MODELLING HYBRID SYSTEMS USING STATECHARTS AND MODELICA

J. A. Ferreira* and J. P. Estima de Oliveira**

* Department of Mechanical Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL)
T. +351 34 370892 F. +351 34 370953 E-mail: jaff@mec.ua.pt

** Department of Electronic Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL)
T. +351 34 370500 F. +351 34 370545 E-mail: jeo@inesca.inesca.pt

Abstract - *The present work evaluates the possibility of using the formalism of Statecharts, with hybrid features, as a graphical support to describe the dynamic behaviour of complex systems in Modelica. Implementation issues, according to the Statecharts semantics, are discussed and the advantages of using an equation based language, to implement the state transitions and nested states, are pointed out. Two levels of Statecharts implementation are considered: statecharts library models and components models. Statecharts library models are responsible for capturing events related to the firing of transitions and to the activation and deactivation of states that must be performed when transitions are taken. At the component model level, the statechart is composed by instantiating the basic models provided by the statecharts library.*

Finally, the statecharts library is evaluated with an example of a hybrid system simulation.

1. INTRODUCTION

In this section, the formalism of Statecharts and the Modelica language will be introduced.

1.1. Statecharts

States and events are considered a rather natural way of describing the dynamic behaviour of complex systems. However, in a complex system there could be a great number of distinct system states that, organized with the flat fashion of the Finite State Machine (FSM) formalism, result in complicated state diagrams. The dependency and parallelism of actions of concurrent subsystems increase exponentially the number of states and transitions in the FSM model (Fig. 2). To overcome the above limitations of FMS, Harel [1] proposed Statecharts as a visual formalism for the specification and modelling of complex reactive systems, expanding FSM formalism with hierarchy, parallelism and broadcast communication.

Hierarchy is a well accepted approach for designing and managing complex designs. In statecharts, hierarchy is used to group sets of states together, allowing high level description and step-wise development. **Depth** is achieved by an OR decomposition of a conventional FSM state, while **orthogonality** is introduced by an AND decomposition. Parallelism inside a statechart is described by AND states, allowing modelling of concurrent activities in the same model through **orthogonal** states. These orthogonal states are all activated when an AND state is entered and are all deactivated when it is exited.

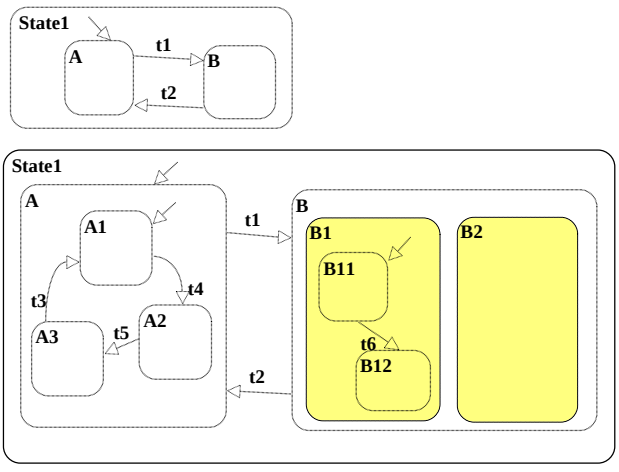

Fig. 1 - Refining states by AND/OR decomposition

Figure 1 shows the high level description of a system with two states A and B. These states can be refined through state decomposition. State A will be a compound OR state with substates A1, A2 and A3. State B will be a compound AND state with substates B1 and B2. This process can proceed until low level description is achieved. When orthogonal components are not truly independent, communication between them are specified by associating an action with a transition. This action is assumed to be broadcasted and so every system component in the statechart will recognize the message. This **broadcast communication** mechanism allows, when one part generates an event, that all the other parts sense it, acting in response if it is so specified.

