Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML*

Holger Giese[1] and Wilhelm Schäfer[2]

[1] Hasso Plattner Institute for Software Systems Engineering
at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
holger.giese@hpi.uni-potsdam.de
[2] Heinz Nixdorf Institute at the University of Paderborn
Zukunftsmeile 1, D-33102 Paderborn, Germany
wilhelm@uni-paderborn.de

Abstract. Software is expected to become the dominant driver for innovation for the next generation of advanced distributed embedded real-time systems (advanced mechatronic systems). Software will build communities of autonomous agents at runtime which exploit local and global networking to enhance and optimize their functionality leading to self-adaptation or self-optimization. However, current development techniques are not capable of providing the safety guarantees required for this class of systems. Our approach, MechatronicUML, addresses the outlined challenge by proposing a coherent and integrated model-driven development approach which supports the modeling and verification of safety guarantees for systems with reconfiguration of software components at runtime. Modeling is based on a syntactically and semantically rigorously defined and partially refined subset of UML. Verification is based on a special type of decomposition and compositional model checking to make it scalable.

1 Introduction

Software has become an intrinsic part of complex distributed embedded real-time systems, also referred to as mechatronic systems. In many cases these systems are used in a safety-critical environment and implement themselves as so-called safety-critical applications. Consequently, the development of software controlling these systems has to undergo a rigorous process including the prevention of faults, employing adequate and well-founded modeling concepts, and the verification of crucial safety properties in order to detect critical faults.

* This work was developed partially in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – at the University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

J. Cámara et al. (Eds.): Assurances for Self-Adaptive Systems, LNCS 7740, pp. 152–186, 2013.

The outlined requirements are also valid for the next generation of advanced mechatronic systems. These systems are expected to behave more intelligently than today's systems by building communities of autonomous agents which exploit local and global networking to enhance their functionality [1] also named cyber-physical systems. Such mechatronic systems will employ different forms of reconfiguration to enable self-adaptation [2] often particularly targeting self-optimization in this domain. These forms of reconfiguration include complex coordination protocols which require execution in real-time, reconfiguration of control algorithms as well as components, and the coordination of the agents at runtime to adjust their behavior to the changing system goals. However, the available development techniques cannot handle systems with these advanced forms of reconfiguration.

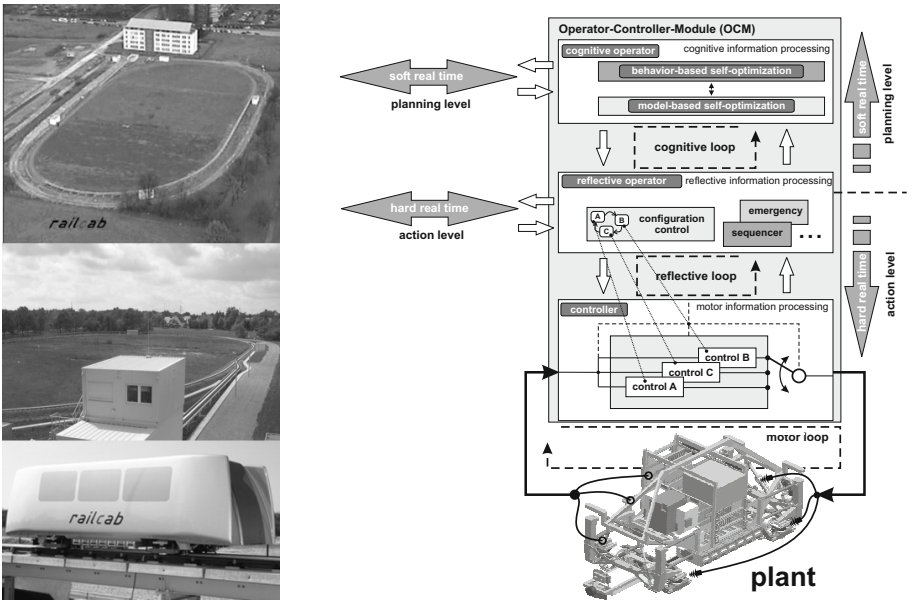
We address this challenge with the model-driven MechatronicUML (mUML) development approach, which combines domain specific modeling and refinement techniques with verification based on compositional model checking. The approach suggests modeling the software by using a refined UML component model, including the detailed definition of ports, connectors, and patterns/collaborations. We further refine the component model to define proper integration between discrete and continuous control so that the reconfiguration of hierarchical component systems can be described in a modular way. Compositional model checking is based on a domain specific decomposition of the system specification into individually checkable components and patterns/collaborations based on a common predefined architectural model. The basis for formal verification, a formal semantics for the concepts taken from UML, is given in [3].

The paper contains the following new contributions: (1) A comprehensive overview of the rationale behind the mUML MDD approach combining modeling and verification, previously covered only partially in [4, 5]. (2) A rigorous integration of the previously only independently outlined mUML modeling concepts for modeling hierarchies of reconfigurable components with hybrid behavior [6–10] and the real-time coordination of mechatronic agents [6, 7]. (3) Finally, an approach for the overall verification based on the modular verification concepts for hierarchies of reconfigurable components with hybrid behavior [6–10] and the compositional verification of the real-time coordination of mechatronic agents [6, 7] which has not been covered before.

The structure of this paper is as follows: The next section provides an overview of the mUML approach and introduces our running example, explains the underlying general architectural model, outlines how self-optimization takes place in this architecture, and provides an overview of the modeling and verification of mUML models. Section 3 introduces the modeling concepts in the form of a component model permitting specific structural reconfiguration as well as behavior specification. The concepts for modeling real-time coordination of the components are also outlined. In Section 4 the local safety criteria which have to be verified are defined and it is shown that their composition ensures global safety. In Section 5 we review existing work in the field and compare it with our approach. The last section concludes the paper.

2 The Approach

As a specific example of an advanced mechatronic system, we use the Paderborn-based RailCab research project (http://www-nbp.upb.de/en), which aims at combining a passive track system with intelligent shuttles operating individually and making independent and decentralized operational decisions. The project is funded by a number of German research organizations. A test track has been built to the scale of 1:2.5 so that the project's concetps can be tested in real operation and not just on paper (cf. Fig. 1(a)).



(a) Test track (b) OCM architecture and its elements

Fig. 1. The test track of the RailCab project and the OCM architecture

The RailCab project aims to provide the comfort of individual transport concerning scheduling and on-demand availability of transportation as well as individually equipped cars together with the cost and resource effectiveness of public transport. The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in the Transrapid (http://www.transrapid.de) to increase passenger comfort while still enabling high speed transportation and (re)using the existing railway tracks.

One particular goal of the project is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on-demand and the shuttles travel only a few centimeters apart from each other (up to

0.5m) so that a high reduction of energy consumption is achieved. Consequently, coordination between the speed control units of the shuttles becomes a safety-critical aspect and results in a number of hard real-time constraints which have to be addressed when designing the control software of the shuttles and the real-time coordination between the shuttles.

2.1 The General Architectural Model

In order to build such a complex software system, the MUML approach follows a general local architectural model of a system component for self-optimizing mechatronic systems given by the Operator-Controller-Module (OCM) as depicted in Fig. 1(b) (cf. [11]).

The OCM reflects the strict hierarchical construction of autonomous mechatronic systems, including the hardware components structured, into three levels: (1) On the lowest level of the OCM is the controller (C) including an arbitrary number of alternative control strategies (also called modes from an external perspective). Within the OCM's innermost loop, the currently active control strategy processes measurements obtained via sensors and produces control signals for the actuators. As it directly affects the plant, it is called a motor loop. The software processing is necessarily quasi-continuous and includes smooth switching between the alternative control strategies described by some form of differential equations or difference equations. (2) The controller is controlled by the reflective operator (RO), in which monitoring and controlling routines are executed. The reflective operator operates in a predominantly event-oriented manner and thus includes a control automaton with a number of discrete control states and transitions between them. It does not access the actuators of the system directly, but may modify the controller and initiate the switch between different control modes and its related strategies. Furthermore, it serves as the connecting element to the cognitive level of the OCM. (3) The topmost level of the OCM is called the cognitive operator (CO). On this level, the system can gather information concerning itself and its environment and use it for the improvement of its own behavior. (i.e. possibly complex, time-consuming computations for long-range planning.)

The distinction between the reflective and cognitive operator clearly decouples control under hard real-time constraints from long-range planning and the resulting input for self-optimization. In general the OCM-hierarchy defines a strictly hierarchical control flow. Each level tries to execute control as much as possible locally, but reconfiguration of components is decided on the next higher level similar to the reference architecture for adaptive and self-managed systems suggested in [12].

To also describe the overall architecture, the OCM hierarchy can be nested, where each nesting level may include an OCM. However, these levels do not include the controller part. Controllers, which implement the continuous part of the software, usually exist only on the lowest level of a nested OCM hierarchy. As an example, consider the above mentioned shuttles of the RailCab project. The architecture is defined by OCMs with their reflective operators and

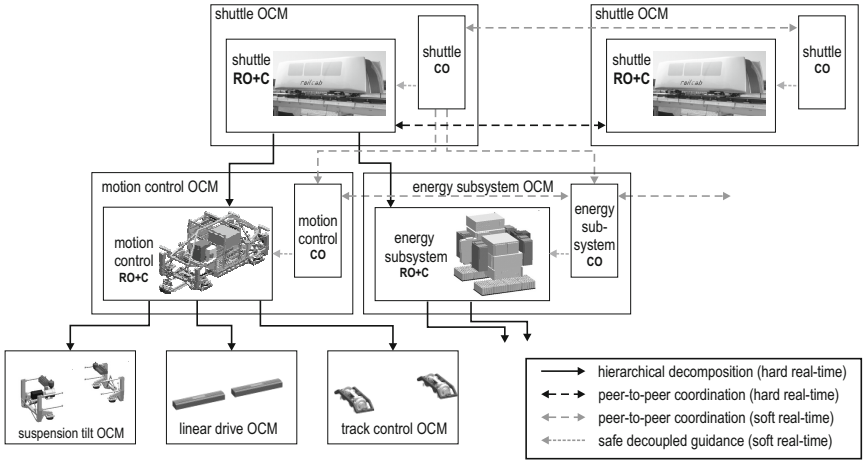


Fig. 2. The hierarchy of OCMs of a shuttle and its connections to other shuttles

the controllers as depicted in Fig. 2. A shuttle consists of components like the suspension/tilt module, the engine, the tracking module etc. which in turn are defined by OCMs.

As a complete mechatronic system usually consists of several concurrently running components, a further possibility for communication between components besides the strict hierarchical control flow exists. Top-level OCMs of several nested hierarchies, which usually represent a major system component, may act as freely interacting software agents in the overall architecture in addition to the strict hierarchies. This means that agents exchange information and collaborate in a peer-to-peer manner but that no central control is defined anymore. As examples of such major system components consider the different shuttles, stations and possibly job brokers involved with the RailCab project. These agents interact with each other in form of collaborations with well-defined role interfaces. In principle, the controllers of different agents can interact with each other, as well as the reflective operators and the cognitive operators, each on their corresponding levels. In any case their interaction is limited to a peer-to-peer style with individual messages rather than centralized, broadcasted messages.

2.2 Self-adaptation and Self-optimization

Self-optimization by means of self-adaptation can be realized in rather different forms in the outlined general architectural model depending on the specific self-optimization goals and the impact the different elements have concerning the characteristics that should be optimized.

The most obvious location for self-adaptive behavior is the cognitive operator. Due to the decoupling from the hard real-time processing complex processing steps for the self-optimization of a single OCM can be realized here. In a

subsequent step they have to be enacted by influencing the behavior of the reflective operator and the controller accordingly. Due to the temporal decoupling the cognitive operator itself can remain outside the critical part of the software and it is sufficient to only consider all possible effects the cognitive operator may have on the reflective operator and controller. Usually, the reflective operator with all its configuration variants that can be steered by the cognitive operator is designed as a safety envelope. By ensuring that all configurations of the reflective operator work properly and abstracting from the possible configuration advices from the cognitive operator, we can still ensure safe self-optimizing behavior. However, what this scheme does not help to guarantee is that the self-optimization itself is successful. An OCM optimizing its reflective operator and controller by taking the long term changes of the controlled hardware due to abrasion into account is an example for such a self-optimization for a single OCM.

