A Modelica Library for Real-Time Coordination Modeling

Uwe Pohlmann[1], Stefan Dziwok[1], Julian Suck[1], Boris Wolf[1], Chia Choon Loh[2], and Matthias Tichy[3]
[1]Software Engineering Group, [2]Control Engineering and Mechatronics Group,
Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany
[upohl | stefan.dziwok | jsuck | borisw | chia.choon.loh] @upb.de
[3]Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden
tichy@chalmers.se

Abstract

Increasingly, innovative functionality in embedded systems is realized by connecting previously autonomous embedded systems. This requires real-time communication and coordination between these connected systems. Modelica and the StateGraph2 library provide a good environment for modeling embedded systems including controllers and physics. However, it lacks appropriate support for modeling the communication and coordination part.

In this paper, we present an extension to the StateGraph2 library that enables modeling asynchronous and synchronous communication and rich real-time constraints. We illustrate our extension of the StateGraph2 library by modeling and simulating two miniature robots driving in a platoon.

Keywords: StateGraph2, Modelica Library, Coordination, Asynchronous Communication, Real-Time

1 Introduction

Embedded software is an important part of today's life. For example, there were about 30 embedded microprocessors per person in developed countries in 2008 and current cars include up to 70 electronic control units with about 1GB of software [4].

One reason for the increasing trend of embedded systems is the introduction of coordination between previously autonomous systems. As a result complex systems of systems arise to realize functionality which cannot be achieved by each system alone [12]. Again, the car industry is an example where vehicles communicate with other vehicles in order to extend the car's vision to areas obstructed by other vehicles [15]. This coordination requires an intensive communication between the systems under real-time constraints.

The embedded software is subject to very high quality requirements as often embedded systems are safety-critical systems where faults can result in severe consequences, e.g., injuries or loss of peoples' lives. Thus, faults of the system have to be avoided as much as possible. Currently, the rate of defects from mechanical parts decreases while the defect rate in electrical parts including software increases [4].

Therefore, appropriate validation and verification activities, e.g., simulation, have to be employed to detect and remove all faults. Model-driven development approaches allow to perform these activities already on the model level in early phases of development. Thus, on the one hand, a verification approach can exploit the abstraction provided by the model to improve the scalability and, on the other hand, verification can already be performed early in the process where no implementation yet exists.

Modelica is an object-oriented, declarative, multi-domain modeling language for describing and simulating models which represent physical behavior, the exchange of energy, signals, or other continuous-time interactions between system components as well as reactive, discrete-time behavior. Modelica uses the hybrid differential algebraic equation formalism as a sound mathematical representation. Furthermore, mature compilation and simulation environments for Modelica exist.

However, Modelica in version 3.2 and particularly the StateGraph2 library lack appropriate support for the sketched case of modeling the real-time coordination between autonomous systems as this coordination is often realized by communication using asynchronous messages and complex state-based behavior [12].

In this paper, we present a Modelica library for modeling communication under hard real-time constraints. Our library extends the StateGraph2 library by providing support for (1) synchronous and asynchronous communication and (2) rich modeling of real-time behavior.

These extensions are based on our previous work on the MECHATRONICUML modeling language [2] and ModelicaML [11].

In the next section, we present our running example. We discuss the limits of the StateGraph2 library with respect to this scenario in Section 3. Our extension to the StateGraph2 library is described in Section 4. We formally define our extension in Section 5. In Section 6, we present the Modelica model of our scenario using our library extensions. After a discussion of related work in Section 7, we conclude and give an outlook on future work in Section 8.

2 Running Example

This section presents our test platform for evaluating real-time coordination scenarios. We present a concrete real-time coordination scenario of a platoon drive as the running example for the paper.

2.1 Intelligent Miniature Robot BeBot

The test platform is a wheeled mobile robot known as BeBot [7]. It is a miniature mobile robot developed at Heinz Nixdorf Institute and has been used in various research projects, e.g., [8]. The BeBot is powered by two DC-motors with integrated encoder.

To use this mobile robot in a simulation environment, a model of the BeBot is developed in Dymola. Basically, the hardware model of the mobile robot can be categorized into three main groups. The first group consists of its casing and electrical circuit boards. All these components are modeled as a rigid body in Dymola. In addition, the shape model from the *MultiBody* library is used to visualize these components in the animation. The second group comprises the wheels of the robot. Under the assumption of pure rolling, these wheels are represented by a pair of wheels with a common axle whereby each wheel is individually controlled. The third group is made of two DC-motors. Each of these

Figure 1: Intelligent Miniature Robot BeBot

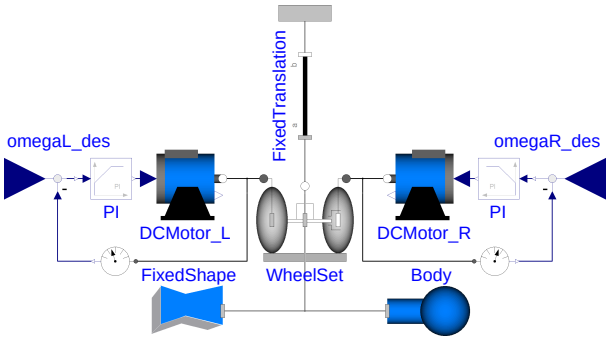

Figure 2: Model of BeBot Mobile Robot in Dymola

motors is represented using a model of a DC-motor. In this model, friction is taken into consideration to provide realistic behavior for the motor. As shown in Figure 2, these components are connected accordingly to create a simple model of the BeBot.

