A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment[1]

Ulrich Donath Jürgen Haufe
Fraunhofer-Institute for Integrated Circuits, Design Automation Division
Zeunerstraße 38, 01069 Dresden, Germany
ulrich.donath@eas.iis.fraunhofer.de juergen.haufe@eas.iss.fraunhofer.de

Torsten Blochwitz Thomas Neidhold
ITI GmbH
Webergasse 1, 01067 Dresden, Germany
torsten.blochwitz@iti.de thomas.neidhold@iti.de

Abstract

The paper presents the use of a subset of UML Statecharts to model discrete control components together with the physical model within a Modelica simulation environment. In addition, we show how statecharts can also be used to describe assertions charts for checking the compliance of user defined model properties and model behaviour during simulation. As the main difference to other approaches, neither Modelica language enhancements nor special libraries are necessary. The statechart model is automatically mapped onto standard Modelica constructs and can be simulated with any common Modelica standard simulator. Controlled by the user, the Modelica model can be automatically instrumented by additional Modelica code to examine the state coverage and transition coverage during simulation.

Keywords: state machine; statechart; control system, assertions, state coverage, transition coverage

1 Introduction

The modelling of discrete and hybrid control algorithms [1] is not a novel application area for Modelica. In the last years, Modelica libraries for Petri Nets [2] [4], Statecharts [3] or StateGraph [5] were introduced. Furthermore, the extension of Modelica with a new statechart section is discussed in [6].

In this paper, we present a new approach for modeling and verification of discrete control components within a Modelica environment. In contrast to the solutions mentioned above, we create the control component models of the physical system outside Modelica. The other modules of the physical system are modeled as usual in the Modelica environment. In a second step, Modelica standard code is generated for the control components automatically. The insertion of the generated code into the Modelica physical model completes the system model (Fig. 1).

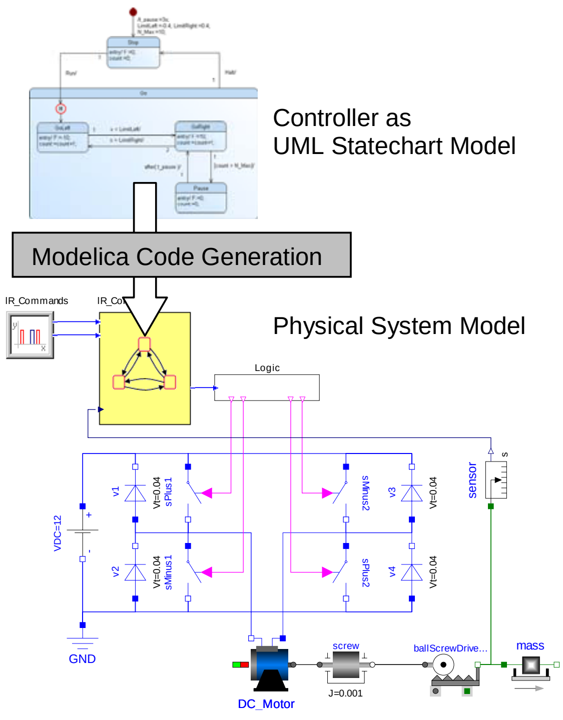


Fig. 1: Using UML Statecharts in SimulationX [9]

As modeling language for the control components we use a subset of UML Statecharts [7]. We derived the subset from an analysis of typical control algorithms in the domains mechanical and automotive engineering.

Besides control components, UML Statecharts proves to be suitable for robust modeling of physical effects or technical sub-systems with discrete states (friction, hysteresis, valves, switches, etc.).

For our approach we see following advantages:

- UML Statecharts are well established for modeling of control algorithms, especially for reactive systems.
- The statechart creation outside Modelica allows the use of off-the-shelf UML development tools.
- The approach provides not only an interface to Modelica. The approach is also open to interface specialized verification tools esp. formal verification tools.
- In the sense of model based design, the approach is expandable for generation of production code for different targets such as PLCs or embedded controllers.
- The generated Modelica code can be simulated with any common Modelica simulator.

