A Catalog of Real-Time Coordination Patterns for Advanced Mechatronic Systems

Technical Report
tr-ri-12-319

Stefan Dziwok*, Kathrin Bröker†, Christian Heinzemann*, and Matthias Tichy‡

* Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Germany
[stefan.dziwok|christian.heinzemann]@mail.uni-paderborn.de

† Computer Science Education, University of Paderborn, Germany
kathyb@mail.uni-paderborn.de

‡ Software Engineering Division, Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden
tichy@chalmers.se

Version: 1.0

Paderborn, February, 2012

1This work was developed in the project ‘ENTIME: Entwurfstechnik Intelligente Mechatronik’ (Design Methods for Intelligent Mechatronic Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, ‘Investing in your future’. This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft. Christian Heinzemann is supported by the International Graduate School Dynamic Intelligent Systems.
Contents

1 Introduction 1

2 MECHATRONIC UML 3

3 Patterns for Real-Time Coordination Protocols 6
   3.1 Real-Time Coordination Patterns .......................... 6
   3.2 Description Format of our Patterns ......................... 8

4 The Catalog of Real-Time Coordination Patterns 10
   4.1 Synchronized Collaboration ................................ 10
   4.2 Fail-Operational Delegation ............................... 14
   4.3 Fail-Safe Delegation ........................................ 17
   4.4 Master-Slave-Assignment .................................. 21
   4.5 Periodic Transmission ....................................... 25
   4.6 Producer-Consumer .......................................... 28
   4.7 Block Execution ............................................. 31
   4.8 Limit Observation ........................................... 34

5 Developing Advanced Mechatronic Systems using Real-Time Coordination Patterns 37
   5.1 Application-specific Adaptation ............................ 37
   5.2 Implementation-specific Refinement ......................... 38

6 Case Study: Cooperating Robots 40

7 Related Work 42

8 Conclusions and Future Work 43

Bibliography 44

List of Figures 47
Chapter 1

Introduction

Mechanical engineering has a long tradition in sustained development of innovation, e.g., innovation in cars in the last century. However, in the last decades, software is the driving force for innovation in mechanical engineering as, e.g., in the automotive domain [SHS+05]. Modern mechanical systems are developed by experts from several engineering disciplines: mechanical engineering, electrical engineering, control engineering, and software engineering. These systems are called mechatronic systems. Mechatronic systems often operate in a safety-critical context, i.e., failures can lead to death or serious injury to people.

Furthermore, previously isolated systems increasingly form systems of systems where autonomous systems communicate with each other by means of complex message exchange protocols [SW07] in an ad-hoc manner. This results in very complex systems.

These trends make the development of advanced mechatronic systems a big challenge. Thus, appropriate development approaches have to be utilized and rigorously followed. Particularly, the software has to be subject of rigorous verification and validation activities as well as appropriate development processes and languages.

Figure 1.1 shows three advanced mechatronic systems developed at the University of Paderborn in the last couple of years. On the left, two autonomous shuttles of the RailCab systems are shown. RailCab shuttles are autonomous railway vehicles which combine the flexibility of individual transport with the energy efficiency of public transport systems. They save energy by forming convoys which reduce the air resistance. In the middle, two miniature robots called BeBots are shown. BeBots form ad-hoc networks in order to jointly execute tasks. The robots can collectively agree on taking different roles to achieve the common task. On the right, two cooperating robots are shown which play ping-pong. They do so without any external global camera system but instead rely on the timely exchange of position, velocity, and trajectory of the batted ball.

In all three advanced mechatronic systems, coordination plays an important role, because they consist of independent, communicating actors (e.g., autonomous mechatronic systems), who join their efforts towards mutually defined goals (cf. [Nat89]). For example, the communication actors decide on a common strategy (e.g., activating the convoy) or they decide on a master who delegates tasks to the slaves. These coordination aspects require sophisticated coordination protocols.
We developed the coordination protocols for these systems based on the patterns presented in [GHJV95, BMR96, Dou99, Dou02] in order to exploit the vast amount of existing experience. The patterns listed in these approaches proved to be very helpful in developing our systems. However, they have the drawback that the patterns are only informally described which may lead to the introduction of errors when they are applied to new systems. As we focus on safety-critical mechatronic systems, a pattern approach which avoids this introduction of errors in the first place is beneficial in order to guarantee the safety of the system.

Based on that experience, we developed Real-Time Coordination Patterns which formalize coordination protocols for mechatronic systems with a particular focus on safety properties and hard real-time constraints. Furthermore, protocols that are based on these patterns enable to decompose the mechatronic system in such a way that a scalable formal verification using model checking can be employed. This is possible because of our previous work on compositional verification [GTB03]. In contrast to the pattern formalism of [GTB+03], we further abstracted from application-specific details for a better reusability, defined a description format for the patterns and built up a catalog of patterns.

In summary, the contribution of this technical report is as follows: (1) we present Real-Time Coordination Patterns as formal representation of reusable coordination protocols, (2) we present the current catalog of our patterns, (3) we present formal refinement steps which define how these patterns are applied and refined, and (4) we report on a case study in which the approach was applied to the aforementioned cooperating robots example.

Chapter 2 presents MECHATRONIC UML, which is the foundation for our approach. In Chapter 3, we introduce Real-Time Coordination Patterns that are patterns for Real-Time Coordination Protocols. We present the current catalog of our patterns in Chapter 4. Then, we show how these patterns are applied to new systems in Chapter 5. Thereafter, we present the cooperating robots case study in Chapter 6. Next, we distinguish our results from related work in Chapter 7. Finally, we conclude with an outlook on future work in Chapter 8.
Chapter 2

**MECHATRONICUML**

MECHATRONICUML [BBD+12] is a language for the model-driven design of software of advanced mechatronic systems. It follows the component-based approach where each component encapsulates a part of the software. In advanced mechatronic systems, the components that constitute the software do not work in isolation, but they have to coordinate their actions using communication for achieving the intended functionality of the system. Therefore, each component defines a set of external interaction points which we call ports. Then, components can communicate via their ports if a connector connects them.

A connection between two components implies that they are able to communicate correctly. The formal requirements for a communication are captured by a protocol definition. The protocol definition formally defines the message exchange and the time constraints that the message exchange needs to adhere to. In MECHATRONICUML, a protocol is defined by a pair of communicating roles and a connector. We call it a Real-Time Coordination Protocol. We describe the behavior of each role with a Real-Time Statechart.

Real-Time Statecharts are an extension of timed automata [BY03] to support, e.g., modeling of worst-case execution times and deadlines for actions. Real-Time Statecharts especially support the specification of asynchronous messages as well as real-time constraints. In addition, Real-Time Statecharts may define variables and operations that are required for the communication.

Figure 2.1 shows an example of a Real-Time Coordination Protocol named Convoy Coordination which is used for coordinating a convoy of RailCabs. The behavior is as follows: Initially, both Real-Time Statecharts are in the states NoConvoy/Default. Then, the rear RailCab, i.e., the RailCab driving behind, may switch to state Waiting by sending an asynchronous message convoyProposal to the front RailCab to initiate the convoy build-up. The front RailCab receives this message and switches to EvaluateProposal. In this state, the front RailCab decides whether a convoy is useful or not. The decision depends, among others, on how long both RailCabs share the same route. For deciding on this, the rear RailCab sends the ID of its destination as a parameter of the convoyProposal. Within 599ms, either the front RailCab rejects the proposal by sending convoyProposalRejected or it accepts by sending startConvoy. In the first case, both
Real-Time Statecharts return to the Default states. In the second case, both Real-Time Statecharts switch to state convoy. For avoiding a deadlock in the rear RailCab, it specifies a time out in state Waiting which causes it to return to Default after 1000ms. The transition, however, has lowest priority (indicated by 1) such that a message will be considered if it has been received. While being in state Convoy, the rear RailCab may propose to break the convoy by sending breakConvoy which causes both to return to the NoConvoy/Default state. The transition from Convoy to EvaluateProposal is needed to prevent deadlocks in case of message loss.

