INTERNATIONAL DESIGN CONFERENCE - DESIGN 2012
Dubrovnik - Croatia, May 21 - 24, 2012.

DESIGN 2012

GENERATING SIMULINK AND STATEFLOW MODELS FROM SOFTWARE SPECIFICATIONS

C. Heinzemann, U. Pohlmann, J. Rieke, W. Schäfer, O. Sudmann and M. Tichy

Keywords: mechatronic systems engineering, software engineering, mechatronicUML

1. Introduction

Innovation in today's technical systems is largely driven by embedded software. Such systems are known as mechatronic systems. For example, it has been estimated that the current generation of upper class cars will contain about one gigabyte of software [Pretschner et al. 2007]. Mechatronic systems pose a challenge for software development as they are often employed in a safety-critical context and they operate under tight resource constraints.

While previously most of the embedded software consisted of single feedback controllers for controlling the dynamic behavior of the physical part of the system, these single controllers are increasingly connected to each other. Then, the behavior of a single controller is depending on the behavior of another controller. This requires discrete state-based software for the specification of asynchronous message exchange in addition to the control software. Consequently, this leads to complex hybrid embedded software. For example in a modern car, the adaptive cruise control system connects the previously isolated feedback controllers for the engine and the braking subsystems.

Furthermore, single technical systems are not working in isolation anymore but rather form systems of systems where autonomous systems coordinate and communicate in an ad-hoc fashion using complex message-based communication protocols [Schäfer and Wehrheim 2007]. In this case, the network topology is not fixed at design time but rather adapts itself at run time. For example, several cars may coordinate their headlights to improve the illumination of the street to reduce the number of accidents.

Owing to these trends, the amount of software and its complexity rises to unprecedented heights. Thus, the key issue for the successful development of such systems is handling the size and the complexity by appropriate development methods and languages as well as supporting tools.

Model-driven development approaches enable to abstract from technical implementation details and, thus, are better suited to develop such systems. MATLAB[1] with its toolboxes Simulink and Stateflow, Dymola[2] based on the open source language Modelica as well as ASCET[3] and SCADE[4] are state of the art software tools for the model-driven development for software in embedded software. They all provide means to model feedback controllers using block diagrams and discrete state-based behavior using a variant of statecharts [Harel 1987].

All these software tools, in principle, support the development of software for the aforementioned use cases. However, they lack appropriate modeling support in the case of communication protocols using

[1] http://www.mathworks.com/products/matlab/index.html
[2] http://www.3ds.com/products/catia/portfolio/dymola
[3] http://www.etas.com/en/products/ascet_software_products.php
[4] http://www.esterel-technologies.com/products/scade-suite/

asynchronous message exchange and complex real-time constraints encompassing several states and transitions [Giese and Henkler 2006].
The engineer may use workarounds, e.g., by a manual implementation using Embedded MATLAB Functions, to emulate the required functionality. This works for simulation and target code generation purposes. However, single simulation runs cannot guarantee the freedom of bugs. In contrast, formal verification [Baier and Katoen 2008], as, e.g., in the Simulink Design Verifier and the SCADE Suite Design Verifier, enables to guarantee freedom of bugs by considering all possible simulation runs.
Though, the aforementioned workarounds make formal verification quickly infeasible as the number of simulation runs explodes because the semantics of asynchronous messages and rich time constraints is hidden in the manual implementation of the workarounds. Consequently, modeling languages are required which support the specification of asynchronous message exchange and rich time constraints as first class modeling entities.
MechatronicUML [Becker et al. 2012] is a modeling language which targets the software embedded in mechatronic systems and specifically addresses the aforementioned case of systems of systems with complex communication protocols. MechatronicUML focuses on the discrete part of the system. It follows the component-based approach for software development. The behavior of discrete components is specified using Real-Time Statecharts, which are a combination of UML state machines [Object Management Group 2011] and timed automata [Alur and Dill 1994]. They support the specification of asynchronous messages and time constraints as first class entities in the modeling language.
The formal verification of MechatronicUML models exploits a sophisticated interface definition between the discrete part, which contains the aforementioned complex message-based communication protocols, and the continuous part, which contains the control software. The interface decouples the discrete part from the continuous part and, thus, enables the efficient automatic formal verification of the discrete part. This formal verification exploits assumptions about the continuous behavior that are guaranteed by extensive manual simulations. After the successful formal verification of the system, the discrete part has to be integrated with the continuous part for holistic simulations and target code generation.
In this paper, we present how we employ MATLAB with the toolboxes Simulink and Stateflow for this step because it is the de facto standard platform for automotive software. We do this by automatically generating Simulink and Stateflow models from MechatronicUML models. This generation enables us exploit the complete MATLAB tool chain, e.g., for target code generation and simulation. However, we have to ensure that the generated Simulink and Stateflow models exhibit the same behavior as the original MechatronicUML models. Consequently, we give arguments in our presentation of the generation that it preserves the behavior and thus the verification results, too.
We illustrate the generation by a running example using the miniature robot BeBot. The BeBot is a small mechatronic system, which has been developed as a test bed for development with a strong focus on ad-hoc communication and collaboration with other BeBots. The BeBot consists of a base module that includes the electrical drive, wheels, the power supply, and a small processing unit mostly for motor control. BeBots can be equipped with an extension module with powerful information processing and wireless communication devices. As such, they provide a sound test platform for the development of innovative car functionalities like the aforementioned communication and collaboration to improve the illumination of streets.
In the next section, we present MATLAB/Simulink and Stateflow in more detail. Section 3 contains a presentation of the main concepts of MechatronicUML. The main contribution, the generation of MATLAB/Simulink and Stateflow models from MechatronicUML models, is presented in Section 4. We show how the complex communication and the time constraints of MechatronicUML are represented in MATLAB/Simulink and Stateflow. We conclude and give a brief outlook in Section 5.