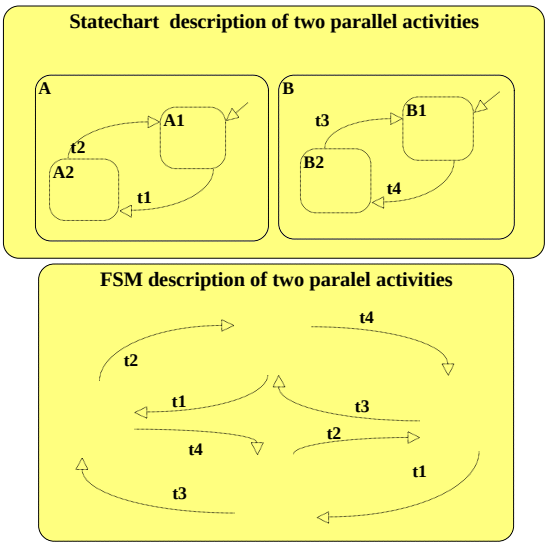

Fig. 2 – Description of parallel activities

Other enhancement over the FMS formalism is the association of an event **action** with a transition, when it is fired, or with a state, while the state is entered or exited. Continuous **activity** can also be associated with a state for modelling continuous behaviour when the state is active. This **action** and **activity** concepts allow the modelling of hybrid systems: actions capture the discrete systems' features while activities describe the continuous part. An example of an hybrid system modelled by an hybrid statechart is shown in figure 3.

This Mouse and Cat competition statechart example, based on the one presented in [2], can be described as follows. After a StartButton is pressed, a mouse starts running (at a constant velocity Vm) from a certain position on the floor following a straight line towards a small hole in the wall, at a distance X0 from the initial position. After a time Delay, a cat is released from the same initial position and chases the mouse at velocity Vc along the same path. The statechart execution can evaluate if the cat catch the mouse, or if the mouse finds the hole, while the cat crashes against the wall.

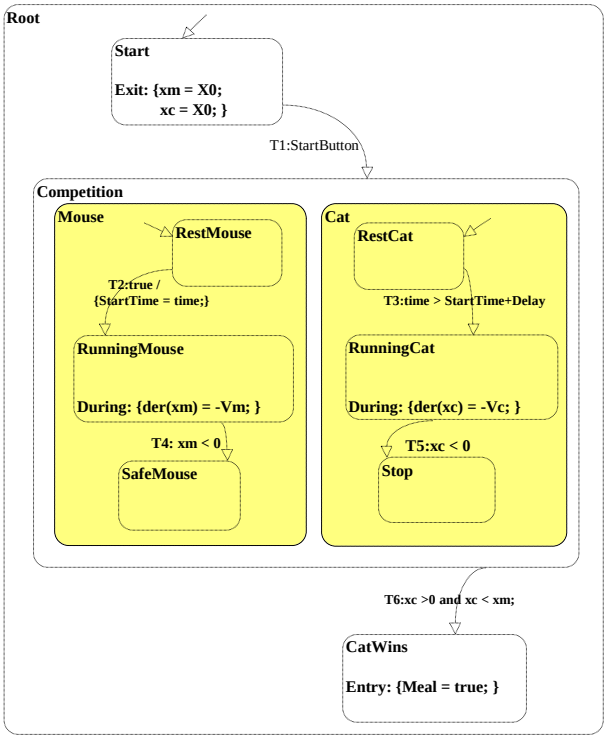

Fig. 3 - The Mouse and Cat competition statechart

1.2. Modelica

Modelica [3], a new object oriented language suitable for the modelling of complex systems from different domains, is being developed by an international effort with the main objective of making easy the exchange of models and model libraries. The language is built on non-causal modelling with algebraic and differential equations and uses object-oriented constructs to facilitate model reuse, through hierarchical modelling, encapsulation, and inheritance. Modelica was used to implement other formalisms such as Bond Graphs [4] or Petri-Nets [5] and has already proved its versatility as can be seen in [6], [7] or [8].

Modelica hybrid features allow modelling of discontinuities, discrete events or changes in structure in a very reliable way. Modelica is based on the object oriented modelling paradigm. The *class* concept (the basic structure element and the main building block of model description) has strong similarities to that of general purpose object oriented languages but also some differences: each *class* has a name and relates a set of components (variables and parameters) to equations or algorithms; there are several restricted classes with specific names, such as *model*, *type* (extension of built-in class *Real*, or of other defined types) or *connector* (does not have equations and can be used in connections).

