Simulation and Visualization Support for User-defined Formalisms Using Meta-Modeling and Hierarchical Formalism Transformation

Manuel A. Pereira Remelhe

Process Control Lab (CT-AST), University of Dortmund,
44221 Dortmund (Germany), m.remelhe@ct.uni-dortmund.de

***Abstract*— A software environment which is dedicated to modeling and simulation of complex technological systems incorporating various domains has to cover a wide range of schematics and graphical formalisms. Usually, these formalisms are based on different computational models and include continuous as well as discrete-event dynamics. Furthermore, in order to improve the design process, an environment of that kind should provide means for adopting additional formalisms tailored to the needs of a specific domain.**

In this paper a hierarchical formalism refinement concept is presented. It allows to introduce efficiently user-defined formalisms along with simulation and analysis facilities into a modeling environment by deriving them from already defined formalisms. This concept has been realized using the modeling language Modelica as the basic model representation and the meta-modeling tool DOME for specifying the graphical editors and the formalism transformations. For the purpose of illustration a basic statechart tool is refined twice. First, a new tool is created which support compound transitions. Then, this new formalism is extended again by introducing differential equations.

***Keywords*— Hybrid System, Formalism Refinement, Meta-Modeling, Simulation, Statechart, Visualization**

I. INTRODUCTION

Design, analysis, and testing of complex dynamic systems, such as robots, aircraft, and multi-product batch plants, increasingly relies on modeling and simulation and requires the integration of different modeling and specification formalisms [1]. For example, a model of a chemical batch plant may consist of models for the thermodynamic and chemical behavior of the medium, a flowsheet of the plant components, a model of the electricity supply system, detailed actuator models comprising the electrical and mechanical behavior, etc. Besides these physical components a chemical plant includes also control functionality such as low-level digital PID type control, high-level supervisory control as well as communication protocols between decentralized logical controllers.

Usually, for simulation of physical system parts quantitative models are used which are based on differential and algebraic equations (DAE). Due to modeling abstractions and physical discontinuities, e.g., when the boiling point is reached, the models may contain some logical equations so that the resulting system of equations becomes a hybrid DAE [3]. In contrast, the simulation of digital parts of a system mostly relies on various specific discrete time computational models which are completely different from the continuous time simulation. Hybrid formalisms, such as hybrid automata, are often used for analysis (verification) purposes and in early design stages. Nevertheless it can be useful to be able to simulate them also, in order to assist the validation of hybrid models and to visualize critical runs.

In order to provide an useful modeling and simulation environment for improving the design process of complex technological systems it is indispensable to integrate at least a small set of widely used powerful formalisms with different computational models, e.g., statecharts, Petri-Nets, hybrid DAE, etc. Furthermore, a software environment of that kind should also be able to integrate additional user-defined formalisms including simulation and visualization facilities, since it would make no sense to force the designers to use other formalisms than those which they are familiar with and which have been tailored over decades to the specific needs of the application domain.

The editor of an additional formalism can be specified and generated automatically using meta-modeling techniques where the graphical appearance and the syntax of the formalism are defined using a graphical or textual meta-language. Normally, one would argue that the way a model has to be simulated depends the semantics of the used formalism, but it is also legitimate to say that the semantics of a specific implementation of a formalism is given by the way a corresponding model is simulated. In this respect, it is possible to define the semantics of a new formalism by specifying an unambiguous transformation into another formalism with established semantics. Consequently, simulation-support and all other utilities became automatically available for the new formalism as well. Introducing such transformation relations between several formalisms results into a so-called formalism transformation graph [8]. Such an environment can be extended easily with domain-specific formalisms at least when there is already a similar formalism.