2. MATLAB/Simulink and Stateflow

MATLAB is an environment for technical computing. Simulink is a platform for multi-domain modeling and simulation. It supports causal block-oriented modeling and the analysis of discrete-time

and continuous-time models. The Simulink simulation engine has several solvers for ordinary differential equations (ODE).
Stateflow extends Simulink by an environment for event-based reactive behavior specification. It uses the finite state machine concept (FSM) which is similar to Harels' statechart formalism [Harel 1987]. It supports hierarchical and parallel states. Stateflow supports modeling of complex control flow by transitions between these states and allows using MATLAB functions as a complex action language.
However, the Stateflow formalism has some drawbacks for modeling communication protocols with real-time requirements between distributed systems. Stateflow is an event-triggered modeling environment. However, it does not offer expressive features for specifying real-time behavior like in timed automata [Alur and Dill 1994] or Real-Time Statecharts. The modeler has to use helping elements from Simulink to count time-ticks in Stateflow. Additionally, Stateflow provides no support for asynchronous, message-based communication with buffers for sent and received messages.
Although Stateflow has output events which can be used to exchange information between different Stateflow blocks, these events are not buffered by the Stateflow receiver block. This means, if the receiver doesn't directly use the received events, these events are lost and cannot be used to coordinate systems. The buffering of messages is very important for the coordination of distributed mechatronic systems, as they are often physically separated and arranged in different locations. Therefore, mechatronic systems often cannot coordinate via shared variables. It is possible to encode asynchronous message-based communication with a combination of several linked Simulink and Stateflow blocks, but this is tedious and hard to maintain manually.

3. MechatronicUML

In this section, we explain MechatronicUML in more detail. The focus of MechatronicUML is the specification of distributed hybrid components that must fulfill real-time requirements such as deadlines. Formal verification techniques are applied to detect flaws of the discrete communication behavior during the design early. Combined with suitable structure and behavior specification languages, MechatronicUML is meant to reduce development time and cost.
The structure is specified by a component model that integrates the specification of the communication of discrete software components and continuous components such as feedback controllers. For the specification of the behavior, we use Real-Time Statecharts. This variant of statecharts provides a concept to specify real-time properties and defines semantics for the exchange of asynchronous messages.
In the following, we firstly introduce a running example (Section 3.1). Afterwards, we briefly describe the component model (Section 3.2). Next, we give an overview of Real-Time Statecharts (Section 3.3) with a focus on two main differences to MATLAB/Simulink and Stateflow: (1) a concept for the communication with asynchronous messages (Section 3.4), and (2) a more comprehensive concept for the specification of real-time properties (Section 3.5).

3.1 Running example

We explain MechatronicUML by using a scenario of a sophisticated collision avoidance system for a driver assistance system. It uses car-to-car communication for enhancing standard sensor-based distance detection by the exchange of precise position data. As mentioned in Section 1, we use the miniature robot BeBot as a test platform for such systems. In our scenario, BeBots have to navigate in a plain area without colliding with each other. For better understandability of the example, we focus on the specification of only two BeBots in the remainder of this paper. However, the specification can be easily extended to support an arbitrary number of BeBots instead of only two [Becker et al. 2012].
The communication of the BeBots is crucial for the implementation of our scenario. Each BeBot has to exchange position data with the other BeBot and adapt its driving direction accordingly. For a coordinated communication, we assign different roles, the distributor role and the client role, to the BeBots. One BeBot, which we call distributor BeBot, is assigned the distributor role. The distributor BeBot collects the position data of both BeBots and sends the data to the other BeBot. The other BeBot, called client BeBot, is assigned the client role. The client BeBot receives the position data of both BeBots and sends only its own position to the distributor BeBot.

