Event Handling in the OpenModelica Compiler and Runtime System

Håkan Lundvall, Peter Fritzson, Bernhard Bachmann*

PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{haklu,petfr}@ida.liu.se

*Dept. Mathematics and Engineering, University of Applied Sciences,
D-33609 Bielefeld, Germany
bernhard.bachmann@fh-bielefeld.de

Abstract This paper gives an introduction and overview of problems and solutions regarding simulating hybrid DAEs (systems of Differential Algebraic Equations) with event-handling, in the context of producing such equations from models in the Modelica language. Implementation and measurements are done in the OpenModelica environment. The basic hybrid DAE solution algorithm is presented, followed by a discussion of handling possibly varying structure of the active part of the hybrid DAE, and how to find consistent initial values at start or restart of simulation. The problem of detecting events during continuous-time simulation is dealt with using crossing functions and an algorithm for proper integration with a numerical solver, here DASRT. Event-related aspects of code generation from models are presented, followed by an example showing how the event mechanism works. Finally, preliminary results from translating and simulating two examples models, the bouncing ball and the full wave rectifier, are presented and compared with those from a commercial simulation tool (Dymola), giving identical results.

1 Introduction

The OpenModelica[1,4,5] environment is an open source Modelica[2,3] modeling and simulation environment developed at PELAB, Linköping University. Previously it was not been possible to simulate hybrid models using the compiler and run-time system, only pure continuous systems. In order to provide a more complete simulation environment and to support the research carried out at PELAB we have recently added hybrid simulation capabilities to the OpenModelica compiler. One of the research topics that is in need of these capabilities is model checking of hybrid systems in Modelica[6].

Other tools available with support for the Modelica language or subsets thereof include Dymola[12], MathModelica[13], MOSILAB[14], SimulationX[15], Modelica which is a part of the Scicos toolbox[16], and the IDA Simulation Environment[17].
2 Conversion of a Modelica Model to a Hybrid DAE

A Modelica model is typically translated to a basic mathematical representation in terms of a flat system of differential and algebraic equations (DAEs) before being able to simulate the model. This translation process elaborates on the internal model representation by performing analysis and type checking, inheritance and expansion of base classes, modifications and redeclarations, conversion of connect equations to basic equations, etc. The result of this analysis and translation process is a flat set of equations, including conditional equations, as well as constants, variables, and function definitions. By the term flat is meant that the object-oriented structure has been broken down to a flat representation where no trace of the object hierarchy remains apart from dot notation (e.g. Class.Subclass.variable) within names.

3 Simulation of Hybrid Models Based on Solving Hybrid DAEs

A Modelica simulation problem in the general case is a Modelica model that can be reduced to a hybrid DAE in the form of equations (1), (2), (3) and (4), together with additional constraints on variables and their derivatives called initial conditions.

The initial conditions prescribe initial start values of variables and/or their derivatives at simulation time=0 (e.g. expressed by the Modelica start attribute value of variables, with the attribute fixed = true), or default estimates of start values (the start attribute value with fixed = false).

The simulation problem is well defined provided that the following conditions hold:

- The total model system of equations is consistent and neither underdetermined nor overdetermined.
- The initial conditions are consistent and determine initial values for all variables.
- The model is specific enough to define a unique solution from the start simulation time \( t_0 \) to some end simulation time \( t_1 \).

The initial conditions of the simulation problem are often specified interactively by the user in the simulation tool, e.g. through menus and forms, or alternatively as default start attribute values in the simulation code. More complex initial conditions can be specified through initial equation sections in Modelica.

There are some issues associated with simulating hybrid modeling. If, for example, the state variables are not continuous and differentiable at all points, a DAE-solver would have to use very small simulation steps around the discontinuity. In the Modelica language this issue is handled by stopping the continuous DAE-solver and locating the time when a discontinuous change might occur, i.e., whenever a relation expression switches from true to false or vice versa. When the
new set of values of the state variables has been computed the solver is restarted. In
this way the DAE-solver only has to handle continuous-time variables. This
technique was developed by Cellier [8].

An overview of other issues that arise in hybrid simulation can be found in [9].
Some of these issues are also treated in this paper.

4 The Form of the Hybrid DAE System

The hybrid DAE needs to represent both continuous-time behavior and discrete-
time behavior. We start with a summary of the notation used in the equations and
continue by formulating the continuous-time part, followed by the discrete-time
part.

