Otter M., Olsson H.:
New Features in Modelica 2.0
2nd International Modelica Conference, Proceedings, pp. 7-1 – 7-12

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

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

New Features in Modelica 2.0

Martin Otter[1] and Hans Olsson[2]
[1]DLR, Oberpfaffenhofen, Germany, Martin.Otter@dlr.de
[2]Dynasim AB, Lund, Sweden, Hans.Olsson@dynasim.se

Abstract

The second major release of Modelica was finished and formally approved at the last Modelica design meeting, January 2002, Lund, Sweden. In this paper, the new features of Modelica 2.0 are described.

1. Introduction

The freely available, object-oriented modeling language Modelica is developed continuously since 1996. Modelica is designed to allow effective, component-oriented modeling of complex engineering systems described by differential, algebraic and discrete equations, e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. A large number of free and commercial libraries of fundamental models are available as well as commercial Modelica simulation environments. More information is provided at http://www.Modelica.org/.

In 1997, the first major version of Modelica was released, followed by four minor revisions released once a year. The second major release of Modelica was completed and formally approved at the last Modelica design meeting, January 2002, Lund, Sweden. The most important design goal was to enhance the development and use of application libraries, incorporating the experience and feedback of library developers, while keeping backward compatibility. A number of language enhancements have been added, significantly facilitating library development and use. In this paper, the new features of Modelica 2.0 are described. The following members of the Modelica Association have contributed to the development of Modelica 2:

P. Aronsson, Linköping University, Sweden.
B. Bachmann , University of Bielefeld, Germany.
P. Beater, University of Paderborn, Germany
D. Brück, Dynasim, Lund, Sweden
P. Bunus, Linköping University, Sweden
H. Elmqvist, Dynasim, Lund, Sweden
V. Engelson, Linköping University, Sweden
P. Fritzson, Linköping University, Sweden
R. Franke, ABB Corporate Research, Ladenburg
P. Grozman, Equa, Stockholm, Sweden
J. Gunnarsson, MathCore, Linköping
M. Jirstrand, MathCore, Linköping
S. E. Mattsson, Dynasim, Lund, Sweden
H. Olsson, Dynasim, Lund, Sweden
M. Otter, DLR, Oberpfaffenhofen, Germany
L. Saldamli, Linköping University, Sweden
M. Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
H. Tummescheit, Lund Institute of Technology, Sweden
H.-J. Wiesmann, ABB Corp. Res., Baden, Switzerland

2. Component Arrays

One part of the redesign of Modelica 2 was based on the experience with the Modelica.Blocks library in Modelica 1. The redesign supports *generic formulation* of blocks applicable to both scalar and vector connectors, *connection* of (automatically) *vectorized blocks*, and *simpler input/output connectors*. This allows significant simplifications of the input/output block library of Modelica, e.g., since only scalar versions of blocks that naturally vectorize have to be provided. Furthermore, new library components can be incorporated more easily. In addition, it is possible to use functions and functional blocks allowing, e.g., the sin-*function* to be inserted in a block diagram.

Since the first release, it was possible in Modelica to define homogenous component arrays, i.e., arrays where the array elements are instances of any desired class, for example:

```
Resistor R[10];
```

is an array of 10 resistors including both the resistor parameters and the resistor equations. In Modelica 2, features have been added for component arrays to widen their applicability.

Component Array Modifications

Assume a component is defined as

```
model FixedFrame
  parameter Real r[3] = {0,0,0};
  parameter Real alpha = 0;
  parameter Real beta  = 0;
  parameter Real gamma = 0;
    ...
```

```
  end FixedFrame;
```

which describes a coordinate system with respect to another one as fixed translation with vector **r** and fixed rotation around angles α, β, γ along x-, y-, and z-axis respectively. A part may have several frames and also other properties and can be defined as

```
model Part
  parameter Integer n=0;
  FixedFrame frames[n];
    ...
end Part;
```

There are different possibilities to define a part which has several frames:

```
Part p(n=2, frames[1](r={1,0,0},
                      alpha = 1),
            frames[2].alpha = -1);
```

Here, every element of the frame vector is explicitly modified. Another alternative is

```
Part p(n=4,frames(beta={1,2,3,4},
                 r = fill(1,4,3)));
```

where the *same* parameter of all frames are modified. For example, frames[:].beta is treated as a vector of 4 elements and therefore a vector of 4 elements has to be provided as modification. On the other hand, frames[:].r is treated as a (4,3) matrix.

```
Part p(n=10, frames(each r={1,0,0});
```

defines 10 frames, using the same vector r for all frames. In a similar way also nested component arrays are handled:

```
Part p[10](each n=3,
           each frames(each beta=1);
```

Here, 10 parts are declared, where every part has 3 identical frames with beta=1. In this application it is not very useful to define so many identical frames. However, in lumped models deduced from the discretisation of partial differential equations, often many elements of a component array have the same value. Example:

```
  parameter Integer n;
  parameter Real L=1 "length";
  parameter Real r=1 "resistance
                      per meter";
protected
  parameter Real Re=r*L/n;
  Resistor R[n+1](R =
              vector([Re/2;
                      fill(Re,n-1);
                      Re/2]));
```

All elements of the resistance vector R are the same, with the exception of the first and last one which each take half of the value of an element Resistance Re.

