Modelica code generation from ModelicaML state machines extended by asynchronous communication

Uwe Pohlmann[1] and Matthias Tichy[2]

[1]Software Engineering Group, Department of Computer Science, University of Paderborn, Germany,
upohl@uni-paderborn.de
[2]Organic Computing, Department of Computer Science, University of Augsburg, Germany,
tichy@informatik.uni-augsburg.de

Abstract

Innovation in cyber-physical systems is today largely driven by embedded software. Thus, appropriate approaches have to be employed to handle the complexity that results from the multi-discipline nature of these innovative cyber-physical systems. Modelica as modeling language specifically targets these multi-discipline systems. The UML profile ModelicaML combines the graphical notation of the UML with the sound formal modeling provided by Modelica. ModelicaML currently does not support modeling asynchronous communication which is increasingly required when cyber-physical systems have to coordinate their behavior. In this paper, we present our approach for Modelica code generation from ModelicaML state machines which have been extended by asynchronous communication. We illustrate our approach by an extended two tanks system that contains two distributed controllers which coordinate themselves by message exchange.

Keywords UML 2.2, State Machine, ModelicaML, Modelica, State Graph2, MechatronicUML

1. Introduction

Today's cyber-physical systems such as aircrafts, spacecrafts, or automobiles are large and very complex. Engineers can design them virtually using digital computers before any physical prototypes are built. The development of virtual models of complex systems requires a close collaboration between engineers from different disciplines. To describe the integration of mechanical and electronic components in consumer products the word mechatronics originated in Japan around 1970. Mechatronics has come to mean multidisciplinary systems engineering and is the synergistic integration of physical systems, electronics, controls, and computers through the design process [5].

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

As all elements of a cyber-physical system interact with each other, engineers cannot engineer them independently. Instead, a tight integration in mechatronic design is the key element as complexity has been transferred from mechanical domain to electronic and computer software domains. "Mechatronics is the best practice for synthesis by engineers driven by the needs of industry and human beings" [5].

Modelica is an object-oriented, declarative, multi-domain modeling language for describing and simulating hybrid models. Such models can 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.

ModelicaML is a UML profile which extends the UML with concepts of Modelica and enables the modeler to specify advanced constructs like requirements for Modelica. The motivation of ModelicaML is to integrate Modelica and UML. UML's strength in graphical and descriptive high-level modeling is combined with Modelica's formal executable models for analyses and trade studies. Therefore, we use Modelica also as an action language for UML models. ModelicaML does not only target modelers who are familiar with UML. Modelica modelers will also benefit from using ModelicaML/UML for editing and maintaining Modelica models, because graphical modeling promises to be more effective and efficient than textual representation. The strength and efficiency of UML for system modeling and simulation has been proven in recent years. Common understanding of models for parties involved in development of systems results in high-quality models. Further, the combination of discrete-time and continuous-time simulation gives modelers a great benefit. The concrete ModelicaML profile documentation can be found in the technical report [16]. Pop et al. defined earlier versions of ModelicaML [14, 13].

Modelica and ModelicaML mainly focus on tightly coupled systems. In these systems the components (mechanical, electrical, embedded control, etc.) from different dis-

ciplines are tightly coupled for an optimal system performance [18]. But today's systems are increasingly distributed and form systems of systems. They typically coordinate via message passing. Statecharts respectively state machines are an appropriate modeling formalism for the specification of the coordination of these distributed systems. Currently, ModelicaML does allow state machines but it does not consider sending and receiving messages at transitions of the state machines.

In this paper, we present an extension of ModelicaML state machines for the specification of sending and receiving messages based on a running example. We present the syntax as well as the semantics of this extension. Finally, we explain the translation of these extended state machines to plain Modelica code.

We present the running example in the following section. In section 3, we present the message extensions to the ModelicaML state machines including the code generation to plain Modelica. We show the simulation results of our example in Section 5. After a discussion of related work in Section 6, we conclude in Section 7 and give an overview of future work.

2. Example

An example for a hybrid mechatronic system is a rainwater system with two tanks as shown in Figure 1. The use case for the example is that water flows into *tank 1* when it is raining. The tank collects all the rain until it is full. If the tank is full the additive water flows into the rain drainage. The second tank stands on a lower floor. It is a closed tank with no drainage. Periodically, water is taken from *tank 2*. Therefore, *tank 2* needs always a certain amount of water which it can get from *tank 1*.

Every tank has a sensor which measures the current level of water in the tank. Each tank has one controller. Both controllers communicate with each other via messages. Each controller has a message box which stores incoming messages. Each controller gets the current value of the level in its tank from the sensor. The controller of *tank 1* regulates the position of the valve. This valve blocks the pipe between the two tanks. When the valve is open, the water from *tank 1* flows into *tank 2*.

The controller of *tank 2* has the master role in the communication protocol when both controllers communicate with each other. Accordingly, the left controller has the slave role. The master controller asks the slave controller via messages to send some water. The slave controller commits or rejects this request. If *tank 2* has got enough water, the master controller asks the slave controller to close the valve of *tank 1* again.

3. State Machines with Message Semantics

In many cases discrete controllers have to interact with each other to achieve a common goal. Further, they are often physically separated and arranged in different locations. Therefore, they cannot access the same memory and communicate via shared variables. For this reason they have to use (parameterized)-messages to interact with each other.

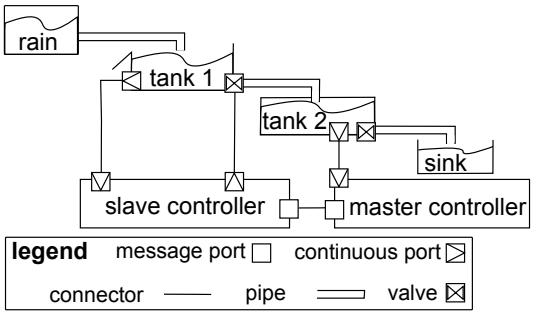

Figure 1. Two Tanks System (cf. [7, p. 391])

The communication needs a protocol as a formal description. We formalize protocols via state machines. Additionally, state machines are a good form to describe discrete behavior of a system.

