A Modelica-based Format for Flexible Modelica Code Generation and Causal Model Transformations

Jonas Larsson[+] Peter Fritzson[*]

[+] Fluid- and Mechanical Engineering Systems, Dept. Mechanical Engineering
[*] PELAB – Programming Environments Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
[+] jonla@ikp.liu.se [*] petfr@ida.liu.se

Abstract

The equation-based modeling language Modelica offers the possibility to extract a differential algebraic equation system (DAE). The DAE in turn can be used in numerical simulation, optimization, sensitivity analysis, diagnosis and more. For each of these categories, there exist software tools that all require slightly different input data, sometimes in symbolic form. Through examples, this paper briefly outlines a Modelica based format which can be used as input to the final code generation. This could yield a modular Modelica compiler in which the creation of diverse code generators is encouraged, which in turn widens the application area for Modelica. However, no formal definition of the format is presented here..

The paper also briefly addresses the issue of possible model causal adaptation needed at the equation level to make the model fit into target application with a causal usage context and how to automate the inclusion of these changes for seamless translation.

Keywords: Modelica, DAE, numerical solver, code generation, DSblock

1 Introduction

Modelica [1][8] is one of several equation-based modeling languages for describing continuous dynamic behavior and discrete events in the context of technical systems. A Modelica model can be flattened from its object-oriented form to a hybrid DAE, i.e., a DAE containing both continuous and discrete variables. One example of a model resulting in a hybrid DAE is that of a chemical process in a plant including a software-based controller and actuated on/off valves. The hybrid DAE is normally optimized such that numerical time-domain solutions will converge rapidly at a low computational cost. Finally, code generation takes place where the actual programming code is created for the application at hand, for example a simulation, optimization, sensitivity analysis, or diagnosis.

This compiling process is complex and among other things includes an extensive flattening procedure, efficient symbolic manipulations and sorting algorithms for a large number of equations. The creation demands a deep knowledge in the Modelica semantics as well as in the area of solving DAEs. As an example, the OpenModelica compiler [2], programmed in a Modelica-like syntax [3] contains some 100 000 lines of code.

In order to facilitate and encourage a widened cooperation in Modelica compiler construction, this paper discusses a Modelica based format for the information available between the optimization and code generation steps, called MOO from now on. The information is essentially several equation sets for initialization, numerical integration, re-initialization at discrete events, and finally output. Today there already exist a Modelica format between flattening and optimization that is a subset of Modelica since the object-oriented syntax is not needed at that stage.

The equation-based MOO format exemplified in this paper enables a discussion between compiler designers and the simulation tool developers at the other end on the interface information needed This can lead to widened support for Modelica in simulation tools and to new solvers, etc. due to a more modular development of code generators.

The second but smaller subject of the paper is on adapting Modelica models in terms of interface variables and the equation set to fit a target application with a specific causality context, such as a model library with conventions for the exchanged variables among models in terms of computational causality, signs, and names. Through the algorithms described,

a model can be fitted with new interface variables and equations if needed.

2 Related Work

The file based approach MOO has its disadvantages, such as low performance in comparison to a monolithic compiler since the internal data structure needs to be converted to and from the Modelica syntax. On the other hand, a programmer does not need to get acquainted with the rest of the compiler code and could save time this way. Moreover, flattened Modelica of the OpenModelica compiler is already used today as input to third party compilers.

Another alternative would be to keep the current situation where the generated C code is utilized through function calls to get access to derivatives, output values etc, as is performed in DSblock [4] and SimStruct [5]. However, this C code can be generated from MOO, so the MOO file is an inclusive option. Also, the advantage with introducing MOO is that the equation sets produced are available on symbolic form, which is important in order to create new types of applications where the equations need to be processed further through symbolic mathematics and other operations. One example is optimization, where the Hessian (matrix of second-order partial derivatives) is of interest. Many mathematical operations can be performed numerically, such as creating the Hessian, but often a mixed symbolic-numeric approach results in higher accuracy and better performance.

The paper [6] describes the syntax and semantics of an interchange format for hybrid models in general and is thus at a higher, more abstract level than MOO. MOO is however specified with [6] in mind not at least since compatibility between the two means that the code generators implemented for Modelica can be applied for other hybrid modeling languages as well with minor effort.

