A Modelica Coordination Pattern Library for Cyber-Physical Systems

Uwe Pohlmann
Fraunhofer IPT, Project Group Mechatronic Systems Design, Software Engineering
Zukunftsmeile 1, 33102 Paderborn, Germany
uwe.pohlmann@ipt.fraunhofer.de

Stefan Dziwok
Software Engineering Group, Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1, 33102 Paderborn, Germany
stefan.dziwok@upb.de

Matthias Meyer
Fraunhofer IPT, Project Group Mechatronic Systems Design, Software Engineering
Zukunftsmeile 1, 33102 Paderborn, Germany
matthias.meyer@ipt.fraunhofer.de

Matthias Tichy
Software Engineering Division, Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
matthias.tichy@cse.gu.se

Sebastian Thiele
Software Engineering Group, Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1, 33102 Paderborn, Germany
sthiele2@mail.upb.de

ABSTRACT

Today's embedded systems often do not operate individually anymore. Instead, they form so called cyber-physical systems, where the overall functionality is provided by the collaboration of systems. Consequently, the design of this collaboration is an important activity during development and strongly affects system quality. In previous work, we presented a catalog of reusable message-based real-time coordination patterns to avoid manual creation of new and, thus, error-prone designs. In this paper, we present an implementation of this catalog by a library in the Modelica language and an appropriate development process. The library stores ready to reuse solutions for common coordination activities and, thus, increases efficiency and effectiveness for use. Furthermore, the use of Modelica enables early holistic simulation of cyber-physical systems including feedback controllers and message-based coordination. We illustrate the library with examples from an autonomous railway vehicle and present an early evaluation.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Design, Languages

Keywords

Cyber-physical systems, design patterns, Modelica, real-time coordination, simulation

1. INTRODUCTION

Cyber-physical systems are systems of embedded systems where the provision of functionality is a result of a collaboration of many individual systems. The collaboration is typically implemented by asynchronous message-based communication between the individual systems which are mostly subject to real-time constraints.

We distinguish two types of communication which address different issues. One type deals with the basic application-independent support for exchanging data consisting of the OSI layers 1-6. The other handles the application-specific communication protocols as in OSI layer 7 to implement the required collaboration between the individual systems. This paper specifically addresses this application-specific communication for which we use the term *coordination* [25]. Examples for the coordination are the synchronization of activities between systems or delegating activities from one system to another.

The coordination between the individual systems is one of the key aspects for the successful deployment of cyber-physical systems. Thus, we presented a catalog of real-time coordination patterns in previous works [11, 10] that can be reused to realize the coordination requirements for a cyber-physical system. These design patterns also support parameters which allow some customization, e.g., concrete timing information. However, the patterns in the catalog are not readily usable as they are not available as a library in a development tool. Consequently, they require a manual error-prone work to implement for each new system.

In other previous works [23], we presented a Modelica library (based on the StateGraph2 Library) that contains modeling primitives for the modeling of state machines with real-time annotations and support for sending and receiving asynchronous messages. While this library supports the modeling of the coordination behavior, it does not contain reusable blocks for higher-level co-

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254640

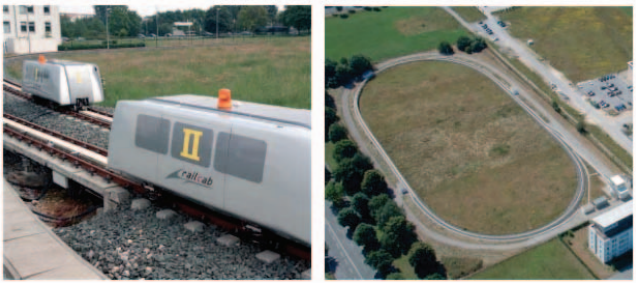

Figure 1: RailCab Prototypes (left) and RailCab Test Track (right)

ordination like the aforementioned real-time coordination pattern catalog.

Several (simulation) concepts and libraries exist that especially ease the communication design and the application of (domain-specific) design patterns. For example, Dalle et al. focus on architectural design patterns and conclude that they "may add to the overall quality of [modeling and simulation] results" [9]. Other examples are DEVSLib [28] and Tiller [29]. However, none of them provides a reusable coordination pattern library and a process for higher-level coordination protocols.

The contribution of this paper is a Modelica library which formalizes our real-time coordination patterns [10] as easily reusable Modelica classes using our existing coordination library [23]. Furthermore, we present a process for developing by using the library. The Modelica classes from the library can be subclassed in order to adapt the behavior for the specific application, e.g., by adding message parameters, adding variables, splitting transitions into multiple transitions, and adding states. The use of Modelica enables the early holistic simulation of the physical system, its environment, the feedback controllers and the asynchronous real-time coordination. Thus, the complete design can be already validated early in the development process. We illustrate the usage of the library by a Modelica model for an autonomous railway vehicle where the library is used to model the coordination for creating a platoon. Furthermore, we present the results of an early evaluation showing increased effectiveness in use with respect to amount of work saved by using the library.

In the next section, we present our running example. Section 3 gives a brief summary of the real-time coordination pattern catalog. Thereafter, we shortly discuss the real-time coordination library that we use as foundation for our pattern library in Section 4. In Section 5, we present the Modelica real-time coordination pattern library in more detail. After an early evaluation of the library in Section 6 and a review of related work in Section 7, we conclude in Section 8 and give an outlook on future work.

2. RUNNING EXAMPLE

The RailCab transportation system[1] (c.f. Figure 1) is a novel transportation system that is based on small and autonomous vehicles on rails, called RailCabs [17]. RailCabs are not coupled mechanically and can therefore form platoons dynamically in order to increase the traffic flow and reduce energy consumption by traveling in the slipstream with a speed of up to 160 km/h. Steerable wheels enable the vehicles to leave the platoon at track switches even in case of small distance between the cabs and at high speeds.

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

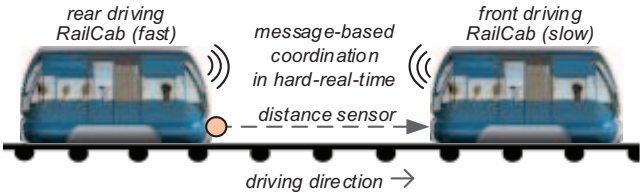

Figure 2: Platoon Scenario of the RailCab System

In order to coordinate their actions, RailCabs communicate via wireless asynchronous messages (including message parameters). This coordination must hold safety-critical requirements, e.g., a deadlock within the coordination is not allowed. Moreover, the coordination has to consider hard real-time constraints, e.g., to restrict at which point of time a message may be sent or received. Their message transmission delay is defined in a fixed range. Furthermore, the transmission of messages is unreliable, since messages can be lost or might not be valid. In addition, the coordination depends on the physical parts of the system and on environment impacts on the system. To conclude, the development of message protocols for the coordination of the RailCabs tends to become very complex.

