Modeling Reconfigurable Mechatronic Systems with Mechatronic UML*

Sven Burmester**, Matthias Tichy, and Holger Giese

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
[burmi|mtt|hg]@upb.de

Abstract. Due to the safety-critical character of mechatronic systems the engineers have to face the following serious problems: The models which are used in design must enable to specify real-time behavior on basis of a semantics that allows an automatic synthesis of code which respects the specified real-time requirements. Further, mechatronic systems usually embed continuous behavior in form of feedback-controllers which leads to hybrid systems. A new field of research deals with the support of dynamic exchange of feedback controllers (reconfiguration) during design. Models have to support a complex, distributed system structure. The system's real-time behavior must be verifiable in spite of the complex structure and the embedded continuous control elements. In this paper we will present our approach which fulfills all these requirements considering an example taken from the RailCab research project.

1 Introduction

A new field of research concerns the software development for reconfigurable mechatronic systems. Mechatronic systems combine technologies from mechanical and electrical engineering as well as from computer science. They are *real-time systems* because reactions to the environment usually have to be completed within a specific, predictable time and they are *hybrid systems* because they usually consist of discrete control modes as well as implementations of continuous feedback controllers. Due to their application domain the behavior, which is largely controlled by software, has to meet safety-critical requirements.

In recent times, mechatronic systems, which had been single, autonomous systems prior to that, have been used in distributed settings, which require extensive coordination. Due to the new requirements stemming from distribution and coordination scenarios, a new generation of *reconfigurable* mechatronic systems has emerged. Those reconfigurable mechatronic systems change their behavior in order to conform with certain roles in a coordination with other mechatronic systems. This reconfiguration

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

** Supported by the International Graduate School of Dynamic Intelligent Systems. University of Paderborn

leads to an increased complexity and thus, makes it more difficult to guarantee safety-critical requirements. To guarantee safety for reconfigurable mechatronic systems, we extend in this paper the Model Driven Architecture (MDA) approach [1] for the design of hybrid mechatronic real-time systems.

Two different views need to be distinguished: the structural view and the behavioral view. The structural view describes the overall system that consists of multiple component instances, which are possibly distributed, interconnected with each other, and are exchanging messages via communication. In the behavioral view, the behavior of single components is specified. As proposed in the MDA approach, structure and behavior are specified with platform independent models which are transformed to platform specific code, later. We apply extended UML diagrams (component diagrams, statecharts) [2] as platform independent models which are extended with diagrams from other domains (block diagrams [3]).

Many existing specification languages provide solutions for some specific subproblems, but they do not provide seamless support for modeling, verification and code generation, which is necessary for the design of such safety-critical systems. Specification languages like ROOM [4] or UML/RT [5] allow a specification of the system structure, but not to specify real-time or hybrid behavior. Other specification languages provide code synthesis (although usually without an adequate support for real-time behavior) [6], but even simple models become so complex, that model checking is not applicable any more. On the other hand model checkers for real-time systems [7] often use a model as basis whose semantics are not realizable on real physical systems (e.g. it is impossible to react on an event without a delay).

Therefore, an approach is required, that is based on a model containing sufficient information to specify real-time behavior and to provide appropriate source code synthesis. Further, methods for verification are required that guarantee the correctness of the distributed overall system. Reconfigurable mechatronic systems are typically too complex for verification of the whole system by model checking. Instead, a general view is required that abstracts from the components' internals. Then, compositional model checking can be applied that considers just this component's external visible behavior. When specifying the details of the component's behavior, it must be guaranteed that the details do not invalidate the component's generalization.

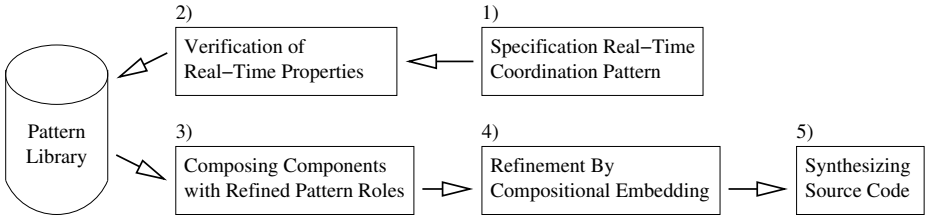

Fig. 1. Seamless support for the design of mechatronic systems

In this paper, we present an approach which allows the development of hybrid mechatronic components. Basically, the approach is divided into two parts (see Figure 1). In the first part (steps 1+2), individual coordination patterns are developed. These coordination patterns have different roles, which contain the real-time logic for the coordination, and a real-time constraint, which is proven w.r.t. certain communication network properties. If such a coordination pattern has been successfully verified, it is added to a pattern library. In the second part (steps 3-5), the mechatronic components are built using the pre-fabricated coordination patterns stored in the library of patterns. The real-time behavior of the component is a refinement of the combination of pattern roles and additional specified behavior. The employed refinement notion ensures the verified real-time properties. In the next step further components (e.g. hybrid ones) are embedded into the superordinated component. Simple consistency checks ensure again that the verified real-time properties of the coordination patterns are still valid in spite of the embedding. As the last step, source code is synthesized for the specified structure and behavior.