3 Limitations

Algorithms, arrays, and function calls of the Modelica language are not the main concern of this paper but are at times commented in terms of implementation.

4 Hybrid Modeling in Modelica

In this chapter the basics of hybrid modelling in Modelica is described as well as the example model used throughout the text. In the following two sections, the resulting typical information from the flattening and optimization steps is described as well as the information needed for code generation with different applications in mind.

A bouncing ball is a good example of a hybrid system. The motion of the ball is characterized by the height `h` and the vertical velocity `v`. The ball moves continuously between bounces, whereas discrete changes occur at bounce times, as depicted in Figure 1 below. When the ball bounces against the ground its velocity is reversed. An ideal ball would have an elasticity coefficient `e` of 1 and would not lose any energy at a bounce. A more realistic ball, as the one modelled below, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.

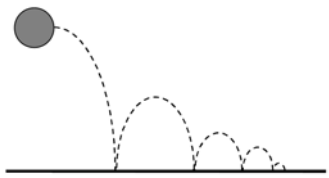

Figure 1. A bouncing ball

The bouncing ball model contains the two basic equations of motion relating height and velocity as well as the acceleration caused by the gravitational force. At the bounce instant where the boolean `impact` variable switches value, the velocity is suddenly reversed and slightly decreased, i.e., `v(after bounce) = -e*v(before bounce)`, which is accomplished by the special syntactic form of instantaneous equation: `reinit(v,-e*pre(v))`. The `pre` operator returns the value of the variable argument just before the latest event, in this case the velocity just before impact. The `edge` operator used in the example is simply defined as: `var and not pre(var)` and thus indicates a change in the discrete variable `var`.

```
model BouncingBall
  parameter Real e=0.7 "bounce coeff";
  parameter Real g=9.81 "gravity acc.";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true);
  Boolean impact;
  Real v_new;
equation
  impact = h <= 0.0;
  der(v) = if flying then -g else 0;
  der(h) =v;
```

```
  when {h <= 0.0 and v <= 0.0,impact} then
    v_new =  if edge(impact) then
               -e*pre(v)
             else
               0;
    flying = v_new > 0;
    reinit(v, v_new);
  end when;
end BouncingBall;
```

The equations within the `when`-equation are executed whenever any of the given boolean expressions changes values. Thus, if the ball is on the ground and has a negative velocity but the impact variable has not changed the velocity will be reset to zero and the boolean variable `flying` is set to false. The derivative of the velocity in the equation section is then also set to zero due to the if-expression.

5 Model Translation to Solvable Mathematical Representation

A Modelica model is first flattened from its object oriented form to a set of hybrid differential and algebraic equations on the hybrid DAE implicit form [8]

$$F\left(\dot{x}(t), x(t), u(t), y(t), t, q(t_e), q_{pre}(t_e), p, c(t_e)\right) = 0 \qquad (1)$$

The vector $q(t_e)$ is discrete-time variables and the corresponding predecessor variable vector $q_{pre}(t_e)$ is denoted `pre(q)` in Modelica. The time t_e used instead of t for these variables indicate that such variables may only change value at event time points t_e. The time t and constant vector p of parameters and constants are made explicit in the equation. The vector $c(t_e)$ is the conditional expressions from for example `if` and `when` constructs, evaluated at the most recent event.

Equation (1) can contain explicit or implicit algebraic relations among the states x which leads to difficulties in DAE solvers since numerical differentiation is needed. Index reduction [7][8] is therefore applied, which means that certain equations of the DAE are differentiated symbolically wrt time. The result can be brought to the form

$$f\left(\dot{x}(t), x(t), u(t), y_f(t), t, q(t_e), q_{pre}(t_e), p, c(t_e)\right) = 0 \qquad (2a)$$

$$g\left(x(t), u(t), y_f(t), y_g(t), t, q(t_e), q_{pre}(t_e), p, c(t_e)\right) = 0 \qquad (2b)$$

where $\partial f / \partial \dot{x}$, $\partial f / \partial y_f$ and $\partial g / \partial y_g$ are nonsingular but most often nonlinear. The derivatives $\dot{x}$ and algebraic variables y_f are found by solving Eq. 2a after which 2b is used for computing y_g, which are output variables. Equation 2 is supplied to an ODE or DAE solver in order to compute x and y in between events.

