Remelhe M.A.P.:

Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment

2nd International Modelica Conference, Proceedings, pp. 203-207

Paper presented at the 2nd International Modelica Conference, March 18-19, 2002,
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Oberpfaffenhofen, Germany.

All papers of this workshop can be downloaded from
http://www.Modelica.org/Conference2002/papers.shtml

Program Committee:

- Martin Otter, Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Robotik und Mechatronik, Oberpfaffenhofen, Germany (chairman of the program committee).
- Hilding Elmqvist, Dynasim AB, Lund, Sweden.
- Peter Fritzson, PELAB, Department of Computer and Information Science, Linköping University, Sweden.

Local organizers:
Martin Otter, Astrid Jaschinski, Christian Schweiger, Erika Woeller, Johann Bals,
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Robotik und
Mechatronik, Oberpfaffenhofen, Germany

Combining Discrete Event Models and Modelica – General Thoughts and a Special Modeling Environment

Manuel A. Pereira Remelhe
Process Control Laboratory
Department of Chemical Engineering
University Dortmund, D-44221 Dortmund
Tel.: +49/231/755-5127, Fax: +49/231/755-5129
Email: m.remelhe@ct.uni-dortmund.de

Abstract

This contribution consists of two parts. In the first part general possibilities for the combination of Modelica models and discrete event models are discussed on a conceptual level. It is shown that it is necessary to support asynchronous behavior and that it is useful to represent sampled data behavior of discrete event systems in an interrupt-driven style for fast simulation. The characterizations of the alternatives are summarized in table 1.
In the second part a modeling environment prototype that provides dedicated editors for different discrete event formalisms and supports hierarchical and heterogeneous models is presented briefly. It transforms a discrete event model into a Modelica class whose behavior is given by a Modelica algorithm and several Modelica functions. Such a discrete event component can be inserted intuitively into a model of a physical system and simulated by standard Modelica-Tools.

Introduction

Sophisticated technological systems often require complex discrete event control. For instance, sequential control is needed for the execution of recipes on chemical batch plants. Redundancy control is crucial for the safety of aircraft. And resource booking systems are needed for coordinating several interacting sequential controllers, e.g., to avoid collisions of robots or to prevent the mixing of parallel running batches in chemical batch plants. These discrete parts often involve hierarchical execution schemes as well as concurrency with synchronous and asynchronous communication. The physical part of such technological systems normally is very large and include complex hybrid dynamics such as Friction, collisions and instantaneous equilibrium reactions.

If simulation is to be used for the estimation of throughput, power consumption, quality quantities etc. the simulation model has to incorporate the discrete event as well as the physical part of the system. Hence, powerful modeling formalisms are required for both, the physical and the discrete part, and an intuitive integration of these parts has to be supported. In addition to the clearness of the modeling paradigm the simulation efficiency is an important aspect.

For the modeling and simulation of general hybrid physical systems Modelica [1] is particular suitable, because it facilitates multi-domain models, allows intuitive modeling of the most hybrid phenomena and enables efficient simulation. Furthermore it is possible to combine physical models with complex discrete event dynamics. In this contribution 4 general approaches for the integration of physical models and discrete-event models are discussed:

1. declarative style (based on equations),
2. imperative style (based on Modelica algorithms and functions),
3. external secondary simulator (based on the external function interface) and
4. external primary simulator (Modelica in the loop).

One well known example for the first approach is the Petri nets library created by Mosterman, Otter and Elmqvist [2]. The Petri net modeling objects, i.e., places and transitions, are represented by library components. These can be instantiated and connected via the connectors to constitute a specific Petri net graph. The equations of the transition and place objects were specified in a way such that the composition of these equations behaves like a Petri net would do.

The second approach implies that the behavior of a discrete event system is encoded imperatively into one Modelica algorithm that may call Modelica functions. In order to support high-level modeling formalisms a special modeling environment has to

be implemented that provides dedicated editors for specific discrete event formalisms and translates automatically a discrete event model into a Modelica component containing such an algorithm. A prototype of such a modeling environment will be presented in the second part of this contribution.