3.2 Component model

A MechatronicUML specification for the structure of the two BeBots in our scenario is shown in Figure 1. The structure of a single BeBot is specified by a *component* in MechatronicUML. Since we use two BeBots in our scenario, we specify two representatives of the BeBot component, called *instances*: bebot1 that is assigned the distributor role and bebot2 that is assigned the client role. The communication between the BeBots is specified by ports and connections between the ports. For example, bebot1 has a distributor port that is connected to the client port of bebot2. For the integration of continuous components, we utilize two kinds of ports. First, *discrete port*s are used for message-based communication. A discrete port has a well-defined interface that defines which messages and parameters can be exchanged. Second, *continuous ports* exchange time varying quantities that have a value at all points in time. This definition is derived from the definition of signals in Simulink.

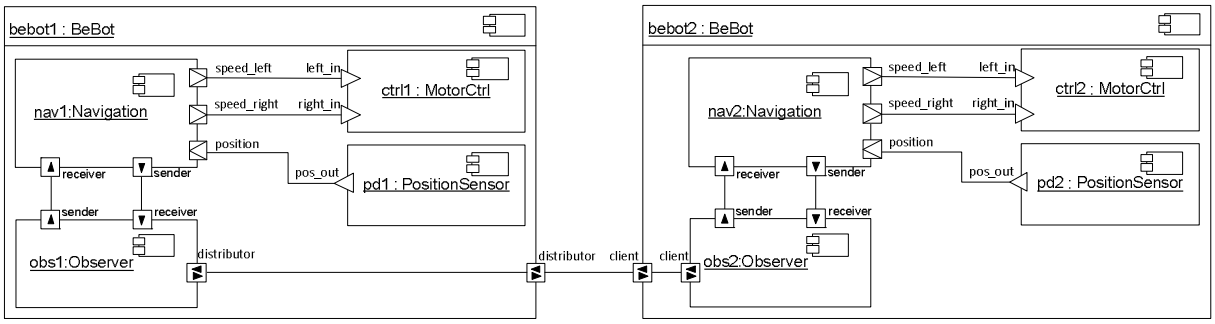

Figure 1. MechatronicUML specification of the structure of two BeBots navigating in an area

Each BeBot component embeds four subcomponents: the two controllers MotorCtrl and PositionSensor, and the two discrete software components Navigation and Observer. The MotorCtrl governs the two motors of the BeBot according to the continuous values that are received through the continuous ports left_in and right_in from the Navigation. The PositionSensor determines the current position of the BeBot and sends it via the continuous port pos_out to the position port of the Navigation.

Since motor control and wireless communication are placed on different modules (see Section 1), the functionality is separated into two discrete software components: the Observer that uses the wireless communication to exchange position data and the Navigation that is connected with the motor control to steer the BeBot through the environment. More precisely, the distance to and the direction of the other BeBot is encoded in a distance vector. This distance vector is received by the Navigation component from the Observer component through the receiver port. Then, the motor speed is calculated in accordance to the distance vector such that a collision between the BeBots is impossible. Based on the own BeBot's position data that the Navigation receives via the continuous port position, it sends its current position to the Observer through the sender port. This position is received by the Observer via the receiver port. In addition to the communication with the Navigation, the Observer implements the previously described behavior for the distributor and the client role. Next, we explain the advantages of Real-Time Statecharts by using the Observer's behavior as an example.

3.3 Discrete behavior model

In Real-Time Statecharts, messages are used to specify the communication between different statecharts. In particular, *asynchronous messages* are used to decouple the communication of statecharts between different components. For the specification of real-time properties, Real-Time Statecharts extend statecharts by *clocks* and corresponding *time guards* as used in timed automata. They describe the internal time interval in which a transition can be fired. Time invariants specify the duration the system is allowed to stay in a state. The formal semantics of Real-Time Statecharts is based on timed automata to enable applying formal verification techniques.

As an example, Figure 2 shows the Real-Time Statechart of the Observer component. It consists of one top-level state Observer_Main which has four orthogonal (parallel) regions: receiver, sender, client, and distributor. These regions are executed in parallel. Each region corresponds to a port of the

Observer component and contains an internal statechart that handles the communication for the corresponding port. If a port is not used in a specific instance of the Observer component, the corresponding region is not present in the statechart. For example, the region client does not exist for the Observer component instance in the distributor BeBot.

