of each role is specified by a Real-Time Statechart (cf. Section 3.1.1). It is possible to define variables and operations within a role. Variables
are used to store information, whereas operations implement the actions of the states and the side effects of the transitions. A Real-Time Statechart is allowed to access the variables and call the operations of the role it belongs to, but it has no access to variables or operations of other roles. Finally, the behavior of a Real-Time Coordination Pattern results from the concurrent execution of both of its roles [BDG+11].

Figure 3.3 shows the syntax of a Real-Time Coordination Pattern. The dashed ellipse contains the name of the pattern and the solid squares represent the roles. The role on the left side is a multi-role (visualized by the cascaded double border line of the square), whereas the one on the right side represents a single-role. The ellipse and squares are connected by two dashed lines, which are labeled by the names of the roles. 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.

![Real-Time Coordination Pattern example.](image)

### 3.1.3 Component Model

The MECHATRONICUML specifies the structure of a system by a hierarchical component model. Each component has clearly defined interfaces and contains ports to communicate with other components. It is possible to model 1:1 communication between two components using single-ports, as well as 1:n communication by the use of multi-ports. Furthermore, a component encapsulates its inner structure and behavior and may not be accessed directly by other components [BDG+11]. Access to components is only provided by their ports. The MECHATRONICUML distinguishes between Atomic Components and Structured Components, which are described in the following sections.

**Atomic Component Type**

Atomic Components are flat components, which form the lowest level of the MECHATRONICUML component model [BDG+11]. They include a behavior specification directly, defined by their ports. Figure 3.4 shows an Atomic Component type with one single-port and one multi-port.
The developer assigns a role of a Real-Time Communication Pattern to each port of an Atomic Component. The port’s behavior is then given by a refinement of the role’s Real-Time Statechart [BDG+11]. Thus, analogous to a role, a port may define attributes to store data and operations that implement the actions of the states and the side effects of the transitions.

The behavior of an Atomic Component results from the behavior of its ports. Therefore, the behavior specification of an Atomic Component is given by a Real-Time Statechart consisting of one state with parallel regions for the statecharts of the ports. At least, it contains one region for each port. Additionally, a synchronization region is available to model the internal behavior of a component. Furthermore, a set of synchronization channels is needed to provide internal communication between the regions.

Figure 3.5 shows a Real-Time Statechart for the Atomic Component type shown in Figure 3.4. The region “single-port” contains the Real-Time Statechart of the single-port and the region “multi-port” contains the one of the multi-port. The internal behavior of the component is represented by the synchronization statechart in region “synchronization”. The communication between the parallel regions is handled by the synchronization channel “syncChannel()”.

Furthermore, a Atomic Component may also define attributes and operations. These attributes and operations can only be accessed by the Atomic Component’s synchronization statechart and not by the port’s statecharts.

We will consider more detailed examples of Atomic Components and their behavior descriptions in Chapter 4 and 5.
3. Fundamentals

Figure 3.5: Statechart of an Atomic Component with a single-port, a multi-port and a synchronization.

**Structured Component Type**

Structured Components are used to build a hierarchical model, as they are assembled by embedding other components. These embedded components are denoted by component parts. A component part specifies a cardinality which defines the minimum and maximum number of part instances allowed within a Structured Component instance. If more than one part instance is allowed, we call the component part a *multi-part*, otherwise a *single-part*.

As ports of a Structured Component do not contain a behavior specification, they simply delegate messages to ports of their component parts. Thus, the behavior of a Structured Component solely results from the behavior of its embedded parts. We call a connection from a port of a Structured Component to a port of a component part a *delegation*. The connection between ports of two component parts is called an *assembly*.

Figure 3.6 shows a Structured Component type “StrucComp” with two embedded component parts. The port “p1” of the Structured Component is connected to the port “p2” of component part “Atom1” via a delegation. Part “Atom2” is a multi-part, whose port “p4” is connected with port “p3” of part “Atom1” by an assembly. Port “p3” is a multi-port, because we need multiple instances of this port to communicate with multiple part instances of component type “Atom2”.
3.1 MechatronicUML

3.1.4 Component Instance Configuration

A component instance is a concrete occurrence of a Atomic- or Structured Component type. During instantiation, ports, embedded parts and component specific attributes of the component type are assigned to concrete values. Thus, a component instance contains concrete port- and part instances.

Component instances are connected by connector instances according to the connected Real-Time Coordination Patterns [BDG+11]. We call a set of connected component instances a Component Instance Configuration. The MechatronicUML uses these configurations to provide run-time support. A developer creates concrete component instances and according configurations at design-time and these are applied to the system at run-time. Each Structured Component instance contains a component instance configuration of its part instances.

Figure 3.7: Component Instance Configuration with two connected component instances.

Figure 3.7 shows a Component Instance Configuration consisting of a component instance “a1” of type “A” and a component instance “b1” of type “B”, which are connected by a Real-Time Coordination Pattern “RCP”. Port “p1” implements role1 of the Real-Time Coordination Pattern while port “p2” implements role2. A component instance looks similar to a component type, but its name begins with a lower letter. The name is followed by a colon and the name of
the component type, that the component instance is derived from. Port instances are labeled in the same way as port types.

3.2 Graphs and Graph Transformations

Graph transformation is widely used for expressing model transformations. As graphs are capable to describe the structures of component-based models, modeling of transformations can be naturally formulated by graph transformations [GY03]. Thus, current reconfigurations of MECHATRONIC UML models are realized by graph transformations. We will introduce the basics of graphs and graph transformations in the following sections.

3.2.1 Graphs

A graph is an abstract representation of a set of objects which can be pairwise connected. These interconnected objects are denoted as vertices, whereas the connections between them are called edges. A graph may be undirected, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another [GY03]. In the following, we will consider directed graphs as shown in Figure 3.8.

![Figure 3.8: Example of a directed graph.](image)

Directed graphs underlie some limitations. It is especially not possible to represent different types of objects by the vertices. As this is necessary for the concept of Story Patterns introduced in Section 3.3.1, we will use extended directed graphs, so called object-oriented graphs, to model reconfigurations. A formal definition of object-oriented graphs can be found in [Zün01].

Let’s take a look at a simple example of how a MECHATRONIC UML component can be represented by an object diagram. Given a Structured Component instance with one embedded Atomic Component instance and a delegated port instance as shown in Figure 3.9.
Figure 3.9: Structured Component instance with an embedded Atomic Component instance.

Figure 3.10 shows a corresponding object diagram of the Structured Component instance given in Figure 3.9, based on the MECHATRONICUML meta model. The nodes represent object instances of the corresponding classes of the meta model. As you can see, the object diagram contains a node for the Structured Component instance “s1” of type “S” and the Atomic Component instance “a1” of type “A”. Furthermore, there are nodes for the two port instances and finally one node for the delegation instance, which connects the two ports. The edges between the nodes define the associations between the object instances.

Figure 3.10: Example of an object diagram of a Structured Component’s instance configuration with one embedded Atomic Component instance.

3.2.2 Graph Transformations

Graph transformations describe the action of automatically creating a new graph out of an original graph. We denote the original graph by host graph. Formally, a graph transformation system consists of a set of graph transformation rules of
the form $L \rightarrow R$, with $L$ being called left-hand side (lhs) (or pattern graph) and $R$ being called right hand side (rhs) (or replacement graph) [EGdL+05]. $L$ defines the pre-conditions for executing the transformation rule, while $R$ represents the post-conditions.

The pre-conditions of a rule are fulfilled, if a match $m$ of the left hand side $L$ can be found in the host graph. This means, it must be possible to map each element of the lhs to an element of the host graph. Executing a transformation rule deletes all elements $\text{oldElem} \in L \setminus (L \cap R)$ from the host graph, which are part of the lhs, but not of the rhs. Analogically, it creates a new element in the host graph for each $\text{newElem} \in R \setminus (L \cap R)$, which appears in the rhs, but not in the lhs [EGdL+05]. All other elements $e \in (L \cap R)$ keep untouched while executing a graph transformation rule. In other words, a transformation rule is applied to the host graph by searching for an occurrence of the left hand side $L$ and by replacing the found occurrence by an instance of the right hand side $R$. We will consider an example of how to use graph transformations to modify the structure of a MECHATRONICUML component instance configuration in the upcoming section.

