Model-Driven Development of Reconfigurable Mechatronic Systems with MECHATRONIC UML*

Sven Burmester**, Holger Giese, and Matthias Tichy

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

Abstract. Today, advanced technical systems are complex, reconfigurable mechatronic systems where most control and reconfiguration functionality is realized in software. A number of requirements have to be satisfied in order to apply the model-driven development approach and the UML for mechatronic systems: The UML design models must support the specification of the required hard real-time event processing. The real-time coordination in the UML models must embed the continuous control behavior in form of feedback-controllers to allow for the specification of discrete and continuous hybrid systems. Advanced solutions further require the dynamic exchange of feedback controllers at runtime (reconfiguration). Thus, a modeling of rather complex interplays between the information processing and the control is essential. Due to the safety-critical character of mechatronic systems, the resulting UML models of complex, distributed systems and their real-time behavior must be verifiable in spite of the complex structure and the embedded reconfigurable control elements. Finally, an automatic code synthesis has to map the specification correctly to code. In this paper, we will present our MECHATRONIC UML approach, which fulfills all these requirements. The approach is motivated and illustrated by means of a running example.

1 Introduction

An emerging field of software engineering research concerns complex, reconfigurable mechatronic systems. Mechatronic systems [1] 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. As incorrect software can lead to failures with fatal consequences, they are also *safety-critical* systems.

* 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.

U. Aßmann, M. Aksit, and A. Rensink (Eds.): MDAFA 2003/2004, LNCS 3599, pp. 47–61, 2005.

Mechatronic systems, which had been single, autonomous systems, have been used in distributed settings, which require extensive coordination, in recent times. 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 comply with certain roles, which result from coordination and contracts with other mechatronic systems.

This reconfiguration leads to an increased complexity and thus makes it more difficult to fulfill safety-critical requirements. To guarantee safety for reconfigurable mechatronic systems, we extend in this paper the Model Driven Architecture (MDA) approach [2] for the design of hybrid mechatronic real-time systems with reconfiguration.

The today existing UML specification languages for technical systems [3,4,5,6,7,8] only provide solutions for either modeling, verification, or code generation, but fail to provide seamless support for all three requirements, which would be necessary for the model-driven development of reconfigurable mechatronic systems. Therefore, a specification language (model) is required, that contains at first sufficient information to specify the real-time behavior of the system in such a manner that high level modeling, verification, and semantically correct source code generation are possible. Methods for verification are required that guarantee the correctness of the whole distributed, hard real-time system. Reconfigurable mechatronic systems are typically too complex to directly verify the whole system using model checking. Instead, the model must enable the compositional model checking which considers just the component's external visible behavior to verify the real-time coordination. When specifying the details of the component's behavior within the model, it must be possible to guarantee that adding the details does not invalidate the component's external behavior taken into account during verification.

In this paper, we present the MECHATRONIC UML approach which allows the model-driven development of complex, distributed, safety-critical real-time systems which supports modeling, verification, and code generation by employing earlier inventions namely hybrid mechatronic components [9] and real-time coordination patterns [10]. 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 which 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 UML diagrams [3] as platform independent models. UML state machines are extended by more expressive constructs for the description of real-time behavior [11,12]. Component diagrams are refined such that block diagrams [13], which are the most common description technique in the domain of feedback control engineering, can be smoothly integrated [9].

The required steps during the model-driven development with the MECHATRONIC UML approach can basically be divided into three phases which will be described in detail in Section 3 (see Figure 1). In the first two phases a correct platform independent model (PIM) is specified: In the first phase (steps 1-3), the system structure is defined which is used to identify where in the system communication is required. Each

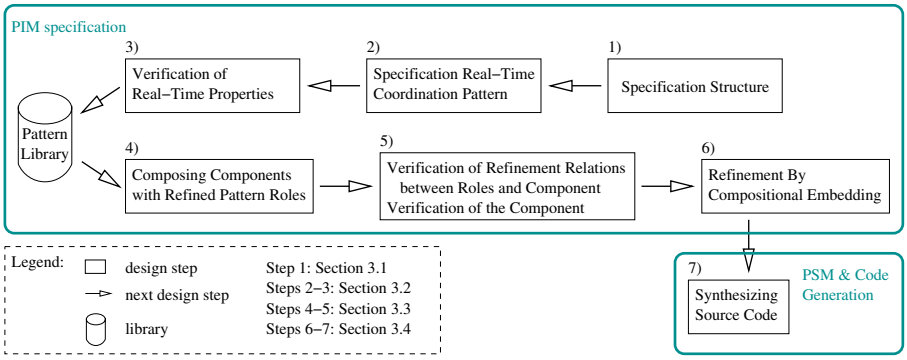

Fig. 1. Seamless support for the design of mechatronic systems

communication is described by an individual *real-time coordination pattern*. 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 designed and successfully verified, it is added to a pattern library for reuse.