Figure 2 shows the state machine which represents the master role of the communication protocol between the two controllers and Figure 3 shows the state machine which represents the slave role of the communication protocol between the two controllers. Figure 4 shows the state machines *protocolMasterControl* and *protocolSlaveControl* of both controllers which make the decisions which affect the further execution of the protocol behavior. A possi-

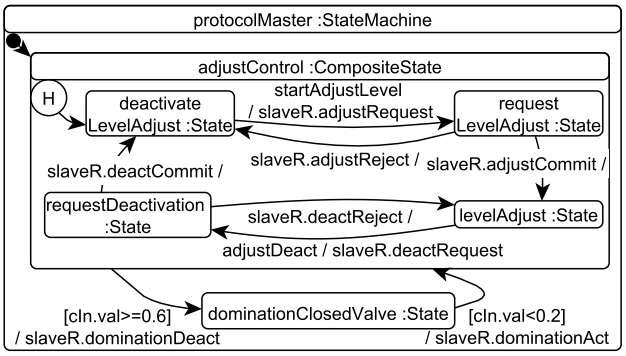

Figure 2. Protocol Behavior of the Master Controller

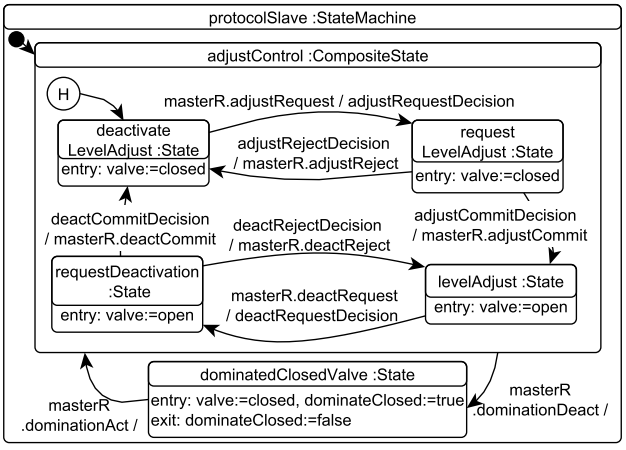

Figure 3. Protocol Behavior of the Slave Controller

ble communication scenario and the resulting sequence of messages are shown in the sequence diagram in Figure 5.

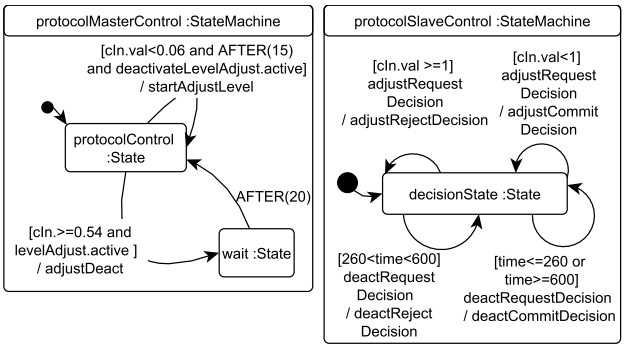

Figure 4. Protocol Control Behavior of the Controllers

For each message that is sent in this example scenario a directed edge is drawn from the sender to the receiver state machine. The *protocolMasterControl* state machine sends the message *startAdjustLevel* to the *protocolMaster* state machine when the level of water in *tank 2* is below a certain level and a minimum of 15 seconds is over. This message starts the protocol behavior. The *protocolMaster* state machine sends the message *adjustRequest* to the slave controller. The slave controller can agree or reject to open its valve. The slave controller opens the valve of *tank 1*, if the state machine *protocolSlaveControl* agreed by sending the message *adjustCommitDecision*. Then water flows from *tank 1* to *tank 2*. The master can ask the slave controller to close the valve if it is in the state *levelAdjust*. The slave controller can reject or commit this proposal.

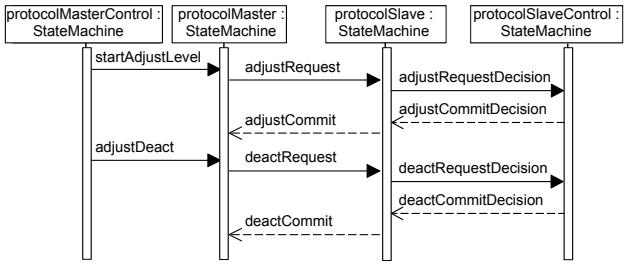

Figure 5. Communication Scenario

To show a more complex state machine with hierarchy and history we include a domination state to the state machines. In this state the master controller dominates the slave controller and forces the slave controller to close its valve if the amount of water of *tank 2* is too high. In real systems this may not be the best solution to model such a behavior. To dominate the slave controller the master controller changes to the state *dominationClosedValve* and forces the slave controller to close the valve of *tank 1*. Before the state change occurs the history state stores the most recent active state of the composite state *adjustControl*. The message *dominationDeact* informs the slave controller about the state change.

The master controller reactivates the most recent active state of the composite state *adjustControl* when the amount of water is below a certain level. The slave controller also reactivates its most recent active state of *adjustControl* when it receives the message *dominationAct* from the master. The whole coordination protocol represents a reactive communication between both controllers.

In the next section we give a brief introduction of syntax and semantics of ModelicaML state machines. Afterwards, we describe the semantics for messages. Messages are currently not defined for ModelicaML state machines. After this, we show a mapping from ModelicaML state machines extended by messages to Modelica code.