4.1 Summary of Notation

Below we summarize the notation used in the equations that follow, with time
dependencies stated explicitly for all time-dependent variables by the arguments \( t \) or
\( t_e \):

- \( p = \{ p_1, p_2, \ldots \} \), a vector containing the Modelica variables declared as
  parameter or constant i.e., variables without any time dependency.
- \( t \), the Modelica variable time, the independent variable of type \textit{Real}
  implicitly occurring in all Modelica models.
- \( x(t) \), the vector of state variables of the model, i.e., variables of type
  \textit{Real} that also appear differentiated, meaning that \textit{der()} is applied to
  them somewhere in the model.
- \( \dot{x}(t) \), the differentiated vector of state variables of the model.
- \( u(t) \), a vector of input variables, i.e., not dependent on other variables, of
  type \textit{Real}. These also belong to the set of algebraic variables since they
do not appear differentiated.
- \( y(t) \), a vector of Modelica variables of type \textit{Real} which do not fall into
  any other category. Output variables are included among these, which
  together with \( u(t) \) are algebraic variables since they do not appear
  differentiated.
- \( q(t_e) \), a vector of discrete-time Modelica variables of type \textit{discrete}
  \textit{Real}, \textit{Boolean}, \textit{Integer} or \textit{String}. These variables change their
  value only at event instants, i.e., at points \( t_e \) in time.
- \( q_{pre}(t_e) \), the values of \( q \) immediately before the current event occurred,
i.e., at time \( t_e \).
- \( c(t_e) \), a vector containing all \textit{Boolean} condition expressions evaluated
at the most recent event at time \( t_e \). This includes conditions from all
if-equations/statements and if-expressions from the original model as well
as those generated during the conversion of when-equations and when-statements.

- \( \text{rel}(v(t)) = \text{rel}(\text{cat}(1, x, \dot{x}, u, y, \{ t \}, q(t_e), q_{\text{pre}}(t_e), p)) \), a Boolean vector valued function containing the relevant elementary relational expressions from the model, excluding relations enclosed by \text{noEvent}(). The argument \( v(t) = \{ v_1, v_2, \ldots \} \) is a vector containing all elements in the vectors \( x, \dot{x}, u, y, \{ t \}, q(t_e), q_{\text{pre}}(t_e), p \). This can be expressed using the Modelica concatenation function \text{cat} applied to these vectors; \( \text{rel}(v(t)) = \{ v_1 > v_2, v_3 \geq 0, v_4 < 5, v_6 \leq v_7, v_12 = 133 \} \) is one possible example.

- \( f(\ldots) \), the function that defines the differential equations \( f(\ldots) = 0 \) in (1a) of the system of equations.

- \( g(\ldots) \), the function that defines the algebraic equations \( g(\ldots) = 0 \) in (1b) of the system of equations.

- \( f_q(\ldots) \), the function that defines the difference equations for the discrete variables \( q := f_q(\ldots) \), i.e., (2) in the system of equations.

- \( f_e(\ldots) \), the function that defines the event conditions \( c := f_e(\ldots) \), i.e., (3) in the system of equations.

- \( x_f(\ldots) \), the function that defines the reinitialization values for the continuous variables \( x(t_e) := f_x(\ldots) \) at events.

In the context of hybrid DAEs the state of a system is not only made up of the values of the set of variables that occur differentiated in the model. The overall state of a system may also include values of discrete variables. In this paper the word state is used in this sense, including the state of the discrete part of the system.

### 4.2 Continuous-Time Behavior

Now we want to formulate the continuous part of the hybrid DAE system of equations including discrete variables. This is done by adding a vector \( q(t_e) \) of discrete-time variables and the corresponding predecessor variable vector \( q_{\text{pre}}(t_e) \) denoted by \text{pre}(q) in Modelica. For discrete variables we use \( t_e \) instead of \( t \) to indicate that such variables may only change value at event time points denoted \( t_e \), i.e., the variables \( q(t_e) \) and \( q_{\text{pre}}(t_e) \) behave as constants between events.

We also make the constant vector \( p \) of parameters and constants explicit in the equations, and make the time \( t \) explicit. The vector \( c(t_e) \) of condition expressions, e.g. from the conditions of \text{if} constructs and \text{when} constructs, evaluated at the most recent event at time \( t_e \) is also included since such conditions are referenced in conditional equations. We obtain the following continuous DAE system of equations that describe the system behavior between events:

\[
\begin{align*}
    f(x(t), \dot{x}(t), u(t), y(t), t, q(t_e), q_{\text{pre}}(t_e), p, c(t_e)) &= 0 \\
g(x(t), u(t), y(t), t, q(t_e), q_{\text{pre}}(t_e), p, c(t_e)) &= 0
\end{align*}
\]
4.3 Discrete-Time Behavior

Discrete-time behavior is closely related to the notion of an event. Events can occur asynchronously, and affect the system one at a time, causing a sequence of state transitions.

