Modeling of mixed Continuous/Discrete Systems in Modelica

Martin Otter, Hilding Elmqvist, Sven Erik Mattsson

Version of March 8, 1999

Contents

1 Introduction

This report describes the handling of mixed continuous/discrete models in Modelica 1.1a. Such types of models are also called *hybrid systems*. It is an updated version of the draft proposal from October 30, 1998, where the changes decided on the Modelica meetings of Nov. 5.–7., 1998 and of Feb. 4.–6., 1999, have been incorporated. Essentially the following features are discussed: (a) Basic elements to model, e.g., sampled data systems or limiters and to synchronize the discrete and continuous parts of a model with each other. (b) A new technique to handle variable structure systems, such as ideal diodes, ideal thyristors, Coulomb friction, phase changes via parameterized curve descriptions of the element characteristics.

With respect to Modelica 1.0, there are nearly no changes in the Modelica grammar, but additional built-in operators as well as modifications in the *language semantics* and in the formulation of ideal elements. The most important changes are:

- The **new** operator for discrete variables is replaced by the more convenient **pre** operator, which characterizes the left limit of a variable.
- The **new** operator for continuous state variables is replaced by the more powerful **reinit** operator.
- Variable structure components, such as an ideal diode, are defined in a fully declarative way.
- The *global* event iteration is removed and replaced by the *local* solution of mixed continuous/discrete systems of equations. This is more efficient and reliable.

2 Basic language elements for hybrid models in Modelica

In this section the basic language elements for hybrid models in Modelica 1.1a are introduced to describe, e.g., the connection of continuous plants and discrete controllers. In the following sections, generalizations are discussed to handle *variable structure* systems in a declarative way.

2.1 Synchronous continuous and discrete equations

The central property is the usage of synchronous continuous and discrete equations based on the single assignment rule, in order to have deterministic and automatic syncronization of the continuous and discrete parts of a hybrid model via dataflow analysis. This idea was introduced in [Elmq93] as a generalization of discrete synchronuous languages [Halb93], such as LICS [Elmq85], Lustre [Halb91], Signal [Gaut94] which are designed to arrive at save implementations of realtime systems and for verification purposes.

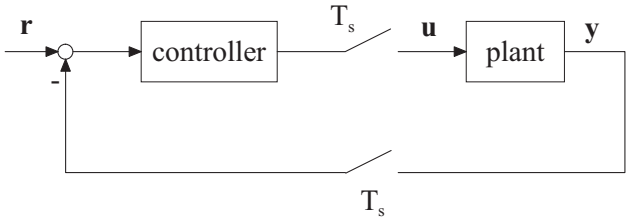

Figure 1: Continuous plant controlled by discrete controller.

A Modelica model basically consists of ordinary differential equations, algebraic equations and discrete equations. A typical example is given in figure 1 where a continuous plant is controlled by a discrete linear controller of the form

$$\mathbf{x}_c(t_i) = \mathbf{A}\cdot\mathbf{x}_c(t_i - T_s) + \mathbf{B}\cdot(\mathbf{r}(t_i) - \mathbf{y}(t_i)) \quad (2.1a)$$

$$\mathbf{u}(t_i) = \mathbf{C}\cdot\mathbf{x}_c(t_i - T_s) + \mathbf{B}\cdot(\mathbf{r}(t_i) - \mathbf{y}(t_i)) \quad (2.1b)$$

using a zero-order hold to hold the control variable **u** between sample instants (i.e., $\mathbf{u}(t) = \mathbf{u}(t_i)$ for $t_i \leq t < t_i + T_s$). This system is described in Modelica in the following way:

$$
\begin{aligned}
&\dot{\mathbf{x}}_p = \mathbf{f}(\mathbf{x}_p, \mathbf{u}) \\
&\mathbf{y} \;\; = \mathbf{g}(\mathbf{x}_p) \\
&\textbf{when } \text{sample}(0, T_s) \textbf{ then} \\
&\quad \mathbf{x}_c = \mathbf{A} \cdot \textbf{pre}(\mathbf{x}_c) \; + \mathbf{B} \cdot (\mathbf{r} - \mathbf{y}) \\
&\quad \mathbf{u} \;\; = \mathbf{C} \cdot \textbf{pre}(\mathbf{x}_c) \; + \mathbf{D} \cdot (\mathbf{r} - \mathbf{y}) \\
&\textbf{end when}
\end{aligned}
$$

where $\mathbf{x}_p$ is the state vector of the *continuous* plant, **u** is the plant input vector, **y** is the vector of measurement signals, $\mathbf{x}_c$ is the state vector of the *discrete* controller and **r** is the reference input. The **pre** operator provides the (known) value of a variable from the previous event instant. The equations in a **when** clause are *conditionally activated* at event instants where the when-condition (here: $\text{sample}(0, T_s)$) *becomes* true.

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**. Note, that 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.

The model above consists of the continuous equations of the plant and of the discrete equations of the controller within the **when** clause. During continuous integration the equations of the **when** clause are de-activated. When the condition of the **when** clause *becomes* true an event is triggered and the equations within the **when** clause are activated. At every time instant, the activated equations express *relations* between variables which have to be *fulfilled concurrently*. Particularily this means that at an event instant it is assumed that the evaluation of the equations of the discrete equations is done in zero time. In other words, time is abstracted from the computations and communications, i.e., computing and communicating at an event instant take no time, see also [Gaut94]. If needed, it is possible to model the computing time by *explicitly* delaying the assignment of variables, e.g.,

$$
\begin{aligned}
&\textbf{when } \text{sample}(0, T_s) \textbf{ then} \\
&\quad \mathbf{x}_c = \mathbf{A} \cdot \textbf{pre}(\mathbf{x}_c) \; + \mathbf{B} \cdot (\mathbf{r} - \mathbf{y}) \\
&\quad \mathbf{w} \;\; = \mathbf{C} \cdot \textbf{pre}(\mathbf{x}_c) \; + \mathbf{D} \cdot (\mathbf{r} - \mathbf{y}) \\
&\quad \mathbf{u} \;\; = \textbf{pre}(\mathbf{w}) \\
&\textbf{end when}
\end{aligned}
$$

In this case, the assignment of **u** is delayed for one sampling instant.

In order that the unknown variables can be *uniquely* computed it is necessary that the number of activated equations and the number of unknown variables in the activated equations at every time instant is identical. To get *deterministic* behaviour, this requirement has to be further restricted as seen by the following example:

```
when condition1 then
    close = true;
end when

when condition2 then
    close = false;
end when
```

When `condition1` and `condition2` never become **true** at the same time instant, we have deterministic behaviour. However, when by accident or by purpose the two conditions 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 decide at compile time whether two conditions may become **true** at the

same event instant or not. Therefore, we make the conservative assumption that *all equations* in a model may be activated at the same time instant during a simulation. Due to this assumption, the *total* number of (continuous and discrete) equations shall be identical to the number of unknown variables. This requirement is called the *single assignment rule*. Application of this rule to the above example results in an error message, because there are *two* equations to compute the single unknown variable `close` which leads potentially to conflicting requirements.

In order to rewrite this model, it is necessary to first give a more complete definition of the **when** clause:

```
when {condition1, condition2, ..., conditionN} then
    ...
end when
```

is identical to the following if clause

```
if edge(condition1) or edge(condition2) ... or edge(conditionN) then
    ...
end if

edge(condition) := condition and not pre(condition)
```

Note, that a **when** clause can be used both in **equation** and **algorithm** sections. With this general form of the **when** clause the example can be rewritten into

```
when {condition1, condition2} then
    close = edge(condition1);
end when
```

which results in *one* equation for the unknown variable. When the two conditions become true at the same event instant, it can be seen that `condition1` has a higher priority than `condition2`, so that the previously conflicting requirements are resolved.

The present definition of a hybrid model is declarative. To perform a simulation, an explicit evaluation sequence has to be determined. By permutation of equations and variables and by symbolically solving equations for the desired unknown variables it is possible to arrive at an explicit forward sequence of assignment statements. This transformation is called BLT-partitioning (= Block Lower Triangular partitioning), see e.g., [Duff86].

Note, that sorting via BLT transformation is performed by assuming that the constants and input signals of the model, the continuous states and the **pre** values are *known*, all other variables are unknown and that all **when** clauses are active. Applying BLT-partitioning to the introductory example of this section results in:

$$
\begin{aligned}
&\mathbf{y} = \mathbf{g}(\mathbf{x}_p)\\
&\textbf{when } \text{sample}(0, T_s) \textbf{ then}\\
&\quad \mathbf{x}_c = \mathbf{A}\cdot\mathbf{pre}(\mathbf{x}_c) + \mathbf{B}\cdot(\mathbf{r}-\mathbf{y})\\
&\quad \mathbf{u} = \mathbf{C}\cdot\mathbf{pre}(\mathbf{x}_c) + \mathbf{D}\cdot(\mathbf{r}-\mathbf{y})\\
&\textbf{end when}\\
&\dot{\mathbf{x}}_p = \mathbf{f}(\mathbf{x}_p, \mathbf{u})
\end{aligned}
$$

Given the continuous states $\mathbf{x}_p$, the reference inputs $\mathbf{r}$ and the previous values of the discrete states $\mathbf{pre}(\mathbf{x}_c)$ it is possible to compute all other unknown variables, both when the equations of the **when** clause are active or when the equations are not active.

BLT-partitioning may identify local *systems of equations* which have to be solved simultaneously, e.g.

$$
\begin{aligned}
&u = \ldots;\\
&\textbf{when } \text{condition1} \textbf{ then}
\end{aligned}
$$

$y = k_1 \cdot x + u$
end when

when condition2 **then**
$x = k_2 \cdot y$
end when

When the two conditions become **true** at the same time instant this leads to the unique solution

$$\begin{aligned} y &:= \frac{1}{1 - k_1 \cdot k_2} u \\ x &:= k_2 \cdot y \end{aligned}$$

However, when only one of the conditions becomes **true**, no unique solutions exists, because only one equation for two unknown variables is present. One could use the previous value of x or y to remove one of the unknowns, but this would lead to a non-deterministic behaviour, because there is no rule which of the two variables should be used and therefore the choice would be arbitrary. As a consequence, we have to reject models containing algebraic loops when the equations of a loop belong to different **when** clauses. In asynchronous languages, this situation corresponds to a *deadlock*, because the two ”when-processes“ refer to each other in a non-resolvable way. In other words, the synchronous principle allows to detect deadlock situations already at compile time. To be more precise, here is the exact rule for ”deadlock“ detection:

> Models are only valid if *all* local algebraic loops in the BLT-form have *one* of the following properties:
>
> - All equations of one loop do *not* belong to a **when** clause.
> - All equations of one loop belong to **when** clauses which have *syntactically* the same when condition.