### 3.3 Reconfiguration of MechatronicUML models

We will consider the example introduced in Section 3.2.1 to demonstrate how the structure of a MECHATRONICUML component instance can be modified by executing a graph transformation rule. First, we will extend the Structured Component instance “s1” by one additional embedded Atomic Component “b1” of type “B” as shown in Figure 3.11.

![Figure 3.11: Structured Component instance with two embedded Atomic Component instances.](image)

Now, we want to replace the embedded component instance “b1” of type “B” by a component instance “c1” of type “C”. Thus, we have to destroy the “b1” instance, its assembly instance and the corresponding port instances on “a1” and “b1”. This means, that these elements have to appear in the lhs, but not in the rhs of our transformation rule. After this, we have to create a new part instance “c1” of type “C”, a new assembly instance and the corresponding port instances to connect the new part instance “c1” with part instance “a1”. So, these elements
have to be part of the rhs, but not of the lhs. Figure 3.12 shows the transformation rule for this scenario and the corresponding match of the lhs in the host graph.

Figure 3.12: Example of a graph transformation rule to reconfigure a Structured Component instance.

### 3.3.1 Story Patterns and Story Diagrams

Graph transformation rules may quickly become confusing when describing more complex transformations of greater models. Thus, [Zün01] introduced a short notion of a graph transformation rule, called *Story Pattern*. Story Patterns base on uml collaboration diagrams [Gro03] and they combine the left hand side and the right hand side of a graph transformation rule into one diagram to simplify the modeling. Stereotypes are added to elements of a Story Pattern, which have to be created (<<create>> or <<+++>>) or rather destroyed (<<destroy>> or <<−−−>>). All elements without a stereotype keep untouched by execution of a Story Pattern. For executing a Story Pattern, the lhs and rhs have to be derived from the Story Pattern and then, a general graph transformation, as described in Section 3.2.2, can be performed.
Figure 3.13 shows a Story Pattern for the example scenario given in the beginning of this section to replace a component instance of type “B” by a component instance of type “C” within a Structured component instance of type “S”. It represents the graph transformation rule defined in Figure 3.12 as a Story Pattern.

Story Pattern can be embedded into uml activity diagrams [Gro10] to model sequences of Story Patterns, including loops and decisions [Zü10]. We call these activity diagrams *Story Diagrams*. Story Diagrams consist of single Story Patterns, which are connected via control flow edges.

These are only the simplified basics of Story Patterns and Story Diagrams, which are required for the understanding of this work and the given examples. For a more detailed explanation, we refer to [FNTZ00] and [Zü10].

### 3.3.2 Component Story Patterns and Component Story Diagrams

Component Story Patterns have been introduced in [HT08] to ease the modeling of graph transformation rules within the MECHATRONIC UML. They base on the concepts of Story Patterns, but they provide modeling of transformations on a higher level, as they use the syntax of the MECHATRONIC UML component instances. This simplifies the modeling process, as it is no more necessary to switch between the component type model and the meta-model to define transformations of a MECHATRONIC UML model. Furthermore, a transformation, defined by a Component Story Pattern, is much more compact than defined by a general Story Pattern.

Component Story Diagrams, analogically to the concept of Story Diagrams, embed single Component Story Patterns and connect them by the use of control flow edges to model a course of transformation actions.
3.3 Reconfiguration of MechatronicUML models

Again, we will consider the example of replacing an embedded component instance by another one of different type within a Structured Component instance, as described in the beginning of this section. Figure 3.14 shows the Component Story Diagram representing the required transformation.

Figure 3.14: Simple Component Story Diagram to reconfigure a Structured Component instance.

To execute the Component Story Diagram, we have to break down each contained Component Story Pattern into its \( \text{lhs} \) and \( \text{rhs} \) to perform the corresponding actions. A system at run-time transforms the Component Story Patterns into equivalent Story Patterns and executes them. Figure 3.15 shows the matchings of the \( \text{lhs} \) and \( \text{rhs} \) for the Component Story Diagram introduced before. To keep it simple, the Component Story Diagram only contains one Component Story Pattern.

Figure 3.15: Example of executing a Component Story Diagram on a component instance configuration.
The left hand side contains a component instance of type “A” connected through an assembly instance to a component instance of type “B”. We map the component-, port- and assembly instances to the instances “a1” and “b1” and their corresponding port- and assembly instances in our instance configuration. Thus, the pre-conditions to execute the Component Story Pattern are fulfilled. As the right hand side does not contain an instance of type “B”, we have to delete the instance “b1” and the corresponding port- and assembly instances in our instance configuration. In contrast to the lhs, the rhs contains an instance of type “C”. Thus, we have to add an instance “c1” with corresponding port- and assembly instances, as well as a new port instance on “a1”, to our instance configuration.
4 MechatronicUML Reconfiguration Concept

This chapter describes a reconfiguration concept for MechatronicUML models. As mentioned in Chapter 1, the aim of reconfiguring a component based software system is to adapt its behavior to changes in their environment. This can be realized by a structural- as well as a parameter adjustment [FGK+04]. The MechatronicUML reconfiguration concept introduced in the following is based on a structural adjustment of component instances. A structural adjustment is more powerful than a parameter adjustment and it offers a wider range of possibilities to model reconfiguration actions [FGK+04]. As parameter adjustment may be needful in some cases as well, the concept also provides a possibility to model this. Furthermore, the concept offers an efficient solution to model programming language independent reconfiguration behavior at design-time, which can be executed at run-time to perform dynamic reconfigurations.

We will pick up some ideas of the reconfiguration approaches for the Fractal framework [BHR09] and the SOFA 2.0 framework [BHP06] introduced in Chapter 2. Besides, we will consider reconfigurations of Atomic- as well as Structured Components and we will finally give an example of how a reconfiguration request could be created within a MechatronicUML system.

In regard to the complexity of this work, the reconfiguration concept does not consider any verification aspects, neither before nor after executing a reconfiguration, so far. Time aspects are also ignored by now. However, we will pick up these issues and give some ideas to extend the reconfiguration approach in Section 7.1.

4.1 Reconfiguration Message Event Flow

The MechatronicUML component model provides a hierarchical configuration by the use of Structured Components [BDG+11]. Thus, it could be necessary to modify the behavior of component instances on each hierarchical level to reconfigure a MechatronicUML system. As described in Section 3.1.3, each component encapsulates its inner structure and behavior and can not be accessed directly by other components [BDG+11]. This means, a Structured Component instance is not eligible to modify the behavior of its embedded part instances directly. Each
4. MechatronicUML Reconfiguration Concept

A component instance has to handle a reconfiguration on its own, as there is no possibility to do this from outside. However, it would be nice to control the reconfiguration process of the whole system and it should be possible to trigger a reconfiguration process from anywhere within the system.

The MechatronicUML reconfiguration concept achieves this by modeling a special reconfiguration interaction of the components. As components can only interact with each other via their ports, starting from now, each component which provides reconfiguration functionality needs a bidirectional reconfiguration communication port. Thus, it is possible to define message types whose corresponding message events trigger reconfiguration actions within a component instance. Reconfiguration messages will contain a parameter “id”, which is the unique key of a reconfiguration action to be performed. Each time the reconfiguration port instance of a component instance receives a reconfiguration message event, the component instance performs a reconfiguration corresponding to this message event. In case of a Structured Component, the reconfiguration message event has to be propagated to the embedded component part instances, as they possibly also need to reconfigure themselves. Thereby, the reconfiguration message event flow covers all hierarchical levels of a MechatronicUML component instance configuration.

As the reconfiguration ports are bidirectional, it is possible to send a reconfiguration message event from a deeper hierarchical level to a higher one and vice versa. In other words, any component instance, regardless of its hierarchical level, could be able to trigger a reconfiguration request for the component instance configuration it belongs to, if the developer enables it. Certainly, this depends on how the system is built and how the reconfiguration process is modeled.