3.1 ModelicaML State Machines

State machines mainly consist of states and transitions between states. States can contain regions to add hierarchy or orthogonal behavior. The hierarchy of states and regions builds a tree structure. A configuration of a state machine defines all active states at a point in time.

A state is entered and activated when an incoming transition fires and a state is exited and deactivated when an outgoing transition fires. A state can have an entry-action, which is executed when the state is entered, an exit-action, which is executed when a state is exited, and do-actions, which are executed as long as a state is active.

A transition is enabled when the boolean guard expression is true and, if required, the boolean message variable is true. If more than one transition is enabled the transitions are in conflict with each other. Only the transition with the highest priority of those transitions, which are in conflict with each other, fires. The transition with the lowest priority integer value has the highest priority.

3.2 Semantics Definition by Modelica

In this section we address the semantics of state machines with the target to map this semantics to Modelica language constructs. A special focus is set on the message semantics. We translate state machines into Modelica algorithmic codes. The state machine semantics definition is closely linked to the UML semantics definition. However, there are some issues in which the state machine semantics differs from UML. This is based on the target language Modelica.

The state machine semantics does not change the synchronous data flow principle of Modelica. The developer must explicitly model the real-time characteristics.

The UML defines the behavior of state machines as follows:

> "*Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences.*" [19, p.564].

The state machines react immediately to each value change of a model variable because Modelica evaluates the algorithm section of the code continuously, namely, after having finished continuous time integration steps and at Modelica event iterations. For a description of the discrete/ continuous modeling/simulation in Modelica based on the synchronous data-flow principle see e. g. Otter et al. [10].

Schamai et al. discuss the execution semantics of ModelicaML state machines [17]. A more detailed definition of ModelicaML state machines syntax and semantics is given by Pohlmann [12].

3.3 Message Syntax and Semantics

A message implementation in Modelica exists in the DEVS approach [15]. We use this approach for sending and receiving messages between different state machines. We define messages by modeling them as operations of the owner class of the state machine. In the two tanks example the messages are defined as operations in the classes *DiscreteMasterController* and *DiscreteSlaveController* which are the model classes of the components *masterController* and *slaveController* as shown in Figure 6. Additionally, the Figure shows the composite structure of the whole system modeled with ModelicaML.

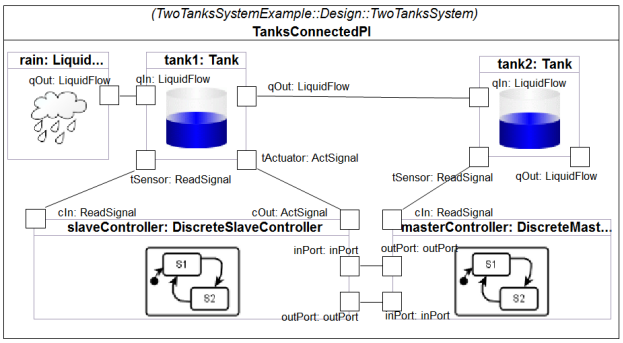

Figure 6. Component Structure

A good way to define messages is to model them as UML operations of the owner class of the state machine. In the two tanks example the messages are defined as operations in the class *BaseController*. The class *BaseController* is the abstract parent class of the model classes *DiscreteMasterCotroller* and *DiscreteSlaveController*. Figure 7 shows the inheritance relation of the involved classes and a part of the needed messages. The state machines of both child classes can use all defined messages. The syntax is the same as in UML for operations.

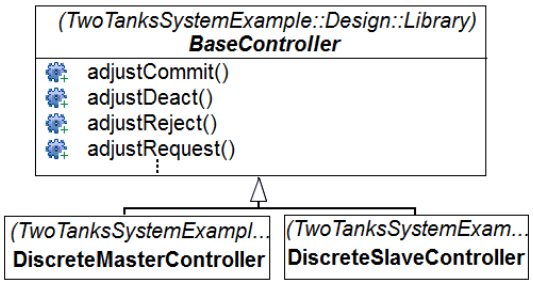

Figure 7. Message Definition of the Controller Classes

In the two tanks example messages could be sent from *slaveController* to *masterController* and vice versa. The destination of a message from *slaveController* to *masterController* is the *inPort* of *masterController*. The *outPort* of *slaveController* is connected via a connector to the *inPort* of *masterController*. In a similar way is the *outPort* of *masterController* connected to the *inPort* of *slaveController*.

In ModelicaML the causality of ports can be specified. They can be declared as input or output ports. In general, a port with causality *input* receives messages and via a port with causality *output* messages are sent. The transmission of messages is instantaneous when no behavior for the connector is defined. The content of a message is independent from its type. Accordingly, a message can contain parameters. Further, various messages can be received simultaneously over an *input*-port or sent via an *output*-port.

An output-port can only be connected to an input-port. An input-port is associated with the destination object. Each message port has its own message box, also called message pool. Additionally, each component has a component message pool which collects all messages from all of its message ports. This makes it easier to search for a certain message, because only the component message pool has to be searched and not all port message pools.

Figure 8 shows the internal representation of the message pools and the message flow via the port between the two controllers. Each component has the input-port message pool *inPortMessagePool* and the component message pool (*controllerMessagePool*).