An event occurs when any of conditions \( c(t_e) \) (defined below) of conditional equations changes value from false to true. We say that an event becomes enabled at the time \( t_e \), if and only if, for any sufficiently small value of \( \varepsilon \), \( c(t_e-\varepsilon) \) is false and \( c(t_e+\varepsilon) \) is true. An enabled event is fired, i.e., some behavior associated with the event is executed, often causing a discontinuous state transition.

Firing of an event may cause other conditions to switch from false to true. In fact, events are fired until a stable situation is reached when all the conditional expressions are false.

However, there are also state changes caused by equations defining the values of the discrete variables \( q(t_e) \), here the same as discrete-time variables, which may change value only at events, with event times denoted \( t_e \). Such discrete variables obtain their value at events, e.g. by solving equations in when-equations or evaluating assignments in when-statements. The instantaneous equations defining discrete variables in when-equations are restricted to particularly simple syntactic forms, e.g. \( \text{var} = \text{expr} \). These restrictions are imposed by the Modelica language in order to easily determine which discrete variables are defined by solving the equations in a when-equation.

Such equations can be directly converted to equations in assignment form, i.e., assignment statements, with fixed causality from the right-hand side to the left-hand side. Regarding algorithmic when-statements that define discrete variables, such definitions are always done through assignments. Therefore we can in both cases express the equations defining discrete variables as assignments in the vector equation (1a), where the vector-valued function \( f_q \) specifies the right-hand side expressions of those assignments to discrete variables.

\[
q(t_e) := f_q(x(t_e), \dot{x}(t_e), u(t_e), y(t_e), t_e, q_{\text{pre}}(t_e), p, c(t_e)) \tag{2}
\]

The last argument \( c(t_e) \) is made explicit for convenience. It is strictly speaking not necessary since the expressions in \( c(t_e) \) could have been incorporated directly into \( f_q \). The vector \( c(t_e) \) contains all Boolean condition expressions evaluated at the most recent event at time \( t_e \). It is defined by the following vector assignment equation with the right-hand side given by the vector-valued function \( f_c \). This function has as arguments the subset of the discrete variables having Boolean type, i.e., \( q^B(t_e) \) and \( q_{\text{pre}}^B(t_e) \), the subset of Boolean parameters or constants, \( p^B \), and a vector \( \text{rel}(v(t)) \) evaluated at time \( t_e \), containing the elementary relational expressions from the model. The vector of condition expressions \( c(t_e) \) is defined by the following equation in assignment form:

\[
c(t_e) := f_c(q^B(t_e), q_{\text{pre}}^B(t_e), p^B, \text{rel}(v(t_e))) \tag{3}
\]
The argument \( v(t) = \{ v_1, v_2, \ldots \} \) is a vector containing all scalar elements of the argument vectors. This can be expressed using the Modelica concatenation function \( \text{cat} \) applied to the vectors, e.g. \( v(t) = \text{cat}(1, x, \dot{x}, u, y(t), q(t_e), q_{\text{pre}}(t_e), p) \). For example, if \( \text{rel}(v(t)) = \{ v_1 > v_2, v_3 >= 0, v_5 < 5, v_7 <= v_12 = 133 \} \) where \( v(t) = \{ v_1, v_2, v_3, v_4, v_6, v_7, v_12 \} \), then it might be the case that \( c(t) = \{ v_1 > v_2 \text{ and } v_3 >= 0, v_{10}, \text{not } v_{11}, v_4 < 5 \text{ or } v_5 <= v_3, v_{12} = 133 \} \), where \( v_{10}, v_{11} \) are Boolean variables and \( v_1, v_2, v_3, v_4, v_6, v_7 \) might be Real variables, whereas \( v_{12} \) might be an Integer variable.

In (3) above, \( \text{rel}(v(t)) = \text{rel}(\text{cat}(1, x(t), \dot{x}(t), u(t), y(t), t, q(t_e), q_{\text{pre}}(t_e), p)) \) is a Boolean-typed vector-valued function containing the relevant elementary relational expressions from the model, excluding relations enclosed by \( \text{noEvent()} \). These relations form the zero-crossing discussed in section 9.

Discontinuous changes of continuous dynamic variables \( x(t) \) can be caused by so-called reinit equations in Modelica. As in the case of discrete variables, such discontinuous changes can only occur at events. The effect of a reinit-equation that is activated at \( t_e \) is an assignment to the continuous variable at time \( t_e \) of the form:

\[
x(t_e) := f_x(x(t_e), \dot{x}(t_e), u(t_e), y(t_e), t_e, q_{\text{pre}}(t_e), p, c(t_e))
\]

(4)

For all variables in \( x(t_e) \) that are not affected by an reinit-equation \( f_x(\ldots) \) takes the value of \( x(t_e) \), leaving the variable unchanged.