In this paper a hierarchical formalism refinement concept is presented which allows to specify new formalisms efficiently by deriving them from already defined formalisms. Hierarchy means here that it is possible to derive further formalisms from previously refined formalisms so that a hierarchical formalism transformation/refinement graph can be build up. As a precondition a basic formalism has to be chosen which can be simulated and to which all other formalisms can be mapped in order to integrate the different models. Therefore, this basic formalism has to support continuous time integration of differential equations as well as discrete time simulation. The refinement is specified by (1) creating a new notation, i.e., ontology, syntax and appearance, based on an existing editor specification (2) specifying a transformation relation from the new notation to the existing one and (3) specifying a visualization mapping from the existing notation

to the new one for enabling specific visualization features for inspection of discrete event simulation results.

This concept has been realized using the Modelica [2,3,4] language as the basic model representation, which is advantageous in many aspects. On the one hand, the Modelica language is designed for modeling and simulation of physical systems based on hybrid DAEs. Due to Modelica's class concept and its declarative equation-based semantics it is possible to define domain-specific component libraries, so that all physical system's schematics can be represented directly in Modelica and are integrated automatically. On the other hand,

Modelica is powerful enough to represent complex discrete-event behavior due to the ability to define so-called algorithm-sections where complex iterations can be performed locally without evaluating all synchronous continuous equations, and due to the support of multiple events instants in one instant of time. This is because in physical systems a cascade of discontinuities may occur. Hence, it is intended to employ Modelica for the quantitatively modeling and simulation of hybrid physical systems, even if they include discontinuities, such as friction or impact.

Due to the declarative equation-based semantics of Modelica certain semantic aspects of complex discrete-event formalisms, e.g., statecharts, can not be reproduced within a Modelica component library. Therefore the meta-modeling tool DOME is used for modeling discrete event formalisms. DOME is designed for specifying graphical notations and generating the corresponding editors. A graphical language called DOME Tool Specification Language (DTSL) is used for specifying the graphical entities, its properties and relations, structural constraints and the visual appearance of the formalisms. More advanced features such as complex syntactical constraints or code generation can be implemented with DOME's Scheme- or Lisp-like extension language *Alter*.

This paper uses Alter for specifying transformations between different formalisms inside of DOME, i.e., it is possible to develop a model in one formalism and to view it in an editor of the underlying formalism after transformation. Thus, the transformation comprises the creation of a new model instance inside of DOME. If the formalism of this dependent model instance has an established semantics, simulation support and other utilities become available automatically to the original model as well. Another interesting concept is to use DOME's visual attributes for visualizing simulation runs directly within the graphical model. For this, additional properties have to be introduced into the formalism specification for storing the state of the model. A efficient way of organizing the feedback of simulation data is to specify for each formalism mapping a suitable formalism data transformation.

In order to illustrate these concepts a statechart [6] editor is regarded which is assumed to have already an interface to the Modelica simulator. First, the meta-specification of this editor is presented. In addition to the basic notation it considers a visualization notation so that the editor can also be used for animating simulation data. Then, it is shown how this statechart formalism can be extended with compound transitions. This is done in three steps: (1) creating an extended notation based on the existing statechart specification, (2) implementing Alter procedures for transforming extended statecharts into basic ones and (3) implement Alter procedures for transforming the visualized state of the basic model into the extended model. Finally, a third formalism including differential equations is build based on the extended statechart formalism.

II. The Meta-Model of the Basic Statechart Formalism

Statecharts were introduced by Harel [6] and have become very popular in the specification of embedded systems, e.g., in cars and airplanes. Statecharts can be regarded as an extension of state transition systems by the introduction of hierarchy (a state can contain substates which either all are active together or are exclusive), concurrency, broadcast communication (events may trigger transitions everywhere in the statechart) and histories (e.g. after handling an exception state, the system returns to the previous state or resumes in a well-defined intermediate state). They provide a compact and intuitive graphical representation of even very complex state transition diagrams. Many variants of statecharts have been developed and are supported by tools for different purposes. A well-known and rigorously defined variant is that of the commercial design tool Statemate [7]. There are three types of states in a statechart (see Fig. 1): OR-states (aDevice, processA, processB, and A1), AND-states (active) and basic states (all other states). Substates of an AND-state are concurrent states, so that if the statechart is in the state active, it is at the same time in the states processA and processB. Substates of an OR-state are exclusive, consequently if the statechart is in state A1, it is in exactly one of the substates, A2 or A3, as well. The default substate, in which the system goes when it enters an OR-state, is indicated by a default transition. Therefore the statechart assumes the state off when starting and then the states active, processA, processB, A1, A2, and B1 become active, when the event turnOn is registered. The optional parts of the transition labels are: "triggering events [conditions] / actions". By means of the history symbol H* the statechart is able to reenter the last active substates in the scope of processA , for example A1 and A3, when it has been in standBy mode and the event goActive occurs.