As our running example, we use the platoon scenario from the domain of the RailCab transportation system. Its description is as follows: Two RailCabs are on the same track and have partly the same route (c.f. Figure 2). Both RailCabs may coordinate their actions via wireless asynchronous messages. The rear RailCab can measure the distance to the front RailCab using a distance sensor. The rear RailCab drives faster than the front RailCab. Eventually, the rear RailCab will catch up with the front RailCab. Then, the RailCabs can form a platoon by driving with the same speed in a close distance so that the rear RailCab can use the slipstream of the front RailCab. While driving in platoon mode, the front RailCab is not allowed to change its velocity independently. If the rear driving RailCab is in platoon mode and the front RailCab is not, the front may brake and the rear RailCab crashes into the front RailCab. Therefore, such a state configuration is not allowed. The protocol must prevent this safety-critical situation. The rear RailCab can deactivate the platoon. In this paper, we do not cover the failure situation that the front RailCab is enforced to brake during platooning for simplicity reasons of our example.

3. REAL-TIME COORDINATION PATTERNS

Coordination scenarios in cyber-physical systems are highly driven by negotiation and synchronization between the different involved partners. For example, in the platoon scenario, both RailCabs have to negotiate over their driving speed and have to synchronize the (de-) activation of the platoon. This coordination requires an intensive communication between the systems under hard real-time requirements.

As the coordination of cyber-physical systems tends to become very complex, it is beneficial to focus on the coordination between the components explicitly via coordination protocols. The language MECHATRONICUML [3] provides an explicit modeling formalism, which is called *Real-Time Coordination Protocol*. This formalism realizes coordination tasks like negotiation and synchronization by asynchronous messages and constraints on the message exchange behavior. For each coordination of two communicating entities, called roles, the developer defines a protocol.

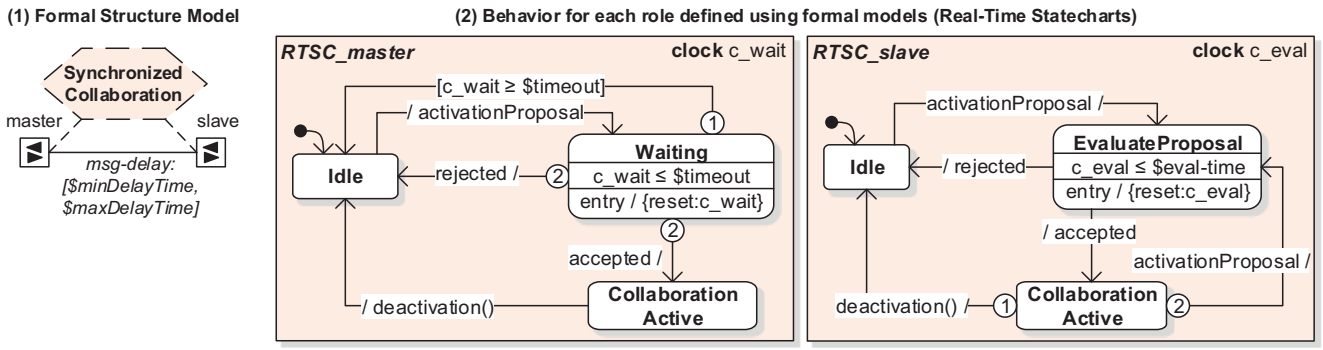

Figure 3: Structure and Behavior of Real-Time Coordination Pattern *Synchronized Collaboration* (cf. [11])

In a first step, the developer defines assumptions on the message transmission between the two roles (e.g., the transmission delay time and if messages may get lost during transmission). Afterwards, developers specify the roles[2]. For each role, the developer specifies (1) a sender and receiver interface with messages (including parameters) the role can send and receive, (2) the incoming message buffer specification (e.g., the buffer size, the buffer displacement strategy), and (3) the allowed message sequences with additional real-time constraints that restrict when to send and receive each message. For specifying the allowed message sequences, MECHATRONICUML defines a state-based formalism called *Real-Time Statechart* – a combination of UML state machines [21] and timed automata [4]. A Real-Time Statechart provides high-level modeling constructs like hierarchical states, domain-specific constructs for sending and receiving asynchronous messages over raise and trigger events, and real-time specific modeling formalism like time constraints and deadlines that are defined over continuous running clocks.

While specifying the asynchronous message-based coordination of several cyber-physical systems using Real-Time Coordination Protocols, we identified that developers have to solve reoccurring coordination design problems. These problems are hard to solve, because the developers have to consider real-time constraints of the asynchronous coordination while adhering to safety-critical requirements. Although, analysis techniques like simulation or model checking enable the developer to identify design errors, they do not provide a solution to remove the fault nor do they support to have a good design in the first place. This often results in a time-consuming development with many design iterations. Therefore, we wanted to support the developer to ease the coordination protocol modeling, especially the (state-based) behavior description.

Our solution for an improved development of coordination protocols is to provide approved solutions to reoccurring design problems by means of design patterns. In literature, several pattern languages exist, e.g., by Gamma et al. [14] and Buschmann et al. [6]. However, none of them especially targets the coordination design of cyber-physical systems by providing solutions for safety-critical coordination problems that need to be formally verified. Therefore, in previous work, we developed a new pattern language called *Real-Time Coordination Patterns* [11]. These patterns abstract from application-specific details (e.g., concrete time values and message parameters) and provide general and reusable solutions for commonly coordination design problems. Thus, developers do not have to design their protocols manually from scratch, but can reuse approved solutions. This should increase the quality of the resulting protocol as well as the development efficiency.

As usual for a pattern language, we described in natural language the aspects of each pattern, e.g., context, problem, solution, structure, and behavior. Additionally, we provide – in contrast to related work in this domain – formal models for each pattern that we successfully verified using a model checker concerning important requirements for the pattern. We identified eight different patterns so far that, among others, support to synchronize a collaboration, to delegate tasks, and to transmit information under real-time. We listed all our patterns in a catalog [10] and successfully applied our patterns while developing new cyber-physical systems.

One of our Real-Time Coordination Patterns is *Synchronized Collaboration*, which we will use for realizing our platoon scenario in Modelica using our new library (cf. Section 5). In the following, we will describe the five aspects mentioned earlier. Its *context* is that two cyber-physical systems can activate a certain collaboration at run-time; the collaboration handles advanced tasks more efficient, but also adds potential risks. The pattern deals with the following *problem*: Due to the asynchronous communication, it may happen that (1) system s1 is in collaboration mode, but system s2 is not, and (2) s2 is in collaboration mode, but s1 is not. The pattern assumes that either variant (1) or (2) can result in a hazard and therefore must be avoided. The pattern's *solution* is that the two systems should act in different roles. A role *master* can propose the activation and can deactivate the collaboration. The role *slave* answers the proposal and reacts on the deactivation. If variant (1) can result in a hazard, then s1 should be the master and s2 the slave; if variant (2) can result in a hazard, then it is the other way round.

