Journal Européen des systèmes automatisés

Object-Oriented and Hybrid Modeling in Modelica

Hilding Elmqvist* — **Sven Erik Mattsson*** — **Martin Otter****

* *Dynasim AB*
Research Park Ideon, SE-223 70 Lund, Sweden

Elmqvist@Dynasim.se
SvenErik@Dynasim.se

** *German Aerospace Center, Institute of Robotics and Mechatronics*
D-82230 Wessling, Germany

Martin.Otter@DLR.de

ABSTRACT. *Modelica is an object-oriented language for modeling of large and heterogeneous physical systems. Typical applications include mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation and distribution of electric power. The unique features of Modelica to model combined continuous time and discrete event systems are discussed. A hybrid Modelica model is described by a set of synchronous differential, algebraic and discrete equations leading to deterministic behaviour and automatic synchronization of the continuous time and discrete event parts of a model.*

RÉSUMÉ. *Modelica est un langage orienté objet pour la modélisation des grands systèmes physiques hétérogènes. Il peut être utilisé pour des systèmes mécatroniques comportant des sous systèmes mécaniques, électriques ou hydralique et un système de commande dans des domaines tels que la robotique, l'aéronautique ou l'automobile mais également pour les processus industriels ou la génération et la distribution de l'énergie électrique. Dans cet article, on s'intéresse principalement aux mécanismes permettant de mixer des modèles continus et à événements discrets et on montre comment, dans Modelica, un modèle hybride est spécifié par un ensemble synchrone d'équations différentielles, algébriques et discrètes. Ceci conduit à un compartment déterministe et à une synchronisation automatique des parties continues et des parties événementielles.*

KEYWORDS: *Modelica, modeling language, object-orientation, hierarchical systems, hybrid modeling*

MOTS-CLÉS : *Modelica, langage de modélisation, orientation objet, systèmes hiérarchiques, modélisation hybride*

1. Introduction

Modeling and simulation are becoming more important since engineers need to analyze increasingly complex systems composed of components from different domains. Examples are mechatronic systems within automotive, aerospace and robotics applications. Such systems are composed of components from domains like electrical, mechanical, hydraulical, control, etc. Current tools are generally weak in treating multi-domain models because the *general* tools are block-oriented and thus demand a huge amount of manual rewriting to get the equations into explicit form. The *domain-specific* tools, such as circuit simulators or multibody programs, cannot handle components of other domains in a reasonable way.

There is too large a gap between the user's problem and the model description that the simulation program understands. Modeling should be much closer to the way an engineer builds a real system, first trying to find standard components like motors, pumps and valves from manufacturers' catalogues with appropriate specifications and interfaces. Only if there does not exist a particular subsystem, the engineer would actually construct it.

In Modelica, differential, algebraic and discrete equations are used for modeling of the physical phenomena. No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable handling of large models having more than hundred thousand equations.

Reuse is a key issue for handling complexity. There have been several attempts to define object-oriented languages for physical modeling. However, the ability to reuse and exchange models relies on a standardized format. It was thus important to bring this expertise together to unify concepts and notations. A design group was formed in September 1996 and one year later, the first version of the Modelica[1] language was available (www.Modelica.org) [MOD 00]. Modelica is intended to serve as a standard format so that models arising in different domains can be exchanged between tools and users. It has been designed by the developers of the object-oriented modeling languages Allan, Dymola, NMF, ObjectMath, Omola, SIDOPS+, Smile und a number of modeling practitioners in different domains. After 23 three-day meetings, during a 4-year period, version 1.4 of the language specification was finished in December 2000. Tools and free model libraries are now available, such as libraries for 1-dim. and 3-dim. mechanical systems, electric and electronic components, hydraulic systems, electric power systems, 1-dim. heat flow, control blocks.

2. Composition Diagrams

Modelica supports both high level modeling by composition and detailed library component modeling by equations. Models of standard components are typically

1. Modelica$^{\text{TM}}$ is a trade mark of the Modelica Association

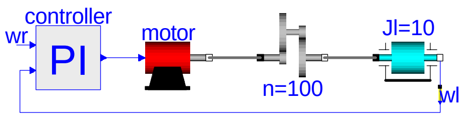


Figure 1. *Composition diagram of a motor drive.*

available in model libraries. Using a graphical model editor, a model can be defined by drawing a composition diagram by positioning icons that represent the models of the components, drawing connections and giving parameter values in dialogue boxes. Constructs for including graphical annotations in Modelica make icons and composition diagrams portable.

An example of a composition diagram of a simple motor drive system as presented in the modeling and simulation tool Dymola [DYM] is shown in Fig. 1. The system can be broken up into a set of connected components: an electrical motor, a gearbox, a load and a control system. The Modelica model is (excluding graphical annotations)

```
model MotorDrive
  PI         controller;
  Motor      motor;
  Gearbox    gearbox(n=100);
  Shaft      Jl(J=10);
  Tachometer wl;
equation
  connect(controller.outPort, motor.inPort);
  connect(motor.flange_b,     gearbox.flange_a);
  connect(gearbox.flange_b,   Jl.flange_a);
  connect(Jl.flange_b,        wl.flange_a);
  connect(controller.inPort,  wl.w);
end MotorDrive;
```

It is a composite model which specifies the topology of the system to be modeled in terms of components and connections between the components. The statement “`Gearbox gearbox(n=100);`” declares a component `gearbox` of model class `Gear-box` and sets the value of the gear ratio, `n`, to 100.

A component model may be a composite model to support hierarchical modeling. The composition diagram of the model class `Motor` is shown in Fig. 2.

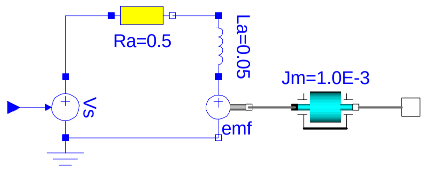


Figure 2. *Composition diagram of model class Motor.*

3. Variables, connectors and connections

Physical modeling deals with the specification of relations between physical quantities. For the drive system, quantities such as angle and torque are of interest. Their types are declared in Modelica as

```
type Angle  = Real(quantity = "Angle",  unit = "rad",
                   displayUnit = "deg");
type Torque = Real(quantity = "Torque", unit = "N.m");
```

where `Real` is a predefined type, which has a set of attributes such as name of quantity, unit of measure, default display unit for input and output, minimum, maximum, nominal and initial value. The *Modelica Standard Library*, which is an intrinsic part of Modelica includes about 450 of such type definitions.