### 4.4 The Complete Hybrid DAE

The total equation system consisting of the combination of (1), (2), (3) and (4) is the desired hybrid DAE equation representation for Modelica models, consisting of differential, algebraic, and discrete equations.

This framework describes a system where the state evolves in two ways: continuously in time by changing the values of \( x(t) \), and by instantaneous changes in the total state represented by the variables \( x(t) \), \( y(t) \), and \( q(t) \). Instantaneous state changes occur at events triggered when some of the conditions \( c(t_e) \) change value from false to true. The set of state variables from which other variables are computed is selected from the set of dynamic variables \( x(t) \), algebraic variables \( y(t) \), and discrete-time variables \( q(t) \).

### 5 Hybrid DAE Solution Algorithm

The general structure of the hybrid DAE solution algorithm is presented in Figure 1, emphasizing the main structure rather than details. First, a consistent set of initial values needs to be found based on the given constraints, which often requires the solution of an equation system consisting of the initial constraints. Then the hybrid DAE solver checks whether any event conditions in when-equations, when-
statements, if-expressions, etc. have become true and therefore should trigger events. If there is no event, the continuous DAE solver is used to numerically solve the DAE until an event is detected or the end of the simulation is reached.

Figure 1. The general structure of the hybrid DAE solution algorithm.

If the conditions for an event are fulfilled, the event is fired, that is, the conditional equations associated with the event are activated and solved together with all other active equations. This means that the variables affected by the event are determined, and new values are computed for these variables. Subsequently a new initial value problem has to be solved to find a consistent set of initial values for restarting the continuous DAE solver, since there might have been discontinuous changes to both discrete-time and continuous-time variables at the event. This is called the restart problem. Of course, firing an event and solving the restart problem may change the values of variables, which in turn cause other event conditions to become true and fire the associated events. This iterative process of firing events and solving restart problems is called event iteration, which must terminate before restarting the continuous DAE solver.
The overall structure of the hybrid DAE solution algorithm is displayed in Figure 1 and summarized below:

1. Solve an *initialization value problem* of finding a consistent set of initial values before starting solution of the continuous part, equation (1), of the hybrid DAE.

2. Solve the *continuous DAE part* (1) of the hybrid DAE using a numerical DAE solver. During this phase the values of the discrete variables \(q\) as well as the values of the conditions \(c\) from the when-equations, -statements, if-expressions, etc. of the model are kept constant. Therefore the functions \(f(\ldots)\) and \(g(\ldots)\) in (1) are continuous functions of continuous variables, which fulfills the basic requirements of continuous DAE solvers.

3. During solution of the continuous DAE, all relations \(rel(\ldots)\) occurring in the conditions \(c\) are constantly monitored. If one of the relations changes value causing a condition to change value from false to true, the exact time instant of the change is determined, the continuous DAE solution process is halted, and an event is triggered.

4. At an event instant, when an event has been fired, the total system of *active* equations is a mixed set of *algebraic* equations, which is solved for unknowns of type *Real*, *Boolean*, and *Integer*.

5. After the processing of an event, the algorithm continues with step (1) of solving the restart problem of finding a consistent set of initial values. After this step solving the continuous part of the hybrid DAE is restarted if the check in (3) does not indicate new events to be processed in (4).

### 6 Varying Structure of the Active Part of the Hybrid DAE

Even though the total hybrid DAE system of equations is structurally time invariant, i.e., the set of variables and the set of equations is fixed over time, it is the case that conditional equations in hybrid DAEs can be activated and deactivated. This means that some variables in the state vectors \(x\) and \(q\) as well as certain equations can be disabled or deactivated at run-time during simulation, as well as enabled or activated.

Such activation or deactivation is caused by events. A disabled variable is kept constant whereas a disabled equation is removed from the total system of active equations that is currently solved. Thus the *active part* of the hybrid DAE can be *structurally dynamic*, i.e., at run-time change the number of *active* variables and equations in the DAE.

An example of varying structure is a system involving friction. A system consisting of two bodies that can either slide or be stuck together have a varying
number of states. When the relative velocity between the bodies becomes zero the bodies stick together and the friction no longer depends on the velocity. If the friction is modeled using only three modes; forward moving, backward moving and stuck, when going from stuck mode to forward moving the relative velocity would be initialized to zero when restarting the simulation, which would cause a new event going back to stuck mode. In order to handle this a model having the two extra modes start forward and start backward can be used. In these modes the relative acceleration is non-zero and that value can be used instead of the velocity to make the system go to stuck mode. See [10] and [11] for more detailed discussions regarding this issue.