In the next section, we discuss related work. Afterwards, we present the application scenario, which we use in this paper to exemplify the application of our approach. In Section 4, we present our approach w.r.t. system structure, real-time behavior, and hybrid behavior. In Section 5 we relate our approach to the UML 2.0 specification [2]. We conclude in Section 6 and present future work.

2 Related Work

The UML [2] can be considered as the currently evolving standard to model complex software systems even in the real-time domain [8–11]. Consequently, in this paper we propose an approach to realize the above outlined vision with UML, even though UML has not been originally designed to support mechatronic systems.

From the large number of object-oriented modeling approaches [4, 12–14] ROOM [4] has finally found its way into the UML 2.0 specification [2]. However, the required support for real-time behavior modeling is still not available, as the ROOM concepts focus on architectural design and do not address the real-time or hybrid behavior of the operational model at all.

The OMG published a RFP (Request for Proposal) for *UML for System Engineering (UML for SE)* [15]. The idea of UML for SE is to provide a language that supports the system engineer in modeling and analyzing software, hardware, logical and physical subsystems, data, personnel, procedures, and facilities. One distinguishing proposal is called *Systems Modelling Language (SysML)*[1], that takes a subset of the UML 2.0 specification and extends it. Main extensions, related to the design of continuous and hybrid systems are *Structured Classes*, that describe the fine structure of a class extended by continuous communication links between ports. In *Parametric Diagrams* the *parametric* (arithmetic) *relations* between numerical attributes of instances are specified and the nodes of Activity Diagrams are extended with continuous functions and in- and outputs. The RFP shows that there is need for a UML extension supporting system engineering and including continuous behavior. The SysML proposal provides this

[1] www.sysml.org

support, but for the specification of parametric relations a static structure is assumed. There is no support for modeling structural changes or replacements of the parametric relations. Further the required support for modeling of realizable real-time behavior or for abstraction that is required for compositional model checking is not available.

Hierarchical timed automata [16], a hierarchial extension of timed automata [17], are often used to model real-time systems. They provide most of the powerful modeling concepts of statecharts. A mapping to multiple parallel running flat timed automata permits to verify the model by using the model checker UPPAAL [7]. In [18] locations of a flat UPPAAL automaton are associated with tasks inclusive worst case execution times (WCETs) and deadlines. This extension enriches the model with the information required for code generation and a prototype synthesizing C-Code has been implemented. As the code generation approach is restricted to flat automata, it does not take the additional syntactical constructs of hierarchical timed automata into account. The code generation scheme is not really sufficient for hard real-time systems, as it does not take into account the delays that occur when transitions are fired, arguing that these delays are small compared with the WCETs. Further there is no support for deployed systems or for systems with a hybrid character.

Time Weaver [19] is a tool for the design of embedded real-time systems. It deals with the deployment of components, their communication and with replica for redundant computation which increases fault tolerance, but the real-time behavior of the single components is just described by periods and deadlines for the computation of the component as a whole. A model-based specification of its internal (real-time) behavior does not exist. Analyses are restricted to scheduling analyses, that prove if deadlines are met. Verification or validation of the correct behavior is not addressed.

3 Application Example

Our approach has been developed within the collaborative research center 614 of the German National Science Foundation (DFG), titled ”Self-optimizing Concepts and Structures in mechanical Engineering” which includes 12 research groups from mechanical engineering, electrical engineering, information and computer science and mathematics.[2]

The general vision of this collaborative research center is to develop concepts and methods to build mechatronics products with inherent intelligence, which react autonomously and flexibly to changing environment and operation conditions.

As a concrete example, a self-optimizing version of the software for the RailCab research project[3] has to be developed which aims at using a passive track system with intelligent shuttles that operate individually and make independent and decentralized operational decisions.

The vision of the railcab project is to provide the comfort of individual traffic concerning scheduling and on-demand availability of transportation as well as individually equipped cars on the one hand and the cost and resource effectiveness of public transport

[2] http://www.sfb614.de
[3] http://www-nbp.upb.de/en

on the other hand. The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in the Transrapid[4] to increase passenger comfort while still enabling high speed transportation and (re-)using the existing railway tracks.