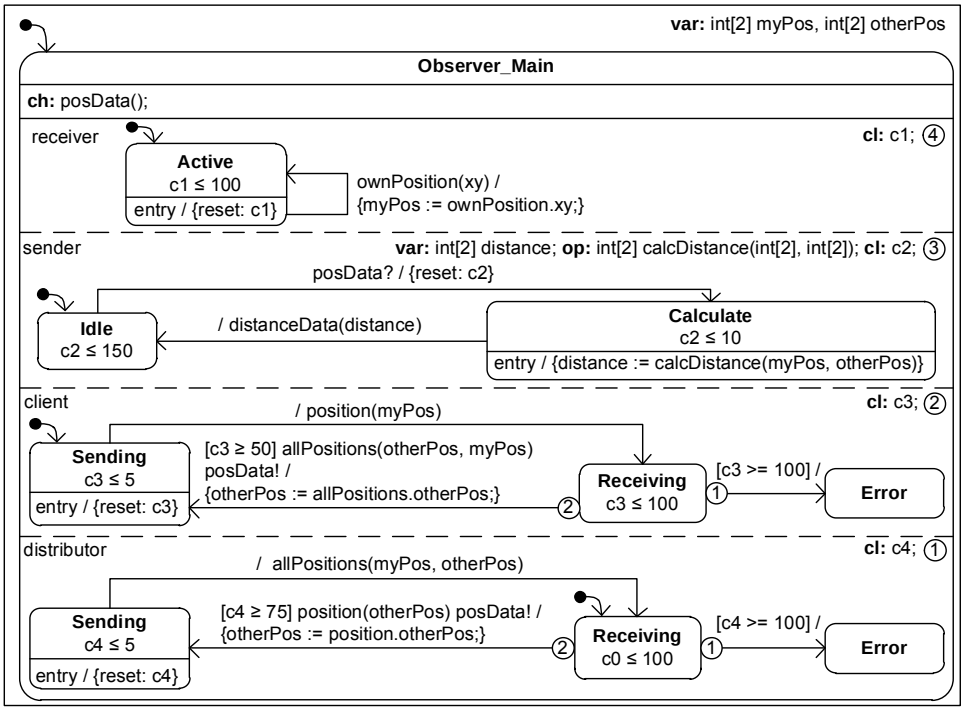

Figure 2. Behavior of the Observer component specified with a real-time statechart

The own position data and the other BeBot's position data are stored in two global variables myPos and otherPos that are defined in the upper right. The BeBot's own position is received in the receiver region from the Navigation. Each time the Navigation sends an ownPosition message with the current position, the position is stored in myPos. The other BeBot's position is received in the client or the distributor region depending on the BeBot's role. The distributor BeBot starts in the state Receiving and waits for the current position of the client BeBot. The client BeBot sends its position by the asynchronous message position to the distributor BeBot. After the distributor BeBot receives the position message, it changes to the state Sending in the distributor region. Moreover, it stores the position of the client BeBot in the variable otherPos, and sends the internal message posData to inform the sender region that new position data is available. Next, the distributor BeBot's Observer changes to the state Receiving again and sends the position data of both BeBots by the message allPosition to the client BeBot.

Simultaneously, in the region sender the transition from Idle to Calculate is executed after the posData message is received. When entering the state Calculate, the Observer executes the operation calcDistance() to determine the distance vector between the BeBots. During the execution of the transition from Calculate to Idle, this distance is send to the Navigation component.

3.4 Asynchronous messages

An asynchronous message is sent immediately. The receiver stores the message in a queue until it processes the message. During the execution of the receiving transition, the message is consumed and removed from the queue. Asynchronous messages do not get lost if they are not received right away. As an example, consider again the exchange of the position data between the client BeBot and the distributor BeBot. The client BeBot sends the position message with its current position through the sender port of the component to the distributor BeBot as specified by the component model in Figure 1. This message is received during the execution of the transition from the state Receiving to Sending

in the distributor region. As long as this transition is not enabled, i.e. the state Receiving is not active or the time guard is not evaluated to true, the message is stored in a queue.

3.5 Real-time properties

Real-Time Statecharts use clocks to represent the internal time of the system. These clocks are initially set to 0 time units. Note, that we use time units instead of SI-units. However, it is possible to transform time units to SI-units as we explain in Section 4.3. Clocks may be used in time guards of transitions and time invariants of states to restrict the system behavior. A transition may only be fired if the time guard evaluates to true for the current clock values, a state may only be active as long as the time invariant evaluates to true. In contrast to Stateflow, clocks are not automatically reset after the system changes to another state. Instead, the keyword reset is used to explicitly reset a clock on executing a transition, on entering a state, or on leaving a state. This semantics of clocks enables to easily specify more complex real-time constraints.
For instance, in the region distributor the clock c4 is reset on entering the state Sending (entry / {reset: c4}). The system is allowed to stay in the state Sending until c4 has reached 5 time units. The effect is that the clock c4 is always 0 on entering the state Sending, but be between 0 and 5 on entering the state Receiving. Therefore, the total amount of time that the system is allowed to stay in the Receiving state depends on the time the system stayed in the Sending state. If the system stayed 5 time units in the state Sending, it is only allowed to stay a maximum of 95 time units in the state Receiving. This affects also the time guard of the transition from the state Receiving to the state Sending. The time guard evaluates to true after c4 reaches 75 time units. Again, if the system stayed 5 time units in the Sending state and afterwards changes to the Receiving state, the time guard evaluates to true 70 time units after entering the Receiving state. If the client does not send its position data until the clock c4 has exceeded 100 time units, the communication is considered erroneous and the system changes to the Error state.