Connections specify interactions between components and are represented graphically as lines between *connectors*. A connector should contain all quantities needed to describe the interaction. Voltage and current are needed for electrical components. Angle and torque are needed for drive train elements.

```
connector Pin              connector Flange
  Voltage      v;            Angle       phi;
  flow Current i;            flow Torque tau;
end Pin;                   end Flange;
```

A connection, **`connect`**`(Pin1, Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins such that they form one node. This implies two equations, `Pin1.v = Pin2.v` and `Pin1.i + Pin2.i = 0`. The first equation indicates that the voltages on both branches connected together are the same, and the second corresponds to Kirchhoff's current law stating that the current sums to zero at a node. Similar laws apply to flow rates in piping networks and to forces in mechanical systems. The sum-to-zero equations are generated when the prefix **`flow`** is used in the declarations. The Modelica Standard Library includes also connector definitions.

4. Partial models and inheritance

An important feature in order to build reusable descriptions is to define and reuse *partial models*. A common property of many electrical components is that they have two pins. Thus it is useful to define an interface model class `OnePort`, that has two pins, `p` and `n`, and a quantity, `v`, that defines the voltage drop across the component.

```
partial model OnePort
  Pin p, n;
  Voltage v;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
end OnePort;
```

The equations define common relations between quantities of a simple electrical component. The keyword **partial** indicates that the model is incomplete and cannot be instantiated. To be useful, a constitutive equation must be added. A model for a resistor extends `OnePort` by adding Ohm's law to define the behavior.

```
model Resistor "Ideal resistor"
  extends TwoPin;
  parameter Resistance R;
equation
  R*p.i = v;
end Resistor;
```

A string between the name of a class and its body is treated as a comment attribute. Tools may display this documentation in special ways. The keyword **parameter** specifies that the quantity is constant during a simulation experiment, but can change values between experiments.

For the mechanical parts, it is also useful to define a partial model with two flange connectors (`der(w)` means the time derivative of `w`)

```
partial model TwoFlanges
  Flange a, b;
end TwoFlanges;

model Inertia "1-dim. rotational inertia"
  extends TwoFlanges;
  parameter Inertia J = 1;
  Angle           phi;
  AngularVelocity w;
equation
  phi  = a.phi;                phi  = b.phi;
  J*der(w) = a.tau + b.tau;    der(phi) = w;
end Inertia;
```

5. Synchronous Equations

After the presentation of the fundamental structuring mechanisms in Modelica and the means to describe continuous models, attention is now given to discrete modeling features. In Modelica the central property is the usage of *synchronous* differential, algebraic and discrete equations. The idea of using the synchronous data flow principle in the context of hybrid systems was introduced in [ELM 93]. For pure *discrete event* systems, the same principle is utilized in synchronous languages [HAL 93] such as SattLine [ELM 92], Lustre [HAL 91] and Signal [GAU 94], in order to arrive at safe implementations of realtime systems and for verification purposes.

A typical example of a hybrid model is given in Fig. 3 where a continuous plant

$$\dot{\mathbf{x}}_p = \mathbf{f}(\mathbf{x}_p, \mathbf{u}) \qquad [1]$$

$$\mathbf{y} = \mathbf{g}(\mathbf{x}_p) \qquad [2]$$

is controlled by a digital linear controller

$$\mathbf{x}_c(t_i) = \mathbf{A}\mathbf{x}_c(t_i - T_s) + \mathbf{B}\left(\mathbf{r}(t_i) - \mathbf{y}(t_i)\right) \quad [3]$$

$$\mathbf{u}(t_i) = \mathbf{C}\mathbf{x}_c(t_i - T_s) + \mathbf{D}\left(\mathbf{r}(t_i) - \mathbf{y}(t_i)\right) \quad [4]$$

using a zero-order hold to hold the control variable **u** between sample instants (i.e., $\mathbf{u}(t) = \mathbf{u}(t_i)$ for $t_i \leq t < t_i + T_s$), where T_s is the sample interval, $\mathbf{x}_p(t)$ is the state vector of the continuous plant, $\mathbf{y}(t)$ is the vector of measurement signals, $\mathbf{x}_c(t_i)$ is the state vector of the digital controller and $\mathbf{r}(t_i)$ is the reference input. In Modelica, the complete system can be easily described by connecting appropriate blocks. However, for simplicity of the discussion, here an overall description of the system is given in one model where the discrete equations of the controller are within the **when** clause.

```
model SampledSystem
  parameter Real Ts=0.1 "sample period";
  parameter Real A[:, size(A,1)], B[size(A,1), :],
                 C[:, size(A,2)], D[size(C,1), size(B,2)];
  constant  Integer nx = 5;
  input  Real r[size(B,2)]  "reference";
  output Real y[size(B,2)]  "measurement";
         Real u [size(C,1)] "control";
         Real xc[size(A,1)] "disc. state";
         Real xp[nx]        "plant state";
equation
  der(xp) = f(xp, u);     // plant
       y  = g(xp);
  when sample(0,Ts) then  // controller
     xc = A*pre(xc) + B*(r-y);
     u  = C*pre(xc) + D*(r-y);
  end when;
end SampledSystem;
```

During continuous integration the equations within the **when** clause are de-activated. When the condition of the **when** clause *becomes* true an event is triggered, the integration is halted and the equations within the **when** clause are active at this event instant. The operator **sample**(...) triggers events at sample instants with sample time T_s and returns **true** at these event instants. At other time instants it returns **false**. The values of variables are kept until they are explicitly changed. For example, **u** is computed only at sample instants. Still, **u** is available at all time instants and consists of the value calculated at the last event instant.