The third and fourth approaches allow the usage of existing discrete event simulators. In the third case a Modelica simulator controls the external simulator, whereas in the fourth case the Modelica simulator is controlled from outside, e.g., by a Stateflow-Simulink model.

Part I: General Thoughts

Interrupt-driven Models

The task of discrete event systems is normally to wait for certain state events in the physical system and react instantly when they occur, e.g., when a predefined temperature is reached, the next step of a recipe is started. Hence, from a functional point of view discrete event systems are interrupt-driven, or to be more precise, driven by state events (and sporadic time events).

A prerequisite for realizing this behavior is the detection and localization of state events during continuous integration. Since integration methods require continuous model equations, in Modelica all inequality expressions of a model are fixed during integration in order to guarantee that discontinuous changes of variables do not occur. In addition, inequality expressions which depend on continuous state variables and which are critical, are monitored during integration. When the logical value of an inequality expression changes, the time instant of the switching point is determined up to a certain precision and the integration is stopped, i.e., a state event is localized. In the case of time events the inequality expression depends only on the time variable and the integration simply stops directly at the predetermined time.

Modelica-Tools such as Dymola [3] support this event handling, if the events can be deduced from the Modelica code, i.e., the inequality expressions must be included in the code. Therefore an interrupt-driven model of a discrete event system can only be realized accurately with the first two approaches. If an external simulator is to be used, a wrapper code could be written in Modelica that defines the externally caused state events. But that would not be feasible for very complex discrete event models and the advantage of using an approved discrete event simulator would get lost due to error-prone hand coding. However, the fourth approach allows to detect the state events at the end of an integration step without using event localization, so that the event instants depend on the step size of the integrator.

Sampled-data Models

Discrete event systems are normally implemented as sampled-data systems. In the most cases the sampling rates are very high, so that the sampling can be neglected which results into interrupt-driven models. However, sometimes it is necessary to consider the sampling. The simplest way to do so is to use the **sample**-operator. In the following example a tank is filled continuously with the rate 1. A controller with 100 samplings per second opens the outlet when the tank level becomes higher than 10 and it closes the outlet when the tank level becomes lower than 0.1 .

```
model ControlledTank
  Real level;
  Boolean valveOpen;
equation
  der(level)= if valveOpen then
        1 - sqrt(level*2.173) else 1;
  when sample(0, 0.01) then
    valveOpen =
      if level > 10 then true
      else if level < 0.1 then false
      else pre(valveOpen);
  end when;
end ControlledTank;
```

The sample-Operator generates regular time events each 0.01 seconds. At every time event the integrator is stopped, the switching equation is evaluated and the integrator is started again. Since the initialization of the integrator consumes the largest amount of time, the simulation can be speed up significantly by transforming this sampled-data model into an interrupt-driven model that emulates the sampled-data behavior. This is done by replacing the **when** clause by the following two **when** clauses:

```
when {level<0.1, level>10} then
  sampleTime=(floor(time*100)+1)/100;
end when;
when time > sampleTime then
  valveOpen = if level>10 then true
        else if level<0.1 then false
        else pre(valveOpen);
end when;
```

In the first `when` clause an effective sampling time is determined, when a significant state event occurs. It is the next regular sample time the `sample` operator would have. In the second `when` clause then the switching equation is evaluated when the sampling time is reached. While no state events occur, the integration does not stop. For the given example the simulation of the interrupt-driven model is 70 times faster than the simulation of the sampled-data model. In consequence, the `sample`-operator should not be used for reactive discrete event systems. Instead, it is always possible to emulate sampled-data behavior, if required.