One particular problem 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 require a small distance between the different shuttles such that a high reduction of energy consumption is achieved. Coordination between 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 building the control software of the shuttles.

When shuttles approach each other they use wireless communication to coordinate the building of the convoy. Dependent on the position within the convoy they have to change their behavior. For example the rear shuttles will no longer hold their velocity on a constant level, but the distance to their front shuttle. Therefore, they dynamically have to exchange the feedback controller which controls its acceleration. Further, a shuttle will reduce the intensity of braking when another one drives in a short distance behind to avoid a rear-end collision. Consequently, the shuttle design must ensure on the one hand that the communication fulfills all safety requirements (e.g. no deadlocks) and that the exchange of the dynamic controller (*reconfiguration*) guarantees safety and stability.

As a running example within this paper we consider a simplified version of the convoy building problem, namely we assume that only convoys of two shuttles are formed.

4 Mechatronic UML Modeling

Component-based Software Engineering [20] is a well known approach for building software systems. For the development of mechatronic systems using UML, we extend the UML by notions for the specification of continuous and real-time behavior. The real-time extensions for the UML are specially geared towards verification of safety-critical properties. In the following, we will describe our approach in detail using the above mentioned example.

4.1 System Structure

UML component diagrams are used for the specification of the structure of our systems. Component diagrams specify components and their interaction in form of connectors. We distinct component types and their instances during runtime. Connectors model the communication between different components via the ports and interfaces and the communication properties w.r.t. message loss, latency, etc. Ports are distinct interaction points between components and are typed by provided and required interfaces.

For our example scenario, Figure 2 shows the component type for the shuttle. The Shuttle component contains a hybrid AccelerationControl (AC) component. This component computes the acceleration needed to achieve a specific goal (keeping a specified

[4] http://www.transrapid.de/en

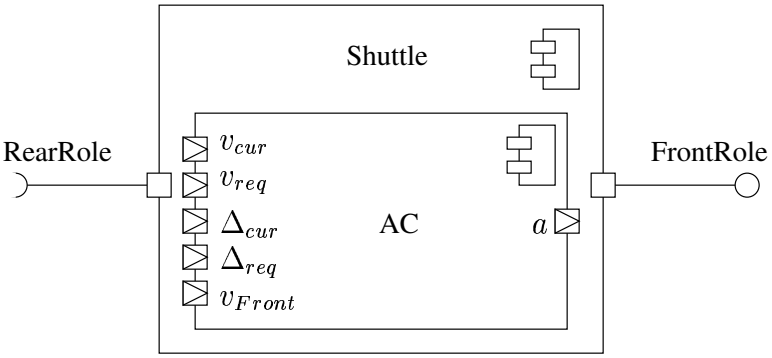

Fig. 2. Type specification of component Shuttle

distance or keeping a specified speed level). The AccelerationControl component has five incoming continuous ports 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, and one outgoing continuous port. This outgoing port sends acceleration values to the appropriate hardware actuator devices. In addition (the details are presented in Section 4.3), the AC component contains discrete behavior to switch between keeping distance or keeping velocity on a constant level, and, thus, is a hybrid component. For clearer presentation, the sensors and actuators connected to the input ports and the output port of the AC component have been hidden.

4.2 Real-Time Behavior

Interaction between component instances during runtime is a major part of mechatronic system design. In our scenario a shuttle forms a convoy with another shuttle via the RearRole and FrontRole interfaces. In the domain of mechatronic systems, a layered system architecture is feasible which guarantees that an autonomous unit like a shuttle reacts in a local environment and the interfaces to its environment are strictly defined (as e.g. a shuttle trying to form a convoy has to interact only with one other shuttle and not maybe with a third one which is a few kilometers away). This domain-specific restriction is the reason why usually only relative simple coordination patterns have to be constructed, i.e. patterns with simple coordination protocols between roles, limited numbers of input signals and a fixed number of roles.