The *structure* of the pattern Synchronized Collaboration is on the left side of Figure 3. The name of the pattern is written within a dashed hexagon. The pattern consists of the two roles master and slave (depicted as two squares and a line that connects the squares with the hexagon). Both may send and receive messages (depicted by the two black triangles). The buffer size of each role is at least three. Buffer displacement may not happen. Further, message transmission may fail. Moreover, the connector assumes a defined message delay time that consists of a minimal and maximal value (defined by the time parameters $minDelayTime and $maxDelayTime). Therefore, messages that arrive before the minimal delay are not immediately inserted into the buffer, but are deferred; messages that arrive after the maximal delay will never be inserted into the buffer and are considered as lost and are not processed.

Figure 3 shows on the right side the *behavior* of the pattern, i.e., one Real-Time Statechart per role. In the following, we focus on the Real-Time Statechart of role master. Its informal description

[2]Note, MECHATRONICUML's definition of elements like role and interface differs from other definitions, e.g., from Röhl et al. [27].

is as follows: First, the collaboration between master and slave is inactive. If the master wants to activate it, it sends an activation proposal via an asynchronous message and changes to state Waiting. Here, the master waits for an answer of the slave. The slave must accept or reject the proposal within the defined evaluation time (specified with the time parameter $eval-time). If it accepts, then the message accepted is sent; if it rejects, then the message rejected is sent. If one of both messages message get lost, i.e., the message is not inserted into the incoming buffer within the maximum delay, the master will not receive an answer at all. Then, the master will change back to state Idle when the pre-defined timeout occurs. At the transition from Waiting to Idle, a clock constraint referring to clock c_wait prevents that the state is left before the timeout (specified with the time parameter $timeout); the state invariant of state Waiting checks that the state is left when the timeout occurs. As a consequence, the timeout may only happen if no message can arrive anymore. Hence, the developer has to define a timeout value that must be greater than two times the maximum message delay plus the slave's evaluation time. If the master receives the message rejected within time, it will change back to state Idle; if the master receives the message accepted, it will activate the collaboration and change to the corresponding state CollaborationActive. As long as the master stays in this state, the collaboration with the slave will remain active. Only when the master decides to deactivate the collaboration, the slave will also deactivate it.

Due to the safety-critical context and the complex functionality, developers have to execute a holistic, discipline-spanning verification of the cyber-physical system under development including the real-time coordination protocols. This is (in most cases) not possible using formal verification like model checking due to the state explosion problem as the system contains both discrete states and continuous-time equations. Therefore, simulation has to be used for validation.

4. REAL-TIME COORDINATION MODELICA LIBRARY

A multi-domain, object-oriented, declarative simulation language for developing complex cyber-physical systems is Modelica [20]. Modelica is a textual simulation language with a graphical syntax based on textual annotations. It is supported by various tools, e.g., Dymola, SimulationX, System Modeler. "The fundamental structuring unit of modeling in Modelica is the class. Classes provide the structure for objects, also known as instances. Classes can contain equations [and algorithms] which provide the basis for the executable code that is used for computation in Modelica. [...] Connections between objects are introduced by connect-equations in the equation part of a class." [20] Modelica uses hybrid differential algebraic equations as mathematical model and has it strength in modeling and simulation of complex and large physical systems [19]. Therefore, it is used by many system engineers to develop physically realistic models. Further, Modelica can handle event-based behavior like other hybrid languages as MATLAB Simulink/ Stateflow [7] or Ptolemy [12].

We developed the *Real-Time Coordination Library*[3] [23] in Modelica to provide system engineers the possibility to extend their existing physical models with coordination behavior, within there known language and tooling. The Real-Time Coordination Library represents the MECHATRONICUML concepts for Real-Time Coordination Protocols including Real-Time Statecharts. The library is freely available under the Modelica 2 licence. Modelica 3.2 [20] offers the StateGraph2 Library [22] to model state-based behavior. StateGraph2 provides a class Step for states and a class Transition. By instantiating both classes as objects and connecting their interfaces with Modelica connectors, a developer can specify state machines. However, StateGraph2 lacks support for any message-based communication constructs. Therefore, our library extends StateGraph2 by providing support for (1) synchronous and asynchronous communication and (2) rich modeling of real-time behavior. Synchronous communication means that two transitions from different parallel automata can synchronize their firing behavior. Asynchronous communication means that an automaton can send a message and can immediately proceed without waiting for an answer. The receiver of a message has a mailbox that queues incoming messages. The receiver decides when to consume the message. A mailbox can only store a maximum number of messages and may have displacement strategies. The provided elements by the Real-Time Coordination Library are Message, Mailbox, Clock, TimeInvariantLess(OrEqual), ClockConstraintLess(OrEqual), ClockConstraintGreater(OrEqual), InputDelegationPort, and OutputDelegationPort.

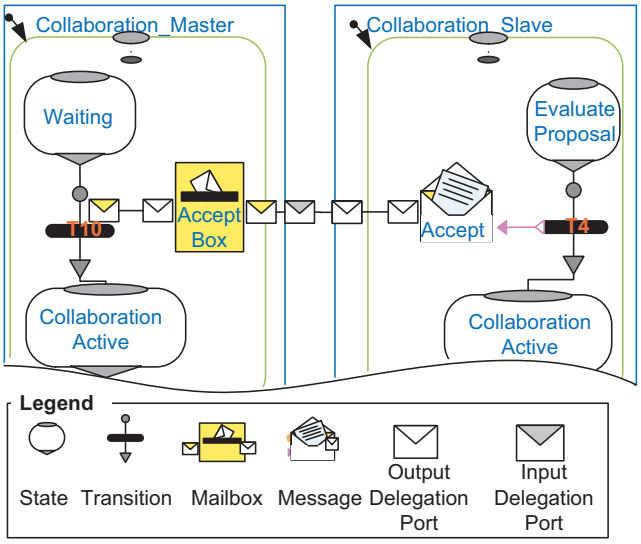

Figure 4: Interacting State Machines Modelled with the Modelica Real-Time Coordination Library and StateGraph2 (cf. [23])

Figure 4 shows an excerpt of the behavior of our scenario implemented in Modelica using the Real-Time Coordination Library. In the example, the RailCabs coordinate each other to form a platoon collaboration. On the assumption that the front RailCab is in the state EvaluteProposal and transition T4 fires, the front RailCab (the slave) sends the Accept message via delegation ports to the mailbox AcceptBox of the rear RailCab (the master). This mailbox enqueues the message until T9 fires and consumes the message. As a result of the available message Accept in the mailbox, transition T9 of the rear RailCab fires and the rear RailCab changes from state Waiting to CollaborationActive.

