Real-Time Coordination Patterns for Advanced Mechatronic Systems

Stefan Dziwok[1], Christian Heinzemann[1], and Matthias Tichy[2]

[1] Software Engineering Group, Heinz Nixdorf Institute,
University of Paderborn, Germany
[stefan.dziwok|christian.heinzemann]@uni-paderborn.de
[2] Software Engineering Division, Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg, Sweden
tichy@chalmers.se

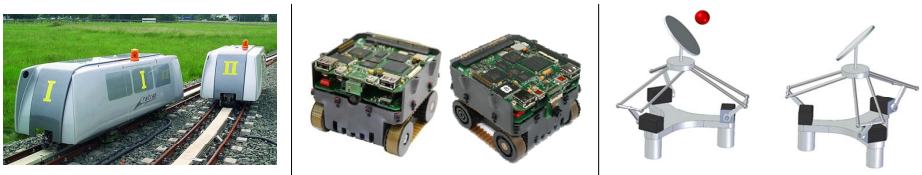
Abstract. Innovation in today's mechanical systems is often only possible due to the embedded software. Particularly, the software connects previously isolated systems resulting in, so-called, advanced mechatronic systems. Mechatronic systems are often employed in a safety-critical context, where hazards that are caused by faults in the software have to be prevented. Preferably, this is achieved by already avoiding these faults during development. A major source of faults is the complex coordination between the connected mechatronic systems. In this paper, we present Real-Time Coordination Patterns for advanced mechatronic systems. These patterns formalize proven communication protocols for the coordination between mechatronic systems as reusable entities. Furthermore, our approach exploits the patterns in the decomposition of the system to enable a scalable formal verification for the detection of faults. We illustrate the patterns with examples from different case studies.

Keywords: Advanced Mechatronic Systems, Patterns, Coordination, Communication, Real-Time, MechatronicUML

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 [17]. 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 where failures can lead to death or serious injury to people.

Furthermore, previously isolated systems increasingly form systems of systems where each autonomous part communicate with each other by means of complex message exchange protocols [16] in an ad-hoc manner. This results in very complex systems.



(i) RailCabs building a convoy

(ii) BeBots exploring the area together

(iii) Robots playing ping-pong without using a camera

Fig. 1. Examples for advanced mechatronic 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.

Figure 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. [15]). 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 [10, 6–8] 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 lack a formal description which may lead to the introduction of errors in the application 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 [12]. In contrast to the pattern formalism of [12], we further abstract from application-specific details for a better reusability, define a description format for the patterns, and have build up a catalog of patterns.

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

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

2 MechatronicUML

MechatronicUML [3] 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. 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 protocol definition formally defines the message exchange and the timing 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 UPPAAL timed automata [5] 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. Their semantics is defined by a mapping to UPPAAL timed automata [11].

Figure 2 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

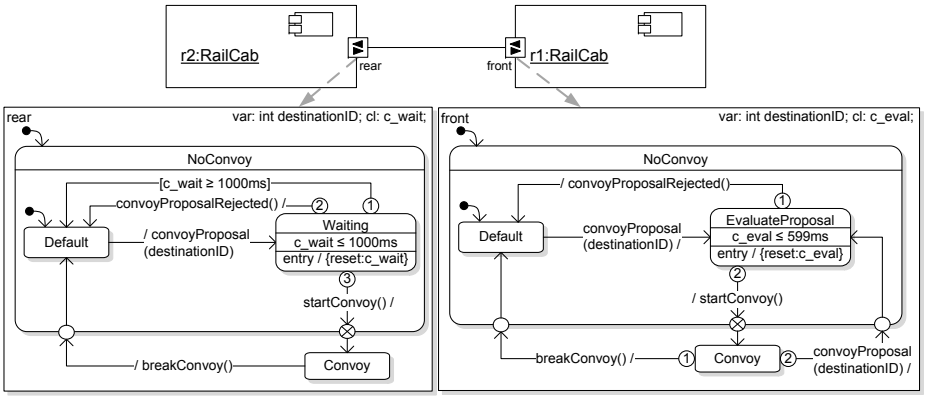


Fig. 2. Real-Time Statecharts of Real-Time Coordination Protocol *Convoy Coordination*