7 Finding Consistent Initial Values at Start or Restart

As we have stated briefly above, at the start of the simulation, or at restart after handling an event, it is required to find a consistent set of initial values or restart values of the variables of the hybrid DAE equation system before starting the continuous DAE solution process.

At the start of the simulation, these conditions are given by the initial conditions of the problems (including start attribute equations, equations in initial equation sections, etc., together with the system of equations defined by (1), (2), and (3)). The user specifies the initial time of the simulation, \( t_0 \), and initial values or guesses of initial values of some of the continuous variables, derivatives, and discrete-time variables so that the algebraic part of the equation system can be solved at the initial time \( t = t_0 \) for all the remaining unknown initial values.

At restart after an event, the conditions are given by the new values of variables that have changed at the event, together with the current values of the remaining variables, and the system of equations (1), (2), and (3). The goal is the same as in the initial case, to solve for the new values of the remaining variables. In the initial case, however, the causality can be different since initial equations are included to calculate start values for the state variables, whereas at restart the state variables are always known. In a mathematical model of a system the restart values of the state variables could be governed by physical conservation constraints[9], whereas in Modelica, any discontinuous change to a state variable has to be made explicit using the \( \text{reinit} \) operator.

In both of the above cases, i.e., at events, including the initial event representing the start of simulation, the process of finding a consistent set of initial values at start or restart is performed by the following iterative procedure, called event iteration:

- Known variables: \( x, u, t, p \)
- Unknown variables: \( \dot{x}, y, q, q_{pre}, \ldots \)
loop
   Solve the equation system (l) for the unknowns,
   with \( q_{pre} \) fixed;
   if \( q = q_{pre} \) then exit loop;
   \( q_{pre} := q \);
end loop

In the above pseudo code we use the notation \( q_{pre} \) corresponding to \( \text{pre}(q) \) in Modelica.

8 Detecting Events during Continuous-time Simulation

Event conditions \( c \) are Boolean expressions depending on discrete-time or continuous-time model variables. As soon as an event condition changes from \text{false} to \text{true}, the event occurs. It is useful to divide the set of event conditions into two groups: conditions which depend only on discrete-time variables and therefore may change only when an event is fired, and conditions which also depend on continuous-time variables and may change at any time during the solution of the continuous part of the DAE. We call the first group \textit{discrete-time conditions}, and the second group \textit{continuous-time conditions}.

The first group causes no particular problems. The discrete-time conditions are checked after each event when the discrete-time variables might have changed. If some of the conditions change from \text{false} to \text{true}, the corresponding events are simply fired.

The continuous-time conditions can be further divided into those depending only on \textit{time} and those depending on \textit{state-variables}. The firing time of the conditions belonging to the former group can be calculated beforehand and the integrator can be set to stop at the calculated time \( t_e \). Whereas the conditions belonging to the latter are more complicated to handle. Each Boolean event condition needs to be converted into a continuous function that can be evaluated and monitored along with the continuous-time DAE solution process. Most numerical software, including DAE solution algorithms, is designed to efficiently detect when the values of specified expressions cross zero.

9 Crossing Functions

To be able to detect when Boolean conditions become \text{true}, we convert each continuous-time Boolean event condition into a so-called \textit{crossing function}. Such a function of time crosses zero when the Boolean condition changes from \text{false} to \text{true}. For example, the simple conditional expression \( y > 53 \) changes from \text{false} to \text{true} when \( y - 53 \) crosses zero from being less than zero to being greater than zero,
as depicted in Figure 2. The body of the corresponding crossing function is simply $y-53$.

![Figure 2. A boolean expression and its zero-crossing function.](image)

The decision to react on changes of event conditions in Modelica from $\text{false}$ to $\text{true}$ rather than from $\text{true}$ to $\text{false}$ is arbitrary; it could also have been the other way around.

Special care must be taken regarding strict relations like $y>53$, since the integrator normally stops at the time instant, $t_e$, where the crossing function is exactly zero and evaluating the triggering relation expression would actually result in a false value. Because of this we consider the derivative of the crossing function corresponding to the relation when evaluating it; if the derivative is positive we let the greater than relation return true even if the left hand side is equal to the right hand side. Since $x<y$ is the same as $\text{not } x>y$ the same considerations must be used regarding non-strict relations as well. Table 1 shows how relations are evaluated for different cases. In short, any expression in the process of becoming true should return true and any expression in the process of becoming false should return false. Another solution to this problem would be to always make sure the integration stops at $t_e+\epsilon$, where $\epsilon$ is a small number strictly larger than zero.