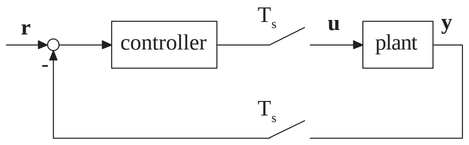


Figure 3. *Sampled data system.*

At a sampling instant t_i, the controller needs the values of the discrete state $\mathbf{x}_c$ for the time t_i and the previous sample instant $t_i - T_s$, which is determined by using the **pre** operator. Formally, the *left limit* $x(t^-)$ of a variable x at a time instant t is characterized by **pre**(x), whereas x itself characterizes the *right limit* $x(t^+)$. Since $\mathbf{x}_c$ is only discontinuous at sample instants, the left limit $\mathbf{x}_c(t_i^-)$ at sample instant t_i is identical to the right limit $\mathbf{x}_c(t_i^+ - T_s)$ at the previous sample instant and therefore **pre**($\mathbf{x}_c$) characterizes this value.

The *synchronous principle* basically states that at every time instant, the *active* equations express *relations* between variables which have to be *fulfilled concurrently*. As a consequence, during continuous integration the equations of the plant have to be fulfilled, whereas at sample instants the equations of the plant and of the digital controller hold *concurrently*. In order to efficiently solve such types of models, all equations are *sorted* by block-lower-triangular partitioning, the standard algorithm of object-oriented modeling for continuous systems (now applied to a mixture of continuous and discrete equations), under the assumption that all equations are active. In other words, the order of the equations is determined by data flow analysis resulting in an automatic synchronization of continuous and discrete equations. For the example above, sorting results in an ordered set of assignment statements:

```
// "known" variables: r, xp, pre(xc)
y := g(xp);
when sample(0,Ts) then
  xc := A*pre(xc) + B*(r-y);
  u  := C*pre(xc) + D*(r-y);
end when;
der(xp) := f(xp, u);
```

Note, that the evaluation order of the equations is correct both when the controller equations are active (at sample instants) and when they are not active.

The synchronous principle has several consequences: First, the evaluation of the discrete equations is performed in zero (simulated) time. In other words, time is abstracted from the computations and communications [GAU 94]. If needed, it is possible to model the computing time by *explicitly* delaying the assignment of variables. Second, in order that the unknown variables can be *uniquely* computed it is necessary that the number of active equations and the number of unknown variables in the active equations at every time instant are identical. This requirement is violated in

```
equation // incorrect model fragment!
  when h1 > 3 then
    close = true;
  end when;
equation
  when h2 < 1 then
    close = false;
  end when;
```

If by accident or by purpose the relations h1 > 3 and h2 < 1 become **true** at the same event instant, we have two conflicting equations for `close` and it is not defined which equation should be used. In general, it is not possible to detect by source inspection whether conditions become **true** at the same event instant or not. Therefore, in Modelica the assumption is used that *all equations* in a model may potentially be active at the same time instant during simulation. Due to this assumption, the total number of (continuous and discrete) equations shall be identical to the number of unknown variables. It is often possible to rewrite the model above by placing the when clauses in an **`algorithm`** section and changing the equations into assignments

```
algorithm
  when h1 > 3 then
    close := true;
  end when;
  when h2 < 1 then
    close := false;
  end when;
```

The algorithm section groups the two **`when`** clauses to be evaluated sequentially in the order of appearance and the second one gets higher priority. All assignment statements within the *same* **`algorithm`** section are treated as a set of n equations, where n is the number of different left hand side variables (e.g., the model fragment above corresponds to one equation). An **`algorithm`** section is sorted as a whole together with the rest of the system. Another assignment to `close` somewhere else in the model would still yield an error.

Handling hybrid systems in this way has the advantage that the *synchronization* between the continuous time and discrete event parts is *automatic* and leads to a deterministic behaviour *without conflicts*. Note, that some discrete event formalisms, such as state transition diagrams or prioritized Petri nets, can be formulated in Modelica in a component-oriented way, using synchronous equations, see [MOS 98] and Section 8. The disadvantage is that the types of systems which can be modeled is restricted. For example, general Petri nets cannot be described because such systems have non-deterministic behaviour.

6. Relation triggered events

During continuous integration it is advantageous that the model equations remain continuous and differentiable, since the numerical integration methods are based on this assumption. This requirement is often violated by if-clauses. For example the simple block of Fig. 4 with input u and output y may be described by

```
model TwoPoint
  parameter Real y0=1;
  input     Real u;
  output    Real y;
equation
   y = if u > 0 then y0 else -y0;
end TwoPoint;
```

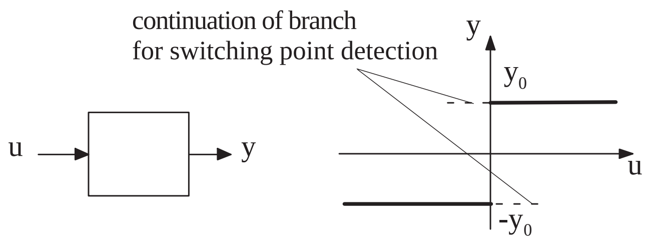


Figure 4. *Discontinuous component.*

At point `u=0` this equation is discontinuous, if the **`if`** expression would be taken *literally*. A discontinuity or a non-differentiable point can occur if a relation, such as $x_1 > x_2$ changes its value, because the branch of an if statement may be changed. Such a situation can be handled in a numerical sound way by detecting the switching point within a prescribed accuracy, halting the integration, selecting the corresponding new branch, and restarting the integration, i.e., by triggering a *state event*.

In general, it is not possible to determine by source inspection whether a specific relation will lead to a discontinuity or not. Therefore, in Modelica it is by default assumed that every relation potentially will introduce a discontinuity or a non-differentiable point in the model. Consequently, relations *automatically* trigger state events (or time events for relations depending only on time) at the time instants where their value is changed. This means, e.g., that model `TwoPoint` is treated in a numerical sound way (the **`if`** condition $u > 0$ is *not* taken literally but triggers a state event).

Modelica has several operators for hybrid systems, such as **`noEvent`**(...) to treat relations literally, **`reinit`**(x, value) to reinitialize a continuous state with a new value at an event instant, **`initial`**() to inquire the first and **`terminal`**() to inquire the last evaluation of the model during a simulation run.