To control the movement of the mobile robot, the velocities of the wheels have to be controlled. Therefore, a speed controller is designed to control the rotation velocity of each wheel. The controller is a PI-controller with anti-wind-up function and it ensures that each wheel rotates at a desired velocity.

2.2 BeBot Platoon Scenario

The scenario consists of two BeBots (see Figure 3). They communicate wirelessly with each other and have a distance sensor at their front. Both have the same software specifications. The BeBots drive on a straight way in the same direction. The front-driving BeBot transports a heavy good to the furthermost place of delivery. The rear-driving BeBot transports several small goods and has to deliver them to several stations. As the front-driving BeBot is heavier than the rear-driving BeBot, its cruising speed is slower than the cruising speed of the rear-driving BeBot. To optimize the energy consumption, BeBots may form a platoon, i.e., the rear-driving BeBot drives in the slipstream of the front-driving BeBot.

During platooning, a collision could occur if the front-driving BeBot must brake very hard (e.g., due to an obstacle on the street) and the rear-driving Be-

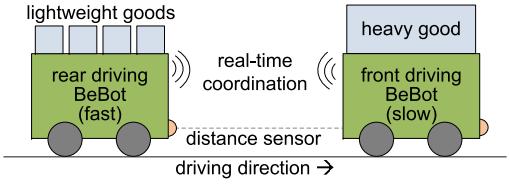

Figure 3: Platoon Scenario with Two BeBots

Bot does not know beforehand that it must brake. To avoid a collision, the front-driving BeBot commands the rear-driving BeBot by sending an asynchronous brake-message to perform a brake maneuver. The brake-message is transmitted to the rear-driving BeBot that is going to brake as soon as it gets this message. This delivery time is safety-critical, because the front-driving BeBot brakes after that time and braking must not result in a collision. A precondition to coordinate such braking behavior is that a BeBot must know if another BeBot is driving behind. Therefore, besides the braking message also messages for starting and ending a platoon are required.

The behavior specification of this scenario can be modeled with statecharts, e.g., to distinguish if a BeBot drives in a platoon or not. By using Dymola, the StateGraph2 library is the first choice. However, the next section shows the limits of StateGraph2 for modeling the behavior of this real-time coordination scenario.

3 Limits of StateGraph2

StateGraph2 [9] is a Modelica library for state-based modeling. It provides the three main classes *Step*, *Transition*, and *Parallel* for modeling statecharts. The class *Step* models discrete system states, the class *Transition* models state changes, and the class *Parallel* models hierarchical and parallel states.

Statecharts are used to describe the behavior of reactive systems. The reactions of such systems are based on their current internal state and the external input. Formalisms for Mealy machines, Harel's statecharts [5], and most common automata-based formalisms support events that can be used for a message-based communication. However, StateGraph2 does not have syntactical constructs. Different steps or transitions can only communicate via shared variables. In real systems, this is not possible when the systems are distributed and have no access to shared memory. The need of shared memory makes it difficult to reuse components as they depend on their environment and not only on their interface description. Therefore, a message-based mechanism is very important. This may be either an asynchronous or a synchronous communication.

StateGraph2 has only a limited support to specify timing behavior. Only the execution of transitions can be delayed. The variable *waitTime* of a *Transition* specifies the time a transition waits before it fires when its guard evaluates to true. If during the waiting period the guard evaluates back to false, the transition does not fire. Therefore, the construct *delayedTransition* of StateGraph2 can be misinterpreted, because the semantics includes more than a simple delay. In contrast to StateGraph2, Timed automata [1] use clocks to store time independently of a concrete state. Clocks can be read and reset in any state and upon firing of a transition. Therefore, this concept is more flexible for specifying timing behavior. To conclude, the variable *waitTime* alone is too limited to describe real-time behavior.

A modeling language for the software of mechatronic systems that supports hierarchical statecharts as well as synchronous and asynchronous communication, and clocks is MECHATRONICUML [2]. The formal behavior definition of this language is based on timed automata [1]. Therefore, our extensions of the StateGraph2 library are based on concepts of MECHATRONICUML. The next section explains these extensions.

4 Real-Time Coordination Library

As stated above, adequate modeling constructs for synchronous as well as asynchronous communication and for real-time behavior are essential for modern embedded systems. Here, we consider synchronous and asynchronous communication to be a message-based communication where the former means that the sender always waits as long as the receiver is not able to consume the message. The latter means that the sender does not wait on a reaction of the receiver and proceeds with its execution that, in particular, might include sending further messages. For asynchronous communication, this implies that the receiver has to have a message buffer which is sufficiently large to prevent loss of messages.

This section introduces our extended version of the StateGraph2 library, called *real-time coordination library*. In particular, Section 4.1 introduces *synchronization ports* and *synchronization connectors* for synchronous communication. Section 4.2 shows *Messages* and *Mailboxes* for asynchronous communication. Finally, Section 4.3 describes *Clocks*, *Invariants* and *Clock Constraints* for the modeling of real-time behavior according to time automata [1].