It is also possible to achieve self-optimization for a whole hierarchy of OCMs where the higher level reflective operators steer the subordinated OCMs to achieve a self-optimization for the whole hierarchy. Again the cognitive operators can play the role of driving the decisions. However, in this case the lower level OCMs and their cognitive operators are guided by the higher level OCMs which determine what are their optimization goals. Thus, here we got a complex interplay of local analysis and planning activities similar to a hierarchical optimization problem where the solutions identified at the higher level OCMs influence the search space that is considered at the lower level OCMs. Similar to the local case the reflective operators and their interaction can be studied without taking the complex behavior in the cognitive operators into account. The composition of the reflective operators become a safety envelop that protects the system against failures in the self-optimization. Again, the scheme does not help to guarantee that the self-optimization across a whole hierarchy of OCMs performs well and necessarily results in an improved system behavior. The energy management in a shuttle is an example for a self-optimization across a hierarchy of OCMs. While the overall OCM has to decide how much energy could be at most consumed by each lower level OCM to achieve the current higher level goals, the lower level OCMs try to optimize their energy consumption and the performance that can be achieved taking the constraints and precedences of the higher level OCMs into account only looking at their local scope.

Furthermore, also the agents and their peer-to-peer coordination can be employed to achieve a self-adaptation of the overall system. However, in this case two rather different cases can occur.

We can have the case that a behavior of a group of agents shows some emergent behavior due to the employed peer-to-peer protocols often referred to as self-organization. These emergent properties of the behavior may result in self-optimization but may also simply provide some required system properties. Here, the role interfaces provide some protection that can be exploited and depending on the complexity of the protocols guarantees for the emergent behavior are possible. The collision freedom for the shuttles later considered in this paper falls under this case.

In contrast to such emergent properties, the peer-to-peer coordination of the agents can also result in a self-optimization by exchanging information about the context such that the other agents benefit from this. Again, the role interfaces permit to ensure that the overall protocol works. However, as the data exchange and the related data processing can be rather complex, the scheme does not permit to guarantee that the information exchange effectively results in self-optimization. Furthermore, the scheme can not exclude that erroneous data result in unsafe behavior. Consequently, in this case no development-time solution is provided and problems with the exchanged data have to be detected at run-time and related fallback strategies must be available (c.f. runtime verification). Shuttles that exchange data about the track characteristics to improve their performance (c.f. [13]) are an example for this case of group-wise self-optimization. Note that for safety reasons besides the optimized controller that exploits the data about the track characteristics in addition a fallback controller that also works in case no data is available and a unit to detect whether the optimized controller does not perform well have been part of the related system design.

2.3 Modular and Compositional Verification

Our MDD approach takes the general model of Fig. 2 as an informal architectural basis. It provides a formal definition of arbitrary OCM hierarchies, their behavior as well as their peer-to-peer communication using a refined UML component model and a refined notion of statecharts including the definition of timing constraints and hybrid behavior. This definition is the input for our model checking approach, which uses standard real-time model checkers, but before using them decomposes the overall system in such a way that the individual parts can be checked separately. Additional checks that the interfaces are well-defined interfaces and that the components refine their interfaces then guarantee, that when composing the models the separately checked safety properties are also guaranteed for complex composed system.

As all safety and time-critical aspects are handled by the reflective operators and controllers, peer-to-peer communication (across the hierarchy) is also allowed between the different cognitive operators at different levels in the nested OCM hierarchies. This may facilitate complex planning and the required information exchange between different components, but the interface between a reflective and cognitive operator in each component and on each nesting level respectively will ensure that no unsafe behavior can result from the interaction with the cognitive operators.

2.4 Tool Support and Code Generation

To complete the approach, MUML is supported by the FUJABA Real-Time Tool Suite CASE tool [13–15] and includes a code-generation scheme [16–19] that maps all the high-level timing constraints of the models to underlying real-time operating system and scheduling technologies. Additional schedulability checks

then ensure that the code executed on real-time operating systems provides the same safety guarantees as the models. Thus, the safety guarantees obtained via checking the models can also be transferred to the code level (c.f. [3]). In addition, an alternative mapping scheme onto Simulink and Stateflow Models [20] has been developed to facilitate also commercial simulation and code-generation tools.

3 Modeling

In this section we describe our solution for modeling OCM hierarchies as well as peer-to-peer networks as outlined in Fig. 2 using our extended UML component model.

3.1 The Hierarchical Component Model

To first capture OCM-like hierarchies, we describe a component model with a static structure adjusted to the needs of mechatronic systems. We then extend this component model to also cover the case of reconfiguration.

Component Structure. To support the coupling of time-continuous control behavior with discrete behavior, we extend the definition of ports in the UML component model. Ports may also be defined by time-continuous variables. While a signal is sent and received at discrete points in time (cf. SignalEvent in UML), a time-continuous variable has a well-defined value for each point in time.

As an example the MUML model of the OCM of the shuttle responsible for travelling either in convoy or stand alone mode is depicted in Fig. 3. The Shuttle component instance sh contains a AccelerationControl (AC) component instance ac representing the reflective operator and controller and a Planer component instance pl representing the cognitive operator. The reflective operator which mediates between the other two OCM components is represented by the shuttle component sh itself. This component computes the acceleration needed to achieve a specific goal (keeping a specified speed level or keeping a specified distance from the predecessor). The AccelerationControl component has five incoming continuous ports and one outgoing continuous port. We distinguish here between *permanent ports* and *optional ports*. The former are depicted by a black triangle and the latter by a white triangle to indicate that they are only active in some of the modes as introduced later when considering reconfiguration.

The incoming continuous ports are for the values current velocity v_{cur}, the current distance Δ_{cur}, and the velocity of the front shuttle v_{Front} provided by sensors, and the required velocity v_{req} and the required distance Δ_{req} which are parameterized reference inputs. The outgoing port sends acceleration values to the appropriate hardware actuator devices. In addition, the ac component contains discrete behavior to switch between keeping a certain distance and keeping the velocity at a constant level, and is thus a hybrid component.

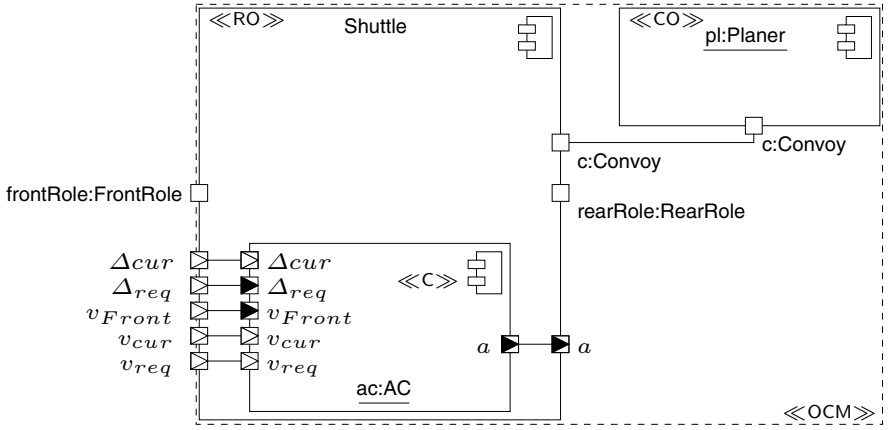


Fig. 3. Example for a component structure of a Shuttle OCM

The specification of component behavior is given by (extended) UML state machines called *Real-Time Statecharts* (RTSC) [16, 21, 22], which provide additional constructs to describe time-dependent behavior and information such as deadlines and worst case execution times (WCET). We introduce RTSC and their extension to a hybrid variant only informally here.

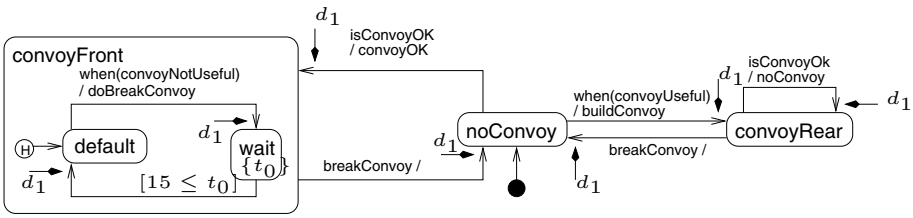


Fig. 4. Behavior of the Shuttle component

In Fig. 4 the internal behavior of the Shuttle component of Fig. 3 is defined by a RTSC. As an example for a typical real-time requirement a deadline interval d_1 is used to describe the state change from state noConvoy to state convoyFront which has to be finished within the given interval. Similarly, deadlines are defined to constrain the time an object may remain in a certain state. Transition guards may contain conditions which depend on the current value of a clock.

In general, clocks, time guards, and time invariants from timed automata [23, 24] are combined with expressive modeling concepts existing in UML state machines. RTSC are thus more expressive than plain or hierarchical timed automata models and permit emulation of limited UML state machine concepts for time such as after and when. In addition RTSC supports the definition of flexible

timer conditions that must held over a series of states. Buffering of timing events is not needed and does not exist in this approach. This avoids non predictable effects which may exist in the UML state machines as well as their extensions which use external timers [25].

Extending RTSC to specify continuous behavior is done similarly to the basic hybrid automata approaches like [26–29] by the possibility of assigning a configuration of controllers to a particular state for *Hybrid RTSCs* (HRTSCs).

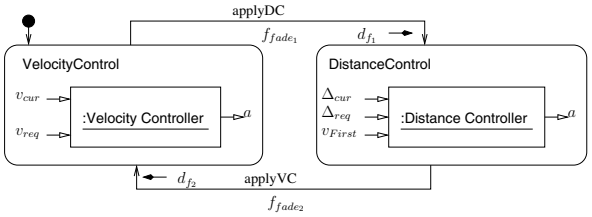


Fig. 5. Behavior of the AC component

An example of this is the hybrid behavior of an Acceleration Control (AC) component which is embedded into the Shuttle component. It consists of two discrete control modes which specify whether the shuttle is operating in velocity control mode or distance control mode respectively (see Fig. 5). Furthermore, it has continuous inputs and outputs. Depending on the active discrete mode, either the current and the required velocity are used as input, or the current and required distance to the front shuttle as well as the velocity of the first shuttle are used. The output a is the acceleration in both modes. In this example each configuration consists of one single feedback controller, while usually a configuration of subordinated blocks representing a number of (continuous) controllers might be assigned to each state.

Switching smoothly between different controllers requires the specification of an output *cross-fading function* (cf. [9]). In our example we have the fading functions f_{fade_1} and f_{fade_2} and a minimal and a maximal *fading duration* (d_{f_1} respectively d_{f_2}), which specify how the outputs of the two controllers have to be faded when changing the controller.

In the example depicted in Fig. 5, the state-dependent continuous behavior is specified by blocks assigned to the states VelocityControl and DistanceControl. If an RTSC contains hierarchies and thus comprises state configurations rather than single states, the assigned controller configuration is the union of all configurations assigned to the states of the current state configuration.

Component Structure with Reconfiguration. While the outlined static component model supports the specification of nested component structures, it does not cover the possibility of components having changing input/output interfaces depending on the current system state. As an example take the AC

component of Fig. 5. If you consider the shuttle component, then the AC component should be in state VelocityControl only if the shuttle is in state noConvoy or convoyFront. In this case the AC component requires two inputs. If the shuttle is in state convoyRear, however, the AC component should be in state DistanceControl and requires three inputs.

To cover nested components with changing input/output (i.e. OCM hierarchies with reconfiguration), we introduce an extension of the known concept of hybrid behavior that assigns a configuration of embedded (possibly hybrid) component instances to each state instead of control behavior only. The related HRTSC depicted in Fig. 6 extends Fig. 4 accordingly.

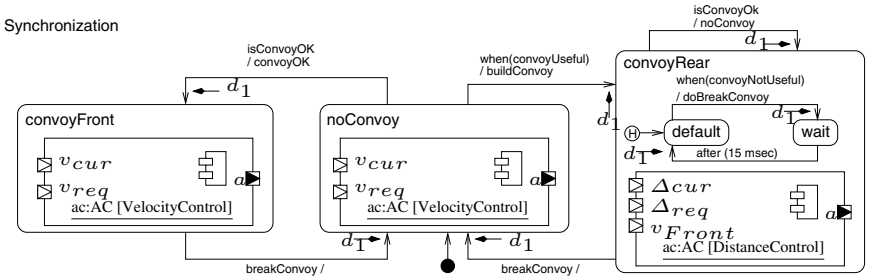


Fig. 6. Behavioral embedding

Fig. 6 depicts the orthogonal Synchronization state, whose sub-states embed different configurations, each consisting of one AC instance ac and its current mode and continuous interface. It is thus specified that ac has to be in mode DistanceControl when Synchronization is in state convoyRear. If Synchronization is in state noConvoy or convoyFront, ac has to be in mode VelocityControl. Consequently a state change within the orthogonal Synchronization state implies a mode change in its embedded ac component.