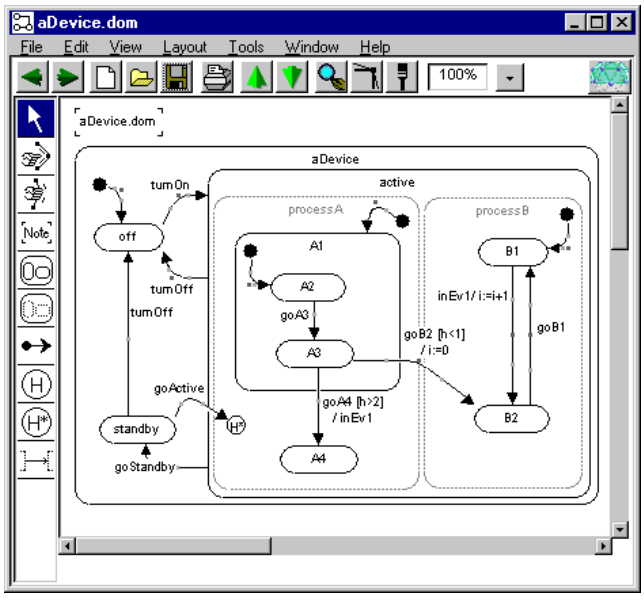

Figure 1: the generated statechart editor

Figure 2 shows a DOME Tool Specification (DTS) of the basic statechart formalism. It defines the classes of the modeling objects. AndState, OrState, History, DeepHistory and Default represent the node types of the formalism and are depicted with double border and rounded corners. The classes In-Out-Connector and State are generalizations of those classes and are denoted by a single border. They are only used for specifying common properties and constraints and can not be instantiated, i.e., they are not modeling objects. Transition is the edge type of the formalism which is indicated by the squared corners. Due to the arrows labeled with *Transition*, an In-Out-Connector object, i.e., an object of the class History, DeepHistory, OrState or AndState, may have incoming and outgoing connections of the type Transition, where as an object of the class Default must not have incoming Transitions. The Node Indicator (small triangle with an 'n') determines that only State objects, i.e., AndStates and OrStates, can be placed directly on the top level of the graph. All other nodes can only be inserted as an element of an OrState object which is expressed by the Element Indicator (small triangle with an 'e'). The labels at the upper connections of the Element Indicators shows the relationship name and, enclosed in brackets, the cardinality constraints. These declare that an OrState may not contain more than one History, one DeepHistory and one Default object and may embed many AndStates and OrStates. Note that an AndState object is not allowed to take up other AndState objects.

The appearance of the modeling objects can be defined easily by selecting from predefined options in the categories shape (rectangular, circular, polyline, ...) , name position, corners (square, rounded, chorded, ...), line thickness, border count, dash pattern and paint pattern. In addition, for each of these categories a preconceived Alter method exists, which can be used to change dynamically particular visual attributes