4.1 Synchronization Connectors and Ports

For the modeling of synchronous communication, we extended transitions by *synchronization ports (sync ports)*. Sync ports sub-divide into *sender sync ports* and *receiver sync ports*. A sender sync port of one transition is connected to a receiver sync port of another transition by a synchronization connector. We repre-

sent a sender sync port as a non-filled orange circle, a receiver sync port as a filled orange circle and a synchronization connector as an orange line. In Figure 4, a synchronization connector connects the sender sync port of transition t1 with the receiver sync port t2.

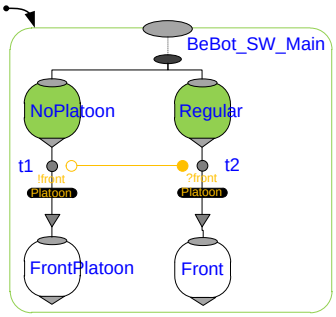

Figure 4: Synchronization Ports and Connectors

A transition that is connected via its sender or receiver sync ports to the receiver or sender sync ports of other transitions is allowed to fire if it is able to fire together with at least one of the connected transitions. For the example in Figure 4, this means that t1 is allowed to fire if t2 is able to fire and vice versa.

We now give a detailed explanation of how the firing of transitions with synchronization is implemented. The implementation is presented with help of the dependency graph in Figure 5.

First, the necessary conditions for firing each of the transitions (without synchronization) have to be satisfied, i.e., the preceding generalized step has to be active, the *condition* of the transition must hold and the optional *condition port* of the transition must be set. If all of these conditions hold, the property preFire of each of the transitions will evaluate to true.

Furthermore, if an *after time* is specified for the transition it must have expired. The after time construct is new and replaces the delay (wait) time from the original version of the StateGraph2 library. It differs from the delay time in that *at least* the after time must have expired to let the transition fire. In contrast, the semantics of the delay time is that the delay time must have

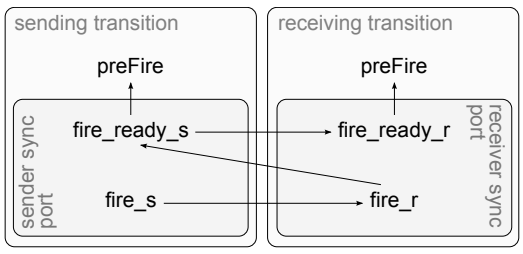

Figure 5: Dependency Graph of Conditions for Firing of Transitions with Synchronization

expired after the transition is fireable in order to let the transition fire. We introduced the after time semantics because it might happen that for two transitions that need to synchronize the time instants in which they are allowed to fire might not match due to their delay time.

If preFire of the sending transition, i.e., the transition whose receiver sync port is connected to the synchronization connector, is true, the signal fire_ready_r of the receiver sync port is set to true. If for the sending transition, i.e., the transition whose sending sync port is connected to the synchronization connector, holds that preFire is true and it receives the signal fire_ready_r over its sender sync port then the signal fire_ready_s of its sender sync port is set to true. If the signal fire_ready_s is true in the receiving transition the signal fire_r of the receiver sync port is set to true. Finally, if fire_r is recognized to be true in the sending transition the signal fire_s of its sender sync port is set to true and both transitions are ready to fire.

4.2 Messages and Mailboxes

For the modeling of asynchronous communication, we introduce two new components named *Message* and *Mailbox*. Each instance of the Message component has two purposes. On the one hand, it defines a certain *message type* by specifying an array of formal parameters which might be of type Integer, Boolean or Real. As an example one message type might be defined by the array $(Integer[2], Boolean[1], Real[1])$. The parameter array of a message type is also called its *signature*. On the other hand, an instance of the Message component is responsible for sending a message whenever a connected transition fires. A transition is able to signal to a Message component instance to send a message if the firePort of the transition is connected to the conditionPort of the Message component instance.

As a visualization example consider the message type confirm in Figure 6. The purple connector connects the firePort of the transition t1, displayed as a non-filled purple triangle, to the conditionPort of confirm where the conditionPort of confirm is represented by a purple triangle. Additionally, confirm has exactly one Integer parameter that is determined by the yellow connector that originates at the port cruisingSpeed and connects to the Integer valued input port of confirm represented by a yellow filled circle.

For each message type exists exactly one instance of the Mailbox component with the same signature. The message type sends its messages to the Mailbox instance. To specify which message type belongs to which Mailbox instance the message_output_port of the

message type is connected to the mailbox_input_port of the Mailbox instance.

A Mailbox instance defines a finite FIFO queue where the size of the queue is settable at design time. In order to let a transition receive a certain message from such a queue its transition_input_port is connected to the mailbox_output_port of the Mailbox instance. Then, the transition is allowed to fire if the Mailbox instance signals that at least one message is present. As an example for the visual representation consider the Mailbox instance confirmBox in Figure 6 that is connected to the transition t2 by a connector.