Referring to the example in Fig. 6, the internals of a component behavior, such as the AC component, need not be known in order to embed a component behavior specification into the HRTSC of its superior component, i.e. assign it to certain states. Internal in this case are the definitions of the controllers as given in Fig. 5. Rather, it is enough to specify an interface statechart for each component that defines the externally relevant behavior. Fig. 7 gives an example of the interface statechart of the AC component. Note that continuous ports only available on a subset of the modes become *optional ports* while those ports supported become *permanent ports* (cf. also Fig. 3).

The externally relevant behavior includes the definition of the different control modes, the modes' continuous inputs and outputs as well as the dependencies between outputs and inputs, and the deadline information for switches between the control modes. The specific control strategy employed in each mode, whether fading is required for a transition, the kind of fading function applied for a

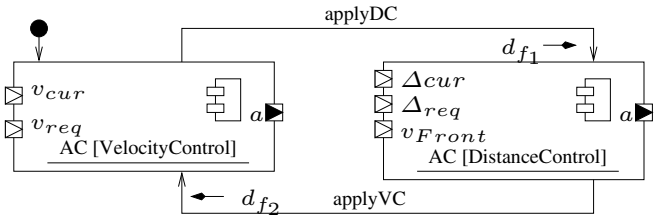


Fig. 7. Interface statechart of the AC component

transition, and which embedded components are active in each mode are implementation details not relevant to the external view of an interface statechart.

In each component may exist states that are related to potentially unsafe situations. Therefore, we refer to a component as locally safe if such states can be excluded for a given context. Also when the components do not work properly together or when the reconfiguration across multiple levels via the interface statecharts does result in any violations of the specified timing constraints, the assumptions of the composed components are not fulfilled and thus their local safety is no longer guaranteed. In the later considered models, in case of such an incompatibility the composed behavior will exhibit a deadlock.[1] Thus we can conclude that a hierarchical system of components is *safe* as long as the components are locally safe and the composition does not result in a deadlock.[2]

In our example the Shuttle builds a hierarchy of components, where each Shuttle instance contains a single supervised embedded component instance of type AC. The local safety guarantees that each local OCM is safe. The deadlock freedom guarantees that the complex reconfiguration fulfills all timing constraints.

Definition 1. *The overall safety of a hierarchical system is given if all component are locally safe and the overall behavior does not contain any deadlock.*

In our example we have the Shuttle agents with a single supervised embedded component instance of type AC. Within the shuttles the switching has to adhere to the timing constraints for cross-fading the outputs and the commitments concerning braking in different collaborations must be not in conflict. The deadlock freedom guarantees that the complex reconfiguration fulfills all timing constraints for the reconfiguration present at the different levels of the hierarchy.

The remaining part of the architecture, consisting of the cognitive operators and their interconnections, is additionally covered by related components that

[1] This concerns the usual definition of a deadlock but also so-called time stopping deadlocks. A time stopping deadlock means that a system cannot progress due to an inconsistency in the definition of timing constraints of transitions or states.

[2] We assume here that required non-local safety properties that relate to a number atomic components in a system are considered part of a local safety properties of a hierarchical element that contains all in the required safety property involved elements.

are connected with the safety-critical hierarchical core only via unsafe ports that decouple the core from the rest.

3.2 The Peer-to-Peer Coordination Model

Besides the hierarchical component structures and their hybrid behavior as addressed in the last section, MUML specifications can also capture peer-to-peer interaction of autonomous mechatronic systems (cf. Fig. 2). At the peer-to-peer level the interaction between components is specified in MUML by so-called coordination patterns. At this level no hybrid behavior exists anymore because communication between components is only based on discrete events and corresponding actions. Therefore these patterns can be described by a refinement of the loosely defined collaboration and pattern concepts in UML using RTSC to specify the behavior of roles and connectors.

Real-Time Coordination Pattern. Real-time coordination patterns allow to specify the interaction between multiple mechatronic agents using UML collaborations so that the behavior is rigorously defined. Therefore a real-time coordination pattern includes a description of the roles involved agents may play . The agents can interact only via these roles and connectors that connects them. Each role and connector in turn are specified by an RTSC that captures the behavior permitted and expected from each role as well as the communication medium.

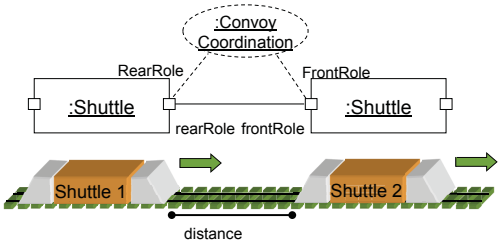


Fig. 8. Component Instance Diagram and Pattern Instance

The communication between two shuttles necessary to build a convoy is one such real-time coordination pattern. Fig. 8 shows a ConvoyCoordination pattern instance between two shuttles. It defines a drastically simplified protocol for building and breaking convoys based on two roles, namely the rear role and the front role (see Fig. 9).

Initially, both roles are in state noConvoy::default, which means that they specify the situation where a shuttle is not a member of a convoy. The rear role non-deterministically chooses whether to propose building a convoy or not. After choosing to propose a convoy, a message is sent to another shuttle, or rather its front role instantiation. The front role non-deterministically chooses to reject or to accept the proposal after at most 1000 msec. In the first case, both statecharts

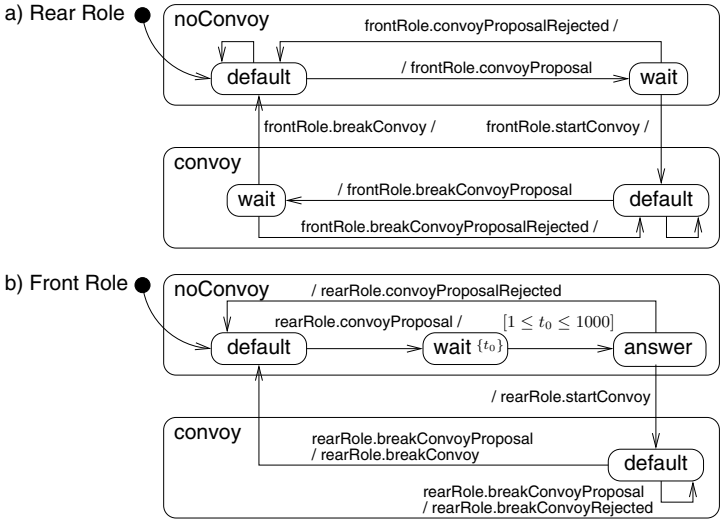


Fig. 9. RTSC of the RearRole role and the FrontRole role

revert to the noConvoy::default state. In the second case, both roles switch to the convoy::default state.

Eventually, the rear shuttle non-deterministically chooses to propose breaking the convoy and sends this proposal to the front shuttle. The front shuttle non-deterministically chooses to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies with an approval message and both roles switch into their respective noConvoy::default states.

The connector which represents the wireless network does not need to be specified by an explicit statechart specification here, but instead by its QoS characteristics such as throughput, maximal delay etc. in the form of connector attributes. In our example we assume that the connector forwards incoming signals with a delay of between 1 to 5 msec. The connector is unsafe in the sense that it might fail at any time so that we set our specific QoS characteristic reliable to false.

The specification of safety properties is given by declarative constraints which are defined using temporal logic using a state-based temporal extension of the Object Constraint Language (OCL) called RT-OCL [30]. As the examples in this paper only contain formulas in pure OCL, we further omit any details of RT-OCL here.

A safety property of this pattern is that a shuttle should only make an emergency brake when it is not taking the front position in a convoy. Using an atomic proposition CanBrakeFully, which specifies whether a shuttle can brake with full strength, the required safety property is that when an implementation of the rear role is in state convoy the atomic proposition CanBrakeFully must be true and

when an implementation of the front role is in state convoy the atomic proposition CanBrakeFully must be false. The following OCL role invariants ψ_1 and ψ_2 are used to describe these restrictions.[3]

```
context <comp> inv: <frontRole>.oclInState(convoy) implies not self.CanBrakeFully    (1)
```

```
context <comp> inv: <rearRole>.oclInState(convoy) implies self.CanBrakeFully        (2)
```

Atomic Agent. When defining the behavior of an agent like a shuttle using predefined patterns such as e.g. the ConvoyCoordination pattern mentioned above, the predefined role behavior has to be refined and synchronized. The following example illustrates this step.

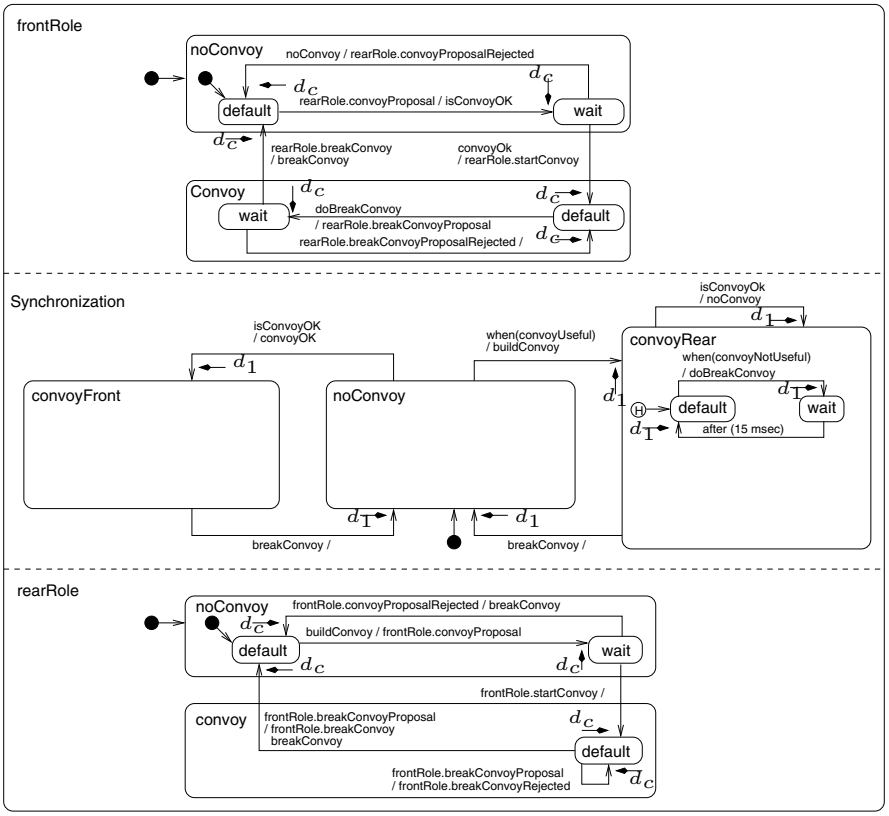


Fig. 10. Behavior of the Shuttle agent

[3] The context <comp> enclosed in angle brackets is employed here as a placeholder for the component which realizes the role via one of its ports.

Fig. 10 depicts the behavior of the Shuttle agent from Fig. 3. The HRTSC consists of three orthogonal states FrontRole, RearRole and Synchronization.

FrontRole and RearRole describe the port behavior. They are refinements of the role behaviors in Fig. 9 and specify in detail the communication that is required to build and to break convoys. Syntactical refinement rules or a special checking procedure, outlined later in Section 4, are used to ensure that the refinement does not invalidate any safety properties which have been verified for the (non-refined) pattern already. Basically, only the non-determinism possibly still existing in a RTSC defining a role is reduced by the refinement.

An additional internal HRTSC is used to specify the synchronization. In our example, Synchronization coordinates the communication and is responsible for initiating and breaking convoys. The three sub-states of Synchronization model whether the shuttle is in the convoy at the first position (convoyFront), at second position (convoyRear) or whether no convoy is built at all (noConvoy).