7. Variable structure systems

If a physical component is modeled detailed enough, there are usually no discontinuities in the system. When neglecting some “fast” dynamics, in order to reduce simulation time and identification effort, discontinuities appear in a physical model. As a typical example, in Fig. 5 a diode is shown, where i is the current through the diode and u is the voltage drop between the pins of the diode. The diode characteristic is shown in the left part of Fig. 5. If the detailed switching behaviour is neglectable with regards to other modeling effects, it is often sufficient to use the ideal diode characteristic shown in the right part of Fig. 5, which typically gives a simulation speedup of 1 to 2 order of magnitudes.

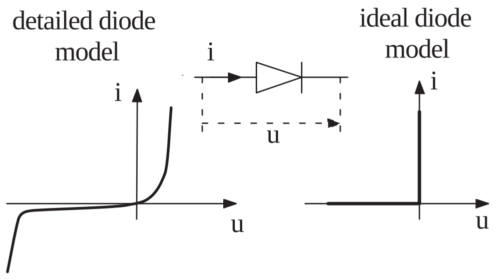
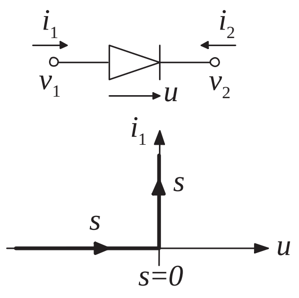


Figure 5. *Detailed and ideal diode characteristic.*

It is straightforward to model the detailed diode characteristic in the left part of Fig. 5, because the current i has just to be given as (analytic or tabulated) function of the voltage drop u. It is more difficult to model the ideal diode characteristic in the right part of Fig. 5, because the current at $u = 0$ is no longer a function of u, i.e., a mathematical description in the form $i = i(u)$ is no longer possible. This problem can be solved by introducing a curve parameter s and describing the curve as $i = i(s)$ and $u = u(s)$. This description form is more general and allows us to describe an ideal diode *uniquely* in a *declarative* way, see Fig. 6.



$$
\begin{aligned}
0 &= i_1 + i_2 \\
u &= v_1 - v_2 \\
\textit{off} &= s < 0 \\
u &= \textbf{if } \textit{off} \textbf{ then } s \textbf{ else } 0 \\
i_1 &= \textbf{if } \textit{off} \textbf{ then } 0 \textbf{ else } s
\end{aligned}
$$

Figure 6. *Ideal diode model*

In order to understand the consequences of parameterized curve descriptions, the ideal diode is used in the simple rectifier circuit of Fig. 7. Collecting the equations of all components and connections, as well as sorting and simplifying the set of equations under the assumption that the input voltage $v_0(t)$ of the voltage source is a known time function and that an ODE formulation is used, i.e. that the states (here: v_2) are assumed to be known and the derivatives should be computed, leads to

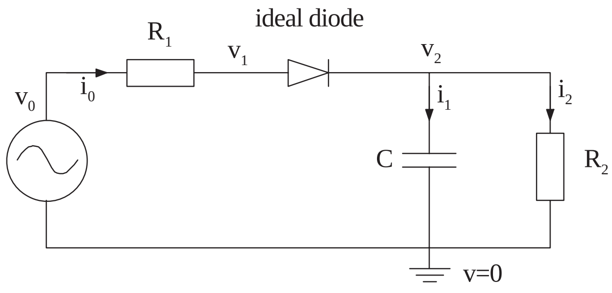


Figure 7. *Simple rectifier circuit.*

$$
\begin{aligned}
\text{off} &= s < 0 \\
u &= v_1 - v_2 \\
u &= \textbf{if}\ \text{off}\ \textbf{then}\ s\ \textbf{else}\ 0 \\
i_0 &= \textbf{if}\ \text{off}\ \textbf{then}\ 0\ \textbf{else}\ s \\
R_1 \cdot i_0 &= v_0(t) - v_1
\end{aligned}
\qquad [5]
$$

$$
\begin{aligned}
i_2 &:= v_2/R_2 \\
i_1 &:= i_0 - i_2 \\
\frac{dv_2}{dt} &:= i_1/C
\end{aligned}
$$

The first 5 equations are coupled and build a system of equations in the 5 unknowns off, s, u, v_1 and i_0. The remaining assignment statements are used to compute the state derivative $\dot{v}_2$. During continuous integration the Boolean variables, i.e., off, are fixed and the Boolean equations are not evaluated. In this situation, the first equation is not touched and the next 4 equations form a *linear* system of equations in the 4 unknowns s, u, v_1, i_0 which can be solved by Gaussian elimination. An event occurs if one of the relations (here: $s < 0$) changes its value.

At an *event instant*, the first 5 equations are a mixed system of discrete and continuous equations which cannot be solved by, say, Gaussian elemination, since there are both Real and *Boolean* unknowns. However, appropriate algorithms can be constructed: (1) Make an *assumption* about the values of the *relations* in the system of equations. (2) Compute the discrete variables. (3) Compute the continuous variables by Gaussian elimination (discrete variables are fixed). (4) Compute the relations based on the solution of (2) and (3). If the relation values agree with the assumptions in (1), the iteration is finished and the mixed set of equations is solved. Otherwise, new assumptions on the relations are necessary, and the iteration continues. Useful assumptions on relation values are for example: (a) Use the relation values computed in the last iteration and perform a fixed point iteration (the convergence can be enhanced by some algorithmic improvements). (b) Try all possible combinations of the values of the relations systematically (= exhaustive search). In the above example, both approaches can be simply applied, because there are only two possible values ($s < 0$ is

false or **true**). However, if n switches are coupled, there are n relations and therefore 2^n possible combinations which have to be checked in the worst case. More information about code generation for Modelica hybrid simulation can be found in [MAT 99]. Modeling ideal friction in Modelica is described in [OTT 99] and [MAT 99].

The technique of parameterized curve descriptions was introduced in [CLA 95] and a series of related papers. However, no proposal was given how to actually implement such models in a numerically sound way. In Modelica the (new) solution method follows logically because the equation based system naturally leads to a system of mixed continuous/discrete equations which have to be solved at event instants.