If two extended StateGraph2 models are included in different component instances they might still communicate asynchronously across the boundaries of these component instances with the help of *delegation ports*. Therefore, one component defines an output delegation port and the other defines an input delegation port. Both delegation ports are connected. Then, the component instance containing the message type connects the message type to the output delegation ports and the component instance containing the Mailbox instance connects the Mailbox instance to the input delegation port. As an example consider Figure 6 which shows two extended State Graph models in two separate component instances communicating over delegation ports that are displayed as envelopes with gray triangle.

Synchronous and asynchronous communication can be combined at one transition. Besides the synchronization conditions the Mailbox instance additionally has to signal to the transition that at least one message is available.

4.3 Clocks, Invariants and Clock Constraints

For the modeling of real-time behavior according to timed automata, we extended the StateGraph2 library by three components named *Clock*, *Invariant* and *ClockConstraint*. Clocks are real-valued variables whose values increase continuously and synchronously with time. Clocks might be reset to zero upon activation of a generalized step or firing of a transition. An invariant is an inequation that specifies an upper bound on a clock, e.g., c < 2 or c <= 2 where c is a clock. Invariants are assigned to generalized steps and are used to specify a time span in which this generalized step is allowed to be active. A clock constraint might be any kind of inequation specifying a bound on a certain clock, e.g., c > 2, c >= 5, c < 2, c <= 5 where c is a clock. Clock constraints are assigned to transitions in order to restrict the time span in which a transition is allowed to fire.

As an example consider Figure 7. The example consists of a clock c, an invariant clockValue <= bound and a clock constraint clockValue >= bound where bound is a positive integral number given as a parameter. Clocks are displayed as a rectangle containing a clock icon, invariants are displayed as rectangles containing the corresponding inequation and a transition icon. Clock constraints are displayed as rectangle containing the corresponding inequation and a step icon. The clock which is used by an invariant or a clock constraint is connected via its y port with the clockValue port of the invariants and clock constraints.

When the generalized step PlatoonProposed is activated, the clock c is reset to zero, which is accomplished by connecting the activePort (non-filled purple triangle) of PlatoonProposed to the u port (non-filled purple circle) of the clock. The invariant is assigned to the step PlatoonProposed by the connector originating at the activePort of PlatoonProposal leading to the conditionPort (filled purple circle) of the invariant. It means that PlatoonProposed is allowed to be active if c has a value less or equal to bound. The transition t1 is assigned the clock constraint by connecting the firePort of the clock constraint with the conditionPort of t1. The clock

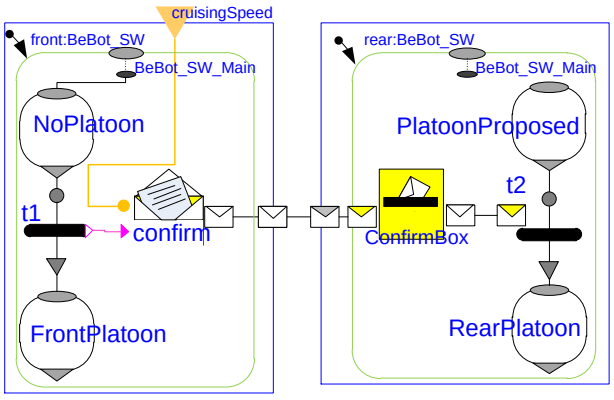

Figure 6: Message Types and Mailbox Instances

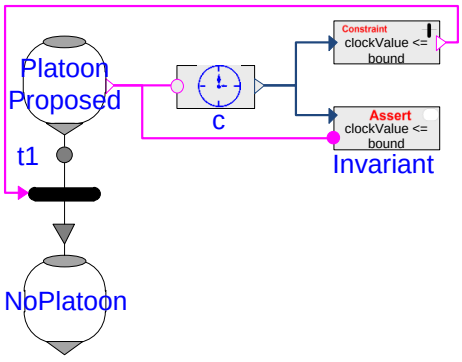

Figure 7: Clocks, Invariants and Clock Constraints

constraint means that t1 is allowed to fire if c has a value greater or equal to time bound.

5 Formal Definition of the Library

This section covers the formal definition of an extended StateGraph2 model. The Real-Time Coordination library extends the structure of the model given in [9] by synchronization connectors, mailboxes, clocks, invariants and clock constraints whereas the former two are required for synchronous and asynchronous communication resp. and the latter three are used for the specification of real-time behavior analogously to time automata [1]. Due to the possibility of synchronization of two transitions, we altered the delay time of a transition to an after time, which has slightly different semantics.

For the definition of the semantics we give an interpretation algorithm that is analogous to the one given in [9]. Additionally, consider the added elements, i.e., when a generalized step is active the corresponding invariant must not be violated. Further, when a transition fires its clock constraint must be satisfied, it must be able to synchronize, and to receive the required messages.

We present the structure in Section 5.1 and introduce an interpretation algorithm that defines the semantics in Section 5.2.

5.1 Structure

The extension is represented by the following tuple

$$Ext := (Sync, MBox, C, INV, CC)$$

where *Sync* denotes the set of synchronization connectors required for synchronous communication. Let *Msg* be the set of messages used for asynchronous communication. Then $MBox : Msg \to \mathbb{N}$ determines for each message how often it is available in its corresponding mailbox. The real-time extension is covered by the set C of clocks, the set *Inv* of invariants and the set *CC* of clock-constraints.