The protocol definition in MECHATRONICUML explicitly considers that a transmission of a message from the sender to the receiver takes time. Therefore, the connectors may receive a transmission delay. In our example, we assume a transmission delay of up to 200ms.

In many cases, the communication of the components is safety critical, i.e., a malfunctioning communication may cause severe damage to property or human lives. In case of the convoy coordination, e.g., RailCabs may come into collision if the RailCab driving behind assumes to be in convoy mode while the RailCab driving in front does not. In convoy mode, the RailCab driving in front must notify its follower before braking. If the RailCab driving in front is not in convoy mode, it will not send the notification. Thus, we must ensure that the RailCab driving behind, i.e., the rear role of the protocol, only enters the state Convoy if the front role is in state Convoy as well. We formalize such properties using the Timed Computation Tree Logic (TCTL) [ACD93]. Thus, the aforementioned property is formalized as $AG\, rear.Convoy \implies front.Convoy$.

Then, the properties are formally verified using model checking, e.g., with UPPAAL [BDL04]. In our verification, we explicitly consider the delay of the connector as well as the case that messages may be lost, e.g., when using an unreliable trans-
mission medium. In addition, we assume that messages are not reordered during the transmission and that they are stored in a FIFO-queue allowing only access to the first element. Then, the protocol introduced in this section remains safe w.r.t. the specified property and free of deadlocks.

The behavior of a component constitutes from the Real-Time Statecharts of the ports. In addition, the component may provide additional internal behavior, e.g., for resolving conflicts between different ports or for providing additional operations as discussed in Section 5.

In most cases, the system under construction cannot be verified as a whole using model checking because of the state space explosion problem (the number of reachable states is often exponential in the size of the specification). To achieve a scalable formal verification, we use the compositional verification approach of [GTB+03]. Here, we verify each Real-Time Coordination Protocol separately before verifying each component. This is enabled by a clear separation of internal behavior of a component and communication behavior using Real-Time Coordination Protocols.
Chapter 3

Patterns for Real-Time Coordination Protocols

In our modeling language MechatronicUML, connectors are first class entities. Therefore, the focus within the first process steps is to design the coordination and communication behavior. Based on given requirements, the developer has to specify one Real-Time Coordination Protocol per connector. While doing this, he has to ensure that safety-critical situations must not occur despite message delay and the possibility of message loss. We enable the developer to identify errors using model checkers like UPPAAL, but faultlessness is hard to achieve and does not correlate with a well-designed protocol. Thus, specifying a Real-Time Coordination Protocol is very complex and thus, very time-consuming and error-prone. Moreover, an already existing protocol is hard to understand.

While modeling the software for different advanced mechatronic systems using MechatronicUML, we identified that the coordination is based on recurring use-cases. This applies for the coordination between autonomous systems, but also for the coordination between components within one system. Therefore, we define general, reusable solutions for these recurring use cases to support the developer and to increase the quality of the resulting Real-Time Coordination Protocols as well as the efficiency of their development. We defined a certain kind of pattern for Real-Time Coordination Protocols, which we call Real-Time Coordination Patterns, to achieve these goals.

3.1 Real-Time Coordination Patterns

In general, a pattern within software design “provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context” [BMR+96].

A Real-Time Coordination Pattern describes a well-proven, reusable, and formal solution to a commonly occurring coordination problem within the domain of advanced mechatronic systems. These systems communicate under hard real-time constraints in a safety-critical environment. Hence, Real-Time Coordination Patterns are a special
kind of software patterns and support inexperienced developers in specifying Real-Time Coordination Protocols. A Real-Time Coordination Pattern is defined such that it respects certain safety properties, which can be formally verified using model checkers. Moreover, if a developer defines coordination protocols based on our patterns, the system under construction can be fully verified based on our compositional verification approach [GTB+03].

A Real-Time Coordination Pattern abstracts from application-specific details to be reusable in different scenarios, e.g., time parameters are defined instead of concrete time values. However, the correctness of a protocol depends on real-time constraints and properties of the connector (e.g., reliability) and is therefore not automatically correct for all possible time parameters and all connectors. Thus, we define the steps a developer has to execute for each pattern to get a correct protocol (see Section 5).

The Real-Time Coordination Patterns that we identified are described in the next Chapter. Currently, we have eight patterns. A brief overview of those is as follows:

**Synchronized Collaboration** synchronizes the activation and deactivation of a collaboration of two roles. The pattern assumes that a safety-critical situation appears if the role, which initialized the activation, is in collaboration mode and the other role is not in collaboration mode. Therefore, the pattern ensures that this situation never happens.

**Fail-Safe Delegation** realizes a delegation of a task from a role master to a role slave. The slave executes the task in a certain time and answers regarding success or failure. If the execution fails, no other task may be delegated until the master ensures that the failure has been corrected. Moreover, only one delegation at a time is allowed.

**Fail-Operational Delegation** realizes a delegation of a task from a role master to a role slave. The slave executes the task in a certain time and answers regarding success or failure. The pattern assumes that a failure is not safety-critical, though only one delegation at a time is allowed.

**Master-Slave-Assignment** is used if two systems can dynamically change between one state in which they have equal rights and another state in which one is the master and the other one is the slave.

**Periodic Transmission** can be used to periodically transmit information from a sender to a receiver. If the receiver does not get the information within a certain time, a specified behavior must be activated to prevent a safety-critical situation.

**Producer-Consumer** is used when two roles shall access a safety-critical section alternately, e.g., one produces goods, the other consumes them. The pattern guarantees that only one is in the critical section at the same time.
Block Execution coordinates a blocking of actions, e.g., due to safety-critical reasons.

Limit Observation is used to communicate if a certain value violates a defined limit or not.

3.2 Description Format of our Patterns

For describing our patterns, we defined a uniform description format such that a developer can understand, compare, and use our patterns more easily.

Several popular description formats for software patterns already exist [BMR+96, GHJV95]. We analyzed how well they fit for our patterns. We concluded that no description format matched exactly with the contents we want to provide. Therefore, we decided to choose the format of Buschmann et al. [BMR+96] and adapt it to our needs.

We use all aspects of their description format except of implementation and example resolved, because we propose to use code generators and we resolve our example already within the other aspects. Furthermore, we divide the aspect see also into the two aspects alternative patterns and combinability, because these are two different contents, which are easier to find and understand for the developer, if they are separated from each other. At last, we add the aspect verification properties to list and explain the verification properties, which must hold for the pattern.

To conclude, each Real-Time Coordination Pattern is described with the following aspects:

Name The name of the pattern and synonyms of the name, if they are known. Moreover, a brief description of the pattern.

Context The context this pattern can be used in.

Problem The problems and design issues this pattern addresses.

Solution A short description about the context the pattern can be used and the design issues and problems that are solved by the pattern.

Structure Describes the structure of the pattern and the roles, which belongs to the pattern. In addition the message interfaces of each role are explained to describe which messages each role can send and receive.

Behavior Describes the run-time behavior of the pattern. It is formally described as a Real-Time Statechart and informally with a textual description.
3.2. DESCRIPTION FORMAT OF OUR PATTERNS

Verification Properties  Each pattern must fulfill a certain list of safety and liveness properties, which can be verified using verification tools, e.g., model checkers.

Consequences  Which consequences (benefits and liabilities) arise if this pattern is used?