In the second phase (steps 4-6), the mechatronic components are built using the prefabricated already verified 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 addition, the component's internal coordination has to be verified to exclude inconsistent behavior or deadlocks. 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. Thus, a complete verification of the system is not necessary, because the verification results of the individual patterns and components still hold for the complete system. In the last phase, the platform specific model (PSM) and finally platform specific source code is synthesized in step 7.

In the next section, we present the application scenario, which is used within this paper to exemplify the application of our approach. In Section 3, we present our approach w.r.t. system structure, real-time behavior, real-time coordination, and the integration of hybrid behavior. Our approach is then compared with the UML 2.0 specification [3] in Section 4 and other related work in Section 5. We finally conclude in Section 6 and present current and future work.

2 Application Example

As concrete example, we present the design of a self-optimizing version of the software for the RailCab research project[1] which aims at using a passive track system with self-optimizing shuttles that operate individually and make independent and decentralized operational decisions. The vision of the railcab project is to provide the comfort of

[1] http://www-nbp.upb.de/en

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 on the other hand. The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in the Transrapid[2] to increase passenger comfort while still enabling high speed transportation and (re-)use of 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 a rear shuttle will no longer hold the velocity on a constant level, but the distance to the front shuttle. Therefore, it dynamically has 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. safe coordination when building or breaking convoys, 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 built.

3 Model-Driven Development with MECHATRONIC UML

For the component-based development of mechatronic systems with 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.

3.1 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 distinguish 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.

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

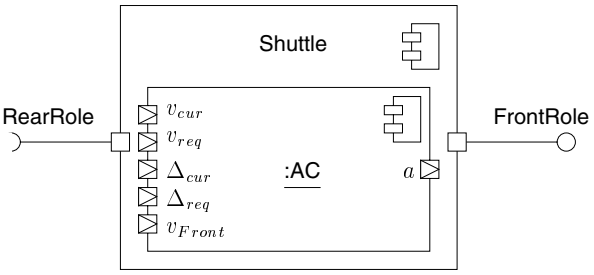

Fig. 2. Type specification of component Shuttle

For our example scenario, Figure 2 shows the component type for the shuttle. The Shuttle component contains a hybrid AccelerationControl (AC) component instance. This component computes the acceleration needed to achieve a specific goal (keeping a specified distance or keeping a specified speed level). The AccelerationControl component has five incoming continuous ports for the 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. Further, AccelerationControl has one outgoing continuous port that sends the acceleration values to the appropriate hardware actuator devices. In addition (the details are presented in Section 3.4), 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.

3.2 Real-Time Coordination

Interaction between component instances during runtime is a major part in the design of complex, reconfigurable, mechatronic systems. In our scenario, a shuttle forms a convoy with another shuttle via the RearRole and FrontRole interfaces. In the domain of mechatronic systems, 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 build a convoy has to interact only with one other shuttle and not 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. building a convoy is one such simple coordination pattern. Figure 3 shows the ConvoyCoordination pattern between two shuttles. The protocol for building and breaking convoys is specified in the roles of this pattern (see Figure 4). Components in the domain of mechatronic systems must meet real-time requirements. Therefore, we use our real-time variant of UML state machines called *Real-Time Statecharts* [11,12] for the specification of role behavior. They allow to apply constructs from timed automata [14,15] like clocks, time guards, time invariants and further annotations like worst case execution times and deadlines (see