It is desired to check at compile time whether two **when** clauses *always* occur at the same time instant, in order that the equations of the two **when** clauses can appear in the same algebraic loop. Since such a check cannot be done in general, we take the conservative approach and just identify a subset of the possible ones by requiring that the when conditions have to be syntactically equivalent (e.g. the same Boolean variable). The minor drawback is that there may be cases where the translator reports an error, although a deadlock cannot occur. By simply rewriting the **when** conditions of the corresponding **when** clauses into a syntactically identical form it is possible to fix such a situation.

Above, we discussed systems of equations containing only Real unknown variables. In the following sections, this approach is generalized to situations where Boolean and Integer variables also appear as unknowns.

To summarize, the *synchronous* principle of hybrid systems is based on the following requirements:

1. At every time instant, the active equations express relations between variables which have to be fulfilled *concurrently*.
2. Computation and communication at an event instant does not take time.
3. The total number of equations is identical to the total number of unknown variables and all unknown variables can be uniquely determined from this set of equations (= single assignment rule).
4. The evaluation sequence of the equations is determined by equation sorting (BLT transformation).
5. Local algebraic loops in the BLT-form between equations of different **when** clauses signal a ”deadlock“ situation which has to be rejected since a unique solution potentially does not exist at some time instants.

This approach of handling hybrid systems has the advantage that the ”synchronization“ between the continuous and discrete parts is automatic and always leads to a deterministic behaviour without conflicts. Furthermore, some difficult to detect errors of asynchronous languages, such as deadlock, can often be determined at compile time already.

The disadvantage is that the types of systems which can be modeled is restricted. For example, general Petri nets cannot be modeled because such systems have non-deterministic behaviour. Petri nets can only be described if transition priorities are introduced. For some applications another type of view, such as a process oriented type of view or CSP, may be more appropriate to model a discrete component. It may be more complicated to model such systems with the sychronous approach. Especially, the single assignment rule sometimes leads to more difficult code because explicit priorities of conditions have to be defined. If a variable should be assigned from different components this may require to introduce a special ”priority“ object which handles this situation.

2.2 Relation triggered state and time events

During continuous integration it is required 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 a simple two point controlller with input *u* and output *y* may be described by the following equations:

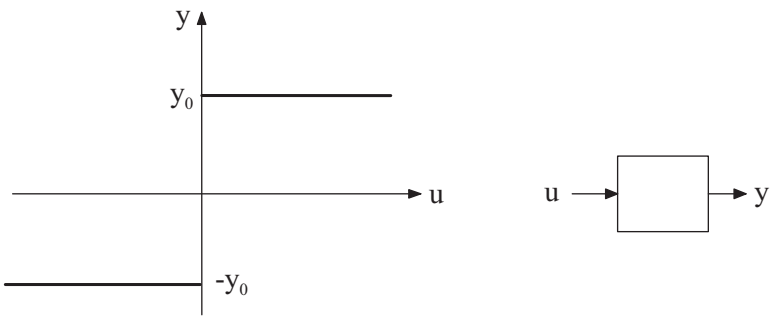

Figure 2: Two point controller.

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

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 handeled in a numerical clean way by detecting the switching point precisely, halting the integration, selecting the corresponding new branch, and restarting the integration, i.e., by triggering a *state event*. This technique was developed by Cellier [Cell79]. For details see also [Eich98], chapter 6.

In general, it is not possible to figure out by source inspection whether a specific relation will lead to a discontinuity or not. Therefore, we take the conservative approach and assume that every relation will introduce a discontinuity or a non-differentiable point in the model. Consequently, relations in Modelica *automatically* trigger state events at the time instants where their value is changed. This means e.g., that model `TwoPoint` is treated in a numerically ”clean“ way.

In some situations, relations do not introduce discontinuites or non-differentiable points. Even if such points are present, their effect may be small, and it may not hurt to just integrate over these points. Finally, there maybe situations where a literal evaluation of a relation is required, since otherwise an ”outside domain“ error occurs, such as in the following example, where the argument of function `sqrt` to compute the square root of its argument is not allowed to be negative:

```
y = if u >= 0 then sqrt(u) else 0;
```

This equation will lead to a run time error, because *u* has to become small and negative before the **then**-branch can be changed to the **else**-branch and the square root of a negative real number has no real result value. In such situations, the experienced modeler may explicitly require a literal evaluation of a relation by using the operator **noEvent**():

```
y = if noEvent(u >= 0) then sqrt(u) else 0;
```

Discrete and nondiscrete variables

Time varying variables in Modelica may either change at event instants *only* or at all time instants, especially during continuous integration. The former variables have to be declared as **`discrete`** and the latter as **`nondiscrete`**. Without explicit declaration, the following default values are assumed:

Base Type	*Variability*
`Real`	**`nondiscrete`**
`Boolean, Integer, String`	**`discrete`**

Therefore in the following example

```
model TestDiscrete
   parameter   Real    y0=1;
   input       Real    u;
               Real    y1, y2, y3;
               Boolean b1, b2;
   nondiscrete Boolean b3;
equation
   b1 = u > 0;             // fine , b1 is discrete
   b2 = noEvent(u >= 0);   // error, b2 is discrete
   b3 = noEvent(u >= 0);   // fine , b3 is nondiscrete

   y1 = if b1 then y0 else -y0;
   y2 = if b2 then sqrt(u) else 0;
   y3 = if b3 then sqrt(u) else 0;
end TestDiscrete
```

It is an error that the **`noEvent`** operator is applied in the relation to compute `b2`, because `b2` is a **`discrete`** variable (= only to be changed at event instants) whereas **`noEvent`**`(u >= 0)` may change its value during continuous integration. Therefore, `b2` has to be explicitly declared as **`nondiscrete`**, as done for `b3`, in order to signal to the translator that the modeller takes responsibility for the introduced discontinuity. Note, that the condition of a **`when`** clause has to be a discrete expression, i.e., it is guaranteed that the equations of the **`when`** body are never evaluated during continuous integration.

Time events

State event detection requires an iteration procedure during simulation. When a relation is just a function of time, the time instant of the event can be determined in advance. This enhances the efficiency considerably,

because no iteration is needed to determine the event instant. Therefore, it is important to handle this special case differently: In Modelica, the modeler can expect that the following special relations are treated as time events[1]:

```
time >= discrete expression
time <  discrete expression
```

The left variable of the relation needs to be the predefined variable *time*, whereas the right side of the relation needs to be an expression which evaluates to a new value at most at event instants. This restriction is important, because only then it can be guaranteed that the value of the next time instant does not change during continuous integration. This is a prerequisite, because the next time event must be known before the integration is restarted. Example:

```
   ...
   parameter Real sampleTime = 0.1;
   discrete  Real nextTime(start = 0);
equation
   ...
   when time >= pre(nextTime) then    // sampled data system
      nextTime = time + sampleTime;
        ...
   end when;
   ...
```

Since `nextTime` is declared as **`discrete`** `Real`, the Modelica translator knows that the value of this variable can only be changed at event instants and therefore the expression `time >= pre(nextTime)` can be transformed into a time event. Note, that the previously introduced **sample** operator is an alternative to the construction of a sample data system above. For reasons which will become clear in the next section, the **sample** operator is slightly more efficient.

2.3 Left and right limit of a variable (operator pre)

In the previous sections, the **pre** operator was introduced informally as access operator for the previous value of a variable. In this section, the exact semantics of this operator is discussed. In particular, the **pre** value of a Real, Integer or Boolean variable y is defined to be the *left limit* and y is defined to be the *right limit* at a time instant t:

$$\mathbf{pre}(y) \equiv y(t^-) \tag{2.2a}$$

$$y \equiv y(t^+) \tag{2.2b}$$

It follows that $\mathbf{pre}(y) = y$ if $y(t)$ is *continuous* at time instant t. As a consequence, during continuous integration $\mathbf{pre}(y) = y$ and only at event instants we may have $\mathbf{pre}(y) \neq y$. This definition has several consequences:

1. The return result of the **pre** operator is treated as a *known* variable which may be used everywhere where a known variable is allowed, especially in **when** clauses and in continuous equations. At the start of the integration $y = \mathbf{pre}(y) = y.start$.

2. At event instants, the whole model is evaluated one time. Before the event restart, all **pre** values are updated in the form

   $$\mathbf{pre}(y) := y \tag{2.3}$$

   in order that during continuous integration no variable is discontinuous.

[1] To be precise, it is a "quality of implementation" of the Modelica translator whether these relations are transformed to time events. However, translators without this proporty will produce unnecessarily unefficient code.

3. If the **pre** operator is used outside of a **when** body in a discrete equation or in the condition of a **when** clause, potentially a discontinuity is introduced after the event restart, as sketched in the following example:

```
      ...
   off := s < 0 or pre(off) and not fire;
der(x) := if off then -x else -2*x;

if event() then // implicitly present at end of code
   pre(off) := off
end if
```

   At an event instant `pre(off)` may be `false` at the beginning of the event and `off` may become `true`. Since `pre(off)` is updated before the event restart, `pre(off)` changes from `false` to `true`. The integrator will make a first step. At the next time instant `off` is recalculated. Since `pre(off)` is no longer `false`, another value for `off` may be computed and the value of the derivative `der`(x) may change discontinuously.

   To avoid such situations, an iteration has to take place at an event instant as long as `pre` variables in equations outside of `when` bodies change. In other words, the above example is actually realized in the following way:

```
loop
      ...
   off := s < 0 or pre(off) and not fire;
der(x) := if off then -x else -2*x;

if event() then // implicitly present
   if off == pre(off) then break;
   pre(off) := off
else
   break;
end if
end loop
```

2.4 Event synchronization

Event synchronization means that some `when`-bodies, probably present in different components, shall be evaluated at the same event instant. In simple cases, event synchronization is realized by just using the same variable names in the corresponding models and connectors, e.g.:

```
  Boolean sampleEvent;
equation
  sampleEvent = sample(0,2);   // sample every 2 seconds
    ...
  when sampleEvent then  // set of equations 1
      ...
  end when;
    ...
  when sampleEvent then  // set of equations 2
      ...
  end when;
```

In Modelica there is no guarantee that two *different* events occur at the same time instant, e.g.:

```
  fastSample = sample(0,1);
  slowSample = sample(0,5);
```

In exact arithmetic, Boolean variables `fastSample` and `slowSample` are **`true`** at the *same* event instant every five seconds. However, in Modelica this behaviour is not guaranteed. If such a property is desired, the synchronization has to be *explicitly* modeled, e.g., by using counters as in the following example:

```
    Boolean fastSample, slowSample;
    Integer ticks(start=0);
equation
    // define fastest sampling rate
       fastSample = sample(0,1);

    // define 5-times slower sampling rate
       when fastSample then
          ticks      = if pre(ticks) < 5 then pre(ticks)+1 else 0;
          slowSample = pre(ticks) == 0;
       end when;

    // define equations for the different sampling rates
       when fastSample then   // fast sampling
          ...
       end when;

       when slowSample then   // slow sampling (5-times slower)
          ...
       end when;
```

The `slowSample` **`when`** clause is evaluated at every 5th occurrence of the `fastSample` **`when`** clause by using the counter `ticks` to count how often the `fastSample` event occured.