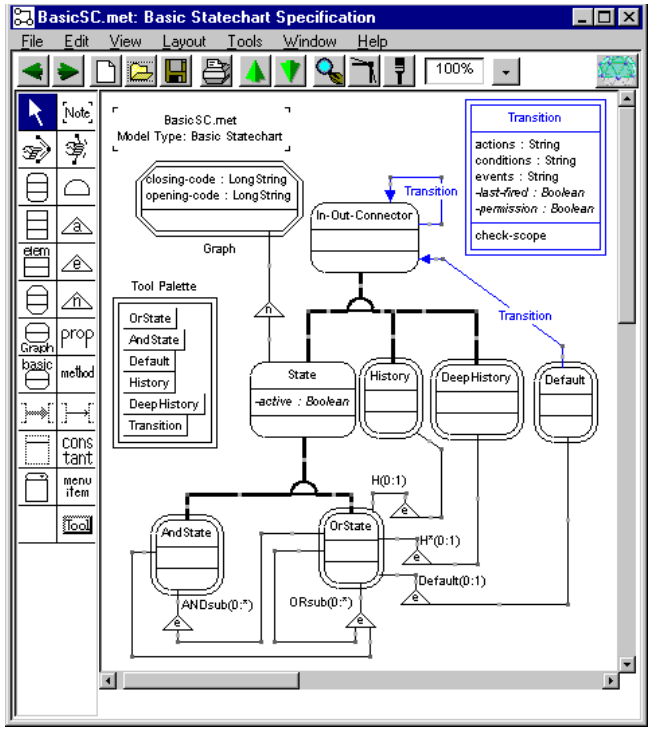

Figure 2: a DTS for the basic statechart formalism

of single objects. This is applied in the basic statechart formalism for changing the dash pattern and the paint pattern of OrStates depending on whether its container is an AndState or not (see figure 1). The implementation of the dash pattern method returns the symbol *dot* if the container of the OrState *self* is an AndState, else the symbol *normal*:

```
(if (is-a? (container self) BasicSCAndState)
'dot 'normal )
```

In order to support the transition labels Transition class contains the string properties event, condition and action (figure 2). By this, it is possible to input the corresponding attributes for each transition instance in a dialog (figure 3). For displaying the labels a preconceived method is implemented that concatenates appropriately the strings inserting the brackets and the slash if necessary.

Figure 3: transition attributes dialog

A Transition edge that originates from a Default node must not leave the scope of the container of the Default node, i.e., it has to lead to one of its direct or indirect substates. Because this syntax rule is too complex for specifying it with graphical constraints, the method check-scope was added to the Transition class. It determines via recursive search of the graph whether this syntax rule is fulfilled or not. If it is not fulfilled a warning is displayed.

A. *Transformation into Modelica*

On the one hand, Modelica is an understandable textual language which allows a hierarchical composition of complete models with a text editor, instantiating and connecting components which may be taken from predefined and user-defined class libraries. On the other hand, Modelica is also designed for supporting graphical model construction using specific graphical Modelica editors such as Dymola[10].

However, in order to couple a Modelica model of the physical system part with a statechart model that, for instance, represents a discrete-event controller, it is necessary to translate the behavior into terms of the Modelica language. As a result a single text file is created which represents the discrete controller as a monolithic input-output-block and which can be used seamlessly in textual and graphical model composition.

The transformation of a statechart model into a Modelica component has been implemented in the Alter language. The semantics of a statechart model which is introduced by this transformation agrees with the semantics of Statemate statecharts [7], i.e., the automatically generated Modelica component behaves like a Statemate statechart would do. The transformation procedure is very complex due to the concurrency and the histories and is not described here. As a prerequisite it is necessary to retrieve the structure of the statechart model. This traversing is done with built-in Alter procedures such as: (arcs aGraph) returns all Transition Objects of the model aGraph, (nodes aGraph) returns all top level states of the model aGraph, (get-property 'ORsub anOrState) returns all directly embedded substates of anOrState, (get-property 'event aTransition) returns the event attribute of aTransition, (container anObject) returns the container object, e.g., an OrState, of anObject

In order not to cut down the expressiveness of the Modelica language for formalisms which are going to be derived from this basic statechart formalism, the Graph class (figure 2) contains the properties opening-code and closing-code which support user-defined wrappings. These strings are placed at the top or at the bottom respectively of the generated Modelica component of the statechart. This feature may be used for example for implementing special Modelica interfaces and declaring additional variables.