A simple example will be used to introduce some of the Modelica features. Consider a tank with oil (to be used in an industrial process) that must be maintained in a

fixed level. The input proportional valve is used to introduce the oil into the tank. The valve is controlled with a simple discrete implementation of a Proportional-Integral (PI) controller.

Modelica have a restricted class called **connector** that can be used to define physical connections between component models. The idea is to have a model for the tank and a model for the controller and connect them with the following connectors.

```
connector ReadSignal
    Real val;
end ReadSignal;
```

```
connector ActSignal
    Real act;
end ActSignal;
```

Class definitions can use inheritance to reuse components of existing classes (by inclusion). The **extends** clause expresses inheritance in Modelica being multi inheritance possible through multiple **extends**. In order to develop reusable descriptions, Modelica allows the definition of partial models than can be inherited, for example as interfaces to other components. In this example, the model PIControler is defined as partial and will be inherited by the PIContinuousController and PIDiscreteController models.

```
partial model PIController
  parameter Real Ts = 0.1; // sampling time[s]
  parameter Real K = 2; // gain
  parameter Real T = 10; // time constant [s]
  parameter Real minV=0, maxV=1; // limits for output
  Real ref, error, outCtr;
  ReadSignal cInp;
  ActSignal cOut;
equation
  error = ref – cInp.val;
  cOut.act = LimitValue (minV, maxV, outCtr);
end PIController;
```

The PI controller is implemented with the following equations:

$$\frac{dx}{dt} = \frac{error}{T};$$

$$vPos = K \cdot (x + error);$$

where: **T** - controller time constant [s];

K - controller gain;

x - controller state [m];

error – level error [m]

vPos – limited valve position [m]

Either continuous and discrete versions for the controller can be defined. The discrete controller implements the state derivative with a finite difference quotient. The Modelica **pre** operator gives the value of a variable just before the event occur, and the discrete controller state only changes its value at event instants, being constant during continuous integration.

```
model PIContinuousController
  extends PIController (K=2, T =10, maxV = 0.02);
  Real x; // state variable of continuous controller
equation
  der(x) = error / T; //
  outCtr = K*(x + error);
end PIContinuousController;
```

```
model PIDiscreteController
  extends PIController (K=2, T =10, maxV = 0.02);
  discrete Real x; //state variable of discrete controller
equation
  when sample(0,Ts) then
    x = pre(x) + error * Ts / T; //
    outCtr = K*(x + error);
  end when;
end PIDiscreteController;
```

Modelica also allows a specialization of a class called function which has only inputs and outputs, one algorithm and no equations. The function LimitValue limits a value between a maximum and a minimum.

```
function LimitValue
    input Real pMin;
    input Real pMax;
    input Real p;
    output Real pLim;
algorithm
  pLim := if p > pMax then pMax
          else if p < pMin then pMin else p;
end LimitValue;
```

The modelling of hybrid systems in Modelica is supported *via* mixed continuous/discrete systems of equations [9]. Discontinuous models can be handled with **if-then-else** expressions, allowing modelling of phenomena with different expressions in various regions of operation. In this example, the tank output flow, qOut, is defined with different equations at different times:

```
qOut = if time > 100 then flowVout else 0;
```