Modelica does not have an *explicit* event *type*, although in some rare situations this may be useful. For example, pushing a button of a hardware device may be abstracted in such a way that a Boolean variable becomes **`true`** at the time instant when it is pressed, and is **`false`** afterwards, until it is pressed again. It would be most convenient, if a separate data type would be available for such cases. E.g. in [Sree91] a complete discrete formalism is developed based on two data types: *conditions* (= Boolean variables) and *events* (= Boolean variables which signal the occurence of an event and which are **`true`** only at event instants). In Modelica, an event data type can be emulated by a Boolean variable where every change in its value signals an event:

```
    input Boolean b;      // set event via "b = not pre(b)"
    Boolean ev;           // true only at event instants
equation
    // Alternative 1
       when {b, not b} then  // action when event "b" occurs
          ...
       end when;

    // Alternative 2
       ev = edge(b) or edge(not b);  // define event "ev"
         ...
       when ev then  // action when event "ev" occurs
          ...
       end when;
```

Usually an event occurs when a condition, such as a Boolean variable, becomes **`true`**. As can be seen, it is easy to trigger an event also in cases whenever the Boolean variable changes its value (as for `b`). Alternatively, a Boolean variable can be defined in such a way, that it is only true at event instants, see the definition of variable `ev`.

2.5 Reinitialization of continuous states (operator reinit)

At event instants, a continuous *state* `x` can be reinitialized before the restart using the built-in operator

```
reinit(x, expr);
```

to introduce a new equation

```
x = expr;
```

where `x` is the new value of the state (= right limit $x(t^+)$ of the variable) and `expr` is an expression to compute this new value. For example, in the following model

```
block PT1reset
   parameter Real T      "time constant";
   parameter Real k      "gain";
   input  Boolean reset  "reset state, when true";
   input  Real    u;
   output Real    y;
   Real           x      "state of block";
equation
   der(x) = (u - x) / T;
       y  = k*x;

   when reset then
      reinit(x, 0.0);
   end when;
end PT1;
```

the first order block `PT1reset` has an additional Boolean input signal `reset` to reset the state of this block to zero, whenever `reset` becomes **`true`**.

On first view, the **`reinit`** operator breaks the single assignment rule because a new equation is introduced but no new unknown variable. This would indeed be the case if just the equation

```
x = expr;
```

is added. However, the operator has the additional semantics that the previously *known* state variable `x` is treated as *unknown* for sorting. Therefore, BLT-partitioning sorts the equations in such a way that this additional equation is placed before any other equation utilizing `x`. This feature allows for example to realize *self-initializing* blocks:

```
block PT1 "self-initializing first order filter"
  parameter Real T "time constant";
  parameter Real k "gain";
  input     Real u;
  output    Real y;
protected
  Real x;
equation
  der(x) = (u - x) / T;
      y  = k*x;
```

```
  when initial() then
    reinit(x, u);   // initialize, such that der(x) = 0.
  end when;
end PT1;

model TestPT1   // initialize all elements to stationary conditions
  input u;
  PT1 b1, b2, b3;
equation
  b1.u = u;
  b1.y = b2.u;
  b2.y = b3.u;
end TestPT1;
```

The built-in operator **`initial`**() is **`true`** at the initial time. Therefore, state `x` of model block `PT1` is initialized such that the internal state `x` is identical to the input signal `u` at the initial time, leading to a vanishing first derivative of `x`. In other words, this block is initialized in a stationary equilibrium condition, independently of the actual input signal. If several of such blocks are connected together in series, as shown in model `TestPT1` above, all of them are initialized in such a way[2]. A Modelica translator would transform this system in the following evaluation sequence:

```
model TestPT1
  input u;
  PT1 b1, b2, b3;
algorithm
  b1.u := u;
  when initial() then
     b1.x := b1.u;
  end when;
  der(b1.x) := (b1.u - b1.x) / b1.T;
  b1.y := b1.k*b1.x;

  b2.u := b1.y;
  when initial() then
    b2.x := b2.u;
  end when;
  der(b2.x) := (b2.u - b2.x) / b2.T;
  b2.y := b2.k*b2.x;

  b3.u := b2.y;
  when initial() then
    b3.x := b3.u;
  end when;
  der(b3.x) := (b3.u - b3.x) / b3.T;
  b3.y := b3.k*b3.x;
end TestPT1;
```

The **`reinit`** operator can also be used to model *impulses*. For example in figure 3 a ball is shown which falls down under the influence of gravity and collides with the ground. The collision is approximated by an impulse according to Newton's law, such that the rebound velocity changes sign and is proportional to the collision velocity with the coefficient of restitution as proportionality factor:

```
model BouncingBall1
```

[2]Note, that in Modelica 1.0 it was not possible to define such a behaviour.

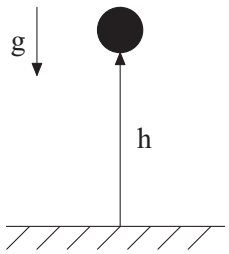

Figure 3: Bouncing ball.

```
   parameter Real e(min=0,max=1) = 0.5  "coefficient of restitution";
   parameter Real g=9.81                "gravity constant";
             Real h                     "height";
             Real v                     "velocity";
equation
   der(h) = v;
   der(v) = -g;

   when h <= 0 then
      reinit(v, -e*pre(v));
   end when;
end when;
```

When the height becomes zero or negative, the integraion is halted and the velocity is reinitialized to a new value according to Newton's law. Note, that in this situation

$$\mathbf{pre}(v) \equiv v(t_h^-) \tag{2.4a}$$
$$v \equiv v(t_h^+) \tag{2.4b}$$

where t_h is the time instant when h becomes zero or negative, $v(t_h^-)$ is the velocity before the collision takes place and $v(t_h^+)$ is the velocity after the collision impulse occured.

The above model has the disadvantage that it will not always work. Assume for example that there is a perfectly plastic collision, i.e., `e=0`. In this case the rebound velocity will be zero and after the event restart the ball will just continue flying downward through the ground starting with a zero velocity. If `e` $>$ `0`, and the simulation takes long enough, the rebound velocity will become very small. Since `h` is small but negative at the event restart due to the state event detection algorithm, the rebound velocity maybe too low in order to lift the ball over the ground (`h` $\geq$ `0`). Also in this case, the ball flies downwards through the ground. Obviously, in both cases the model does not work as desired.

For this special case it is possible to reformulate the model in such a way that the mentioned difficulties disappear by switching between two different modes of the system (= free flight or laying on ground):

```
model BouncingBall2
   parameter Real e(min=0,max=1) = 0.5  "coefficient of restitution";
   parameter Real g     =9.81;          "gravity constant";
   parameter Real vSmall=0.00001;       "minimal rebound velocity";
             Real h                     "height";
             Real v                     "velocity";
   Boolean flying(start=h.start > 0)    "mode of system";
equation
   if pre(flying) then
      der(h) =  v;
      der(v) = -g;
   else
```

```
      der(h) = 0;
      der(v) = 0;
   end if

   when h < 0 then
      reinit(h, 0.0);
      reinit(v, -e*pre(v));
      flying = v > vSmall;
   end when;
```

In this model the height is reinitialized to zero after the collision. Therefore it is guaranteed that a new event is triggered when h becomes negative afterwards and it is never possible that the ball flies through the ground. When the rebound velocity becomes very small, a lot of events will be generated. To prevent this situation, it is assumed that the ball remains laying on the ground forever once the rebound velocity becomes smaller as a certain margin.

2.6 Summary of hybrid operators

Built-in operators of Modelica have the same syntax as a function call. However, they do not behave as a mathematical function, because the result depends not only on the input arguments but also on the status of the simulation. In table 1 all the special operators used for hybrid modeling are collected. Some of them have been already discussed in detail in the previous subsections.

Open issues

In some cases resources, such as memory or connections to physical ports, have to be allocated before a simulation starts and have to be freed when the simulation terminates, both for a regular and an erroneous stop. It is not yet clear how this is handeled in Modelica.

3 Relationship to other synchronous languages

In this section a more detailed discussion of the synchronous languages Lustre [Halb91] and Signal [Gaut94] is carried out in order to point out the relationship with the Modelica hybrid model.

Lustre and Signal are both synchronous data flow languages and have the single assignment principle. Contrary to Modelica, they are designed to model discrete systems only. As pointed out in the Lustre paper: ”... sequencing and synchronization constraints arise from data dependencies.“ Furthermore, it might be interesting to note:

> ”This is a nice feature which allows natural derivation of parallel implementations.“

In Lustre, any variable denotes a flow, i.e. a pair of

- a possible infinite sequence of values
- 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 operates on variables and expressions sharing the same clock.

Lustre has some "temporal" operators like:

- **pre** (previous). **pre**(e) gives a sequence which is obtained by shifting the values of e one ”clock step“.
- $->$ (followed by) is used to give initial values. The program

Operator	*Meaning of operator*
initial()	Returns **true** at the initial time instant.
terminal()	Returns **true** at a *regular* halt of the simulation, but not in case of an error stop.
noEvent(expr)	Real elementary relations within expr are taken literally, i.e., no state or time event is triggered.
sample(start,interval)	Returns **true** and triggers time events at time instants "start + $i\cdot$ interval $(i = 0, 1, \ldots)$. During continuous integration the operator returns always **false**. The starting time "start" and the sample interval "interval" need to be parameter expressions and need to be a subtype of Real or Integer.
pre(y)	Returns the left limit $y(t^{pre})$ of variable $y(t)$ at a time instant t. At an event instant, $y(t^{pre})$ is the value of y after the last event iteration at time instant t. The **pre** operator can be applied if the following three conditions are fulfilled simultaneously: (a) variable y is a subtype of Boolean, Integer or Real, (b) the operator is applied in a **when** body or y is declared as **discrete**, (c) the operator is not applied in a function class. At the initial time **pre**`(y) = y.start`, i.e., the left limit of y is identical to the start value.
edge(b)	Returns "b **and not pre**(b)" for `Boolean` variable b. The same restrictions as for the pre operator apply.
reinit(x,expr)	Reinitializes state variable x with "expr" at an event instant. Argument x need to be (a) a subtype of Real and (b) the **der**-operator need to be applied to it. "expr" need to be an Integer or Real expression. The **reinit** operator can only be applied once for the same variable x.
abs(v)	Is expanded into "(**if** $v >= 0$ **then** v **else** $-v$)". Argument v needs to be an Integer or Real expression. Note, outside of a **when** clause state events are triggered.
sign(v)	Is expanded into "(**if** $v > 0$ **then** 1 **else if** $v < 0$ **then** -1 **else** 0)". Argument v needs to be an Integer or Real expression. Note, outside of a **when** clause state events are triggered.
sqrt(v)	Returns "**if noEvent**($v >= 0$) **then** squareRoot(v) **else** OutsideDomainError". Argument v needs to be an Integer or Real expression.