B. *Visualizing simulation runs*

After simulating an overall Modelica model consisting of physical and discrete-event parts, the visualization of the resulting continuous and discrete-event simulation data is usually needed. Plotting the trajectories over the time is mostly an adequate solution for inspection of the continuous evolution. In contrast, the behavior of a complex discrete-event model may include cascades of events at the same instant of time so that it is more intelligible to animate step-by-step the transitions and the active states in that very model.

In the approach presented in this paper the editor which is applied for modeling a statechart is also used for visualizing the simulation data. For this, the meta-model of the basic statechart formalism (figure 2) contains additional properties which control particular visual attributes of the statechart modeling objects via the preconceived Alter methods. Consider for example, the class State in figure 2. It contains the Boolean property *active* (it is hyphenated for showing that it can not be edited by the user). The corresponding attribute *active* of an OrState or an AndState object indicates whether the state is active or not at the currently examined step. Using the following implementation for the Line Thickness method, the state object gets a thick border line if the global variable *visualization-mode* is true and the state is active: (if (and visualisation-mode (get-property 'active self)) 2 1).

In a similar way the attribute *last-fired* of a Transition object indicates whether it has been taken in the last step, and is visualized by the thickness of the arrow. The attribute *permission* shows that a transition will be taken in the next step, i.e., the origin state is active, the condition is fulfilled and the event occurs. This is denoted by a double line arrow (figure 4, from A2 to A3). In addition different colors are used for emphasizing the objects.

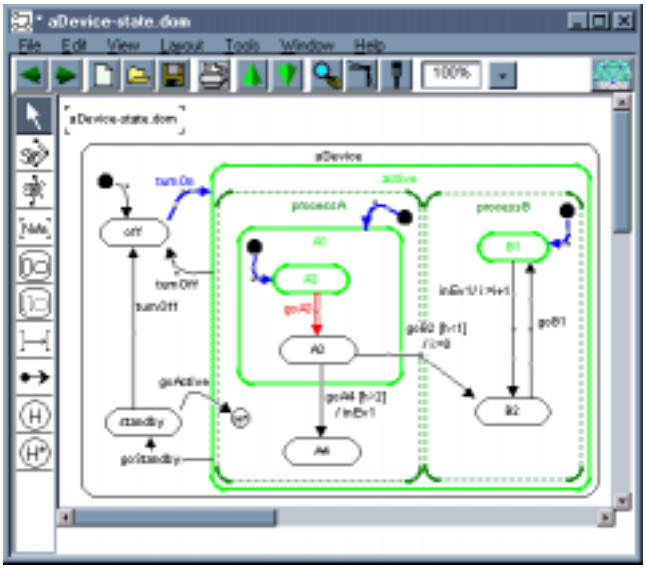

Figure 4: Visualization of a single step

With such features the data of only one step can be viewed. In order to examine the full sequence of steps of a simulation run, an Alter procedure is used which loads the simulation data and allows to go through the sequence displaying the data step by step or to jump to an arbitrary step. Note that due to the semantics of Modelica a sequence of events may occur in one instant of time.

At this point, technological systems can be modeled using the multi-domain modeling language Modelica for the physical parts, and the presented statechart editor for the discrete-event part (if statecharts is the adequate formalism). Then, the statechart models can be translated into Modelica modules which can be connected with the physical components resulting into a complete Modelica model. Using an ordinary Modelica simulation environment such as Dymola, this model can be automatically transformed and compiled into an executable simulation program based on standard (DAE-)integrators. Finally, the discrete-event part of the simulation results may be animated in the modeling editor.

Because of the meta-modeling capabilities of DOME it was relatively easy to implement the statechart editor. With the help of DOME's extension language Alter it was also simple to realize the visualization features. The very complex part is the interface to Modelica, i.e., the transformation into a Modelica component and the inverse transformation of the simulation results. Therefore in the next section it is shown how the basic statechart formalism can be extended easily without knowing anything about the interface to Modelica.