Examples  Examples illustrate in which applications this Real-Time Coordination Pattern is useful.

Refinement Possibilities  Here we describe possibilities for the refinement of the pattern, may be if you want to use it in another context.

Variants  For many patterns you have different variants regarding the structure and the behavior.

Alternative Patterns  Which patterns are related to this pattern and how do they differ from each other? What are advantages and disadvantages?

Combinability  Patterns can be combined with each other to fulfill a complex use case. Patterns that can be combined with this pattern are explained here.
Chapter 4

The Catalog of Real-Time Coordination Patterns

In this chapter, we will introduce the eight Real-Time Coordination Patterns which we have defined so far. Hereby, the descriptions respects the defined description format of Section 3.2.

4.1 Synchronized Collaboration

**Name**  Synchronized Collaboration (also known as: Strategy Coordination)

This pattern synchronizes the activation and deactivation of a collaboration of two systems. The pattern assumes that a safety-critical situation appears if the system, which initialized the activation, is in collaboration mode and the other system is not in collaboration mode. Therefore, the pattern ensures that this situation never happens.

**Context** Two independent systems can collaborate in a safety-critical environment, though cooperation adds more hazards.

**Problem** If one system believes they are working together, but the other one does not know this, this may create a safety-critical situation for the first system. This must be avoided. This problem occurs, if the communication is asynchronous or the communication channel may be unreliable.

**Solution** Define a coordination protocol that enables to activate and deactivate the collaboration while it considers the given problems. The systems should act with different roles: One is the master and the other is the slave. The system where the aforementioned safety-critical situation appears must be the master. The master is the one that initiates the activation and the deactivation. The activation should be a proposal so that the slave can decide if the collaboration is possible and useful. The deactivation should be a direct command, because the master can deactivate the collaboration as soon as it is no longer useful.
**4.1. SYNCHRONIZED COLLABORATION**

**Structure** The pattern consists of the two roles master and slave and a connector (cf. Fig. 4.1(a)). Both roles are in/out roles. Which message each role can receive and send is shown in the message interfaces (cf. Fig. 4.1(b)). The master may send the messages activationProposal and deactivation to the slave. The slave may send the messages activationAccepted and activationRejected to the master. The time parameter of the role master is $\text{timeout}$, the time parameter of role slave is $\text{eval-time}$. The connector may lose messages. The delay for sending a message is defined by the time parameters $\text{delay-min}$ and $\text{delay-max}$.

![Structure of Synchronized Collaboration](image)

(a) Structure of Synchronized Collaboration

![Interfaces of Synchronized Collaboration](image)

(b) Interfaces of Synchronized Collaboration

**Figure 4.1: Structure and Interfaces of Synchronized Collaboration**

**Behavior** The behavior is shown in Fig. 4.2.

First, the collaboration is in both roles inactive. The slave is passive and has to wait for the master that he decides to send a proposal for activating the collaboration. If this is the case, the slave has a certain time to answer if he accepts or rejects the proposal. If the slave rejects, the collaboration will remain inactive. If the slave accepts, he activates the collaboration and informs the master so that he also activates the collaboration. If the master receives no answer in a certain time (e.g. because the answer of the slave got lost), he cancels its waiting and may send a new proposal. Only the master can decide to deactivate the collaboration. He informs the slave so that he also deactivates it.

**Verification Properties** There will never be a deadlock within the protocol: $\text{AG not deadlock}$. If the master is in state CollaborationActive, then the slave must always be in state CollaborationActive: $\text{AG master.CollaborationActive implies slave.CollaborationActive}$. A possible assignment of the time variables is as follows:
master

slave

Figure 4.2: Behavior of Synchronized Collaboration

- master.timeout: 100
- slave.eval-time: 600
- connector.delay-min: 50
- connector.delay-max: 199

Consequences Both roles must have a pre-defined behavior when the collaboration is active or inactive. At run-time, the behavior must be adapted accordingly, because master and slave decide on this defined behavior to activate or deactivate the collaboration. Moreover, the slave cannot deactivate the collaboration.

Examples Two RailCabs are driving on the same track. The rear RailCab wants to create a convoy to take advantage of the slipstream. However, it has to drive with a small gap to the front RailCab. Therefore, the rear RailCab cannot avoid a collision, if the front RailCab brakes hard without informing the rear RailCab. Synchronized Collaboration enables to build a secure convoy if the rear RailCab acts as the master and the front...
RailCab acts as the slave and the rear RailCab only drives with a small gap as long as the convoy collaboration is active.

**Variants** The slave may also send a deactivation message. Though, this requires a reply of the master, otherwise the verification property does not hold.

**Alternative Patterns** Choose *Master-Slave-Assignment*, if you want dynamically defines a special collaboration for roles with equal rights.

**Combinability** Combine it with *Master-Slave-Assignment*, if you want to dynamically define who should be master and slave.
4.2 Fail-Operational Delegation

**Name**  Fail-Operational Delegation (initially defined by [BDG+11])

This pattern realizes a delegation of a task from a role master to a role slave. The slave executes the task in a certain time and answers regarding success or failure. The pattern assumes that a failure is not safety-critical, though only one delegation at a time is allowed.

**Context**  Delegate tasks between communicating actors.

**Problem**  If the communication is asynchronous and the communication channel is unreliable, the role that sends the task, does not know if the other role has received it. Though, the task has to be done.

**Solution**  Define a coordination protocol that enables a role master to delegate tasks to a slave. A failed task execution does not need to be handled before a new task can be delegated. The master delegates the task and wait for its completion. After a specified time, the master cancels the waiting. The slave executes this task in a certain time and reports if the task was done successfully or if the execution failed.

![Structure of Fail-Operational Delegation](image1)

![Interfaces of Fail-Operational Delegation](image2)

Figure 4.3: Structure and Interfaces of *Fail-Operational Delegation*

**Structure**  The pattern consists of the two roles master and slave (cf. Fig. 4.3(a)). Both roles are in/out roles.
Which message each role can receive and send is shown in the message interfaces (cf. Fig. 4.3(b)). The master may send the message order to the slave. The slave may send the messages done and fail to the master.

The time parameter of the role master is $timeout$, the time parameter of role slave is $worktime$. The connector may lose messages. The delay for sending a message is defined by the time parameters $delay-min$ and $delay-max$.

**Figure 4.4: Behavior of Fail-Operational Delegation**

**Behavior** The behavior is shown in Fig. 4.4.

The role master consists of the initial state Inactive and the state Waiting. From state Inactive, the message order() can be send to the slave and the state changes to Waiting. Upon the activation of Waiting the clock c0 is reset via an entry-action. An invariant using c0 ensures that Waiting is left not later than $timeout$ units of time after its activation. There are three outgoing transitions from which the one with the highest priority is triggered by the message done and leads to Inactive. The message fail triggers the other transition and leads also to Inactive. If there is a timeout, the state changes also back to Inactive.

The role slave represents the counter-part to the master role and consist of the initial state Inactive and the state Working. The message order() triggers the transition from Inactive to Working. Upon the activation of Working the clock c0 is reset via an entry-action. An invariant using c0 ensures that Working is left not later than $worktime$ units
of time after its activation. There are two outgoing transitions. The one with the highest priority sends the message `done()` to the master and the state changes back to `Inactive`. If an error occurs, the message `fail()` will be send to the master and the state changes also back to `Inactive`, too.

**Verification Properties** There will never be a deadlock within the protocol: $AG \neg \text{deadlock}$.

A possible assignment of the time variables is as follows:

- `master.timeout`: 50
- `slave.worktime`: 40
- `connector.delay-min`: 2
- `connector.delay-max`: 5

**Consequences** Only one order can be delegated at the same time. If the slave could be handle several tasks in parallel, the pattern should be adjusted.