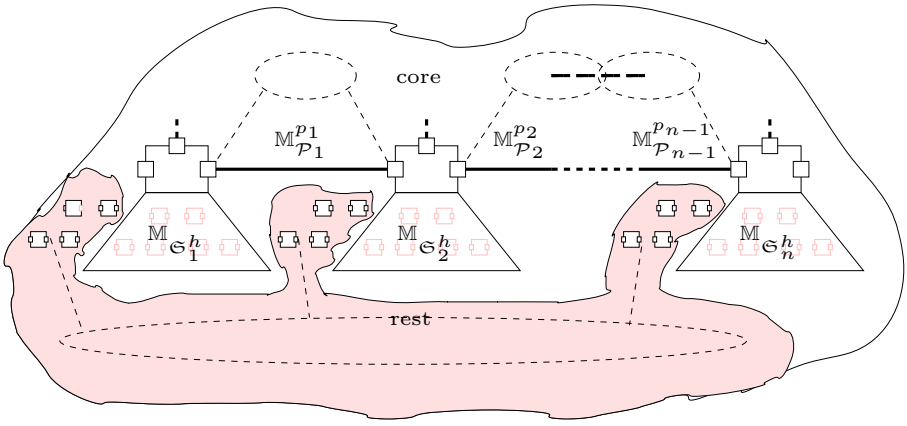


Fig. 11. Peer-to-peer composition of agents (upper part), hierarchies within the agents (middle part) and decomposition into a safety-critical core and a arbitrarily structured rest (bottom and middle part)

A system built by a set of atomic agents and pattern instances (Fig. 11 upper part) then describes the free peer-to-peer interaction of the agents by pattern instances.

Definition 2. *A peer-to-peer system is* safe *if for the behavior ensures that*

- *all agents/components are locally safe,*
- *no deadlock can occur,*
- *all RT-OCL constraints of pattern instances are fulfilled, and*
- *all OCL role invariants of agent instances are fulfilled.*

In our example we have the Shuttle agents connected via instances of the ConvoyCoordination pattern. The RT-OCL constraints guarantee that no collision is

possible for shuttles connected by ConvoyCoordination pattern instances. The OCL role invariants ensure that the agents behave consistently with the guarantees related to their roles, e.g. a shuttle will brake accordingly when the state of the role of the ConvoyCoordination pattern requires it. However, the pure peer-to-peer system does not cover the embedding of subordinated components such as the AC component.

Hierarchical Agents with Reflective Operator. In case of a hierarchical agent, besides refining and synchronizing the assigned role behavior, the reconfiguration of the embedded hybrid component hierarchy also has to be specified.

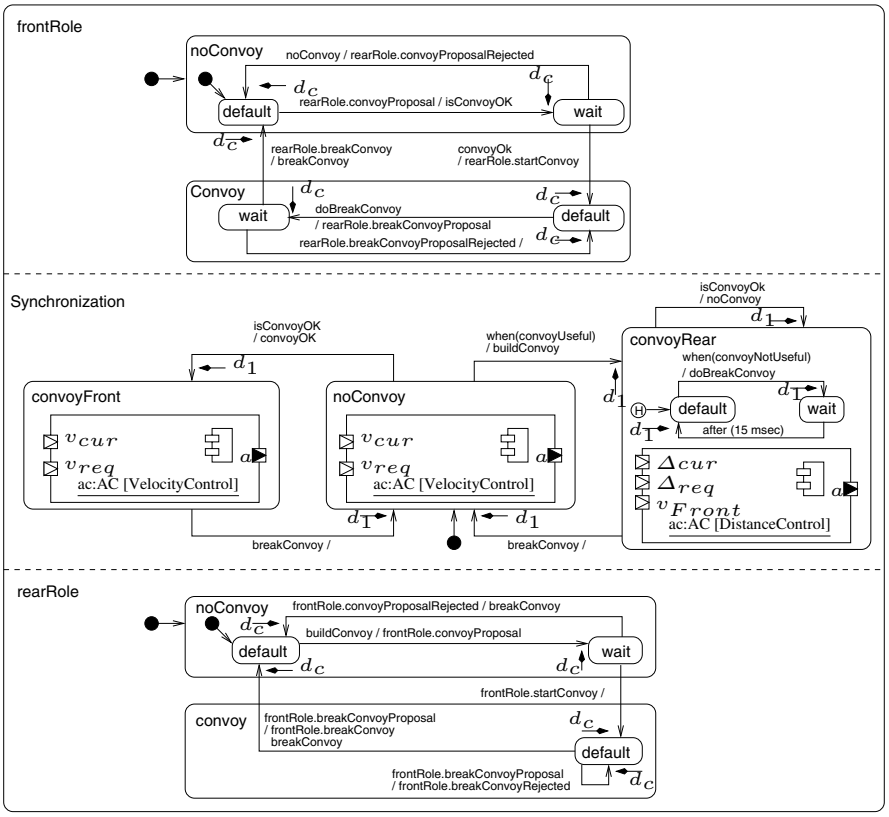


Fig. 12. Behavior of the Shuttle agent

An instance of the hybrid component type AC is assigned to the different three sub-states of Synchronization. In state convoyFront and state noConvoy the embedded controller is run in mode VelocityControl, but in state convoyRear mode DistanceControl is used to ensure a proper distance to the other shuttle (Fig. 12).

Our approach for the modeling of a hierarchy of OCMs, as depicted in Fig. 2 without cognitive operators, is based on the observation that it can be modeled as hierarchal agents that coordinate themselves by pattern instances. While the free peer-to-peer interaction of the top-level OCMs can be captured by pattern instances, a hierarchical system of configurable components can be used to cover the hierarchies of reflective operators and controllers (Fig. 11 middle part).

Hierarchical Agents with Cognitive Operator. However, the cognitive operators do not really fit into this picture as they do not fit into the hard real-time processing scheme established by the hierarchy of reflective operators but form their own, often less strict, structures. For the cognitive operators, verification is usually not feasible, or at least extremely expensive, due to their intelligent behavior. Their tight integration within the safety-critical and hard real-time part of the system is thus problematic. However, we can require the boundary ports to be *unsafe ports* to express that you cannot rely on the offered interaction. Therefore, such unsafe ports can be used to decouple the safety-critical parts of the system from those for which safe operation cannot be guaranteed such as the cognitive operators.

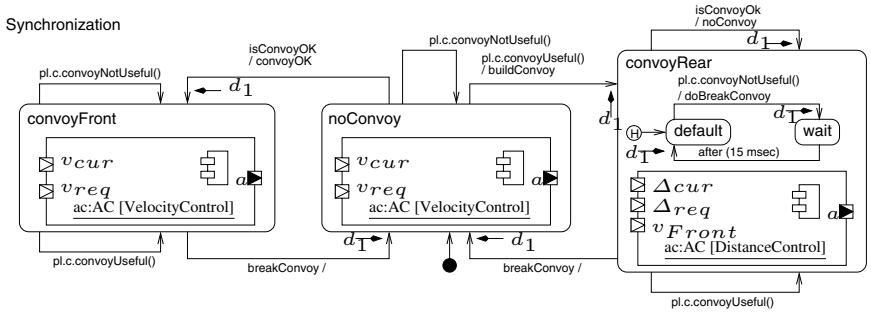


Fig. 13. Decoupling of the unreliable, soft real-time, planning component

In Fig. 13, the unsafe port pl.c connecting the soft real-time planning of the cognitive operator with the safety-critical, hard real-time processing of the reflective operator as depicted in Fig. 3) is used to steer proposing convoys. If the planning component suggests building a convoy with the rear shuttle, it indicates this by sending a pl.c.convoyUseful() message. If it deduces that a convoy with the rear shuttle should be broken up it sends pl.c.convoyNotUseful(). The depicted behavior is therefore able to handle both messages in any state so that erroneously sent messages of the planning component cannot result in unsafe shuttle behavior.

Our approach for the modeling of the safety-critical core of a hierarchy of OCMs as depicted in Fig. 2 employs hierarchical agents and pattern instances as outlined in Fig. 11. The remaining part of the architecture consisting of the cognitive operators and their interconnections is additionally covered by related

components that are connected with the safety-critical core only via unsafe ports as depicted in Fig. 11.

Definition 3. *A system with included core system is* safe *if the core behavior ensure that*

- *all agents/components are locally safe,*
- *no deadlock can occur,*
- *all RT-OCL constraints of pattern instances of the core are fulfilled, and*
- *all OCL role invariants of agent instances of the core are fulfilled.*

In our example we have the Shuttle agents connected via instances of the ConvoyCoordination pattern. For each Shuttle instance, a single supervised embedded component instance of type AC exists. The deadlock freedom guarantees that the complex reconfiguration fulfills all timing constraints. The RT-OCL constraints guarantee that no collision is possible for shuttles connected by ConvoyCoordination pattern instances. The OCL role invariants ensure that the agents behave consistently with the guarantees related to their roles, e.g. a shuttle will brake accordingly when the state of the role of the ConvoyCoordination pattern requires this.

4 Modular and Compositional Verification

This section outlines how the modular and compositional formal verification of self-optimizing systems developed with the MUML approach can guarantee safety. These results are based on the rigorous definitions for the employed concepts also provided in this section (please note that some more fundamental definitions and additionally required consistency and well-formedness conditions can be found in [3]).

The key aspect of our approach is that our notion of a consistent core system enables a modular and compositional verification where only the single component types and patterns with their interfaces resp. embeddings are considered. For hierarchal systems (including agent subsystems) we can exploit the modular structure to derive the safety of all included component instances only looking at the single component types as well as their interface and embeddings (see Section 4.1 and Theorem 1). A compositional scheme also allows the safety of all pattern instances to be ensured only by looking into the patterns and their roles and connectors as well as the conformance of all components with respect to the ports which are attached to the roles (see Section 4.2 and Theorem 2). Due to the manner in which the core is decoupled from the rest of the system via unsafe ports, we can further show that this result cannot be invalidated by the rest of the system (see Section 4.3 and Theorem 3). These separate results can be employed to guarantee that a complete system is safe via modular and compositional verification (see Section 4.3 and Corollary 1), essentially only by looking at the types and without considering the complete system or any larger subsystem with all its component and pattern instances explicitly. Consequently, existing model checking techniques can be employed as the usual state explosion due to parallel composition of multiple instances is avoided.

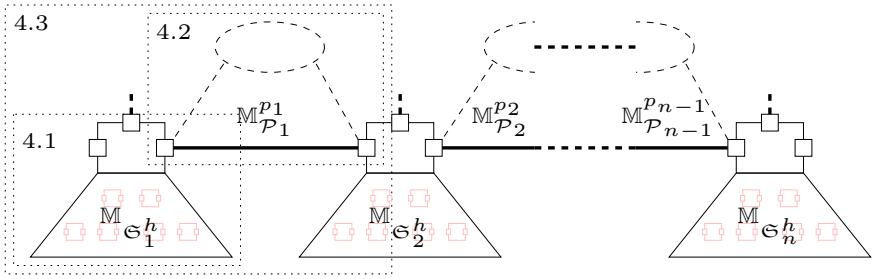


Fig. 14. Decomposition of the core system for the verification

4.1 Hierarchical Component Model

Syntax, Semantics and Safety. In the following we describe the resulting behavior using automata ($\mathbb{M}$) as well as their parallel composition ($\|$). The formal semantics are defined in [3]. We also refer to [3] for the additional required consistency and well-formedness conditions between the component model, structure, and behavior.

An OCL or RT-OCL property ϕ is well-defined for a behavior $\mathbb{M}$ if ϕ only refers to properties of the states of $\mathbb{M}$. In the following, we describe that an OCL or RT-OCL property ϕ holds for a given real-time behavior $\mathbb{M}$ by $\mathbb{M} \models \phi$. In addition, the special symbol δ is used to specify that a deadlock exists. We further restrict the considered RT-OCL properties to compositional ones that were preserved by the parallel composition ($\|$) if the composition result is deadlock free (c.f. [6,7]).

