Generating Modelica Models from Software Specifications for the Simulation of Cyber-physical Systems

Uwe Pohlmann, Jörg Holtmann, Matthias Meyer
Software Engineering,
Project Group Mechatronic Systems Design,
Fraunhofer Institute for Production Technology IPT
Zukunftsmeile 1, 33102 Paderborn, Germany
Email: [uwe.pohlmann | joerg.holtmann | matthias.meyer] @ipt.fraunhofer.de

Christopher Gerking
Software Engineering Group,
Heinz Nixdorf Institute,
University of Paderborn
Zukunftsmeile 1, 33102 Paderborn, Germany
Email: christopher.gerking@upb.de

***Abstract*—Future smart systems will provide functionality by dynamically interacting with each other in cyber-physical systems. Such interactions require a message-based coordination under hard real-time constraints. This is realized by complex software, which combines discrete, state-based behavior with continuous behavior controlling the dynamics of the physical system parts. The development methods and tools for these kinds of software are not well integrated so far. For the modeling and simulation of physical and continuous control behavior, Modelica can be used. For modeling the discrete coordination behavior, MECHATRONICUML (MUML) can be used, which in addition offers a formal verification of safety requirements like deadlock-freedom of interactions, for example. We introduce in this paper an automatic transformation for formally verified MUML models into Modelica to ensure that the discrete state-based software correctly interacts with the continuous control software, physical parts, and a plant model. We illustrate this concept by means of a car-to-car coordination scenario.**

I. INTRODUCTION

The next generation of smart systems like cars, production machines, or energy systems will no longer operate in isolation but heavily interact with each other and their environment in so-called cyber-physical systems (CPS). The key aspect of CPS is the provision of sophisticated functionality by an extensive collaboration of the individual systems under hard real-time constraints, which is realized by complex software. On the one hand, the software comprises feedback controllers, e.g., electronic stability control, which continuously controls the dynamic behavior of the physical parts of the systems. This *control behavior* is specified by means of continuous models based on differential equations. On the other hand, a growing part of the software is dedicated to the message-based coordination of systems, e.g., car-2-car communication. This *coordination behavior* is specified by means of discrete, state-based models. Especially, the interplay of the control and the coordination behavior needs to be designed and verified very carefully to avoid failures in safety-critical applications.

For the design of the control behavior, state-of-the-art tools and languages like MATLAB Simulink/Stateflow or Modelica can be used. They allow building a model of the physical parts of a system and their dynamic behavior together with feedback controllers. The feedback controllers can then be validated by means of simulation.

However, since such languages and tools lack sufficient support for modeling and formal verification of the coordination behavior, we are developing the MECHATRONIC-UML (MUML) method [4], [3]. MUML provides a software development process for CPS, starting with formal requirements specification and ending with target-code generation. Furthermore, MUML provides a language that separates the different software development concerns, i.e., control behavior and coordination behavior. The language allows to model the coordination behavior by building component-based software models with message-based interactions. It takes hard real-time constraints into account and defines clear interfaces to continuous components providing the control behavior.

As a key feature, MUML allows to formally verify whether the models providing coordination behavior are guaranteed to fulfill safety requirements like, e.g., deadlock-freedom of interactions, using timed model checking. The formal verification exploits on the one hand the interface definition between the coordination behavior and the control behavior in order to verify the coordination behavior automatically and efficiently. On the other hand, it exploits assumptions about the control behavior that are guaranteed by extensive manual simulations.

To ensure the fulfillment of the assumption that the control behavior together with the physics preserves the formally verified, timed coordination behavior, we present in this paper an approach for validating the interconnected discrete-continuous behavior by means of a *hybrid simulation*. Therefore, we automatically transform MUML models to Modelica. In [20] we used MUML modeling concepts to build up the Modelica Real-Time Coordination Library. However, this does not include the formal verification of models. In other previous work, we already presented a similar transformation to MATLAB Simulink/Stateflow [13] and a transformation to C-Code with a functional mockup interface (FMI) [21]. In contrast to these approaches that use ordinary differential equations, Modelica "has significant advantages, particularly for specifying physical models based on differential-algebraic equations" [14]. Furthermore, we worked on a transformation of ModelicaML to

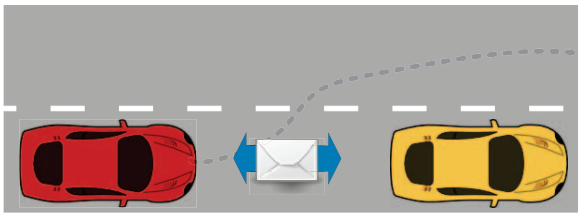

Fig. 1. Overtaking Coordination by means of Asynchronous Message Exchange

Modelica [23], [19]. In contrast to the ModelicaML approach that generates plain Modelica algorithmic code, we combine in this paper the use of existing Modelica libraries with model transformation techniques.

The contribution of this paper is an automatic transformation of a formally verified software specification to Modelica involving existing Modelica libraries. As a result, the coordination behavior of cyber-physical systems can be simulated integrated with the control behavior, physical parts, and a plant model. We provide a toolchain including editors for modeling and automatic model-to-model and model-to-text transformations.