If messages get lost, the master has to wait until the slave must be finished. Therefore, this pattern should only be used if the probability of message loss is low.

If the execution of an order failed, the next order can be delegated immediately. Therefore, this pattern should only be used, when a failed order does not produce an unsafe state that must be repaired before the next order can be delegated.

**Examples** You can find an example for this pattern in the Bebot-System. Here the pattern is used to delegate the task of checking the validity of a target position.

**Variants** You can extend the number of masters so that you have multiple masters that delegate an order to a slave. Moreover, you can extend the number of slaves so that a master can delegate to multiple slaves. Hereby, it is possible that all slaves have to do the same order, or every order is assigned to only one slave.

If the master is able to recognize that a failure in the system occurred and the slave must stop its work to prevent a safety-critical situation, then the master should be able to send a message to cancel the work of the slave.

**Alternative Patterns** Use `Fail-Safe-Delegation`, if a failure leads to a safety-critical situation that must be corrected first before a new task can be delegated.

**Combinability** Combine it with `Master-Slave-Assignment`, if you want to dynamically define who should be master and slave.
4.3 Fail-Safe Delegation

**Name**  Fail-Safe Delegation (initially defined by [Rie11])

This pattern realizes a delegation of a task from a role master to a role slave. The slave executes the task in a certain time and answers regarding success or failure. If the execution fails, no other task may be delegated until the master ensures that the failure has been corrected. Moreover, only one delegation at a time is allowed.

**Context**  Delegate tasks between communicating actors.

**Problem**  If the communication is asynchronous and the communication channel is unreliable, the role that sends the task, does not know if the other role has received it. Though, the task has to be done.

**Solution**  Define a coordination protocol that enables a role master to delegate tasks to a slave. A failed task execution is handled before a new task can be delegated. The master delegates the task and wait for its completion. After a specified time, the master cancels the waiting. The slave executes this task in a certain time and reports if the task was done successfully or if the execution failed. If it failed, the slave does not execute new tasks until the master sends the signal that the error is resolved.

![Structure of Fail-Safe Delegation](image)

(a) Structure of *Fail-Safe Delegation*

<table>
<thead>
<tr>
<th>fs-delegation_master-to-slave</th>
<th>fs-delegation_slave-to-master</th>
</tr>
</thead>
<tbody>
<tr>
<td>order()</td>
<td>done()</td>
</tr>
<tr>
<td>continue()</td>
<td>fail()</td>
</tr>
</tbody>
</table>

(b) Interfaces of *Fail-Safe Delegation*

Figure 4.5: Structure and Interfaces of *Fail-Safe Delegation*

**Structure**  The pattern consists of the two roles master and slave (cf. Fig. 4.5(a)). Both roles are in/out roles.
Which message each role can receive and send is shown in the message interfaces (cf. Fig. 4.5(b)). The master may send the messages order and continue the slave. The slave may send the messages done and fail to the master.

The time parameter of the role master is $\text{timeout}$, the time parameter of role slave is $\text{worktime}$. The connector may lose messages. The delay for sending a message is defined by the time parameters $\text{delay-min}$ and $\text{delay-max}$.

![Behavior Diagram](image)

**Figure 4.6: Behavior of Fail-Safe Delegation**

**Behavior** The behavior is shown in Fig. 4.6.

The role master has the initial state $\text{Idle}$. From this state the master can send the message order() to the slave and the state changes to $\text{Waiting}$. An entry-action in this state resets the clock $c0$. If the clock $c0$ reaches the value of $\text{timeout}$, the master assumes that the order or the answer message got lost or that the slave has fallen out. Then, the state will leave to $\text{Idle}$. If the master receives the message fail() the state will change to $\text{FailSafe}$. If the master receives the message done() the state changes back to $\text{Idle}$. When the master receives the message fail(), it changes to state $\text{FailSafe}$. The pattern assumes that if the master is in state $\text{FailSafe}$, the master execute actions to resolve the problem. Afterward, it sends message continue() changes back to $\text{Idle}$.

The role slave is the correspondent part to the master and consists of the initial state $\text{Idle}$ and the states $\text{Working}$ and $\text{FailSafe}$. If it receives the message order the state changes...
4.3. FAIL-SAFE DELEGATION

to Working. This state can be leave as soon as the order is done. Then the slave sends done to the master and the state changes back to Idle.

An entry-action in the state Working resets the clock _c0_. If the clock _c0_ reaches the value of $worktime$ and the order is not finished yet, the slave has to cancel the order, sends the message fail to the master, and changes to state FailSafe. If the order fails, the slave changes to state FailSafe, too. This state can be leave with the message continue. Then the slave changes back to state Idle. It may happen that the slave receives the message order while it is in state FailSafe. This is only the case, if a message before got lost. As the slave is not allowed to execute the order, it sends the message fail immediately and remains in state FailSafe.

**Verification Properties** There will never be a deadlock within the protocol: $AG \neg deadlock$.

A possible assignment of the time variables is as follows:

- master.timeout: 50
- slave.worktime: 40
- connector.delay-min: 2
- connector.delay-max: 5

**Consequences** Only one order can be delegated at the same time. If the slave could be handle several tasks in parallel, the pattern should be adjusted.

If messages get lost, the master has to wait until the slave must be finished. Therefore, this pattern should only be used if the probability of message loss is low.

**Examples** BeBots are working in a warehouse and have to move goods. A centralized system (the master) delegates BeBots (the slaves) to move a good to a certain position. The slave may crash which causes a blocking of the gangway or errors at the sensors or actors of the BeBot. If this is the case, the system must clean the place of accident and has to made sure that the BeBot is free of errors.

**Variants** You can extend the number of masters so that you have multiple masters that delegate an order to a slave. Moreover, you can extend the number of slaves so that a master can delegate to multiple slaves. Hereby, it is possible that all slaves have to do the same order, or every order is assigned to only one slave.

If the master is able to recognize that a failure in the system occurred and the slave must stop its work to prevent a safety-critical situation, then the master should be able to send a message to cancel the work of the slave.
**Alternative Patterns**  Use *Fail-Safe Delegation* which implements a fail operational strategy. This strategy assumes that a failed order does not need to be handled and the master can directly send a new order.

**Combinability**  Combine it with *Master-Slave-Assignment*, if you want to dynamically define who should be master and slave.
4.4 Master-Slave-Assignment

**Name**  Master-Slave-Assignment

This pattern is used if two systems can dynamically change between one state in which they have equal rights and another state in which one is the master and the other one is the slave.

**Context**  Equal, independent systems want to cooperate.

**Problem**  A system wants to cooperate with another system. During this time, they depend on each other and a safety-critical situation occurs, if they remain self-determined. Furthermore, the communication channel may be unreliable and the systems and the communication channel may fall out fully.

**Solution**  Define a pattern so that two equal roles can dynamically change into a state where one is the master that may delegate tasks or proposals to the other role (the slave). If the master or the communication channel falls out, the slave will recognize this, because master and slave exchange alive-messages with each other, and will leave his slave position.

![Diagram](attachment:master-slave-assign_peer.png)

**Figure 4.7:** Structure and Interface of Master-Slave-Assignment
**Structure** There are two peer roles, because they have the identical behavior (cf. Fig. 4.7(a)). Each role can become the master or slave at run-time. Both roles are in/out roles and have the same message interfaces for sending and receiving (cf. Fig. 4.7(b)). Thus, both peers may send the messages youSlave, confirm, noSlave, alive, and alive2 to the other peer.