Tank instances can be connected to controllers instances through its connectors. The mass balance equation supposes constant pressure and is defined in the tank model as a continuous equation. The input flow is related to the valve position by a flow gain.

```
model Tank
  ReadSignal tOut;
  ActSignal tInp;
  parameter Real flowVout = 0.01; // [m3/s]
  parameter Real area = 0.5; // [m2]
  parameter Real flowGain = 10; // [m2/s]
  Real h (start = 0); // tank level [m]
  Real qIn; // flow through input valve [m3/s]
  Real qOut; // flow through output valve [m3/s]
equation
  der(h) = (qIn - qOut) / area; // mass balance equation
  qOut = if time > 100 then flowVout else 0;
  // interface between discrete and continuous part
  qIn = flowGain * tInp.act;
  tOut.val = h;
end Tank;
```

```
model TankAplication
  PIDiscreteController PIDiscrete;
  Tank T(area= 1, flowVout = 0.02);
equation
  connect(T.tInp , PIDiscrete.cOut);
  connect(T.tOut , PIDiscrete.cInp);
  PIDiscrete.ref = 0.5;
end TankAplication;
```

Discrete event and discrete time models are supported by **when** statements. The equations in a **when** clause are conditionally activated at event instants where the **when** condition becomes true. For example, the controller equations are implemented in the **when** clause. The **sample**(0,Ts) function generates events at sample intervals, Ts. When such events occur the integration process is halted, the new limited value for valve position (controller output) is generated and then the continuous integration is restarted. Modelica considers that event handling takes zero time.

The tank application example is modelled with a continuous plant being controlled by a discrete controller, and this is done in the same model. This becomes possible because the Modelica compiler creates a flat hybrid Differential and Algebraic Equations (hybrid DAE) system. These equations result from the expansion of the inheritance tree, parameterization of classes and components and from the generation of connection equations from connect statements. This flat hybrid DAE should have the same number of variables and equations. This is also fulfilled by the discrete part, because it uses the synchronous data flow principle with the single assignment rule. This assumption leads to a deterministic behaviour and to an automatic synchronization of the discrete and continuous equations of the model.

2. IMPLEMENTATION ISSUES

The graphical syntax for Statecharts has been accorded quite early. However, the definition of formal semantics is not yet established [10]; in fact, being an unofficial language, Statecharts clearly has no official semantics. In accordance to the type of system to model, several variants in statecharts have been proposed [11]. Two components of Statecharts semantics can be considered: the static part and the dynamic (run time) part. Static semantics is the description of structural constraints that cannot be adequately captured by syntax descriptions, for example the interpretation and detection of invalid transitions. A list of rules to implement static semantics was introduced in [12]. Dynamic semantics deal with the run time evolution of a statechart.

The synchronous hypothesis [13] of Statecharts implementation is naturally fulfilled by the synchronous principle of hybrid systems provided by Modelica. Since all the equations are evaluated concurrently, and when an event occurs the continuous integration is halted, it is guaranteed that any actions, triggered by the statechart events, are always performed just before the continuous integration proceeds.

2.1 Handling of nested states with Modelica constructions

In hierarchical statecharts, activation and deactivation events must be propagated to substates. In Modelica, hierarchical statecharts are handled with auxiliary hierarchical connections. With this approach all the states are initialized in a similar way, and only the hierarchical connections have to be taken. When hierarchical activation/deactivation events occur they are handled by boolean variable equations, without the need of algorithms.

The hierarchical activation/deactivation events are propagated at the same time to all the states belonging to the hierarchy. The one-way propagation of events, ancestor states to descendent states, is handled with two hierarchical connectors (input, output) in each state, except for the root state.

2.2 Statecharts library in Modelica

In this work, Modelica was used to implement a subset of the Statecharts formalism features that are needed for hybrid systems modelling. Two Statecharts implementation levels are considered: Statecharts library models and component models. Statecharts library models are responsible for capturing events related to the firing of transitions and to the activation and deactivation of states (and substates) that must be performed when transitions are taken. Component models create the statechart, with the basic models provided by the Statecharts library, by making the state-transition-state connections, and define the transition events or describe the continuous activities within states.

This section presents some implementation details of the Statecharts library, that includes: models to connect transitions to states (**FireCut**, **SetCut**); models to represent the different states (**BasicStateS**, **OrthogonalStateS, StateS**, **RootStateS**); a model to represent hierarchical state to state connections (**HierarchicalConnection**); a model to represent transitions (**TransitionS**).