<table>
<thead>
<tr>
<th>$\text{der}(x-y)$</th>
<th>$x&gt;y$</th>
<th>$x&gt;y$</th>
<th>$X&lt;y$</th>
<th>$x&lt;0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;0</td>
<td>true</td>
<td>True</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
<td>True</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>&lt;0</td>
<td>false</td>
<td>False</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 2. Values of relation expressions at events.

Searching for zero crossings during integration can be done in the following way:

- When the solver is about to accept a step or when time reaches a point were output is wanted, evaluate the crossing functions resulting in a set of crossing function indices for functions returning non-negative results.
For each function in the set, find the exact time instant when the function is zero, using the bisection method.

Choose the crossing function or functions with the earliest corresponding time instant $t_e$.

Evaluate the system at that time instant.

Return a list of the found crossing functions indices and the state at time $t_e$.

One problem with this approach is that if the integration step size is too large events can be missed. Consider the following model:

```model MissingEvents
  Real x;
  Real y;
  Real c;
  equation
    der(x) = 1;
    y = x*sin(x*1000);
    when y > 0.8 and not c > 0 then
      c = time;
    end when;
  end MissingEvents;
```

Since the when-equations depend only on an output variable that is not considered during integration, the step size used by the integrator can become larger than the period of $y$. This means that the crossing function returns negative values each time it is called despite the fact that it would have returned positive values had it been called for other time instants. This is a limitation that is shared with other tools such as Dymola.

Given an output interval length of 0.1 simulation of the above model results in $c=0.9$ at end of simulation when the correct value should be $c=0.8119$.

10 Integrating the Numerical Solver with Event-Handling

In the OpenModelica environment a version of DASSL with root finding is used (DASRT)[7]. In order to integrate event handling in the compiler and run-time system, the compiler must produce crossing functions and handlers for the events; the actual search for zero crossings is left to the solver.

The following functions must be made available to the solver:

```function DAE_res()
  Equations for state variables on residual form.
end function

function ZeroCrossing()
  Contains the crossing functions indexed from 0 to $ng-1$, where $ng$ is the number of crossing functions.
end function
```
The sign of the crossing function is chosen in such a way that the function always goes from negative to positive when an event is to be triggered. This is no requirement of the solver, but it is useful when the runtime system checks whether any new events got triggered as a result of variable changes due to the handled event. The runtime system only has to go through the crossing functions and see if any of them has become positive. Before each restart of the solver, the signs of the crossing functions are recalculated so that all the functions are negative. If a crossing function is equal to zero when the solver is to be started, it is disabled by setting it to -1, because the solver cannot handle crossing functions that are zero at startup.

Pseudo code for the simulation loop is shown below.

Integration time variable: $t$
Time of next simulation output: $t_{out}$
Queue of events: $eventQueue$

```plaintext
call DASRT to integrate from $t$ to $t_{out}$;
loop
  if $t >= t_{out}$ then exit loop;
  if DASRT stopped at a root then
    loop
      emit variable values to the result file;
      for each root
        call handleZeroCrossing;
      end for
      emit variable values to the result file;
      call newEventCheck;
      call startEventIteration;
      call DASRT to restart the integration and integrate to $t_{out}$;
      if DASRT did not find roots then exit loop;
    end loop;
  end if;
  emit variable values to the result file;
  $t_{out} := t_{out} + step$;
  call DASRT to integrate from $t$ to $t_{out}$;
end loop;
emit variable values to the result file;
end

function startEventIteration
  loop
    if $eventQueue$ is empty then exit loop;
    event := pop $eventQueue$;
    if event represents a when clause then
      call handleEvent;
    else
      call handleZeroCrossing;
  end loop;
end function
```
end if;

call newEventCheck;
call ZeroCrossing;
for each sign change
    push eventQueue, crossing function index
end loop;
end loop;
end function;

11 Code Generation

In the equation sorting algorithm we assume that the equations conform to certain rules which simplifies the sorting:

There are no if-equations with non-constant conditions. Such equations are first transformed to equations of the form 0 = if cond then <truepart> else <falsepart>;. In this way they can be treated as regular equations.

The conditional expressions of when-clauses only contain Boolean variables. If a condition involves a relation expression a help variable is introduced along with an equation binding it to the expression it replaces. This way when the solver stops as a result of a crossing function becoming positive, handling routines only have to set the appropriate discrete variable and then the event iteration mechanism handles the triggering of the when-equations.