III. THE EXTENDED STATECHART FORMALISM

The extended statechart formalism provides the additional modeling objects AndNode and OrNode which can help to make certain models more intelligible. The semantics of the AndNode is that all transitions that are connected directly the that node have to fire synchronously, if and only if all conditions are fulfilled and all events occur. The semantics of the OrNode is that one of the incoming transitions and one of the outgoing transitions have to fire synchronously. Thus, a OrNode may be used as a fork or a join (figure 5).

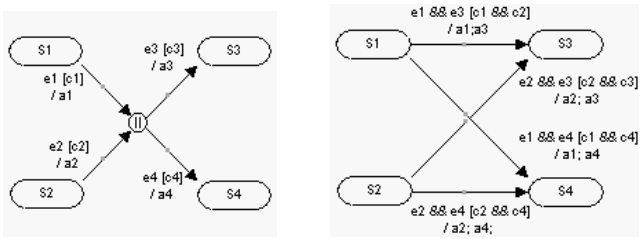

Figure 5: a) statechart fragment with OrNode
b) equivalent basic statechart fragment

The specification of the new editor is based on the DTS of the basic statechart formalism. The only difference with respect to the editing capability is that the two classes AndNode and OrNode have been added (figure 6) so that, for example, it becomes possible to model the statechart of figure 7a.

In order to specify the semantics and to enable the simulation support an unambiguous transformation procedure is implemented which transforms a model of this extended formalism into a newly created basic statechart model. This is done in two steps. First, all objects and properties which do not change are reconstructed in the basic model using generic Alter procedures which can also be used for other formalisms. These procedures receive a list of pairs of corresponding classes and create for each original object an related object in the basic model, if possible. When an object is instantiated the corresponding original object stores a reference to its related object in its *descendant* property. This is the only purpose of the generalization class *transformations* (figure 6). The result of this effortless step is shown in figure 7b.

Thus, the parts where the semantics changes, i.e., all objects that are associated with logical connectors, can be translated separately in a second step. This is the only difficult task due to the complex semantic issues. For example, it has to be decided how to transform a group of transitions that are linked to each other with several logical connectors. For the sample statechart the final result is shown in figure 7c. Note that it is necessary to use additional textual information in the transition label from Step1 to Step2 in order to represent the same behavior.