Basis for the hierarchical component model are *reconfigurable component types* defined as follows:

Definition 4. *A* reconfigurable component type $\mathcal{C} = (\mathcal{S}_\mathcal{C}, \mathbb{M}_\mathcal{C}, \phi_\mathcal{C})$ *is given by a mode-dependent internal structure* $\mathcal{S}_\mathcal{C}$ *for a mode set* L*, an internal behavior* $\mathbb{M}_\mathcal{C}$ *with mode set* L*, and a local safety property* $\phi_\mathcal{C}$*.*

A *mode-dependent internal structure* for a given mode set L is a tuple $\mathcal{E} = (\mathcal{I}_\mathcal{S}, \mathcal{E}_\mathcal{S}, map_\mathcal{S})$, where $\mathcal{E}_\mathcal{S}$ is a function describing the *mode-dependent embedded components* for a given mode set L (it assigns to each state $l \in L$ a set of embeddings that are pairs of the form $(o, (\mathbb{M}, l))$ where o is an occurrence name and $(\mathbb{M}, l)$ is a pair consisting of an interface statechart $\mathbb{M}$ and one of its modes l). $\mathcal{I}_\mathcal{S}$ is a function describing the *mode-dependent interface* for a given mode set L (it assigns an interface consisting of a set of pairs of unique port names and port declarations to each state $l \in L$), and $map_\mathcal{S}$ is a *mode-dependent mapping* for a given mode set L (it assigns a mapping describing the connectors between ports in each mode to each state $l \in L$). More details can be found in [3].

The behavior $\mathbb{M}_\mathcal{C}$ also includes the forwarding behavior related to each mode in L which ensures that signals from the embedded components are routed as specified in the HRTSC (see [3]).

A reconfigurable hierarchical subsystem consisting of a number of reconfigurable components as depicted in Fig. 11 (middle) is then constructed as follows:

Definition 5. *A* hierarchical subsystem with reconfiguration $\mathfrak{S}^h$ *is a tuple* $(O_{\mathfrak{S}^h}, c_{\mathfrak{S}^h})$ *with* $O_{\mathfrak{S}^h} \subseteq \wp(\mathcal{N}_{\mathcal{C}}^+)$ *a set of instance names and* $c_{\mathfrak{S}^h}$ *a function which maps each instance* $o \in O_{\mathfrak{S}^h}$ *to a related reconfigurable component type.*

The *behavior* and *safety property* of a such hierarchical subsystem with reconfiguration $\mathfrak{S}^h = (O_{\mathfrak{S}^h}, c_{\mathfrak{S}^h})$ is then given by

$$\mathbb{M}_{\mathfrak{S}^h} := \parallel_{o \in O_{\mathfrak{S}^h}} \mathbb{M}^o_{c_{\mathfrak{S}^h}(o)} \qquad \phi_{\mathfrak{S}^h} := \bigwedge_{o \in O_{\mathfrak{S}^h}} \phi^o_{c_{\mathfrak{S}^h}(o)}.$$

For such a hierarchical subsystem with reconfiguration we have to ensure that all components are locally safe and have to exclude that the interaction or timing constraints invalidates the local safety of the components. Therefore, we define it as safe if its behavior guarantees the local safety of the components and is deadlock free (by providing a formal version for Definition 1).

Definition 6. *A hierarchical subsystem with reconfiguration* $\mathfrak{S}^h$ *is* safe *if its behavior* $\mathbb{M}_{\mathfrak{S}^h}$ *is well-formed, ensures local safety (*$\mathbb{M}_{\mathfrak{S}^h} \models \phi_{\mathfrak{S}^h}$*), and deadlock free (*$\mathbb{M}_{\mathfrak{S}^h} \models \neg\delta$*).*

Modular Verification. We exploit the well-defined hierarchy of an agent to prove local safety and deadlock freedom. Thus, we first look at the atomic components and the bottom of the hierarchies, then the embedding steps, and finally demonstrate that for the required behavioral consistency of the whole hierarchy, these two local checks are sufficient. To denote the behavior that results when restricting a hybrid reconfiguration automata to the real-time behavior and clocks we use an operator $RT()$ (see [3]).

Atomic Component Types. The locally safe operation of a component type $\mathcal{C}$ requires that component behavior $\mathbb{M}_{\mathcal{C}}$ itself cannot result in a deadlock. A component without embedding is therefore *locally safe* if

$$\mathbb{M}_{\mathcal{C}} \text{ well-formed} \quad \wedge \quad RT(\mathbb{M}_{\mathcal{C}}) \models \phi_{\mathcal{C}} \wedge \neg\delta. \tag{3}$$

To further ensure that, in a given system, all embedded occurrences represented by their interface statecharts $\mathbb{M}^i_I$ are *behaviorally consistent* with the related embedded components with respect to behavior $\mathbb{M}^o_{\mathcal{C}}$, we have to ensure that real-time refinement holds:

$$RT(\mathbb{M}_{\mathcal{C}}) \sqsubseteq_{RT} \mathbb{M}^i_I. \tag{4}$$

Model checking is employed to fully automate the checking of condition 3. In many cases the required refinement condition 4 means the safe transfer of timing

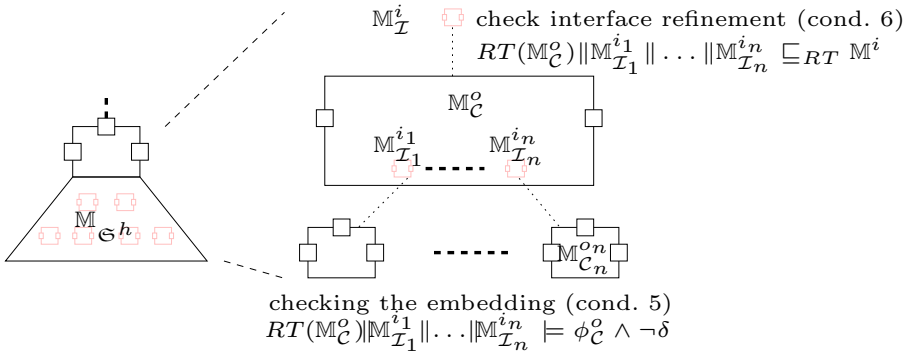


Fig. 15. Hierarchical verification via interface abstraction and component-wise checks

constraints from one level to the next one, and can be guaranteed following syntactical refinement rules (cf. [9]).

In more complex cases, model checking also has to be employed (cf. [31–33]).[4]

Hierarchical Component Types. In a strict hierarchical system the continuous model for a state of the system can become *undefined* if the resulting continuous equations contain a cycle. Refinement guarantees that any dependency between an input and output in the behavior of embedded component occurrences $\mathbb{M}^{o1}_{\mathcal{C}_1}, \ldots, \mathbb{M}^{on}_{\mathcal{C}_n}$ of a component is also present in their interface statecharts $\mathbb{M}^{i_1}_{I_1}, \ldots, \mathbb{M}^{i_n}_{I_n}$. Therefore, checking that the interface statechart combined with the embedding HRTSC $\mathbb{M}^o_{\mathcal{C}}$ is well-formed and sufficient to exclude cycles in the resulting continuous models.

Additionally, the locally safe synchronization of the fading-durations in the different components has to be ensured. We need to therefore ensure that the composition of the component behavior with the embedded interface statecharts cannot result in a deadlock.

A component with embedding is therefore only *locally safe* for the accordingly relabeled embedded interface statecharts $\mathbb{M}^{i_1}_{I_1}, \ldots, \mathbb{M}^{i_n}_{I_n}$ if

$$\mathbb{M}^o_{\mathcal{C}} \| \mathbb{M}^{i_1}_{I_1} \| \ldots \| \mathbb{M}^{i_n}_{I_n} \text{ well-formed} \wedge RT(\mathbb{M}^o_{\mathcal{C}}) \| \mathbb{M}^{i_1}_{I_1} \| \ldots \| \mathbb{M}^{i_n}_{I_n} \models \phi^o_{\mathcal{C}} \wedge \neg\delta. \quad (5)$$

As with the atomic case in condition 4 we also have to show that the interface statechart $\mathbb{M}^i_I$ alone is a real-time abstraction of the HRTSC $\mathbb{M}^o_{\mathcal{C}}$ combined with the interface statecharts of all subcomponents. For *behavioral consistency* it is necessary that the real-time abstraction of the component behavior in form of the interface statechart $\mathbb{M}^i_I$ in fact refines the behavior which results when we

[4] As with the general form of hybrid systems considered here reachability is undecidable [34] and we cannot expect to find an automatic solution for the general problem. However, the developed techniques cover all relevant cases in practice for MUML.

compose the component behavior $\mathbb{M}^o_{\mathcal{C}}$ with all accordingly relabeled embedded interface statecharts $\mathbb{M}^{i_1}_{I_1}, \ldots, \mathbb{M}^{i_n}_{I_n}$:

$$RT(\mathbb{M}_{\mathcal{C}}) \| \mathbb{M}^{i_1}_{I_1} \| \ldots \| \mathbb{M}^{i_n}_{I_n} \sqsubseteq_{RT} \mathbb{M}^i_I. \tag{6}$$

Model checking is employed to fully automate the checking of condition 5 like in case of checking condition 3. Therefore, the interface statecharts are considered in addition to the component behavior.

In case of simple interface statecharts condition 6 can be checked at the syntactical level. It only has to be considered whether each transition in the HRTSC and the related transitions in the interface statecharts of the aggregated subcomponents are consistent (cf. [9]).

Assume the example in Fig. 12 and 6 which specifies that a change from state noConvoy to convoyRear has to be finished after 200 msec and that this change implies a change of the embedded AC component from VelocityControl to DistanceControl. Then, in Fig. 7, the minimal fading duration may not be above 200 msec.

Besides this purely syntactical check for simple interface statecharts, the embedding of more general notions of interface statecharts can be addressed using model checking (cf. [31–33]).

Thus, for components with embedded components, either syntactical checks or more advanced model checking techniques can be employed to check condition 6.

Hierarchical Systems. Checking local safety for all component types embedded in one hierarchal component guarantees that the whole hierarchy cannot become deadlocked. The following theorem proves that the local safety and behavioral consistency, which has been checked for each embedding, is sufficient to ensured that the behavior, which results when the component and all its direct subcomponents $\mathbb{M}^{o_1}_{\mathcal{C}_1}, \ldots, \mathbb{M}^{o_n}_{\mathcal{C}_n}$ are considered and is a refinement of the HRTSC $\mathbb{M}^o_{\mathcal{C}}$ ($\sqsubseteq_{RT}$).

Theorem 1. *For a consistent hierarchical subsystem $\mathfrak{S}^h = (O^c_{\mathfrak{S}^h}, c_{\mathfrak{S}^h})$ with unique top-level component $o \in O_c \cap \mathcal{N}_{\mathcal{C}}$, only locally safe embedded components $o_1, \ldots, o_n$ ($O_c = \{o, o_1, \ldots, o_n\}$; see condition 3 and 5) and where all embeddings are behaviorally consistent (see condition 4 or 6) holds for the real-time abstraction $RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)})$ of the HRTSC $\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)}$:*

$$RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)} \| \mathbb{M}^{o_1}_{c_{\mathfrak{S}^h}(o_1)} \| \ldots \| \mathbb{M}^{o_n}_{c_{\mathfrak{S}^h}(o_n)}) \sqsubseteq_{RT} RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)}) \text{ and} \tag{7}$$

$$RT(\mathbb{M}^o_{c_{\mathfrak{S}^h}(o)} \| \mathbb{M}^{o_1}_{c_{\mathfrak{S}^h}(o_1)} \| \ldots \| \mathbb{M}^{o_n}_{c_{\mathfrak{S}^h}(o_n)}) \models \phi^o_{c_{\mathfrak{S}^h}(o)} \wedge \neg\delta \tag{8}$$

Proof. (sketch) We can show the required result by induction over the depth over the hierarchical component structure using condition 4 and condition 6, substituting the component step-wise for the component interfaces for the whole subsystem beneath. For each such substitution we can conclude from the condition for the type that the same relation holds for all specific instances. Thus, the top-level component is refined by the whole hierarchical component concerning its

real-time behavior, based on the fact that $\sqsubseteq_{RT}$ is a precongruence for $\|$, and that we only require a finite number of substitution steps. Furthermore, $\phi^o_{c_{\in^h(o)}} \wedge \neg\delta$ is guaranteed as the safety properties in $\phi^o_{c_{\in^h(o)}}$ are compositional and $\neg\delta$ is guaranteed as deadlock freedom has been checked for the top component.

4.2 Peer-to-Peer Coordination Model

Syntax, Semantics and Safety. The main ingredients of peer-to-peer systems are the real-time coordination patterns and agents. We will also cover the integration of the pure peer-to-peer scheme with the hierarchies present in the agents as well as the decoupled cognitive operators via unsafe ports.