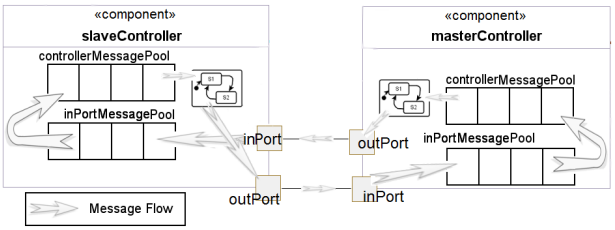

Figure 8. Internal Message Pool Representation and Message Flow

A component monitors the message pools by periodically sampling the status of the message pool. Before the system can react on a received message, the message pool is completely iterated and all received variables are read. For each message type one boolean variable exists. An available message is dispatched from the message pool and the corresponding variable becomes true, if it previously was false. If the message variable is true, the message is put back into the message pool. The message queue is ordered as a FIFO-queue. The execution semantics mostly does not depend on the order of messages in the message pool. It depends only on the order of messages in the message pool if messages from the same type are received via different ports at the same time. In this case we use an explicit order of the input-ports to get an explicit prioritization. We do not differ between messages in the message pool which we received from other components or messages which are raised within the same component. At one event iteration at a time instance it is not possible that multiple transitions react on the same message type. If there are multiple transitions they have to wait until the next event iteration.

Figure 9 shows an example. Both state machines are currently located in the marked states $A2$, $B2$. In the previous step the transition $A1 \rightarrow A2$ sent *message1* and simultaneously, the transition $B1 \rightarrow B2$ sent *message2*. The Figure shows the current state of the component message pools below the components. However, the relevant information

for the execution semantics is the priority order of the enabled outgoing transitions ($A2 \rightarrow A3$ [priority value = 2], $A2 \rightarrow A5$ [priority value = 3]) of state $A2$. Transition $A2 \rightarrow A3$ has the higher priority and fires. *Message2* is dispatched and state $A3$ becomes active.

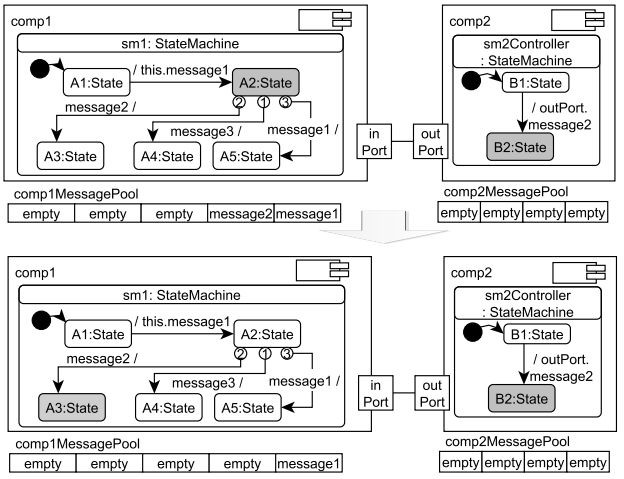

Figure 9. Message Execution Semantics

4. From State Machines to Modelica Code

The translation from ModelicaML models to Modelica consists of two steps. Firstly, the static semantics of the model is validated. The validation process automatically traverses the state machine graph. During this traversing a template checks the model for certain conditions such as if each composite state has at least one initial state. For each found error an error marker is set in the Eclipse problem view. The modeler can start the validation process at any time. Secondly, the modeler can start the code generation process if the model is error free. We use the template-based model-to-text approach of Acceleo[1] for the code generation[2].

4.1 Modelica Code for the State Machine Structure

The code generation of the static part of state machines generates a Modelica record structure for each state machine and its regions and states. A Modelica record is a class for specifying data without behavior. The behavior is located in an algorithm section of the state machine owner model class.

The state machine and each state have the variables *active*, *timeAtActivation*, *stime*, and *selfTransitionActivated*. The variable *active* indicates whether a state is active ($active = 1$) or inactive ($active = 0$). The variable *timeAtActivation* is set to the current simulation time when the state is activated. The variable *stime* is a local timer. The value is calculated by the statement $time - timeAtActivation$. The variable *selfTransitionActivated* is an auxiliary variable, which is needed to identify if a self-transition has been fired. Listing 1 shows the Modelica record definition of a simple state.

[1] http://www.acceleo.org

[2] The developed code generator for ModelicaML state machines without messages is a result of the master thesis of Pohlmann [12] which was done in a cooperation with Schamai from EADS Innovation Works. It is available at http://www.openmodelica.org/index.php/developer/tools/134.

Listing 1. Simple State Record Definition

```
record SimpleState
  Boolean active;
  Real timeAtActivation;
  Real stime;
  Boolean selfTransitionActivated;
end SimpleState;
```

The Modelica record definition of a state machine consists of several variables. The variable *startBehavior* is used to initialize the state machine behavior. Each region has the variable *numberOfActiveStates*. This variable is used to assert that in a region there is never more than one state active at the same time. Listing 2 shows the Modelica structure code for the *protocolMasterControl* state machine shown in the left part of Figure 4.

Listing 2. State Machine Record Definition

```
record masterController_SM_protocolMasterControl
  Boolean active;
  Real timeAtActivation;
  Real stime; // local timer.
  Boolean selfTransitionActivated;
  Boolean startBehavior;
  protocolMasterControl_Region_0 Region_0;
  // REGION records
    record protocolMasterControl_Region_0
    // SIMPLE STATES instantiation
    SimpleState protocolControl;
    SimpleState wait;
    InitialState Initial_0;
    Integer numberOfActiveStates;
   end protocolMasterControl_Region_0;
...
end masterController_SM_protocolMasterControl;
```

A special record is required if, instead of an initial-pseudostate, a *shallowHistory*-pseudostate is used like in Figure 2. The record of the composite state *adjustControl* must have a type with an enumeration of all states of the region. Further, the variable *lastActive* is an instance of this type. Listing 3 shows an example for such a record.