To illustrate our approach, we use a car-to-car communication scenario as a running example. Two autonomous cars coordinate their overtaking behavior by means of an asynchronous message exchange protocol. The coordination addresses the safety-critical nature of overtaking maneuvers: it excludes a potential cause of collisions by ensuring that, whenever the rear car overtakes, the front car does not accelerate. For this purpose, both cars are equipped with coordination software. Coordination is achieved through asynchronous message exchange, excluding unsafe state combinations during operation.

Fig. 1 illustrates the coordination behavior for overtaking maneuvers. To initiate the overtaking, a car may send an overtaking request to another car in front. The front car may accept or decline the initial request. In case of acceptance, the front car immediately guarantees to drive with a constant speed. Thus, it does not only enable overtaking by the rear car, but also prevents rear-end collisions due to sudden braking while the rear car accelerates. Consequently, the rear car starts to accelerate when receiving the acceptance message. When changing the lane for overtaking, it informs the front car by means of an asynchronous message. On reception of this message, the front car still guarantees not to accelerate, but can brake safely without a possible rear-end collision. When the former rear car finishes overtaking by changing the lane back, it notifies the former front car by means of another message. On reception of this message, the maneuver is entirely finished and the former front car is allowed to accelerate again.

In the following section, we use the coordinated overtaking example to describe the MUML method. Afterwards, in Sec. III, we introduce the Real-Time Coordination Library, on which we build our transformation from MUML to Modelica. Sec. IV describes the development process using our approach, and in Sec. V we present the mapping of our software specification to Modelica. Sec. VI covers related work. Finally, in Sec. VII, we conclude and discuss future work.

II. MechatronicUML Design Method

The safety-critical nature of today's CPS imposes hard real-time constraints on the embedded software. These constraints require consideration at an early stage of the design process. In this section, we present the MUML method [4], [3] for the software design of CPS. MUML comprises a domain-specific process and modeling language (DSL) with a focus on coordination behavior, including an asynchronous message exchange. However, apart from that, the DSL also enables a tight integration of these discrete aspects with continuous control behavior.

MUML supports the early analysis of its domain-specific design models: on the one hand, timed model checking is applicable to the discrete system part, providing a formal verification of the coordination behavior with respect to safety-critical requirements. On the other hand, after the successful verification of the coordination behavior, the integration with continuous aspects is subject to a simulative validation. Both formal verification and simulation are carried out by exporting the domain-specific models to commercial off-the-shelf analysis tools. MUML's DSL comprises various modeling concepts: on the one hand, it enables structural modeling in terms of interface description, component types, and component instance configurations (CIC). On the other hand, MUML provides so-called Real-Time Statecharts for the modeling of real-time coordination behavior. Similar to the UML Alf language [17], MUML comprises an action language for the specification of operations inside Real-Time Statecharts. We provide tool support for MUML in terms of the Fujaba Real-Time Tool Suite[1].

Fig. 2 illustrates a MUML model of the overtaking coordination for autonomous cars. The upper part of the figure comprises a CIC, consisting of two discrete component instances and three continuous component instances. For simplicity reasons, the figure omits the component type specification and port interface description. Each discrete component contains the state-based behavior for one of the involved cars. Each discrete component is equipped with a discrete port that sends and receives discrete messages to/from the counterpart. Since MUML reflects an asynchronous message exchange, the sender does not block until the receiver will process the message. Furthermore, the receiver buffers received messages and therefore supports later processing.

In addition to the discrete ports for the asynchronous message exchange, each of the two discrete component instances in Fig. 2 incorporates two hybrid ports to control the continuous velocity of the left and right wheels separately. The redSW component instance also comprises an additional hybrid port in order to sense the distance to the front car. Fig. 2 comprises three continuous component instances, representing controllers, physical parts, or plant model. Whereas these component instances interact with the discrete system part via hybrid ports, their behavior is out of the scope of modeling with MUML.

The lower part of Fig. 2 illustrates MUML's modeling approach for the coordination behavior. MUML employs

[1] Eclipse update site: http://muml-ci.cs.upb.de/job/MUML/lastSuccessfulBuild/artifact/targetPlatform/, Wiki: https://trac.cs.upb.de/mechatronicuml/