4. Generating MATLAB/Simulink and Stateflow models from MechatronicUML

In this section, we show how MechatronicUML models are translated into Simulink and Stateflow models. The main challenge for the translation is to preserve the behavior of the MechatronicUML model which has been formally verified. We especially need to consider this for the translation of asynchronous communication and time constructs which are not directly supported by Simulink and Stateflow. In the following, we outline the generation of Simulink block diagrams and Stateflow charts from MechatronicUML models in Section 4.1. Then, we highlight the elements that are used to realize asynchronous communication in Simulink and Stateflow in Section 4.2. Finally, we show how the advanced clock concept of Real-Time Statecharts may be encoded in Stateflow in Section 4.3.
We have implemented the generation of Simulink and Stateflow models from a given MechatronicUML model using a Triple-Graph-Grammar-based model transformation approach [Schürr 1994]. The tool is available for download as free software on our website[5].

4.1 Generating Simulink block diagrams from component definitions

The component specification of MechatronicUML is translated into a Simulink block diagram. The basic generation is straightforward. For each component, we create one subsystem block with the same name. The ports of the component are translated to inputs and outputs of the respective block. In contrast to Simulink, MechatronicUML supports bidirectional discrete ports, i.e., such ports may send and receive messages. Such ports are translated into an input and an output. Figure 3 shows the result of translating bebot1 of Figure 1 to Simulink.
Simulink and Stateflow provide no support for asynchronous, message-based communication with queues for sent and received messages. These concepts are supported natively by MechatronicUML and, thus, need to be encoded manually into Simulink and Stateflow. The CommunicationSwitch is one of the elements necessary for providing asynchronous message-based communication. It implements a dispatching of messages on one level of hierarchy in the system (cf. Section 4.2).

[5] http://www.cs.uni-paderborn.de/index.php?id=muml-simulation

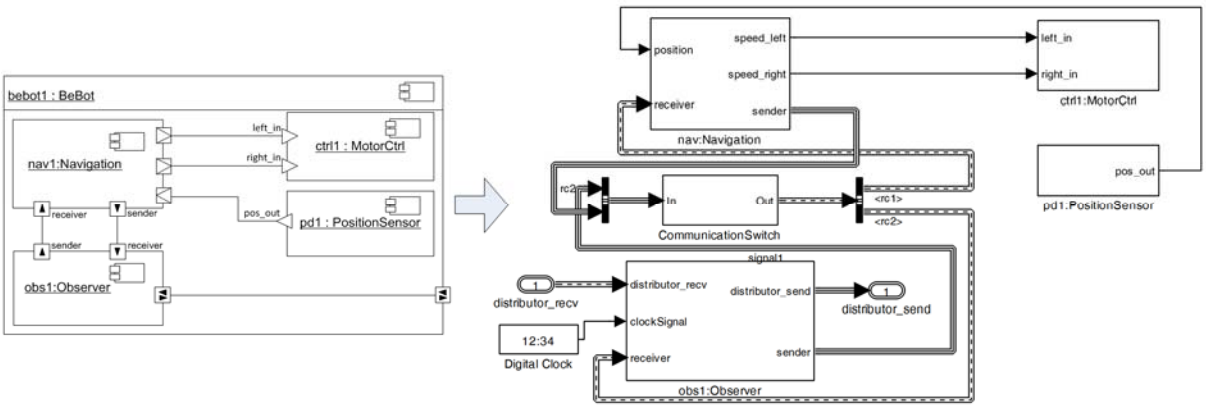

Figure 3. Simulink block diagram for the distributor BeBot

The controllers ctrl1 and pd1 of Figure 1 are connected via continuous ports to the Navigation component. Such ports are connected directly in Simulink.

In MechatronicUML, each non-hierarchical component contains a Real-Time Statechart. Each Real-Time Statechart is translated into a Stateflow chart. States, transitions, entry actions, and exit actions are translated to their counterparts in Stateflow. If a state of a Real-Time Statechart embeds parallel regions as, e.g., Observer_Main of Figure 2, we set the attribute Decomposition of that state to Parallel. For each region, we create one state in that top-level state with its attribute Decomposition set to Exclusive. That substate contains the Real-Time Statechart of the respective region. Figure 5 shows an excerpt of the Observer Statechart of Figure 2 where only the substate for the distributor region is modeled completely.