Block Vectorization

Connectors of signals of the Modelica.Blocks library are in Modelica 1.4 defined as

```
connector InPort
  parameter Integer n=1;
  input Real signal[n];
end InPort;
```

That is, the connector consists of a vector of Reals which are used as input signals. Such a connector is utilized in an input/output block as:

```
block FirstOrder
  parameter Real T=1 "time const.";
  InPort  inPort;
  OutPort outPort;
     ...
end FirstOrder;
```

Accessing the input signal of such a block is inconvenient:

```
FirstOrder b;
...
b.inPort.signal[1] // input signal
```

In Modelica 2 it is possible to define a connector as an extension from the base types, i.e., the following definition is possible:

```
connector InPortNew = input Real;
```

Also annotations for the graphical layout of icon and diagram layer of such a connector can be defined. Therefore, this connector may be dragged from a library window in a model window to construct a new model graphically. In a model, this connector is used as:

```
block FirstOrderNew
  parameter Real T=1 "time const.";
  InPortNew  u;
  OutPortNew y;
     ...
end FirstOrderNew;
```

Accessing the input signal of this block is now much simpler:

```
FirstOrderNew b;
...
b.u // input signal
```

In the 1.4 version of the Modelica.Blocks library, most blocks are manually vectorized, e.g., to define an instance which has 10 input and 10 output signals and 10 first order blocks for every signal path. This complicates the class definitions in Modelica.Blocks considerably, and in all cases, except Sources.KinemanticPTP, a vectorized block behaves as a vector of scalar blocks. With the extensions described above, this is much simpler. For example, a scalar Sine block may be defined as:

```
block Sine
  import Modelica.Math.*;
  import.Modelica.Constants.*;
  parameter Real Amplitude =1;
  parameter Real frequency =1;
  parameter Real phase=0;
  InPortNew  u;
  OutPortNew y;
equation
  y = Amplitude*
    sin(2*pi*frequency*time+phase);
end FirstOrderNew;
```

This looks like a text-book example of a sine source. Using 3 Sine sources is now performed by component arrays:

```
Sine s[3](each frequency=50,
          phase = {0,2,-2});
```

Note, that it is easy to define that all sine-sources shall have the same frequency, but different phases roughly corresponding to 3 electrical phases. A state space model may be defined as:

```
block StateSpace
  final parameter Integer nx =
                          size(A,1);
  final parameter Integer nu =
                          size(B,2);
  final parameter Integer ny =
                          size(C,1);
  parameter Real A[: ,nx];
  parameter Real B[nx, :];
  parameter Real C[: ,nx];
  parameter Real D[ny,nu];
  InPortNew  u[nu];
  OutPortNew y[ny];
  Real       x[nx]
equation
  der(x) = A*x + B*u;
      y  = C*x + D*u;
end StateSpace;
```

Connecting the 3 sin-sources as input to an instance of StateSpace which has three inputs can be performed in the following way:

```
  Sine s[3](each frequency=50,
            phase = {0,2,-2});
  StateSpace b(B=[0,0,1;...],...);
equation
  connect(s.y, b.u);
```

This is a connection of **s[:].y** with **b.u[:]**. This is allowed due to a new connection rule, provided the dimension sizes match, which is the case here.

3. Enumeration Types

Modelica 2 introduces enumerations to construct new base types which consist of countable sets of elements. Example:

```
type TextStyle = enumeration(
        Bold, Italic, UnderLine);
```

This declaration defines a new type TextStyle. An instance of this type may have only the values TextStyle.Bold, TextStyle.Italic or TextStyle.UnderLine. Such a type can be used in the following way:

```
TextStyle t1 = TextStyle.Bold;
TextStyle t2 = t1;
```

Currently, the only operations defined for enumeration types are the equal ("=") and the assignment (":=") operations. Furthermore, the relational operators <, <=, >, >=, ==, <> can be applied. The result depends on the order of the element in the enumeration declaration. For example TextStyle.Bold < TextStyle.Italic. It is planned to provide more operations in future Modelica releases, e.g., to access array indices by enumerations or inquire the next or previous enumeration element.

Enumerations are useful for defining properties and options in an understandable and safe way. Since enumerations are internally mapped to an Integer type, processing them is safer and much more efficient than if properties or options would be defined as Strings. Compared to using Integer constants it is clearer, requires less typing, and is

safer since each enumeration is a separate type. In Modelica 2, several enumeration types are predefined, such as StateSelect (see next section) and enumerations in graphical annotations.