For UML Statechart entry, an additional Graphical User Interface (GUI) containing a UML Statechart editor is necessary which comes usually with the UML development tool. The UML tool should meet following requirements:

- The GUI as well as the UML tool code generator needs the ability to be customized.
- The UML tool should support the interaction between GUI and generated code to establish a comfortable visualization and animation.

In section 5 we present an UML Statechart editor which is completely integrated into the GUI of SimulationX [9].

The paper is organized as follows. Section 2 gives an overview on the supported UML Statechart subset. In section 3 some techniques are introduced which allow an efficient verification of the statechart models. Section 4 presents a prototypic implementation of our approach. An outlook on future work is given in section 5.

2 UML Statechart Subset

In this section we present the subset of UML Statecharts which is implemented in our prototype (see section 4). The subset contains the minimum of UML Statechart constructs to model a control component in a comfortable way:

- States: Simple States, Non-Concurrent Composite States, Pseudo States.
- Transitions: Signal Triggers, Change Triggers, Time Triggers, Guards.
- Activities: Modelica text.

The UML Statechart subset as well as the resulting Modelica code is illustrated with a linear drive as an example.

2.1 Example

The linear drive (Fig. 2) is controlled by the Controller module. Inputs for the Controller are the operator commands Run and Halt as well as the position x of the linear drive. As output, the controller delivers the DC motor supply voltage U=-10V for left run, U=+10V for right run and U=0V for stop.

The specification of the controller is such as follows:

- Start after Run is given and drive to left
- Run 10 times between left and right end position
- Pause 3 seconds, afterwards continue
- Stop immediately after command Halt was given
- Restart with the action which was suspended after Halt, when Run is given again.

The physical system model of the linear drive example depicted in Fig. 2 is a simplification of the more complex model shown in Fig. 1.

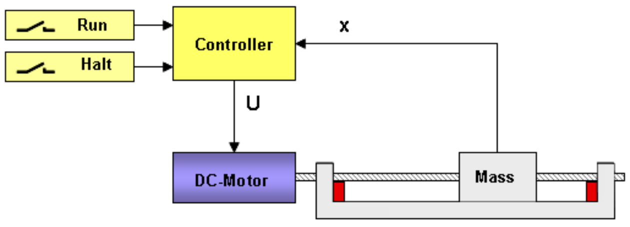


Fig. 2: Over-all structure of the linear drive

2.2 States

The control program (Fig. 3) is divided into the states Stop and Go. Stop is a simple state, whereas Go is a composite state with the nested simple states GoLeft, GoRight, and Pause. In consideration of hierarchy, the graph with the states Stop and Go is the top-level graph, implicit denoted as Main. The subjacent graph comprises the sub-states of Go.

For Modelica representation of state activities and state transitions, a state variable is declared for each hierarchy level. Their type declarations contain the enumerations of the state names. Each composite state is added with the enumeration InActive to indicate the inactivity of the composite state.

2.2.1 Simple States

In our subset, simple states may optionally have entry-activities, exit-activities, and activities which are initiated by internal transitions. These activities are simple Modelica algorithms. Modelica when-clauses are not allowed here.