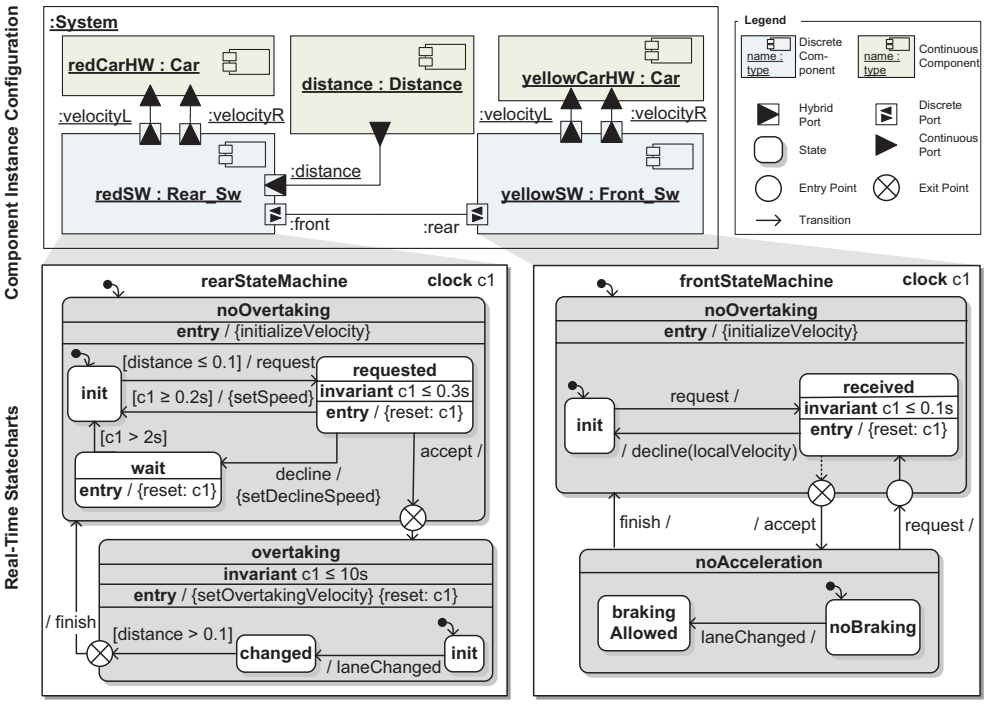

Fig. 2. MUML Model of the Overtaking Coordination

Real-Time Statecharts to specify the impact of the discrete message exchange on the continuous controlling strategy. Real-Time Statecharts integrate UML state machines with real-time concepts known from timed automata [1]. In particular, a Real-Time Statechart may access an arbitrary number of real-valued clocks. These clocks enable the design of state-based coordination behavior that is sensitive to specific real-time constraints. An individual Real-Time Statechart is attached to each discrete component instance in Fig. 2 and controls the port's asynchronous message exchange behavior. Furthermore, Real-Time Statecharts read or write the values of the hybrid ports.

In the following, we describe how the Real-Time Statecharts in Fig. 2 resemble the coordination behavior of our car-2-car communication scenario. Each of the statecharts comprises a clock named c1 to keep track of particular real-time constraints. If the rear car senses a front distance of less than 0.1 over its hybrid distance port, it sends an overtaking request message over its discrete front port to the front car, and switches from state init to requested. At the same time, it resets the clock c1 to 0 as part of requested's entry event. Consequently, the invariant $c1 \leq 0.3s$ describes the real-time constraint that state requested must be left again within 0.3 seconds. This can happen either due to evaluating the clock constraint $c1 \geq 0.2s$ to true (i.e., a timeout), or due to an appropriate accept or decline response, which is received on time.

When receiving the initial request message, the front car switches from state init to received. After resetting the clock c1 as part of received's entry event, the invariant $c1 \leq 0.1s$ expresses the maximum time that the front car may spend on evaluating the request. We focus on the case of a successful overtaking maneuver, i.e., the front car changes from state received to noAcceleration, leaving the hierarchical noOvertaking state via an exit point. At the same time, it responds to the request by means of an accept message.

When the rear car receives the accept message on time, i.e., by no later than 0.2 seconds after the request has been sent, it starts to overtake by deactivating noOvertaking via an exit point and switching to state overtaking. As a consequence, the setOvertakingVelocity action increases the velocity values of the hybrid velocityL and velocityR ports, such that the car changes the lane for overtaking. The rear car signalizes the completed lane change by means of a laneChanged message, and switches to changed. If the overtaken front car receives this message, it can brake safely and therefore switches to brakingAllowed, still ensuring to not accelerate in any unsafe way. Finally, if the overtaking rear car senses a distance greater than 0.1, it finishes the maneuver by deactivating the overtaking state via an exit point and switching back to noOvertaking. On reception of the finish message, the overtaken front car also switches back to noOvertaking. The entry actions of the noOvertaking states set the velocity for both cars back to the initial values.

III. Modelica Real-Time Coordination Library

In previous work, we developed the *Real-Time Coordination Library*[2] [20] for modeling coordination behavior in Modelica. The Real-Time Coordination Library is heavily used and partly extended in this paper for the automatic transformation from MUML to Modelica. Modelica is a multi-domain, object-oriented language for developing and simulating complex CPS. Its mathematical model is based on hybrid differential

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

algebraic equations. Dymola is the major simulation tool. "The fundamental structuring unit of modeling in Modelica is the class. Classes provide the structure for objects, also known as instances. Classes can contain equations [and algorithms], which provide the basis for the executable code that is used for computation in Modelica. [...] Connections between objects are introduced by connect-equations in the equation part of a class." [16] Modelica has its strength in modeling and simulation of complex and large physical systems [14]. Therefore, it is used by many system engineers to develop physically realistic models.

Modelica 3.2 has no own language part for state-based coordination behavior. It offers the StateGraph2 Library [18] to model coordination behavior as a structure of objects. StateGraph2 consists of a class Step for states of automaton and a class Transition for state changes. Developers can build up state machines by instantiating these classes as objects, and connecting the corresponding class interfaces by Modelica connectors. Many automaton formalisms synchronize their behavior by events, or exchange information by messages that are events in combination with certain data. As StateGraph2 lacks support for any message-based communication, our library extends StateGraph2. The concepts and the formal model of the Real-Time Coordination Library are similar to Real-Time Statecharts from MUML. The Real-Time Coordination Library provides support for asynchronous communication in terms of a class Message that can transport a certain number of variable values. The firing state machine can immediately proceed without having to wait for an answer. It is also possible to receive messages in any active state configuration, as they are queued in mailboxes. A mailbox refers to a certain maximum number of messages and may have displacement strategies. The receiver state machine decides when it consumes a message from the mailboxes. Since messages can be sent via different hierarchy levels and between different components, the classes InputDelegationPort and OutputDelegationPort can be used to route a message. Currently, it is not possible to send different messages over the same port. Additionally, the Real-Time Coordination Library provides rich modeling of real-time behavior. Therefore, the classes Clock, TimeInvariantLess(OrEqual), ClockConstraint-Less(OrEqual), ClockConstraintGreater(OrEqual) are available in the library.