Equation 2 is the presumed resulting equation set from the optimization phase of the model compiling process.

Apart from the equations in (2), there exist assignments of discrete variables in normal equations as well as in the instantaneous `when` equations which are active only at events. These assignments can be written

$$q(t_e) := f_q\left(\begin{matrix}\dot{x}(t_e), x(t_e), u(t_e), y_f(t_e), y_g(t_e), t_e, q(t_e), \\ q_{pre}(t_e), p, c(t_e)\end{matrix}\right) \qquad (3)$$

The boolean conditions vector is stated [8] as follows:

$$c(t_e) := f_e\left(q^B(t_e), q^B_{pre}(t_e), p^B(t_e), rel(v(t_e))\right) \qquad (4)$$

The function f_e takes as arguments the subset of discrete variables and parameters of boolean type and the boolean vector *rel* contains all elementary relations of the model. Two possible relations are {a > b, c < a} which yields a possible $c(t_e)$ as {a>b and not(c<a), a>b or c<a}.

6 Modelica Representation for Input to Code Generation

A simulation comprises initialization in order to reach consistent start values for dependent variables, numerical integration between events, event detection during integration by detecting changes in the boolean relations *rel*, solving both continuous and discrete variables at events, and termination. The MOO file contains information needed for all these operations and is described in this section. The information in the MOO file is based on how the present OpenModelica compiler is implemented and on [5] and [8] - [11].

Since Modelica algorithms demand that all variables on the right hand side are defined, a Modelica model is more suitable than a function since the model can contain equations as well. The model below is filled with some of the Modelica statements discussed from now on.

```
model PreCodeInfo
   // Parameter, constant and
   // variable declarations
equation
   // Equations
end PreCodeInfo;
```

6.1 Constants and Parameters

The parameter and constant declarations from the flattened Modelica model are replicated. Attributes such as minimum and maximum values and units are important information since they can be used for error control in the generated code. Parameters such as `notset` below with no fixed values are computed in the initialization described later. This is also true for bound parameters, i.e., computed from other parameters.

```
parameter Real e=0.7;
parameter Real g=9.81;

parameter Real other=e*g(min=…);
parameter Real notset(fixed=false);
```

6.2 Variable Declarations

Variables have start values and at times a prefix stating whether the variable is discrete, input, or output. States are found by searching equations for the `der` operator applied to the variable in question in some equation. The variable declarations for the bouncing ball model are:

```
discrete Real v_new;
discrete Boolean impact;
discrete Boolean flying(start=true);
Real v;
Real h;
```

6.3 Initialization

Before simulation, the initial values for all variables, including states and nonfixed parameters, need to be computed. A similar procedure takes place after an event has occurred. Initial equations (from the `initial equation` construct in Modelica) are part of the total initial system of equations to be solved as well as `when`-equations with the `initial()` condition and equations formed from initial values of variables with the `fixed` attribute set to `true`.

An equation list is formed containing all model equations, excluding those within `when`-clauses without an `initial()` condition. Also, for continuous variables `v` with `start=startExp` and `fixed=true`, equations on the form `v=startExp` are added. For discrete variables under the same conditions, equations are added in which `pre(v) = startExp`. If `fixed=false` then `startExp` is only used as start value for the initialization, as is the case here. The `when` equation `reinit(v,v_new)` is treated as `v=v_new` in an initial equation if the `when`-condition contains `initial()`. For all `when`-equations with no `initial()` in the `when`-conditions, `v=pre(v)` is added for all `v := expr`.

It may very well be that the initialization is not a well posed problem, i.e., that the number of equations differs from the number of dependent variables. In the bouncing ball model case, the start value of `h` needs to be made fixed and `v` has no start value altogether. The discrete variables `v_new` and `flying` are initialized as well using default value 0 and start value true.