Table 1: Hybrid Operators.

```
n = 0 -> pre(n) + 1:
```

is thus a counter starting at 0.

- **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.
- **current** interpolates an expression on the clock immediately faster than its own. The interpolation is a "zero order hold".

Signal has corresponding temporal operators:

- "x $ 1" means x delayed one step, i.e. corresponding to **pre**(x).
- **when** as in Lustre.
- "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 ”equations“ with just one variable at the left hand side, i.e. causality is given. However, sorting of the equations is performed.

There are mechanisms in both languages to describe difference equations using a ”delay“ operator. There are sample operators (when) and Lustre has an interpolation operator.

Possible generalization to mixed continuous and discrete 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 inbetween. 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. (SampleTime could be a variable that always contains the next time for sampling.)

Discrete systems

A linear discrete system can be described by difference equations as follows.

$$x(t_{i+1}) = a \cdot x(t_i) + d \cdot u(t_i) \tag{3.5}$$
$$y(t_i) = c \cdot x(t_i) + d \cdot u(t_i) \tag{3.6}$$

or by shifting x one sample interval:

$$x(t_i) = a \cdot x(t_{i-1}) + d \cdot u(t_i) \tag{3.7}$$
$$y(t_i) = c \cdot x(t_{i-1}) + d \cdot u(t_i) \tag{3.8}$$

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

- sampling - u might be continuous and $u(t_i)$ is the sampled signal.
- 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.
- shift operator - Since the state variable is refered to at two different time instants, a shift operator is typically introduced. Forward shift is typically denoted z and backward shift q^{-1}, see e.g., [Astr90].

Example

Using these features we could write a discrete system 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);
```

Instead of using a **when**-operator, we could use a function, for example called **sample**, instead:

```
Sample = Time >= pre(SampleTime);
SampleTime = sample(Time + DT, Sample);

x = a*pre(x) + b*sample(u, Sample);
y = c*pre(x) + d*sample(u, 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**, instead of both **pre** and **sample**:

```
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. It might then be convenient to introduce some grouping mechanism where the sampling condition is only mentioned once, for example:

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

or using the dual **new**-operator instead of the **pre**-operator.

```
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 such a case it is avoided using dot-notation everywhere.

One can view the equations in two ways:

- the equality sign is always valid and sampling of all signals is done when the condition becomes true.
- the equality sign is only valid when the condition becomes true (instantaneous equation). This corresponds to the reality in a computer control algorithm which is 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 in an equivalent model using the **new** operator. However, the **pre** operator is more convenient to use:

- A discrete variable v is always an unknown variable, whereas **pre**(v) is always known. If the **new** operator technique is used, it is not known whether v is a known or an unknown variable. Usually, v is an unknown variable (= the right limit). However, if **new**(v) is used somewhere else in the model, v is a known variable (= the left limit). Understanding an unknown model becomes therefore more difficult.

- It is possible to write: "`edgeb = b` **`and not pre`**`(b)`", i.e., `edgeb` is **`true`** when the Boolean variable `b` becomes true. It is not possible to write something similar with the **new** operator technique, because (a) one has to know whether **new**`(edgeb)` is used somewhere else in the model and (b) it can only be applied for Boolean variables `b` where the **new** operator is applied (`edgeb =` **`new`**`(b)` **`and not`** `b`), but *not* for *all* Boolean variables.
- When connecting two variables from different components, say `c1.b` from component `c1` and `c2.b` from component `c2` an equation is generated "`c1.b = c2.b`". Assume that `c1.b` is defined by an equation in `c1`. In this case, both the right limit `c1.b` as well as the left limit **pre**`(c1.b)` can be accessed in component `c2` via `c2.b` and **pre**`(c2.b)`. With the **new** operator technique this is not directly possible, because **new**`(c2.b)` characterizes the right limit of a variable whereas `c2.b` is the left limit and in equations generated from the **connect** statement, it is not possible to use the **new** operator. As a result, it is *not* possible to report both the left and the right limit of a variable from one component to the next one via a simple connection.

4 Methods to handle variable structure systems

In this section the most important techniques to handle variable structure systems are sketched at hand of the simple example of a rectifier circuit in figure 4. This system is difficult to treat because the diode shall be modeled as an *ideal switch*.

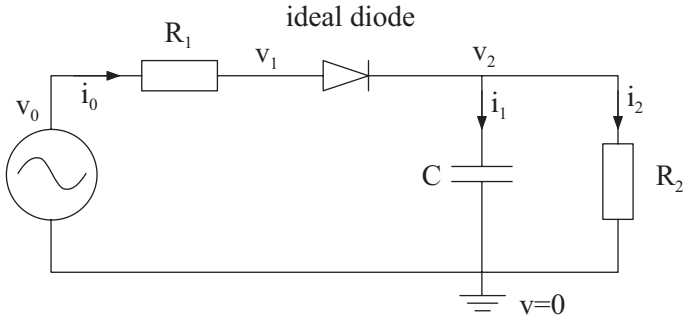

Figure 4: Simple rectifier circuit.

4.1 Variable structure equations and finite automata

In some modeling systems, such as Dymola [Dymo98, Elmq93] and gPROMS [gPRO98, Bart92], variable structure systems are specified with *variable structure equations* which are controlled by *finite automata* to describe the switching behaviour. Basically, this is also the technique used in Modelica 1.0. In [Most96] generalizations are given in case impulsive components are present. The characteristic of an ideal diode is shown in figure 5. The switching structure of the diode is described by the state machine in the right part of the figure: When the diode is in its **off** state, the current is zero. The diode switches into its **on** state, when the voltage drop v becomes positive. In the **on** state the voltage drop is zero. The diode switches back to the **off** state when the current becomes negative. In other words, the diode is described by the following variable structure equation

$$0 = \textbf{if } \text{off } \textbf{then } i \textbf{ else } v \tag{4.9}$$

which is controlled by a state machine via the Boolean variable **off**. With the technique described in [Most98], state machines can be modeled in Modelica by using Boolean equations to model the state machine components. Essentially, this means that an ideal diode is described by the following Modelica model:

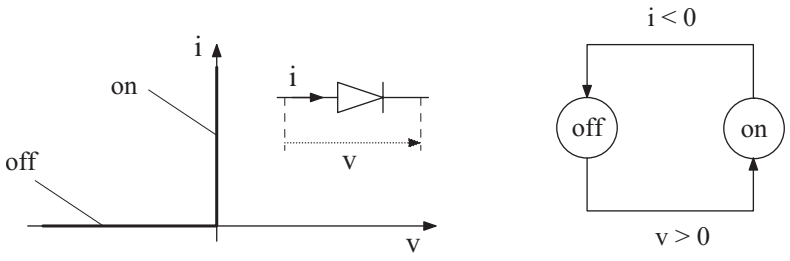

Figure 5: Ideal diode.

```
model IdealDiode
   extends TwoPin;
   Boolean off;
equation
   // variable structure equation
      0 = if pre(off) then i else v

   // state machine (can be made nicer by hiding boolean
   // equations in appropriatly defined model classes)
      off = pre(off) and v<=0 or not pre(off) and i<0
end IdealDiode;
```

where the following auxiliary classes are used:

```
connector Pin
   Voltage      v;
   flow Current i;
end Pin;

partial model TwoPin
   Pin p, n;
protected
   Current i;  // current flowing from p to n pin
   Voltage v;  // voltage drop between p and n pin
equation
   p.i =  i;
   n.i = -i;
   v   = p.v - n.v;
end TwoPin;
```

This diode model is used to describe the circuit of figure 4. The Modelica model is flattened and the equations are sorted. This results in the following state space model of the circuit. For notational convenience, the variable names of figure 4 are used, instead of the Modelica names, e.g. $i_0 = Diode.i$. Note, that the voltage $v_0(t)$ of the voltage source is a known input function and that the voltage drop v_2 of the capacitor is the system state which is assumed to be known:

$$\begin{aligned} R_1 \cdot i_0 &= v_0(t) - v_1 \\ v_D &= v_1 - v_2 \\ 0 &= \textbf{if } \textbf{pre}(\text{off}) \textbf{ then } i_0 \textbf{ else } v_D \end{aligned}$$

$$\begin{aligned} i_2 &:= v_2/R_2 \\ i_1 &:= i_0 - i_2 \\ \frac{dv_2}{dt} &:= i_1/C \\ \text{off} &:= \textbf{pre}(\text{off}) \textbf{ and } v_D <= 0 \textbf{ or not pre}(\text{off}) \textbf{ and } i_0 < 0 \end{aligned} \tag{4.10}$$

The first three equations are coupled and must be solved simultaneously as a linear system of equations with the unknowns i_0, v_1, v_D. Since **pre**(off) is always known, there are no difficulties to solve this algebraic loop. Afterwards, the next assignments can be evaluated to compute the derivative of the state variable $\dot{v}_2$. Only at an event instant, the last (Boolean) assignment statement is executed.

An event occurs, if one of the relations ($v_D <= 0$ or $i_0 < 0$) changes its value. At an event instant, a new value of "off" is computed based on the previous values **pre**(off), v_D and i_0. This means that the complete model equations have to be again evaluated with "**pre**(off) = off". The Modelica semantics defines that this iteration continuous until none of the **pre**-operators changes any longer (see also section 2.3). In other words, the following loop is implicitly present:

```
loop
    -- equations
    if off == pre(off) then exit loop
    pre(off) := off
end loop
```

Afterwards the integration is restarted. If several ideal diodes are within an algebraic loop, several iterations may be necessary until the correct switching status is found. This approach can be seen as a fixed point iteration scheme to determine new consistent initial conditions.