Asynchronous Behavior

In the first approach the modeling objects of a formalism are represented by Modelica objects whose behavior is defined declaratively using equations.
These equations are treated in the same way as the equations of the physical systems part so that a discrete transition of the discrete event model is connected to a complete evaluation of the whole system of equations including the physical systems equations. This synchronous behavior is disadvantageous when a formalism is used that performs a sequence of intermediate transitions in order to achieve a consistent state. Here, the effective transition that should be observable from the physical system incorporates several internal transitions that are asynchronous to the those of the physical system. Otherwise inconsistent intermediate states of the discrete event model could have an illegal effect on the physical system. Furthermore complex discrete event systems often involve complex hierarchical execution schemes: on each level a process can include sub-processes, and concurrent subsystems may run asynchronously. Therefore, it is crucial to support asynchronous behavior for modeling complex discrete event controllers. Hence, the equation-based approach is only suitable for simple cases where as the other three approaches can realize such complex behavior due to the algorithm-based imperative definition.

Other Aspects

Modelica algorithms are more limited in comparison to real programming languages, because Modelica's data structure is static, i.e., it is not possible to instantiate new variables or components at run-time. Therefore all variables that might be needed possibly at run-time have to be installed at compile-time. For example an event list of a scheduler of a discrete event simulator has to be realized by a fixed length vector in principle. If during a simulation the number of elements for the event list exceeds the length of the vector, the simulation has to abort.

Some discrete event systems have only discrete-time input signals. In this case state or time events are generated outside of the discrete event component, e.g., by limit switches in the physical system, and the usage of an external simulator is straightforward.

The models created with the Petri nets library look pretty much like Petri nets. The difference to the original formalism is that there no ports, so that the objects can be connected directly with lines and therefore only two classes of objects are needed. However, many formalisms have more complex syntax and graphics that can not be represented adequately using object-oriented composition diagrams. Statecharts for example has a state hierarchy concept without encapsulation of inner states. Consequently, dedicated graphical editors should be used in general. The following table summarizes the characterizations of the 4 approaches.

Table 1: Characterization of the 4 approaches

Approach	1	2	3	4
allows interrupt-driven models (localization of state events)	x	x	-	-
allows asynchronous behavior (hidden iterations)	-	x	x	x
allows dynamic data structures	-	-	x	x
adequate graphics	-	x	x	x
heterogeneous discrete event models	?	x	?	?
reliable discrete event simulation	?	?	x	x

Part II: A Prototype of a Modeling Environment

In order to enable heterogeneous discrete event models including domain-specific formalisms a modeling environment prototype has been developed that provides dedicated editors for the different formalisms. The environment is based on the meta-modeling tool DoME [4]. This tool generates automatically graphical editors based on a formal specification of the syntax and on parameters for the graphical appearance of the formalism. Therefore prototypes of new editors can be implemented within a few hours and a final version can be achieved within a few days. This is important for further development of domain-specific formalism.