As said before, the set of messages results from all possible combinations of message parameters. We abstracted from message parameters here, simply saying that there exists a set of distinct messages. Furthermore, in the implementation of our extension there exists the *MailBox* component for the realization of asynchronous communication. Since the number of messages included in a certain mailbox suffices to be able to determine whether a transition that requires such a message is able to fire, we abstracted from the mailboxes here in form of the *MBox* function. The following definition consists of elements that where already defined in [9]. For the sake of completeness, we describe and list them.

With the help of our extension *Ext*, we define an *extended StateGraph2 model (ESGM)* Γ as follows:

$$\Gamma := (V_c, G, T, G_I, G_E, Ext)$$

where

- V_c is a set of Boolean expression as defined in [9].
- G is a set of generalized steps $G = \{g_1, g_2, \ldots\}$
  A generalized step g_i is defined as a 7-Tuple
  $$g_i = (\Gamma_s, I, O, S, R, Inv_{g_i}, RESET_{g_i})$$
  where
  - Γ_s is a possibly empty set of sub-graphs $\Gamma_s = \{\gamma_1, \gamma_2, \ldots\}$. A sub-graph $\gamma_i \in \Gamma_s$ is again an ESGM. Note that this recursive definition allows an arbitrary deep nesting of ESGMs.
  - I is a vector of in (entry) ports $I = [i_1, i_2, \ldots]$. An in port is a connection point incoming transitions of g_i are connected to.
  - O is a vector of out (exit) ports $O = [o_1, o_2, \ldots]$. An out port is a connection point outgoing transitions of g_i are connected to.
  - S is a possibly empty vector of suspend ports $S = [s_1, s_2, \ldots]$. A suspend port is a connection point outgoing transitions of g_i are connected to. The difference to out ports is that the active generalized steps of sub-graphs of g_i are stored for later restore.
  - R is a possibly empty vector of resume ports $R = [r_1, r_2, \ldots]$. A resume port is a connection point ingoing transitions of g_i are connected to. The difference to in ports is that the active generalized steps of sub-graphs of g_i that were active when g_i was left by a suspend port are restored.
  - $Inv_{g_i} \subseteq Inv$ is a set of invariants. An invariant describes that a clock must never exceed a certain bound when the generalized step is active. It is denoted as an inequation of the form $c \leq n$, where $c \in C$ is a clock and $n \in \mathbb{N}$ is a natural number (including zero).
  - $RESET_{g_i} \in C$ is a set of clocks that are to be reset to zero when the generalized step is activated.

  A generalized step that has in and out ports but no other ports and no sub-graphs, i.e., $I \neq \emptyset$, $O \neq \emptyset$ and $R = S = \Gamma_s = \emptyset$ is called *step*. A generalized step that has resume ports, suspend ports or sub-graphs, i.e., $R \neq \emptyset$, $S \neq \emptyset$ or $\Gamma_s \neq \emptyset$ holds, is called *parallel step*.
- T is a set of transitions $T = \{t_1, t_2, \ldots\}$. A transition

$t_i \in T$ is defined by the 10-tuple

$$t_i = (p^{IR}_{t_i}, p^{OS}_{t_i}, C_{t_i}, A_{t_i}, CC_{t_i}, R_{t_i}, S^R_{t_i}, S^S_{t_i}, M^R_{t_i}, M^S_{t_i})$$

where

- $p^{IR}_{t_i}$ is a connected port of an in or resume vector of a succeeding generalized step $g_i \in G$.
- $p^{OS}_{t_i}$ is a connected port of an out or suspend vector of a preceding generalized step $g_i \in G$.
- $C_{t_i} \in V_c$ is the fire condition associated with t_i.
- $A_{t_i} \in \mathbb{R}$ is the after time associated with t_i. Note, that we consciously chose the name after time instead of delay time as in the original definition in [9] since the semantics of the after time will be different from the one of the delay time.
- $CC_{t_i} \in CC$ are the clock constraints associated with t_i.
- $R_{t_i} \in C$ are the clocks to be reset when t_i fires.
- $M^R_{t_i} \subseteq Msg$ is the message that must be received when t_i fires.
- $M^S_{t_i} \subseteq Msg$ is the message that is sent when t_i fires.
- $S^R_{t_i} \subseteq Sync$ is the synchronization connector that has to be set by another transition when t_i fires.
- $S^S_{t_i} \subseteq Sync$ is the synchronization connector that is set if t_i is firable.

We further define that a transition might have at most one message that is to be received and at most one message that is to be sent, i.e., $|M^R_{t_i}| \leq 1$ and $M^S_{t_i} \leq 1$ resp., and at most one synchronization connector over which a signal is sent or received, i.e., $|S^R_{t_i}| + |S^S_{t_i}| \leq 1$.

- $G_I \subseteq G$ contains the initial generalized step of Γ.
- $G_E \subseteq G$ contains the exit generalized step of Γ.

As a well-formedness constraint, we assume that every ESGM has exactly one initial state and at most one exit state, i.e., $|G_I| = 1$ and $|G_E| \leq 1$. Furthermore, we assume that the *uppermost ESGM* $\Gamma = (V_c, G, T, G_I, G_E, Ext)$, i.e., that ESGM that is not embedded by any other ESGM, does not have an exit generalized step, i.e., $G_E = \emptyset$.