When dealing with hybrid simulations one could make a distinction between time-events and state-events. By time-events we mean events triggered by expressions not depending, directly or indirectly, on any state variables. The triggering time of such events can be calculated beforehand and the root finder of the solver need not be used. This is more efficient, but in this first version we do not make this distinction.

Apart from the functions listed in section 10 the simulation code generated by OpenModelica contains the following functions:

function DAE_output() Equations for output variables
handleZeroCrossing() Called by the simulation loop when the solver has stopped as a result of a crossing function passing zero.

handleEvent() Called by the event iteration loop whenever a Boolean variable has become true as a result of a zero crossing or as a result of another event.

newEventCheck() Called once for each iteration in the event iteration loop to check if any new event was triggered. Any

In the first step of the equation sorting part of the compilation, all equations and variables are gathered. In this stage all equations appearing inside when clauses are checked to see if they conform to the requirements of when-clauses and all
variables that appear as left hand sides of these equations are marked as discrete. These equations are considered during the rest of the equation sorting, but instead of being output in the functions used by the DAE solver (functionDAE_res() and functionDAE_output()), they are put in the handleEvent() function and are therefore only calculated at events instead of for each iteration of the solver.

Next, all expressions in the model, apart from those appearing inside noEvent(), etc, are searched for relation expressions, e.g., $x > 5$. Each such expression, labeled with a list of all equations in which it occurs, is added to a list of zero crossings (rel). If rel contains more than one element with identical relation expressions, those elements are merged together by appending the lists of equations in them. Each element in rel generates one crossing function in zeroCrossing() and a section in handleZeroCrossing() in which the equations assigned to the relation is placed. All variables depending on variables updated in handleZeroCrossing() are also updated.

In the newEventCheck() function a test of the form “if (edge(HelpVar[…] ))” is generated for each introduced help variable. If the variable has become true then the index of the corresponding when-clause is placed on a queue holding the events yet to be handled. If there are variables depending on any discrete variable according to the sorting of the equations, then these variables are updated. If this causes a zero crossing to change its sign then the index of that zero crossing is placed in the event queue.

The event iteration starts by checking for any new events that has been fired as result of the crossing function passing zero. This is done by calling newEventCheck() in the generated simulation code followed by a call to function_ZeroCrossing(). For each crossing function reporting a positive result the index of that crossing function is placed on the event queue. Then the first event in the queue is handled and the check for new events is carried out again. This continues until the event queue is empty. At this time new consistent restart conditions are calculated and the solver is restarted.

Since each unique relation expression generates its own zero crossing function it is possible to write a model that causes the integrator to stop even though no event actually occurs, e.g., the code: “when b or x > 3 then …”, would cause an unnecessary interruption of the integrator if the variable b is already true. It would be possible detect such situations and disable the zero crossing function for cases when it would not influence the result. We have currently chosen not to implement this detection to simplify the implementation.

12 Example

In the following example we demonstrate how the event iteration mechanism works. When $x$ reaches 2.0, a chain of events starts that ends with the setting of $z$ to true. The iteration involves both when-conditions being set directly as in the case of $y$, and indirectly by setting $a$ to 2.0 causing $h2$ to becoming true. The whole chain of
events takes place in the same iteration without the continuous solver ever having to restart in the middle.

```model EventIteration
  Real x(start=1.0), dx;
  discrete Real a(start=1.0);
  Boolean y(start=false), z(start=false);
  Boolean h1, h2;
  equation
    der(x) = dx;
    dx = a*x;
    h1 = x >= 2;
    h2 = dx >= 4;
    when h1 then
      y = true;
    end when;
    when y then
      a = 2.0;
    end when;
    when h2 then
      z = true;
    end when;
  end EventIteration;
```

The above model gives a set of crossing functions consisting of:

- \( Z_{C1}(t) = \text{sign}(1) \times (x(t) - 2) \)
- \( Z_{C2}(t) = \text{sign}(2) \times (dx(t) - 4) \)

Help variables for when conditions are introduced:

- \( \text{HelpVar}(1) = h1 \)
- \( \text{HelpVar}(2) = y \)
- \( \text{HelpVar}(3) = h2 \)