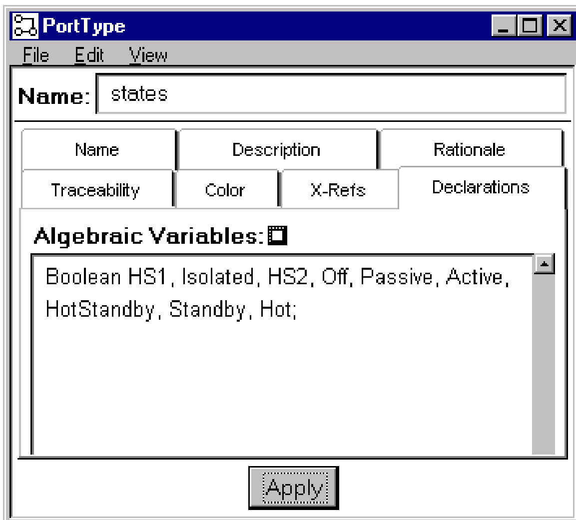

Figure 1: Defining a port type named "states" using Modelica syntax

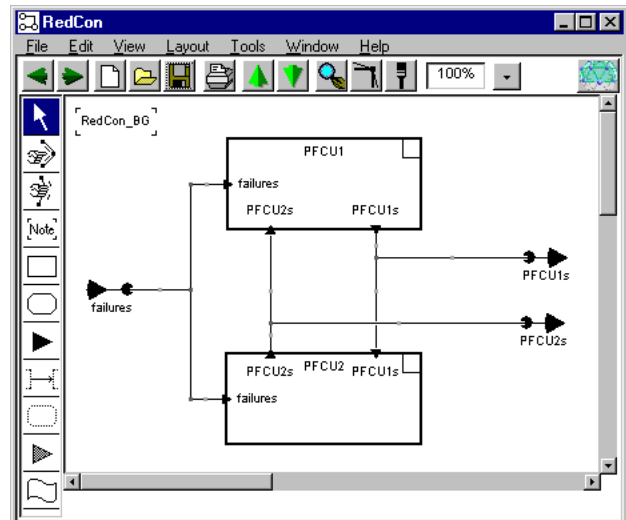

Figure 2: A block diagram implementation named "RedCon_BG"

An hierarchical block diagram formalism is implemented that allows the basic blocks to be modeled with any reactive formalism. At present, statecharts [5] and sequential function charts [6] are integrated, but other formalisms can be added in future.

Hierarchical Block Diagram Formalism

In the framework of the block diagram formalism an archetype concept is used. Each archetype defines a specific class of objects and can be instantiated several times. The definition of an archetype incorporates the ports of the objects (fig. 3), archetype attributes, object attributes and one or more alternative internal implementations, i.e., different interchangeable realizations. In order to cope with the possible combinations of different implementations one or more configurations can be defined that determine unambiguously which implementation is used for which instance.

In our case port types and block types can be defined. Port types are archetypes with archetype attributes (port variables and sub-ports), but without an implementation (fig.1). Ports can be instantiated within block archetypes and port archetypes. The block archetypes contain ports and no further attributes. The implementations of a block archetype can be a block diagram or a statechart or a sequential function chart (fig. 2 and 4). A block diagram contains the outer ports of the corresponding archetype and sub-blocks adorned with their own ports. In a block diagram one output port can be connected to several input ports, but one input port must not be connected with more than one output port (fig. 2).