In the past, ideal switching elements have been handled by (a) using variable structure equations which are controlled by *state transition diagrams* to describe the switching behaviour, see e.g. [BAR 92, ELM 93, MOS 96], or by (b) using a *complementarity formulation*, see e.g. [LÖT 82, PFE 96, SCH 98]. The approach (a) has the disadvantage that the continuous part is described in a declarative way but not the part describing the switching behaviour. As a result, e.g., algorithms with better convergence properties for the determination of a consistent switching structure cannot be used. Furthermore, this involves a global iteration over *all* model equations whereas parameterized curve descriptions lead to local iterations over the equations of the involved elements. The approach (b) seems to be difficult to use in an object-oriented modeling language and seems to be applicable only in special cases (e.g. it seems not possible to describe ideal thyristors).

Mixed systems of equations do not only occur if parameterized curve descriptions are used, but also in other cases, e.g. whenever an if-statement is part of an algebraic loop and the expression of the if-statement is a function of the unknown variables of the algebraic loop. Consider the part of a drive train in Fig. 8. A simple way to model the losses in a gear due to friction between the gear teeth is by taking into account the gear efficiency according to:

$$\tau_2 \;=\; \bar{\eta} \cdot i \cdot \tau_1 \qquad [6]$$

where τ_1 is the input torque of the gear, i is the gear ratio, τ_2 is the load torque and $\bar{\eta}$ is the gear efficiency

$$\bar{\eta} \;=\; \begin{cases} \eta & for \quad \tau_1 \cdot \omega_1 > 0 \\ 1/\eta & for \quad \tau_1 \cdot \omega_1 < 0 \\ \text{undefined} & for \quad \tau_1 \cdot \omega_1 = 0 \end{cases} \qquad [7]$$
$$0 < \eta \leq 1$$

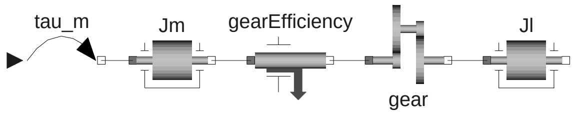


Figure 8. *Drive train with gear losses.*

that is, depending on the energy flow ($\tau_1 \cdot \omega_1$) the input torque τ_1 is either multiplied by η or by $1/\eta$. If the energy flow vanishes, the tooth in contact changes from front flank to trailing flank contact or vice versa. In this short phase no torque is transmitted, i.e., backlash is present. For simplicity, this transition phase may be neglected. Using such a very simplified model of the gear losses leads to the following sorted and simplified set of equations of the drive train in Fig. 8:

$$\begin{aligned} b &= \tau_1 \cdot \dot{\varphi}_l > 0 \\ \tau_2 &= (\textbf{if } b \textbf{ then } \eta \textbf{ else } 1/\eta) \cdot \tau_1 \\ J_m \cdot i \cdot \ddot{\varphi}_l &= \tau_m - \tau_1 \\ J_l \cdot \ddot{\varphi}_l &= i \cdot \tau_2 \end{aligned}$$

where J_m is the inertia of the motor, J_l is the inertia of the load, φ_l is the absolute angle of the load inertia and $\tau_m(t)$ is the motor torque calculated elsewhere (i.e., τ_m is a known quantity here). These are 4 coupled equations in the 3 Real unknowns $\ddot{\varphi}_l$, τ_1, τ_2 and 1 Boolean unknown b. The same techniques as before can be used to solve this system of equations at event instants.

8. Petri net modeling in Modelica

As was pointed out at the end of Section 5 some discrete event formalisms, such as state transition diagrams or prioritized Petri nets, can be formulated in Modelica in a component-oriented way based solely on a few model classes. This has the advantage that the synchronization between such a formalism and the continuous time part of a model is performed automatically and that no special graphical editor or external program is needed to define a discrete event model. In this section it is described at hand of a special type of Petri net, in which way Modelica can be utilized for such an approach.

Full realizations of more complex discrete-event formalisms, such as *state charts* [HAR 87] or *sequential function charts* [Int 93], cannot be conveniently done in a component-oriented way in Modelica. Reasons are that a special graphical layout may be needed which is not available in Modelica and/or that the basic structuring mechanism of Modelica — a hierarchy of encapsulated components — is not sufficient (e.g., the deep history connector of a state chart depends on the global structure of the statechart). In such cases, the discrete-event formalism has to be provided in an external program which is interfaced to Modelica based on the well-defined external function interface of Modelica. Alternatively, a dedicated graphical editor for the discrete-event formalism may be provided, from which appropriate Modelica code is generated [REM 00]. This has the advantage that the event handling of Modelica can be accessed easily.

8.1. *Petri Net Semantics*

The basic elements of a Petri net are places and transitions with directed connections between them. Places and transitions appear alternatingly, and, therefore, a Petri net is a directed bipartite graph. We will consider *Normal Petri Nets*, i.e., the case when each place contains zero or one token. Tokens are moved when transitions fire according to the following rules: If (i) all of the transition's input places have a token, (ii) none of the output places have a token, (iii) all tokens from the input places are removed and a token is added to each output place.

Note, that the *finite state machine* formalism is a subset of Petri nets where only transitions with one input and one output are allowed. In addition, there is a global constraint that only one initial token is present.

To illustrate the use of Petri nets, consider two reactors that are both repeatedly run through a heating/reaction phase followed by a cooling phase. During the reaction phase a shared resource, for example, a container for a chemical, needs to be utilized exclusively. The mutual exclusion for sharing of the common resource as well as the sequencing of the phases is handled by the Petri net in Fig. 9.

Consider, the transition `Xcold` which will fire provided `x<0.1` and `Xcooling` is active, i.e., has a token and `Resource` has a token. There is a similar condition for the transition `Ycold` to fire. If the conditions on `x` and `y` are fulfilled, either of the transitions `Xcold` and `Ycold` could fire. If `Xcold` fires, tokens are removed from `Xcooling` and `Resource` and a token is placed in `Xheating`. Since `Resource` does not have a token any longer, `Ycold` cannot fire.