This approach has the advantage, that quite general systems can be described and that the switching structure is clearly displayed in the finite automaton, see e.g. figure 5. The essential disadvantage is that the finite automaton is an *algorithm* which describes exactly in which way the iteration has to take place to determine a new consistent state. This has several undesirable effects:

- The continuous components are described in a *declarative* way based on equations. Idealized physical components such as ideal switches or diodes are acausal by nature and should therefore completely described in a declarative way and not partly in a functional style.
- The fixpoint iteration scheme does not always converge. Since the algorithm is built into the model, it is not possible to use other algorithms which may be more efficient and/or more reliable to find a new consistent state.
- The fixpoint iteration scheme is always done over the *complete* model equations, although only part of the equations may be affected for finding the new consistent state, as it is also the case in the example above. For bigger models the efficiency is therefore unnecessarily reduced.
- In some rare cases, where the *consistent state* depends on the model status before the event occured (this is e.g. the case for an ideal thyristor, see section 5.2), the fixpoint iteration scheme may converge, but to a non-physical solution, because the model status before the event occured, is no longer known, after one iteration took place. In such a case it is not possible to describe the problem with local automata for every component. Instead a global automaton is needed which depends on all coupled ideal elements. However, such a solution is not practical.

4.2 Complementarity formulation

A completely different approach was developed by Lötstedt in a series of articles, e.g. [Loet82], and further improved by several other reseachers, see especially the book by Pfeiffer and Glocker [Pfei96]. The central idea is to transform the model equations into the following *linear complementarity problem*:

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}; \quad \mathbf{y} \geq 0; \quad \mathbf{x} \geq 0; \quad \mathbf{y}^T\mathbf{x} = 0 \tag{4.11}$$

(4.11) is a linear system of equations in the unknowns $\mathbf{x}$ and $\mathbf{y}$. By the additional conditions it is required that all unknowns are not negative and that for every pair (x_i, y_i) at least one of the two elements is zero.

Several algorithms exist to determine the solution of (4.11), most important the algorithm of Lehmke, see [Murt88] for details. The ideal diode can be described in a *declarative* way by complementarity equations:

$$i \geq 0; \quad (-v) \geq 0; \quad i \cdot (-v) = 0 \tag{4.12}$$

These equations just state the ideal diode characteristic in the left part of figure 5. By replacing the diode equations of (4.10) with (4.12) and by eliminating variable v_1 from the linear system of equations, (4.10) can be transformed into the following sorted model equations:

$$\begin{aligned} -v_D &= R_1 \cdot i_0 + v_0 - v_2 \\ & \quad i_0 \geq 0; \quad (-v_D) \geq 0; \quad i_0 \cdot (-v_D) = 0 \\ i_2 &:= v_2 / R_2 \\ i_1 &:= i_0 - i_2 \\ \frac{dv_2}{dt} &:= i_1 / C \end{aligned} \tag{4.13}$$

The first two lines define a linear complementarity problem with

$$\mathbf{y} = -v_d; \quad \mathbf{x} = i_0; \quad \mathbf{A} = R_1; \quad \mathbf{b} = v_0 - v_2$$

The state derivative $\dot{v}_2$ is computed by the remaining three assignment statements. During continuous integration either v_D or i_0 is zero and the other variable is computed from the first equation. An event occurs, if one of the relations ($i_0 \geq 0$ or $(-v_D) \geq 0$) changes its value. At an event instant, the linear complementarity problem is solved with one of the available algorithms, e.g. with the algorithm of Lehmke.

This approach has the advantage that the ideal elements are described in a *declarative* way. As a consequence, several algorithms can be used to solve the consistent reinitizalization problem. Furthermore, only the minimal set of equations are involved when solving the problem, and no longer the whole set of model equations. The disadvantages of this approach are:

- It can be difficult to transform an ideal component into the required complementarity form. Especially, it is non-trivial to describe friction, see [Pfei96]. Furthermore, it is not known how to describe (a) an ideal thyristor or (b) Coulomb friction where the maximum static friction force is bigger (and not identical) to the sliding friction force at zero velocity.
- It seems to be difficult to transform a model given as a connected set of local components *automatically* into the standard complementarity form (4.11).

These disadvantages presently prevent the usage of this method in an object-oriented modeling language, such as Modelica.

4.3 Parameterized curve descriptions

It is natural to ask why the ideal diode of figure 5 is difficult to describe mathematically. The answer simply is that it is only difficult, as long as we restrict ourselves to use an $y = f(x)$ description of the diode characteristic. However, curves can also be defined in a parameterized form as

$$\begin{aligned} y &= f(s) \\ x &= g(s) \end{aligned}$$

using a curve parameter s. This description form is more general and allows us also to describe an ideal diode, see figure 6:

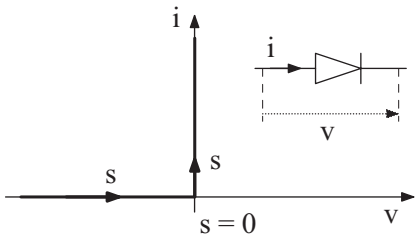

Figure 6: Ideal diode described as parameterized curve.

$$\begin{aligned} v &= \textbf{if } s < 0 \textbf{ then } s \textbf{ else } 0 \\ i &= \textbf{if } s < 0 \textbf{ then } 0 \textbf{ else } s \end{aligned} \tag{4.14}$$

(4.14) is a unique, mathematical description of the ideal diode (6) and does not contain any algorithm in it.

This technique was introduced in [Clau95] and a series of related papers. However, no proposal was given how to actually implement parameterized curve descriptions in a numerically sound way. In Modelica, this is the standard way to realize ideal models. Below, a *new* technique is introduced how to simulate such types of systems in a numerically clean way.

According to (4.14), a declarative model of an ideal diode can be defined in Modelica as:

```
model IdealDiode
   extends TwoPin;
   Real    s;
   Boolean off;
equation
   off = s < 0;
   v   = if off then s else 0;
   i   = if off then 0 else s;
end IdealDiode;
```

This model is easier to understand as the previous two versions, in particular if it comes with a plot of the ideal diode characteristic with the indicated s-parameterization.

Using the above diode model for the rectifier circuit of figure 4 leads to the following sorted set of equations determined by BLT partitioning (remember that $v_0(t)$ is a known input function and v_2 is a state which is assumed to be known):

$$\begin{aligned} \text{off} &= s < 0 \\ v_D &= v_1 - v_2 \\ v_D &= \textbf{if off } \textbf{then } s \textbf{ else } 0 \\ i_0 &= \textbf{if off } \textbf{then } 0 \textbf{ else } s \\ R_1 \cdot i_0 &= v_0(t) - v_1 \end{aligned} \tag{4.15}$$

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

The first 5 equations are coupled and build a system of equations in the 5 unknowns $\text{off}, s, v_D, v_1, i_0$. The remaining assignment statements are again 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 build a linear system of equations in the 4 unknowns s, v_D, v_1, i_0 which can be solved by Gaussian elemination. 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

Boolean and continuous equations which has to be solved. It turns out that several meaningful algorithms can be constructed to solve such systems of equations. This is the topic of section 6 on page 30. Similiarily to the solution by a transformation into a linear complementarity problem, we end up with a local set of equations which has to be solved at event instants. This new approach is advantageous because

- it is natural to describe physical components in a parameterized form,
- it is simple to transform an object-oriented model *automatically* into a standard form of a mixed set of continuous/discrete equations, where the unknows are of type Boolean, Integer, Real.
- several meaningful algorithms can be constructed to solve this mixed set of equations. These algorithms are more efficient and at least as reliable as the fixed point iteration scheme of the first approach (variable structure equations + finite automata).

Since finite automata can be expressed in form of Boolean equations [Most98], the first approach (variable structure equations + finite automata) can be treated as a special case of the new method. Therefore, the proposal just adds new functionality for simpler, more efficient and more reliable treatment of variable structure systems without introducing restrictions. Clearly, it allows to describe more general systems as with the complementarity formulation.

As a side effect, an algebraic system of equations may contain *only* Boolean or other discrete equations, e.g. discrete algebraic equations from different controllers which form an algebraic loop. Probably, this feature is advantageous. This has to be analysed in more detail (not yet done).

5 Examples for parameterized curve descriptions

In this section several additional examples are given to describe variable structure systems with parameterized curves leading to mixed systems of continuous/discrete equations.

5.1 Hysteresis

In figure 7 the characteristic of a simple *hysteresis* block is shown where an input u results in an output $y = +/-y_0$. In the range $-1 < u < +1$ two possible values for y exist for a given u value. The actual value is chosen based on the *branch* of the *previous* solution. A corresponding Modelica model has the following structure:

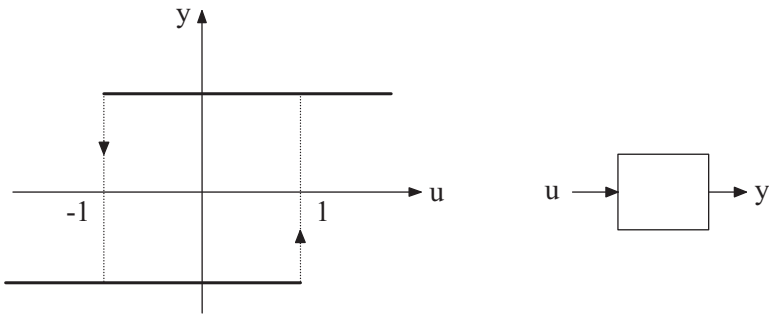

Figure 7: Simple hysteresis model.

```
block Hysteresis
   parameter Real y0=1;
   input     Real u;
   output    Real y;
```

```
protected
   Boolean high (start = u.start > -1);
equation
   high = u >= 1 or pre(high) and u > -1;
   y    = if high then y0 else -y0;
end Hysteresis;
```

If Boolean variable `high` is **true**, the upper branch of the hysteresis block is used. This is the case if either u is bigger as one or if the upper branch was used since the last event instant and u is greater than -1.

This block is not described as a parametrized curve. It was discussed to introduce the technique of a branch selection mode, which will be needed for the next elements.

5.2 Ideal thyristor

The characteristic of an *ideal thyristor* is shown in figure 8. It is similar to an ideal diode with the difference that the voltage drop v may also be positive and that the switch closes only if v is positive *and* the additional Boolean input signal `fire` is **true**. With regards to an ideal diode model a complication arises, because

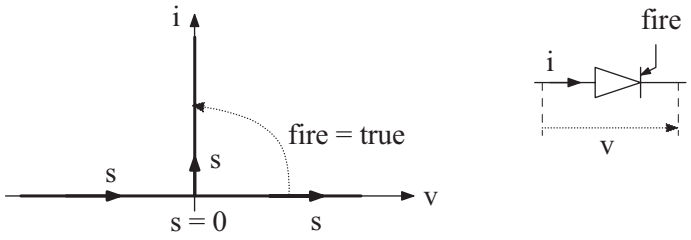

Figure 8: Ideal thyristor model.

the origin $s = 0$ belongs to three branches. If we have the requirement that every neighbouring points on the curve also have values of the path parameter s which are close together, it is not possible to *uniquely* parameterize the whole curve with just *one* curve parameter. Instead, using just one curve parameter leads to two branches of the curve which have identical s-values. By just providing a value of the curve parameter $s > 0$, it is therefore not possible to uniquely define a point on the curve. An additional variable is needed which characterizes the branch of the curve to resolve the ambiguity. Similarily to the hysteresis model of the last chapter, there is the additional requirement that the thyristor should stay on the branch of the curve where it was previously, if this is possible. These considerations lead to the following Modelica model of an ideal thyristor:

```
model IdealThyristor
   extends TwoPin;
   input Boolean fire;
protected
   Boolean off (start = true);
equation
   off = s < 0 or pre(off) and not fire;
     u = if off then s else 0;
     i = if off then 0 else s;
end IdealThyristor;
```

Variable `off` characterizes the active branch of the thyristor. If s is negative, the branch is uniquely identified. Otherwise, the **pre** branch together with the firing condition determines the actual active branch.