Figure 5 shows an enriched version of the rear RailCab behavior. Developers add timing behavior in form of the clock c, a clock constraint, and an invariant. The invariant constraints the amount of time in which the state Waiting is allowed to be active. Therefore, the clock value c_wait must be smaller or equal than the parameter value Invariant.bound. If a simulation run violates this constraint, the simulation run stops at this point in time with a concrete error, which developers have to fix. The clock c_wait is reset to zero

[3] https://github.com/modelica-3rdparty/RealTimeCoordinationLibrary

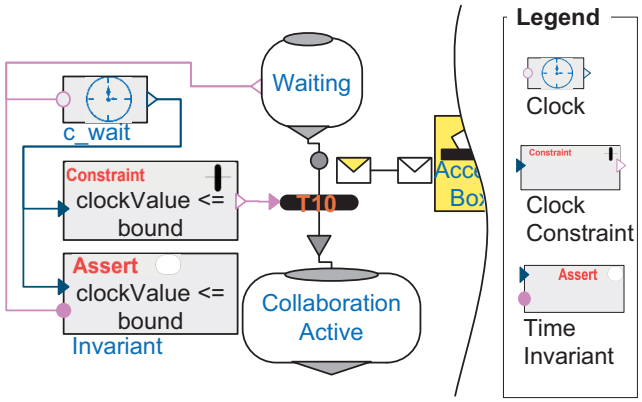

Figure 5: Modelica State Machine Including Timing Behavior (cf. [23])

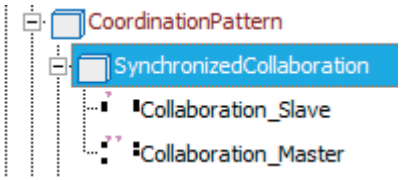

Figure 6: Synchronized Collaboration Pattern from Modelica Library

when the state Waiting gets active. The used clock constraint limits the time in which the transition T9 is allowed to fire. In the example, T9 is only allowed to fire until c_wait is smaller or equal to Constraint.bound.

5. MODELICA REAL-TIME COORDINATION PATTERN LIBRARY

Using our Real-Time Coordination Library, we enable developers to model and simulate their real-time coordination protocols. However, we realized that the effort for modeling protocols is quite a lot. Therefore, we enriched our Real-Time Coordination Library with our catalog of Real-Time Coordination Patterns to increase the effectiveness and the efficiency of the development. To be more precise, we implemented for each of the eight patterns a Modelica package[4] (each package contains Modelica classes of type model for the communication roles). Additionally, we define a systematic process for applying a pattern-based development.

As an example, Figure 6 shows the Modelica package for the Synchronized Collaboration pattern, with its Modelica model classes Collaboration_Slave and Collaboration_Master for both involved roles. Each of these model classes contain objects for states, transitions, messages, mailboxes, clocks, invariants and clock constraints.

Figure 7 shows in the left part the Collaboration_Master class and in the right part the Collaboration_Slave class. Each class consists of three states and five transitions from the StateGraph2 Library. The required objects for Messages, Mailboxes, and DelegationPort are instantiated from the Real-Time Coordination Library. For simplicity reasons, the diagram does not show the instantiated Clock and Invariant objects in this paper.

[4] https://github.com/modelica-3rdparty/RealTimeCoordinationLibrary/blob/master/CoordinationPattern.mo

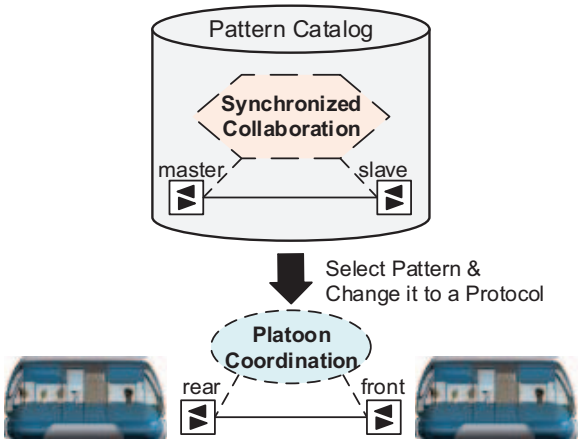

Figure 9: Select a Pattern from the Catalog and Change it to an Application-Specific Protocol

Figure 8 shows the process for using the library. The process can be embedded in a more advanced discipline-spanning development process [16]. Our process starts with (informal) coordination requirements and defines five steps for specifying application-specific coordination protocols using the Real-Time Coordination Pattern Library and a sixth step that integrates these protocols in a holistic Modelica simulation model. In the following paragraphs, we will describe these six steps.

1st Step – Pattern Selection.

First, developers have to choose a pattern for coordination based on the given (informal) requirements. For our platoon scenario, Synchronized Collaboration is appropriate (cf. Figure 9). The collaboration is the platoon, which may be active or inactive. The additional risk, mentioned by the pattern, arises if the platoon is active, because driving in a small distance can result in a crash. Furthermore, the critical situation appears only if the rear RailCab is in platoon mode, but the front RailCab is not.

2nd Step – Protocol Role Creation.

Second, developers create a new class for the rear RailCab that fulfills the master coordination protocol role and a class for the front RailCab that fulfills the slave coordination protocol role. Developers could also create a RailCab model that is able to fulfill both roles, but we omit this case here for simplicity reasons. Developers model each of the new created protocol role classes as a Modelica class of type model that inherits from the role class of the pattern. Our example obtains two new Modelica classes: Protocol_Slave_Role and Protocol_Master_Role. Listing 1 shows the master protocol class which extends the master pattern class CoordinationPattern.SynchronizedCollaboration.Collaboration_Master. As a result, the protocol role is a child of the pattern role class. A different possibility is that developers make a copy of the pattern role class. In this case, the complexity of the pattern implementation is not hidden and a bug correction of a pattern would have not an effect on existing protocol role implementations. The advantage is that object names could be set freely and the new class could be adapted without any restrictions.