Fig. 3 shows an excerpt of our scenario implemented in Modelica using the StateGraph2 Library and the Real-Time Coordination Library. In the model, the rear car waits for an Accept message of the front car to start the overtaking. If the message arrives via the InAccept delegation port, it is queued in the AcceptBox. If state NoOvertaking is active and an Accept message is in the mailbox, then transition T1 fires. As a result, Modelica deactivates the state NoOvertaking and activates the state Overtaking. If the rear car has finished the overtaking maneuver, transition T2 fires and the message Finish is sent via the delegation port OutFinish to the other car.

Fig. 4 shows the sub-state machine of state noOvertaking. This figure introduces the elements Clock, Clock Constraint, and State Invariant for specifying real-time behavior. The invariant restricts the amount of time in which the state Requested is allowed to be active. Therefore, the value of clock c1 must be at most 0.3s. If a simulation run violates

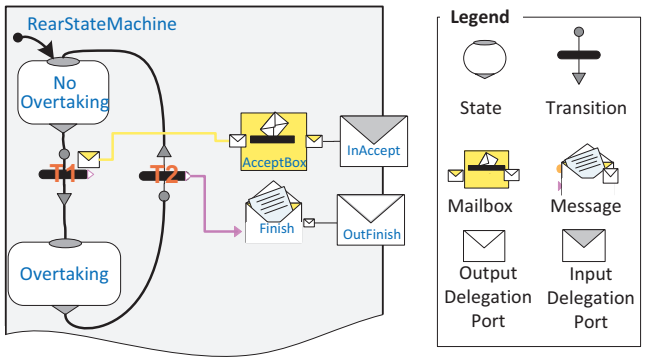

Fig. 3. A Part of the rearStateMachine (cf. Fig. 2) Modeled with the Real-Time Coordination Library

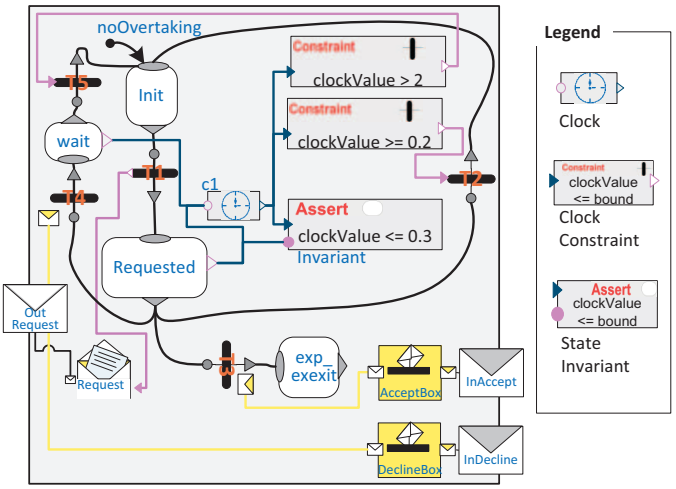

Fig. 4. NoOvertaking Sub-State Machine with Clock, Clock Constraint, and State Invariant

this constraint, it stops at this point in time and signalizes an error to be fixed by developers. State Requested and state wait are connected with the clock c1, which means that both states reset this clock to zero when one of it gets active. The clock constraints restrict the time in which the transitions T2 and T5 are allowed to fire. In the example, T2 is only allowed to fire if the value of c1 is greater or equal to 0.2s. T5 is only allowed to fire if the value of c1 is greater to 2s.

IV. Development Process

In this section, we sketch the development process that we perform for modeling and analyzing the overtaking scenario with our approach. Fig. 5 depicts this process. In step 1, we determine the discrete components for our scenario and assemble them. We check static semantics constraints of this model. In step 2, we define the coordination behavior between the front and rear car. The MUML design method enables to prove specific safety requirements fulfilled by the overtaking coordination. Safety requirements describe certain runtime configurations of the state-based behavior, which must be present or absent on every possible execution path and at each point in time. For example, the overtaking coordination behavior guarantees that, whenever the rear car is in state overtaking, the front car is in state noAcceleration. In MUML, we employ the timed model checker Uppaal to

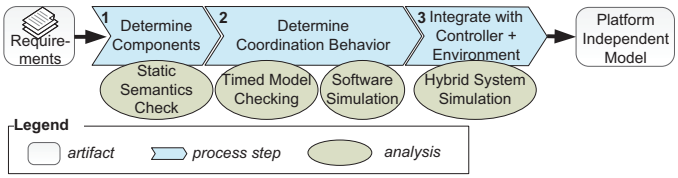

Fig. 5. Development Process with Automatic Analysis Techniques

prove such requirements automatically [11]. Using UPPAAL, we also prove further safety requirements, such as the absence of deadlocks and message buffer overflows in the above coordination behavior. Further, we make a first transformation to Modelica and simulate the discrete coordination behavior in Dymola to test our software specification. Afterwards, in step 3, we integrate the discrete software with continuous controllers, physical parts such as the hardware of the car, and a plant model. We define the interface from discrete components to continuous components, and vice versa, by adding input and output ports. Furthermore, we use the values of the new input ports in our state machine for guards (e.g., ([distance $\leq$ 0.1] in the transition from state requested to state init of the rearStateMachine in Fig. 2). Additionally, we set the values of the output ports by state and transition actions.