The interaction between two shuttles w.r.t. forming a convoy is one such simple coordination pattern. Figure 3 shows the ConvoyCoordination pattern between two shuttles. The protocol for forming and breaking convoys is specified in the roles of this pattern (see Figures 4 and 5). Components in the domain of mechatronic systems must meet real-time requirements. Therefore, for the specification of role behavior we use our real-time variant of statecharts called *Real-Time Statecharts* [21, 22]. They allow to apply constructs from timed automata [7, 17] like clocks, time guards and time invariants and further annotations like worst case execution times and deadlines (see Section

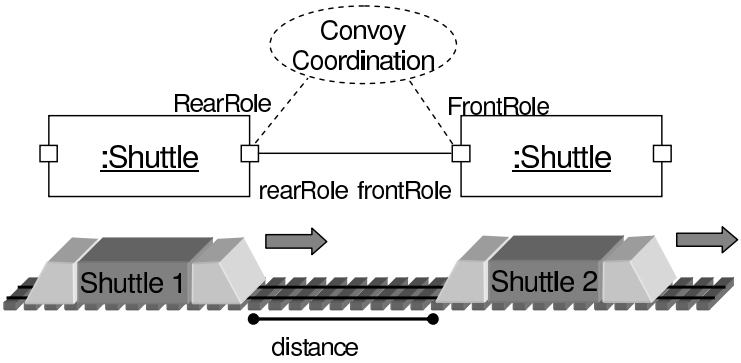

Fig. 3. Component Diagram and Patterns

4.3). As shown in [21], these annotations enable an automatic implementation on a real physical machine with limited resources.

If an event has the form interface.message it means that the transition is triggered when message is received via the interface interface. Side-effects of the form interface.message describe the sending of message to a receiver which is connected via interface. Later, we will use events where no interface is specified. Then message is local and sent or received within the same statechart.

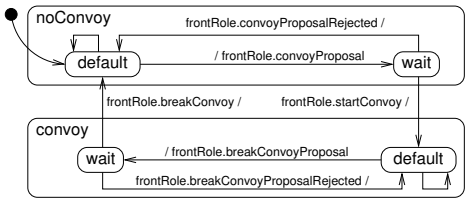

Fig. 4. Statechart of the RearRole role

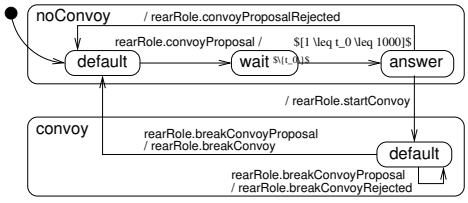

Fig. 5. Statechart of the FrontRole role

Initially, both roles are in state noConvoy::default, which means that they are not in a convoy. The rear role non-deterministically chooses whether to propose forming a convoy or not. After having chosen to propose a convoy, a message is sent to the other shuttle resp. its front role. The front role chooses non-deterministically to reject or to accept the proposal after max. 1000 msec. In the first case, both statecharts 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 a break of the convoy and sends this proposal to the front shuttle. The front shuttle chooses non-deterministically to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies by an approval message, and both roles switch into their respective noConvoy::default state.

For the connector which represents the wireless network we do not apply an explicit statechart, but instead specify 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 1 up to 5 msec. The connector is unsafe in the sense that it might fail at any time, such that we set our specific QoS characteristic reliable to false.

To provide fail safe behavior, the following RT-OCL [23] constraint must hold. It demands that a combination of role states where the front role is in state noConvoy and the rear role is in state convoy is not possible. This is required because such a situation would allow the front shuttle to brake with full intensity although another shuttle drives in short distance behind.

```
context DistanceCoordination inv:
    not (self.oclInState(RearRole::Main::convoy) and
         self.oclInState(FrontRole::Main::noConvoy))
```

It is shown in [24], that this property holds. As mentioned there, those patterns are individually constructed and verified. Afterwards, they are combined in structurally refined versions in the different components. In our example the Shuttle component is a combination of refined versions of the RearRole and the FrontRole (see Section 4.3). For a component, which combines different patterns respective the roles, the verified properties still hold due to the approach presented in [24]. Thus, components for mechatronic systems are developed in a way similar to a construction kit using several proven and verified building blocks and refine them to suit different requirements.

In this section, we presented how to specify and verify single real-time coordination patterns. In the next section, we show how components are developed without compromising the verification results by composing roles of different coordination patterns and refining their behavior. In this refinement process hybrid and real-time behavior of the single components is modeled.

4.3 Component specification

Shuttle Component Figure 6 depicts the behavior of the Shuttle component from Figure 2, taken from [24] and extended with real-time annotations. The Real-Time

Statechart consists of three orthogonal states FrontRole, RearRole, and Synchronization. FrontRole and RearRole are refinements of the role behaviors from Figures 4 and 5 and specify in detail the communication that is required to build and to break convoys. Synchronization coordinates the communication and is responsible for initiating and breaking convoys. The three sub-states of Synchronization represent whether the shuttle is in the convoy at the first position (convoyFront), at second position (convoyRear), or no convoy is built at all (noConvoy). The whole statechart is a refinement of both role descriptions as it just resolves the non-determinism from Figures 4 and 5 and does not add additional behavior.