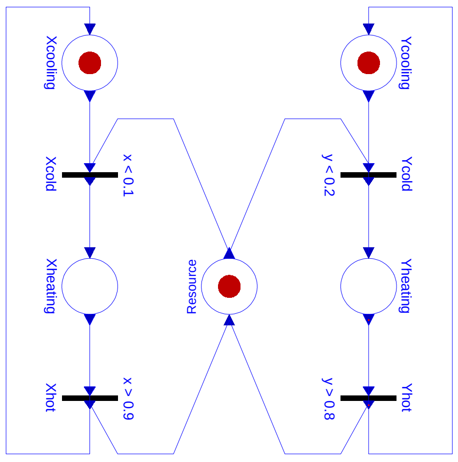


Figure 9. *Petri net example.*

We will consider *prioritized Petri nets* for which a deterministic choice is made when there are several possible transitions to fire. In this case the `Resource` place will give the priority depending on how the connections to transitions are made.

An implementation in Modelica has model classes `Place` and `Transition`. Information about status and firing is communicated through connections. Both places and transitions have input and output arrays of connectors. A `Place` has a boolean variable `token` and equations to determine if it has a token or not. It will send information to all its output transitions whether it has an `available` token or not. Its input transitions will receive information if the place is `occupied` or not.

A condition for a `Transition` to fire can, as discussed above, be written as follows, using functions operating on vectors of Booleans:

```
fire = condition and AllTrue(inp.available) and
       not AnyTrue(out.occupied);
```

The fire condition is sent as a `set` condition to all output places and as a `reset` condition to all input places. A `Place` changes its status as

```
token = AnyTrue(inp.set) or
        (pre(token) and not AnyTrue(out.reset));
```

The reporting of the state about accessible token to output transitions takes care of giving priority to transitions. The first transition gets information if the place has a token

```
out[1].available = pre(token);
```

The token is hidden to the second transition if the first transition decides to fire and sends a reset condition:

```
out[2].available = out[1].available and not out[1].reset;
```

The general case can be written in Modelica as

```
for i in 1:nReset loop
  out[i].available =
    if i==1 then pre(token)
    else out[i-1].available and not out[i-1].reset;
end for;
```

A Place needs to signal if it can accept a token by telling if it has a token or is about to receive one via transitions with higher priority. This is described as

```
for i in 1:nSet loop
  inp[i].occupied =
   if i==1 then pre(token)
   else inp[i-1].occupied or inp[i-1].set;
end for;
```

A more detailed analysis of this approach to describe Petri Nets is given in [MOS 98].

8.2. *Modelica Petri Net Library*

The above discussion will now be formalised in the form of a Petri Net library.

The connectors needed to carry out the signaling discussed above are

```
connector Reset "From Place to Transition"
  Boolean available "State of connected place";
  Boolean reset "True, if transition fires";
end Reset;

connector Set "From Transition to Place"
  Boolean occupied "State of connected place";
  Boolean set "True, if transition fires";
end Set;
```

The `Place` and `Transition` models are described as

```
model Place
  parameter Boolean initialToken=false;
  parameter Integer nSet=1, nReset=1;
  Set inp[nSet]; // array of connectors
  Reset out[nReset];
  Boolean token(start=initialToken);
equation
  // New token state for next iteration
  token = AnyTrue(inp.set) or
          (pre(token) and not AnyTrue(out.reset));
  // Report state to output transitions
  for i in 1:nReset loop
    out[i].available = if i == 1 then pre(token)
      else out[i - 1].available and not out[i - 1].reset;
  end for;
  // Report state to input transitions
  for i in 1:nSet loop
    inp[i].occupied = if i == 1 then pre(token)
      else inp[i - 1].occupied or inp[i - 1].set;
  end for;
end Place;

model Transition
  input Boolean condition;
  parameter String condLabel;
  parameter Integer nReset=1, nSet=1;
  Reset inp[nReset];
  Set out[nSet];
protected
  Boolean fire;
equation
  fire = condition and AllTrue(inp.available) and
         not AnyTrue(out.occupied);
  inp.reset = fill(fire, nReset);
  out.set = fill(fire, nSet);
end Transition;
```

The following utility functions are utilized

```
function AnyTrue "Logical OR of vector"
  output Boolean result;
  input  Boolean b[:];
algorithm
  result := false;
  for i in 1:size(b, 1) loop
    result := result or b[i];
  end for;
end AnyTrue;

function AllTrue "Logical AND of vector"
  output Boolean result;
  input  Boolean b[:];
algorithm
  result := true;
  for i in 1:size(b, 1) loop
    result := result and b[i];
  end for;
end AllTrue;
```

8.3. *Example – Common resource*

The Petri net in Fig. 9 can be modeled in Modelica as[2]

```
model ResourceHandling
 "Petri net for exclusive access of a common resource"
 NormalPetriNet.Place
   Xcooling(initialToken=true), Xheating,
   Ycooling(initialToken=true), Yheating,
   Resource(nSet=2, nReset=2, initialToken=true);
 NormalPetriNet.Transition
   Xcold(nReset=2, condLabel="x<0.1"),
   Xhot(nSet=2, condLabel="x>0.9"),
   Ycold(nReset=2, condLabel="y<0.2"),
   Yhot(nSet=2, condLabel="y>0.8");
equation
 connect(Xcold.out[1], Xheating.inp[1]);
 connect(Xheating.out[1], Xhot.inp[1]);
 connect(Xhot.out[1], Xcooling.inp[1]);
 connect(Ycold.out[1], Yheating.inp[1]);
 connect(Yheating.out[1], Yhot.inp[1]);
 connect(Yhot.out[1], Ycooling.inp[1]);
 connect(Xcold.inp[1], Xcooling.out[1]);
 connect(Ycold.inp[1], Ycooling.out[1]);
 connect(Ycold.inp[2], Resource.out[2]);
 connect(Xhot.out[2], Resource.inp[1]);
```

2. Dymola [DYM] allows Modelica components and connections of components to be defined with a graphical editor and the result stored as Modelica code with graphical annotations.

```
 connect(Xcold.inp[2], Resource.out[1]);
 connect(Yhot.out[2], Resource.inp[2]);
end ResourceHandling;
```