Afterwards, we transform this final MUML model to Modelica. We load the Modelica model with Dymola. In Dymola, we specify the continuous behavior of the car hardware and add ports and connections between the continuous components. Finally, we simulate the integrated, discrete-continuous model and analyze this coordination and control behavior by means of hybrid simulation.

V. TRANSFORMATION FROM MECHATRONICUML TO MODELICA

Fig. 6 sketches the overall toolchain for the transformation of MUML models specified in the FUJABA Real-Time Tool Suite to UPPAAL, Graphviz[3], and Dymola. A test version of our transformation is available in the FUJABA Real-Time Tool Suite[4]. In the following, we explain the particular transformation steps within this toolchain.

The left-hand side of Fig. 6 indicates the particular parts of a typical MUML model as described in Sec. II. In order to formally verify that the coordination behavior excludes deadlocks and unsafe state combinations, we use the timed model checker UPPAAL. UPPAAL performs a formal verification of safety requirements on timed automata. We refer to [6] for a detailed presentation of model checking principles. As a result of the transformation to UPPAAL, we can prove the absence of violated safety requirements under the assumption that the continuous behavior in interplay with the physics preserves the timed state-based behavior. We transform MUML Real-Time Statecharts to UPPAAL to verify specified safety requirements [11]. The safety requirement that no deadlocks are allowed is sketched in Fig. 6 by a red annotation. The safety requirements are specified by a dialect of TCTL. If a violation of a safety requirement is detected, UPPAAL provides a corresponding counterexample. Since the transformation towards UPPAAL is out of the scope of this paper, the corresponding elements in Fig. 6 are faded out.

In order to improve the understandability of the generated Modelica code, we perform a transformation to and from the external layout tool Graphviz (step 1 in Fig. 6) before the actual transformation from MUML to Modelica. Without layout annotations, it would be more difficult for the engineer to understand the model within Dymola. First, the MUML model is transformed to the DOT language of Graphviz. Graphviz automatically calculates a layout. Afterwards, a back-transformation from Graphviz annotates the particular MUML model elements with calculated layout information. Fig. 6 sketches the layout annotations as blue elements. This layout information is utilized in the subsequent transformation to Modelica for generating the Modelica graphical annotations.

To ensure the fulfillment of the assumption that the control behavior together with the physics preserves the timed coordination behavior, we validate the integrated overall behavior by hybrid simulation of the system. Therefore, we transform the complete MUML model to Modelica code for Dymola in the transformation step 2 in Fig. 6. As transformation language, we use the model-to-text transformation framework Acceleo[5]. We present this transformation in detail in the following. We encompass the component instance configuration, consisting of discrete components (i.e., containing state-based coordination behavior) as well as continuous components (i.e., stubs for the control behavior to be embellished within Dymola).

In the following section, we describe how we map the structural MUML model parts to Modelica, while Section V-B explains the mapping of MUML behavior.

A. Transformation of Structural Model Parts

For each discrete software component instance, like YellowSW, we create a Modelica model class. For example, Fig. 7 shows the mapping of the MUML software component instance yellowSW to the Modelica class yellowSW. For each sender and receiver message of each port we create corresponding OutputDelegationPort and InputDelegationPort from the Real-Time Coordination Library. Further, we create for the hybrid ports corresponding Modelica ports. Finally, we instantiate the behavior of the component and connect the outer and inner ports with each other (cf. next section).

Additional to the component transformation we have to transform the whole component instance configuration (CIC) of a MUML model. We transform the MUML CIC (cf. Fig. 2) into a Modelica model class that has the same name. For example, Fig. 8 shows the Modelica connection diagram of the class System. The class instantiates all contained component instances of the CIC as objects. For hierarchically structured components, we create classes for embedded components, instantiate these classes and connect the instance ports. Finally, the ports of the component instances are connected via Modelica connect statements. Therefore, firstly the message ports are connected. Each message of each port gets an own connection. Secondly, the outgoing ports of a discrete component are connected with the incoming ports of a continuous component. Thirdly, the incoming ports of a discrete component are connected with the outgoing ports of a continuous component.

[3] http://www.graphviz.org/
[4] https://trac.cs.upb.de/mechatronicuml/wiki/SEAA2014
[5] http://www.eclipse.org/acceleo/