4. State Selection Control

The continuous part of a Modelica model is mapped to a DAE, a differential-algebraic equation system, of the form

$$\mathbf{0} = \mathbf{f}(d\mathbf{x}/dt, \mathbf{x}, \mathbf{y}, t)$$

where $\mathbf{x}(t)$ are variables appearing differentiated and $\mathbf{y}(t)$ are pure algebraic variables. Conceptually, this DAE is transformed in to state space form

$$\begin{aligned} d\mathbf{x}_s/dt &= \mathbf{f}_1(\mathbf{x}_s, t) \\ \mathbf{x}_n &= \mathbf{f}_2(\mathbf{x}_s, t) \\ \mathbf{y} &= \mathbf{f}_3(\mathbf{x}_s, t) \end{aligned}$$

where $\mathbf{x}_s(t)$ are a subset of $\mathbf{x}$ which are independent from each other and $\mathbf{x}_n(t)$ are the other variables of $\mathbf{x}$. Variables $\mathbf{x}_s(t)$ are called **states** of the model. A numerical integration method essentially discretizes $\mathbf{x}_s$ over time, whereas all other variables are calculated as the solution of an algebraic system of equations at the actual time instant. The selection of $\mathbf{x}_s$ is not unique. Different choices may lead to drastically different numerical behaviour. A dynamic automatic selection of $\mathbf{x}_s$ by a tool is always possible, [4]. However, experience shows that user insight may lead to better choices or avoid the need for dynamic selection. On the other hand automatic selection is an efficient and reliable method, and users should not be forced to manually perform a complete manual state selection merely because dynamic state selection might be inefficient for some models. For this reason, in Modelica 2 it is possible to guide the state selection via the new attribute **stateSelect** of Real variables. The attribute has values from the enumeration **StateSelect** defined as:

```
type StateSelect = enumeration(
          never, avoid, default,
          prefer, always);
```

For "**never**", a variable will never be selected as a state, whereas for "**always**" the variable shall always be used as a state. For "**default**", which is the default for all Real variables, the states are automatically selected among the variables which appear differentiated. If "**prefer**" is used, the variable need not to be differentiated and is preferably used as state over those having the default value. Finally, for "**avoid**", the variable is only selected as a state, if it appears differentiated and if no other selection of variables with "default", "prefer", or "always" value is possible. A state preference definition may be given in the following way:

```
Real w(stateSelect =
            StateSelect.prefer);
```

Examples for appropriate state selection (from [2]):

Accuracy:

In rotating machinery systems used for power transmission (but not for positioning drive systems) and in power systems, angular positions of shafts are increasing with time, but relative positions between shafts are rather constant, at least in normal operation. Say that two rotating inertias are connected by a spring such that the relative distance between them are 0.1 rad and that their angular speed is 1000 rad/s. If the positions are calculated with a relative accuracy of 0.001, after 10 seconds there is hardly any accuracy in calculating the distance by taking the difference. The difference behaves irregularly and gives an irregular torque if simulations take too long. It is very difficult for a tool to find this out without actually doing simulation runs. Therefore, it is useful to define **StateSelect.prefer** for all relative variables in force elements (e.g., spring, damper, clutch). This will be performed in the next version of the Modelica.Mechanics.Rotational library.

Avoiding function inversion:

In thermodynamic problems property functions are utilized. These functions usually assume two variables to be inputs (for example pressure and enthalpy) and calculate other properties (such as temperature, density). Thus, if such variables are selected as state variables it is "simply" calling property functions to calculate other needed variables. Otherwise, it is necessary to solve non-linear equation systems to calculate the input variables of the property functions. Therefore, a good choice is to use **StateSelect.prefer** on all input variables of property functions, or use **StateSelect.avoid** on output variables from property functions.

Less nonlinear equations:

For three-phase power systems several choices of states are possible, especially selecting states from the stator side or from the rotor side. The first

choice leads to a **non-linear** DAE, whereas the second one leads (under certain assumptions) to a **linear** DAE. In a periodic steady state, the first choice results in a **periodic** solution of the states whereas in the second choice the states are identical to **zero**. As a result, selecting states from the rotor side (= Park transformation) leads to a more efficient and more reliable numerical solution and therefore these variables should have the attribute value **StateSelect.prefer**.

Avoid dynamic state selection:

For 3-dimensional mechanical systems having closed kinematic loops, an automatic static selection of states is not possible. Instead, the states have to be dynamically selected and changed during simulation in order to keep the (time-varying) Jacobian of the system non-singular. In many cases a suitable set of state variables is known, e.g., the relative position and velocity variables of the joints driving the mechanism. If these variables have attribute value **StateSelect.always** the simulation is more efficient which is especially important for real-time simulations.