5.2 Interpretation Algorithm

1. Activate the initial generalized step $g \in G_I$. If g has sub-graphs, then recursively activate the initial generalized steps of all of its embedded sub-graphs.
2. Determine the set $T_{fireable}$ of all transitions t_i that satisfy:
   - its condition C_{t_i} is true,
   - the required after time A_{t_i} has passed,
   - its in or resume port $p^{IR}_{t_i}$ is set to true,
   - if its preceding generalized step has sub-graphs, the exit generalized steps of all of these sub-graphs are recursively activated
   - if $M^R_{t_i} \neq \emptyset$ and $m \in M^R_{t_i}$ is the message to be received by t_i, the Mailbox of m contains at least one message, i.e., $MBox(m) > 0$.
   - there exists no other transition $t_j \in T_{fireable}$ that has the same preceding generalized basic step and has higher priority than t_i where the priority results from the index of the transition in the port vector (see [9]).
3. For all $t_i \in T_{fireable}$ do:
   i. if $S^S_{t_i} \neq \emptyset$ and $s \in S^S_{t_i}$ is the synchronization connector of t_i for sending a signal, set s to true
4. Determine the set $T_{syncable}$ of all transitions $t_i \in T_{firable}$ that satisfy:
   - either $S^R_{t_i} = \emptyset$ or
   - if $S^R_{t_i} \neq \emptyset$ and $s \in S^R_{t_i}$ is the synchronization connector of t_i, t_i is set to true
5. For all $t_i \in T_{syncable}$ fire t_i as follows:
   i. Deactivate the preceding generalized step g of t_i. If g_i includes sub-graphs deactivate these sub-graphs recursively.
   ii. Activate the succeding generalized step g' of t_i. If g' includes sub-graphs activate these sub-graphs recursively as follows:
      - if t_i is connected to g' by a resume port, the generalized steps of g' and of all sub-graphs of g' that where active the last time g' was active are recursively activated
      - else, activate all initial generalized steps of g' and its sub-graphs recursively.
   iii. if $M^R_{t_i} \neq \emptyset$ and $m \in M^S_{t_i}$ is the message to be received by t_i, then take one message out of the the Mailbox of m, i.e., $MBox := (MBox \setminus \{(m,d)\}) \cup \{(m,d-1)\}$ where $d \in \mathbb{N}$ is the amount of messages in the mailbox before t_i fires.
   iv. if $M^S_{t_i} \neq \emptyset$ and $m \in M^S_{t_i}$ is the message to be sent by t_i, then put one message into the Mailbox of m, i.e., $MBox := (MBox \setminus \{(m,d)\}) \cup \{(m,d+1)\}$ where $d \in \mathbb{N}$ is the amount of messages in the mailbox before t_i fires.
6. Goto 2.

6 Case Study

This section shows how we modeled the platoon scenario. First, we used the StateGraph2 library in combination with our Real-Time Coordination library to specify the discrete software. Then, we connected the software model with the simulation model of the BeBot hardware that we have presented in Section 2.1. This section shows an excerpt of our model. The complete

model is delivered within our Real-Time Coordination library.

Figure 8 shows the discrete behavior specification that we modeled as class BeBot_SW in Dymola. We used the *Step* components and the *Parallel* component from the StateGraph2 library. From the Real-Time Coordination library, we used the *Transition* components, the *Message* components, the *Mailbox* components, and the *DelegationPort* components. We omit guards, connection lines between synchronizations, and timing constraints.

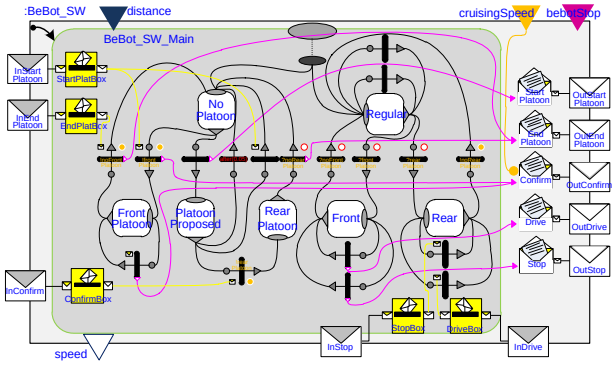

Figure 8: Platoon Scenario Behavior Modeled

The interface of the class BeBot_SW defines three incoming parameters: the distance to a BeBot, that drives in front, the cruisingSpeed of the BeBot, and bebotStop that defines if the BeBot has to stop. The outgoing parameter is the speed of the BeBot. Furthermore, five asynchronous messages are defined that can be sent and received: StartPlatoon to propose to start a platoon, Confirm to confirm the start proposal, EndPlatoon to command the end of the platoon, Stop to command a rear-driving BeBot to stop, and Drive to inform a rear-driving BeBot that it no longer has to stop.