Figure 4.1 shows a Structured Component with full reconfiguration functionality and two embedded Atomic Components. The element “RPM” (cf. Section 4.3.2) handles the reconfiguration message event flow. It is connected via its inner-rc multi-port to the outer-rc ports of the embedded components to propagate reconfiguration message events. It furthermore receives reconfiguration message events from outer components via its outer-rc port. The element “REC”, which is responsible to execute a reconfiguration, will be described in Section . Section 4.3.3 will introduce the element “CSM”, which holds a certain configuration state.
4.1 Reconfiguration Message Event Flow

Figure 4.1: Fully reconfigurable Structured Component with two embedded Atomic Components

Figure 4.2 shows a sequence diagram of an example reconfiguration message event flow of a Structured Component instance with two embedded part instances in case of a successful reconfiguration. The Structured Component instance gets a reconfiguration request from outside and propagates this request to “Embedded Atomic Component1 Instance” first. This instance executes the corresponding reconfiguration of its structure and sends a success response to the Structured Component instance. After this, the Structured Component instance propagates the reconfiguration request to the second embedded Atomic Component instance, which as well reconfigures itself and sends a success response. Finally, the Structured Component instance executes the remaining modifications of its own structure and sends a success response to the sender of the reconfiguration request.
4. MechatronicUML Reconfiguration Concept

Figure 4.2: Sequence Diagram of reconfiguration message event flow within a Structured Component instance with two embedded Atomic Component instances.

The upcoming Sections 4.3, 4.4 will describe the reconfiguration of Structured- and Atomic Components in more detail.

4.2 Reconfiguration by Component Story Diagrams

The concepts of Story Diagrams and Component Story Diagrams have been introduced in Chapter 3 (Section 3.3.1, 3.3.2). The new reconfiguration concept proposes to use Component Story Diagrams to specify how the structure of a component has to be changed to perform a reconfiguration, as they are independent of the implementation programming language.

We prefer Component Story Diagrams instead of defining a reconfiguration language like FScript [DLLC09, DR10], used in the Fractal framework, because they offer a more user friendly way to model a reconfiguration behavior on a high level and therefore ease the entire modeling process. As Component Story Diagrams are modeled on component type level and can furthermore be executed on component instance level, they provide a very usable way to define the structural modification of a component instance.

The current definition of Component Story Diagrams [HT08] allows to directly create and destroy port instances on embedded component instances. As this violates the principle that components may not be accessed by other components directly [BDG+11], we will prohibit this in the future. Only a component instance itself should be allowed to create or destroy its port instances. We will provide this
by the reconfiguration message event flow interaction of the component instances. A more detailed explanation of this issue will be given in the subsequent sections and by the example in Chapter 5.

4.3 Reconfiguration of Structured Components

The behavior of a MechatronicUML Structured Component only results from the behavior of its component parts and multi-parts and the connections between them [BDG+11]. This limits the possibility to model the reconfiguration behavior of a Structured Component on component type level, because it does not contain an element which is able to hold the reconfiguration information. The embedded parts and multi-parts alone are not sufficient to model a complete reconfiguration behavior of a Structured Component. As mentioned in Section 4.2, the reconfiguration action is defined by a Component Story Diagram, which has to be executed on the component instance configuration of the Structured Component instance. Part and multi-part instances, which simply represent embedded Atomic- or Structured Component instances, are only allowed to change their own configurations, but no configuration of other component instances. They are particularly not able to execute a Component Story Diagram which changes the configuration of the Structured Component instance containing them, because each component encapsulates its inner structure and behavior [BDG+11]. Therefore, the MechatronicUML has to be extended by a new element which offers a possibility to attach Component Story Diagrams to a Structured Component type and execute them on the corresponding Structured Component instances.

Referring to the concepts of controller elements used in the FRACTAL framework [DR10] and microcomponents introduced in SOFA 2.0 [BHP06] (c.f. Chapter 2), this MechatronicUML reconfiguration approach adds a new element called Reconfiguration Execution Controller (REC) to a Structured Component. The Reconfiguration Execution Controller holds the required Component Story Diagrams for reconfiguration, it is able to execute them on the component instance configuration of the Structured Component instance and it provides access to activate a reconfiguration process. A detailed specification of the Reconfiguration Execution Controller is given in the subsequent Section 4.3.1.

Furthermore, we need an element which handles reconfiguration message events and propagates them to the embedded parts. Thus, we will introduce the Reconfiguration Process Manager (RPM) in Section 4.3.2.

Finally, we would like to store the current configuration state of a Structured Component instance. Therefore, we will introduce a third new element, called Configuration State Manager (CSM), in Section 4.3.3.
4.3.1 Reconfiguration Execution Controller (REC)

Each Structured Component has to be extended by a \textit{Reconfiguration Execution Controller} (REC) to provide reconfiguration functionality. A REC is a special element which is always part of a Structured Component. It is allowed to execute a Component Story Diagram on the component instance configuration of its Structured Component instance to reconfigure it. A REC contains a \textit{reconfiguration execution port} to send and receive reconfiguration message events. Furthermore, the REC includes an allocation table of reconfiguration message ids and Component Story Diagrams. In contrast to the reconfiguration communication port introduced in Section 4.1, whose behavior can be modeled freely without limitations, the reconfiguration execution ports of all RECs have to behave in the same way. They receive reconfiguration message events and execute corresponding Component Story Diagrams.

Thus, the functionality of the REC during runtime is straightforward. Each time it receives a reconfiguration message event, it looks up the corresponding Component Story Diagram in the allocation table and tries to execute it. Finally, it sends back a message whether the reconfiguration has been successful or not. The REC does not contain any further logic or behavior, it simply executes Component Story Diagrams.

Figure 4.3 shows a Structured Component with a Reconfiguration Execution Controller. The REC is pictured as a rectangle with a cogwheel in the upper right corner to differentiate it from an embedded part. The REC lies directly against the upper border of the Structured Component’s lower compartment to visualize that the REC directly belongs to it. The reconfiguration port is labeled by the letter “r” to differ from other ports. Furthermore, the figure shows an example of an allocation table of message ids and Component Story Diagrams attached to the REC.
4.3 Reconfiguration of Structured Components

4.3.2 Reconfiguration Process Manager (RPM)

A Structured Component instance has to exchange reconfiguration message events with other component instances on higher and deeper hierarchical levels. As a Structured Component instance should first evaluate an incoming reconfiguration request before executing the corresponding reconfiguration, the reconfiguration port of the Reconfiguration Execution Controller should not be accessible from outside the component. Therefore, an additional element is needed to model this communication- and evaluation behavior. We denote this new element as Reconfiguration Process Manager (RPM), which can be modeled by an Atomic Component. The RPM holds the logic to handle a reconfiguration request within a Structured Component. Thus, the RPM and the REC can be considered as the core elements of reconfiguration. The RPM decides what to do, while the REC knows how to it and executes the modification.

The RPM has two reconfiguration communication ports (rc-port) to send and receive reconfiguration message events. One inner-rc-port to provide interaction with embedded components via their rc-ports and another outer-rc-port to interact with components from outside. For this reason, the Structured Component also needs an outer-rc-port which is delegated to the outer-rc-port of the RPM to make the port accessible from outside.

Furthermore, each embedded part instance of a Structured Component instance
has to be connected to an inner-rc-port instance of the RPM. Thus, the inner-rc-port has to be a multi-port, as an arbitrary number of part instances could be connected. The outer-rc-port is a single port as it can only be connected to an inner-rc-port of another RPM.

Figure 4.4 shows a Structured Component with a RPM connected to the RPC via the “Reconfiguration Execution Pattern”. The RPM has an inner-rc multi-port to propagate reconfiguration message events to embedded parts. The outer-rc port of the Structured Component is delegated to the outer-rc port of the RPM to exchange reconfiguration message events with other components.

The reconfiguration behavior of the Structured Component can now be modeled within the rc-ports. The outer-rc-port instance will receive a reconfiguration message event and firstly propagate it through its inner-rc-port instances to the embedded part instances to activate their reconfiguration processes. Secondly, it could be necessary to modify the configuration of the Structured Component itself, for example, to create or destroy some port-, assembly- or delegation instances. To do this, the reconfiguration message event has to be propagated to the REC. Thus, a third port is needed to connect the RPM to the reconfiguration execution port of the REC using a Reconfiguration Execution Pattern. The reconfiguration process of the embedded part instances should be finished before the REC mod-
ifies the structure of the Structured Component instance. This can be achieved by not sending the reconfiguration message event to the RPC until a response of each part instance has been received.

Reconfiguration Execution Pattern

The Reconfiguration Execution Pattern defines the communication behavior between an entity which controls the reconfiguration process (in this case the RPM) and the REC. It has one role “Reconfiguration Commander” which, in general, will be implemented by the RPM and a second role called “Reconfiguration Executer” which will always be implemented by the REC. Figure 4.5 shows the Reconfiguration Execution Pattern.

![Reconfiguration Execution Pattern](image)

Figure 4.5: Real-Time Coordination Pattern “Reconfiguration Execution”

Role “Reconfiguration Commander”

The role “Reconfiguration Commander” behaves straightforward. Each time it receives a reconfiguration message event, it sends a message event, including the id of the received reconfiguration message event as a parameter, to the role “Reconfiguration Executer”.