The time parameters of a peer are $\$timeout1$, $\$timeout2$, and $\$period$. The connector may lose messages. The delay for sending a message is defined by the time parameters $\$delay-min$ and $\$delay-max$.

The state `Master` lost, they return from state `confirm` it confirms this using the message `alive2` messages the state changes to `Slave` If both peers had `alive2` message was received for a certain number of times (this is defined by the variable $\$tries$).

If a peer confirms the proposal and the initiator receives it, it changes to state `Master`. If both peers had `alive2` message was received for a certain number of times (this is defined by the variable $\$tries$).

A slave (i) can receive an `alive2` message from the master and has to answer with an `alive2` message, (ii) can receive an `youSlave` message and has to answer with a confirm message, (iii) has to leave the assignment if it receives the `noSlave` message and has to

![Figure 4.8: Behavior of Master-Slave-Assignment](image-url)

**Behavior** The behavior is shown in Fig. 4.8.

Both peers are in the initial state NoAssignment. A peer may send the message `youSlave` if it had rested in this state at least $\$waittime$ time units. After sending this messages the state changes to MasterProposed. If the other peer receives this message, it confirms this using the message confirm and changes to state Slave. If both peers had send the message `youSlave`, they both return to state NoAssignment. If messages are lost, they return from state MasterProposed after $\$timeout1$ time units.

If a peer confirms the proposal and the initiator receives it, it changes to state Master. The state Master must be leaved after $\$period$ time units either with (i) sending an `alive2` message to the slave, (ii) consuming an `alive2` message that was send from the slave, (iii) breaking the assignment by sending the `noSlave` message to the slave, or (iv) with a timeout that occurs if no `alive2` message was received for a certain number of times (this is defined by the variable $\$tries$).
change to state NoAssignment, or (iv) has to change to state NoAssignment, because no message was received after $timeout1$ time units. This state change is allowed, because after that time, the slave can assume that the master or the communication channel has fallen out.

**Verification Properties** There will never be a deadlock within the protocol: $AG \neg \text{deadlock}$. If one of both peers is in state Master, then the other peer must always be in state Slave: $AG \text{peer1.Master} \implies \text{peer2.Slave}$. It must be possible for a peer to reach the state Master: $EF \text{peer.Master}$.

A possible assignment of the variables is as follows:

- peer.waittime: 50
- peer.timeout1: 100
- peer.period: 100
- peer.timeout2: 700
- peer.tries: 5
- connector.delay-min: 49
- connector.delay-max: 49

**Consequences** By using the messages alive and alive2, the slave may reset its slave status without compromise the safety-critical cooperation. Though, it must be predefined what the timeout of a system or the communication channel may be. Yet, the

Furthermore, a developer has to consider that a peer cannot decline a proposal and cannot propose the end of the collaboration.

**Examples** Two RailCabs are driving on the same track. If the rear RailCabs indicates, that its distance to the front BeBot is below a certain limit, it may assigns itself as the master and the front RailCabs as the slave. After this, both RailCabs can form a secure convoy using the *Synchronized Collaboration* pattern.

**Variants** You can extend the number of peers to more than two if you make sure, that only one peer can be in state Master at the same time and all other peers must be in state Slave. Furthermore, you can enable the slave to decline a proposal or propose to the end of the assignment.

**Alternative Patterns** This pattern is based on the pattern *Synchronized Collaboration*. However, the roles of *Synchronized Collaboration* are pre-defined. Moreover, *Master-Slave-Assignment* assumes that the strategy of the master is different from the strategy of the slave. Within Synchronize Strategy it is possible, that master and slave use the same strategy.
Combinability There exists several patterns that are based on the two roles master and slave. Combine them with with *Master-Slave-Assignment*, if you want to dynamically define who should be master and slave.
4.5 Periodic Transmission

**Name**   Periodic Transmission (initially defined by [BDG+11] and [Dre11])

This pattern can be used to periodically transmit information from a sender to a receiver. If the receiver does not get the information within a certain time, a specified behavior must be activated to prevent a safety-critical situation.

**Context**  Information exchange between two systems.

**Problem**  If the receiver does not get the information within a certain time, a safety-critical situation can occur. This must be prevented.

**Solution**  If the receiver does not get the information within a certain time, a specified behavior must be activated to prevent the safety-critical situation.

![Diagram](attachment:Periodic_Transmission.png)

Figure 4.9: Structure and Interfaces of *Periodic Transmission*

**Structure**  The pattern consists of the two roles sender and receiver (cf. Fig. 4.9(a)). sender is an in-role. receiver is an out-role.

Which message each role can receive resp. send is defined in the message interface (cf. Fig. 4.9(b)). Here, the sender may send the message data() to the receiver.

The time parameter of the role sender is $period$, the time parameter of role slave is $timeout$. The connector may lose messages. The delay for sending a message is defined by the time parameters $delay-min$ and $delay-max$. 
Behavior  The behavior is shown in Fig. 4.10.

The role sender consists of the initial state PeriodicSending only. The sender must send each $\$period$ time units a message data to the receiver.

The role receiver consists of the initial state PeriodicReceiving and the state Timeout. The standard case is that the receiver receivers a message data periodically. Though, if the message data got lost or the sender falls out, the receiver changes to state Timeout and activates a certain behavior to avoid the safety-critical situation. As soon as the receiver receives a message data again, it changes back to state PeriodicReceiving.

Verification Properties  There will never be a deadlock within the protocol: $AG \ not \ deadlock$.

Several assignments of the time variables are possible. Though, they are really meaningful, if the receiver only changes to state Timeout if it is not possible anymore that the message will arrive. Therefore, a meaningful assignment is as follows:

- sender.period: 50
- receiver.timeout: 55
- connector.delay-min: 2
- connector.delay-max: 5

Consequences  For the use of this pattern, you have to predefine the time period in which the data should be transmitted.
4.5. PERIODIC TRANSMISSION

**Examples**  A Bebots drives in an unknown area. It uses its distance sensors to detect an obstacle. The measured distance is transmitted periodically between two components of the BeBot system. If the component receives no distance information within a certain time, it has to stop the movement of the BeBot to avoid collisions.

**Variants**  The data may be sent non-periodically, e.g., event-based. Though, this requires a reliable connector.

**Alternative Patterns**  Use the pattern Limit Observation pattern, if there is a predefined limit and the sender informs event-based if the limit is redeemed or violated.

**Combinability**  Combine it with Master-Slave-Assignment, if you want to dynamically define who should be sender (resp. master) and receiver (resp. slave).
4.6 Producer-Consumer

**Name**  Producer-Consumer (initially defined by [Rie11]). Also known as *Alternating Lock*.

This pattern is used when two roles shall access a safety-critical section alternately, e.g., one produces goods, the other consumes them. The pattern guarantees that only one is in the critical section at the same time.

**Context**  Working in a safety-critical section.

**Problem**  There exists a section where information or goods can be stored. The size of the section is 1. Furthermore, there exists two different systems. The one produces the information/good, the other consumes/clears it. The consumer may not act, if nothing is produced. Therefore, consuming and producing must alternate.

Moreover, you have to satisfy that only one system / component is in the critical section at the same time. Otherwise, a safety-critical situation. Therefore, the participants must be asure that nobody is in the critical section, when they enter it.

**Solution**  Define a coordination protocol that specifies a bidirectional alternating lock. A producer produces the goods and informs the consumer as soon as the producing is finished and blocks is activities as long as the consumer does not send that it consumed the information/good.

![Figure 4.11: Structure and Interfaces of Producer-Consumer](image)

(a) Structure of *Producer-Consumer*

(b) Interfaces of *Producer-Consumer*
Structure  The pattern consist of two roles producer and consumer (cf. Fig. 4.11(a)). Both roles are in/out-roles.