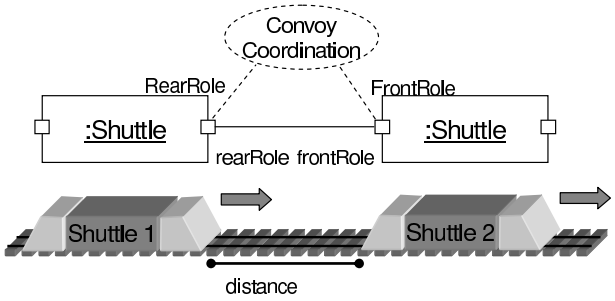

Fig. 3. Component Diagram and Patterns

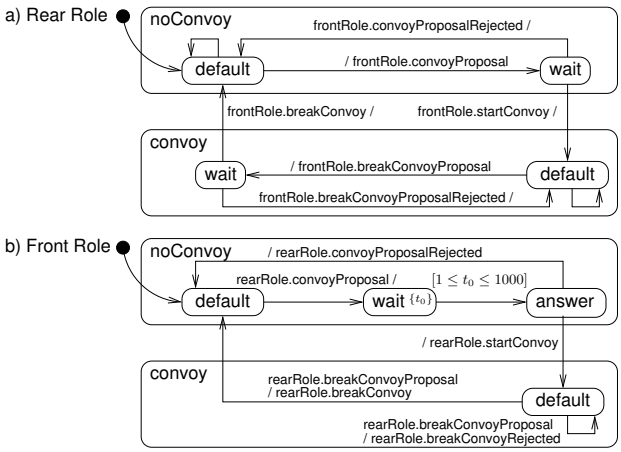

Fig. 4. Statechart of the RearRole role and the FrontRole role

Section 3.3). As shown in [11], 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.

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 building 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 states.

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 safe behavior, the following RT-OCL [16] 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, which causes a rear-end collision.

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

It is shown in [10], that this property holds. As mentioned there, those patterns are individually constructed and verified. 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 our example, the Shuttle component is a combination of refined versions of the RearRole and the FrontRole. For a component, which combines different patterns respective the roles, the verified properties still hold due to the approach presented in [10]. 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.

3.3 Local Real-Time Behavior

Figure 5 depicts the behavior of the Shuttle component from Figure 2, taken from [10] 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 Figure 4 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 whether 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 the roles from Figure 4 and does not add additional behavior.

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* [11] for specification. Real-Time Statecharts respect that the firing of transitions consumes time and that real physical, mechatronic systems can never react

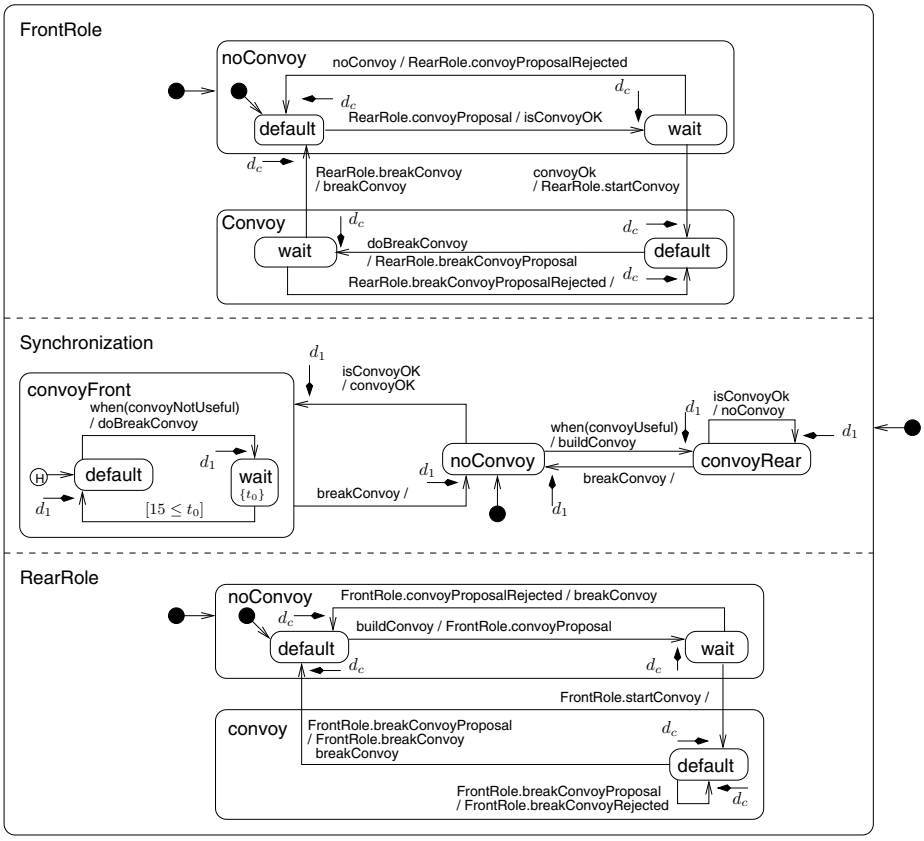

Fig. 5. Behavior of the Shuttle component

in *zero time*, but always with a delay. To represent this in the model, we make use of the deadline construct:

In Figure 5 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.