The continuous parts can, for example, be modeled in Modelica by extending the model ResourceHandling.

```
model CommonResource
 "Example to demonstrate exclusive access of a common resource"
  extends ResourceHandling;
  parameter Real a=1, b=0.5, f=2.5, g=1;
  Real x, y;
equation
  der(x) = -a*x + (if Xheating.token then f else 0);
  der(y) = -b*y + (if Yheating.token then g else 0);
  Xcold.condition = x < 0.1;      Xhot.condition  = x > 0.9;
  Ycold.condition = y < 0.2;      Yhot.condition  = y > 0.8;
  when x < 0.1 then
    reinit(x, 0);
  end when;
  when y < 0.2 then
    reinit(y, 0);
  end when;
end CommonResource;
```

The continuous part are just linear models in this case with the energy term dependent on the state of the heating places. The status of the Petri net can be accessed by using dot notation: `Xheating.token`, i.e. the energy term can be written **`if`** `heating.token` **`then`** `f` **`else`** `0`. In a corresponding way, the transition conditions can be defined as: `Xcold.condition = x < 0.1;`.

When simulating the model for 10 seconds, the plot in Fig. 10 shows that in some cases two heating phases of `X` occur during one cooling phase of `Y`. However, after that, the `Y` reactor has to wait for the `Resource`.

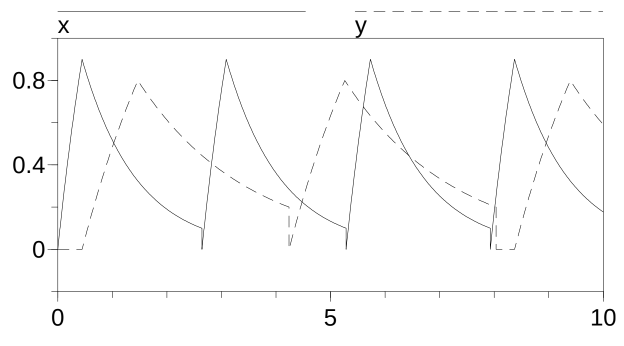


Figure 10. *Heating and cooling of X and Y.*

9. Other synchronous languages

In this section certain aspects of the synchronous languages Lustre [HAL 91] and Signal [GAU 94] are discussed in order to point out the relationship with the Modelica hybrid model. It also gives a rationale for the language constructs chosen in Modelica.

Lustre and Signal are both synchronous data flow languages and have the single assignment principle. As pointed out in [HAL 91]: "... sequencing and synchronization constraints arise from data dependencies" However, contrary to Modelica, they are designed to model discrete systems only.

In Lustre, any variable denotes a flow, i.e, a pair of (i) a possible infinite sequence of values and (ii) a clock; representing a sequence of times. This is a conceptual model and any implementation would only store a small window of values and times. The usual operators operate on variables and expressions sharing the same clock. Temporal operators are

1. **pre**(e) gives a sequence obtained by shifting the values of e one "clock step".
2. `->` (followed by) is used to give initial values, e.g., "`n = 0 ->` **`pre`**`(n) + 1:`" gives a counter starting at 0.
3. **when** samples an expression according to a slower clock. "e **when** b", where b is a boolean expression, returns an expression with the same clock as b and the values at those times taken from e.
4. **current** interpolates an expression on the clock immediately faster than its own. The interpolation is a "zero order hold".

Signal has corresponding temporal operators:

1. "x $ 1" means x delayed one step, i.e., **pre**(x).
2. **when** as in Lustre.
3. "x **default** y" means a merging of the sequences x and y. If at one time no value of x is present, then the y value is taken.

Both languages have "equation" with just one variable at the left hand side, i.e., causality is given. However, sorting of the equations is performed. There are thus mechanisms in both languages to describe difference equations using a "delay" operator. There are sample operators (when). Lustre has an interpolation operator.

9.1. *Possible generalizations to hybrid signals*

It seems most natural to have a definition for discrete signals for any time, not only for a sequence of clock times, i.e., to consider them as piecewise constant signals. This corresponds also to the physical reality of for example a variable in a computer. The value of it does exist also between the executions of the algorithm since it is stored in memory. The interpolation operator is then automatically available since a variable always has a value and can be accessed any time.

The sample operator "x **when** b" could naturally be extended to handle the case where x is a continuous signal. The values of x are sampled at certain time instants and the result is kept constant in-between. One could define the sampling instants as "when b changes" but it seems more convenient to use the definition "when b becomes true". One could then, for example, write `u when time >= SampleTime` to sample the continuous variable `u` at `Time==SampleTime` where `SampleTime` could be a variable that always contains the next time for sampling.

9.2. *Possible language constructs for sampled data systems*

A linear sampled system can be described as

$$x(t_{i+1}) = a \cdot x(t_i) + d \cdot u(t_i) \quad [8]$$

$$y(t_i) = c \cdot x(t_i) + d \cdot u(t_i) \quad [9]$$

or by shifting x one sample interval:

$$x(t_i) = a \cdot x(t_{i-1}) + d \cdot u(t_i) \quad [10]$$

$$y(t_i) = c \cdot x(t_{i-1}) + d \cdot u(t_i) \quad [11]$$

In any case, we can distinguish three kinds of features:

1. *Sampling* - the input u might be continuous and $u(t_i)$ is the sampled signal.

2. *Sample and hold* - the output might go to a continuous subsystem. Since the value $y(t_i)$ is only valid at certain time instants, some kind of interpolation is needed. Zero-Order Hold is typical, i.e. that y is piecewise constant.

3. *Shift operator* - Since a difference equation refers to values of the variables at several time instants, it is convenient to introduce a shift operator. The forward shift operator is often denoted by q and defined as $q(x(t_i)) = x(t_{i+1})$. The inverse is called the backward shift operator and denoted by q^{-1}, see, e.g., [ÅST 96].