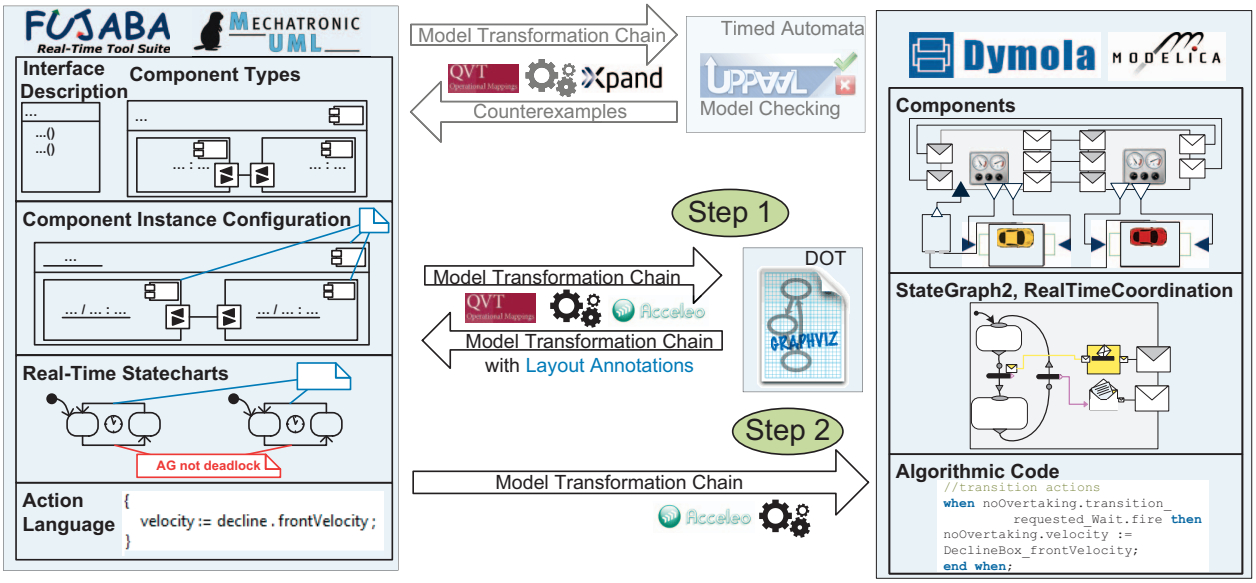

Fig. 6. Toolchain for Transformation from FUJABA Real-Time Tool Suite to UPPAAL, Graphviz, and Dymola

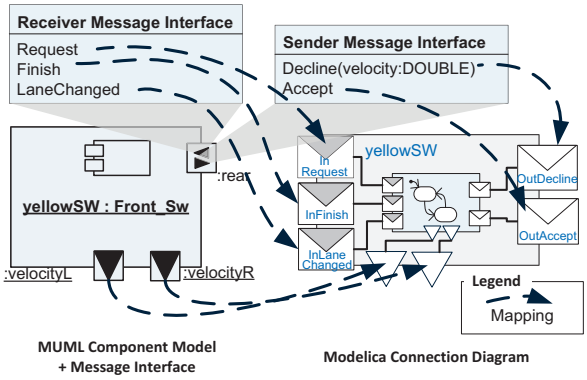

Fig. 7. Mapping of a MUML Component to a Modelica Class

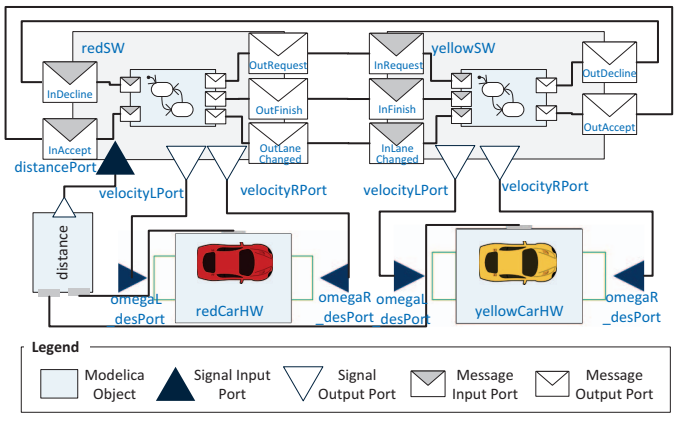

Fig. 8. Modelica Connection Diagram of Class System

B. Transformation of Behavioral Model Parts

For each Real-Time Statechart, we generate an individual Modelica class. The states of Real-Time Statecharts can be hierarchical and can contain several parallel subordinate state machines, which refer to a new state machine. Therefore, at the Modelica level, we obtain the following recursive tree structure of Modelica classes: "StateMachine–States–StateMachine".

Fig. 9 shows the MUML noOvertaking state machine and the same state machine represented in Modelica. The mapping of the states and entry/exit-points is indicated by the same color. The mapping of the clock, actions, and messages is indicated by the dashed arrows. We declare the interfaces for hybrid ports and delegation variables for reading access. Afterwards, we define all sub-states of the state machine. noOvertaking contains the states init and received, and transitions between these states. To specify timing behavior, we define the clock c1 and therefore use the corresponding class of our Real-Time Coordination Library. As state machines can receive messages, they also get delegation message ports for received and sent messages. We connect the Modelica object ports with each other. For example, the clock c1 should be reset when the sub-state noOvertaking.init gets active. Therefore, we connect the activePort with the input port u of the clock.

Finally, we generate code for all actions that can be invoked at transitions and in states. All sub-states of a state machine have reading access to variables, which are declared in the state machine. Furthermore, they can read the values of incoming hybrid ports. As Modelica only allows single assignment of variables, we write the variables in the algorithmic part of the top-level state machine class. We transfer the MUML action language expressions to Modelica algorithmic code. Algorithmic code allows imperative programming in Modelica, in contrast to equations that have a declarative semantics. Fig. 9 shows the mapping of the entry action of the state noOvertaking to Modelica code. The Modelica when statement is executed only once, when the condition NoOvertaking.active becomes true. The invoke conditions for actions are the fire attribute of a transition or the active attribute of a state.

For generating the layout information of the state machines elements in Modelica, we use the DOT language of Graphviz.