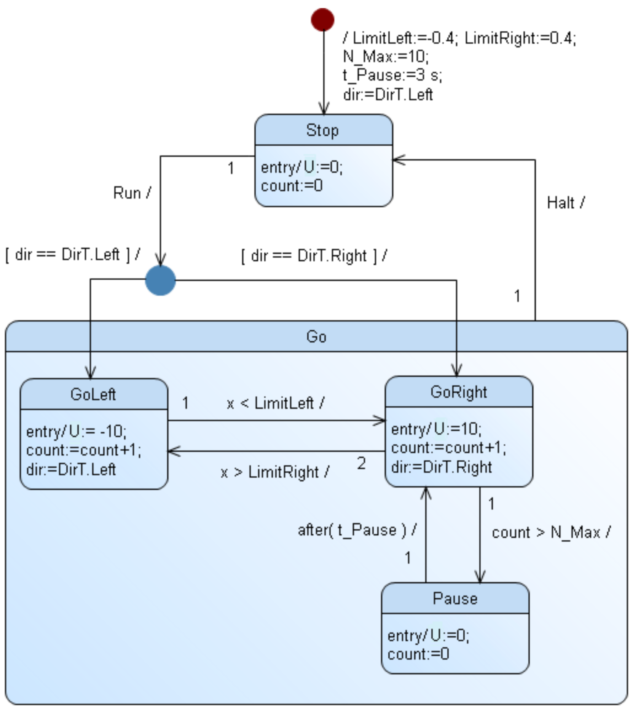


Fig. 3: Statechart model of the control program

2.2.2 Composite States

In comparison to simple states, composite states are extended each with a composition compartment. In our approach, this compartment comprises only one region of sub-states – this means, concurrency of activities can not occur within one statechart instance. Concurrency is only possible between multiple statechart instances.

In the generated Modelica code, all entry-activities of the nested sub-states are gathered in a when-clause separately (see 2.4 Entry-activities of Go). All other activities are included in an if-clause which describes the transitions of the composite state.

2.2.3 Pseudo States

We support following pseudo states: initial state, junction, and shallow history. An initial state indicates the default starting point of processing the statechart or a composite state. A junction merges multiple incoming transitions into a single outgoing transition, or conversely, split an incoming transition into multiple outgoing transitions. A shallow history stores the most recent active sub-state of a composite state after leaving it. When the composite state is newly entered via shallow history this sub-state becomes active again.

2.3 Transitions

Following kinds of transitions may be used: simple transitions (connecting two states), self-transitions (the same state acts as both the source and the destination), compound transitions (connecting many states via junction pseudo states), group transitions (originating from composite states), and internal transitions of simple states. A trigger, a guard, and a transition-activity may label a transition.

The triggering of a group transition implies the exiting of all the sub-states of the composite state and executing their exit-activities starting with the innermost states. An internal transition executes without exiting or re-entering the state in which it is defined.

In our approach, each transition is triggered with a single trigger as described below.

2.3.1 Signal Trigger

Generally, a signal trigger represents the receipt of an asynchronous signal instance [7]. In our interpretation, a signal is either a record typed message with e.g. one integer and real component or a boolean typed variable, typically a controller input command. Every new signal is notified by toggling a flag which is an additional component of the message. In case of a boolean variable, Run and Halt in the example, the variable itself is toggled.

Signals are produced either by modules of the physical system or inside the statechart instance.

Signal type definition:

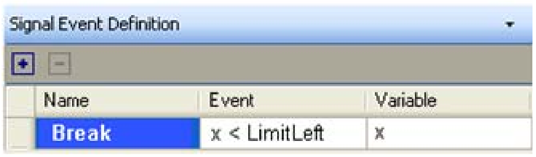
```
type SignalT = record SIGNAL
   Boolean  flag;
   Integer  int_val;
   Real     real_val;
end SIGNAL;
```

Signal assignment in physical system module:

```
SignalT Run;
when ( time >= 1 ) then
  Run.flag := not Run.flag;
  Run.int_val := 100;
end when;
```

A toggled signal is detected by the Modelica change-function, for instance *change (Run.flag)*.

In addition to UML, signals can be also defined in a Signal Definition Table (Fig. 4). In the table, signal events are derived from the achievements of predefined thresholds of physical system quantities or of internal statechart variables.


Signal Event Definition

Name	Event	Variable
Break	x < LimitLeft	x

Fig. 4: Example of the Signal Definition Table

2.3.2 Change Trigger

A change trigger specifies an event that occurs when a boolean-valued expression becomes true as a result of a change in value of one or more attributes [7]. In UML the life time of the change event is a semantic variation point. Related to control tasks, in our approach the change event remains true as long as the evaluation of the change expression results in true. In our linear drive example, change triggers are x < LimitLeft, x > LimitRight, and count > N_Max.

The Modelica representation of this behavior is given by an if-clause in the transition block (see 2.4 Transitions of Go).