Real-Time Coordination Pattern. Channel delays and reliability are both of crucial importance to the real-time coordination patterns. We address them explicitly by giving one RTSC for each connector. A real-time pattern is then formally defined as follows:

Definition 7. *A* real-time coordination pattern *(collaboration type) $\mathcal{P}$ is a tuple $(R_\mathcal{P}, \Psi_\mathcal{P}, \phi_\mathcal{P}, \mathcal{C}_\mathcal{P})$ with $R_\mathcal{P}$ a set of roles $(r_i, \mathbb{M}^{r_i}_\mathcal{P})$ for $r_i \in \mathcal{N}_\mathcal{R}$ a role name and $\mathbb{M}^{r_i}_\mathcal{P}$ a role behavior in the form of a RTSC, a set $\Psi_\mathcal{P}$ of OCL invariants $\psi_1, \ldots, \psi_k$ for each role, the RT-OCL pattern constraint $\phi_\mathcal{P}$, and the atomic component type $\mathcal{C}_\mathcal{P}$ representing the connectors.*[5]

For connector instance (collaboration instances) $o \in \mathcal{N}^+_\mathcal{C}$ of the pattern $\mathcal{P}$ we refer to the pattern constraints as $\phi^o_\mathcal{P}$ and to the related *behavior* as $\mathbb{M}^o_\mathcal{P}$ which is derived from $\mathbb{M}_\mathcal{P}$ for $\mathcal{C}_\mathcal{P} = (\mathcal{S}_\mathcal{P}, \mathbb{M}_\mathcal{P})$ by renaming the ports accordingly.

Agents. To capture agents which realize the peer-to-peer interaction of autonomous mechatronic systems as well as the embedding of complex hierarchies (cf. Fig. 2), we define an agent as a special component as follows:

Definition 8. *An* agent *is a reconfigurable hierarchical component type $\mathcal{A} = (\mathcal{S}_\mathcal{A}, \mathbb{M}_\mathcal{A}, \phi_\mathcal{A})$ with $\mathcal{S}_\mathcal{A} = (\mathcal{I}_\mathcal{A}, \mathcal{E}_\mathcal{A}, map_\mathcal{A})$ such that the interface is mode-independent ($\mathcal{I}_\mathcal{A}$ is constant) and the internal behavior $\mathbb{M}_\mathcal{A}$ is decomposed into $\mathbb{M}^s_\mathcal{A} \| \mathbb{M}^{p_1}_\mathcal{A} \| \ldots \| \mathbb{M}^{p_h}_\mathcal{A}$ where $\mathbb{M}^{p_i}_\mathcal{A}$ refines the port behavior $\mathbb{M}^{p_i}_i$ for $\mathcal{I}_\mathcal{A} = \{(p_1, \mathbb{M}_1), \ldots, (p_h, \mathbb{M}_h)\}$.*

The ports of an agent are assumed either to be not considered within our behavioral model (unsafe ports) or related to a specific role of a pattern instance (regular port). The set of associated OCL role invariants $\psi_1, \ldots, \psi_h$ of $\mathcal{A}$ by $\Psi_\mathcal{A}$ and the resulting overall component OCL role invariant $\psi_\mathcal{A}$ is therefore derived by combining the related OCL role invariants $(\psi_1 \wedge \cdots \wedge \psi_h)$. For agent instances $o \in \mathcal{N}^+_\mathcal{C}$ we refer to the component OCL role invariant as $\psi^o_\mathcal{A}$. A hierarchical system with an agent as top level component type is further named *agent subsystem*.

[5] A real-time pattern is *consistent* if the roles cover the complete interaction which is possible via the component representing the related connectors (see [3]).

Peer-to-Peer System. Peer-to-peer systems are built by composing atomic agents via pattern instances as depicted in Fig. 11. The composition of atomic agents via pattern instances depicted in Fig. 11 as upper part of the core can be formally defined as follows:

Definition 9. *A peer-to-peer system* $\mathfrak{S}^p$ *is a tuple* $(O^c_{\mathfrak{S}^p}, O^p_{\mathfrak{S}^p}, c_{\mathfrak{S}^p}, p_{\mathfrak{S}^p}, map_{\mathfrak{S}^p})$ *with* $O^c_{\mathfrak{S}^p} \subseteq \wp(\mathcal{N}_\mathcal{C})$ *a set of instance names of atomic agents* $c_1, \ldots, c_n$, $O^p_{\mathfrak{S}^p} \subseteq \wp(\mathcal{N}_\mathcal{C})$ *a set of instance names of the connector components representing patterns* $p_1, \ldots, p_m$ *with* $O^c_{\mathfrak{S}^p} \cap O^p_{\mathfrak{S}^p} = \emptyset$, $c_{\mathfrak{S}^p}$ *a function which maps to each instance* $c_i \in O^c_{\mathfrak{S}^p}$ *a related component type,* $p_{\mathfrak{S}^p}$ *a function which maps to each instance* $p_j \in O^p_{\mathfrak{S}^p}$ *the related pattern, and* $map_{\mathfrak{S}^p} : (O_p.\mathcal{N}_\mathcal{Q}) \to ((O_c \cap \mathcal{N}_\mathcal{C}).\mathcal{N}_\mathcal{Q})$ *a bijective mapping which connects ports of components representing the pattern connectors with the ports of the root components of agents.*

For such a peer-to-peer system $\mathfrak{S}^p = (O^c_{\mathfrak{S}^p}, O^p_{\mathfrak{S}^p}, c_{\mathfrak{S}^p}, p_{\mathfrak{S}^p}, map_{\mathfrak{S}^p})$ we have to combine the behavior related to the agent instances and pattern instances. The overall behavior and safety property of a system as depicted in Fig. 11 as upper part of the core is given by

$$\mathbb{M}_{\mathfrak{S}^p} := \left(\parallel_{c \in O^c_{\mathfrak{S}^p}} \mathbb{M}^c_{c_{\mathfrak{S}^p}(c)} \right) \parallel \left(\parallel_{p \in O^p_{\mathfrak{S}^p}} \mathbb{M}^p_{p_{\mathfrak{S}^p}(p)} \right) \qquad \phi_{\mathfrak{S}^p} := \bigwedge_{c \in O^c_{\mathfrak{S}^p}} \phi^c_{c_{\mathfrak{S}^p}(c)}.$$

The safety of a peer-to-peer system can then be formally defined referring to the overall behavior, pattern constraints, and component invariants as follows (by providing a formal version for Definition 2):

Definition 10. *A peer-to-peer system* $\mathfrak{S}^p = (O^c_{\mathfrak{S}^p}, O^p_{\mathfrak{S}^p}, c_{\mathfrak{S}^p}, p_{\mathfrak{S}^p}, map_{\mathfrak{S}^p})$ *is* safe *if the following conditions are fulfilled by the behavior* $\mathbb{M}_{\mathfrak{S}^p}$*:*

- *all agents are locally safe and no deadlock occurs:* $\mathbb{M}_{\mathfrak{S}^p} \models \phi_{\mathfrak{S}^p} \wedge \neg\delta$ (9)
- *All RT-OCL constraints of patterns are fulfilled:* $\mathbb{M}_{\mathfrak{S}^p} \models \wedge_{o \in O^p_{\mathfrak{S}^p}} \phi^o$ (10)
- *All OCL role invariants of the agents are fulfilled:* $\mathbb{M}_{\mathfrak{S}^p} \models \wedge_{o \in O^c_{\mathfrak{S}^p}} \psi^o$. (11)

Compositional Verification. To guarantee that the peer-to-peer real-time coordination of the whole system is safe, we have to look locally at the patterns and the role refinement by the agents (cf. Fig. 16) before we can compose these results for the peer-to-peer coordination.

Pattern Verification. We verify whether the behavioral requirement in form of safety properties specified by means of RT-OCL hold for a real-time pattern. If the requirement is fulfilled, the pattern is locally safe. Formally, a real-time pattern $\mathcal{P} = (R_\mathcal{P}, \Psi_\mathcal{P}, \phi_\mathcal{P}, \mathcal{C}_\mathcal{P})$ with a set R of roles with a name and a RTSCs for

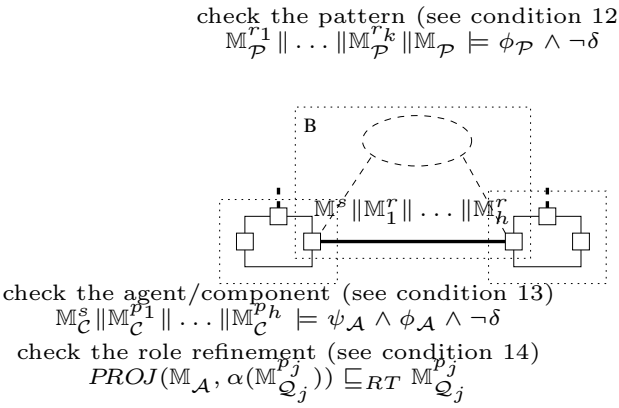


Fig. 16. Verification of peer-to-peer structures via patterns and role refinement

each role $(r_1, \mathbb{M}^{r_1}_{\mathcal{P}}), \ldots, (r_k, \mathbb{M}^{r_k}_{\mathcal{P}})$ and behavior $\mathbb{M}_{\mathcal{P}}$ for the connector component $\mathcal{C}_{\mathcal{P}} = (\mathcal{S}_{\mathcal{P}}, \mathbb{M}_{\mathcal{P}})$ is a *locally safe* real-time pattern if:

$$\mathbb{M}^{r_1}_{\mathcal{P}} \| \ldots \| \mathbb{M}^{r_k}_{\mathcal{P}} \| \mathbb{M}_{\mathcal{P}} \models \phi_{\mathcal{P}} \wedge \neg\delta. \tag{12}$$

The behavior $\mathbb{M}^{r_1}_{\mathcal{P}} \| \ldots \| \mathbb{M}^{r_k}_{\mathcal{P}} \| \mathbb{M}_{\mathcal{P}}$ is supposed to be a closed real-time behavior and can thus be verified using a real-time model checker for RTSC by checking whether the constraint $\phi \wedge \neg\delta$ holds.

In our example we generate model checker input from the RTSC for FrontRole, RearRole and an additional RTSC for the implicitly defined connector.

Agent Verification. Besides the patterns also the agents have to be verified. We have to verify whether the agent behavior respects the role RTSC and the role invariants defined as local safety of the agent.

An agent $\mathcal{A} = (\mathcal{S}_{\mathcal{A}}, \mathbb{M}_{\mathcal{A}}, \phi_{\mathcal{A}})$ with internal behavior $\mathbb{M}_{\mathcal{A}}$ can be decomposed into $\mathbb{M}^{s}_{\mathcal{C}} \| \mathbb{M}^{p_1}_{\mathcal{C}_1} \| \ldots \| \mathbb{M}^{p_h}_{\mathcal{C}_h}$. The RTSCs $\mathbb{M}^{p_1}_{\mathcal{C}_1}, \ldots, \mathbb{M}^{p_h}_{\mathcal{C}_h}$ have to refine the port behavior $\mathbb{M}^{p_1}_{\mathcal{Q}_1}, \ldots, \mathbb{M}^{p_h}_{\mathcal{Q}_h}$ for $\mathcal{I}_{\mathcal{A}} = \{(p_1, \mathbb{M}_{\mathcal{Q}_1}), \ldots, (p_h, \mathbb{M}_{\mathcal{Q}_h})\}$ and the HRTSC $\mathbb{M}^{s}_{\mathcal{C}}$ describes the component internal synchronization, and the reconfiguration and embedding of subordinated hybrid reconfigurable components. Such an agent with agent OCL role invariant $\psi_{\mathcal{A}}$ the is *locally safe* if:

$$\mathbb{M}^{s}_{\mathcal{C}} \| \mathbb{M}^{p_1}_{\mathcal{C}_1} \| \ldots \| \mathbb{M}^{p_h}_{\mathcal{C}_h} \models \psi_{\mathcal{A}} \wedge \phi_{\mathcal{A}} \wedge \neg\delta \tag{13}$$

Using the related RTSC $RT(\mathbb{M}^{s}_{\mathcal{C}})$ instead of the HRTSC $\mathbb{M}^{s}_{\mathcal{C}}$ we can use a real-time model checker to prove $\psi_{\mathcal{A}} \wedge \phi_{\mathcal{A}} \wedge \neg\delta$. As $\psi_{\mathcal{A}} \wedge \phi_{\mathcal{A}} \wedge \neg\delta$ does not refer to any continuous variables which are not clock variables, the verification result for $RT(\mathbb{M}^{s}_{\mathcal{C}} \| \mathbb{M}^{p_1}_{\mathcal{C}_1} \| \ldots \| \mathbb{M}^{p_h}_{\mathcal{C}_h})$ also holds for $\mathbb{M}^{s}_{\mathcal{C}} \| \mathbb{M}^{p_1}_{\mathcal{C}_1} \| \ldots \| \mathbb{M}^{p_h}_{\mathcal{C}_h}$ if the embedding is safe (cf. Section 4.1). Note that, as $RT(\mathbb{M}^{s}_{\mathcal{C}}) \| \mathbb{M}^{p_1}_{\mathcal{C}_1} \| \ldots \| \mathbb{M}^{p_h}_{\mathcal{C}_h}$ is an open model, we assume the erratic but guaranteed execution of external signals when performing the model checking as outlined in [14].