- The initial constraints give initial values \( x(0)=1 \), \( a(0)=1 \), \( h1(0)=false \) and \( h2(0)=false \). Before integration starts the sign vector is set so that all crossing functions start negative. i.e., \( \text{sign}(1)=-1 \) and \( \text{sign}(2)=-1 \).
- At time \( t_e \), \( Z_{C1}(t) \) reaches zero and the integration stops
- The event handler for \( Z_{C1} \) is called evaluating all equations dependent on the relation \( x >= 2 \)
  \( \text{HelpVar}(1) = h1 = x>=2 \)
- In `newEventCheck edge(HelpVar(1))` evaluates to true and an event for the first when-clause is placed in the queue.
- Event iteration starts and the handler for the first when-clause is called setting \( y \) to true and updating any variables depending on \( y \)
  \( \text{HelpVar}(2) = y \)
- In `newEventCheck`, the expression `edge(HelpVar(2))` evaluates to true and an event for the second when-clause is placed in the queue.
- The crossing functions are checked. The iteration loop starts over and calls the event handler for the second when-clause, setting \( a=2 \) and updating all dependent variables:
dx = a*x
\[ \text{der}(x) = dx \]

- Nothing is found in `newEventCheck` since no discrete variable changed.
- The crossing function `ZC2` on the other hand now has a positive value and a zero crossing event is placed in the queue.
- The loops start over again and the handler for `ZC2` is called evaluating all equations dependent on \( xd > 4 \);
  \[ \text{HelpVar}(3) = h2 = x\text{de} > 4 \]
- In `newEventCheck`, the expression `edge(\text{HelpVar}(3))` evaluates to true and an event for the third when-clause is placed in the queue.
- The crossing functions are checked.
- The loop starts over and calls the event handler for the second when-clause, setting \( z = \text{true} \).
- No other variable depends on \( z \) so no more events are added to the queue.
- When the event queue is empty the sign vector is recalculated and the continuous solver is restarted from time \( t_e \).

### 13 Measurements and Evaluation

#### 13.1 Bouncing Ball

As a small test case for the implementation we have used a model of a bouncing ball.

```plaintext
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
  equation
    impact = h <= 0;
    der(v) = if flying then -g else 0;
    der(h) = v;
    when {impact, h <= 0 and v <= 0} then
      v_new = if edge(impact) then -e*pre(v) else 0;
      flying = v_new > 0;
      reinit(v, v_new);
    end when;
end BouncingBall;
```

When the bounces, i.e., values of \( h \), become very low, lower than the simulation tolerances, it is useful to include logic in the model that disables the bounces, e.g.
introduce the flying variable in the model. A realistic physical interpretation is that the ball lies on the floor. If this is not done, depending on the solver used, either a lot of events are generated when the ball is near the ground, which would slow down simulation, or the ball could “fall through the floor”, when, due to numerical effects, the height is nearly zero but negative at the event time and the bounce is too small to return a positive height. A plot of the height of the ball is shown in Figure 3.

![Figure 3. Simulation result of the Bouncing Ball example.](image)

When we compare the trigger times of the bouncing events to the analytically calculated times they differ in time from the analytically calculated by less than $9.8 \cdot 10^{-6}$ s. In the simulation relative and absolute tolerances were set to $1.0 \cdot 10^{-3}$

### 13.2 Full Wave Rectifier

The second test case consists of an electrical circuit involving four ideal diodes connected in as a full wave rectifier bridge. The graphical representation of the model can be seen in Figure 4. This model is built entirely from components in the Modelica standard library. To verify the correctness of the simulation we compare the results produced by our OpenModelica implementation to the results acquired running the exact same model in Dymola.
Figure 4. The component diagram of the full wave rectifier example.

The OpenModelica result plot can be seen in Figure 5. The result of the OpenModelica simulation has been compared to a simulation of the same model in Dymola and the results match exactly. In this example more than one event can fire simultaneously. If both diodes $d_1$ and $d_2$ have the same knee voltage, they open and close simultaneously as a result of a single zero crossing. A mixed equation system including the continuous voltage and current variables as well as the Boolean off variable in the diodes, is solved.

Figure 5. Simulation result of the full wave rectifier example.
14 Conclusions

In this paper we have presented an overview of the methods for discrete event handling that we have implemented in the OpenModelica compiler. In the test models we show that event iteration works as expected and that the simulation results correspond well to analytically calculated results.

We have pointed out some of the challenges of handling events in Modelica such as how to handle strict inequalities by taking into account the direction in which the crossing function passes zero. We have also noted that there is a risk of missing events triggered by non-state variables if slowly varying state variables allow large integration steps but the triggering variable oscillates quickly.

The implementation is however not yet complete. In the future we also wish to implement separate handling of time-events, i.e., events where the time of the event does not depend on state and can be calculated beforehand, so that the integrator does not have to search for the roots.

15 Acknowledgements

This work has been supported by Swedish Foundation for Strategic Research (SSF) in the ECSEL graduate school, by Vinnova in the Safe and Secure Modeling and Simulation Project, and by the Swedish Research Council (VR).

16 References