2.3.3 Time Trigger

A time trigger specifies a time event, which models the expiration of a specific deadline [7]. We restrict the deadline to a relative expression. The expression is relative to the time of entry into the source state of the transition triggered by the event, e.g. after (t_Pause). The time event is generated only if the state machine is still in that state when the deadline expires.

In Modelica this behavior is reflected in following steps: Firstly, if the source state entry is detected in a when-clause, a time variable is set to the time limit (see 2.4 Entry-activities of Go). Secondly, a when-clause checks if the simulation time exceeds the time limit. If true, a timeout signal is toggled. This when-clause belongs to the event generation block of the module (see 2.4 Event generations). Thirdly, if the source state will inactive due to another transition, the time variable is reset.

2.3.4 Guards

A guard is a Boolean expression written in terms of parameters of the triggering event or attributes of the context object [7]. It is evaluated only once whenever it's associated event fires. If it is false, then the transition does not fire and the event is lost. In the linear drive example, the guards [dir==DirT.Left], [dir==DirT.Right] determine the target sub-states of Go after the Run command is given.

2.3.5 Firing Priorities

It is possible that more than one transition could be concurrently fired to change the state, e.g., they have the same trigger event and their guard expressions results in true. Then, in UML, an implicit priority rule is applied based on the relative position of the source state in the state hierarchy [7]. In addition, we allow the user to assign the transition priorities explicitly. A transition priority is denoted by a number 1, 2, 3… where 1 symbolizes the highest priority. These priorities are depicted near the start points of the transition arrow lines. Chosen by the user, the priorities of group transitions are either higher or lower than priorities of inner transitions of composite states.

The resulting priority number determines the position of the transition in the check for firing. In the linear drive example, Halt shall have the highest priority to stop the machine, especially in case of an emergency.

2.4 Over-all Modelica Code Structure

The Modelica representation of a statechart consists of following sections:

- Declaration of state variables, input/output signals, internal signals, system variables, auxiliary variables, parameters.
- Initialization of state variables and auxiliary variables, execution of initial transition activities (when-clause).
- Event generation block: generation of signal events according to signal event definition table, generation of timeout events (when-clauses).
- Entry-activity block: detection of state entries, execution of entry-activities, assignment of time limits to time variables, generation of completion events of composite states (when-clauses).
- Transition block: event detection, assignment of next state, execution of exit-activities and transition-activities, reset of time variables (if-clauses).

For the linear drive example the Modelica code is given below. To shorten, the declaration section is omitted.

```
Module Controller
```

Initialization:

```
when initial()then
   mainState:=MainStateT.Stop;
   ontoMainState:=MainStateT.Stop;
   goState:=GoStateT.InActive;
   entryGoState:=GoStateT.InActive;
   LimitLeft:=-0.4; LimitRight:=0.4;
   t_Pause:=3;
   t_PauseFinished:=0;
   count:=0; N_Max:=10;
   dir:=DirT.Left;
   timeout:=false;
   completeGo:=false;
end when;
```

Event generations:

```
when (time>=t_PauseFinished) then
   timeout:=not timeout;
end when;
```

Entry-activities of Main:

```
when (mainState==MainStateT.Stop) then
   U:=0; count:=0;
elsewhen (mainState==MainStateT.Go) then
   goState:=entryGoState;
end when;
```

Entry-activities of Go:

```
when (goState==GoStateT.GoLeft) then
   U:=-10; count:=count+1;
   dir:=DirT.Left;
elsewhen (goState==GoStateT.GoRight) then
   U:=10; count:=count+1;
   dir:=DirT.Right;
elsewhen (goState==GoStateT.Pause) then
   U:=0; count:=0;
   t_PauseFinished:=time+t_Pause;
elsewhen (goState==GoStateT.InActive) then
   completeGo:=not completeGo;
end when;
```

Transitions of Main:

```
if (pre(mainState)==MainStateT.Stop) then
   if (change(Run)) then
      if (dir==DirT.Left) then
         mainState:= MainStateT.Go;
         entryGoState:=GoStateT.GoLeft;
      elseif (dir==DirT.Right) then
         mainState:= MainStateT.Go;
         entryGoState:=GoStateT.GoRight;
      end if;
   end if;
elseif(pre(mainState)==MainStateT.Go)then
   if (change(completeGo)) then
      mainState:=ontoMainState;
   end if;
end if;
```