In our example the invariant for the shuttle component is automatically derived from the role invariants ψ_1 and ψ_2 (see constraints (1) and (2)). In the resulting invariant, frontRole and rearRole are now specific names for navigation to the associated ports. The same abstract shuttle property CanBrakeFully is replaced by the or-combination of all states of the synchronization chart which fulfill them: not self.CanBrakeFully equals Synchronization::convoyFront) and self.CanBrakeFully equals Synchronization::convoyFront or Synchronization::convoyyRear or shorter not Synchronization::convoyFront).

```
context Shuttle inv:
 (oclInState(frontRole::convoy) implies oclInState(Synchronization::convoyFront)) and
 (oclInState(rearRole::convoy) implies not oclInState(Synchronization::convoyFront))
```

Regular Ports. Each port RTSC M_j^r of the agent $\mathcal{A}$ has to refine the port behavior $\mathbb{M}_{\mathcal{Q}_j}^{p_j}$ for $\mathcal{I}_{\mathcal{A}} = \{(p_1, \mathbb{M}_1), \ldots, (p_h, \mathbb{M}_h)\}$ which is equal to the connected pattern role behavior. The whole agent behavior $\mathbb{M}_{\mathcal{A}}$ restricted to the interface of the port $(p_j, \mathbb{M}_j)$ has to result in such a refinement.

$$PROJ(\mathbb{M}_{\mathcal{A}}, \alpha(\mathbb{M}_{\mathcal{Q}_j}^{p_j})) \sqsubseteq_{RT} \mathbb{M}_{\mathcal{Q}_j}^{p_j}. \tag{14}$$

where $PROJ(\mathbb{M}, A)$ denotes the automaton which results when all transitions with input and output signals not present in A are replaced by non-deterministic ones (cf. [14]) for an automaton $\mathbb{M}$ and a given set of labels A.

To ensure that $\mathbb{M}_{\mathcal{A}}$ refines each of the role protocols associated to its ports, we propose the use of syntactical refinement rules which ensure $PROJ(\mathbb{M}_{\mathcal{A}}, \alpha(\mathbb{M}_{\mathcal{Q}_j}^{p_j})) \sqsubseteq_{RT} \mathbb{M}_{\mathcal{Q}_j}^{p_j}$. Requiring disjoint signal labels and checking in addition $\mathbb{M}_{\mathcal{A}} \models \neg\delta$, we can then ensure that condition 14 holds. Alternatively, model checking can be employed to also fully automate this task (cf. [31–33]).

The RTSC in Fig. 12 is a refinement of the roles from Fig. 9. Consequently, it needs to be ensured that the embedding of AC only refines the specified real-time behavior from Fig. 12 and does not add additional behavior, or be in conflict with the real-time specification of this super-ordinated component.

Unsafe Ports. In order to be able to verify partial systems, we have initially introduced the classification *unsafe* ports. For these unsafe ports proper decoupling but no refinement has to be checked to ensure safety. The idea includes two steps. (1) When checking condition 13 for the component type that has an unsafe port in the case of either a top level or embedded component instance, we simply consider the transitions that interact with an unsafe port to occur erratic and non-urgent. (2) An additional verification step proves that the unsafe port cannot block the component behavior. We therefore use a function *NDET* to transform the RTSC of the port into another one where all external communication is replaced by purely erratic non-deterministic behavior.

To ensure safety we then have to check that the unsafe role RTSC $\mathbb{M}_{\mathcal{Q}_j}^{p_j}$ transformed by *NDET* is deadlock free so that the component can never be blocked via this port.

$$NDET(\mathbb{M}_{\mathcal{Q}_j}^{p_j}) \models \neg\delta \tag{15}$$

In our example the *decoupled interaction* designed in Fig. 13 results in an unsafe port. The resulting check transforms the role RTSC using *NDET* and then checks the resulting RTSC for deadlocks.

Peer to Peer Model Verification. These separate results can be composed using a compositional reasoning scheme to conclude that the peer-to-peer coordination is safe (as outlined in [7] for the case without unsafe ports).

Theorem 2. *A consistent* peer-to-peer system $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is* safe *if*

- *all patterns are locally safe (condition 12),*
- *all agents/components are locally safe (condition 13),*
- *all ports are behavioral consistent (condition 14), and*
- *all unsafe ports are behavioral consistent (condition 15).*

Proof. (sketch) As we restricted the RT-OCL constraints to compositional properties and only considered the real-time behavior of the top-level components, we can use the border built by the ports and roles to also prove the constraints $\phi_{\mathcal{P}}$ and invariants $\psi_{\mathcal{C}_j}$ compositionally. We therefore use the local checks for the real-time patterns and top-level hybrid components and the refinement $\mathbb{M}_{\mathcal{A}} \sqsubseteq_{RT} \mathbb{M}^{p_1}_{\mathcal{Q}_1} \| \ldots \| \mathbb{M}^{p_h}_{\mathcal{Q}_h}$ (cf. [7]). The advantage of the compositional approach is that it permits us to verify condition 16, 17, 18 and 19 without building the state space for $\mathbb{M}_{\mathfrak{S}^c}$. Instead, only the consistency of the overall system, the local safety for all patterns, components, and the proper behavioral consistency of the ports concerning the fulfilled roles has to be ensured.

If unsafe ports are also present in the safety-critical subsystem, the check $NDET(\mathbb{M}^{p_j}_{\mathcal{Q}_j}) \models \neg\delta$ guarantees that the remaining uncovered environment cannot invalidate the result which has been achieved for the verified ones, even though they may not conform to behavior specified by the pattern roles.[6]

For the peer-to-peer interaction, we employ model checking to check conditions 12, 13, 14, and 15. The compositional reasoning sketched in Theorem 2 proves that these local checks result in guarantees for the overall system.

It the next section we will extend this result to systems with hierarchical embedding.

4.3 Overall Model

Syntax, Semantics and Safety. Our approach for the modeling of the safety-critical core of a hierarchy of OCM, as depicted in Fig. 2, is based on the observation that it can be formally defined by a set of hierarchical agents and pattern instances (Fig. 11). While the free peer-to-peer interaction of the top-level OCMs can be captured by pattern instances, a hierarchical system of configurable components can be used to cover the hierarchies of reflective operators and controllers.

[6] We assume that no invariants and constraints for the un-verified patterns and components exist and the related elements in the formal model are set to true.

The outlined composition of agents via pattern instances to build the safety-critical core can be formally defined by combining Definition 5 and 9 as follows:

Definition 11. *A* core system $\mathfrak{S}^c$ *is a tuple* $(O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *with* $O^c_{\mathfrak{S}^c} \subseteq \wp(\mathcal{N}^+_{\mathcal{C}})$ *a set of n names of agents relating to the hierarchial systems* $(O^c_{\mathfrak{S}^c} = O^c_1 \uplus \cdots \uplus O^c_n$ *and all* $\mathfrak{S}^h_i = (O^c_i, c_{\mathfrak{S}^c}|_{O^c_i})$ *are hierarchical systems),* $O^p_{\mathfrak{S}^c} \subseteq \wp(\mathcal{N}_{\mathcal{C}})$ *a set of instance names* $p_1, \ldots, p_m$ *of the connector components representing patterns with* $O^c_{\mathfrak{S}^c} \cap O^p_{\mathfrak{S}^c} = \emptyset$, $c_{\mathfrak{S}^c}$ *a function which maps to each instance* $c_i \in O^c_{\mathfrak{S}^c}$ *a related component type,* $p_{\mathfrak{S}^c}$ *a function which maps to each instance* $p_j \in O^p_{\mathfrak{S}^c}$ *the related pattern, and* $map_{\mathfrak{S}^c} : (O_p.\mathcal{N}_{\mathcal{Q}}) \rightarrow ((O_c \cap \mathcal{N}_{\mathcal{C}}).\mathcal{N}_{\mathcal{Q}})$ *a bijective mapping which connects ports of components representing the pattern connectors with the ports of the root components of agents.*

The remaining part of the architecture consisting of the cognitive operators, other components outside the core, and their interconnections is also covered by related components whose connection with the safety-critical core are only unsafe ports (cf. Fig. 11).

To cover the complete system, including the cognitive operators, we employ the following extension:

Definition 12. *A* system $\mathfrak{S}$ *is a tuple* $(O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *which includes a core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$.[7]

For a system $\mathfrak{S}$ and core $\mathfrak{S}^c$ we have to combine the behavior related to the component instances and pattern instances to get the overall behavior:

$$\mathbb{M}_{\mathfrak{S}} := \left(\underset{c \in O^c_{\mathfrak{S}}}{\|} \mathbb{M}^c_{c_{\mathfrak{S}}(o_p)} \right) \| \left(\underset{p \in O^p_{\mathfrak{S}}}{\|} \mathbb{M}^p_{p_{\mathfrak{S}}(o_p)} \right) \quad \mathbb{M}_{\mathfrak{S}^c} := \left(\underset{c \in O^c_{\mathfrak{S}^c}}{\|} \mathbb{M}^c_{c_{\mathfrak{S}^c}(p)} \right) \| \left(\underset{p \in O^p_{\mathfrak{S}^c}}{\|} \mathbb{M}^p_{p_{\mathfrak{S}^c}(p)} \right).$$

In our example we have the top-level component Shuttle which is connected via *map* with instances of the ConvoyCoordination pattern. A single supervised embedded component instance of type AC exists for each Shuttle instance.

The overall safety of a system combining Definition 6 and 10 can then be defined referring to the overall behavior, pattern constraints, and component invariants as follows (by providing a formal version for Definition 3):

Definition 13. *A* system $\mathfrak{S} = (O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *with included core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is* safe *if the following conditions for the behavior are fulfilled:*

- *Local safety is fulfilled for the core:* $\mathbb{M}_{\mathfrak{S}^c} \models \wedge_{c \in O^c_{\mathfrak{S}^c}} \phi^c_{c_{\mathfrak{S}^c}(c)}$ (16)
- *Deadlock freedom is guaranteed for the core:* $\mathbb{M}_{\mathfrak{S}^c} \models \neg\delta$ (17)
- *All RT-OCL constraints for patterns are fulfilled:* $\mathbb{M}_{\mathfrak{S}} \models \wedge_{o \in O^p_{\mathfrak{S}^c}} \phi^o$ (18)

[7] Inclusion of a system in another system is defined formally in [3].

– *All OCL role invariants of agents are fulfilled:* $\mathbb{M}_{\mathfrak{S}} \models \wedge_{o \in O^c_{\mathfrak{S}^c}} \psi^o_{c_{\mathfrak{S}^c}(c)}$ (19)

Condition 17 ensure that the liveness properties defined in the role and port protocols are guaranteed by the core behavior, while condition 16, 18 and 19 ensure that the safety properties of the patterns and involved agents/components in the core are fulfilled by the overall behavior.

Overall Verification. For the decomposition of a system into a safety-critical core and the rest as depicted in Fig. 11 we now show that the safety of the core is not affected when composed with an arbitrary rest system.

Theorem 3. *Any system* $\mathfrak{S} = (O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *with included core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is safe if the core* $\mathfrak{S}^c$ *is safe.*

Proof. (sketch) Conditions 16 and 17 obviously holds as only the core is considered and the safety of the core guarantees it. Due to the checks for the unsafe ports, condition 18 and 19 can also be preserved when composing the core with the rest.

The following Corollary summarizes that the outlined separate results for the compositional and modular verification of MUML models can be combined to ensure the safety of the overall system (as defined in Definition 13).