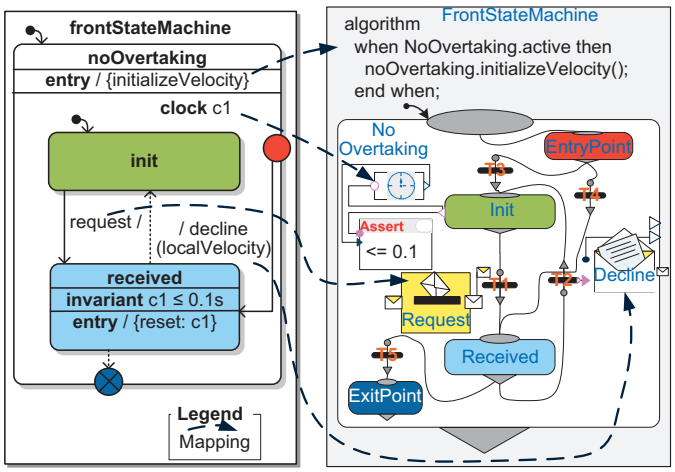

Fig. 9. Mapping of a MUML Real-Time Statechart to a Modelica Class

Graphviz generates a text file with layout information as output. We parse this output to annotate the MUML Real-Time Statecharts. The transformation from MUML to Modelica uses this information to generate the Modelica annotations for the graphical placement of objects and the routing of connections. We represent each state as a DOT node. We do the same for transitions, since transitions are individual objects in Modelica. Furthermore, we create edges between states and transitions as they are required for connect statements in Modelica. Additionally, required mailboxes, messages, and port-objects are transformed to nodes and connections between these elements as edges.

C. Discussion

During our proof of concept, we detected the following limitations and issues of our current approach. First, the transformation of MUML to Modelica only supports Modelica primitive data types and arrays where all elements have the same primitive type. Therefore, it is not possible to use self-defined Modelica records or SI-Units from the Modelica Standard Library. Second, all transitions at the Modelica level have a minimum delay of $1\mu s$ to avoid algebraic loops. Therefore, it is not possible to fire more than one transition in one state machine per μs.

A further issue is that MUML supports variable component and port instantiation at the architectural level. In contrast, Modelica only support static types during runtime. Therefore, we generate a new type class for each MUML component instance. The generation of Modelica models may be improved by using the Modelica conditional component concept to vary the number of components and ports on the instance level. Unfortunately, this Modelica concept is not well documented. We plan to investigate this aspect in future work.

VI. Related Work

Donath et al. [8] provides a domain-specific language for state machines in SimulationX, which is based on UML. They use a model-to-text transformation to transform these models into Modelica algorithmic code. In contrast to our approach they do not cover the needs of software engineers in the domain of safety-critical real-time systems coordinating each other. Among other things, component-based development, message-based communication, timed behavior, and formal verification belong to these needs. Furthermore, the transformed state machines are not visualized in Modelica.

Dempsey offers a toolchain from MATLAB Simulink models into Modelica using Simelica and the AdvancedBlocks library [7]. Unfortunately, Stateflow blocks are not supported, which are required to model state-based software behavior.

Lundvall et al. [15] propose an algorithm and toolchain to translate a Modelica model into a format for formal model checkers. The developer models on a low level and has much freedom in Modelica. Therefore, the resulting state space can get very large in model checkers. As a result, model checking takes very long time or is not possible at all. To prevent a state explosion, we define in our component model clear interfaces. The interfaces decouple the system parts and enable a compositional model checking [12]. Further, we use more abstract models with less freedom, which results in a smaller state space.

Otter et al. propose a transformation of StateGraph2 to NuSMV [18] for formal verification. In contrast to our approach that uses UPPAAL for timed model checking, NuSMV is only able to verify untimed behavior. This invalidates the approach for the use in the context of real-time CPS.

Dressler [9] describes a toolchain from JGrafchart to Modelica. JGrafchart is a tool to model state machines. The state machine elements are transferred into Modelica classes and objects without the usage of libraries. The state-based behavior is transferred into Modelica algorithmic code. The layout of the state machines is transferred from the JGrafchart diagrams into Modelica annotations. In contrast to our approach no external tool is used for the calculating a new layout. Toolchains for (timed) model checking are also available for JGrafchart [2]. In contrast to our approach, JGrafchart does not support component-based software development with message-based communication.

Elmqvist et al. [10] introduced a new concept in Modelica 3.3 for state machines. In contrast to StateGraph2 that uses Modelica objects, this is a new sublanguage for state machines in Modelica. Up to now, there is no support of message-based communication between state machines. Therefore, we do not use this new formalism as the implementation form for state-based behavior. Currently, there is no transformation to a timed model checker available to verify safety requirements.

VII. Conclusions and Future Work

This paper presents an automatic model transformation to Modelica and a description how we integrate the transformation into an existing toolchain. The toolchain enables the combination of formal verification by timed model-checking and validation by Modelica simulation required in the software development for CPS. We specify formal component-based software specifications with purely stated-based behavior by means of MUML using the Fujaba Real-Time Tool Suite. For the fulfillment of safety requirements, we formally verify this coordination behavior by means of the Uppaal model checker. The introduced transformation to Modelica covers the component-based structure including message interfaces,

the state-based component behavior, the interfaces between the coordination and the control behavior, as well as the computation of layout information. In Dymola, we embellish the continuous control behavior, the physical parts, and the plant model. We perform a validation by means of a hybrid simulation in Dymola of the resulting overall behavior model.