Within BeBot_SW, two parallel branches were defined. The first branch handles the platoon activation and deactivation and consists of the steps NoPlatoon, PlatoonProposed, and FrontPlatoon. The second branch handles the coordinated braking within a platoon and consists of the steps Regular (a BeBot has no limitations regarding braking), Front (a BeBot has first to inform the rear-driving BeBot before braking), and Rear (a BeBot must brake when the front-driving BeBot commands it). The synchronization between the two branches is realized by using synchronous communication, e.g., if step FrontPlatoon is activated, then step Front will also be activated at the same time. Among others, this class contains a timing constraint that the state PlatoonProposed is no longer active than 50ms.

Figure 9 shows the two connected instances front and rear of the class BeBot_SW. Furthermore, it shows two instances of the BeBot hardware model (see Figure 2) and how they are connected with the software models. The instance distance of the class Distance calculates the distance of the rear BeBot to the front BeBot. We do not display the connections to the inputs cruisingSpeed and stop.

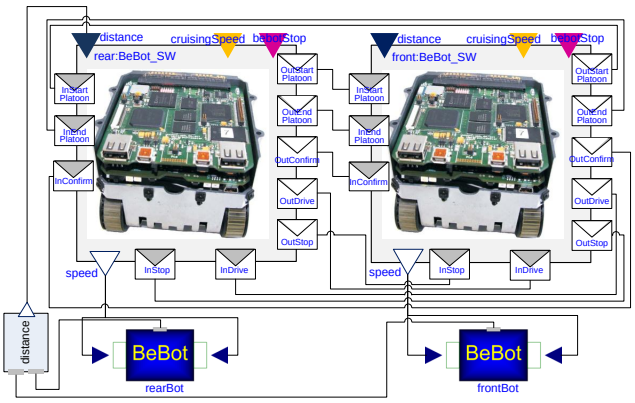

Figure 9: Platoon Scenario Instance Model

Figures 10 and 11 show the results of a simulation run of the model. Figure 10 shows the asynchronous messages that were sent between the rear- and the front-driving BeBot. Figure 11 shows the speed result of both BeBots during a performed simulation. Right at the start, the rear-driving BeBot speed was higher. As the distance had reached a size where a platoon was needed, the rear-driving BeBot sent the message StartPlatoon. At time 8.6, the rear-driving BeBot received the message Confirm(1) so it had adjusted its speed to 1. At time 25, the stop input of the rear-driving BeBot raised to 1. Therefore, the rear-driving BeBot ended the platoon by sending the message EndPlatoon and stopped for 10s. Then the rear-driving BeBot started again to close the gap by driving faster and to start a new platoon.

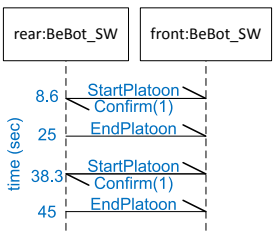

Figure 10: Sequence Diagram

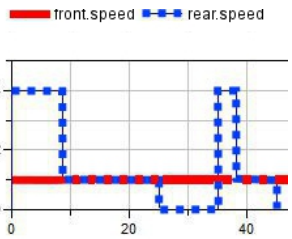

Figure 11: Simulation Plot

Figure 12 shows the 3D view of the simulation run. The left shows the moment when the rear BeBot drives faster than the front BeBot and the right shows when both BeBots drive in the platoon with the same speed.

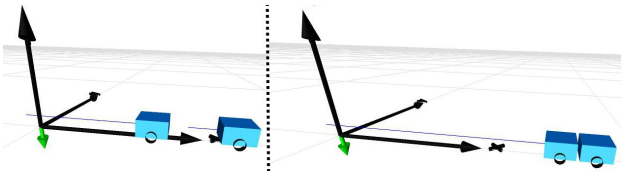

Figure 12: 3D View of Simulation

7 Related Work

This section presents some other approaches for modeling discrete state-based behavior for simulating hybrid cyber-physical systems. We focus on the capabilities to model and simulate real-time properties and constraints of the behavior, synchronize parallel behavior, and to communicate via asynchronous messages.

7.1 SimulationX

SimulationX supports an own representation of state machines which follows the model of UML state machines but only implements a limited subset [3]. SimulationX state machines have no support for parallel behavior and therefore no support of synchronizations. The asynchronous signals have no support for an arbitrary number of parameters and are lost when the receiver is not enabled to consume them immediately. They have no concept of a mailbox for storing messages. SimulationX supports only limited timing support. Its time events only react to an expression which is relative to the active state time of the transition which is triggered by the *after* event. It is not possible to model time invariants as first class entities. As SimulationX supports Modelica, it is possible to port the concepts that we present in this paper to SimulationX.

7.2 ModelicaML

ModelicaML is a UML Profile [13] which extends UML *Classes* and *Properties* with *Stereotypes* for Modelica. Therefore, it is possible to model with UML Classes as in Modelica. Further, ModelicaML defines a mapping of UML state machines and simple internal events to plain Modelica algorithmic code [14]. For more complex messages it is possible to use external C-functions [11]. A code generation algorithm does the mapping of UML state machines to Modelica code automatically. In contrast to the State Graph2 extension presented in this paper it is hard to edit the state machine behavior directly in Modelica because it is encoded in a complex algorithm. Further, ModelicaML has no support for synchronization of parallel behavior from different regions as presented in Section 4.1. ModelicaML supports only rudimentary timing behavior as first class entity with its AFTER-macro [14]. This construct is a transition guard relative to the active time of a state. ModelicaML also does not support time invariants of states. As ModelicaML supports Modelica, it is possible to port the concepts that we present in this paper to ModelicaML.