The priorities of regions and transitions in Real-Time Statecharts are encoded by a user defined execution order in Stateflow. In MechatronicUML, a high number indicates a high priority and, thus, the highest priority is translated to execution order 1. Lower priorities get increasing execution orders, respectively.

4.2 Using asynchronous message-based communication in Simulink and Stateflow

We enable asynchronous communication in Simulink by adding two additional types of blocks to the block diagram: a CommunicationSwitch block which is contained in hierarchical components and a Link Layer block which is contained in non-hierarchical components.

The CommunicationSwitch shown in Figure 3 dispatches asynchronous messages within a subsystem. Thus, all connections between message ports are replaced by a connection to and from the CommunicationSwitch. Thereby, the CommunicationSwitch decouples the implementation of a block from the message dispatching in the real system. The CommunicationSwitch is generated for simulation purposes and meant to be replaced by a network interface for the real system later.

A message is a six-tuple (package_id, sender_id, receiver_id, message_id, parameter, timestamp). The package_id is an integer assigning a sequential number to a message. This may be used to track lost messages. The sender_id is the network address of the sender; the receiver_id is the network address of the intended receiver of the message. The message itself is encoded by an unsigned integer in contrast to the Strings used in MechatronicUML because Simulink does not support variable sized Strings. In our implementation, each message may only contain exactly one parameter of type double. Thus, messages with more than one parameter need to be split into several messages. The timestamp encodes the point in time at which the message was sent. The messages are encoded into a bus signal using the six fields described above. This bus is then connected to the CommunicationSwitch which dispatches the messages according to their intended receiver.

In our example, the component Observer is a non-hierarchical component, i.e., it contains a Real-Time Statechart, but no other component parts. We translate such components as shown for the Observer in Figure 4. The resulting Simulink block contains a Stateflow block and a Link Layer block for each message port of the MechatronicUML component. For presentation purposes, we omit the sender and receiver ports and only show the Link Layer block for the distributor port.

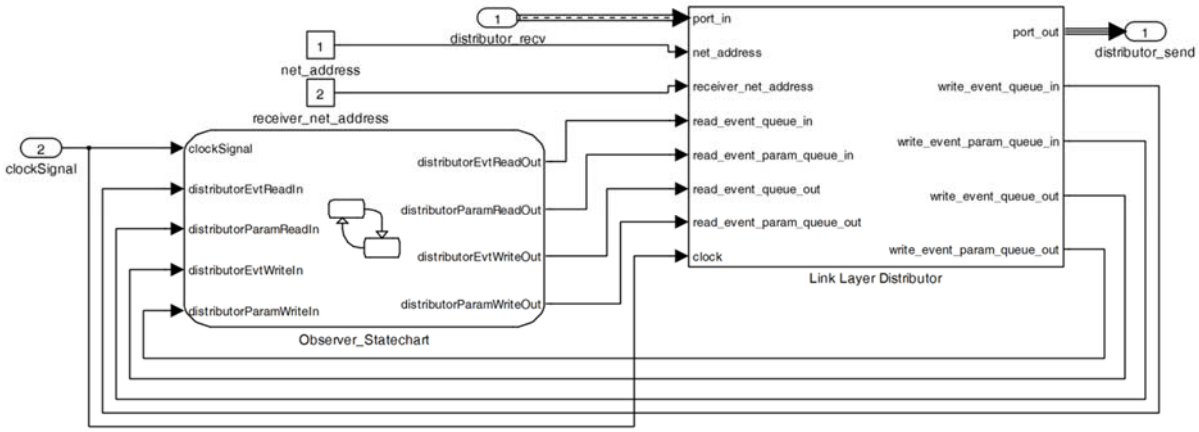

Figure 4. Simulink block diagram for Observer containing a Stateflow block

The Link Layer block takes the signals for the port (distributor_recv and distributor_send) as input and output signals as well as a uniquely identifying network address (net_address). The receiver_net_address is the network address of the receiver of messages which are sent via this Link Layer, i.e., the target port that this component is connected to. These addresses are then used by the CommunicationSwitch for realizing the message dispatching.

In addition, the Link Layer defines four queues that are used to buffer the messages. The in-queues (event_queue_in and event_queue_param_in) store the messages that are received via the input port and provide them to the Stateflow chart. Accordingly, the out-queues buffer the messages that are sent by the Stateflow chart and until they are sent via port_out. The Stateflow chart only uses the in-queues and out-queues provided by the Link Layer block and does not directly access the message bus from the CommunicationSwitch. Thus, the implementation of the Stateflow chart does not depend on the concrete implementation of the asynchronous communication in Simulink.