The thyristor model is a good candidate to give additional reasoning why the **pre** operator is more convenient than the **new** operator. Rewriting the model with the **new** operator leads to:

```
new(off) = s < 0 or off and not fire;
       u = if new(off) then s else 0;
       i = if new(off) then 0 else s;
```

The model looks a bit more complicated because the **new** operator is used at several place. Worse, it is not possible to derive this model by inheritance from an ideal diode or an ideal switch model, because the equations of the basic equations to compute *u* and *i* are different. With the **pre** operator this is possible.

5.3 Coulomb Friction

The characteristic of a Coulomb friction element is shown in figure 9 under the assumption that the normal force is constant. In such a case, the (tangential) friction force is uniquely determined as a function of the relative velocity *v*, provided *v* is not zero. If the relative velocity vanishes, the friction element becomes stuck, i.e., the friction force is now a constraint force which is determined by the equilibrium conditions. As will become clear, describing such an element mathematically in a clean way is quite complicated. In

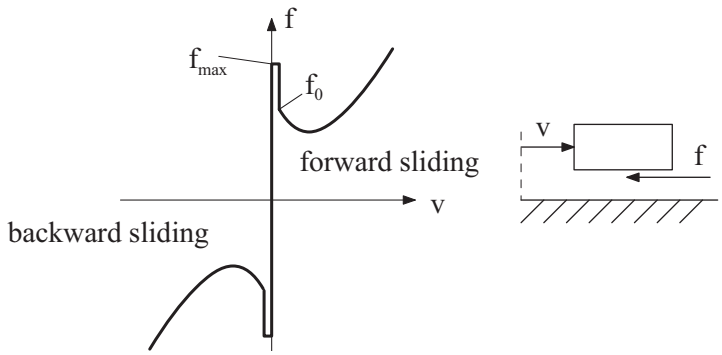

Figure 9: Coulomb friction element.

order to simplify our task, we will temporarily examine the model shown in figure 10, where the friction characteristic in the sliding phase is just a straight line and $f_{max} = f_0$. This model is quite similar to an

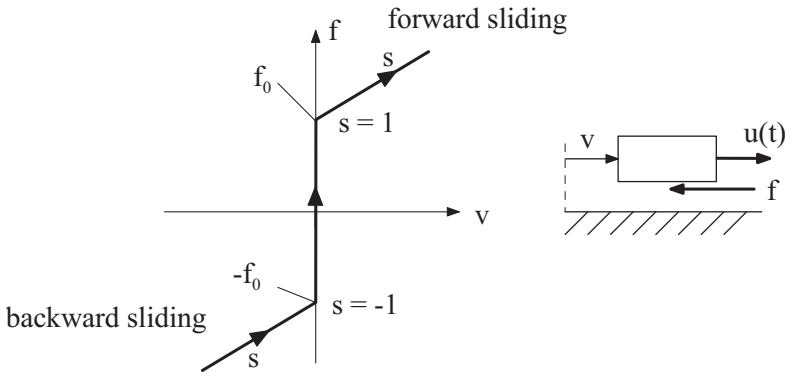

Figure 10: Simplified Coulomb friction element.

ideal diode model and we expect that this element can be described by the following parameterized curve equations:

```
if s > 1 then
   v = s - 1;
   f = f0 + f1*(s-1);
```

```
else if s < -1 then
   v = s + 1;
   f = -f0 + f1*(s+1);

else
   v = 0;
   f = f0*s;
end if
```

This model is correct and completely describes the friction element of figure 10. However, currently we do not know how to transform such a model *automatically* in a form which can be handeled by a numerical integrator.

Let us analyse the difficulties by applying this model to the simple block on a rough surface shown in figure 10 which is described by the following equation:

```
m*der(v) = u - f;
```

Note that m is the mass of the block and $u(t)$ is the given driving force. If the element is in its forward sliding mode, i.e., $s \geq 1$, this model is described by the following equations:

```
m*der(v) = u - f;
v = s - 1;
f = f0 + f1*(s-1);
```

or sorted into a forward sequence:

```
     s := v + 1;
     f := f0 + f1*(s-1);
der(v) := (u - f) / m;
```

Note, that v is the state variable (which is assumed to be known in the model) and $\mathbf{der}(v)$ can be easily computed. If the relation $s > 1$ crosses zero an event occurs, $v == 0$ at the event instant, and the model is described by the following equations:

```
m*der(v) = u - f;
v = 0;
f = f0*s;
```

Since we get an equation for the state variable v, the relative velocity can no longer be used as a state, i.e., we have a higher index system. This requires to differentiate the second equation ones leading to $\mathbf{der}(v) = 0$. Inserting all relationships in the first equation finally results in the following sorted set of equations

```
s := u/f0;
v := 0;
f := f0*s;
```

The problem here is that we have a conditional index change of the model and this situation is difficult to handle because equations need to be differentiated depending on the value of some Boolean variables. However, at least with an interpretative system we could handle this problem.

Unfortunately, there is a more involved difficulty: Assume that `s := u/f0` becomes bigger as one, i.e., an event occurs since $s-1$ has a zero crossing. The model has to switch back to the sliding mode equations. In this mode, v is a state and $s := v+1$ is computed from v. If v would be positive, this is fine. However, in general v is computed by other equations in the stuck mode such that it is zero (here: $v = 0$ is set; in general a system of equations has to be solved, especially if two or more friction elements are dynamically coupled). Since this is a numerical computation, we can only expect that v is small. In particular, v may be small but *negative*, e.g., $v = -10^{-15}$. As a result $s := v+1$ becomes a little bit less than

one and the relation $s-1$ crosses again zero, i.e., the element switches back in the stuck mode. However, in the stuck mode the same situation as before is present which means that the element will again switch in the forward sliding mode. In other words, no (numerical) solution exists to this set of equations.

Assume that the other situation occurs, i.e., v is small and *positive*. Then, it is *never* possible to switch back at this time instant, because in the forward sliding mode v is a (known) state and s is computed via $s = v+1$. Switching back to the stuck mode would require that s becomes less than one which means that v becomes negative. However, v cannot change at this time instant because it is a state. It is necessary to be able to switch forth and back between the different modes at zero velocity in order that one of the algorithms of the next section is able to determine new consistent initial conditions for the event restart. Obviously, this is not possible.

The described difficulties exist for all hybrid models with a conditional index change, e.g., also for an ideal diode which is connected in parallel to a capacitor. It is just particularily critical for the friction element, because even the most simple mechanical model with Coulomb friction already has a conditional change of the index.

As already mentioned, for the friction element we do not know how to resolve the discussed problems in an automatic way. Therefore, we need to be pragmatic and describe the friction element at *zero velocity* via the diagram of figure 11 where the relative acceleration is used for the abscissa axis and a new s-parameterization is introduced. Therefore, at *zero velocity*, friction is described by the following model:

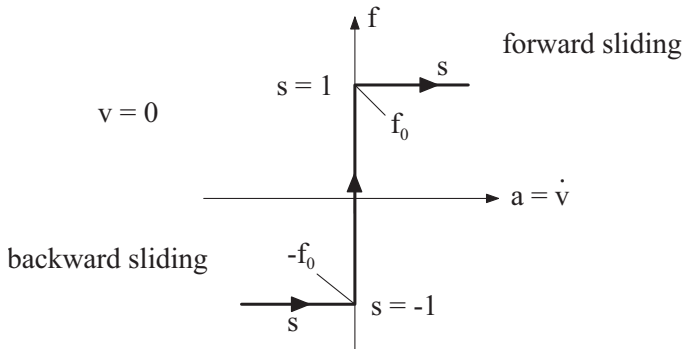

Figure 11: Friction element at zero velocity.

```
a = der(v);
if s > 1 then
   a = s - 1;
   f = f0;
else if s < -1 then
   a = s + 1;
   f = -f0;
else
   a = 0;
   f = f0*s;
end if
```

Assume that the element switches to $s > 1$, i.e., into the forward sliding mode. In this situation v is small, but may have any sign. Since the acceleration is positive ($a = s-1$), after a small time instant, the velocity will become positive, too. If after some further simulation the *velocity* becomes negative, an event occurs, and the above model describes again the behaviour of the element at zero velocity. These considerations lead to the following Modelica model:

```
parameter Real f0, f1;
```

```
   Real           a, v, f, s;
   Boolean        fullSliding(start=true);
equation
    a = der(v);

    if pre(fullSliding) and s > 1  then
       v = s - 1;
       f = f0 + f1*v;
       fullSliding = true;

    else if pre(fullSliding) and s < -1 then
       v = s + 1;
       f = -f0 + f1*v;
       fullSliding = true;

    else if s > 1 then  // not pre(fullSliding)
       a = s - 1;
       f = f0;
       fullSliding = v > 0;

    else if s < -1 then
       a = s + 1;
       f = -f0;
       fullSliding = v < 0;

    else
       a = 0;
       f = f0*s;
       fullSliding = false;
    end mif
```

Similarily to the thyristor model, we need to known the branch where the friction element was in the previous time instant, because we have a non-unique s-parameterization. Here, the branches are distinguished by the Boolean variable `fullSliding`. If `fullSliding` is **`true`**, the diagram according to figure 10 applies. If the variable is **`false`**, the diagram according to figure 11 is used. In other words, the equations for the two diagrams are just combined.

The friction model does not contain an explicit equation $v = 0$ in the locked mode. However, since $a = 0$ and $v(t_e) = 0$ when the integration is started, the relative velocity will remain zero. This well-known trick avoids a difficult problem: If an equation $v = 0$ is added, the number of degrees of freedom is reduced, i.e., the number of state variables is reduced and the set of equations needs to be resorted. If only a constraint equation on acceleration level is added, the number of degrees of freedom does not change and only the zero/non-zero structure of the linear system of equations changes in which the friction equations appear. However, since these linear system of equations are solved numerically at run-time, this does not hurt. The disadvantage of this approach is that it is not fully "clean". Due to a drift-off effect during integration, the relative velocity will not remain zero if the integration time is very long. Practically, this is nearly never a problem, because the simulation time for technical systems is seldomly so long that this drift-off effect becomes visible.