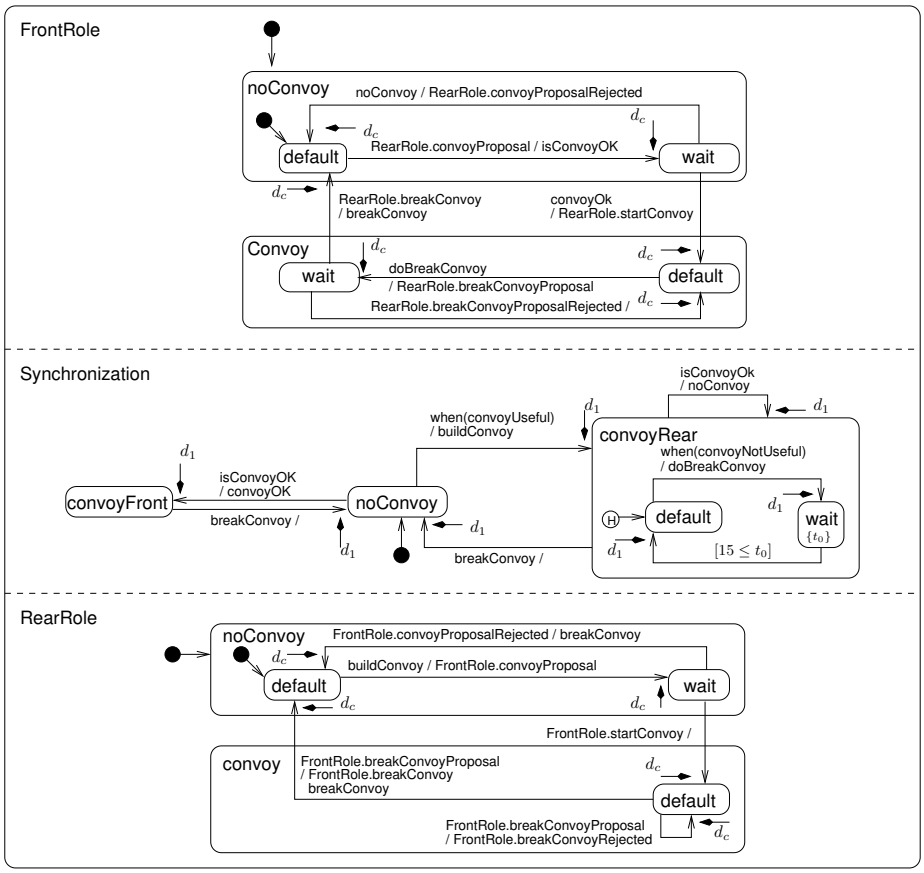

Fig. 6. Behavior of the Shuttle component

As mentioned above components in the domain of mechatronic systems must meet real-time requirements. In the specific example it needs not only to be specified that, e.g. RearRole has to send a startConvoy message after receiving convoyOK, but also that this has to be finished within a specific, predictable time. Therefore, we apply our

Real-Time Statecharts [21] for specification. Real-time statecharts respect that the firing of transitions consumes time and that real physical, mechatronic systems can never react in *zero time*, but always with a delay. To represent this in the model, we make use of the deadline construct:

In Figure 6 so called *deadline intervals* d_c and d_1 are used to specify a minimum and a maximum duration for the time between triggering a transition and finishing its execution. E.g. sending the message convoyProposalRejected to RearRole has to be finished within the time specified by d_c after receiving the message noConvoy in state FrontRole::noConvoy::wait. As another example for predictable timing behavior (real-time behavior) the change in Synchronization from noConvoy to convoyFront has to be finished within d_1.

AC Component The AC component is a hybrid component. It consists of two discrete control modes which represent whether the shuttle is under velocity control or under distance control (see Figure 7). Further it has continuous in- and outputs. Dependent on the active discrete mode either the current and the required velocity are used for the velocity controller or the current and required distance to the front shuttle as well as the velocity of the first shuttle are used for the distance controller. The output a is the acceleration in any mode.