For modeling asynchronous messages in Stateflow, we utilize the queue signals shown in Figure 6 that are used as inputs and outputs of the Stateflow chart. In MechatronicUML, we distinguish between *trigger* messages that are received and enable transitions and *raised* messages that are sent when firing a transition. We use three embedded MATLAB functions checkQueue, dequeue, and enqueue for implementing that behavior. The use of these functions is illustrated in Figure 5 which shows an excerpt of the Real-Time Statechart of the Observer of Figure 2 containing only the contents of the distributor region.

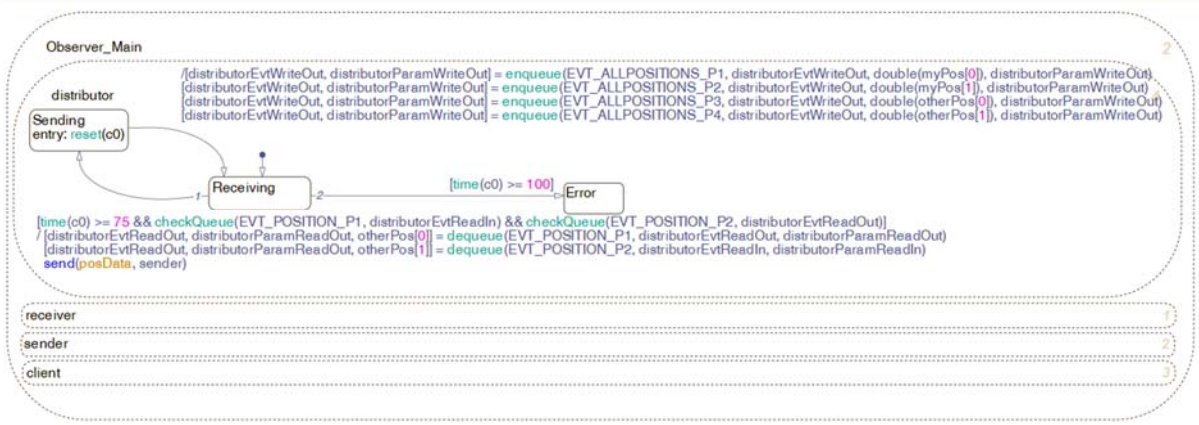

Figure 5. Stateflow chart for Observer showing an excerpt of the distributor behavior

In Figure 2, the transition from Receiving to Sending in region distributor is enabled by the trigger message position. In Stateflow, we may only fire the transition from Receiving to Sending if the message positions is contained in the queue for received messages. This preserves the semantics of MechatronicUML where a transition may only be fired if the message has been received. We check this by using the function checkQueue which returns true if and only if the message is contained in the

queue. Since the message carries an array of two elements and our implementation only supports one parameter per message, the message is split into two messages in Stateflow each carrying one value as a parameter. The message is consumed and thereby removed from the queue by calling the dequeue function which returns the changed queues as well as the received parameter value. The parameter value is then assigned to the variable otherPos. A raised message is translated to a call of the enqueue function which adds the message and the parameter to the respective out-queues. Then, the Link Layer block sends the respective message in the next simulation step. This preserves the semantics of MechatronicUML which requires the message to be sent right away. An example is given by the transition Sending to Receiving in Figure 5.
Figure 5 only shows a small excerpt of our example. It clearly shows that modeling asynchronous communication in Simulink and Stateflow is tedious and introduces a huge, additional amount of complexity when modeling this manually. Such additional complexity increases the possibility of introducing errors to the system. In MechatronicUML, we are able to reduce the visual complexity because we handle asynchronous communication as a first class entity of our modeling language. In addition, we are able to apply formal verification to show the correctness of interactions which is, in Stateflow, also hindered by the manual implementation of asynchronous communication.

4.3 Using real-time clocks in Stateflow

Real-Time Statecharts allow specifying complex timing constraints as described in Section 3.5. In contrast, Stateflow includes only simple temporal logic operators: its after() and before() operators can only be used to refer to the time elapsed since activation of the associated state. Especially, Stateflow does not provide clock variables that are independent from the associated state and can be used to refer to the time elapsed since their last reset. Therefore, the advanced timing concepts of MechatronicUML have to be mapped to helper constructs in Stateflow.
Each clock variable of MechatronicUML is mapped to a double variable in Stateflow. However, this variable is not accessed directly, but with the help of two embedded MATLAB functions: reset() to reset a clock variable and time() to retrieve a clock value. As shown in Figures 5 and 6, we use a Digital Clock block as an absolute time input (clockSignal) for the time calculation in a Stateflow chart. When resetting a clock using reset(), the current clockSignal is assigned to the clock variable. The time() function returns the difference between the current clockSignal and the value of the clock variable, i.e., the time since its last reset. Since MechatronicUML uses time units instead of SI-units, the user must specify a mapping of time units to SI-units. In our example, we assume that each time unit corresponds to 1ms. Then, time() automatically converts the clock value to the correct SI-unit, allowing to use the same constant values as in MechatronicUML. By using the clock values in guards, we restrict firing of transitions to the same time intervals which have been specified in MechatronicUML which preserves the semantics of our model.