The resulting set of equations is shown below where both start values of discrete and continuous variables are computed. Two algorithms for finding a zero free diagonal [12] and then transforming into a block lower triangular form [13] have been applied. This results in a sequence of equation systems that can be solved more efficiently than the complete implicit equation system. The first index in the equations below is the block and the other the equation number. Where possible, the variables have been solved for and are given as the first element in the list, followed by the equation right hand side and then a string commenting on whether the second argument is the right hand side of an assignment or the residual.

```
iEq[1,1] = {pre(v_new),0,"ass"};
iEq[2,2] = {v_new,pre(v_new),"ass"};
iEq[3,3] = {pre(flying),true,"ass"};
iEq[4,4] = {flying,pre(flying),"ass"};
iEq[5,5] = {h,1,"ass"};
iEq[6,6] = {v,0,"ass"};
iEq[7,7] = {der(h),v,"ass"};
iEq[8,8] = {impact,h<=0.0,"ass"};
iEq[9,9] = {der(v),if flying then -g else
          0, "ass"};
```

For equations that cannot be solved explicitly, such as `nonlinear(v)=expression`, the equation is simply written in the residual form, shown below. Any other residual is formed by subtracting the second argument of the equation expressions above from the solved variable, such as `pre(flying)-true` in the third equation.

```
iEq[1] = {v,nonlinear(v)-expression,"res"};
```

In the bouncing ball model case, the variables can be computed sequentially. For each block however, if containing several equations, the mixed set of equations containing both discrete (Eq. (3)) and continuous variables (Eq. (2)) needs to be solved either through fixed point iteration over the discrete variables followed by normal solving of remaining variables or through other means such as optimization of the complete equation set in the block.

Jacobians

Jacobians can be defined only for the continuous equation subsets of each block since the discrete variables are discontinuous. Newton-Raphson iteration or other techniques can then be utilized for both linear and nonlinear equation blocks in order to provide a consistent and simple code generation. The Jacobian elements can in the case of a linear block alternatively be made input to a linear solver such as those contained in LAPACK [14] . Whether a block is linear or nonlinear is easily found by analyzing and searching the Jacobian for non-constant elements.

The Jacobians usually increase performance but demand symbolic manipulations and are therefore valuable to relieve the code generation software from. Consider the nonlinear equation system below:

```
iEq[1,1] = {a,sin(a)-userf(b),"res"};
iEq[1,2] = {b,cos(a)-abs(b),"res"};
```

The equations form a block of two equations and the Jacobian becomes :

```
iJac[1] = {{1,cos(a)},{2,-der_userf(b))}};
iJac[2] = {{1,-sin(a)},{2,-(if b<= 0 then -
1 else 1)}};
```

where only non-zero elements are included. In this case, the partial derivative function `der_userf` has been defined by the modeler as below:

```
function userf
   input Real x;
   output Real y;
algorithm
   // . . .
end userf;
der_userf = der(userf,x);
```

When such a partial derivative definition exists, automatic differentiation as in [15] can be utilized in many cases, at least if the derivative function is not defined as external. For external functions, only numerical differentiation remains in the general case.

6.4 Numerical Integration

After the initialization, numerical integration of the derivatives can take place in simulation in order to compute the states. For this purpose the ODE of Eq. (2) is supplied on the form previously described where the equations after the last differential equation are separated (2b) and computed in the output section together with simple equations that have been removed during optimization. In the ball model case the derivative equations can be extracted from the complete set of equations below

```
Eq[1,1] = {der(h),v,"ass"};
Eq[2,2] = {der(v),if flying then -g else
                    0,"ass"};

Jac[1] = {{1,1}};
Jac[2] = {{1,1}};
```

6.5 Reinitalization

When an event has occurred, certain `when` conditions will change and thus a certain new set of `when` equations will be active. The equation system will therefore change between events. This can be exploited by preparing efficient code for each combination of events. The number of combinations easily becomes large however and that is why the Modelica compiler Dymola [9] performs an offline simulation in order to see which combinations are likely to occur. A general solution works as well but with lower performance and is the one sketched here.