Transitions of Go:

```
if (pre(goState)==GoStateT.GoLeft) then
   if (change(Halt)) then
      goState:=GoStateT.InActive;
      ontoMainState:=MainStateT.Stop;
   elseif (x<LimitLeft) then
      goState:=GoStateT.GoRight;
   end if;
elseif(pre(goState)==GoStateT.GoRight)then
   if (change(Halt)) then
      goState:=GoStateT.InActive;
      ontoMainState:=MainStateT.Stop;
   elseif (count>N_Max) then
      goState:=GoStateT.Pause;
   elseif (x>LimitRight) then
      goState:=GoStateT.GoLeft;
   end if;
elseif (pre(goState)==GoStateT.Pause)then
   if (change(Halt)) then
      t_PauseFinished:=time;
      goState:=GoStateT.InActive;
      ontoMainState:=MainStateT.Stop;
   elseif (change(timeout) then
      goState:=GoStateT.GoRight;
   end if;
end if;
```

```
end Controller;
```

3 Verification

The main tool for the verification of the Modelica model is the simulator. In this section we describe techniques to increase the efficiency of the simulation based verification: Design Rule Check, State Coverage Analysis and Transition Coverage Analysis, and Assertion Charts.

3.1 Design Rule Check

During graphical entry and compilation of statecharts the following design rules are currently checked:

- Only one initial state is allowed on each hierarchy level.
- An initial state has exactly one outgoing transition. Trigger and guards are not allowed.
- Pseudo states must not connected by transitions.
- A split junction has only one incoming transition. Only this transition has a trigger.
- A merge junction has only one outgoing transition. Only this transition has a trigger.
- Self-transitions are not allowed for composite states.
- Each state, except initial state, has at least one incoming transition.

- Isolated sub-graphs are not allowed.
- A warning is given when some outgoing transitions of one state have the same priority number.

3.2 State and Transition Coverage

The analysis of both state activations and transition activations are measures for the achieved functional coverage during the simulation run.

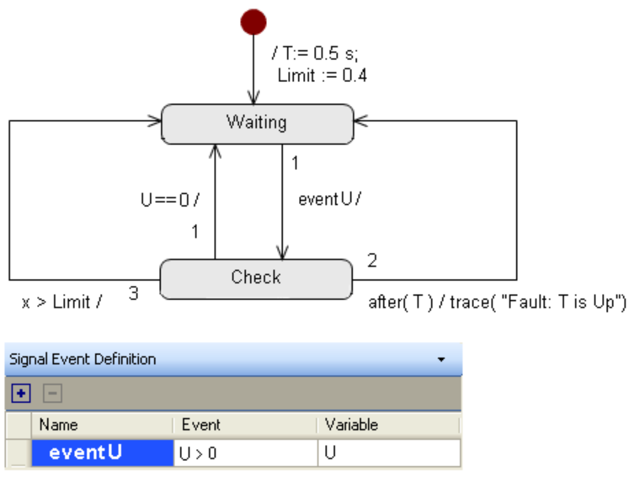
To measure the activation of states and transitions, additional Modelica code is automatically inserted into generated Modelica code (described in section 2.4) by the Modelica code generator (see section 4.2). The additional code basically consists of a set of counters. A unique counter is associated with any entry-activity block and any transition block. The counters are implemented as integer vectors. Every state or transition activation leads to an increment of the associated counter component.

For comfortable use, the actual achieved counting results may continuously back-annotated into the UML Statechart schematic.

3.3 Assertion Charts

Our Assertion Chart approach is derived from [10]. Assertion Charts aim for watching the behavior of the system in respect of specified properties. With the help of Assertion Charts, we describe both non-temporal and temporal system properties which have to be examined.

For the linear drive, e.g., an assertion may be: If the DC motor supply voltage U alters to U > 0, then drive position x will exceed its limit within T seconds, otherwise a fault will be reported (Fig. 5).