Figure 4.6 shows a Real-Time Statechart of the role “Reconfiguration Commander”. Transition “t1” fires, if a message event “reconfRequest(string id)” occurs. Firing transition “t1” results in triggering a “doReconf(string id)” event to the role “Reconfiguration Executer” and state “Wait” becoming active. Receiving a “success” message event reactivates state “Idle”.

31
4. MechatronicUML Reconfiguration Concept

Role “Reconfiguration Executer”

The role “Reconfiguration Executer” receives a “doReconf(string id)” event from the role “Reconfiguration Commander” and calls an action which executes the Component Story Diagram, which corresponds to the received message id.

Figure 4.7 shows a Real-Time Statechart of the role “Reconfiguration Executer”. Transition “t1” fires, if a message event “doReconf(string id)” occurs. Firing transition “t1” activates state “Execution”, whose entry action “ExecuteComp-StoryDiag(string id)” executes the corresponding Component Story Diagram on the Structured Component’s instance configuration. After this execution, the role “Reconfiguration Executer” sends a “success()” message to the role “Reconfiguration Commander” and the state “Idle” becomes active again.

4.3.3 Configuration State Manager (CSM)

Reconfigurations may underlie conditions to be allowed. For example, specific reconfigurations could only be feasible, if the system currently has a certain configuration. Thus, a current configuration state of the component instance is needed to verify, if a reconfiguration may be executed or not. Thus, an element called Configuration State Manager (CSM) can be added to a Structured Component to model the available configuration states. The CSM simply needs to hold the current configuration state and provide a possibility to interact with components which need to know the current state. Although the Configuration State Manager could be modeled by an arbitrary kind of complex element, we’ll keep it simple.
for now and use an Atomic Component with one port to hold the configuration state logic. Some ideas of a more complex CSM modeling will be considered in Section 7.1.

The Configuration State Manager has to be connected to the Reconfiguration Process Manager via the “stateMan” port using the Configuration State Pattern as shown in figure 4.8. Thus, we have to add an additional port “stateReq” to the RPM. The RPM gets the current configuration state from the CSM and also informs it about reconfiguration state changes.

![Structured Component Diagram](image)

Figure 4.8: Structured Component with Reconfiguration Process Manager, Reconfiguration Execution Controller and Configuration State Manager.

Configuration State Pattern

Figure 4.9 shows the Configuration State Pattern with the two roles “State Requester” and “State Manager” which defines the communication behavior between the Reconfiguration Process Manager and the Configuration State Manager. The role “State Manager” holds the available configuration states, at which one of them is active. The role “State Requester” sends a “stateRequest” message to the role “State Manager”, whereon the role “State Manager” sends a “stateResponse” message, including the id of the currently active state as a parameter.
4. Mechatronic UML Reconfiguration Concept

Concrete Real-Time Statecharts of the roles “State Requester” and “State Manager” will be shown in Section 5.3.

4.4 Reconfiguration of Atomic Components

Reconfiguration actions of Atomic Components are also defined by Component Story Diagrams. Given that an Atomic Component contains a behavior description directly and it does not have to reconfigure any inner parts, there is neither a need for a REC nor a RPM or a CSM within an Atomic Component. The reconfiguration behavior can directly be modeled within the Atomic Component type. It simply needs the already introduced additional reconfiguration communication port, which describes the complete reconfiguration behavior. Analogically to the REC, the reconfiguration port instance has to listen for message events (including a parameter “id”) and execute corresponding Component Story Diagrams, if such a message event arrives. Figure 4.10 shows an Atomic Component with a reconfiguration port “outer-rc”.

The behavior of an Atomic Component’s port is defined by a Real-Time Statechart (see Section 3.1.1). Furthermore, actions can be defined within an Atomic Component, which can be executed by a synchronization statechart of the component. Thus, it is possible to define a Component Story Diagram as an action.
4.4 Reconfiguration of Atomic Components

of an Atomic Component and execute it out of a synchronization statechart to reconfigure the Atomic Component. Therefore, the outer-rc port has to send a synchronization, including the Component Story Diagram to be executed as a parameter, to the synchronization statechart, each time it receives a reconfiguration message. After this, the synchronization statechart executes the Component Story Diagram.

A simple reconfiguration behavior of an Atomic Component containing a reconfiguration port is shown by the Real-Time Statechart in figure 4.11.

![Real-Time Statechart of an Atomic Component with simple reconfiguration behavior.](image)

Compared to the reconfiguration a Structured Component, the region “reconfiguration” performs the actions of the Reconfiguration Process Manager, whereas the tasks of the Reconfiguration Execution Controller are executed within the “synchronization” region. The reconfiguration port (region “reconfiguration”) listens for the reconfiguration message event “reconfRequest(string id)”. If this message event occurs, and the parameter id equals “reconf1”, then the transition “t1” fires and sends a synchronization via the synchronization channel “doReconf(ComponentStoryDiagram csd)” to the synchronization statechart. Furthermore, state “Execution” of region “reconfiguration” becomes active. The synchronization “doReconf” has a parameter “csd”, which defines the Component Story Diagram to be executed.

The synchronization triggers transition “t2” to fire, which sets the variable “re-
confCSD” of type Component Story Diagram to the value of parameter “doReconf.csd”. Subsequently, the state “Execution” of the synchronization statechart becomes active. As the variable “reconfCSD” of type Component Story Diagram is an entry action of the state “Execution”, it will immediately be executed and the reconfiguration of the Atomic Component takes place. After executing the Component Story Diagram “reconfCSD”, transition “t3” of the synchronization statechart fires, the “Idle” state becomes active again and a synchronization “reconfDone()” has been triggered. This enables transition “t4” to fire which results in state “Scan” becomes active and a “success” message event has been sent to the sender of the “reconfRequest” Message Event. By now, we ignore the case of a faulty reconfiguration and refer to further developments to solve this problem.

This simple reconfiguration behavior can be extended by adding more reconfiguration message ids to listen for and define their corresponding Component Story Diagrams. This can easily be done by adding transitions from state “Scan” to state ”Execution”, containing guards for the ids of the reconfiguration message events which should trigger the reconfigurations. These transitions must also send a synchronization “doReconf” with a concrete Component Story Diagram as a parameter value, depending on the reconfiguration message event id defined by the guard, to the synchronization statechart (analogically to transition “t1”).

A parameter adjustment can also be modeled by this Real-Time Statechart by simply changing parameter values instead of executing Component Story Diagrams when a defined reconfiguration message event arrives.

**Reconfiguration Propagation Pattern**

Embedded Atomic Components are connected to the inner-rc multi port of a Structured Component using the Reconfiguration Propagation Pattern. It consists of a multi-role “Reconfiguration Master”, which will primary define the behavior of the inner-rc multi-port of the RPM and a role “Reconfiguration Slave”, which can be implemented by the outer-rc port of a Structured- as well as Atomic Component. We give an idea, of how statecharts of these roles could look like in the following.

![Figure 4.12: Realtime Coordination Pattern “Reconfiguration Propagation”](image-url)
Role “Reconfiguration Master”

The role “Reconfiguration Master” specifies the behavior of a Structured Component instance to propagate incoming reconfiguration messages to the embedded part instances and to send back success messages, when a reconfiguration has been executed successfully. The reconfiguration of all part instances must be finished, before a Structured Component instance is allowed to reconfigure itself by modifying the highest level of its component instance configuration.

![Real-Time Statechart of the role “Reconfiguration Master”](image)

Figure 4.13: Role “Reconfiguration Master” of Pattern “Reconfiguration Propagation”

Figure 4.13 shows a Real-Time Statechart of the role “Reconfiguration Master”. The inner-rc port of a Structured Component is a multi-port. Thus, an arbitrary number of inner-rc port instances could exist and an incoming reconfiguration message has to be propagated to each of these port instances. Therefore, the Real-Time Statechart consists of two regions “adaptation” and “sub-role[k]”. The “sub-role[k]” region represents the behavior of the individual port instances, whereas the region “adaptation” handles the propagation on the whole. The statechart “Active” within the “sub-role[k]” region exists for each port instance. Starting from state “Idle” in region “sub-role[k]”, receiving a “reconfRequest(string id)” message triggers a synchronization “doReconf(string id)”, which enables state “Init” of region “adaptation”. This starts the propagation process. It means, if one reconfiguration port instance receives a reconfiguration message, this message

37
will be propagated to all other reconfiguration port instances. This works as follows.