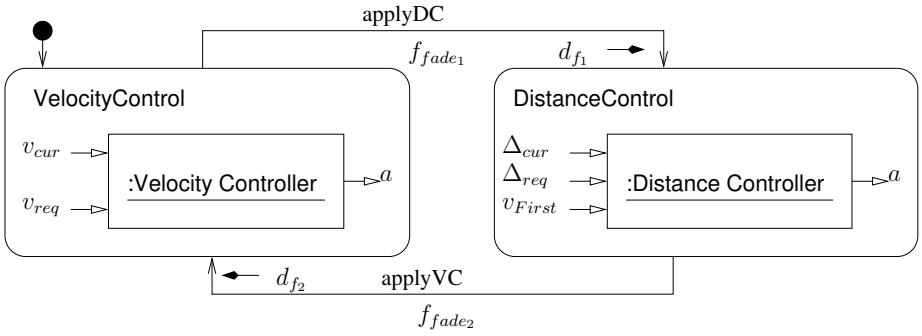

Fig. 7. Behavior of the AC component

In order to embed the continuous controllers into the discrete states, the Real-Time Statecharts are extended to hybrid ones. In Hybrid Statecharts each discrete state is associated with a configuration of embedded component instances [25]. In this example, each configuration consists of just one single feedback controller.

When a change occurs between the discrete states a discrete switch between the controllers could lead to an unsteadiness in the output signal a. This unsteadiness will stimulate additional oscillations which could lead to instability even when both controllers are stable on their own. In order to avoid these unsteadinesses output cross-fading is applied [25]. This is specified by a *fading function* f_{fade_1} resp. f_{fade_2} and a

minimal and a maximal *fading duration* (d_{f_1} resp. d_{f_2}) which is specified as an interval as well.

Although the hybrid AC component has 5 different continuous input signals, never all of them are required. When the component is in velocity control mode only v_{cur} and v_{req} are required, in distance control mode only Δ_{cur}, Δ_{req}, and v_{First} are required. These dynamic interfaces are visualized by the so called *Interface Statechart* in Figure 8.

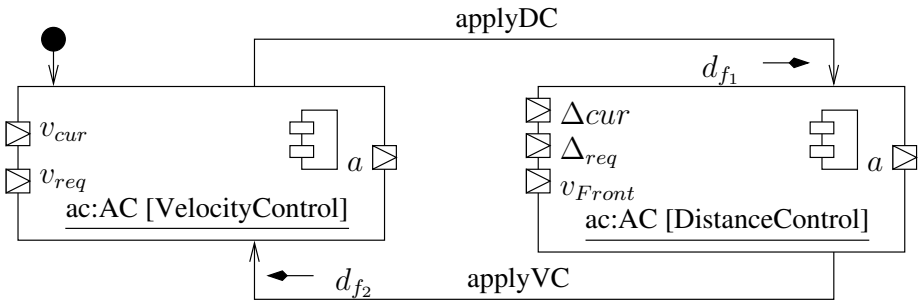

Fig. 8. Interface Statechart of the AC component

The Interface Statechart abstracts from the component's internals as it just contains the externally relevant behavior: the different control modes, the modes' continuous in- and outputs, and the deadline information for switches between the control modes. Whether fading is required and which kind of fading function is applied and which continuous feedback controllers are applied is not important for the external view. This interface representation is used when the different components are embedded into each other (see below).

Behavioral Embedding In the previous sections two components and their behaviors have been specified. Although they are embedded hierarchically from the structural point of view (cf. Figure 2), their behavior is executed concurrently. We say AC is *hierarchically, parallel embedded* into Shuttle. As it makes no sense for AC to be in state DistanceControl while Synchronization is in state convoyFront, which represents the situation when there is no further shuttle before, the two behavior descriptions have to be coordinated.

Therefore, the Shuttle statechart from Figure 6 is extended to a Hybrid Statechart. Figure 9 depicts the orthogonal Synchronization state, whose sub-states embed different configurations each consisting of one AC instance *and its current internal state and continuous interface*. So in Figure 9 is specified that AC has to be in state DistanceControl when Synchronization is in state convoyRear. If Synchronization is in state noConvoy or convoyFront AC has to be in state VelocityControl. Consequently, a state change within the orthogonal Synchronization state implies a state change in its embedded AC component. As only the external visible information of the AC component is important when

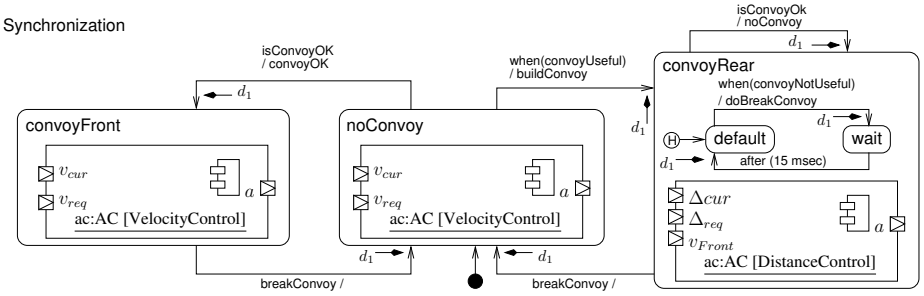

Fig. 9. Behavioral embedding

it is embedded, the form of the embedded component is equal to the single states of the Interface Statechart from Figure 8.