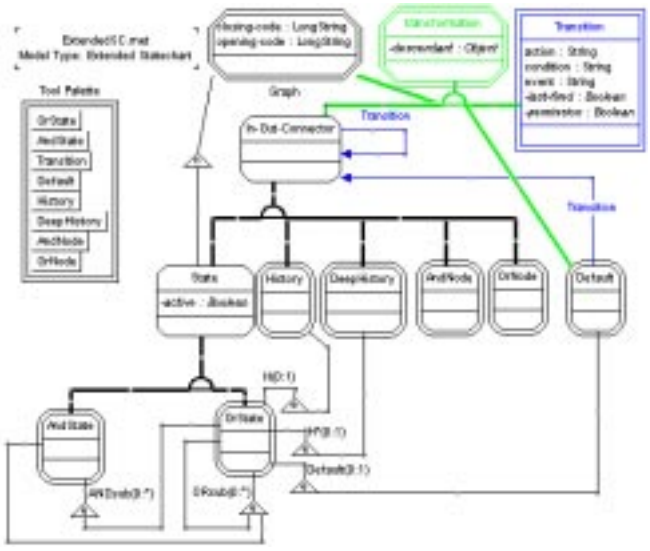

Figure 6: DTS of the extended statechart formalism

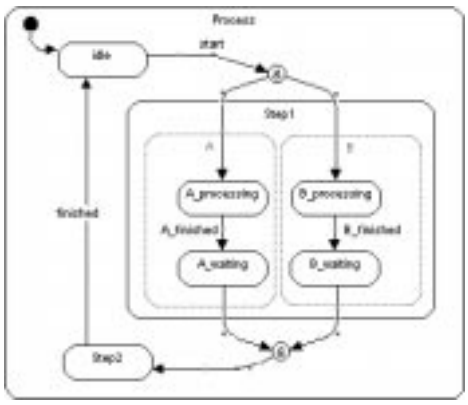

Figure 7a: the extended statechart model

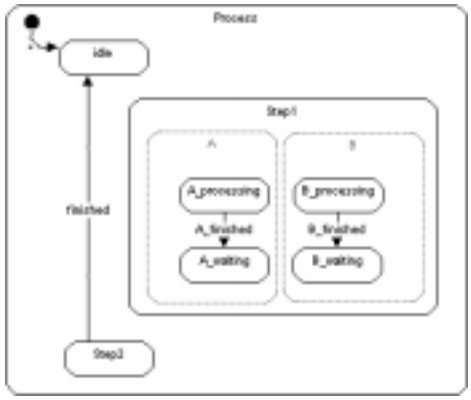

Figure 7b: remaining objects

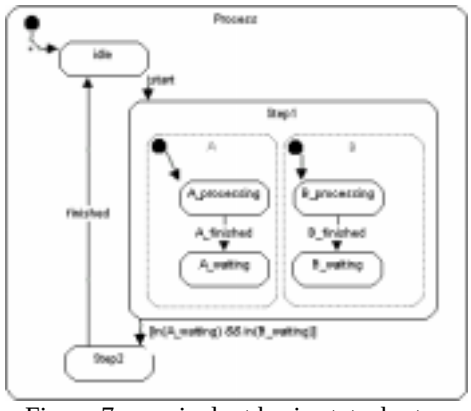

Figure 7c: equivalent basic statechart

With these prerequisites it is possible to generate automatically a Modelica component from an extended statechart model. The transformation process which consists of the creation of the basic statechart model and its translation into Modelica is performed in the background.

For visualizing simulation data in the extended statechart editor, additional Alter procedures are needed which map the visualization attributes of the basic statechart model into the attributes of the extended statechart model. For those objects which are common to both models generic procedures have been implemented, where as for the other objects situation specific solutions were found. Normally such mappings correspond to logical expressions on the properties of the underlying formalism. In the case of figure 7a, for example, all transitions that are linked to the upper logical connector get the same attributes as the corresponding transition of figure

7c. The navigation through the simulation results is realized by calling the corresponding procedures of the underlying formalism, such as show-next-step, show-previous-step or show-step-number, and updating the visual attributes using the mappings mentioned above.

With the combination of meta-modeling techniques and formalism transformation an extension of an existing formalism has been implemented providing simulation and visualization support with minimal effort. For this task only knowledge of the relationships between both formalisms was needed; it was not necessary to inspect the transformation and visualization procedures of the underlying formalism.

Because the extended formalism supports exactly the same functionality as the basic one, it is possible to implement another formalism based on the extended statecharts using the same methodology as described above. For example, with a few changes to the meta-model of the extended statechart it is possible to create an editor for a hybrid statechart formalism where the states can contain equations which are only active if the corresponding state is active (figure 9). Of course, the translation of the equations is not easy, because they have to be parsed, the consistency has to be checked and a equation section has to be created which is written to the closing-code