Active state “Init” triggers a synchronization “next[1](id)” and activates state “Propagate”. Through the synchronization “next[1](id)”, the state “Wait” of the statechart of the first port instance becomes active and a “reconfRequest(string id)” has been send via the first port instance. Thus, the reconfiguration request has been propagated to the first part instance. Receiving a “reconfSuccess” message from the first part instance lets the first port instance become idle and send a “next[2](id)” synchronization. This enables the second port instance to send a “reconfRequest(string id)” message to its connected part instance. The second port instance again waits for a “reconfSuccess” message of its part instance. If the message arrives, it sends a “next[3](id)” to activate the next port instance. This will be repeated, until all port instances have propagated the reconfiguration message. If this is the case, the last port instance triggers a “next[n+1](id)” synchronization, which enables state “Reconfigure” of region “adaptation” and sends an “IntRecReq(id)” message to REC, which invoke the reconfiguration of the Structured Component instance itself. If the REC has finished the reconfiguration, a “reconfSuccess” message will be received from it and the success response will be send back to the sender of the “reconfRequest” message.

Role “Reconfiguration Slave”

The role “Reconfiguration Slave” describes the behavior of a component instance, which receives a reconfiguration request. The component instance reconfigures itself and sends back a message, that the reconfiguration has been successful.

Figure 4.14 shows an example Real-Time Statechart of the role “Reconfiguration Slave”.

![Figure 4.14: Role “Reconfiguration Slave” of Pattern “Reconfiguration Propagation”](image-url)
Pattern instantiation is an open issue of the MechatronicUML development. While pattern instantiation within a MechatronicUML system has already been considered in some publications [BDG+11], there is no concept for pattern instantiation between autonomous MechatronicUML systems, so far. We call the level on which autonomous MechatronicUML system communicate with each other the system level. Considering the case study scenario introduced in Section 1.2, the autonomous BeBots have to communicate with each other when driving in a convoy. Thus, a Real-Time Coordination Pattern has to be instantiated to provide this communication. To instantiate a Real-Time Coordination Pattern, the BeBots first have to negotiate, which one has to implement which role of the pattern. In order to do this, the BeBots need a possibility to establish contacts. In this section, we will introduce an idea to solve this problem by extending a Structured Component’s Reconfiguration Process Manager.

4.5.1 Extension of Reconfiguration Process Manager

Two autonomous MechatronicUML systems need a general possibility to contact each other. Referring to the bluetooth link management protocol [BS02], we propose that a MechatronicUML system should periodically send a broadcast signal and scan for these signals of other MechatronicUML systems. By now, we use the RPM of a Structured Component to manage the establishment of contacts with advise of the fact, that this could be done by another element in future developments. We add link ports to the RPM and the Structured Component which provide a possibility to send broadcast message events. Autonomous MechatronicUML systems contact each other via these ports using the Broadcast Link Pattern. The Broadcast Link Pattern provides a not fixed communication between the autonomous systems. It is a general pattern whose implementation is the same for all MechatronicUML components. It provides a generic way to establish contacts and it does not contain any component specific behavior. Thus, a second pattern is needed to instantiate certain Real-Time Coordination Patterns between two autonomous systems. This pattern should include component specific behavior, as it defines which Real-Time Coordination Patterns could be instantiated and under which circumstances this could be done. Especially, this pattern is responsible to negotiate which role of a Real-Time Coordination Pattern has to be implemented by which component instance and it has to invoke the required reconfigurations to implement the pattern. Therefore, we call it the “Negotiation Pattern”. Furthermore, we extended the Reconfiguration Process Manager by a negotiation port, to be able to implement to “Negotiation Pattern”. A Structured Component with an extended RPM is shown in Figure 4.15.
4.5.2 Broadcast Link Pattern

The Broadcast Link Pattern is used to establish contacts between two autonomous MECHATRONICUML systems, analogous to the concept of the bluetooth link management protocol [BS02]. It is a general pattern, which instantiates negotiation ports on the corresponding component instances. Through this, the component instances become connected using the Negotiation Pattern.

We will demonstrate the functionality of the Broadcast Link Pattern by an example, where two autonomous MECHATRONICUML systems establish contacts. At first, the two systems are only able to communicate via broadcast signals as they are not directly connected through a coordination pattern. System 1 sends a broadcast signal inquiryRequest. System 2 receives this signal and sends a broadcast signal inquiryResponse, which includes the unique id of system 2. System 1 receives this inquiryResponse signal and sends a personalized pageRequest signal, which includes the id of system 1 as “senderId” as well as the id of system 2 as “receiverId”. System 2 receives the pageRequest signal and checks, if the “receiverId” equals to its own id. If this is the case, it creates the required negotiation port instances and sends back a pageResponse signal. The pageResponse signal contains the id of system 2 as “senderId” and the id of system 1 as “receiverId”. Finally, system 1 receives the pageResponse signal, validates that it is a correct response to the pageRequest signal sent before and then creates the negotiation port instances on its part.
We assume that the assembly between the negotiation port instances is virtual, as there is no system which could hold it. Therefore, the negotiation pattern has been instantiated completely when both systems have instantiated their negotiation ports.
Figure 4.16 shows a sequence diagram of the interaction to establish contacts between two MECHATRONICUML systems.

Figure 4.16: Sequence Diagram for establishment of contacts between two autonomous systems.

Figure 4.17 shows the Broadcast Link Pattern. It consists of two identical roles “Link Participant” which have to be broadcast roles. A Link Participant sends inquiry requests and scans them at the same time. As the MECHATRONICUML language does not have a definition for broadcast roles or ports yet, we will give a first idea of how to implement them in the following. Nevertheless, this should also be a subject of further developments.
4. MechatronicUML Reconfiguration Concept

Figure 4.17: Broadcast Link Pattern for establishment of contacts between two autonomous systems.

Role Link Participant

Figure 4.18 shows a cutout of the Real-Time Statechart of the Atomic Component representing the Reconfiguration Process Manager. It consists of three regions. One region for the reconfiguration execution port, another one for the link port and a third one for the synchronization statechart. The Real-Time Statechart “Link Participant” in region “link port” represents the behavior of the role “Link Participant”.

The role “Link Participant” has to implement the behavior represented by the sequence diagram shown in Figure 4.16. As the role appears on both sides of the pattern, it has to implement a sender as well as a receiver of an inquiryRequest signal at the same time. Besides, the statechart consists of the two regions “scanning” and “sending”. The “scanning” region is needed to scan for incoming signals, whereas the “sending” region enables to send signals.

To explain the functionality of the statechart, we consider two instances of it, as it is always implemented twice. On the one hand by a component which implements the left role, on the other hand by a component which implements the right role of the Broadcast Link Pattern. To ease the explanation, we will consider one component to be the master which initiates the communication and the other one to be the slave.

At first, the region “sending” of the master sends an “InqRequest()” signal each ten time units by firing transition “t1”. If the slave role receives an “InqRequest()” signal, the transition “t2” triggers transition “t3” via the synchronization channel “SendInqResponse()” to send an “InqResponse(id)” signal to the master. In this case, the value of the parameter “id” is the id of the slave component instance. Afterwards, the master receives this signal and transition “t4” fires, which triggers a synchronization “sendPageRequest(InqResponse.senderId)” and therefore enables transition “t5” to send a “PageRequest(senderId,recId)” signal, including its own id as the sender and the id of the responding slave as the receiver.
4.5 Pattern Instantiation on System Level

Figure 4.18: Real-Time Statechart of a RPM, whose link port implements the "Broadcast Link Pattern".
The slave receives this message and fires transition “t6” to check, if the “recId” parameter equals its own id. If this is the case, a correct PageRequest has been received and transition “t7” fires. Thus, the state “Create Negotiation Port” becomes active and the synchronization “SendPageResponse(cpId)” triggers transition “t8” to fire, which causes to send a PageResponse(senderId,recId) message to the master.

The master receives this message, transition “t13” fires and if the “recId” parameter equals the id of the master, the state “Create Negotiation Port” becomes active. In this case, the master and the slave are both in the state “Create Negotiation Port” and both statecharts fire transition “t9” which sends a synchronization “CreateNegotiationPortOnRPM()” to the synchronization statechart of the RPM component instance.