whether a convoy is useful or not. The decision depends, among others, on how long both RailCabs share the same route. For this reason, 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 sender to 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. For example in case of the convoy coordination, RailCabs may collide 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) [1]. Thus, the aforementioned property is formalized as $AG\ rear.Convoy\ implies\ front.Convoy$.

The properties are formally verified using timed model checkers like UPPAAL [4]. 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 transmission 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. 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 4.

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 [12]. Here, we verify each Real-Time Coordination Protocol separately before verifying each component. This is enabled by clearly separating component internal behavior and communication behavior using Real-Time Coordination Protocols.

3 Patterns for Real-Time Coordination Protocols

In 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 so, he has to ensure that it is free of faults despite hard real-time constraints, message delay and the possibility of message loss. Although the developer is able to identify faults regarding the behavior description using model checkers, removing the faults is a non trivial task. In general, specifying a Real-Time Coordination Protocol is very complex and thus, very time-consuming and error-prone. Moreover, the intention of an already existing protocol is often hard to grasp.

While modeling Real-Time Coordination Protocols for different advanced mechatronic systems, 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 (we call them Real-Time Coordination Patterns). These patterns support the developer by offering solutions that contain formal models and a comprehensive documentation. By doing so, our goal is to increase the quality of the resulting Real-Time Coordination Protocols as well as the efficiency of their development.

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" [6].

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 [12].

A Real-Time Coordination Pattern abstracts from application-specific details to be reusable in different scenarios. For example, 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 4).

The Real-Time Coordination Patterns that we identified so far are collected and described within a pattern catalog [9]. Currently, the catalog consists of eight patterns. A briefly overview of those 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 master role to a slave role. 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 master role to a slave role. 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. For example, 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 [6, 10]. We analyzed how well they fit to our patterns and, hence, decided to choose the format of Buschmann et al. [6] and adapt it to our needs.

We use all attributes of their description format except of *implementation* and *example resolved* because we propose to use code generators and resolve our example already within the other attributes. Furthermore, we divide the attribute *see also* into the two attributes *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 attribute *verification properties* to explain the verification properties that must hold for the pattern.

To conclude, Real-Time Coordination Patterns are described with the following attributes: *name* (including a short summary), *context*, *problem*, *solution*, *structure*, *behavior*, *verification properties*, *consequences*, *examples*, *variants*, *alternative patterns*, and *combinability*.

3.3 Example: Synchronized Collaboration

The Real-Time Coordination Protocol *Convoy Coordination* (Fig. 2) is a good solution when two communicating actors have to synchronize the (de-) activation of a concrete collaboration. Therefore, we abstracted it from all its application specific details and defined the Real-Time Coordination Pattern *Synchronized Collaboration* (Fig. 3). We will now give a short description for this pattern. We omitted the attributes *variants*, *alternative patterns*, and *combinability* because of the limited space of the paper. The full description can be found within our pattern catalog [9].

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 that 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.