The above model was derived under the assumption that $f_{max} = f_0$. In figure 12 the more general case is shown, where this assumption does not hold. Previously, one Boolean variable `fullSliding` was sufficient to resolve the non-uniqueness of the s-parameterization. This is no longer possible, because at zero velocity in the range $1 \leq |s| \leq f_{max}/f_0 = peak$ there are two additional possible solutions. The following physical rule is used to resolve this non-uniqueness: Assume that the friction element was in the backward sliding mode and that the velocity becomes positive. In this case one has to select the locked

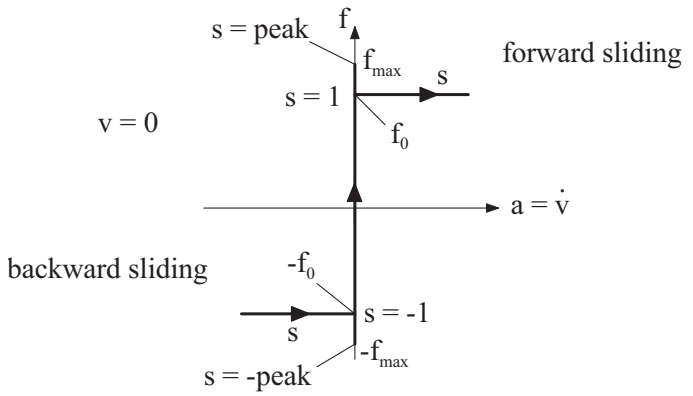

Figure 12: Friction element at zero velocity with $f_{max} \neq f_0$.

mode, if this is possible and not the forward sliding mode, because the friction force has to exceed f_{max} before it can switch to the forward sliding mode. In other words, if the friction element is stuck or in the backward sliding mode, $s > peak$ is required, in order to switch to the forward sliding mode. However, if the element is already in the forward sliding mode, $s > 1$ sufficies to remain in this mode. These considerations finally lead to the following Modelica model of a friction element defined in a *declarative* style:

```
model Friction
   parameter Force f0             "friction force at zero velocity";
   parameter Force f1             "f = f0 + f1*v":
   parameter Real  peak=1 (min=1) "fMax = peak*f0;
   Flange    p, n                 "flanges of element";
protected
   Position     r   "relative position";
   Velocity     v   "relative velocity";
   Acceleration a   "relative acceleration";
   Force        f   "friction force";
   constant Integer Forward  =  2, StartForward  =  1, Stuck = 0,
                    Backward = -2, StartBackward = -1;
   Integer mode;
equation
   // relative quantities
      r = n.r - p.r;
      v = der(r);
      a = der(v);

   // friction law
      if pre(mode) == Forward and s > 1  then
         v = s - 1;
         f = f0 + f1*(s-1);
         mode = Forward;

      else if pre(mode) == Backward and s < -1 then
         v = s + 1;
         f = -f0 + f1*(s+1);
         mode = Backward;

      else if pre(mode) <= Stuck and s > peak or
              pre(mode) >  Stuck and s > 1       then
         a = s - 1
```

```
        f = f0
        mode = if v > 0 then Forward else StartForward;

      else if pre(mode) >= Stuck and s < -peak or
              pre(mode) <  Stuck and s < -1       then
        a = s + 1
        f = -f0
        mode = if v < 0 then Backward else StartBackward;

      else
        a = 0
        f = f0*s
        mode = Stuck;
      end if
end Friction;
```

where the following auxiliary class is used:

```
connector Flange
   Position   r;
   flow Force f;
end Pin;
```

Note, that `Integer` variable `mode` is used to characterize the different non-unique situtations. Non-zero velocity is represented by `mode = Forward` and `mode = Backward`. Here *s* is proportional to the velocity *v*. Zero velocity is characterized by `mode = StartForward`, `Stuck` and `StartBackward`. If `mode = StartForward` or `StartBackward`, *s* is proportional to the acceleration. If `mode = Stuck`, *s* is proportional to the unknown constraint force.

For simplicity, in the above model, just a linear functional relationship of the friction force from the relative velocity is assumed in the sliding phase. In the Modelica library, one should either use a table or a replaceable class to define any type of characteristic.

6 Solution methods for mixed systems of equations

As introduced previously, variable structure elements can be described as parameterized curves which lead to mixed sets of continuous/discrete equations, with unknowns of type Real, Boolean and Integer. By BLT-partitioning these equations are sorted into an explicit forward sequence and the algebraic loops of minimal dimensions (with respect to permutation of equations and variables) are determined. For simplicity, we will assume *temporarily* that all continuous equations in algebraic loops are linear in the continuous variables. If a relation is directly used in an if-expression, it is replaced by a Boolean auxiliary variable together with an explicit assignment of the relation to this variable. As a result, the algebraic loops have the following structure

$$\begin{aligned} \mathbf{y} &:= \mathbf{f}(\text{relation}(\mathbf{x},\mathbf{y}),\mathbf{y}) \\ \mathbf{A}(\mathbf{y})\,\mathbf{x} &= \mathbf{b}(\mathbf{y}) \end{aligned} \tag{6.16}$$

where

1. **y** are *unknown* discrete variables of type `Boolean`, `Integer` or/and `discrete Real`.
2. **x** are *unknown* Real variables.
3. Matrix **A** is square and regular for all values of **y**. Otherwise, an error occurs because in this case no unique solution exists (either no or an infinite number of solutions).

4. **f** is a function of relations of **x**, **y**, e.g. $x_2 > y_1$, and of the unknowns **y**. Example:

$$\begin{aligned} y_1 &:= x_1 > x_2 \\ y_2 &:= y_1 \textbf{ and } x_3 < 0 \end{aligned}$$

   Temporarily we assume that the discrete equations are solvable in an explicit forward sequence, e.g., the following equations are not allowed:

$$\begin{aligned} y_1 &:= x_1 > x_2 \textbf{ or } y_2 \\ y_2 &:= y_1 \textbf{ and } x_3 < 0 \end{aligned}$$

5. The **noEvent**() operator is not applied to any of the relations, in order that no discontinuous change occurs during integration, i.e., that **y** is constant during integration, see also discussion below.

From the construction it is clear that (6.16) can be generated *automatically* from a flat Modelica model. During continuous integration, **y** is fixed and does not change its value. The continuous unknown variables **x** are computed from the linear system of equations ($\mathbf{Ax} = \mathbf{b}$). An event occurs, if one of the relations changes its value, i.e., the relations used in function **f** are used as crossing functions. At an event instant, the complete system of equations (6.16) is utilized to determine the unknown variables **x** and **y**. In the following, several algorithms are discussed to compute such a solution.

6.1 Fixed point iteration

The most simple algorithm is a fix point iteration scheme. This scheme is identical to the one used in the ”variable structure equation + finite state machine“ approach. The essential difference is that the iteration is done locally and not over the complete model equations. As a consequence, it is more efficient, if the algebraic loop does not contain all model equations, which is usually the case:

```
x := x(when event occured)
y := y(from last event)
loop
   last(y) := y
        y  := f(relation(x,y), y)
   if y == last(y) then exit;
   A := A(y)
   b := b(y)
   <solve Ax = b for x>
end loop;
```

The main advantage of this fix point iteration scheme is its simplicity. The experience with the fix point iteration scheme used in Dymola shows that in most practical cases the convergence is fast (within 3 or 4 iterations). However, there is of course no guarantee that the iteration converges.

6.2 Exhaustive search

Since all relations (relation(**x**,**y**)) can take on only a countable set of values, one can try all possible combinations of values, i.e., perform an exhaustive search. This leads to the following algorithm:

```
while <not all values for relation(x) tried>
   lastRelation(x,y) := <next possible value set>
   y := f(lastRelation(x,y),y)
   A := A(y)
   b := b(y)
   <solve Ax = b for x>
   if relation(x,y) == lastRelation(x,y) then exit;
end loop;
```

The main advantage of this algorithm is that it is the *only* algorithm which determines the solution savely after a finite number of steps, if a solution exists, or exits with an error message that no solution exists for the systems of equations. The disadvantage is that the number of possible values for `relation(x,y)` can be huge. E.g. if 10 relations are present, there exist $2^{10} = 1024$ combinations, i.e., in the worst case 1024 iterations (and solutions of the linear system of equations) are needed. In practical cases, this situation is not so worse as it looks like. E.g. often electric power systems have at most 6 coupled electrical switches (diodes or thyristors) which leads to at most $2^6 = 64$ iterations. In some cases, the number of combinations can be reduced, because it is clear that not all combinations can occur, e.g., for the relations $s > 1$ and $s < 1$ of a friction element it is not possible that both relations become **`true`** simultaneously, i.e., there are only 3 instead of 4 possible values.

Note, that the exhaustive search is also the only known algorithm to determine safely the solution of the linear complementarity problem (4.11). Only in special circumstances, e.g. if **A** is symmetric and positive definite, the convergence of the Lehmke and other algorithms to the solution can be guaranteed, see [Murt88].

6.3 Improved fix-point iteration

The fix point iteration scheme can probably be improved by restricting the step-size in such a way that the next solution vector just jumps to the ”nearest“ next interval. For example assume that three relations on a variable *x* are used, such as ”y = if x < -1 then -1 else if x > +1 then +1 else 0”, the current value of x is 2, and after the iteration the value of x is ≤ 1. In such a case, the next iteration starts with a value of x=+1, because this x is at the border of the ”next“ interval. A corresponding algorithm has the following structure:

```
x := x(when event occured)
y := y(from last event)
loop
   last(y) := y
        y  := f(relation(x,y), y)
   if y == last(y) then exit;
   A := A(y)
   b := b(y)
   <solve Az = b for z>
   <determine the biggest stepsize h, with 0 < h <= 1 such that
    x + h*(z - x) does not change relation(x,y)>
   x := x + (h+eps)*(z - x)
end loop;
```

This algorithm has to be analysed more carefully. There is the guess that this algorithm is more robust as the simple fixed point iteration scheme, i.e., that it converges to a solution even if the fixed point iteration scheme does not converge.

6.4 Generalizations

Three algorithms have been presented to solve a mixed set of continuous and discrete equations. It is also possible to combine these algorithms, e.g. to start with a fix point iteration scheme and if this scheme does not converge, say, within 10 steps, the algorithm can be switched to an exhaustive search.

Previously, we assumed that the algebraic loop is linear in the real variables. All algorithms can still be applied, even if this assumption does not hold. The only change is that a nonlinear system of equations has to be solved in every iteration step, instead of a linear one.

Temporarily, we assumed that the discrete variables can be sorted into an explicit forward sequence.