The equation set is created for the case where all `when`-equations are active. After an event, the effective equation system is gathered from this list using information on which `when`-equations are active. The states are considered known unless they are part of any active `reinit` equations in which case the reinits are treated as state assignments. In the example below, the complete equation set is shown together with the Jacobian. Here, the complete set is the same as the one formed when one of the `when`-conditions becomes true.

```
riEq[1,1] = {impact,h<=0.0,"ass"};
riEq[2,2] = {v_new,if edge(impact) then -
              e*pre(v) else 0,"inst"};
riEq[3,3] = {v, v_new,"inst"};
riEq[4,4] = {flying, v_new>0,"inst"};
riEq[5,5] = {der(h),v,"ass"};
riEq[6,6] = {der(v),if flying then -g else
                    0,"ass"};

riJac[5] = {{5,1}};
riJac[6] = {{6,1}};
```

The equation system above has been sorted and partitioned, which is too demanding during simulation. The Jacobian can however be used at all times since it only is defined for the continuous equations and is not affected by causality.

6.6 Event Detection and Location

Any discontinuity in the ODE equation set during simulation can be captured through zero crossing functions that switch sign at the discontinuity. One function is created for each unique relation in *rel*.

These functions are provided to the numerical integrator in order for the integrator to identify the time point of the discontinuity and restart the simulation at that point with new values for the states. The zero crossing functions below are given together with which equations they either are part of (can be both in Eq. (2) and (3)) or can make active by being part of `when` conditions.

```
z = {0.0-h,{1,2,3,4}}; 0.0-v,{2,3,4}};
```

Zero crossing functions reflect changes in relations that affect boolean variables that are part of `when`-conditions. To determine the current active set of `when`-equations it is necessary to know which `when`-equations a boolean equation triggers. In this case, the `impact` equation triggers the second `when`-condition and therefore could make equations 2, 3 and 4 active.

```
c = {1,{2,3,4}};
```

6.7 Code Generation

Given the complete Modelica model and naming conventions described above code can be generated for usage in a simulation environment, for example. Since it is Modelica syntax, the Modelica parser can be reused as well as parts of existing Modelica compilers for generating C code for individual Modelica expressions.

By translating the Modelica code to ModelicaXML [16], an XML implementation, a standard XML parser can be used instead together with widely used XML tools for traversing the XML structure and extracting and transforming information.

7 Model Adaptation for Causal External Usage

Imagine the scenario where a Modelica model is to be used within several causal models, for example within the simulation environments EASY5 and Simulink. One example could be a Modelica model of an engine, or even a complete standard Modelica library of driveline components that modelers of different enterprises have the interest of making part of their complete vehicle simulations performed in domain specific simulation tools with no support for equation-based models. A recent case is the CAPSim simulation centre [17] for hybrid vehicle simulation that has a model library open for usage and where the intention is to make the models usable for many simulation environments.

A flexible compiler supports straightforward code generation suited for these different environments, but the models themselves need to be adapted at the equation level as well, as is described in this section.

Consider a mechanical inertia model with outside connectors containing position and force. The connector class is specified in Modelica as shown below. The `flow` keyword tells the compiler to sum corresponding variables in a connection with other connectors. The non-flow variables are set equal. The mentioned inertia model connectors are part of the external interface and it is through those the communication of connector variables to and from other models takes place.

```
connector flange_pos
  Real pos(unit="m") "Pos into";
  flow Real force(unit="N") "Pos out";
end flange_pos;
```

Now consider a model where the outside connectors contain speed and force instead. Connecting the two models could be performed through a model that contains an equation for the relation between speed and position. This could even be automated by identifying differential variables and corresponding derivatives through unit checking.

```
model intermediate
  flange_speed fs;
  flange_pos fp;
equation
  fs.speed=der(fp.pos);
  fs.force=fp.force;
end intermediate;

connector flange_speed
  Real speed(unit="m/s");
  flow Real force(unit="N");
end flange_speed;
```

But in a causal target environment, the connectors are causal. The next example demonstrates a wrapper that assigns causalities to an acausal fource source model to make it suitable for insertion into a specific causal target model.

```
model force_source
  flange_pos flange_p;
equation
  flange_p.force = 0.2*cos(time);
end force_source;

model wrapper
  flange_pos flange_p;
  input Real speed;
  output Real force;
equation
  speed=der(flange_p.pos);
  force=flange_p.force;
end wrapper;
```