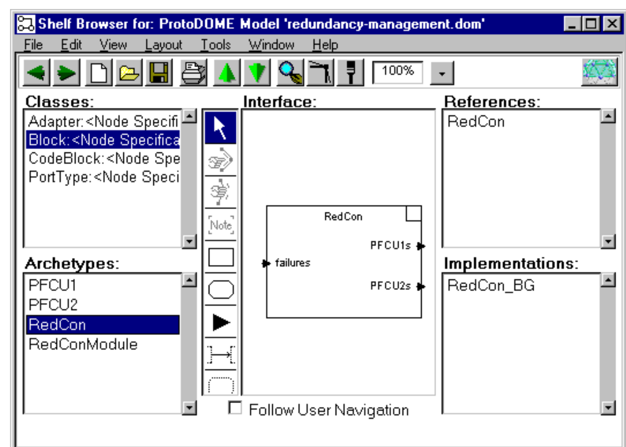

Figure 3: The shelf view on the archetypes

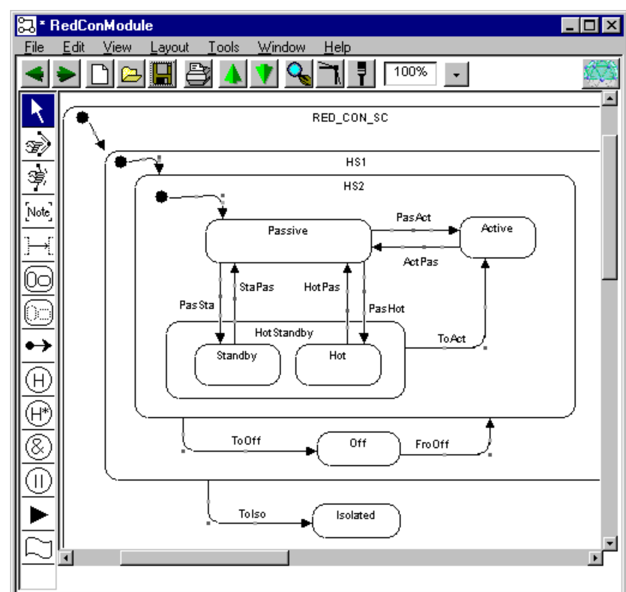

Figure 4: A statechart model that represents an implementation of an block

The Translation into Modelica

The translation procedures for the different formalisms are programmed using the extension facilities of DoME: The *Scheme* variant *Alter,* a *Lisp* like functional programming language, and *Smalltalk* in which DoME itself is implemented.

For the translation of a hierarchical model a specific implementation configuration is used. All parts that are automatically generated are encapsulated in one Modelica model class. The connectors of this class correspond to the outer ports on the highest model level. Each port type is directly represented by a connector definition. For each block implementation a record is generated that defines the components (variables and/or sub-blocks) of the corresponding implementation. Since the instantiation of such a record is connected to the instantiation of its components, the instance of the top-level block contains the whole data structure of the hierarchical model.

```
// block1 contains block2 and block3:
record Data_block1
   ...
   data_Block2 block2;
   data_Block3 block3;
   ...
   Boolean trig;
end Data_block1;
```

In addition for each block implementation a set of Modelica functions is generated that define its behavior. The following tasks are realized by single Modelica functions:
- initialization of the components (variables and sub-blocks) of the block implementation
- detecting the need for state transitions based on given input values
- performing state transitions based on given input values
- data logging for visualization

For a specific block implementation the argument type and the output type of these functions is the corresponding record. Hence, the initialization function of the top-level block gets its data object (a record) as an argument and calls the initialization function of its sub-blocks using the corresponding sub-objects included in the record:

```
function init_block1
   input Data_block1 par;
   output Data_block1 data;
algorithm
   data := par;
   data.block2 :=
            init_block2(data.block2);
   data.block3 :=
            init_block2(data.block3);
   ...
end init_block1;
```

Finally, the generated Modelica model class contains an algorithm that originates all function calls:

```
model DiscreteController
  < Port Definitions >
  < Record Definitions >
  < Function Definitions >
  InputPort inputConnector;
  OutputPort outputConnector;
  Data_block1 block1;
algorithm
  block1.inputPort := inputConnector;
  when initial() then
    block1 := init_block1(block1);
  end when;
  block1 := detect_block1(block1);
  if block1.trig then
    block1 := peform_block1(block1);
  end if;
  outputConnector :=
                    block1.outputPort;
end DiscreteController;
```

Such a Modelica model of a discrete event component can be inserted intuitively into a model of a physical system and simulated by standard Modelica-Tools.

References

[1] Modelica. Homepage: http://www.Modelica.org/

[2] Mosterman P.J., Otter M. and Elmqvist H.: Modeling Petri-Nets as Local Constraint Equations for Hybrid Systems using Modelica. *Proceedings of the 1998 Summer Computer Simulation Conference (SCSC'98)*, Reno, U.S.A., 19.-20. Juli 1998.

[3] Dymola. Homepage: http://www.Dynasim.se/.

[4] DoME. http://www.htc.honeywell.com/dome/.

[5] D. Harel: Statecharts: A Visual Formalism for Complex Systems. *Sc. of Comp. Prog.* 8, pp 231-274, 1987.

[6] International Electrotechnical Commission. *International Standard IEC 1131 Programmable Controllers, Part 3, Programming Languages*. IEC, Geneva, 1993.

[7] 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.

[8] M. A. Pereira Remelhe: Simulation and Visualization Support for User-defined Formalisms Using Meta-Modeling and Hierarchical Formalism Transformation. *Proceedings of the 2001 IEEE International Conference on Control Applications*, México City, 2001