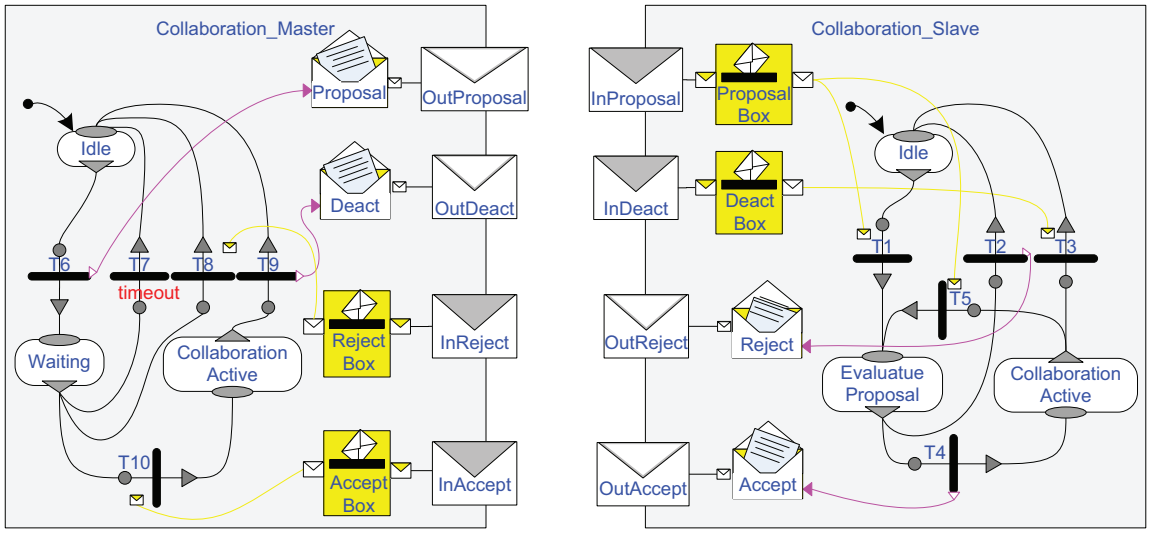

Figure 7: Master and Slave Role of the Synchronized Collaboration Pattern

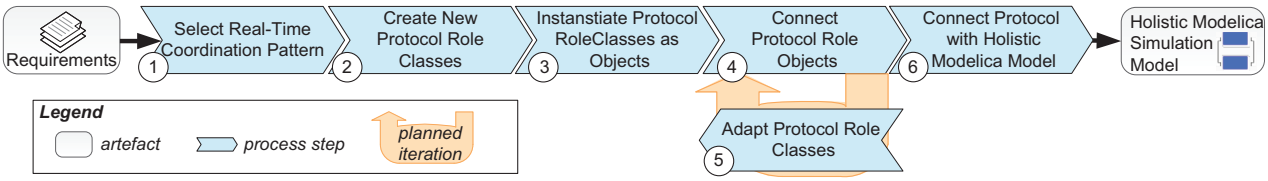

Figure 8: Process for Developing with Modelica Real-Time Coordination Pattern Library

Listing 1: Create Child of Role Master of Pattern Synchronized Collaboration

```
model Protocol_Master_Role
extends CoordinationPattern.
SynchronizedCollaboration.Collaboration_Master(
    ...);
```

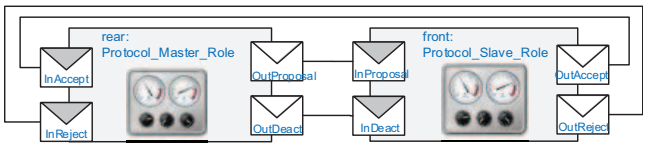

Figure 10: Modelica System Class with Both Connected Protocol Roles

3rd Step – Protocol Role Instantiation.

Afterwards, developers create a new class for the whole system and instantiate both protocol role classes as objects. Listing 2 shows the class System which instantiates the class Protocol_Slave_Role with the name front and the class Protocol_Master_Role with the name rear.

4th Step – Connection.

In the forth step, developers have to connect all ports of the objects rear and front. Therefore, they use the Modelica connect statement as shown in the equation section in Listing 2. Figure 10 shows the graphical Modelica representation.

Listing 2: Instantiation and Connection of Role Child Classes

```
model System
Protocol_Slave_Role front;
Protocol_Master_Role rear; ...
equation
connect(rear.InReject,front.OutReject);
connect(rear.InAccept,front.OutAccept);
connect(front.InDeact,rear.OutDeact);
connect(rear.OutProposal,front.InProposal);
```

5th Step – Adaptation.

As stated in Section 3, Real-Time Coordination Patterns abstract from application-specific details like concrete timing information so that developers can apply them in different applications. As a consequence, the developers need to adapt the created role child classes to the needs of their application. For the adaptation, we distinguish three kinds: (1) Mandatory adaptation: Developers must define the values of the pattern parameters. This includes concrete timing values, a concrete message delay, and concrete buffer properties like the buffer size. (2) Optional lightweight adaptation: This group consists of optional adaptations that do not effect the intent of the pattern. These adaptations include (2.1) adding a String variable for custom names (e.g., protocol, roles, states) to concretize their application-specific meaning, (2.2) adding new message parameters, (2.3) changing the state hierarchy (increasing or flattening), (2.4) adding variables and clocks, and (2.5) splitting transitions into several transitions with intermediate states. (3) Optional heavyweight adaptation: This group consists of optional adaptations that *can* effect the intent of the pattern. This group consists

Listing 3: Send Current Cruising Speed Via Accept Message

```
model Protocol_Slave_Role
(// begin modifications
OutAccept(redeclare Integer integers[0] ,
redeclare Boolean booleans[0],
redeclare Real reals[1])...);
Modelica.Blocks.Interfaces.RealInput
    cruisingSpeed;
equation
    connect(Accept.u_reals[1], cruisingSpeed)
```

Listing 4: Customize Protocol Role Master Synchronized Collaboration

```
model Protocol_Master_Role
...
(// begin modifications
AcceptBox(numberOfMessageReals=1,
    delayTime=0.05),
T10(numberOfMessageReals=1),
InAccept(redeclare Integer integers[0],
redeclare Boolean booleans[0],
redeclare Real reals[1]), ...);
```

of all possible adaptations that are not part of the other two groups, e.g., adding entirely new states, transitions, messages, invariants, clock constraints. We do not forbid these adaptations, because we would otherwise decrease the reusability of our patterns. As a technical limitation, deleting objects and connections from a pattern is not possible, because this is not possible in Modelica when using inheritance. If the pattern roles are copied and then adapted objects can also be deleted.

In our scenario, the front RailCab adds its current velocity as parameter to the platoon accept message, such that the rear RailCab is able to adapt its velocity when getting the accept message. Therefore, developers add the variable Modelica.Blocks.Interfaces.RealInput cruisingSpeed to the class Protocol_Slave_Role as Listing 3 shows and connects the variable cruisingSpeed to the input of the message Accept in the equation section. Further, the outgoing delegation ports of this message, the incoming delegation ports, the receiving mailbox, and transitions which receive this message have to be adapted.