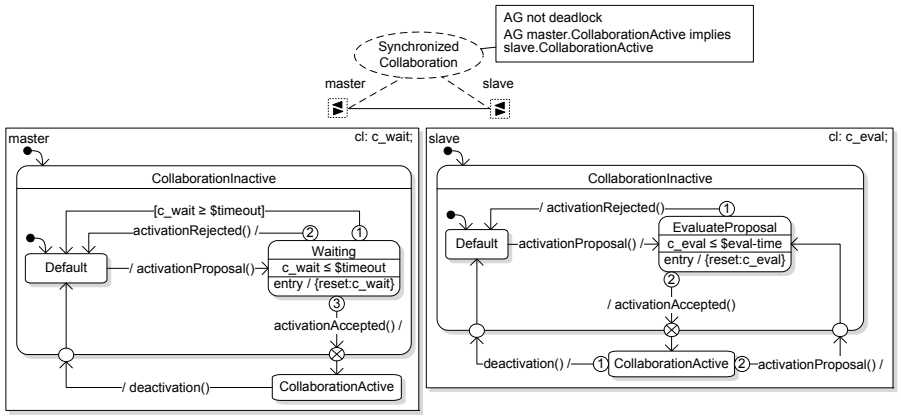


Fig. 3. Structure and behavior of Real-Time Coordination Pattern *Synchronized Collaboration*

Context: Two independent systems can dynamically collaborate in a safety-critical environment.

Problem: Switching between collaboration and no collaboration adds hazards. It may be the case that one system assumes they are working together while the other one does not think so. This must be avoided. The possibility for this problem occurrence increases, if the communication is asynchronous and the communication channel is unreliable. This patterns assumes that the safety-critical situation only occurs, if system s1 assumes they are working together and system s2 does not think so. The other way round is considered as not safety-critical.

Solution: Define a coordination protocol that enables to activate and deactivate the collaboration while it considers the given problems. The systems act with different roles: System s1 is the master and system s2 is the slave. The master initiates the activation and the deactivation. The activation is a proposal so that the slave can decide whether the collaboration is possible and useful. The deactivation is a direct command so that the master can deactivate the collaboration as soon as it is no longer useful.

Structure: The pattern consists of the two roles master and slave and a connector (Fig. 3). 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 master role is $timeout, the time parameter of slave role is $eval-time. The connector may lose messages. The delay for sending a message is defined by the time parameters $delay-min and $delay-max.

Behavior: The Real-Time Statecharts of both roles are shown in Fig. 3. A short description is as follows: First, the collaboration is in both roles inactive. The slave is passive and has to wait for the master to decide to send a proposal for activating the collaboration. In this 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: `AG not deadlock`. If the master is in state CollaborationActive, then the slave must always be in state CollaborationActive:
`AG master.CollaborationActive implies slave.CollaborationActive`.

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.

4 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 the development of a concrete system. The general process for each pattern is depicted in Fig. 4.

The developer starts with the requirements for 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 Section 4.1. 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

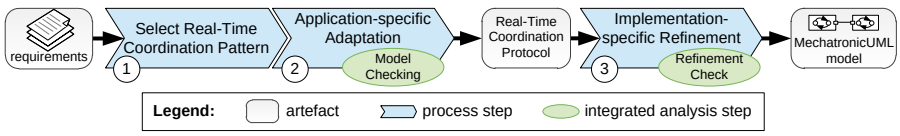


Fig. 4. Process for Developing with Design Patterns

system to specify their communication. That requires an *implementation-specific refinement* which we introduce in Section 4.2. The correctness of the refinement is ensured by a refinement check. Finally, the result is a MechatronicUML model of the system under construction.

4.1 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 details, e.g., concrete timing information, and use generic names for states and messages. The Real-Time Coordination Pattern *Synchronized Collaboration* in Fig. 3 gives an example.

When applying a Real-Time Coordination Pattern to an application of a specific domain, the developer needs to specify concrete values for all time parameters and the properties of the connector. That includes message delay, 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, messages, clocks, variables, operations) to concretize their application-specific meaning, (2) adding new message parameters, (3) changing the state hierarchy (increasing or flattening), (4) adding 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. Given the essential timing information, we can perform model checking on the Real-Time Coordination Protocol to ensure that it satisfies all verification properties specified in the pattern definition. The model checking task is carried out by a timed model checker, e.g., UPPAAL [4].

4.2 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 to integrate it 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. A RailCab may only enter the convoy mode if it is correctly registered at a track side control unit. The registration is performed by another Real-Time Coordination Protocol.