This kind of modeling has the advantage that it supports the decomposition into multiple components that is required to handle the complexity in mechatronic systems. Further the control engineering know-how is separated from the software engineering know-how: The discrete coordination and communication is specified by the statechart from Figure 6, the continuous behavior and the restrictions of the controller exchange is specified in Figure 7 and the later integration is specified in Figure 9. Another advantage is the support for flexible continuous interfaces.

In order to ensure that the results of the compositional verification are not invalidated by the detailed realization of the Shuttle component, the component realization has to be a refinement of the role behavior (see Section 4.2). The statechart from Figure 6 is a refinement of the roles from Figures 4 and 5. Consequently, it needs to be ensured that the embedding of AC still just refines the specified real-time behavior from Figure 6 and is not adding additional behavior or is in conflict with the real-time specification of this superordinated component.

Assume, for example, in Figures 6 and 9 is specified 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 Figure 8 the minimal fading duration may not be above 200 msec.

This example demonstrates how consistency is approved by simple syntactical checks between the superordinated component and the Interface Statecharts of the embedded components: In the above example $d_{f_1} \subseteq d_c$ must be satisfied. Such checks have to be enforced for every possible change of the global state (the current global state consists of the current states of all components). Due to the hierarchical parallel embedding, the global state space is restricted: Although Synchronization consists of 3 states and AC of 2 states, the hierarchical parallel composition consists not of $2 * 3 = 6$ states, but just of 3 states.[5] This information is contained in the specification in Figure 9 and does not need to be derived by a costly reachability analysis. Consequently, the number of consistency checks to be enforced are usually not exponential in the num-

[5] This is because the state combinations (convoyFront, DistanceControl), (noConvoy, DistanceControl), and (convoyRear, VelocityControl) are not reachable.

ber of states. If these consistency checks are successful the results of the compositional model checking are valid even for components that embed further components in the hierarchical, parallel manner. A detailed description and formalization can be found in [25].

5 Discussion

The UML 2.0 specification is the de facto standard in modeling. As our and the UML's modeling philosophy have some things in common, we decided to extend the UML 2.0 specification for the design of hybrid mechatronic real-time systems: The specification of the whole system's structure by hierarchical components, ports, interfaces and connectors is in both approaches similar.

The communication protocols are specified in UML through so called *Protocol State Machines (PSM)*, which have the syntax of finite state machines. Instead of PSMs we apply our real-time extension of the UML statecharts, the *Real-Time Statecharts* to specify real-time communication protocols. In Real-Time Statecharts it is possible to specify deadlines, worst case execution times, clocks, clock resets, time guards, and time invariants. We use these models to ensure correctness by model checking.

The behavior of a single component is specified by Hybrid Statecharts instead of UML statecharts. We extended the concept of Real-Time Statecharts to Hybrid Statecharts as the hierarchical composition of components results in multiple behavior descriptions, which have to be coordinated. Therefore each discrete state is associated with a configuration of embedded components. Further it is possible to apply output cross-fading when reconfiguring the configurations. The notion of Hybrid Statecharts helps to ensure, that the properties, verified through compositional model checking, still hold and they enable to embed components from other domains (e.g. hybrid components).

Consequently our approach does not change the UML approach, it just extends its notation and semantics. Therefore, it is possible for system engineers to design hybrid mechatronic real-time systems, by following a well-known approach.

6 Conclusion and Future Work

Reconfigurable mechatronic systems in the domain of safety-critical distributed systems must be carefully designed. We presented a design approach which not only considers real-time issues but also allows for a mixture of discrete event-based as well as continuous behavior. In addition, the applied modeling approach contains means for the compositional verification of safety-critical properties. Finally, the models built with our approach are used to synthesize source code, which respects the real-time issues and the safety-critical properties. By creating the Mechatronic UML, we extended the industry standard UML rather than creating a completely new specification language in order to built upon experience of the developer.