Name	Event	Variable
eventU	U > 0	U

Fig. 5: Assertion Chart example of the linear drive

In our case, the charts are composed like ordinary statecharts and run simultaneously to them. We deliver a set of typical Assertion Charts concerning dedicated event sequences and time limits. The predefined Assertion Charts are parameterized.

4 Implementation Prototype

The section introduces the first implementation of our approach into the SimulationX Modelica environment [8][9]. The implementation consists of the components Statechart Editor, Modelica Code Generator, Run Time Visualization (simulation driven statechart animation, association of the State Coverage and Transition Coverage Analysis results). All additional functionality can be controlled by the user within the SimulationX GUI. For easy use, a set of typical control domain Assertion Charts comes with the implementation.

4.1 Statechart Editor

The Statechart Editor (Fig. 6) is seamlessly integrated into the SimulationX framework. The Statechart Editor has the following features:

- Schematic entry of the statecharts,
- Definition of statechart activities as Modelica code,
- Definition of triggers,
- Definition of local types, variables and parameters,
- Definition of the interface to the physical system.

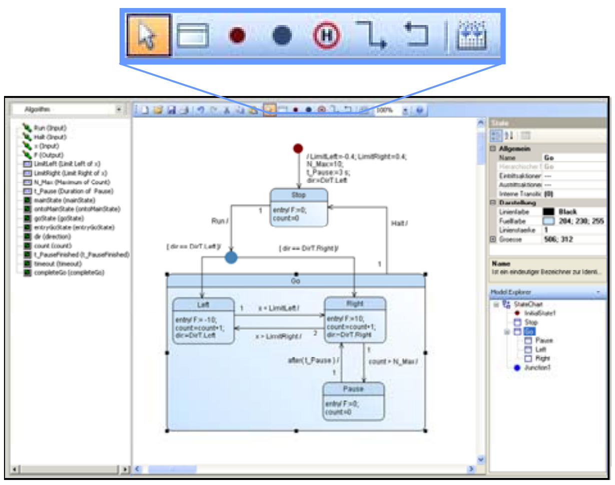


Fig. 6: Statechart Editor window

Pressing the compile button starts the Modelica Code Generator. The structure of code is outlined in section 2.4.

4.2 Modelica Code Generator

The Modelica Code Generator generates standard Modelica code from the UML Statechart model. The code is user-driven instrumented with additional code to perform the State Coverage Analysis and Transition Coverage Analysis and to associate Assertion Charts with selected signals. In addition, the Modelica Code Generator accomplishes the Design Rule Check. Besides Modelica code the generator architecture is open to generate code for various targets such as formal verification tools or production code for PLCs or embedded controllers.

4.3 Runtime Visualization

SimulationX supports the visualization of statecharts at runtime by a specialized view. During simulation the active state and the latest transition are highlighted if a time step or time interval is completed (Fig. 7). Controlled by the user, state and transition coverage are also displayed. The actual state and transition counter readings are annotated on the states and on the end points of transitions, respectively.

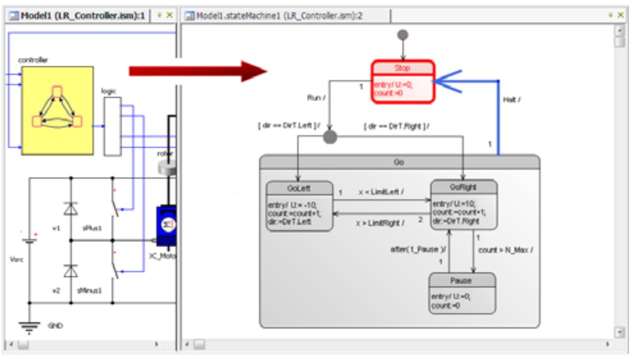


Fig. 7: Back-annotation of active states

For checking the functionality of whole system, the common view of internal controller variables and variables of the equipment are essential. Therefore all variables of the statechart, inclusive the state variables, may be accessed by the user.

In case of the linear drive example the causality of the system behavior (Fig. 8) is reflected by the signals Run and Halt of the operator, the variables goState and count of the controller, its output U, and the position x of the drive. The correlations between these quantities may be checked by assertion charts during the simulation.