The assignment of a protocol to a concrete component might also 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 are applied

to a Real-Time Coordination Protocol must not invalidate the verified safety and liveness properties which is achieved by a refinement.

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 synchronizations, and (4) splitting transitions into a sequence of states and transitions. We allow to add invariants to the states and time guards to the transition that originate from splitting transitions.

After applying the lightweight changes, we need to verify that they have been applied correctly. Model checking the whole Real-Time Coordination Protocol, again, is costly and not necessary in this case. Instead, we only need to verify that each role of the Real-Time Coordination Protocol has been refined correctly. That requires the refined role to be checked against the role obtained after application-specific adaptation. If the refinement has been done correctly, the Real-Time Coordination Protocol with the refined role fulfills all verified properties. In [13], we have shown that checking for correct refinement of a single role is more efficient than a repetition of the verification of the whole Real-Time Coordination Protocol. Formal definitions for refinements of timed automata have been introduced in [19] and [13].

5 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 pattern description format helpful? (3) Is our proposed process after selecting a pattern appropriate?

Our new case study were the cooperating robots (Fig. 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) Initially, one of both robots receives the ball. (2) Balls with different properties should be supported. (3) The robot that initially receives 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 initially receives the ball has to ensure that the other robot must be ready and knows the ball properties before the ball is hit to it. Otherwise, the other robot cannot hit the squash ball correctly or will not perform a hit at all. Both problems lead to an unwanted behavior.

A computer science student, who has basic knowledge regarding model-driven development, carried out the design of the MechatronicUML model and especially the modeling of the coordination and communication. We gave him a detailed introduction of MechatronicUML, 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.

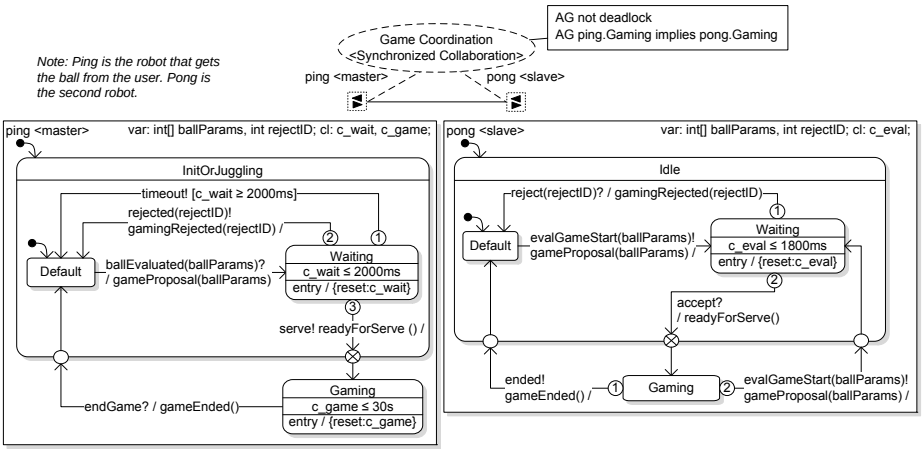


Fig. 5. Real-Time Coordination Protocol *Game Coordination* that is adapted and refined for the coordination of the cooperating robots

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

For example, 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 that initially receives the ball 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 (Fig. 5), 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.

Concluding, 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.

6 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. [10] and the patterns *Master-Slave*, *Forwarder-Receiver*, *Client-Dispatcher-Server*, and *Publisher-Subscriber* by Buschmann et al [6]. 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 depend on its current state and on additional real-time constraints.

Real-Time Coordination Patterns are protocol patterns for the domain of advanced mechatronic systems. Other domains-specific languages also defined patterns for communication, e.g., in the domain of multi-agent-system. For example, AgentUML is a modeling language to specify agent interaction protocols [2]. 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 [7, 8] defined real-time design patterns for the collaboration between components, e.g., *Watchdog*. The behavior is described by UML state machines including useful real-time constraints and message exchanges. However, our behavior is described using Real-Time Statecharts, which are more expressive (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. [18] 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 consider safety-critical situations (in the domain of advanced mechatronic systems) that must not happen. They offer a solution which ensures that these situations never appear. Furthermore, we define a process that preserves these characteristics during the application of the pattern.