In the first place it is necessary to define models for the connectors state-transition-state. Some models and connectors were defined by Mosterman et al. in [5] when implementing Petri-Nets in Modelica.

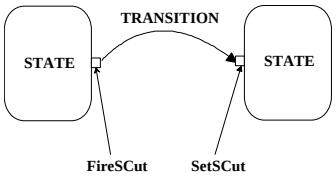

Fig. 4 - Connectors state-transition-state

```
connector FireSCut
  Boolean state "state of connected state (source state)";
  Boolean fire "True, if transition fires";
end FireSCut;
```

```
connector SetSCut
  Boolean fire "True, if transition fires";
end SetSCut;
```

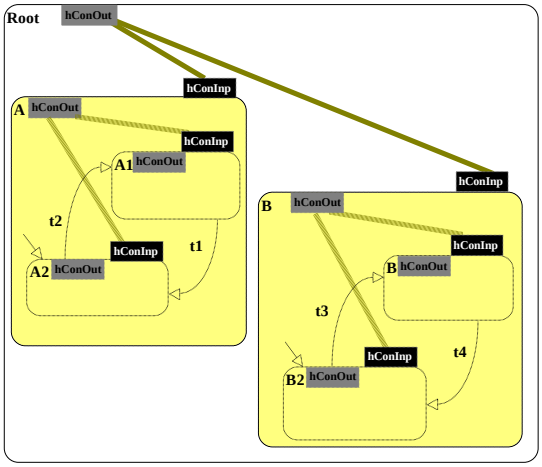

Fig. 4 – Hierarchical connections to propagate events

Hierarchical connectors propagate the activation and deactivation messages to nested states as can be seen in figure 5.

```
connector HierarchicalConnection
  Boolean act;  // signals an hierarchical activation
  Boolean deact; // signals an hierarchical deactivation
end HierarchicalConnection;
```

A transition is triggered by an **event**, it can have a **guard** condition, and an **action** can be taken when firing. These transition properties are handled by boolean variables. The transition fires if its source state is active and its guard transition is true when the trigger event occurs.

```
model TransitionS
  FireSCut inp; // connection of source state
  SetSCut out;  // connection of target state
  input Boolean eventT;    // trigger event
  input Boolean guardT;    // transition guard condition
  output Boolean actionT; // signals the transition action
equation
     out.set = inp.state and eventT and guardT; // firing
     inp.fire = out.set;
     actionT = out.set;
end TransitionS;
```

The common structure and equations for the models representing states is inherited from the abstract model **BasicStateS**. Each state has a hierarchical input and output connection from where it receives hierarchical activation and deactivation messages. The *active*, *activate* and *deactivate* boolean variables signal the status of the state.

```
model BasicStateS
  Boolean active;    // signals if the state is active;
  Boolean activate;       // signals that activation
  Boolean deactivate;   // signals state deactivation
  HierarchicalConnection hConInp;
  HierarchicalConnection hConOut;
…
equation
…
  active = (pre(active) or activate) and not deactivate;
end BasicStateS;
```

```
model StateS
  extends BasicStateS;
  parameter Boolean defaultState = false;
  parameter Boolean history = false; //if history is set this
  //state must memorize its last status just before deactivation
  constant Integer nSet = 1;// input transitions of this state
  constant Integer nFire = 1; // output transitions of this state
  Boolean lastState (start = defaultState);
     // memorizes the state status just before deactivation
  SetSCut  inp[nSet];          // connected input transitions
  FireSCut out[nFire];        // connected output transitions
  Boolean set, fire;
 …
equation
  set = AnyTrue(inp.set);
  fire = AnyTrue(out.fire);
  activate = if history then pre(set) or
                    (hConInp.act and lastState)
             else pre(set) or (hConInp.act and defaultState);
  deactivate = pre(fire) or hConInp.deact;
…
  when hConInp.deact then
    lastState = pre(active);
  end when;
end StateS;
```