Which message each role can receive and send is shown in the message interfaces (cf. Fig. 4.11(b)). The producer may send the message produced to the consumer. The slave may send the message consumed to the producer.

The connector must not lose messages. The delay for sending a message is defined by the time parameters $delay-min$ and $delay-max$.

![Diagram of Producer-Consumer pattern]

Figure 4.12: Behavior of the Producer-Consumer

Behavior  The behavior is shown in Fig. 4.12.

The role producer has the initial state Producing and has reserved the critical section. If he leaves the critical section, with the message produced the consumer reaches the state Consuming and no other resources can be produced. If the role consumer receives the message produced, it knows the producer has leaved the critical section and it can enter it by itself. If the producer receives the messages consumed, the consumer has leaved the critical section and the producer can enter it again.

Verification Properties  There will never be a deadlock within the protocol: $AG \not= deadlock$. It will never be the case that the producer is in state Producing and the consumer is in state Consuming: $AG \not= (\text{producer.Producing and consumer.Consuming})$.

A possible assignment of the time variables is as follows:

- connector.delay-min: 2
- connector.delay-max: 5
Consequences  Only one producer and one consumer are interacting with each other.
   The verification properties only hold for the given behavior model, if you can guarantee that no message is lost.
   As there are no state invariants, each role must not leave the critical section.

Examples  Two components of the BeBot have access to a shared memory. The first components can write on the memory, the second can read. While writing, it is forbidden to read, while reading it is forbidden to write.

Variants  Add invariants to the Producing and Consuming state to restrict the time they can be in the critical section.
   You can extend the number of producers and / or the number of consumers. For many consumers, it may be possible to consume in parallel (e.g. reading a shared memory).
   State invariants could restrict the time, the consumer and or the producer can be active.
   The similarity to semaphores can be used to expand the pattern so that you can have a variable size for the inventory.
   Change the pattern in such a way, that it allows message lost.

Alternative Patterns  This pattern is related to the pattern Block Execution, which provides also the possibility to lock actions within another component or to unlock them.
   The difference to Producer-Consumer is that only one component sets the lock and the unlock.

Combinability  Combine it with Master-Slave-Assignment, if you want to dynamically define who should be producer (resp. master) and consumer (resp. slave).
4.7 Block Execution

This pattern coordinates a blocking of actions, e.g., due to safety-critical reasons.

**Name**  Block Execution (initially defined by [Dre11]). Also known as *Start-Stop*, and *Guard*.

**Context**  A system operates under changing conditions.

**Problem**  A system executes a certain task that must be stopped, e.g. if a safety-critical station appears or if it is not necessary that it operates.

**Solution**  Respect the principle to separate concerns and therefore define a coordination protocol between a guard and an executor. Enable the guard to monitor the environment resp. the current situation. Only if acting is safe resp. necessary, the guard grants permission to the executor to act. At first, the permission denied, because the guard first has to explore the situation.

![Diagram of Block Execution](image)

---

**Structure**  The pattern consists of the roles *guard* and *executor* (cf. Fig. 4.13(a)). The role *guard* is an out-role; the role *executor* is an in-role.

Which message each role can receive and send is shown in the message interfaces (cf. Fig. 4.13(b)). The guard may send the messages *free* and *block* to the executor.
The connector must not lose messages. The delay for sending a message is defined by the time parameters $delay-min$ and $delay-max$.

![Diagram of Block Execution]

**Behavior** The behavior is shown in Fig. 4.14.

The role guard consists of the initial state Blocked and the state Free. The guard sends the message free to the executor as soon as the executor may work and changes to state Free. As soon as the guard detects that the executor must stop his work, it sends the message block and changes to state Blocked.

The role executor consists of the initial state Blocked and the state Free. When the executor receives the message free, it change to state Free and starts its work. When the executor is in state Free and receives the message block, it changes to state Block and stops its work.

**Verification Properties** There will never be a deadlock within the protocol:

$AG \text{ not deadlock}$.

If role guard is in state Free, the executor will also be in state Free in the future:

$AG (guard.FREE \implies AF \text{ executor.FREE})$

If role guard is in state Blocked, the executor will also be in state Blocked in the future:

$AG (guard.BLOCKED \implies AF \text{ executor.BLOCKED})$

A possible assignment of the time variables is as follows:

- connector.delay-min: 2
- connector.delay-max: 5

**Consequences** It is typically for this pattern, that the guard does not delegate a certain job to the executor, but only activate or deactivate the actions of the executor. Therefore, the executor must know its actions from beginning.
Furthermore, this actions must be blockable (paused or canceled) within a short amount of time.

The verification properties only hold for the given behavior model, if you can guarantee that no message is lost.

The blocking and the working may be infinite long. You can add time invariants to restrict this.

**Examples** Within a BeBot, among others, two components exist: DistanceControl and Exploration. DistanceControl measures the distance to objects within the environment. Exploration lets the BeBot move within the environment to gather information. If DistanceControl indicates an object, the BeBot can collide with, it can block the actions of the Exploration, so that the BeBot does not collide. DistanceControl would be act with role guard, Exploration with role executor.

**Variants** It is possible, that there are several guards and only one executor (n:1). It is possible, that there is one guard and several executors (1:n).

State Free and / or state Blocked can both have invariants to reduce the time, when the executor is blocked or free to act.

Change the pattern in such a way, that it allows message lost.

**Alternative Patterns** Fail-Safe-Delegation and Fail-Operational-Delegation both enable a master to delegate a job to his slave. Here it is pre-defined which job the slave has to do. Within the guard-pattern, the role guard does not need to know which concrete job the executor has to be done.

**Combinability** The pattern guard can be combined with several patterns to block one of the acting roles (master, sender, client) of these patterns. For example three components A, B and C exist. The communication between component A and B is defined by pattern Guard. The communication between component B and C is defined by pattern Fail-Safe-Delegation. Component A (acting as role guard) can block component B to delegate new orders to Component C (which is acting as the slave), if the delegation of an order within the master depends on the fact, that role executor is in state Free.
4.8 Limit Observation

**Name**  Limit Observation (initially defined by [Dre11]).

This pattern is used to communicate if a certain value violates a defined limit or not.

**Context**  Information exchange between participants.

**Problem**  Two participants exist within a system. One collects numerical information, the other wants to know them. In particular, he wants to know if the numerical information violates a certain limit or not.

**Solution**  The goal should be to avoid as much communication as possible. Therefore, define a coordination protocol that consists of the two roles **provider** and **observer**. The provider collects the data and only informs the observer if the limit is violated or redeemed. At first, it is unknown if the limit is violated or redeemed, because the provider first has to explore the situation.

In addition, the pattern warranted a disjunction of the observation and the processing and analysis of the environment situation.

![Structure of Limit Observation](image)

(a) Structure of *Limit Observation*

<table>
<thead>
<tr>
<th>lim-obs_provider-to-observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>limitViolated()</td>
</tr>
<tr>
<td>limitRedeemed()</td>
</tr>
</tbody>
</table>

(b) Interface of *Limit Observation*

Figure 4.15: Structure and Interface of *Limit Observation*

**Structure**  The pattern consists of the roles **provider** and **observer** (cf. Fig. 4.15(a)). The role **provider** is an out-role; the role **observer** is an in-role.
Which message each role can receive and send is shown in the message interfaces (cf. Fig. 4.15(b)). The provider may send the messages limitViolated and limitRedeemed to the observer.

The connector must not lose messages. The time parameter of the role provider is $worktime$. The delay for sending a message is defined by the time parameters $delay-min$ and $delay-max$.