A sampled system might be written as

```
Sample     = time >= pre(SampleTime);
SampleTime = (time + DT) when Sample;
x = a*pre(x) + b*(u when Sample);
y = c*pre(x) + d*(u when Sample);
```

Note that a hold operator was not needed because the equations are always valid and are giving piecewise constant signals for x and y. It would be possible to introduce just one operator **new** to replace **pre** and **when**:

```
Sample = time >= SampleTime;
new(SampleTime, Sample) = time + DT;
new(x, Sample) = a*x + b*u;
new(y, Sample) = c*x + d*u;
```

We notice that in all cases, we have to repeat the sampling condition in many places. For this reason, in Modelica a grouping mechanism is used where the sampling condition is only mentioned once:

```
when Time >= pre(SampleTime) then
  SampleTime = time + DT;
  x = a*pre(x) + b*u;
  y = c*pre(x) + d*u;
end when;
```

Alternatively, the dual **new** operator could be used as:

```
when time >= SampleTime then
  new(SampleTime) = time + DT;
  new(x) = a*x + b*u;
  y      = c*x + d*u;
end when;
```

This is similar in spirit to the **with** statement for accessing components of a record in, for example, Pascal. In that case using dot-notation everywhere is avoided.

The equations in a when-clause can be viewed in two ways: (1) An equation is always valid and sampling of signals is done when the condition becomes true. (2) An equation is only valid, i.e., it is activated, when the condition becomes true (instantaneous equation). This corresponds to the reality in a computer control algorithm where the statements are only executed at certain time instances.

Basically, the **pre** and **new** operator are equivalent and it is always possible to automatically transform a model written with the **pre** operator into an equivalent model using the **new** operator. It was decided to introduce the **pre** operator in Modelica.

10. Conclusions

Modelica is based on synchronous differential, algebraic and discrete equations, leading to a unified mathematical description of continuous time and discrete event parts of a model. Ideal switching elements, such as ideal diodes, can be described in Modelica using the technique of parameterized curves and Modelica tools can simulate such models in an efficient and reliable way. Normal Petri nets can be implemented by place and transition objects with appropriate communication.

Acknowledgements

The authors would like to thank the other members of the Modelica Association for inspiring discussions and their contributions to the Modelica design.

11. References

[ÅST 96] ÅSTRÖM K., WITTENMARK B., *Computer-Controlled Systems — Theory and Design,* Prentice Hall, 3 edition, 1996.

[BAR 92] BARTON P., “The Modelling and Simulation of Combined Discrete/Continuous Processes”, PhD thesis, University of London, Imperial College, 1992.

[CLA 95] CLAUSS C., HAASE J., KURTH G., SCHWARZ P., "Extended Amittance Description of Nonlinear n-Poles", *Archiv für Elektronik und Übertragungstechnik / International Journal of Electronics and Communications*, vol. 40, 1995, p. 91–97.

[DYM] DYMOLA, "Dynasim AB, Lund, Sweden, Homepage: http://www.dynasim.se/.".

[ELM 92] ELMQVIST H., "An Object and Data-Flow based Visual Language for Process Control", *ISA/92-Canada Conference & Exhibit*, Toronto, Canada, 1992, Instrument Society of America.

[ELM 93] ELMQVIST H., CELLIER F., OTTER M., "Object–Oriented Modeling of Hybrid Systems", *Proceedings ESS'93, European Simulation Symposium*, Delft, The Netherlands, 1993, Society for Computer Simulation International, p. xxxi-xli.

[GAU 94] GAUTIER T., GUERNIC P. L., MAFFEIS O., "For a New Real-Time Methodology", *Publication Interne No. 870, Institut de Recherche en Informatique et Systemes Aleatoires*, Campus de Beaulieu, 35042 Rennes Cedex, France, 1994.

[HAL 91] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D., "The synchronous data flow programming language LUSTRE", *Proc. of the IEEE*, vol. 79, 1991, p. 1305–1321.

[HAL 93] HALBWACHS N., *Synchronous Programming of Reactive Systems*, Kluwer, 1993.

[HAR 87] HAREL D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, vol. 8, 1987, p. 231–274.

[Int 93] INTERNATIONAL ELECTROTECHNICAL COMMISSION, *International Standard IEC 1131 Programmable Controllers, Part 3, Programming Languages*, IEC, Geneva, 1993.

[LÖT 82] LÖTSTEDT P., "Mechanical systems of rigid bodies subject to unilateral constraints", *SIAM J. Appl. Math.*, vol. 42, 1982, p. 281–296.

[MAT 99] MATTSSON S. E., OTTER M., ELMQVIST H., "Modelica hybrid modeling and efficient simulation", *Proceedings of the 38th IEEE Conference on Decision and Control*, Phoenix, Arizona, USA, Dec 1999, p. 3502–3507, Invited session paper.

[MOD 00] MODELICA, "A unified object-oriented language for physical systems modeling", Modelica homepage: http://www.Modelica.org/current/ModelicaSpecification14.pdf, 2000.

[MOS 96] MOSTERMAN P., BISWAS G., "A Formal Hybrid Modeling Scheme for Handling Discontinuities in Physical System Models", *Proceedings of AAAI-96*, Portland, OR, USA, 1996, p. 985–990.

[MOS 98] MOSTERMAN P., OTTER M., ELMQVIST H., "Modeling Petri Nets as Local Constraint Equations for Hybrid Systems using Modelica", *SCSC'98*, Reno, Nevada, USA, 1998, Society for Computer Simulation International, p. 314–319.

[OTT 99] OTTER M., ELMQVIST H., MATTSSON S. E., "Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle", *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99*, Hawaii, USA, Aug 1999, IEEE Control Systems Society.

[PFE 96] PFEIFFER F., GLOCKER C., *Multibody Dynamics with Unilateral Contacts*, John Wiley, 1996.

[REM 00] REMELHE M. A. P., "Abbildung von State Charts in einen Modelica Algorithmus", report , 2000, Lehrstuhl für Anlagensteuerungstechnik, Universität Dortmund, Germany.

[SCH 98] SCHUHMACHER J. M., VAN DER SCHAFT A. J., "Complementarity Modeling of Hybrid Systems", *IEEE Transactions on Automatic Control*, vol. 43, 1998, p. 483–490.