Listing 3. ShallowHistory Record Definition

```
record Region_0_HistoryState
  Boolean active;
  Real timeAtActivation;
  Real stime;
  Boolean selfTransitionActivated;
  type HistoryStateT = enumeration(
     deactivateLevelAdjust, requestLevelAdjust,
     levelAdjust, requestDeactivation);
     HistoryStateT lastActive;
end Region_0_HistoryState;
```

Besides the generation of the state machine structure, the structure for messages is implemented. A message is stored as a record. This record has at least the integer variable *msgType*. The *msgType* value must have a unique

value, because it is the identifier of the message type. Further, a message can have user defined attributes. Listing 4 shows an example of a message record. It has the integer variable *port* which encodes the port-id over which the message should be sent and the integer variable *msgType* which identifies the type of a message. Additionally, attributes which hold parameterisable values could be added.

Listing 4. Message Type Record Definition

```
replaceable record stdMessage
    Integer port;
    Integer msgType;
end stdEvent;
```

For the two tanks example the message type values are encoded as defined in Table 1.

10 = adjustRequest	11= adjustCommit
12 = adjustReject	13 = adjustDeact
14 = deactRequest	15 = deactCommit
16 = deactReject	17 = dominationDeact
18 = dominationAct	19 = startAdjust
20 = startAdjustLevel	21 = adjustRequestDecision
22 = adjustRejectDecision	23 = adjustCommitDecision
24 = deactRequestDecision	25 = deactRejectDecision
26 = deactCommitDecision	

Table 1. Message Type Encoding

Because of the limitations that Modelica cannot handle data structures with a dynamic size, such as a linked list, we implement the message mechanism like Sanz [15] by using Modelica external functions. The external function invokes a C-implementation that stores messages in the dynamic memory. The dynamic memory address of the component message pool is stored in the variable *cmpMsgPoolAdr*. Additionally, for each message port the variables *inputMsgPoolAdr*, *outputMsgPoolAdr* are used. They store the dynamic memory location of the port message pools. The variables *numIn*, *numOut*, *numreceived* are auxiliary variables. For each message a boolean variable exists which has the name of the message. Listing 5 shows the code which is needed for the messages and pools of *component1*.

Listing 5. Message Representations

```
//Number of Ports
parameter Integer numIn = 1
parameter Integer numOut = 1
//Memory address of the pools
Integer cmpMsgPoolAdr;
Integer inputMsgPoolAdr[numIn];
Integer outputMsgPoolAdr[numOut];
// Number of received messages
Integer numreceived;
// Message occurrence variables
 Boolean adjustCommit;
 Boolean adjustReject;
 Boolean adjustDeact;
 Boolean adjustRequest;
 Boolean deactRequest;
 Boolean deactCommit;
 Boolean deactReject;
 ...
```

4.2 Modelica Code for State Machine Behavior

The order of the generated algorithmic code is very important for the semantics of the state machines. The order of the resulting Modelica algorithm code is defined as pseudocode by the Algorithms 4.1, 4.2, 4.3, and 4.4.

At the beginning of the Modelica algorithm code, the initialization of the state machine is stated. Afterwards, the code for message handling is stated, if required. Accordingly, the region codes are generated in the order of their execution order which is defined by a priority value.

Algorithm 4.1: STATEMACHINEBEHAVIORCODE($StateMachine$)

```
initializeStateMachine
messageHandlingStatements
for each Region.sortRegion()
  do {REGIONBEHAVIORCODE(Region)
```

Algorithm 4.2 shows the structure of the region behavior code. The region behavior code starts with the local time management. This local time management sets and calculates the variables *timeAtActivation* and *sTime*. Afterwards, the history nodes, if present, are set to the currently active states. Now the transition code is generated for the region. Accordingly, the code for the do-actions is generated. At the end of the region behavior code the region behavior code of composite states and submachine states is stated. The code generation invokes the Algorithm 4.2 REGIONBEHAVIORCODE recursively at this point.

Algorithm 4.2: REGIONBEHAVIORCODE($Region$)

```
set local Time Behavior
set History Node to Current Active State
TRANSITIONBEHAVIORCODE(Region)
execute do code
for each CompositeState
  do { for each Region.sortRegions()
       do {REGIONBEHAVIORCODE(Region)
for each SubMachineState
  do { for each Region.sortRegions()
       do {REGIONBEHAVIORCODE(Region)
```

Algorithm 4.3 shows the structure of the transition behavior code. The code starts with the behavior code for the initial- and shallowHistory-pseudostates. These states are auxiliary states and are directly processed in favor. The behavior code for each state is nested within an if-clause. This clause is only processed if the parent state is still active. Therefore, it is ensured that transitions can never fire if the parent is not active.

Algorithm 4.3: TRANSITIONBEHAVIORCODE($Region$)

```
initialBehaviorCode
shallowHistoryBehaviorCode
if (parent is still active)
  then { for each State
         do { if (state is pre active)
              then {TRANSITONCODE(State)
```

Algorithm 4.4 shows the structure of a simple transition code. The concrete execution of a compound transition depends on the pseudostates involved in the chain of transitions from one state to another. This example directly connects two states. The transition with the highest priority is in the if-clause. Then the other transitions follow in the order of their priority in the else-if-clauses. If the guard is empty the condition is set to true. A trigger is optional. Entry-, and exit-actions are only generated if available.

Algorithm 4.4: TRANSITIONCODE(*State*)

comment: First Transition (Highest Priority)

if guard [and trigger]

then
- execute exit-action
- deactivate active state
- execute effect
- activate transition target state
- execute entry-action

comment: Second Transition (Lower Priority)

else if guard [and trigger]

then
- execute exit-action
- deactivate active state
- ...

In the following sections we describe the Modelica code for particular elements.

State Machine Initialization State machines are initialized when the simulation is started. The initialization sets all initial- or history-pseudostates of all regions to true.

Further, the message pools for the component and all discrete input-ports are created. Listing 6 shows the algorithmic code which creates the needed message pools initially. Listing 7 shows the invoked Modelica function which calls the corresponding *C*-Code <*events.c*>[3] of a message pool from Sanz.