This requirement can be relaxed. E.g., if the following equations are present

$$\begin{aligned} y_1 &:= x_1 > x_2 \textbf{ or } y_2 \\ y_2 &:= y_1 \textbf{ and } x_3 < 0 \end{aligned}$$

one can use the relations ($x_1 > x_2$, $x_3 < 0$) *and* y_2 as iteration variables in the fix point iteration scheme. In other words, y_i values have to be selected as additional iteration variables, in order to get an explicit forward evaluation sequence. Since the y_i variables of type Integer (without minimum and maximum values) as well as discrete Real variables do not have a countable set of values, the exhaustive search algorithm is no longer possible for these kind of systems.

Note, that the implementation of the fix point iteration scheme is not more difficult as previously. The main difference is just that there are local fixed point iterations instead of one global iteration.

7 Event detection

In this section some difficulties are discussed which are related to the detection of event instants.

Integrators provide a basic functionality to detect and stop the integration at the zero crossing of a crossing function (so called "root finding" property). Usually, such integrators have the following properties:

1. A crossing function $z = z(\mathbf{x}(t), t) = z(t)$ has to be defined which is at least continuous during integration.

2. At the start t_0 of the integration it is required that $z(t_0) \neq 0$, in order that a zero crossing can be detected.

3. When a zero crossing of a crossing function occurs, the time instant t_z of the zero crossing point is determined upto a specified precision and the integration is halted at the event instant t_e ($|t_z - t_e| \leq \varepsilon_t$).

4. At the event instant t_e it is guaranteed that the crossing function is identical to zero or has an opposite sign as during integration, i.e., $z(t_0) \cdot z(t_e) \leq 0$.

All these requirements make sense from the integrator point of view in order to provide a mathematically "clean" implementation. However, they complicate the code generation of a Modelica model, e.g., because it must be guaranteed that all crossing functions are not zero at an event restart. In the following it is discussed how relation triggered state events can be mapped to the discussed model of an integrator.

The exact meaning of a Modelica relation "v_1 rel_op v_2" with v_1, v_2 Real variables and rel_op = "$>$, $\geq$, $<$, $\leq$" is given in table 2.

Modelica relation	v_1 rel_op v_2
at an event instant	v_1 rel_op v_2
during integration	v_1 rel_op v_2 from last event

Table 2: Meaning of a Modelica relation.

This means that at an *event instant* a relation is taken literally and that during *continuous integration* the value of the relation computed at the last event instant, i.e., a *fixed value*, is used.

Additionally, every Modelica relation implicitly defines a *crossing function*. This is a bit tricky, due to the proporties of an integrator with root finding option discussed above. First of all, a relation is transformed into one of the standard forms

$$z > 0; \quad z \geq 0; \quad z < 0; \quad z \leq 0 \tag{7.17}$$

E.g. $x_1 > x_2$ is transformed into $z = x_1 - x_2;\ z > 0$. Basically, z is used as crossing function for the integrator. There are two difficulties:

1. A crossing function needs to be non-zero at an event restart, although z may be identical to zero.

2. After a zero crossing took place, the corresponding relation should change its value, although z may be identical to zero at the event instant. If for example z was positiv, a zero crossing of the relation $z \geq 0$ took place, and $z = 0$ at the event instant, the relation does *not* change due to the zero crossing.

Both cases can be handeled by shifting the actual crossing function a little bit depending on the situation. In figure 13 a meaningful strategy is displayed. If z is negative at the event restart, the crossing of the $+\varepsilon$

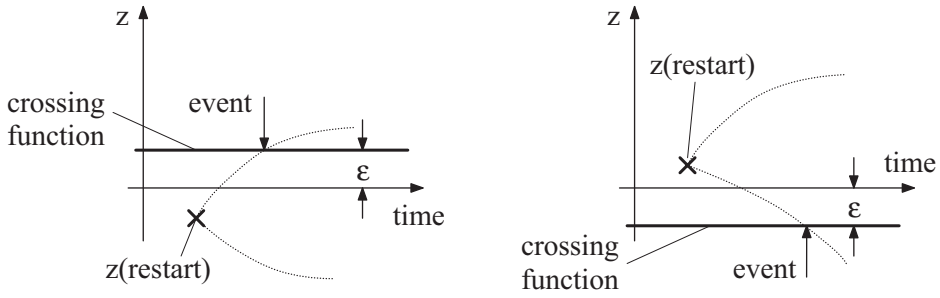

Figure 13: Crossing functions for Modelica relations.

line triggers an event. This ensures that both the crossing function is not zero at the event restart and that a certain margin is introduced in order that $z = 0$ cannot appear at the zero crossing point. Contrary, if z is positive at the event restart, the crossing of the $-\varepsilon$ line triggers an event. If z is identical to zero at the event restart, the actual relation is needed in order that the zero crossing line is placed in the domain where the relation has its opposite value. This discussion is summarized in the following table:

relation	at restart	*crossing function*	event, when
$z > 0;\quad z \geq 0$	**true**	$z(t) + \varepsilon$	$z(t) \leq -\varepsilon$
	false	$z(t) - \varepsilon$	$z(t) \geq +\varepsilon$
$z < 0;\quad z \leq 0$	**true**	$z(t) - \varepsilon$	$z(t) \geq +\varepsilon$
	false	$z(t) + \varepsilon$	$z(t) \leq -\varepsilon$

Table 3: Crossing functions associated with Modelica relations.

In relations involving Real variables the "==" relational operator is not allowed because this would require a more complicated strategy introducing two crossing functions for one relation. Furthermore, the central feature that a relation is taken literally at event instants has to be given up. Instead a ε range around zero has to be introduced in which the == relation is **true**.

For the numerical integrators of the Godess project, Olsson ([Olss98], page 44) introduced a more useful definition of state events, by requiring a domain of validity $z_i(\mathbf{x}, t) \geq 0$. An event occurs, if this domain is left, i.e., if one of the z_i becomes negative. This approach is advantageous, because the integrator can check the valid domain before the start of the simulation. Furthermore, $z_i = 0$ is allowed for the start of the integration and at event restarts. It is straightforward to modify the above approach for the handling of Modelica relations, in order that it can also be used together with the Godess approach.

Open issue

The ε has to be selected in such a way that $|z| + \varepsilon$ is not identical to zero when z vanishes at an event instant, i.e., ε must be a little bit bigger as the smallest positive number representable on the machine, e.g., $\varepsilon = 10^{-60}$.

For variable structure systems there is another reason to introduce a small hysteresis via an ε, because algebraic loops are solved in an iterative process to determine new consistent initial conditions and the solutions of different model structures are close together (e.g. for the friction element, different structures are present whether *s* is a little bit less than 1 or a little bit bigger than 1). A hysteresis at the event instant can reduce the danger that numerical errors will result in a wrong model structure.

In such a case, the ε should be in the order where $|z|$ can be effectively regarded as zero. E.g. if a typical *z* value is in the order of one, one could select $\varepsilon = 10^{-12}$. In general the modeling system cannot figure out a meaningful value for ε. Therefore, the user should have the possibility to influence it directly. It is an open issue how an appropriate ε can be defined. E.g. Real numbers could get an additional attribute **scale** in order that the scaled number is in the order of 1. This scale factor may be provided by the user and can be used to select an appropriate ε, e.g., "$\varepsilon := scale \cdot 1000 \cdot \varepsilon_{machine}$". Note, that no decision has yet been made for Modelica in this respect.

8 Acknowledgement

We would like to thank the members of the Modelica group for useful discussions and hints, especially Rüdiger Franke, Pieter Mosterman and Hans Olsson.

References

[Astr90] Åström K.J., and Wittenmark B.: *Computer-Controlled Systems.* Theory and Design. Prentice-Hall, Second Edition, 1990.

[Bart92] Barton P.I.: *The Modelling and Simulation of Combined Discrete/Continuous Processes*. Ph.D. Thesis, University of London, 1992

[Cell79] Cellier F.E.: *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools.* Dissertation, Diss ETH No 6483, ETH Zürich, CH–8092 Zürich, Switzerland, 1979.

[Clau95] ClauSS C., Haase J., Kurth G., and Schwarz P.: *Extended Amittance Description of Nonlinear n-Poles.* Archiv für Elektronik und Übertragungstechnik / International Journal of Electronics and Communications, 40, pp. 91-97, 1995.

[Duff86] Duff I.S., Erismann A.M., and Reid J.K.: *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1986.

[Dymo98] Dymola: *Homepage: http://www.dynasim.se/*

[Eich98] Eich-Soellner E., and C. Führer. *Numerical Methods in Multibody Dynamics*. Teubner, 1998.

[Elmq85] Elmqvist H.: *LICS – Language for Implementation of Control Systems,* Technical report, CODEN:LUTFD2/(TFRT-3179)/1-130/(1985), Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1985.

[Elmq93] Elmqvist H., F. E. Cellier, and M. Otter: *Object–Oriented Modeling of Hybrid Systems*. Proceedings ESS'93, European Simulation Symposium, S. xxxi-xli, Delft, Niederlande, Okt. 1993.

[Gaut94] Gautier T., P. Le Guernic, and O. Maffeis. *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.

[gPRO98] gPROMS: *Homepage: http://www.psenterprise.com/gPROMS/*

[Halb91] Halbwachs N., P. Caspi, P. Raymond, and D. Pilaud. *The synchronous data flow programming language LUSTRE*. Proc. of the IEEE, 79(9), pp. 1305–1321, Sept. 1991.

[Halb93] Halbwachs N. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[Loet82] Lötstedt P.: *Mechanical systems of rigid bodies subject to unilateral constraints*. SIAM J. Appl. Math., Vol. 42, No. 2, pp. 281-296, 1982.

[Most96] Mosterman P. J., and G. Biswas: *A Formal Hybrid Modeling Scheme for Handling Discontinuities in Physical System Models*. Proceedings of AAAI-96, pp. 905-990, Aug. 2.-4., Portland, OR, 1996.

[Most98] Mosterman P. J., M. Otter, and H. Elmqvist: Modeling Petri Nets as Local Constraint Equations for Hybrid Systems using Modelica. SCSC'98, Reno, Nevada, 1998.

[Murt88] Murty K. G.: *Linear complementarity, linear and nonlinear programming*. Heidermann-Verlag, Berlin, 1988.

[Olss98] Olsson H.: *Runge-Kutta Solution of Initial Value Problems*. PhD. dissertation, Departement of Computer Science, Lund University, Lund Sweden, CODEN:LUTEDX/(TECS-1009)/1-178, (1998)

[Pfei96] Pfeiffer F., and C. Glocker: *Multibody Dynamics with Unilateral Contacts*. John Wiley, 1996.

[Sree91] Sreenivas R.S., and Krogh B.H.: *On condition/event systems with discrete state realizations*. Discrete Event Dynamical Systems 1, pp. 209-236, 1991.