The causal model is created as below by connecting the wrapper to the force source. A Modelica com-

piler is able to create code for this model and saves information about the connectors in the comments of the input and output variables.

```
model causal
  wrapper w;
  force_source fs;
  input Real speed "($Conn:flange_p)";
  output Real force "($Conn:flange_p)";
equation
  connect(w.flange_p,fs.flange_p);
  w.speed=speed;
  force=w.force;
end causal;
```

7.1 Automatic Adaptation

The content of the wrapper model depends on both source and target model connector variables. New variables may need to be created, as the speed in this case, unit conversions may need to be applied as well as sign changes. All in all, a large number of combinations exist for which wrappers need to be created to support the usage of the model in arbitrary causal models. But the standards for exchanging variables among causal models in a certain simulation environment can be declared in template connectors, where also sign conventions are declared. In the force source example, the connector template could look like

```
model template_connector
  input Real speed(unit="m/s") "Pos out";
  output Real force(unit="N") "Pos into";
end template_connector;
```

From the template connector, a compiler can create the wrapper and causal model automatically as is described from now on. By comparing the connector class `flange_pos` with `template_connector`, it can be seen that two variables have the same units; the forces. The variables `pos` and `speed` have different units, but here it can be seen that speed is the derivative of `pos` by analyzing the units. The wrapper model is now complete, the forces are set equal and the derivative relation is established. If the speed of `template_connector` would have had the unit "mm/s" instead, a factor 1000 would have been needed in the wrapper equation.

There is no sign convention information in the Modelica models apart from comments. This information can be used as a starting point, looking for keywords such as "out of" and "into" and similar. But of course more explicit information would be valuable. In this case the sign conventions differ and the wrapper equations will both contain negations.

Here follows yet another template connector that could be used instead of the one defined earlier. The resulting causal model is compatible with the use of so called bilateral delay lines [18], which provide means for robust coupled simulation in which every component model solves its own equations and communicates at discrete time points with surrounding models. The variable C is in this case a force wave that passes back and forth in the delay line, as is the case in metal rods, for example. The end result is an overall numerical solution that mimics the time delay existing in physical interactions in nature. The variable Z is a kind of impedance. Without going further into the details of delay lines, the example shows how new equations can be automatically added to existing models in a mathematically sound way in order to adjust the models towards various boundary conditions.

```
model template_connector
  input Real C(unit="N");
  input Real Z(unit="N.s/m");
  output Real speed(unit="m/s");
  output Real force(unit="N");
equation
  force = C + Z*speed;
end template_connector;
```

7.2 Causality Combinations

It could be that the force source model cannot be solved with the specified input-output causality in the target connector. Since in causal environments, every second model has inverted causalities for compatibility in variable exchange, this can be tested for the target connector to see if that works better.

The force source model can be solved with speed as input and force as output, but not the other way around. The test can be performed through maximum matching algorithms [12] present in standard Modelica compilers.

What has not been mentioned so far is how to find a matching target connector for a certain outside connector of the Modelica model. For a match to exist there must be a one to one match between an outside connector variable and a causal target connector variable in terms of units. The units may be of differentiated/integrated form in the target connector. The target connector may have more variables if they are output variables and are part of supplied equations.

If the model has a number of connectors of different classes, such as both mechanical, electrical and other types, the testing becomes tedious and the testing can be handed over to the compiler to either generate all solvable combinations of inverted and non-inverted target connector causalities or generate the "best" solution. One measure for how good a solution is to see how high index the adapted model has. One example is feeding a mass with a position rather than

force and thus removing its inertia effect. The user interface could be implemented such that there is a choice to enforce the default target boundary conditions or to give priority to low index.

The model adaptation method where low index is prioritized described in this section has been implemented and tested in a Modelica compiler [19] supporting only continuous equations without arrays. The model adaptation takes place after flattening where the outside connectors of the flattened model have not been flattened so that they can still be identified. The algorithms have been proven useful since the compiler automatically produces numerically sound code for a given set of target connector templates and a source Modelica model.

8 Conclusions

A partial implementation of the Modelica intermediate code format has been done in the OpenModelica compiler to be used as input to external code generators. The intent is that this information can be produced and is complete for generating simulation code from hybrid Modelica models.