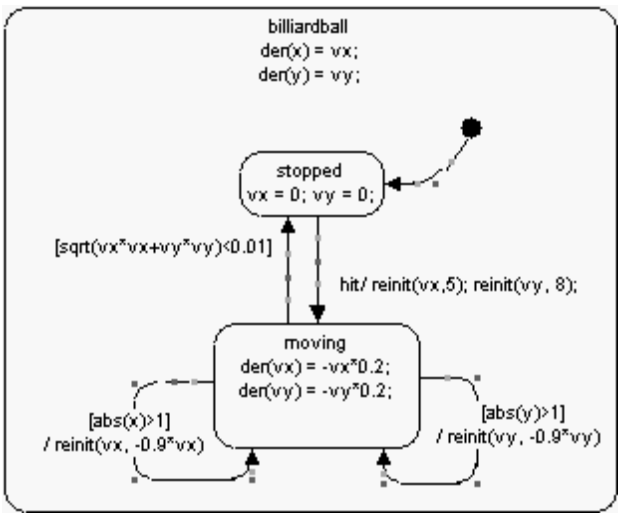

Figure 9: a hybrid statechart

```
der(x) = vx;  der(y) = vy;
if stopped then      der(vx) = 0;  der(vy) = 0;
else                 der(vx) = -vx*0.2;   der(vy) = -vy*0.2;
end if;

t0 = initial();
t1 = pre(stopped) and hit;
t2 = pre(moving) and sqrt(vx*vx+vy*vy)<0.01;
t3 = pre(moving) and abs(x)>1;
t4 = pre(moving) and abs(y)>1;

when {t0, t1, t2, t3} then
  reinit(vx, if t0 then 0 else if t1 then 5 else if t2 then 0 else -0.9*vx);
end when;
when {t0, t1, t2, t4} then
  reinit(vy, if t0 then 0 else if t1 then 8 else if t2 then 0 else -0.9*vy);
end when;
```

Figure 10: the generated equation section

property of the graph object. The reason for this difficulties is the nature of the formalism, so that they can not be avoided. Figure 10 shows the code of the hybrid statechart (der(x) is the time derivative of x).

By specifying hierarchically such formalism transformations an *Formalism Transformation Graph* which was introduced in [8], can be implemented. The effort for this is relatively low as long as the formalisms are rather similar.

IV. Summary

In the preceding sections a pragmatic approach for an extensible modeling and simulation software system for hybrid systems was presented. It was argued that it is advantageous to support domain-specific formalisms, i.e., to provide means for adapting, extending or creating new additional graphical editors. For physical system parts which are dominated by continuous dynamics, the modeling language Modelica is generic enough to support multi-domain modeling, e.g., the combination of electrical, hydraulic and mechanic schematics, whereas discrete-event formalisms and hybrid formalisms need specific editors.

It was shown that it is relatively easy to implement different statechart editors by using the meta-modeling features of DOME. Hierarchical formalism transformations turned out to be a rather efficient way for providing simulation and visualization support for these user-defined editors, if the formalisms are similar. This is because the user has to deal only with the semantic differences between two formalisms, where as the rest is done automatically.

Acknowledgement

The financial support from the German Research Foundation (DFG) under grants EN152/22-2 is gratefully acknowledged.

V. References

[1] M. Otter, M. A. Pereira Remelhe, S. Engell, P.Mosterman, “Hybrid Models of Physical Systems and Discrete Controllers,” *at - Automatisierungstechnik*, vol. 48, no. 09, pp. 426-437, 2000.

[2] Modelica. Homepage, http://www.Modelica.org/.

[3] M. Otter, H. Elmqvist, S.-E. Mattsson, “Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle,” in *Proc. of the 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD '99*, IEEE Control Systems Society, Hawaii, USA, August 1999.

[4] H. Elmqvist, S.-E. Mattsson, M. Otter, “Modelica – A Language for Physical System Modeling, Visualisation and Interaction,” in *Proc. of 1999 IEEE Symp. on Computer-Aided Control System Design, CACSD '99*, IEEE Control Systems Society, Hawaii, USA, August 1999.

[5] Eric Engstrom and Jonathan Krueger, “Building and Rapidly Evolving Domain-Specific Tools with DOME,” in *Proceedings of the IEEE International Symposium on Computer-Aided Control Systems Design 2000*, CACSD 2000, Anchorage, Alaska, USA, 2000.

[6] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming* 8, pp. 231-274, 1987.

[7] A. Naamad, D. Harel, “The STATEMATE Semantics of Statecharts,” *ACM Transactions on Software Engineering and Methodology*, 4, 1996.

[8] Hans L. M. Vangheluwe, “DEVS as a Common Denominator for Multi-formalism Hybrid System Modelling,” In *Proceedings of the IEEE International Symposium on Computer-Aided Control Systems Design 2000*, CACSD 2000, Anchorage, Alaska, USA, 2000.

[9] M. Remelhe and S. Engell, “Structuring Discrete Models in Modelica,” In Proceedings ADPM 2000, Dortmund, 2000.

[10] Dymola, Homepage http://www.dynasim.se/