Orthogonal states are "child" states of AND states and, in the present implementation, cannot have connected transitions, and so they only can be activated through hierarchical activation messages. The **RootStateS** detects the beginning and the end of the model execution in order to send hierarchical activation and deactivation messages to the statechart. These messages are then propagated by hierarchical connections.

```
model OrthogonalStateS
  extends BasicStateS;
equation
  activate   = hConInp.act;
  deactivate = hConInp.deact;
end  OrthogonalStateS;
```

```
model RootStateS
  HierarchicalConnection hConOut;
equation
  hConOut.act = if initial() then true else false;
   hConOut.deact = if terminal() then true else false;
end RootStateS;
```

2.3 Use of the Statecharts library at the component level: an example

At the component level, the statechart is built by instantiating the state and transition models with the appropriate parameters. Then, the transitions are connected to states, hierarchical connections are made to establish the hierarchical state relationship, and transition events and guard conditions are defined. Finally, appears the definition of all the event actions and activities that must be carried out by the component, such as the actions associated with transitions and states, and the activities to be performed in states. A subset of the code that implements the component model, for the mouse and cat competition statechart of figure 3, is presented shortly:

```
model MouseCatCompetition
  parameter Real Delay = 2;
  Real xc, xm;// State models
  RootStateS  Root;
  StateS Start (defaultState = true);
  StateS StopCat;
  TransitionS T1; // transitions models
..TransitionS T6;
  …
equation
  // transition to states connections
  connect(Start.out, T1.inp);
  …
  // hierarchical state connections
  connect(Root.hConOut,Start.hConInp);
  // Trigger events and Guard conditions for transitions
  T3.eventT = ( time > StartTime + Delay );
  T6.eventT = ( xc > 0 and xc < xm );
// actions for transitions: these are event actions
  when T2.actionT then
     StartTime = time;
  end when;
  // Entry and exit actions for states: these are event actions
  when Start.activate then
     reinit(xc , X0);
     reinit(xm , X0);
  end when;
  when CatWins.activate then
     Meal = true;
  end when;

  // During activities: continuous activity of states
  der (xm) = if RunningMouse.active then -Vm else 0;
  der (xc) = if RunningCat.active then -Vc else 0;

end MouseCatCompetition;
```

The above models were tested with the Dymola package [14], Dymosim simulator using the Lsodar integration algorithm, with different velocities and delays. Lsodar [15] is a variable order integration algorithm that uses a multiple-step method and a root finder to handle state events. A hybrid statechart should be solved by a method with a root finder because, after state events, consistent initial values for state variables have to be calculated. This is usually a task for iterative root finding schemes. However, this simple example was also simulated with single step methods, like Euler's and Runge-Kutta methods, with a step size of 0.01 seconds.

The graphical results presented below illustrate the dynamic behaviour evolution of the statechart during a time period (5 seconds) for the following configuration: Delay = 2 sec; Stop Time = 5 sec; Velocities (Vc = 5 m/s; Vm = 2 m/s). After the StartButton event, the mouse starts its run at a constant velocity (2 m/s). With a delay of 2 sec, the cat catch the mouse with a velocity of 5 m/s; the cat eats the mouse 1.4 sec after that. The start state is active until StartButton is pressed.