Tool support (in form of a plug-in for the Fujaba Tool Suite[6]) for the specification, verification and automatic source code synthesis of the Real-Time Statecharts and the

[6] www.fujaba.de

real-time coordination patterns exists and early evaluation reports have been successful. For the support of hybrid behavior a prototypic implementation exists and we are currently working on the tool support. In the future, we will employ graph transformations [26] to describe the reconfiguration of the behavior w.r.t. the online addition or removal of coordination pattern roles. By this reconfiguration the hybrid components can reconfigure themselves to different coordination scenarios to optimize their memory and processing power footprints. These reconfigurations specified by graph transformations are also targets for the verification of safety-critical properties. Reliability and availability are other important aspects of reconfigurable mechatronic systems. We will generalize and apply the approaches of [27, 28] in the domain of mechatronic systems and combine them with the approach outlined in this paper.

Acknowledgements The authors thank Oliver Oberschelp for the support in the control engineering domain.

References

1. Object Management Group: Model Driven Architecture (MDA) Edited by Joaquin Miller and Jishnu Mukerji. (2001)
2. Object Management Group: UML 2.0 Superstructure Specification. (2003) Document ptc/03-08-02.
3. Ogata, K.: Modern Control Engineering. Prentice Hall (2002)
4. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley and Sons, Inc. (1994)
5. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Technical report, ObjectTime Limited (1998)
6. Saksena, M., Karvelas, P., Wang, Y.: Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models. In: The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Newport Beach, California (2000)
7. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology **1** (1997)
8. Bichler, L., Radermacher, A., Schürr, A.: Evaluation uml extensions for modeling realtime systems. In: Proc. on the 2002 IEEE Workshop on Object-oriented Realtime-dependable Systems WORDS'02, San Diego, USA, IEEE Computer Society Press (2002) 271–278
9. OMG: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (2002)
10. Gu, Z., Kodase, S., Wang, S., Shin, K.G.: A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada. (2003)
11. Masse, J., Kim, S., Hong, S.: Tool Set Implementation for Scenario-based Multithreading of UML-RT Models and Experimental Validation. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada. (2003)
12. Awad, M., Kuusela, J., Ziegler, J.: Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion. Prentice Hall (1996)
13. Douglass, B.P.: Real-Time UML: Developing Efficient Objects for Embedded Systems. The Addison-Wesley Object Technology Series. Addison-Wesley (1999) Second Edition.
14. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley (2000)

15. Object Management Group: UML for System Engineering Request for Proposal. (2003) Document ad/03-03-41.
16. David, A., Möller, M., Yi, W.: Formal Verification of UML Statecharts with Real-Time Extensions. In Kutsche, R.D., Weber, H., eds.: 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France. Volume 2306 of LNCS., Springer (2002) 218–232
17. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. In: Proc. of IEEE Symposium on Logic in Computer Science. (1992)
18. Amnell, T., David, A., Fersman, E., Pettersson, M.O.M.P., Yi, W.: Tools for Real-Time UML: Formal Verification and Code Synthesis. In: Workshop on Specification, Implementation and Validation of Object-oriented Embedded Systems (SIVOES'2001). (2001)
19. de Niz, D., Rajkumar, R.: Time Weaver: A Software-Through-Models Framework for Embedded Real-Time Systems. In: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (LCTES), San Diego, California, USA, ACM Press (2003)
20. Szyperski, C.: Component Software, Beyond Object-Oriented Programming. Addison-Wesley (1998)
21. Giese, H., Burmester, S.: Real-Time Statechart Semantics. Technical Report tr-ri-03-239, University of Paderborn, Paderborn, Germany (2003)
22. Burmester, S., Giese, H.: The Fujaba Real-Time Statechart PlugIn. In: Proc. of the Fujaba Days 2003, Kassel, Germany. (2003)
23. Flake, S., Mueller, W.: An OCL Extension for Real-Time Constraints. In: Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Volume 2263 of LNCS. Springer (2002) 150–171
24. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland. (2003)
25. Burmester, S., Giese, H., Schäfer, W., Oberschelp, O.: Hybrid UML Components for the Correct Design of Complex Selfoptimizing Mechatronic Systems. In: Proc. of International Symposium on Foundations of Software Engineering (FSE), Newport Beach, CA USA (submitted), ACM Press (2004)
26. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1. World Scientific, Singapore (1999)
27. Tichy, M., Giese, H.: Seamless UML Support for Service-based Software Architectures. In: Proc. of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003, Luxembourg. Volume 2952 of Lecture Notes in Computer Science. (2003)
28. Tichy, M., Giese, H.: A Self-Optimizing Run-Time Architecture for Configurable Dependability of Services. In de Lemos, R., Gacek, C., Romanovsky, A., eds.: Architecting Dependable Systems II. Lecture Notes in Computer Science. Springer Verlag (2004) (to appear).