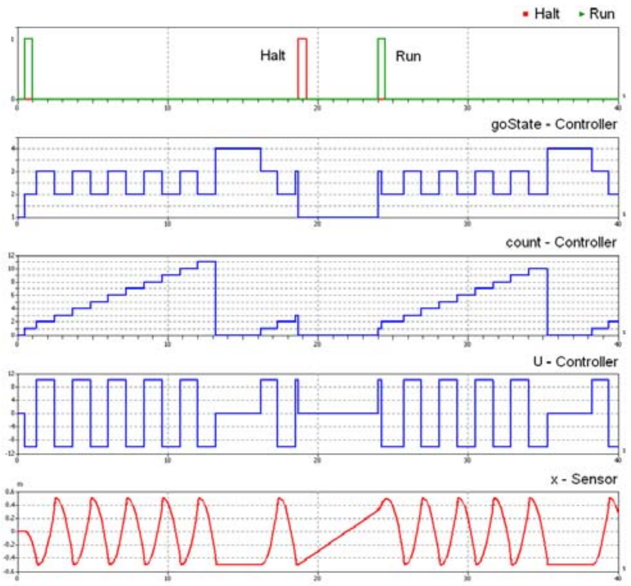


Fig. 8: Wave-forms of the linear drive example

5 Conclusions

We have applied our methodology to control systems in the area of automotive, robotics and manufacturing systems engineering. The approach introduced in the paper proves to be very comfortable for modeling of control components and physical system within the SimulationX environment. Especially the integrated verification support decreases modeling and simulation cycles and leads to robust control components with high test coverage in relative short time. Our next steps are the successive extension of the approach to a Modelica model based design environment for control algorithms.

Next steps are:

- Enhancements in code generation

  Right now, we generate Modelica code to validate the control component behavior within the physical system. For the implementation of the control component models, target code has to be derived from the models according to the target hardware platform, e.g. for PLC systems or embedded systems.

- Enhancements in formal verification

  In the recent years, considerable progress has been achieved in the field of formal verification tools for model checking especially for digital integrated circuit verification. Therefore, we are encouraged to interface our modeling approach to selected model checker tools by transformation of our statechart models to the model checker input description.

References

[1] Donath, U.; Haufe, J.; Schwarz, P.: Mehr-Ebenen-Simulation automatisierungstechnischer Prozesse und Steuerungen (in German). 4. Fachtagung Entwurf komplexer Automatisierungssysteme, Braunschweig, pp. 543-553, June 7-9, 1995

[2] Mosterman, P.J., Otter, M., Elmqvist, H.: Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica. Summer Computer Simulation Conference '98, Reno, Nevada, USA, pp. 314-319, July 19-22, 1998

[3] Ferreira, J.A., de Oliveira, J.E.: Modeling Hybrid Systems Using Statecharts and Modelica. 7th IEEE International Conference on Emerging Technologies and Factory Automation, Barcelona, Spain, 18-21 Oct., 1999.

[4] Fabricius, St.: Extensions to the Petri Net Library in Modelica. http://www.modelica.org, 2001

[5] Otter, M.; Årzén, K.-E.; Dressler, I.: StateGraph - A Modelica Library for Hierarchical State Machines. Proc. 4th Int. Modelica Conf., Hamburg, Germany, March 7-8, 2005, pp. 569-578.

[6] Nytsch-Geusen, Ch: The use of the UML within the modeling process of Modelica-models. Proc. ECOOP, Berlin, Germany, July 30, 2007.

[7] UML Superstructure Specification, v2.0. http://www.omg.org

[8] Neidhold, Th. et al: Modeling of State Machines in SimulationX. 10th ITI Simulation Workshop, Dresden, Germany, September 20-21, 2007.

[9] http://www.simulationx.com

[10] Drusinsky, D: Modeling and Verification Using UML Statecharts. Elsevier Inc., Oxford, 2006

[1] The project was founded by the European Regional Development Fund (ERDF) and by the Free State of Saxony in the framework of technology promotion.