Thus, transition “t10” of the synchronization statechart fires and state “Create Negotiation Port On RPM” becomes active. As the Component Story Diagram “createNegoPortOnRPM” (cf. Figure 4.19) is an entry action of this state, the Component Story Diagram will be executed and a new negotiation port instance will be added to the RPM component instance.

After this, transition “t11” fires and triggers transition “t12”, contained in the statechart of the reconfiguration execution port, to fire. This results in sending a doReconf(“negotiation”) event with parameter id=“negotiation” to the Reconfiguration Execution Controller. We assume that the id “negotiation” is assigned to the Component Story Diagram “createNegoPortOnHost” (cf. Figure 4.20) in the allocation table of the REC. As introduced in Section 4.3.2, this causes the REC to execute the Component Story Diagram “createNegoPortOnHost”, which adds a new negotiation port instance to the Structured Component instance and delegates it to the negotiation port instance of the RPM.

As these reconfiguration steps have been performed within the master component instance as well as the slave component instance, the negotiation ports of both instances have been instantiated and delegated to the corresponding RPM. Thus, we consider the Negotiation Pattern to be instantiated.

Figure 4.19 shows the Component Story Diagram “createNegoPortOnRPM”, which will be executed on a Reconfiguration Process Manager component instance to add a negotiation port.
4.5 Pattern Instantiation on System Level

Figure 4.19: Component Story Diagram “createNegoPortOnRPM”.

Figure 4.20 shows the Component Story Diagram “createNegoPortOnHost”. It used create a new negotiation port instance on a Structured Component’s instance configuration and delegate the new port instance to the negotiation port instance of the Reconfiguration Process Manager.

Figure 4.20: Component Story Diagram “createNegoPortOnHost”.

4.5.3 Negotiation Pattern

If two autonomous MECHATRONICUML systems want to communicate with each other, they have to instantiate a certain Real-Time Coordination Pattern. To do this, they first have to negotiate, which pattern should be used and furthermore, which system should implement which role of this pattern. We propose to define a Real-Time Coordination Pattern called Negotiation, which specifies this negotiation communication behavior of the systems.

Figure 4.21 shows the Negotiation Pattern with two identical roles “Negotiator”. The Negotiation Pattern is limited to a 1:1 communication between the “Negotiator” roles. In fact, it would be nice to negotiate with more than one other system at a time. Therefore, the pattern should provide a n:m communication. Unfortunately, the MECHATRONICUML language does not provide n:m communication patterns so far. So, we keep the Negotiation Pattern as 1:1 communication by now and mention, that it should be extended to n:m communication as soon as the MECHATRONICUML provides this functionality. We will show a concrete
Real-Time statechart of the role “Negotiator” in Section 5.4.

Figure 4.21: Real-Time Coordination Pattern “Negotiation”.

Figure 4.22 shows a Structured Component, including all new elements introduced by the MECHATRONICUML reconfiguration concept in this chapter.

Figure 4.22: Structured Component including new elements of the reconfiguration concept.

It contains a Reconfiguration Process Manager, which handles reconfiguration requests. In case of the Structured Component represents an autonomous system,
the RPM provides a broadcast link port, which enables to establish communication with other autonomous systems using the Broadcast Link Pattern. The negotiation port is then needed to negotiate how a certain Real-Time Coordination Pattern could be instantiated by use of the Negotiation Pattern.

If the Structured Component is embedded into another component, then it does not need a link or a negotiation port. Instead, it contains an outer-rc reconfiguration communication port to send and receive reconfiguration messages using the Reconfiguration Propagation Pattern.

Furthermore, the Structured Component contains a Reconfiguration Execution Controller, which performs reconfigurations of the Structured Component’s instance configuration by executing Component Story Diagrams. The RPM is connected to the REC by the Reconfiguration Execution Pattern. Thus, the RPM is able to order the REC to execute reconfigurations by sending reconfiguration messages.

The RPM also propagates incoming reconfiguration messages to the embedded parts, using the Reconfiguration Propagation Pattern.

Finally, the Structured Component contains a Configuration State Manager, which holds the current configuration state. This current state can be requested by the RPM via a connection of the Configuration State Pattern.
5 Reconfiguration Example

This chapter demonstrates the reconfiguration concept for MechatronicUML systems by an example. The example is based on the BeBot case study introduced in Section 1.2. It shows up the modeling of BeBots and their reconfiguration on MechatronicUML component type level as well as the corresponding behavior of their component instances at run-time.

We will first present a simplified MechatronicUML model of the BeBot in Section 5.1. After this, we will consider the reconfiguration details of the case study scenario in Section 5.2 and introduce concrete Real-Time Statecharts of the Configuration State Pattern and the Negotiation Pattern in Section 5.3 and 5.4. Finally, we will demonstrate the reconfiguration behavior of the BeBot component instances at run-time in Section 5.5.

5.1 BeBot MechatronicUML model

We will use a very simplified MechatronicUML model of the BeBot to demonstrate the reconfiguration concept. The BeBot component consists of a Reconfiguration Process Manager, a Reconfiguration Execution Manager and a Configuration State Manager as introduced in Chapter 4. As the BeBot can be considered as an autonomous system, it also includes a link and a negotiation port. Furthermore, the BeBot contains two embedded Atomic Components “LeaderControl” and “MemberControl”. The “LeaderControl” component is needed to drive as a convoy leader, as it calculates the positions of the member BeBots and sends this position data to them via the “lc” port. The “MemberControl” component is needed to drive in a convoy as a member. It receives position data from the “LeaderControl” component of the leader BeBot via the “mc” port and orders the member BeBot to drive to these positions. Figure 5.1 shows the Structured Component type “BeBot”.
5. Example Scenario Details

Let’s remember the case study scenario introduced in Section 1.2. Given several BeBots, which drive in an unknown environment. The BeBots try to build convoys, for the sake of energy saving. We consider a situation, where two BeBots already drive in a convoy and a third BeBot tries to join them. Thus, the third BeBot has to establish contacts to the leader BeBot of the convoy and negotiate, if it is feasible to join the convoy. We assume, that the BeBot is allowed to become a new member of the convoy. Therefore, the new member BeBot as well as the leader BeBot have to reconfigure their component instance configurations.

A component instance of type “BeBot” has to instantiate a component instance of type “LeaderControl” to become a convoy leader and it has to instantiate a component instance of type “MemberControl” to become a convoy member. One “lc” port instance of the leader BeBot instance must be connected to the “mc” port instance of each member instance using the “Convoy Coordination” Real-Time Coordination Pattern.

Figure 5.2 shows the three MECHATRONICUML component instances of the BeBots used in our example. The BeBot on the left side is the leader of the existing convoy. Thus, it contains a “LeaderControl” component instance to calculate position data of the convoy members. The BeBot on the bottom ("member1") is already a member of the convoy. Therefore, it includes a “ConvoyMemberCon-
5.3 Configuration State Pattern

The Configuration State Pattern (cf. Section 4.3.3) describes the communication between the Reconfiguration Process Manager and the Configuration State Manager. The RPM implements the role “State Requester”, whereas the role “State Manager” will be implemented by the REC.
5. RECONFIGURATION EXAMPLE

5.3.1 Role State Requester

The role “State Requester” describes the behavior to request the current configuration state of a system from the role “State Manager”. In our case, it is furthermore able to order the “State Manager” role to change the current state by sending an according message.

Figure 5.3 shows the Real-Time Statechart of the role “State Requester”. The “Idle” state is normally the active state. From here, a transition can be fired to send a “StateRequest” message to the “State Manager” role. This would activate the state “Wait”, which waits for a “State Response” message. Receiving this message would reactivate the “Idle” state. Furthermore, the messages “goMember”, “goLeader” and “goNoConvoy” can be sent to change the current configuration state of the component instance, which is hold in the Real-Time Statechart of the role “State Manager”.

![Real-Time Statechart of the role “State Requester”](image)

Figure 5.3: Role “StateRequester” of Real-Time Coordination Pattern “Configuration State”.

5.3.2 Role State Manager

The “State Manager” role specifies the available configuration states of a system. In our case, a BeBot can either be a convoy leader, a convoy member, or it does not belong to a convoy. Each of these situations require different configurations of the BeBot component instance. Thus, the “State Manager” role consists of the three states “Leader”, “Member” and “NoConvoy”.