3.4 Controller Integration

The Acceleration Control (AC) component contained in the Shuttle component (cf. Figure 2) 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 6). 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 cur-

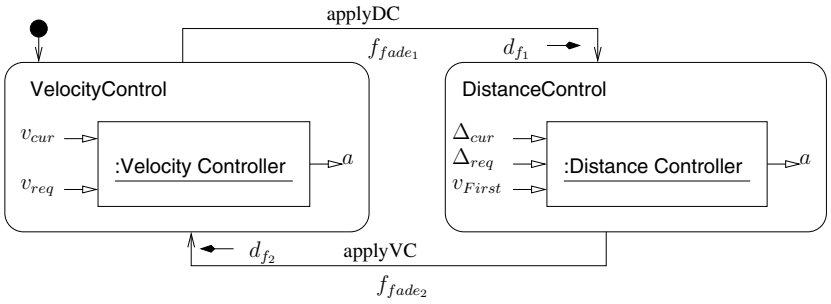

Fig. 6. Behavior of the AC component

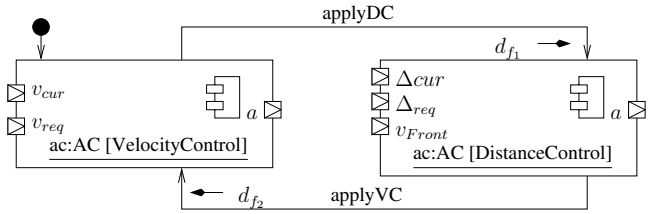

Fig. 7. Interface Statechart of the AC component

rent 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.

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 [9]. 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 excitations 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 [9]. 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 five 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 7.

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 components are associated to the discrete states is not important for the external view.

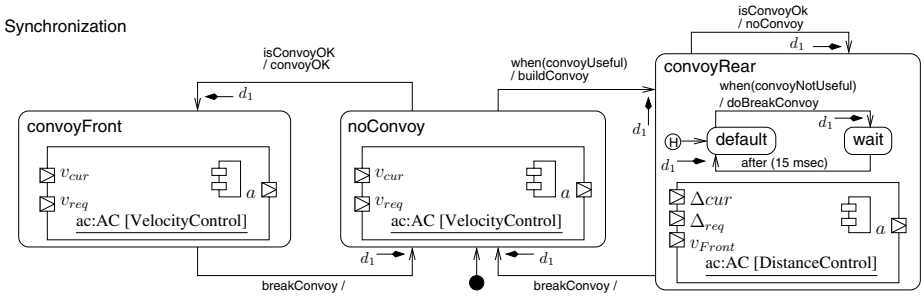

Fig. 8. Behavioral embedding

This interface representation is used when the different components are embedded into each other (see below).

The Shuttle and the AC component, which have been designed independent of each other, are embedded hierarchically from the structural point of view (cf. Figure 2). As 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 5 is extended to a Hybrid Statechart. Figure 8 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 8 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 it is embedded, the form of the embedded component is equal to the single states of the Interface Statechart from Figure 7.

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 5, the continuous behavior and the restrictions of the controller exchange is specified in Figure 6 and the later integration is specified in Figure 8. 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 3.2). The statechart from Figure 5 is a refinement of the roles from Figure 4. Consequently, it needs to be ensured that the embedding of AC still just refines the specified real-time behavior from Figure 5 and is not adding additional behavior or is in conflict with the real-time specification of this superordinated component.

Assume, for example, in Figures 5 and 8 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 7 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 hierarchically, parallel embedding, the global state space is restricted: Although Synchronization consists of 3 states and AC of 2 states, the hierarchical parallel composition does not consist of $2 * 3 = 6$ states, but just of 3 states.[3] This information is contained in the specification in Figure 8 and does not need to be derived by a costly reachability analysis. Consequently, the number of consistency checks to be enforced are thus not exponential in the number of states. If these consistency checks are successful, the results of the compositional model checking presented in Section 3.2 are valid even for components that embed further components in the hierarchical, parallel manner (cf. [9]).