Listing 4 shows these mandatory adaptations of the receiving protocol role in the modification part. The redeclare construct replaces the declaration of the parent class. In Listing 4 the variables integers, booleans, and reals of the delectation port InAccept are replaced. The array reals[1] gets the size one instead of zero because the velocity of the slave RailCab is communicated. For our example, the maximum delayTime for sending a message from the sender to the receiver is 0.05*seconds*. Therefore, the mailbox gets a delay of this time period before a message is enqueued.

In the example, the rear RailCab as the master adapts its velocity when getting the accept message with the velocity of the front RailCab. Therefore, developers have to assign the value first to the local variable velocityOfSlave when transition T10 fires. Listing 5 shows that the variable velocityOfSlave gets the value of the first real parameter of the received Accept message at transition T10 from the class Protocol_Master_Role, when T10 fires.

Developers need an adaptation of the pattern if they want to reach different states if the platoon is rejected or a timeout occurs. Therefore, developers can split up transitions or states by redeclaring them. For example, they create a new class Adaptation, which extends the Modelica_StateGraph2.PartialParallel class. Figure 11 shows this adaptation class which has the explicit states Reject and Timeout. Depending on the Boolean value of the variables reject and

Listing 5: Add Action to Transition T1 of class Protocol_Master_Role

```
model Protocol_Master_Role
...
Real velocityOfSlave;
algorithm
when T10.fire then
    velocityOfSlave :=
        T10.transition_input_port[1].reals [1];
end when;
equation
if CollaborationActive.active then
  myVelocity = velocityOfSlave;
else
  myVelocity = cruisingSpeed;
end if;
```

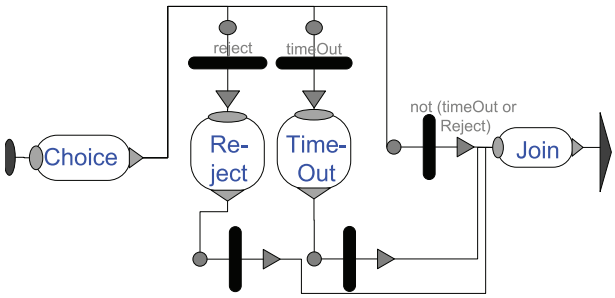

Figure 11: Modelica State Machine Modeled in the Adaptation Class

timeout the corresponding states get active. The variable reject gets true if transition T8 fires (Figure 7) and the variable timeout gets true if T7 fires. Listing 6 shows how the state Idle within the Protocol_Master_Role class is adapted by redeclaring it with the Adaptation class to integrate the adapted behavior.

Next, the developers have to assign application-specific values to the role parameters. In our scenario, the slave has the parameter $evaluationTime that specifies the time that the slave is allowed to evaluate the collaboration proposal. In our example, developers set it to 0.1 seconds. The master has the parameter $timeout that specifies the time that the master waits at most for the answer of the slave for a request of building a platoon. The $timeout is at least the sum of the $evaluationTime of the slave and two times the maximum message delay. In our example, it is set to 0.2 seconds. Listing 7 shows the customized example of Listing 2.

To conclude, we only applied mandatory and lightweight changes to our scenario, but no heavyweight changes. Thus, the intent of the pattern is not affected.

If the scenario would be more complex, new message delegation ports could be introduced, which have to be connected again. Therefore, step four and five could have several iterations.

6th Step – Holistic Modeling.

Finally, developers can add the objects for feedback controllers, physical parts, like the RailCab drive, and environmental parts for

Listing 6: Redeclare Idle State as Adaptation Class

```
model Protocol_Master_Role
  extends ...
    ( // begin modifications
      redeclare Adaptation Idle);
```

Listing 7: Adapted Inherit Parameter Values of the Protocol Roles

```
model System
Protocol_Master_Role front(evaluationTime=0.1);
Protocol_Master_Role rear(timeout=0.2);
...
```

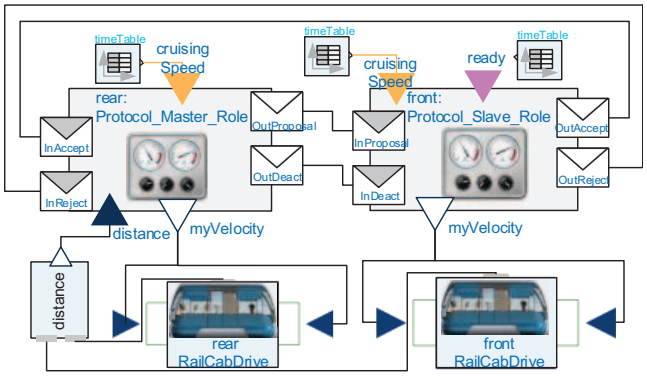

Figure 12: Adapted Modelica System for the Platoon Scenario

a testing scenario. As a result, developers get a holistic Modelica simulation model of the coordination behavior. Therefore, they can simulate a complex coordination scenario behavior in interaction with physical parts and the environment. Figure 12 shows the final simulation model, which is also included in the library.

6. EVALUATION

To get significant indications that our engineering solution with patterns is better than without patterns, the following goal should be reached: "Analyze the specification process of the coordination behavior for cyber-physical system for the purpose of evaluate the (1) effectiveness[5] and (2) efficiency[6] in use from the viewpoint of developers that add coordination protocols to an existent Modelica model".

The two research questions for our evaluation are: (1) "Is the effectiveness in use higher when engineering coordination protocols by using the pattern library?" and (2) "Is the efficiency in use higher when engineering coordination protocols by using the pattern library?". The metric for effectiveness for our pattern library is how many tasks are solved completely by developers in relation to the measured fault density. The fault density is defined as the number of encoded errors within a Modelica model, related to the size of the model. An error is a violated safety requirements of the developed protocol, e.g., the protocol is not deadlock-free, it is possible that the master thinks it collaborates but the slave thinks it does not. The efficiency in use could be measured by the time that developers need to solve a task, or the number of modeling tasks, in relation to the effectiveness. Our hypothesis is that the effectiveness and efficiency in use become better when using the pattern library.

To get a first impression of the modeling effort, comparing modeling with and without the library, we count the number of tasks

[5]Effectiveness in use: "The degree to which specified users can achieve specified goals with accuracy and completeness in a specified context of use." [18]

[6]Efficiency in use: "The degree to which specified users expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use." [18]

Task	Number of Tasks for Editing Collaboration _Slave	Collaboration _Master
Create States	3	3
Create Transitions	4	4
Create Messages	2	2
Create Message Boxes	2	2
Create Clock	1	1
Create Time Invariant	1	1
Create Clock Constraint	1	1
Create Parameter	1	1
Change Parameter	8	7
Create Connections	19	21
Create Ports	4	4
Sum	**46**	**47**
Overall Sum	**46+47=93**	

Table 1: Effort for Modeling Pattern Synchronize Collaboration Roles

during modeling our scenario. The modeling effort is important for the efficiency in use. For the measurement, we define a list of tasks that developers have to perform during creating the model that Figure 12 shows. Table 1 shows the required tasks for creating the roles of synchronized collaboration pattern of the Real-Time Coordination Pattern Library. For creating the pattern, developers have to do $46+47=93$ tasks. This modeling effort has to be done only once while creating the pattern library. Developers, who use the Real-Time Coordination Pattern Library, save this effort.