Figure 5.4 shows the Real-Time Statechart of the role “State Manager”. Initially, a BeBot is not driving in a convoy, thus the state “NoConvoy” is active. From here, receiving a “goLeader” message would activate the state “Leader”, whereas a “goMember” message activates the state “Member”. A BeBot driving as a convoy leader can not directly be reconfigured to drive as a convoy member and vice versa. Thus, the “Leader” state as well as the “Member” state only have one outgoing transition to the “NoConvoy” state, triggered by the message “goNoConvoy”. Furthermore, each state reacts on a “StateRequest” message by sending a “StateResponse(string state)” message, which contains the state’s id as
5.4 Negotiation Pattern

The Negotiation Pattern (c.f. Section 4.5.3) specifies the behavior of how to instantiate a concrete Real-Time Coordination Pattern to connect two autonomous MechatronicUML systems. The pattern consists of two identical roles “Negotiator”. In our scenario, the pattern describes how to instantiate the Real-Time Coordination Pattern “Convoy Coordination”, which is needed for the communication of a convoy leader with the convoy members.

5.4.1 Role Negotiator

The role “Negotiator” negotiates, which role of the Real-Time Coordination Pattern “Convoy Coordination” a component instance has to implement. The “member” role of the “Convoy Coordination” pattern can only be implemented once by each component instance, as a BeBot can only be a member of one convoy. Thus, a BeBot driving as a convoy member does not try to instantiate the pattern. A BeBot, which is not driving in a convoy, always tries to implement the leader role first. Therefore, it asks for the current configuration state of the counterpart and if this state also equals “noConvoy”, it reconfigures itself to be a convoy leader and demands the counterpart to reconfigure itself to implement the “member” role. If the counterpart’s current state is “leader”, then the BeBot stops the negotiation process and waits for orders of the counterpart, as a leader BeBot will
always try to add new members to its convoy.

We assume, that a leader BeBot should control the negotiation. Therefore, a BeBot driving as a convoy leader always asks for the current configuration state of the counterpart and initiates a corresponding reconfiguration to add a new member to the convoy, if the counterpart BeBot is not driving in another convoy yet. Both “Negotiator” roles would invoke the negotiation process as soon as the Negotiation Pattern has been instantiated. This could result in deadlocks, as both roles wait for a response of the other role. Therefore, we also implemented a random delay to ensure, that the BeBots do not start the negotiation process at the same time.

Figure 5.5: Role “Negotiator” of Real-Time Coordination Pattern “Negotiation”.

Figure 5.5 shows the Real-Time statechart of the role “Negotiator”. Starting from state “Idle”, an “ExtStateRequest” message (which has been send by the counterpart “Negotiator” role) would trigger to send a “StateRequest” message to the Configuration State Manager to get the current configuration state. If a “StateResponse” message has been received from the CSM, an “ExtStateResponse” message, containing the current configuration state as a parameter value, will be sent to the counterpart role and the “Idle” state becomes active again.

A random variable “offset” defines the delay to start the negotiation process. Thus, if the elapsed time is greater than this offset, a transition fires and sends a “StateRequest” message to the CSM to get the current configuration state of the BeBot. Additionally, the state “GetState” becomes active. Receiving a “StateResponse” message from the CSM triggers the next transitions, depending on the value of the parameter “state”. If “state”=“member”, the “Idle” state becomes active, because there is nothing to negotiate for a Bebot driving as a convoy mem-

54
ber. In case of “state” = “leader”, the “Leader” state becomes active, whereas the “NoConvoy” state will be activated if the value of “state” equals “noConvoy”. Furthermore, the transitions, which activate the “Leader” or the “NoConvoy” state, send an “ExtStateRequest” to the counterpart BeBot. Active state “NoConvoy” triggers a transition to activate the “InitLeader” state by receiving a “ExtStateResponse” message from a counterpart BeBot, whose current configuration state equals “noConvoy”. Each other current configuration state of the counterpart causes to reactivate state “Idle”, as the counterpart BeBot already belongs to a convoy. In case of the counterpart BeBot is a convoy leader, it will initialize the instantiation of the “Convoy Coordination” pattern on its own. If it is a convoy member, then it is impossible to instantiate the pattern. Reactivating state “InitLeader” triggers a transition which sends a “CreateLeader-Control” to the Reconfiguration Execution Controller and a “goMember” message to the counterpart BeBot. Thus, the BeBot implements the “leader” role of the “Convoy Coordination” pattern by instantiating a “LeaderControl” instance and the counterpart BeBot reconfigures itself to implement the “member” role. If the “Leader” state is active, receiving a “ExtStateResponse” message triggers a transition to activate the “AddMember” state, if the current configuration state of the counterpart equals “noConvoy”. Otherwise, the “Idle” state becomes active again, because there is nothing to negotiate anymore, as only a BeBot with current configuration state “noConvoy” is able to become a new member of an existing convoy. The “AddMember” state triggers a transition which invokes a reconfiguration to create a new “LeaderControl” port instance, that implements the “leader” role of the Real-Time Coordination Pattern “Convoy Coordination”. Additionally, a “goMember” message has been sent to the counterpart BeBot to invoke a reconfiguration on it. After this, the state “Idle” becomes active. Receiving a “goMember” message while state “Idle” is active causes a reconfiguration, which instantiates a “MemberControl” component instance and according port instances to implement the “member” role of the “Convoy Coordination” pattern. This will be achieved by sending a “addMemberControl” message to the Reconfiguration Execution Controller.

5.5 Reconfiguration in Practice

We are now going to describe the reconfiguration process to add the new member BeBot to the existing convoy in detail.

At first, the new member BeBot has to establish contacts with the leader BeBot. This will be realized by use of the Broadcast Link Pattern. The functionality of the Broadcast Link Pattern has been demonstrated in Section 4.5.2, thus, we make it short at this point. The Broadcast Link Pattern creates negotiation ports on the leader BeBot component instance as well as the new member BeBot component instance. Thus, it instantiates the Negotiation Pattern, which enables the two
Bebots to negotiate how to instantiate the “Convoy Coordination” pattern. Figure 5.6 shows the component instances after instantiating the Negotiation Pattern.

In the next step, the BeBots negotiate, which one has to implement which role of the “Convoy Coordination” Pattern. In this case, the assignment of roles is clear, as there is already a leader BeBot. Thus, related to the Real-Time State-chart of the role “Negotiator” (cf. Section 5.4.1), the leader BeBot reconfigures itself by adding a new port instance “lc2” to its “LeaderControl” instance. This will be done by executing the Component Story Diagram shown in Figure 5.7 on the “LeaderControl” instance. Additionally, it has to delegate this port instance to a new port instance “lc2” of the BeBot component instance. Executing the Component Story Diagram “createLeaderPortOnHost” (cf. Figure 5.8) on the leader BeBot component instance realizes this.

Figure 5.6: BeBot instances reconfiguration scenario step 1.

Figure 5.7: Component Story Diagram “createLeaderPort”. 
Figure 5.8: Component Story Diagram “createLeaderPortOnHost”.

Figure 5.12 shows the resulting component instances, after reconfiguring the leader BeBot component instance. The Broadcast Link Pattern connection has been cut, after the “Negotiation” pattern connection has been instantiated.

Figure 5.9: BeBot instances reconfiguration scenario step 2.

After this, the leader BeBot sends a “goMember” message to the new member BeBot via the negotiation port. This causes the new member BeBot to reconfigure itself to become a convoy member. Therefore, it has to instantiate a new “MemberControl” instance. Furthermore, it has to create a new port instance “mc1” on the BeBot component instance and delegate it to the “mc1” port instance of the “MemberControl” instance. Additionally, an “inner-rc1” port must be added to
the RPM component instance to connect it with the outer-rc port instance of the “MemberControl” instance. This has to be done in two steps. At first, we execute the Component Story Pattern “createReconfPortOnRPM” (cf. Figure 5.10) on the RPM component instance. Thus, the new inner-rc port instance of the RPM component instance has been created.

![Figure 5.10: Component Story Diagram “createReconfPortOnRPM”.](image)

Then, we are able to execute the Component Story Diagram “createMemberControl” (cf. Figure 5.11) on the BeBot component instance. This Component Story Diagram creates a new “MemberControl” instance “mc” and a new port instance “mc1” on the BeBot component instance. It furthermore delegates the “mc1” port instance to the one of the “MemberControl” instance and creates an assembly instance to connect the inner-rc port instance of the RPM instance with the outer-rc port instance of the “MemberControl” instance.