Our modeling method MUML allows the application of convenient high-level modeling constructs for the software specification with a domain-specific process and in a modeling language that is well-suited for software engineers. The formal verification of these software specifications proves the absence of safety requirements violations within the coordination behavior. The automatic model transformation to Modelica saves time-consuming and error-prone manual work for remodeling the corresponding information. The visualization of the information within Modelica based on the computed layout information helps the engineer to easily understand the model, and the automatic layout calculation minimizes the effort for rearranging the Modelica diagrams. In addition to the formal verification of the coordination behavior, the validation by means of simulation within Dymola allows to investigate the integrated CPS behavior w.r.t. its environment in a holistic manner. Therefore, our toolchain closes the gap between these two complementary analysis techniques.

In the future, we plan to extend our toolchain into different directions. Besides taking care of the open issues mentioned in Sec. V-C, this encompasses the following aspects. First, we want to conceive a systematic testing methodology that defines test scenarios for Modelica models based on the CPS requirements (e.g., [22]). Second, we plan to foster traceability in the toolchain as demanded by typical development process models and standards for CPS (e.g., Automotive SPICE or ISO 26262 in the automotive sector) to provide a more convenient support for the software engineer. On the one hand, this encompasses the traceability between the different models among each other, for example, the navigation from Modelica to a corresponding model element in MUML if changes are required. On the other hand, this covers the investigation of Dymola simulation runs within MUML and the annotation of MUML model elements with simulation results from Dymola (cf. [11], for example). Finally, we want to analyze if we could extend Modelica 3.3 state machines [10] for our use cases from pure synchronous state machines to a globally asynchronous locally synchronous (GALS) state machines [5].

Acknowledgments

This research and development project is funded by the German Federal Ministry of Education and Research within the Leading-Edge Cluster “Intelligent Technical Systems OstWestfalenLippe” (it’s OWL) and managed by the Project Management Agency Karlsruhe (PTKA). We thank Marcus Hüwe and Boris Wolf for conceptions and implementation.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[2] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. Verification of plc programs given as sequential function charts. In *Integration of Software Specification Techniques for Applications in Engineering*, pages 517–540. Springer, 2004.

[3] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann. The MechatronicUML method: Model-driven software engineering of self-adaptive mechatronic systems. In *Proceedings of the 36th International Conference on Software Engineering (Posters)*, ICSE’14. ACM, May 2014.

[4] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, S. Thiele, W. Schäfer, M. Meyer, U. Pohlmann, C. Priesterjahn, and M. Tichy. The MechatronicUML design method - process and language for platform-independent modeling. Technical Report tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn, Mar. 2014. Version 0.4.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[7] M. Dempsey. Automatic translation of Simulink models into Modelica using Simelica and the AdvancedBlocks library. In *Proc. of the 3rd Int. Modelica Conf.*, 2003.

[8] U. Donath, J. Haufe, T. Blochwitz, and T. Neidhold. A new approach for modeling and verification of discrete control components within a Modelica environment. In *Proc. of the 6th Int. Modelica Conf.*, pages 269–276, 2008.

[9] I. Dressler. Code generation from JGrafchart to Modelica. Master’s Thesis ISRN LUTFD2/TFRT--5726--SE, Department of Automatic Control, Lund University, Sweden, march 2004.

[10] H. Elmqvist, F. Gaucher, S. E. Mattsson, and F. Dupont. State machines in Modelica. In *Proc. of the 9th Int. Modelica Conf.*, number 078, pages 37–46, 2012.

[11] C. Gerking. Transparent UPPAAL-based verification of MechatronicUML models. Master’s thesis, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2013.

[12] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of 9th ESEC and 11th ACM SIGSOFT FSE*, 2003.

[13] C. Heinzemann, U. Pohlmann, J. Rieke, W. Schfer, O. Sudmann, and M. Tichy. Generating Simulink and Stateflow models from software specifications. In *Proc. of the Int. Design Conf.*, May 2012.

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

[15] H. Lundvall, P. Bunus, and P. Fritzson. Towards automatic generation of model checkable code from Modelica. In *Proc. of the 45th Conf. on Simulation and Modelling of the Scandinavian Simulation Society*, pages 23–24, 2004.

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

[17] Object Management Group. *Action Language for Foundational UML (ALF) 1.0*, Dec. 2011. Document formal/2010-10-05.

[18] M. Otter, M. Malmheden, H. Elmqvist, S. Mattsson, and C. Johnsson. A new formalism for modeling of reactive and hybrid systems. In *Proc. of the 7th Modelica Conf.*, 2009.

[19] U. Pohlmann. A UML based modeling language with operational semantics defined by Modelica. Master’s thesis, University of Paderborn, Department of Computer Science, Software Engineering Group, 2010.

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

[21] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner. Generating functional mockup units from software specifications. In *Proc. of the 9th Int. Modelica Conf.*, pages 765–774, 2012.

[22] W. Schamai, P. Helle, P. Fritzson, and C. J. Paredis. Virtual verification of system designs against system requirements. In *Models in Software Engineering*, pages 75–89. Springer, 2011.

[23] W. Schamai, U. Pohlmann, P. Fritzson, C. J. Paredis, P. Helle, and C. Strobel. Execution of UML state machines using Modelica. In *Proc. of EOOLT*, pages 1–10, 2010.