Developers, who create their application specific protocols, like the synchronization of the platoon behavior for the RailCabs, can extend existing pattern implementations. After creating the role classes, some tasks have to be performed for the application-specific details. The tasks vary depending on the concrete application. For our example, we performed 65 tasks (cf. Table 2) for creating the protocol roles and 29 tasks for creating the adaption class, which are not shown here in detail. Further, developers have to instantiate protocol roles and connect roles. Afterwards, developers have to connect existing physical and environmental parts with the protocol roles. Therefore, 19 tasks are required, which are also not shown in detail here. These tasks also need to be done if Real-Time Coordination Library is not used. The overall amount of tasks to do with the help of the library are $65+29+19=113$.

All of these tasks – except the state redeclaration – have also to be done when developing without the library. Developers have to perform $93+(65-1)+29+19=205$ tasks, modeling without usage of the pattern library. This effort does not include all the work which is necessary for thinking about good solutions and for debugging error-prone models. To conclude, if we just consider the number of tasks, the developers' effort is only $113*100/205 \approx 55\%$ when using our Real-Time Coordination Pattern Library compared to their effort with the Real-Time Coordination Library.

An empirical experiment for the evaluation would be that a group of students gets an introduction into Modelica modeling, the Real-Time Coordination Library, and the pattern catalog [10]. Afterwards, the group is split up in two groups. Both groups get different scenario descriptions, e.g., the description of our scenario in Section 2, and an appropriate Modelica model containing the physical, and environmental parts of the system. The first group has to define the coordination behavior only with the Real-Time Coordination Library; the second group has to define the behavior with the Real-Time Coordination Pattern Library. During the experi-

Task	Number of Tasks for Editing	
	Protocol _Slave_Role	Protocol _Master_Role
Change Parameter	8	8
Do Port Redeclarations	12	12
Do State Redeclarations	0	1
Create Variables	2	3
Create When Tasks	0	6
Create If Tasks	0	4
Create Connections	3	0
Create Ports	3	3
Sum	**28**	**37**
Overall Sum	**28+37=65**	

Table 2: Effort for Modeling Protocol Synchronize Collaboration Roles

ment, we measure the metrics that we defined before. Up to this point in time, the experiment has not been carried out.

7. RELATED WORK

DEVS [8] is a formalism to specify discrete event systems by state transition tables. DEVSLib [28] is a Modelica library, which provides messages and mailboxes. In contrast to messages and mailboxes of our Real-Time Coordination Library, which is written in pure Modelica, DEVSLib uses the external function interface to C of Modelica. A C-library provides the functionality to store the message information. DEVSLib uses Modelica connections to exchange the virtual address of the external storage and to provide a shared memory. In contrast to our Real-Time Coordination Pattern Library, DEVSLib currently provides no reusable implementation of existing coordination solutions.

Ptolemy [12] is a hybrid approach, which supports the notion of discrete event, finite-state machine. Ptolemy supports sending and receiving of message which have parameters like our messages [2], but has no pattern-based support for developing these message-based coordination protocols. Further, Ptolemy only supports continuous time in form of ordinary differential equations. In contrast to Modelica that "has significant advantages, particularly for specifying physical models based on differential-algebraic equations" [19].

Hamri and Baati [15] adapt existing design patterns (e.g., Gamma et al. [14]) to improve the design of DEVS-based models and simulators. Ferayorni and Sarjoughian [13] enrich DEVSJAVA (an object-oriented version of DEVS using Java) by making the design patterns of Gamma et al. (Composite, Facade, and Observer) available to the software developer of the astronomical observatory domain. In contrast to both works, we focus on domain-specific design patterns for the message-based coordination of cyber-physical system and provide a process for applying and adapting the patterns.

Reo [1] is a channel-based coordination model for component-based systems or applications. Using Reo, designers can compose complex connectors out of simple ones. In contrast to our Real-Time Coordination Pattern Library, Reo's main focus is the connector design and not the behavior design of each role. Moreover, Reo does not provide concepts for defining the component behavior based on the designed coordination. Brandt et al. simulate Reo circuits using Modelica [5]. However, they do not provide reusable Modelica models or a library of design patterns.

The TrueTime Modelica network library [26] offers elements to simulate the sending of variables over different network protocols. Thus, the library offers classes for data-link layer protocols (OSI layer 2) of wired and wireless networks. A network is modeled as a set of FIFO input and output queues and a shared communication medium. In contrast to our Real-Time Coordination Pattern Library, they offer no support for the OSI layer 7 to specify the collaboration between the individual systems.

Tiller [29] describes six Modelica design patterns. In contrast to our library, Tiller focuses on architectural patterns and provides Modelica code snippets. Dalle et al. [9] also focus on architectural patterns and conclude that all of the analyzed patterns are applicable to modeling and simulation software. We, instead, focus on coordination patterns and provide a corresponding reusable library.

ModelicaML [24] is a UML profile that enables to generate Modelica code from a subset of UML models. The provided code generator transforms UML state machines with asynchronous communication into Modelica algorithmic code. In contrast to our Real-Time Coordination Pattern Library, they do not provide a mechanism for reusing approved coordination solutions.

8. CONCLUSION

A key aspect of cyber-physical systems is the coordination of individual systems in real-time to provide a desired functionality. We increase the effectiveness and efficiency of the development of high-quality real-time coordination by providing a catalog of real-time coordination patterns. Such patterns enclose recurring and well-proven coordination solutions in an application-independent and easily reusable way.

In this paper, we extend a previously developed version of the pattern catalog and make it available as a Modelica library. We define a systematic process for developing concrete application-specific coordination behavior using the pattern library. This way, we enable Modelica experts to model and simulate high-quality coordination behavior together with the feedback controllers and the dynamic behavior of the physical parts of cyber-physical systems. A first evaluation shows that developers need to perform significantly fewer tasks for developing coordination behavior when using our library.