![Figure 5.11: Component Story Diagram “createMemberControl”.](image)
Now, the “Convoy Coordination” pattern has been instantiated, as the port instance “lc2” of the leader BeBot component instance implements the role “leader” and the port instance “mc1” of the new member BeBot component instance implements the role “member”. Figure 5.11 shows the BeBot component instances after executing the reconfiguration. The leader BeBot and the new member BeBot are now able to communicate with each other via a connection of the Real-Time Coordination Pattern “Convoy Coordination”.

Figure 5.12: BeBot instances reconfiguration scenario step 3.
6 Implementation

The concepts of MECHATRONICUML are implemented within the Fujaba4Eclipse Realtime Toolsuite, based on the Eclipse Modeling Framework (EMF). In this chapter, we will describe the extensions of the MECHATRONICUML implementation, which are necessary to provide the functionalities of the reconfiguration concept.

First, we have to think about how the new elements, introduced by the reconfiguration concept, could be represented by the existing implementation. The Reconfiguration Process Manager can be modeled by an Atomic Component. Also the Configuration State Manager represents nothing else than an Atomic Component, by now. Thus, we do not need to extend the current implementation to model these elements.

However, the Reconfiguration Execution Controller is a special element, which cannot be represented by any class of the existing implementation, because it provides specific reconfiguration features. Therefore, we add a new class called “ReconfigurationExecutionController” to the MECHATRONICUML meta-model. Furthermore, a new class for a reconfiguration port is needed, as it is visualized in a different way than other ports (cf. Section 4.3.1). For this reason, we add one more class to the MECHATRONICUML meta-model called “ReconfigurationPort”.

![Figure 6.1: Cutout of the meta model for MECHATRONICUML component type level, extended by new elements of the reconfiguration concept.](image)

The MECHATRONICUML implementation distinguishes between a meta-model for component types and one for component instances. Figure 6.1 shows a cutout
of the meta-model for component types, extended by the new classes “ReconfigurationExecutionController” and “ReconfigurationPort”.

The new class “ReconfigurationExecutionController” contains an attribute “MessageIdToCSD” of type EMap, which represents the allocation table of reconfiguration message ids to Component Story Diagrams. A REC always belongs to a Structured Component and it may not exist without it. Thus, a containment relationship connects the class “StructuredComponent” and the class “ReconfigurationExecutionController”, with a cardinality of 0..1, as a Structured Component may contain a REC, but it must not.

The new class “ReconfigurationPort” inherits from the existing class “Port”. Additionally, it has an association to the class “ReconfigurationExecutionController”, which defines, that a REC always contains exactly one reconfiguration port.

Figure 6.2 shows a cutout of the meta-model for component instances. Analogically to the meta-model for component types, it has been extended by the new classes “ReconfigurationExecutionControllerInstance” and “ReconfigurationPortInstance”. Furthermore, the class “ReconfigurationExecutionControllerInstance” has an operation called “ExecuteComponentStoryDiagram()”, which enables it to execute Component Story Diagrams.

Figure 6.2: Cutout of the meta model for MECHATRONICUML component instance level, extended by new elements of reconfiguration concept.
Modern technical systems must often cope with changes in their environment. In the majority of cases, these changes demand the system to adapt its behavior. Thus, the system has to be flexible during run-time. A powerful possibility of a component-based system to provide this flexibility is to perform a dynamic reconfiguration of its structure.

In this work, we introduced a reconfiguration concept for a component-based modeling language for mechatronic systems, called MECHATRONICUML. The reconfiguration concept provides modeling of reconfiguration behavior on design time, as well as executing dynamic reconfigurations at run-time. We used the concept of Component Story Diagrams to model reconfiguration behavior on a high level. Therefore, reconfigurations are performed by executing Component Story Diagrams on component instance configurations.

We extended the Structured Components of the MECHATRONICUML by a Reconfiguration Execution Controller, which stores Component Story Diagrams to reconfigure a Structured Component instance and is additionally able to execute them on the corresponding component instance configuration. We introduced a reconfiguration message event flow, which is used to handle a reconfiguration process within a hierarchical component model. We furthermore added a Reconfiguration Process Manager to each Structured Component, which controls the reconfiguration process of the component instance and holds the reconfiguration logic. Available configuration states of a Structured Component can be defined within a new element called Configuration State Manager.

Secondly, we gave an idea of how to solve the problem of pattern instantiation on system level. For this reason, we introduced the Broadcast Link Pattern to establish contacts between autonomous MECHATRONICUML systems. In addition, we developed a Negotiation Pattern, which enables autonomous systems to instantiate Real-Time Communication Patterns on system level.

Furthermore, we demonstrated the feasibility of the reconfiguration concept by an example. Finally, we extended the meta-model of the MECHATRONICUML implementation within the Fujaba4Eclipse Realtime Toolsuite to provide the functionalities of the reconfiguration concept.
7. Conclusions

7.1 Future Prospects

The reconfiguration concept introduced in this work is a practical solution to model and execute reconfigurations of mechatronic systems using the MECHA-TRONICUML language. However, the concept underlies some limitations, which are worth to be discussed in future developments.

One important future research subject is to extend the reconfiguration concept by safety issues. As reconfigurations are executed at run-time, it must be guaranteed that they will not break down the target system. For example, a system has to verify that a planned reconfiguration is allowed to be executed, before applying the changes to its configuration. This could be achieved by satisfying a set of consistency criterion, in particular transactional integrity (atomicity, consistency of the final state, isolation) and termination of the reconfigurations [DL06].

Currently, the developer is responsible to build a consistent MECHATRONICUML system. This is sub-optimal and should be enhanced in future works, because it is very fault-prone and unsafe. Especially, there is no possibility to roll back a reconfiguration if it runs into an error while executing. This case would break down the system. Furthermore, it should not be allowed to reconfigure a component instance, if it currently performs an action which must not be aborted abruptly. A component instance has to be in a so called quiescent state [KM98] to be reconfigurable. A MECHATRONICUML system should automatically check this before it reconfigures an embedded component instance. The quiescent state approach has already been improved by some developments which introduce version consistency [MBG^11] as a new criterion for safe dynamic reconfigurations. Version consistency, under certain circumstances, allows to reconfigure a component instance which is not in a quiescent state. Thus, it seems to be a promising approach as it could reduce the execution time of a reconfiguration process, compared to the quiescent state approach.

The current reconfiguration concept does not consider any time consumption of reconfiguration actions. It assumes that the reconfiguration actions perform in real-time. This is not very realistic and should be improved in the future, for example by adding a timing behavior to the Real-Time Statecharts of the Real-Time Coordination Patterns, which are used for reconfiguration purposes.

We modeled the available reconfiguration states of a Structured Component by a Configuration State Manager, which is a simple Atomic Component, so far. Thus, the reconfiguration state logic is hold in a statechart of the CSM’s port. This is maybe not sufficient, if the component’s configuration state logic is more complex. Furthermore, it would be helpful to identify only a certain part of a configuration state, which is relevant for a currently performing reconfiguration. This would speed up the configuration state request. Therefore, the CSM could be represented by a more complex element than an Atomic Component, for example,
Moreover, it could be needful to extend a component instance by an element which monitors its functionality and automatically triggers a reconfiguration in case of a failure. For example, a Structured Component instance permanently observes its part instances and is able to automatically replace one, if a part instance doesn’t run correctly.

Pattern instantiation on system level is another issue for future developments. We already presented an idea of how to solve this problem by use of a Broadcast Link Pattern and a Negotiation Pattern in Section 4.5. This idea could be used to develop a general solution for establishing contacts between autonomous systems, modeled by the MECHATRONICUML. To do this, the contact establishing functionality should be transfered from the Reconfiguration Process Manager to a new element, which has to be developed. This new element should ensure, that it only responds to inquiry requests of systems which are not yet connected through a Negotiation Pattern. Otherwise, an ongoing re-initialization of the Negotiation Pattern connection would take place. It would also be interesting to check up the possibility to model check a Broadcast Communication Pattern. One idea to provide this is to develop an environment model, providing a broadcast channel for each autonomous system located within the environment.

At least, as considered in Section 4.5.3, the MECHATRONICUML language should be extended by n:m Real-Time Coordination Patterns, which allow modeling of a multi-role to multi-role communication on component type level. This would permit a communication between more than two autonomous systems at the same time. For Example, this would be very needful if three or more BeBots meet each other and have to negotiate, e.g. by Leader Election, which one has to implement which role of a Communication Pattern to build up a convoy.
References


References


References