![Behavior Diagram]

**Behavior** The behavior is shown in Fig. 4.16.

The role provider starts in state MeasuringLimit and stays there not longer than $worktime$ units of time. In this state the first measurement will be done and the provider checks if the limit is redeemed or violated. If it is redeemed the state changes to LimitRedeemed and the message limitRedeemed is send to the observer. If the limit is violated, the state changes to LimitViolated and the message limitViolated is send to the observer. If the provider is in state LimitViolated and recognizes that the results of the measurements changes so that the limit is not violated anymore, the provider changes to state LimitRedeemed and sends the message limitRedeemed. If the provider is in state LimitRedeemed and recognizes that the results of the measurements changes so that the limit is violated, the provider changes to state LimitViolated and sends the message limitViolated.

The observer is the correspondent part of the provider and is initially waiting for the provider if the limit is violated or redeemed. It reacts on the messages of the provider and changes to state LimitExceeded if the value exceeds the limit or to LimitRedeemed if value redeems the limit.

Figure 4.16: Behavior of Limit Observation
**Verification Properties**  There will never be a deadlock within the protocol: $AG \text{ not deadlock}$.

If the provider is in state LimitViolated, the observer will be in state LimitRedeemed in the future: $AG (provider.LimitViolated \implies AF observer.LimitRedeemed)$

If the provider is in state LimitRedeemed, the observer will be in state LimitViolated in the future: $AG (provider.LimitRedeemed \implies AF observer.LimitViolated)$

A possible assignment of the time variables is as follows:

- provider.worktime: 40
- connector.delay-min: 2
- connector.delay-max: 5

**Consequences**  Within the system, a limit must be pre-defined. If the limit is changed during run-time, the provider must be informed. It may be possible that the observer does not need to be informed, if the limit changes.

The verification properties only hold for the given behavior model, if you can guarantee that no message is lost.

**Examples**  One Example for the use of this pattern could be, that you have a limit value a BeBot can approach an object, so you can use the pattern Limit Observation like a distance control.

**Variants**  You can extend the number of observers (1:n), if different components want to know if the limit exceeds. Another alternative is, that you extend the number of providers (n:1), so that the observer reacts, if one of the providers recognizes an exceeded limit. Thus, a m:n variant is also possible.

Furthermore, it could be useful to periodically inform the observer, that nothing has changed. Using this, the observer can be sure, that the provider is still active.

You can differentiate the violation, whether the measured value is above or below a certain limit.

**Alternative Patterns**  There are relations to the Request Information pattern: The difference is, if you use the pattern Limit Observation the information source informs about changes, if you use Request Information, the other component has to ask for this.

Another related pattern is Guard, where to role executor is blockable by the role guard. Within the pattern Guard, the sending role guard decides, if the executor should be blocked. Within the pattern Limit Observation, the receiving role observer can decide which actions should be done, if the limit exceeds.
**Combinability**  Combine it with *Fail-Operational-Delegation*, if you want that the observer can change the limit during run-time.
Chapter 5
Developing Advanced Mechatronic Systems using Real-Time Coordination Patterns

In this section, we illustrate how a developer may use the provided Real-Time Coordination Patterns during development of a concrete system. The general process for each pattern is depicted in Fig. 5.1.

The developer starts with the requirements to the coordination protocol. Based on these requirements, the developer selects a Real-Time Coordination Pattern which suites the requirements in Step 1. In Step 2, the developer may adapt the Real-Time Coordination Pattern to the concrete domain of the system under development. We call this an application-specific adaptation which we describe in detail in Chapter 5.1. At the end of Step 2, we perform model checking to ensure that all verification properties are met. The result is a Real-Time Coordination Protocol. In Step 3, this protocol is applied to the components of the system to specify their communication. That requires an implementation-specific refinement which we introduce in Chapter 5.2. The correctness of the refinement is ensured by a refinement check. Finally, the result is a MECHATRONICUML model of the system under construction.

5.1 Application-specific Adaptation

Real-Time Coordination Patterns are intended to be reused in different applications that operate in different domains. Consequently, they abstract from all application-specific

Figure 5.1: Process for Developing with Design Patterns
5.2. IMPLEMENTATION-SPECIFIC REFINEMENT

details, e.g., concrete timing information, and use generic names for states and messages. The Real-Time Coordination Pattern *Synchronized Collaboration* of Fig. 4.2 gives an example.

When applying a Real-Time Coordination Pattern to an application running in a specific domain, the developer needs to specify concrete values for all time parameters. In addition, the developer needs to specify the properties of the connector. That includes the message delay, the consideration of message loss, and the concrete implementation variant of a buffer.

Besides the mandatory steps described above, the developer may adapt the Real-Time Coordination Pattern to the application. This adaptation includes: (1) renaming elements (protocol, roles, states, asynchronous messages, clocks, variables, operations) to concretize their application-specific meaning, (2) add new message parameters, (3) change the state hierarchy (increasing or flattening), (4) add variables and clocks, and (5) splitting transitions into several transitions with intermediate states. Further adaptations, e.g., adding entirely new states, transitions, and messages, change the solution provided by the pattern significantly. Then, it cannot be assured that the verification properties are still meaningful and sufficient for guaranteeing the safety of the resulting protocol.

After executing all adaptation steps, we obtain a Real-Time Coordination Protocol for the specific application. Finally, we perform model checking on the Real-Time Coordination Protocol to ensure that it satisfies all verification properties specified in the pattern definition. Model checking is only possible for Real-Time Coordination Protocols because the timing information, which is essential for the verification, is only available for the Real-Time Coordination Protocol and not for the pattern. The actual model checking task is carried out by a timed model checker as, e.g., UPPAAL [BDL04].

### 5.2 Implementation-specific Refinement

In this step, we assign the resulting Real-Time Coordination Protocols to the components of the system under construction to define their communication. The assignment of a Real-Time Coordination Protocol to a component requires a refinement of the protocol. That refinement is used to integrate the Real-Time Coordination Protocol with the internal behavior of the component and to resolve conflicts or dependencies between several protocols. As an example for such dependencies, consider the RailCab system. There, a RailCab may only enter the convoy mode if it is correctly registered at a track side control unit. In addition, the assignment of a protocol to a concrete component might require the implementation of component specific operations. In our example, the front role of the convoy coordination protocol needs to be extended by an implementation that determines whether a convoy is useful or not.

The changes which we allow for the implementation specific refinement are so-called lightweight changes only. The lightweight changes that we support are: (1) adding deadlines to transitions, (2) adding actions to states and transitions, (3) adding synchroniza-
tions, and (4) splitting transitions into a sequence of states and transitions. The states that originate from splitting transitions may receive invariants; the transition may receive additional time guards.

The lightweight changes do not require to repeat the full verification of the Real-Time Coordination Protocol. They only require to verify that each role of the Real-Time Coordination Protocol has been refined correctly. Checking for a correct refinement requires the refined role to be checked against the role obtained after application-specific adaptation.

In the literature, different kinds of refinement definitions are available, e.g., in [BK08], [TY01], and [HH11]. All of these preserve a different subset of CTL and TCTL formulae. The concrete choice of a refinement definition, thus, depends on the properties which have been verified for the Real-Time Coordination Protocol. In our example of Fig. 2.1, we only verified properties that must hold for all paths through the statechart. A timed simulation relation [TY01] preserves these. In [HH11], we have shown that checking for correct refinement is more efficient than a repetition of the verification.
Chapter 6

Case Study: Cooperating Robots

After we collected a set of eight Real-Time Coordination Patterns, we started a case study to answer the following questions: 1) Are our patterns reusable? 2) Is our pattern catalog including our patterns description format helpful? 3) Is our proposed process after selecting a pattern appropriate?