We have four major topics for future work: (1) We will carry out a more sophisticated evaluation of our approach as already indicated in Section 6. (2) In order to develop a concrete application-specific coordination protocol, a pattern from the library needs to be adapted manually (step 5 of our process). Currently, we offer no support for checking automatically whether these adaptations violate essential properties of the pattern like deadlock-freedom or safety-properties. However, our MECHATRONICUML modeling tool does not only offer the pattern-based modeling of coordination behavior but also its formal verification with the model checker UPPAAL. Therefore, we plan to develop an automatic transformation of MECHATRONICUML coordination models into the Modelica language based on our new library. Coordination behavior can then be modeled and verified with the MECHATRONICUML tool and translated into Modelica for a holistic simulation. (3) A system is often involved in several different coordination protocols at the same time. In our RailCab example, the rear driving RailCab might transmit its current speed regularly to the front driving RailCab using the pattern PeriodicTransmission, but only when both RailCabs are in an active SynchronizedCollaboration. This would require a synchronization of the involved protocol roles across hierarchical levels. So far, the Real-Time Coordination Pattern Library is lacking adequate modeling constructs for such a synchronization as well the process is lacking explicit support for combining protocols. (4) The underlying communication infrastructure, i.e., the lower OSI levels 1-6, may have various effects on the coordination which we

consider only in the form of fixed message delays so far. In future work, we will take more of these effects into account like message loss or dynamic message delays due to network utilization by integrating existing libraries into our own library, e.g., the library True Time [26].

Acknowledgments

This work was partially developed in the Leading-Edge Cluster 'Intelligent Technical Systems OstWestfalenLippe' (it's OWL). The Leading-Edge Cluster is funded by the German Federal Ministry of Education and Research (BMBF).

9. REFERENCES

[1] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[2] K. Bae, P. C. Ölveczky, T. H. Feng, E. A. Lee, and S. Tripakis. Verifying hierarchical ptolemy ii discrete-event models using real-time maude. *Science of Computer Programming*, 77(12):1235–1271, 2012.

[3] S. Becker, C. Brenner, C. Brink, S. Dziwok, C. Heinzemann, R. Löffler, U. Pohlmann, W. Schäfer, J. Suck, and O. Sudmann. The MECHATRONICUML design method. Technical Report tr-ri-12-326, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2012. v0.3.

[4] J. Bengtsson and W. Yi. Timed automata. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004.

[5] C. Brandt, F. Santini, N. Kokash, and F. Arbab. Modeling and simulation of selected operational it risks in the banking sector. In M. Klumpp, editor, *Proc. of European Simulation and Modelling Conf., EUROSIS-ETI*, pages 192–200, 2012.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[7] R. Colgren. Basic matlab. *Simulink And Stateflow, AIAA (American Institute of Aeronautics & Ast*, 2006.

[8] A. Concepcion and B. Zeigler. Devs formalism: A framework for hierarchical model development. *IEEE TSE*, 14(2):228–241, 1988.

[9] O. Dalle, J. Ribault, and J. Himmelspach. Design considerations for m&s software. In *Winter Simulation Conference*, pages 944–955, 2009.

[10] S. Dziwok, K. Bröker, C. Heinzemann, and M. Tichy. A catalog of real-time coordination patterns for advanced mechatronic systems. Technical Report tr-ri-12-319, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Feb. 2012.

[11] S. Dziwok, C. Heinzemann, and M. Tichy. Real-time coordination patterns for advanced mechatronic systems. In M. Sirjani, editor, *COORDINATION 2012*, LNCS 7274, pages 166–180, June 2012.

[12] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[13] A. E. Ferayorni and H. S. Sarjoughian. Domain driven simulation modeling for software design. In *Proc. of the 2007 Summer Computer Simulation Conf.*, SCSC '07, pages 297–304, San Diego, CA, USA, 2007.

[14] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA, 1995.

[15] M. E. Hamri and L. Baati. On using design patterns for DEVS modeling and simulation tools. In *Proc. of the 2010 Spring Simulation MultiConf.*, SpringSim '10, pages 121:1–121:9, San Diego, CA, USA, 2010.

[16] C. Heinzemann, O. Sudmann, W. Schäfer, and M. Tichy. A discipline-spanning development process for self-adaptive mechatronic systems. In *Proceedings of the 2013 International Conference on Software and System Process*, ICSSP 2013, pages 36–45. ACM, New York, NY, USA, May 2013.

[17] C. Henke, M. Tichy, T. Schneider, J. Böcker, and W. Schäfer. Organization and control of autonomous railway convoys. In *AVEC'08*, pages 318–323, 2008.

[18] ISO/IEC. 25010 systems and software engineering – system and software product quality requirements and evaluation (square) – system and software quality models, 2011.

[19] E. A. Lee. Disciplined heterogeneous modeling. In *Model Driven Engineering Languages and Systems*, pages 273–287. Springer, 2010.

[20] A. Modelica. Modelica - a unified object-oriented language for physical systems modeling, language specification, version 3.2, 2010.

[21] Object Management Group. Unified modeling language, superstructure, 2011.

[22] M. Otter, M. Malmheden, H. Elmqvist, S. E. Mattsson, and C. Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. In *Proc. of the 7th Modelica Conf.*, pages 364–377, 2009.

[23] U. Pohlmann, S. Dziwok, J. Suck, B. Wolf, C. C. Loh, and M. Tichy. A Modelica library for real-time coordination modeling. In *Proc. of the 9th Int. Modelica Conf., Munich, Germany*, pages 365–374, 2012.

[24] U. Pohlmann and M. Tichy. Modelica code generation from ModelicaML state machines extended by asynchronous communication. In *Proc. of the 4th Int. Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, Zurich, Switzerland*, pages 75–84, Sept. 2011.

[25] M. Radestock and S. Eisenbach. Coordination in evolving systems. In O. Spaniol, C. Linnhoff-Popien, and B. Meyer, editors, *Trends in Distributed Systems CORBA and Beyond*, volume 1161 of *LNCS*, pages 162–176. Springer, 1996.

[26] P. Reuterswärd, J. Åkesson, A. Cervin, and K.-E. Årzén. Truetime network–a network simulation library for Modelica. In F. Casella, editor, *Proc. of 7th Int. Modelica Conf.*, pages 657–662, Como, Italy, 2009.

[27] M. Rohl and A. M. Uhrmacher. Definition and analysis of composition structures for discrete-event models. In *Simulation Conference, 2008. WSC 2008. Winter*, pages 942–950, 2008.

[28] V. Sanz, A. Urquia, and S. Dormido. Introducing messages in Modelica for facilitating discrete-event system modeling. In P. Fritzson, F. E. Cellier, and D. Broman, editors, *EOOLT*, volume 29 of *Linköping Electronic Conf. Proc.*, pages 83–93, 2008.

[29] M. M. Tiller. Patterns and anti-patterns in Modelica. In B. Bachmann, editor, *Proc. of 6th Int. Modelica Conf.*, pages 647–656, Bielefeld, Germany, 2008.