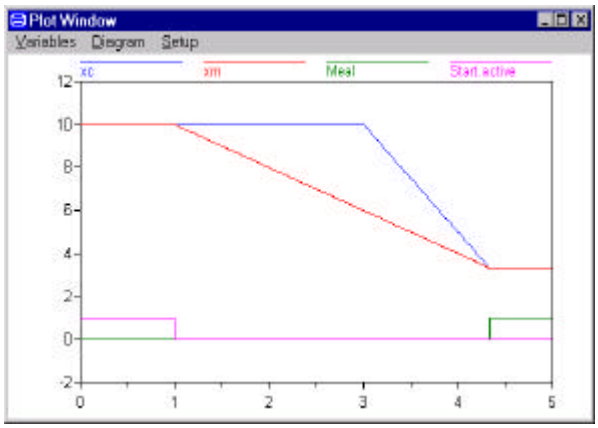

Fig. 6 – Plot of the simulation results using Dymola

3. CONCLUSIONS

The use of the Statecharts formalism to describe the dynamic behaviour of complex systems is growing in the recent years. The introduction of hybrid features, describing continuous activities within states, opens new perspectives of using Statecharts to model complex hybrid systems. The new modelling language, Modelica, was used to implement a subset of the Statecharts formalism features, enabling the description of the dynamics of hybrid systems through a well understood graphical formalism, that can be easily translated with a Modelica compiler. The developed library can also be used in a graphical environment with drag and drop features, allowing the composition of the statechart. The connections state-transition-state can be established by drawing lines through connectors. By inspecting the state rectangles, the hierarchical relationship between states can be pointed out. That way, the statechart can be automatically translated into Modelica code. The equation based modelling of the Modelica language, along with the connector constructions, has proved its efficiency by passing activation/deactivation messages instantaneously through nested states. Also, the broadcast communication mechanisms of Statecharts can easily be fulfilled just by setting the value of a variable, that is automatically transmitted to all the statechart components; in fact, the statechart is implemented with a differential algebraic equation system (DAE) that is evaluated concurrently.

In a near future, the objective is to enhance the Statecharts library with new features like inter-level and compound transitions or behaviour inheritance.

REFERENCES

[1] - D. Harel. Statecharts: A visual formalism for complex systems. In: Science of Computer Programming 8, 1987.

[2] - Y.Kesten, A.Pnueli. Timed and hybrid statecharts and their textual representation. In: J.Vytopil (ed.): Formal Techniques in Real-Time and Fault-Tolerant Systems. Springer-Verlag, 1992.

[3] - Modelica: Homepage: «http://www.modelica.org».

[4] - J. F. Broenink. Object-oriented modeling with bond graphs and Modelica. In: Proceedings of the International Conference on Bond Graph Modeling and Simulation, ICBGM '99 (part of WMC '99, the Western MultiConference), San Francisco, CA, January 17-20, 1999.

[5] - P. Mosterman, M.Otter, H. Elmqvist. Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica. In: proceedings of the Summer Computer Simulation Conference -98, Reno, Nevada, USA, July 19-22, 1998.

[6] - H. Tummescheit, J. Eborn. Design of a Thermo-Hydraulic Model Library in Modelica. In: Proceedings of the 12th European Simulation Multiconference, ESM'98, June 16-19, 1998, Manchester, UK.

[7] - M. Otter, C. Schlegel, H. Elmqvist. Modeling and Realtime Simulation of an Automatic Gearbox using Modelica. In: Proceedings of the European Simulation Symposium, ESS'97, Passau, Germany, October 19-22, 1997.

[8] - S. Mattsson. On Modeling of Heat Exchangers in Modelica. In: Proceedings of the European Simulation Symposium, ESS'97, Passau, Germany, October 19-22, 1997.

[9] - M. Otter, H. Elmqvist, S. E. Mattsson: Hybrid Modelling in Modelica based on the Synchronous Data Flow Principle. Accepted for the 1999 IEEE Symposium on Computer-Aided Control System design, CACSD'99, Hawaii, August 22-27, 1999.

[10] - D. Harel, A. Naamad. The STATEMATE semantics of Statecharts. In: ACM Transactions on Software Eng. Method. October 1996.

[11] - M. Beeck. A Comparison of Statecharts Variants. In: Lecture Notes in Computer Science 863, Springer, Berlin, 1994.

[12] - H. Hong, J. Kim, S. Cha, Y. Know. Static Semantics and Priority schemes for Statecharts. In: Proceedings of the Computer Software and Applications Conference '95, Texas USA, Aug. 1995.

[13] - A. Pnueli, M. Shalev. What is in a Step: On the semantics of Statecharts. In: Theoretical Aspects on Computer S, Volume 256 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1991.

[14] - Dymola package: «http://www.dynasim.se».

[15] - A. Hindmarsh. ODEPACK, a systemized collection of ODE solvers. Scientific Computing, R.S. Stepleman et. Al, Amsterdam, 1983.