Listing 6. Initialization of Message Pools

```
for i in 1:numIn loop
    inputMsgPoolAdr[i] := CreatePool();
end for;
cmpMsgPoolAdr := CreatePool();
end if;
```

Listing 7. Modelica Function which creates a new pool for messages in the dynamic memory

```
function CreatePool
output Integer q;
external "C" q =  QCreate();
  annotation (Include="#include <events.c>");
end CreatePool;
```

Message and MessageTrigger A message event is the dispatch of an asynchronous message instance. A boolean message trigger represents the receipt of this instance. A message can be created and sent by invoking a special *sendMessage* Modelica-function by a transition action. The message function is similar to the *sendEvent*-function of Sanz [15]. The *sendMessage* function is shown in listing 8.

Listing 8. Sends a message to a pool

```
function sendMessage
  input Integer poolAdr;
  input stdMessage e;
  output Integer out;
external "C" out =
        QAdd(poolAdr,e.port,e.msgType,e.value,0);
  annotation (Include="#include <events.c>");
end sendMessage;
```

At the beginning of the state machine behavior code the message handling code is stated. This code has the function to handle incoming messages and to put them from the different input-ports into the message pool of the component. Further, it handles the occurrence of messages in the message pool. If a message occurs and the corresponding variable is false, it sets the variable to true.

Listing 9 shows an example of the message handling. In the first for loop all messages from all input-ports are collected in the component message pool. The address of the memory location of the message pool is stored in the variable *cmpMsgPoolAdr*. Afterwards, the component message pool is searched for messages by the statement `message := getMessage(cmpMsgPoolAdr);`. The value of the variable *message.msgType* is used to check if a certain message is in the pool. If a message is in the pool then the corresponding message variable is set to true. For example if a message of `message.msgType=10` is in the pool and the variable *adjustRequest* is false, then the variable is set to true. Otherwise the message is put back into the message pool by the statement `sendMessage(cmpMsgPoolAdr,message)`.

Listing 9. messageHandlingStatements

```
for i in 1:numIn loop
    numreceived := numMessages(inputMsgPoolAdr[i]);
    for j in 1:numreceived loop
        message := getMessage(inputMsgPoolAdr[i]);
        message.port := i;
        sendMessage(cmpMsgPoolAdr,event);
    end for;
end for;
numreceived := numMessages(cmpMsgPoolAdr);
for j in 1:numreceived loop
    message := getMessage(cmpMsgPoolAdr);
    if message.msgType == 10
        and adjustRequest == false) then
        adjustRequest := true;
    ...
    else
        sendMessage(cmpMsgPoolAdr,message);
    end if;
...
```

A transition, which wants to react on the occurrence of a specific message, uses the boolean message variable as reference in its guard. When the message is available the guard becomes true and the transition fires and the transition action is executed. The transition action

[3] http://www.euclides.dia.uned.es/DESLib/Files/DESLib_1.2.zip

resets the message variable back to false. For example in Figure 2 the transition from $requestLevelAdjust \rightarrow deactivateLevelAdjust$ has as guard the variable *adjustReject* and the transition action statement `adjustReject:=false`. At the moment when the message *adjustReject* arrives from *protocolSlave* the variable is set to true by the message handling. The transition fires and resets the variable back to false, so that the next time when the message is available it could be set to true.

Local Time Behavior Each state has the time variables *timeAtActivation* and *stime*. The variable *timeAtActivation* is set to the current time when a state is entered. The variable *stime* is set in each integration step to the difference of the current simulation time and the *timeAtActivation*. If a state is not active the local timer *stime* is set to zero.

Initial/ShallowHistory-Pseudostate Behavior If the *initial*-pseudostate is active it is deactivated and the target of the outgoing transition is activated. If the *shallowHistory*-pseudostate is active the most recent active state is activated. If no most recent active state exists the target of the outgoing transition is activated.

Simple Transition Behavior A simple transition is a transition which directly connects two states. It is activated if the source state is pre-active. The Modelica pre-function is used to ensure that a state is active at the beginning of the event iteration. Therefore, it is not possible that a state becomes active and not active during the same iteration of an algorithm section.

For example if the state *deactivateLevelAdjust* from the state machine *protocolMaster* (see Figure 2) is active, the message *startAdjustLevel* has already arrived and therefore the boolean variable *startAdjustLevel* is true. Then the transition fires and sends the message *adjustRequest* to the state machine *protocolSlave* (Figure 3) of component *slaveController*. Listing 10 shows the resulting Modelica code.

Listing 10. Simple Transition Behavior

```
if pre(protocolMaster.Region_0
  .adjustControl.active) then
if pre(... .adjustControl.
    deactivateLevelAdjust.active) then
 if  startAdjustLevel then //guard
    //...; exit behavior
    ... .deactivateLevelAdjust.active := false;
    //start effect behavior
    startAdjustLevel:=false;
    message.msgType:=10;
    sendMessage(pre(cmpMsgPoolAdr,message);
    //end effect behavior
    ... .requestLevelAdjust.active := true;
     // ... ; entry behavior
  end if;
end if;
end if;
```

Highlevel Transition Behavior Highlevel transitions are outgoing transitions of composite or submachine states. When a highlevel transition deactivates these hierarchical states it has to be ensured that all active nested substates are deactivated before the composite-/ submachine is deactivated. Consider the state machine in Figure 2. Suppose, the state *adjustControl* is activate and the guard `cIn.val>=0.6` is true. Then the currently active state *deactivateLevelAdjust*, *requestLevelAdjust*, *levelAdjust* or *requestDeactivation* is deactivated. Afterwards, the state *adjustControl* itself is deactivated. Accordingly, the highlevel transition $adjustControl \rightarrow dominationClosedValve$ is taken and the message *dominationDeact* with message type identifier *17* is sent to the *protocolSlave* state machine. Finally, state *dominationClosedValve* is activated. Listing 11 shows the corresponding Modelica code.