Our new case study were the cooperating robots (cf. Fig. 1.1 (iii)) that have to play ping-pong using different squash balls without needing a camera to trace the ball. Instead, the two fully independent robots use contact sensors to trace the ball and use communication to inform each other. Among others, the following requirements were defined: (1) The robot that gets the ball from the user may vary. (2) Balls with different properties should be supported. (3) The robot that gets the ball, first has to juggle it alone to identify the ball properties. (4) The game is restricted to a maximum of 30s. (5) The robot that gets the ball from the user has to ensure that the other robot must be ready and must know the ball properties before the ball is hit to it. Otherwise, the other robot cannot hit the squash ball correctly and the ball may fly towards the audience, which can lead to serious injuries.

A computer science student, who has basic knowledge regarding model-driven development, carried out the case study. We gave him a detailed introduction of MECHATRONIC UML, our existing case studies including the documentation and our pattern catalog. Afterward, we defined the requirements of the application. The student worked primarily on his own except some questions of him regarding the given documents.

As a result, the student defined four Real-Time Coordination Protocols to realize the robot-to-robot communication. Three of four were based on our Real-Time Coordination Patterns, namely: Master-Slave-Assignment, Synchronized Collaboration, and Fail-Operational Delegation. As the forth protocol is a good solution for an alternating transmission, we want to abstract it to a new pattern. Regarding the internal communication within each robot, the student used just three protocols (two of them were patterns and the third is a good candidate for a new pattern) to define all 13 internal connectors.

E.g., the student selected the pattern Synchronized Collaboration to synchronize the start and the end of the game between the robots while ensuring that the robot, which gets the ball from the user, may only start the game if the other robot is aware of that and does know the ball properties. He adapted the pattern to the Real-Time Coordination
Protocol Game Coordination (cf. Fig. 6.1), e.g., he defined the time variables, renamed some elements, and added a new invariant for state Gaming of statechart ping to restrict the length of the game. The model checker did not found any errors within the resulting protocol. Therefore, the student assigned the protocol to a connector and refined it by synchronization channels to integrate it with the internal behavior of the connected components.

Thus, the student successfully reused our Real-Time Coordination Patterns, adapted them to Real-Time Coordination Protocols, and reused these protocols, but with different refinement-variants. Regarding the pattern catalog, the student found the description and its format fitting and on the correct level of abstraction. The student was able to carry out the steps of our proposed process, which defined the application-specific adaptation and the implementation-specific refinement, in an efficient manner.
Chapter 7

Related Work

Patterns regarding the coordination and communication between classes, components, and systems already exist and were a great help for defining our own patterns. However, most of them only illustrate the communication informally using sequence diagrams. If at all, they only define simple timing behavior. Examples for these are the patterns Chain of Responsibility, Command, and Observer by Gamma et al. [GHJV95] and the patterns Master-Slave, Forwarder-Receiver, Client-Dispatcher-Server, and Publisher-Subscriber by Buschmann et al [BMR+96]. In contrast to these pattern systems, we formally specify the coordination using Real-Time Statecharts. Using them, the messages a communication participant may receive and send depends on its current state and on additional real-time constraints.

Alongside MECHATRONIC UML, similar domain-specific solutions exist, e.g., for the domain of multi-agent-system. For example, AGENT UML is a modeling language to specify agent interaction protocols [BMO01]. For such protocols the Foundation of Intelligent Physical Agents defined so-called protocol templates, e.g., the Propose Interaction Protocol, which proposes an interaction that can be accepted or rejected. Agent interaction protocols combine sequence diagrams with the notion of state diagrams, though they do not support real-time constraints that are mandatory in our domain of advanced mechatronic systems.

Douglass [Dou99, Dou02] defined real-time design patterns, e.g., for the collaboration between components. The behavior is described by UML Statecharts including useful real-time constraints and message exchanges. For example, he defines the pattern Watchdog. However, our behavior is described using Real-Time Statecharts, which are more expressively (e.g., they can define how long a state may be active). Furthermore, Douglass does not define how a developer may adapt this pattern for his application.

We define our patterns in a formal way and describe the process of their subsequent adaptation and refinement. Taibi et al. [Tai07] describe several approaches regarding the formalization of patterns and their subsequent refinement, but they do not focus on coordination protocols of advanced mechatronic systems.

In contrast to the mentioned related work, our patterns describe safety-critical situations (in the domain of advanced mechatronic systems) that must not happen. They of-
fer a solution which ensures that these situations never appear. Furthermore, we define a process that preserves these characteristics during the application of the pattern.
Chapter 8

Conclusions and Future Work

In this technical report, we propose patterns for Real-Time Coordination Protocols, which we call Real-Time Coordination Patterns. They describe safety-critical problems that appear when a developer designs the coordination through communication between advanced mechatronic systems. Furthermore, our patterns suggest a solution that is reusable in different applications and specifies the behavior in such a way that it can be proven regarding safety-critical requirements. We use a real application example to explain the need of our patterns. Moreover, we define how developers should develop the coordination when they use our patterns. We identified eight patterns through several case studies and were able to reuse them in a new case study. We mainly differ from existing pattern systems, because we specify safety-critical requirements that our patterns ensure.

Our patterns may help developers to increase the quality of coordination protocols for advanced mechatronic systems and to improve the efficiency developing them.

Several topics require further investigations: (1) We have to carry out a comprehensive evaluation to confirm our results. (2) We want to examine more case studies for advanced mechatronic systems to identify additional Real-Time Coordination Patterns. (3) In this paper we only introduced patterns for a one-to-one communication. However, Real-Time Coordination Patterns for one-to-many and many-to-many communication also exist. Therefore, we want to increase our catalog with such patterns. (4) To enable a developer to improve the search for an appropriate pattern, we are currently designing an ontology to store our patterns within the Semantic Web as suggested in [HC07]. Afterward, we want to enable the developer to search after patterns within our MECHATRONIC UML modeling tool Fujaba Real-Time Tool Suite. (5) Our patterns have many variants. Therefore, we want to define a feature model for each pattern so that a developer may select a feature configuration and the system automatically constructs the corresponding structure, behavior model, and verification properties.
Bibliography


[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors,


[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA, USA, 1995. 2, 8, 42


List of Figures

1.1 Examples for advanced mechatronic systems ........................................ 2
2.1 Real-Time Statecharts of Real-Time Coordination Protocol Convoy Co-
ordination ................................................................. 4
4.1 Structure and Interfaces of Synchronized Collaboration ..................... 11
4.2 Behavior of Synchronized Collaboration ....................................... 12
4.3 Structure and Interfaces of Fail-Operational Delegation .................... 14
4.4 Behavior of Fail-Operational Delegation ..................................... 15
4.5 Structure and Interfaces of Fail-Safe Delegation .............................. 17
4.6 Behavior of Fail-Safe Delegation ............................................ 18
4.7 Structure and Interface of Master-Slave-Assignment ...................... 21
4.8 Behavior of Master-Slave-Assignment ..................................... 22
4.9 Structure and Interfaces of Periodic Transmission ............................ 25
4.10 Behavior of Periodic Transmission .......................................... 26
4.11 Structure and Interfaces of Producer-Consumer ............................ 28
4.12 Behavior of the Producer-Consumer ....................................... 29
4.13 Structure and Interface of Block Execution .................................. 31
4.14 Behavior of Block Execution ............................................. 32
4.15 Structure and Interface of Limit Observation ................................ 34
4.16 Behavior of Limit Observation ............................................. 35
5.1 Process for Developing with Design Patterns .................................... 37
6.1 Real-Time Coordination Protocol Game Coordination that is adapted
and refined for the coordination of the cooperating robots ................... 41