Corollary 1. *A* system $\mathfrak{S} = (O^c_{\mathfrak{S}}, O^p_{\mathfrak{S}}, c_{\mathfrak{S}}, p_{\mathfrak{S}}, map_{\mathfrak{S}})$ *with included core system* $\mathfrak{S}^c = (O^c_{\mathfrak{S}^c}, O^p_{\mathfrak{S}^c}, c_{\mathfrak{S}^c}, p_{\mathfrak{S}^c}, map_{\mathfrak{S}^c})$ *is* safe *if*

- *all embedded component types are locally safe (condition 3 and 5),*
- *all embeddings are behaviorally consistent (see condition 4 or 6),*
- *all patterns are locally safe (condition 12),*
- *all agents are locally safe (condition 13),*
- *all regular ports are behaviorally consistent (condition 14), and*
- *all unsafe ports are behaviorally consistent (condition 15).*

Proof. (sketch) For the non-hierarchial case the result follows from Theorem 2. To also take the hierarchical embedding of subordinated components into account, we refer to Theorem 1 which guarantees that the whole behavior is well-formed and that the real-time behavior of the top-level components is always only refined by the behavioral consistent embeddings. The safety of the core system is not affected by the rest of the system following Theorem 3.

5 Related Work

The UML concepts themselves without the MUML refinements and UML extension for real-time, such as the UML Profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) [35] and its extension MARTE-DAM [36] for dependability analysis are not sufficient for the model-driven development of advanced mechatronic systems as targeted in the paper. They are neither defined

rigorously enough to support a decomposition for the analysis nor appropriately tailored to support systems with self-optimization. The System Modeling Language (SysML) [37], which combines UML concepts with system engineering concepts has the same limitations and restricts its attention to requirement and early phases and thus does not provide the required support for the later phases.

The Koala Component Model for Consumer Electronics Software [38] is one example where reconfiguration has been taken into account in a similar setting. However, the model is restricted to the component structure only and does not cover real-time or hybrid behavior.

In the OMEGA project [39], the UML has been extended by additional time constructs. However, in contrast to our approach, there is no support for hybrid behavior, and compositional verification is only supported by semi-automatic verification via theorem proving.

The description of control algorithms by time-continuous variables and corresponding ports is similar to other approaches such as HyROOM [40] and the underlying HyCharts [41]. Masaccio [42] and CHARON [28, 43, 44] also support the component-based modeling of hybrid systems and verification. The software's architecture is specified similarly to ROOM/UML-RT and the behavior is specified by statecharts whose states are associated with systems of ordinary differential equations, differential constraints or Matlab/Simulink block diagrams. These approaches provide means for the reconfiguration of systems in terms of changing the continuous behavior. However, it is only possible to reconfigure the model inside a component on one hierarchy-level. Our approach allows for a complex reconfiguration altering the structure across more than one hierarchy-level. For a more comprehensive comparison of a number of modeling techniques for advanced mechatronic systems, which also addresses the adaptation aspect, we refer to [45].

Concerning the verification of adaptive behavior, only first attempts exist. In [46], as in our presented work, verification techniques are employed to ensure that the self-adaptive behavior does not result in any harm. We additionally include self-coordinating behavior, suggest a compositional approach which can also be employed to study systems unable to be addressed as a whole. We also take real-time and continuous behavior into account. In [47], required properties for untimed models are checked under the assumption that a self-x capability of a system will fix certain types of problems in the long run. Also, we have to provide guarantees for the self-optimizing mechatronic systems in all possible cases under hard real-time constraints. Due to the multiple involved agents and their limitations (only local knowledge and limited reasoning capabilities) we cannot rely on the adaptation capabilities of the agents. Another direction for assurance is runtime verification [48]. However, it would be too late for the considered class of safety guarantees if the problems are detected at runtime. Only in cases where the runtime verification is not required in hard real-time (and thus remain outside the safe core) such an approach seems reasonable. A possible orthogonal extension of the presented approach is therefore to perform such runtime verification steps in the cognitive operator.

6 Conclusion

The MUML approach enables the model-driven development of mechatronic systems with advanced capabilities such as self-adaptive run-time behavior by providing the following three building blocks: (1) A suitable modeling approach for hierarchical structures of OCMs is provided which supports the specification of hybrid behavior and the reconfiguration of subsystems in order to support the reliable self-adaptation of the OCMs. (2) For the level of freely interacting software agents, the flexible but safe real-time coordination between the autonomous mechatronic agents is achieved employing the coordination pattern concept. (3) The approach integrates the hierarchical OCM structures and flexible interaction of software agents in such a manner that safety properties can be verified based on compositional checking so that the approach becomes scalable.

Acknowledgements. We thank Christian Heinzemann and Stefan Henkler for their comments on earlier versions of this paper. We are also grateful to all the other PhD students and students who worked on MUML or implemented the tool support for the described concepts as extensions for the Fujaba UML CASE tool.

References

1. Schäfer, W., Wehrheim, H.: The challenges of building advanced mechatronic systems. In: FOSE 2007: 2007 Future of Software Engineering, pp. 72–84. IEEE Computer Society, Washington (2007)
2. Sztipanovits, J., Karsai, G., Bapty, T.: Self-adaptive software for signal processing. Commun. ACM 41(5), 66–73 (1998)
3. Giese, H., Schäfer, W.: Model-driven development of safe self-optimizing mechatronic systems with mechatronic uml. Technical Report tr-ri-12-322, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany (2012), `http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/GS12.pdf`
4. Burmester, S., Tichy, M., Giese, H.: Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In: Aßmann, U. (ed.) Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, pp. 155–169 (June 2004)
5. Burmester, S., Giese, H., Tichy, M.: Model-Driven Development of Reconfigurable M. In: Aßmann, U., Akşit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 47–61. Springer, Heidelberg (2005)
6. Giese, H.: A Formal Calculus for the Compositional Pattern-Based Design of Correct Real-Time Systems. Technical Report tr-ri-03-240, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Deutschland (July 2003)
7. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the 9th European Software Engineering Conference held Jointly with 11th ACM SIGSOFT international Symposium on Foundations of Software Engineering (ESEC/FSE 2011), pp. 38–47. ACM Press (September 2003)

8. Burmester, S., Giese, H., Oberschelp, O.: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: Araujo, H., Vieira, A., Braz, J., Encarnacao, B., Carvalho, M. (eds.) Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal, pp. 222–229. INSTICC Press (August 2004)
9. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 179–188. Springer, Heidelberg (2004)
10. Burmester, S., Giese, H., Oberschelp, O.: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: Informatics in Control, Automation and Robotics. Kluwer Academic Publishers, Dordrecht (2005)
11. Hestermeyer, T., Oberschelp, O., Giese, H.: Structured Information Processing For Self-optimizing Mechatronic Systems. In: Araujo, H., Vieira, A., Braz, J., Encarnacao, B., Carvalho, M. (eds.) Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), pp. 230–237. INSTICC Press, Setubal (2004)
12. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington, DC (2007)
13. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. International Journal on Software Tools for Technology Transfer (STTT) 10(3), 207–222 (2008)
14. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In: Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML 2004, pp. 1–20 (October 2004)
15. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In: Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA (May 2005)
16. Burmester, S., Giese, H., Schäfer, W.: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 25–40. Springer, Heidelberg (2005)
17. Burmester, S., Giese, H., Gambuzza, A., Oberschelp, O.: Partitioning and Modular Code Synthesis for Reconfigurable Mechatronic Software Components. In: Bobeanu, C. (ed.) Proc. of European Simulation and Modelling Conference (ESMc 2004), Paris, France, pp. 66–73. EOROSIS Publications, Paris (2004)
18. Giese, H., Henkler, S., Hirsch, M.: A multi-paradigm approach supporting the modular execution of reconfigurable hybrid systems. Simulation 87(9), 775–808 (2011)
19. Oberschelp, O., Gambuzza, A., Burmester, S., Giese, H.: Modular Generation and Simulation of Mechatronic Systems. In: Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics, SCI, Orlando, USA (July 2004)

20. Heinzemann, C., Pohlmann, U., Rieke, J., Schäfer, W., Sudmann, O., Tichy, M.: Generating simulink and stateflow models from software specifications. In: Proceedings of the 12th International Design Conference, DESIGN 2012 (May 2012) (accepted)
21. Giese, H., Burmester, S.: Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany (June 2003)
22. Burmester, S., Giese, H.: The Fujaba Real-Time Statechart PlugIn. In Giese, H., Zündorf, A., eds.: Proc. of the first International Fujaba Days 2003, Kassel, Germany. Volume tr-ri-04-247 of Technical Report., pp. 1–8. University of Paderborn (October 2003)
23. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology 1(1) (1997)
24. Henzinger, T.A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
25. OMG: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (September 2002)
26. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: The Next Generation. In: Proc. of the 16th IEEE Real-Time Symposium. IEEE Computer Press (December 1995)
27. Bender, K., Broy, M., Peter, I., Pretschner, A., Stauner, T.: Model based development of hybrid systems. In: Modelling, Analysis, and Design of Hybrid Systems. LNCIS, vol. 279, pp. 37–52. Springer, Heidelberg (2002)
28. Alur, R., Dang, T., Esposito, J., Fierro, R., Hur, Y., Ivancic, F., Kumar, V., Lee, I., Mishra, P., Pappas, G., Sokolsky, O.: Hierarchical Hybrid Modeling of Embedded Systems. In: First Workshop on Embedded Software (2001)
29. Lynch, N.A.: Input/Output Automata: Basic, Timed, Hybrid, Probabilistic, Dynamic,.. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 191–192. Springer, Heidelberg (2003)
30. Flake, S., Mueller, W.: An OCL Extension for Real-Time Constraints. In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 150–171. Springer, Heidelberg (2002)
31. Giese, H., Hirsch, M.: Modular Verification of Safe Online-Reconfiguration for Proactive Components in Mechatronic UML. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 67–78. Springer, Heidelberg (2006)
32. Giese, H., Hirsch, M.: Modular Verification of Safe Online-Reconfiguration for Proactive Components in Mechatronic UML. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 67–78. Springer, Heidelberg (2006)
33. Giese, H., Hirsch, M.: Checking and Automatic Abstraction for Timed and Hybrid Refinement in Mechtronic UML. Technical Report tr-ri-03-266, University of Paderborn, Paderborn, Germany (December 2005)
34. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences 57, 94–124 (1998); A preliminary version appeared in the Proceedings of the 27th Annual Symposium on Theory of Computing (STOC), pp. 373–382. ACM Press (1995)
35. OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1 (June 2011)
36. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within MARTE. Softw. Syst. Model. 10(3), 313–336 (2011)
37. Object Management Group: Systems Modeling Language (SysML) Specification (January 2005)

38. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The koala component model for consumer electronics software. Computer 33(3), 78–85 (2000)
39. Graf, S., Hooman, J.: Correct Development of Embedded Systems. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 241–249. Springer, Heidelberg (2004)
40. Stauner, T., Pretschner, A., Péter, I.: Approaching a Discrete-Continuous UML: Tool Support and Formalization. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 242–257. Springer, Heidelberg (2001)
41. Stauner, T.: Systematic Development of Hybrid Systems. PhD thesis, Technische Universität München (2001)
42. Henzinger, T.A.: Masaccio: A Formal Model for Embedded Components. In: Watanabe, O., Hagiya, M., Ito, T., van Leeuwen, J., Mosses, P.D. (eds.) TCS 2000. LNCS, vol. 1872, pp. 549–563. Springer, Heidelberg (2000)
43. Alur, R., Ivancic, F., Kim, J., Lee, I., Sokolsky, O.: Generating embedded software from hierarchical hybrid models. In: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems, pp. 171–182. ACM Press (2003)
44. Alur, R., Grosu, R., Lee, I., Sokolsky, O.: Compositional Refinement of Hierarchical Hybrid Systems. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 33–48. Springer, Heidelberg (2001)
45. Giese, H., Henkler, S.: A survey of approaches for the visual model-driven development of next generation software-intensive systems. Journal of Visual Languages and Computing 17, 528–550 (2006)
46. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE 2006: Proceeding of the 28th International Conference on Software Engineering, pp. 371–380. ACM Press, New York (2006)
47. Güdemann, M., Ortmeier, F., Reif, W.: Formal Modeling and Verification of Systems with Self-x Properties. In: Yang, L.T., Jin, H., Ma, J., Ungerer, T. (eds.) ATC 2006. LNCS, vol. 4158, pp. 38–47. Springer, Heidelberg (2006)
48. Goldsby, H.J., Cheng, B., Zhang, J.: AMOEBA-RT: Run-Time Verification of Adaptive Software. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 212–224. Springer, Heidelberg (2007)