7 Conclusions and Future Work

In this paper, we proposed 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 used a real application example to explain the need of our patterns. Moreover, we defined 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 Protocols for one-to-many and many-to-many communication also exist. Therefore, we want to extend 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 [14]. Afterward, we want to enable the developer to search after patterns within our MechatronicUML 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.

Acknowledgment We thank Marcel Sander for being the test person of our case study *cooperating robots*. This 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.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and Computation 104, 2–34 (1993)
2. Bauer, B., Müller, J.P., Odell, J.: Agent uml: A formalism for specifying multiagent interaction. In: Ciancarini, P., Wooldridge, M.J. (eds.) Agent-Oriented Software Engineering. pp. 91–103. Springer (2001)
3. Becker, S., Brenner, C., Dziwok, S., Gewering, T., Heinzemann, C., Pohlmann, U., Priesterjahn, C., Schäfer, W., Suck, J., Sudmann, O., Tichy, M.: The Mechatron-

icUML method – process, syntax, and semantics. Tech. Rep. tr-ri-12-318, Software Engineering Group, University of Paderborn (Feb 2012)
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. pp. 200–236. No. 3185 in LNCS, Springer–Verlag (Sep 2004)
5. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. Lecture Notes in Computer Science, vol. 3098, pp. 87–124. Springer (2003)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley (1996)
7. Douglass, B.P.: Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns. Addison-Wesley, Boston, MA, USA (1999)
8. Douglass, B.P.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley, Boston, MA, USA (2002)
9. Dziwok, S., Bröker, K., Heinzemann, C., Tichy, M.: A catalog for Real-Time Coordination Patterns of advanced mechatronic systems. Tech. Rep. tr-ri-12-319, University of Paderborn (Feb 2012)
10. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA, USA (1995)
11. Giese, H., Burmester, S.: Real-time statechart semantics. Tech. Rep. tr-ri-03-239, Software Engineering Group, University of Paderborn (Jun 2003)
12. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. In: Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '03). pp. 38–47. ACM Press (Sep 2003)
13. Heinzemann, C., Henkler, S.: Reusing dynamic communication protocols in self-adaptive embedded component architectures. In: Proceedings of the 14th International Symposium on Component Based Software Engineering. pp. 109–118. CBSE '11, ACM (Jun 2011)
14. Henninger, S., Corrêa, V.: Software pattern communities: Current practices and challenges. In: Proceedings of the 14th Conference on Pattern Languages of Programs 2007 (PLoP '07), Monticello, Illinois, USA, September 5-8, 2007 (Sep 2007)
15. National Science Foundation: A report by NSF-IRIS review panel for research on coordination theory and technology. Tech. rep., NSSF Forms and Publication Unit, National Science Foundation, Washington, D.C (1989)
16. Schäfer, W., Wehrheim, H.: The Challenges of Building Advanced Mechatronic Systems. In: FOSE '07: 2007 Future of Software Engineering. pp. 72–84. IEEE Computer Society (2007)
17. Scharnhorst, T., Heinecke, H., Schnelle, K.P., Bortolazzi, J., Lundh, L., Heitkämper, P., Leflour, J., Maté, J.L., Nishikawa, K.: Autosar - challenges and achievements 2005. In: 12th International VDI Congress Electronic Systems for Vehicles 2005, Baden-Baden. VDI Berichte, vol. 1907. VDI (2005)
18. Taibi, T. (ed.): Design Patterns Formalization Techniques. IGI Publishing, Hershey, Pennsylvania, USA (Mar 2007)
19. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstracting bisimulations. Formal Methods in System Design 18(1), 25–68 (Jan 2001)