7.3 MATLAB/Simulink, Stateflow

MATLAB provides the custom modeling language Stateflow for state based behavior. Stateflow has interfaces to the Simulink environment. Stateflow has some drawbacks for modeling communication protocols with real-time requirements between distributed systems. For clocks, helping elements from Simulink to count time-ticks are needed. Stateflow also has no concept of asynchronous, message-based communication with mailboxes for sent and received messages. Stateflow events are not buffered by the receiver and could be lost if the receiver is busy. It is possible to encode asynchronous message-based communication. Therefore, you need a complex combination of several linked Simulink and Stateflow blocks, which is hard to maintain manually [6, 10].

8 Conclusions and Future Work

Today, autonomous embedded systems are increasingly connected to each other to realize new innovative functionality, e.g., in the case of vehicle-to-vehicle communication to realize platooning.

We presented an extension of the StateGraph2 library that enables modeling a real-time communication and coordination between autonomous embedded systems by providing library elements for asynchronous and synchronous communication as well as real-time constraints. We modeled two miniature robots that drive in a platoon with our library to simulate it.

We plan to make several additions to our library. Asynchronous message exchange between autonomous systems may suffer from message loss or message delays. Therefore, we plan to enable modeling different probabilistic quality of service characteristics, e.g., message delays and message losses. The new Modelica version 3.3 have built-in support of finite state machines, which makes the StateGraph2 library obsolete. However, the new built-in finite state machines does not support asynchronous message-based communication, so we suggest to use our extensions for asynchronous

message-based communication. The integration is up to further research.

With respect to tool chains, we want to implement automatic transformations from MECHATRONICUML to the presented extended StateGraph2 library. This allows us to reap the benefits from formal verification by model checking, which is possible for models of the MECHATRONICUML [2], and integrated simulation including feedback controllers and physics by using Modelica. Finally, we will use our library in several other case studies, including a de-centralized industrial dough mixing system.

Acknowledgments

This work was developed in the project 'ENTIME: Entwurfstechnik Intelligente Mechatronik' (Design Methods for Intelligent Mechatronic Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, 'Investing in your future'. Chia Choon Loh is supported by the International Graduate School Dynamic Intelligent Systems.

References

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

[2] S. Becker, C. Brenner, S. Dziwok, T. Gewering, C. Heinzemann, U. Pohlmann, C. Priesterjahn, W. Schäfer, J. Suck, O. Sudmann, and M. Tichy. The mechatronicuml method - process, syntax, and semantics. Technical Report tr-ri-12-318, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2012.

[3] 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 *Proceedings of the 6th Modelica Conference, Bielefeld*, pages 269–276, 2008.

[4] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *IEEE Computer*, 42(4):42–52, 2009.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[6] C. Heinzemann, U. Pohlmann, J. Rieke, W. Schäfer, O. Sudmann, and M. Tichy. Generating simulink and stateflow models from software specifications. In *Proceedings of the International Design Conference, DESIGN 2012, Dubrovnik, Croatia*, May 2012.

[7] S. Herbrechtsmeier, U. Witkowski, and U. Rückert. Bebot: A modular mobile miniature robot platform supporting hardware reconfiguration and multi-standard communication. In *Progress in Robotics, Communications in Computer and Information Science. Proceedings of the FIRA RoboWorld Congress 2009*, volume 44, pages 346–356, Incheon, Korea, 2009. Springer.

[8] C. C. Loh and A. Trächtler. Laser-sintered platform with optical sensor for a mobile robot used in cooperative load transport. In *Proceedings of the 37th Annual Conference on IEEE Industrial Electronics Society*, pages 888–893, November 2011.

[9] M. Otter, M. Malmheden, H. Elmqvist, S.E. Mattsson, C. Johnsson, D. Systèmes, and S.D. Lund. A new formalism for modeling of reactive and hybrid systems. In *Proceedings of the 7th Modelica'2009 Conference, Como, Italy*, 2009.

[10] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. From verification to implementation: A model translation tool and a pacemaker case study. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012), Beijing, China*, April 2012.

[11] U. Pohlmann and M. Tichy. Modelica code generation from ModelicaML state machines extended by asynchronous communication. In *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2011, Zurich, Switzerland*, 2011.

[12] W. Schäfer and H. Wehrheim. The Challenges of Building Advanced Mechatronic Systems. In Lionel C. Briand and Alexander L. Wolf, editors, *FOSE*, pages 72–84, 2007.

[13] W. Schamai. Modelica modeling language (ModelicaML) : A UML profile for Modelica. Technical report, Linköping University, Department of Computer and Information Science, The Institute of Technology, 2009.

[14] W. Schamai, U. Pohlmann, P. Fritzson, C. J.J. Paredis, P. Helle, and C. Strobel. Execution of uml state machines using modelica. In *Proceedings of EOOLT*, pages 1–10, 2010.

[15] C. Weiß. V2X communication in Europe - From research projects towards standardization and field testing of vehicle communication technology. *Computer Networks*, 55(14):3103–3119, 2011.