The model adaptation functionality for using Modelica models in an external causal context needs to be implemented in a compiler supporting the complete Modelica language, e.g. soon OpenModelica. This can pose problems, not at least due to the increasingly complex definition of connectors in Modelica. Nonetheless the model adaptation can be made less automatic if needed, for example by defining explicit transformations among different connector definitions.

References

[1] Modelica Association (2005): **Modelica – A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification**. Version 2.2

[2] P Fritzson, et al (2002): **The Open Source Modelica Project**. In Proceedings of The 2nd International Modelica Conference, 18-19 March, 2002, Munich, Germany. See www.ida.liu.se/projects/OpenModelica

[3] P. Fritzson, A. Pop and P. Aronsson (2005): **Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica**. Proceedings of the 4th International Modelica Conference, Hamburg, pp. 519-525

[4] M. Otter and H. Elmqvist (1995): **The DSblock model interface for exchanging model components.** In Proceedings of the 1995 EUROSIM Conference, Elsevier Science Publishers, pp. 505-510

[5] P. J. Mosterman and J. E. Ciolfi (2002): **Embedded Code Generation for Efficient Reinitialization**. Proceedings of 2002 IFAC 15th Triennial World Congress, Barcelona, Spain.

[6] A. Pinto, L. P. Carloni, R. Passerone and A. Sangiovanni-Vincentelli (2006): **Interchange Semantics for Hybrid System Models**. Proceddings 5th MATHMOD Vienna

[7] C. Pantelides (1988). **The Consistent Initialization of Differential Algebraic Systems.** SIAM Journal on Scientific and Statistical Computing, Vol. 9

[8] P. Fritzson (2004): **Principles of Object-Oriented Modeling and Simulation with Modelica 2.1**. Wiley-IEEE Press, ISBN 0-471-47163-1

[9] S.E. Mattsson, M. Otter, M, and H. Elmqvist (1999): **Modelica Hybrid Modeling and Efficient Simulation**, the 38th IEEE Conference on Decision and Control, CDC'99, Phoenix, Arizona, USA

[10] T. Park, P.I. Barton (1996): **State Event Location in Differential-Algebraic Models.** ACM Transactions on Modeling and Computer Simulation, Vol. 6, No. 2, pp. 137-165

[11] P. I. Barton and C. K. Lee (2002): **Modeling, Simulation, Sensitivity Analysis and Optimization of Hybrid Systems**. ACM Transactions on Modeling and Computer Simulation, Vol. 12, No. 4, pp. 256-289

[12] I. S. Duff and J. K. Reid (1981): **Algorithm 575 Permutations for a zero-free diagonal**. ACM Transactions on Mathematical Software, Vol. 7, pp. 387-390

[13] I. S. Duff and J. K. Reid (1978) **An implementation of Tarjan's algorithm for the block triangularization of a matrix**. ACM Transactions on Mathematical Software, Vol. 4, pp. 137-147

[14] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen (1995): **LAPACK Users' Guide**. SIAM Philadelphia, Pennsylvania, 2 edition

[15] H. Olsson, H. Tummescheit, and H. Elmqvist (2005): **Using Automatic Differentiation for Partial Derivatives of Functions in Modelica**. Proceedings of the 4th International Modelica Conference, Hamburg, pp. 105-112

[16] A. Pop and P. Fritzson (2003): **ModelicaXML: A Modelica XML Representation with Applications**, Proceedings of the 3rd International Modelica Conference, Linköping, November 3-4, pp. 419-430

[17] J. Fredriksson, J. Larsson, J. Sjöberg and P. Krus (2006): **Evaluating Hybrid Electric and Fuel**

Cell Vehicles using the CAPSim Simulation Environment. The 22nd International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium & Exposition, October 23-28

[18] D. M. Auslander D.M (1968): **Distributed System Simulation with Bilateral Delay-Line Models**. Journal of Basic Engineering, Transactions of ASME, pp. 195-200

[19] J. Larsson (2007): **A Framework for Implementation-Independent Simulation Models**. Accepted for publication in Simulation: Transactions of the Society for Modeling and Simulation International