Listing 11. Highlevel Transition Behavior

```
if pre(protocolMaster.Region_0
                .adjustControl.active) then
  if(cIn.val>=0.6) then
    if (...deactivateLevelAdjust.active) then
      ...deactivateLevelAdjust.active:=false;
    end if;
    if (...requestLevelAdjust.active) then
      ...requestLevelAdjust.active:=false;
    end if;
    ...
    ...adjustControl.active := false;
    message.msgType:=17;
    sendMessage(pre(outputMsgPoolAdr[1]),message);
    ...dominationClosedValve.active := true;
  end if;
end if;
```

5. Evaluation

We modeled the rainwater two tanks system as shown in Figure 1 with ModelicaML. The resulting composite structure of the system is shown in Figure 6. We modeled the sketched state machines of Figures 2, 3, and 4 with ModelicaML state machines and embedded them as behavior of the model classes *DiscreteMasterController* and *DiscreteSlaveController*. Further, we extended this model with our Modelica-Functions, like *sendsMessage* as shown in Listing 8. Further, we added Modelica code for the message passing and variables for our message types to the ModelicaML model. We generated from that ModelicaML model the whole Modelica code. The model and the generated code is available online at[4]. We simulated it with Dymola 7.4.

Figure 10 shows a part of the simulation results. The variable *rain.qOut.lflow* in the upper diagram shows the amount of rain which flows into *tank 1*. The inflow of *tank 1* is not controllable by the slave controller of *tank 1*. At simulation $time = 16$ the rain decreases. The variable *tank1.qOut.lflow* shows the amount of water which flows from *tank 1* into *tank 2* when the valve is opened. The valve is opened at simulation $time = 15$. Then $0.6\frac{m^3}{s}$ water leave *tank 1* for 24.3 seconds. The diagram in the middle shows the resulting water level in both tanks. When the valve is opened the water level of *tank 1* decreases and the water level of *tank 2* increases. In our example *tank*

[4] `http://www.cs.uni-paderborn.de/fileadmin/Informatik/FG-Schaefer/Personen/upohl/downloads/TwoTanksSystemExample.zip`

2 is bigger than *tank 1* and therefore the amount of water in *tank 2* increases slower than the decrease of water in *tank 1*. The lower diagram shows the state *levelAdjust* of the master controller. At $time = 15$ the state *levelAdjust* become active, because the self-transition of state *protocolControl* in state machine *protocolMasterControl* fires (see Figure 4). This starts the communication between both controllers to open the valve. At $time = 39.3$ the state *levelAdjust* is deactivated and the valve is closed.

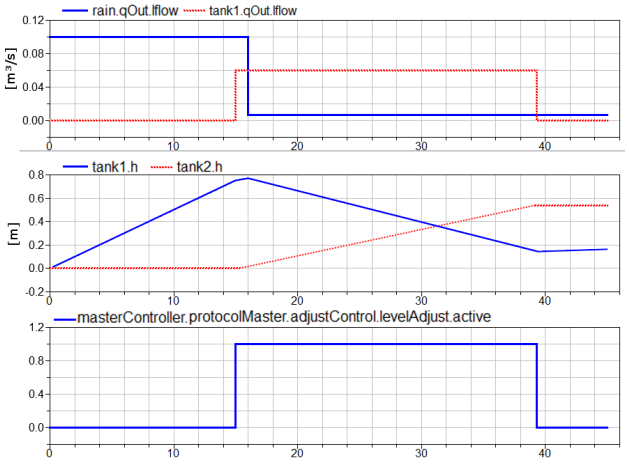

Figure 10. Simulation Results

The sequence diagram in Figure 11 shows the communication that appeared within the simulation. The messages coincide with our predicted scenario (see Figure 5). For each message that is sent a directed edge is drawn from the sender to the receiver state machine. Additionally, the time when the message is sent is annotated below the edge. When a message is drawn below another message and has the same time of sending it is sent after the upper message. Currently, ModelicaML only supports the specification of timing behavior in a rudimentary way. Therefore, we do not specify discrete timing behavior in our protocol. As a result it is difficult to grasp the timing behavior of the discrete controller and the communication protocol. This can be seen within the sequence diagram, because several messages that are sent one after another have the same time of sending.

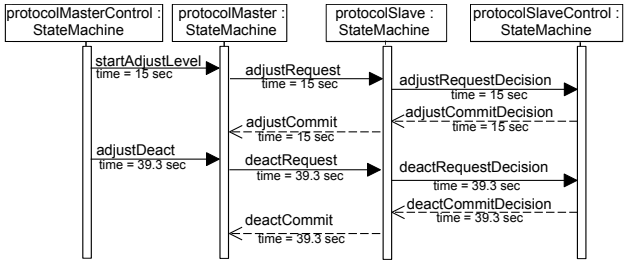

Figure 11. Communication Simulation with Concrete Time Annotation

6. Related Work

6.1 MechatronicUML

MechatronicUML [2, 9, 8, 4] is an approach for the model-driven development and verification of mechatronic real-time systems. The MechatronicUML refines the UML in order to make it applicable to mechatronic real-time systems. The component-based architecture of mechatronic systems is modeled using discrete and continuous components. Their discrete behavior is modeled using real-time statecharts, an extension of UML state machines with constructs from timed automata [1]. Continuous behavior is modeled with the help of the CAMeL-View [3] tool which allows the object-oriented modeling of different parts of mechatronic systems like multi-body system dynamics, control technology, and hydraulics. The MechatronicUML focuses on the real-time coordination between mechatronic systems and supports its formal verification with respect to safety properties. The presented extension of ModelicaML with message exchange has been based on the MechatronicUML. However, it enables the seamless modeling of mechatronic systems using a single formalism and tool in contrast to the MechatronicUML.