4 MECHATRONIC UML and Standard UML

The UML 2.0 [3] can be considered as the currently evolving *de facto* standard for modeling complex software systems. Event though the standard UML 2.0 is not specifically tailored for technical systems, it is frequently applied also in this domain (cf. [17,4,18,19]) and actually includes most of the concepts of the Real-Time Object-Oriented Modeling (ROOM) approach [20]. However, as the ROOM concepts focus on architectural design and do not address the real-time or hybrid behavior of the operational model at all, UML supports real-time aspects only rudimentarily and hybrid behavior not at all. In the presented MECHATRONIC UML approach, the architectural design must employ standard UML components and patterns in a well-defined rigorous manner. The real-time communication protocols of each port or pattern role have to be specified. While UML 2.0 offers so called *Protocol State Machines* (PSM) to do so, we require that our real-time extension of the UML state machines named *Real-Time Statecharts* are employed.

A relevant UML extension w.r.t. real-time is the *UML Profile for Schedulability, Performance, and Time* [4]. The profile defines general resource and time models which are used to describe the real-time specific attributes of the modeling elements such as schedulability parameters or quality of service (QoS) characteristics. Besides an abstract *logic model*, a more concrete *engineering model* can be specified by using these extensions. The engineering model is later used for the required model analysis and code generation. However, appropriate concepts for the real-time modeling at the logic model level are missing and real-time aspects are only present at the level of the engineering model. Thus, the developer has to map his logical model onto the technical concepts such as threads and periods manually. Then, he has to test and adjust the logical model as well as its mapping to the engineering model manually until the engineering model mets all real-time constraints.

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

The presented approach in contrast addresses real-time aspects at the logical model level. The employed Real-Time Statecharts support deadlines, worst case execution times, clocks, clock resets, time guards, and time invariants. Therefore they provide powerful abstract means to specify complex timing requirements. A formally defined semantics for them further enables the compositional verification by means of model checking. MECHATRONIC UML thus really enables the model-driven development of real-time systems as all required timing requirements are contained in the (logical) model and the synthesis of the mapping to threads and their periods can be done automatically.

A request for proposals for *UML for System Engineering (UML for SE)* [21] by the OMG currently address UML in the context of technical systems. 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. The presented approach addresses some of these issues, but mainly focuses on the specific requirements of hybrid, reconfigurable, mechatronic systems.

One distinguishing proposal for UML for SE is the *Systems Modelling Language (SysML)*,[4] which extends a subset of the UML 2.0 specification. One extension, 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. This enables to model simple difference equations, but using this approach to model complex feedback-controllers leads to an overwhelming complexity. The specification or the integration of continuous behavior in form of continuous components is not supported. Further SysML does not support reconfiguration, as the specification of parametric relations is always static.

In contrast to UML 2.0 and the SysML proposal, our approach provides the required support for modeling of hybrid, reconfigurable systems by first refining UML ports into discrete, continuous, and hybrid ones such that hybrid components can be modeled with UML components. To specify the reconfiguration and hybrid behavior of these components, we extended Real-Time Statecharts towards Hybrid Statecharts which employ UML instance diagrams of the subordinated components to specify the state-dependent embedding and coordination. The formal definition of the embedding for Hybrid Statecharts enables to check efficiently whether an embedding is consistent. A consistent embedding further ensures, that the real-time properties, verified through compositional model checking, still hold for the more detailed hybrid system behavior.

5 Related Work

Besides UML and its profiles, a number of proprietary approaches for the modeling of technical systems with UML exist.

Within the IST project AIT-WOODDES *hierarchical timed automata* (HTA) [5] have been invented to enable the modeling and verification of complex real-time behavior. HTA are a hierarchial extension of timed automata [15] and they provide most

[4] http://www.sysml.org

of the powerful modeling concepts of statecharts as well as clocks. A mapping to multiple parallel running flat timed automata permits to verify the model by using the model checker UPPAAL [14]. Code synthesis has also been addressed in [22], however, the approach is restricted to flat automata and does not take into account the delays that occur when transitions are fired. Our approach for code generation respects hierarchy, parallelism, and the real-time specifications [11,23].