5. Conclusion and outlook

In this paper we presented MechatronicUML, a modeling language which specifically targets the software embedded in technical systems. Real-Time Statecharts are used to model the discrete communication behavior between different systems. In contrast to existing development environments like MATLAB/Simulink and Stateflow, MechatronicUML provides sophisticated support for asynchronous communication and clocks. Furthermore, it supports the verification of safety properties, helping finding bugs early in the development, which is crucial to reduce development time and costs. For these reasons, we consider MechatronicUML a model-driven development technique highly suitable to tackle the increasing complexity of modern mechatronic systems.
As the main contribution, we described how a software specification in MechatronicUML can be automatically translated to Simulink and Stateflow retaining the original MechatronicUML semantics and, thus, the verification results. In particular, we map asynchronous communication in MechatronicUML to helper constructs like switches and Link Layer blocks that encapsulate this asynchronous communication in Simulink and Stateflow. In addition, we use digital clock blocks in combination with helper functions to translate the clock variables and clock constraints of MechatronicUML.

Thereby, we have combined the modeling and verification strengths of MechatronicUML with the advanced simulation and code generation capabilities of Simulink and Stateflow. We have implemented the generation of Simulink and Stateflow models from a given MechatronicUML model using a model transformation approach.
Besides the generation of Simulink and Stateflow models, we are currently developing a transformation from MechatronicUML to Modelica [Fritzson 2004]. Modelica suffers from similar drawbacks as Simulink and Stateflow, mainly missing concepts for clocks and asynchronous communication. Thus, similar concepts like the ones described in this paper have to be employed to implement such a transformation.
Modern mechatronic systems often incorporate self-adaptation or, in the case of systems of systems, dynamic communication topologies. This is also supported by MechatronicUML. However, Simulink does not support the dynamic instantiation of components which is necessary to model such dynamic system structures. Thus, as future work, we plan to investigate how such changing system structures can be represented in Simulink.

Acknowledgement

We thank Jana Bröggelwirth, Andrey Pines, and Andreas Volk for implementing the tool support for the transformation. This work was partially 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. This work was partially 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'. Christian Heinzemann and Jan Rieke are supported by the International Graduate School Dynamic Intelligent Systems funded by the state of NRW.

References

Alur, R,. Dill, D.L.,"A theory of timed automata", Theor. comput. sci., Vol.126, No.2, 1994, pp 183-235.

Baier, C., Katoen, J.-P., Principles of Model Checking, MIT Press, 2008.

Becker, S., Brenner, C., Dziwok, S., Gewering, T., Heinzemann, C., Pohlmann, U., Priesterjahn, C., Schäfer, W., Suck, J., Sudmann, O., Tichy, M., "The MechatronicUML Method – Process, Syntax, and Semantics", Technical Report no. tr-ri-12-318, Software Engineering Group, Heinz Nixdorf Institute, Paderborn, February 2012.

Fritzson, P.A., "Principles of object-oriented modeling and simulation with Modelica 2.1", Wiley, 2004.

Giese, H., Henkler, S., "A survey of approaches for the visual model-driven development of next generation software-intensive systems. Journal of Visual Languages and Computing, volume 17, pages 528–550, 2006.

Harel, D.,."Statecharts: A visual formalism for complex systems", Science of computer programming, Vol.8, No.3, 1987, pp 231-274.

Object Management Group. UML 2.4.1 Superstructure Specification, 2011. Document – formal/2011-08-05.

Pretschner, A., Broy, M., Krüger, I., Stauner, T., "Software engineering for automotive systems: A roadmap.". Future of Software Engineering, IEEE Computer Society, 2007, pp. 55-71.

Schäfer, W., Wehrheim, H., "The challenges of building advanced mechatronic systems". Future of Software Engineering, IEEE Computer Society, 2007, pp. 72-84.

Schürr, A.: "Specification of graph translators with triple graph grammars". Graph-Theoretic Concepts in Computer Science, 20th International Workshop, Vol. 903 of LNCS, Herrsching, Germany, 1994, pp. 151–163.

M.Sc. Christian Heinzemann
Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn,
Zukunftsmeile 1, 33102 Paderborn, Germany
Telephone: +49-5251-60-2306
Email: c.heinzemann@upb.de
URL: http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik.html