6.2 StateGraph2

The Modelica StateGraph2 library [11] is a free Modelica library providing components to model discrete events, reactive and hybrid systems with deterministic hierarchical state diagrams. It utilizes Modelica as action language. Via special blocks actions can be defined in a graphical way depending on the active step. StateGraph does not support a comprehensive set of state machines as defined in the UML and reused in ModelicaML. Larger StateGraph models are not easy to grasp as they are graphically more complex than UML state machines. Additionally, in contrast to our approach, StateGraph does not support message exchange.

6.3 SimulationX

The SimulationX modeling and simulation tool does directly support the specification of a subset of UML state machines [6]. The subset is rather restricted as they do not support orthogonal states. Orthogonal states are an important feature as they enable the modeling of parallel activities and, thus, can greatly reduce the model complexity. They also provide no mechanism like submachines to group a state machine into multiple parts and to reduce the visual complexity and to reuse once defined state machines. Furthermore, they do not support asynchronous message exchange.

7. Conclusion and Future Work

In this paper, we have presented an extension to ModelicaML for the specification of message exchange. We have illustrated our extension by a rainwater two tanks system using two controllers which exchange messages for their coordination. Furthermore, we presented how we translate the ModelicaML models to plain Modelica code and a helper C-function and finally, showed a simulation run. Our translation to Modelica code has been implemented as

plugins for the eclipse case tool Papyrus[5]. We are working on finishing the code generation with respect to messages.

Currently, we work on a new version of our translation directly to the StateGraph2 library. We want to combine StateGraph2 with algorithmic code for constructs which depend on a particular order of execution. Translating to StateGraph2 has the advantage that the resulting models are structured similarly to the ModelicaML state machines and are, thus, easier to understand by the developer than the generated Modelica code. Though, we have to extend the StateGraph2 library to support message sending and receiving for that translation.

Furthermore, we want to add more syntactical constructs to ModelicaML state machines for the better specification of temporal behavior. Specifically, clocks, time guards and invariants as used in timed automata will be added to ModelicaML state machines. Finally, we want to translate simulation runs done in a Modelica tool back to ModelicaML. For example, the states and transitions which are taken during a simulation run can be appropriately visualized using sequence diagrams.

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'.

Matthias Tichy is currently on leave from the Software Engineering Group at the University of Paderborn.

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Steffen Becker, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann, Claudia Priesterjahn, Wilhelm Schäfer, Oliver Sudmann, and Matthias Tichy. Mechatronicuml - syntax and semantics. Technical Report tr-ri-11-325, Software Engineering Group, University of Paderborn, Heinz Nixdorf Institute, 2011.

[3] Sven Burmester, Holger Giese, Stefan Henkler, Martin Hirsch, Matthias Tichy, Alfonso Gambuzza, Eckehard Münch, and Henner Vöcking. Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camel-view. In *Proceedings of ICSE*, pages 801–804, 2007.

[4] Sven Burmester, Holger Giese, and Matthias Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronic uml. In *Proceedings of MDAFA*, pages 47–61, 2004.

[5] Kevin C. Craig. Mechatronic system design. In *Proceedings of the Motor, Drive & Automation Systems Conference*, 2009.

[6] Ulrich Donath, Jürgen Haufe, Torsten Blochwitz, and Thomas 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.

[7] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 1st edition, 2004.

[8] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proceedings of 12th ACM SIGSOFT FSE*, pages 179–188, 2004.

[9] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the compositional verification of real-time uml designs. In *Proceedings of 9th ESEC and 11th ACM SIGSOFT FSE*, pages 38–47. ACM Press, 2003.

[10] Martin Otter, Hilding Elmqvist, and Sven E. Mattsson. Hybrid modeling in Modelica based on the synchronous data flow principle. In *Proceedings of CACSD*, pages 151–157, 1999.

[11] Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven E. Mattsson, and Charlotta Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. In *Proceedings of the 7th Modelica Conference*, pages 364–377, 2009.

[12] Uwe Pohlmann. A uml based modeling language with operational semantics defined by modelica. Master thesis, University of Paderborn, Department of Computer Science, Software Engineering Group, 2010. Master Thesis.

[13] Adrian Pop, David Akhlevidiani, and Peter Fritzson. Integrated UML and modelica system modeling with ModelicaML in Eclipse. In *Proceedings of the 11th IASTED*, 2007.

[14] Adrian Pop, David Akhlevidiani, and Peter Fritzson. Towards unified system modeling with the ModelicaML UML profile. In *Proceedings of EOOLT*, pages 13–24, 2007.

[15] Victorino Sanz, Alfonso Urquia, and Sebastian Dormido. Introducing messages in modelica for facilitating discrete-event system modeling. In *Proceedings of EOOLT*, pages 83–93, 2008.

[16] Wladimir 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.

[17] Wladimir Schamai, Uwe Pohlmann, Peter Fritzson, Christiaan J.J. Paredis, Philipp Helle, and Carsten Strobel. Execution of uml state machines using modelica. In *Proceedings of EOOLT*, pages 1–10, 2010.

[18] Rajarishi Sinha, Vei-Chung Liang, Student Member, Christiaan J. J. Paredis, and Pradeep K. Khosla. Modeling and simulation methods for design of engineering systems. *Journal of Computing and Information Science in Engineering*, 1:84–91, 2001.

[19] Object Management Group UML. Unified modeling language, superstructure, v2.2. Technical report, 2009.

[5] `http://www.openmodelica.org/index.php/developer/tools/134`