The aim of the IST OMEGA project [6] is to ensure the correctness of embedded systems. In the approach, the UML has been extended by additional time constructs and a formally defined semantics is intended. However, unlike our approach, there is no support for hybrid behavior and compositional verification. Verification is only supported for the semi-automatic verification via theorem proving.

Like the presented approach, HyROOM [7] and the underlying HyCharts [8] support the component-based modeling of hybrid systems. The software's architecture is specified similar to ROOM/UML-RT and the behavior is specified by statecharts whose states are associated with systems of ordinary differential equations and differential constraints or Matlab/Simulink block diagrams. These approaches provide means for the reconfiguration of systems in terms of changing the continuous behavior. But it is only possible to reconfigure the model inside a component on one hierarchy-level. In contrast to that, our approach allows for a complex reconfiguration altering the structure and concerning more than one hierarchy-level. Support for compositional verification of models is not addressed by any of these approaches.

6 Conclusion and Future Work

Reconfigurable mechatronic systems in the domain of safety-critical distributed systems must be designed with great care. MECHATRONIC UML not only supports the model-driven development of such systems respecting real-time requirements, 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, source code is synthesized from the models, which respects the real-time constraints and safety requirements of the model.

MECHATRONIC UML further refines the industry standard UML where possible and provides a well defined UML subset as well as a guideline how to develop safety-critical reconfigurable mechatronic systems.

Tool support (in form of a number of plug-ins for the Fujaba Tool Suite[5]) for the specification, verification and automatic source code synthesis of the Real-Time Statecharts and the real-time coordination patterns exists. For the support of hybrid behavior a prototypic implementation exists and we are currently working on the tool support.

In the future, we plan to employ graph transformations [24] 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 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.

[5] http://www.fujaba.de

We further plan to integrate MECHATRONIC UML with our approaches for automatic deployment [25] and dependability [26] with UML.

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

References

1. Bradley, D., Seward, D., Dawson, D., Burge, S.: Mechatronics. Stanley Thornes (2000)
2. Object Management Group: Model Driven Architecture (MDA) Edited by Joaquin Miller and Jishnu Mukerji. (2001)
3. Object Management Group: UML 2.0 Superstructure Specification. (2003) Document ptc/03-08-02.
4. OMG: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (2002)
5. 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
6. Graf, S., Hooman, J.: Correct Development of Embedded Systems. In Oquendo, F., Warboys, B., Morrision, R., eds.: Proceedings of the First European Workshop on Software Architecture, EWSA2004. Volume 3047 of Lecture Notes in Computer Science., St Andrews, UK, Springer Verlag (2004) 241–249
7. Stauner, T., Pretschner, A., Péter, I.: Approaching a Discrete-Continuous UML: Tool Support and Formalization. In: Proc. UML'2001 workshop on Practical UML-Based Rigorous Development Methods – Countering or Integrating the eXtremists, Toronto, Canada (2001) 242–257
8. Stauner, T.: Systematic Development of Hybrid Systems. PhD thesis, Technische Universität München (2001)
9. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA, ACM (2004)
10. 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)
11. Giese, H., Burmester, S.: Real-Time Statechart Semantics. Technical Report tr-ri-03-239, University of Paderborn, Paderborn, Germany (2003)
12. Burmester, S., Giese, H.: The Fujaba Real-Time Statechart PlugIn. In: Proc. of the Fujaba Days 2003, Kassel, Germany. (2003)
13. Ogata, K.: Modern Control Engineering. Prentice Hall (2002)
14. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology **1** (1997)
15. 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)
16. 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

17. 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
18. 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)
19. 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)
20. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley and Sons, Inc. (1994)
21. Object Management Group: UML for System Engineering Request for Proposal. (2003) Document ad/03-03-41.
22. 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)
23. Burmester, S., Giese, H., Gambuzza, A., Oberschelp, O.: Partitioning and Modular Code Synthesis for Reconfigurable Mechatronic Software Components. In: Proc. of European Simulation and Modelling Conference (ESMc'2004), Paris, France. (2004) (accepted).
24. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1. World Scientific, Singapore (1999)
25. Tichy, M., Schilling, D., Giese, H.: Design of Self-Managing Dependable Systems with UML and Fault Tolerance Patterns. In: Proc. of the Workshop on Self-Managed Systems (WOSS) 2004, FSE 2004 Workshop, Newport Beach, USA. (2004)
26. 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. Volume 3069 of Lecture Notes in Computer Science. Springer Verlag (2004) 25–51