Sensors:

A sensor may measure the speed "v" of a translational connector. Since the speed is not part of the connector, but the position "s" is, an equation of the form "**der**(s) = v" is present in the sensor, i.e., "s" appears differentiated and can be potentially used as a state. However, in most case the selection of "s" as a state is not appropriate, since introduction of a variable for just plotting should not influence the state selection. Therefore, an attribute value of **StateSelect.avoid** should be preferably used for differentiated variables in sensor objects (here: "s").

The general advice is that selection of states ought to be done automatically. This is also possible and unproblematic in most models. Only if there are good reasons, as pointed out above at hand of several examples, the modeler may give hints for state selection. Note, that in a library the values **StateSelect.never** or **StateSelect.always** should not be used, because a library has usually not enough information to rigidly force a state selection.

5. Improved Initialization

Modelica 2.0 introduces a mathematically rigid specification of the initialization of Modelica models, i.e., of hybrid differential algebraic equations. The new language constructs permit flexible specification of initial conditions as well as the correct solution of difficult, non-standard initialization problems occurring in industrial applications, for example:

- Stationary initialization around a constant reference velocity of an aircraft.
- Stationary initialization around periodic solutions, needed in power systems or in detailed engine models.
- Stationary initialization of continuous systems controlled by sampled data systems (the states of the discrete controllers are computed in such a way that the overall system is in a steady state when simulation starts).
- Initialization of discontinuous or variable structure systems, e.g., systems containing friction or backlash.

Since this is a large topic by itself, only a short overview is given here. Details are presented in the companion paper [3].

Before any operation, in particular simulation, is carried out with a Modelica model, initialization takes place to assign **consistent** values for all variables present in the model, including derivatives, **der**(…), and **pre**-variables, **pre**(…). The initialization uses all equations and algorithms that are utilized during the simulation.

In the most simplest case, when only continuous equations are present without algebraic dependencies of states (= no higher DAE index), a Modelica model is mapped to the following differential-algebraic equation system (DAE):

$$\mathbf{0} = \mathbf{f}(d\mathbf{x}/dt, \mathbf{x}, \mathbf{y}, t)$$

where $\mathbf{x}(t)$ are variables appearing differentiated, $\mathbf{y}(t)$ are algebraic variables and $\dim(\mathbf{f}) = \dim(\mathbf{x}) + \dim(\mathbf{y})$. These equations have to be fulfilled at all time instants, especially also at the initial time t_0. During simulation, an integrator calls the model providing basically $\mathbf{x}$ as input. Therefore, the model equations are solved under the assumption that $\mathbf{x}$ is known. During initialization, $\mathbf{x}$ is, however, unknown. As a result, there are only $\dim(\mathbf{x}) + \dim(\mathbf{y})$ equations for $2*\dim(\mathbf{x}) + \dim(\mathbf{y})$ unknowns during initialization. In the most general case this means that the modeler has to provide additionally $\dim(\mathbf{x})$ equations $\mathbf{g}(..)$ at the initial time resulting in the following system of equations

$$\mathbf{0} = \begin{bmatrix} \mathbf{f}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), \mathbf{t}_0) \\ \mathbf{g}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), \mathbf{t}_0) \end{bmatrix}$$

which has to be solved for the unknowns $d\mathbf{x}/dt(t_0)$, $\mathbf{x}(t_0)$, $\mathbf{y}(t_0)$. In general this means that the standard algorithms, such as BLT transformation, should be applied to this system, in order to compute the solution reliably and efficiently. From a user's point of view this procedure means that dim(**x**) equations have to be additionally provided for the initial time, e.g., $\mathbf{x}(t_0) = \mathbf{x}_0$ or $d\mathbf{x}/dt(t_0) = \mathbf{0}$. In Modelica 2 these initial equations can either be defined in the new **initial equation** / **initial algorithm** sections or as start value with attribute fixed = **true**. For example two initial equations $x1(t_0) = 1$ and $dx2/dt(t_0) = 0$ may be defined as:

```
  Real x1(start=1, fixed=true);
  Real x2 //default: fixed=false
initial equation
  der(x2) = 0;
equation
  der(x1) = -x1 + x2;
  der(x2) = -x2;
```

If there are constraints between states, the number of initial equations to be provided is less than dim(**x**). It may be difficult for a user of a large model to figure out how many initial equations have to be added. Therefore, it is essential that a Modelica environment has appropriate support. For example, Dymola performs index reduction and selects state variables for the simulation model [1], [3], [4]. Thus, it establishes how many states there are and how many initial conditions have to be additionally provided. If there are too many initial equations, Dymola outputs an error message indicating a set of initial equations or fixed start values from which initial equations must be removed or start values inactivated by setting fixed = **false**. If initial conditions are missing, Dymola makes automatic default selection of initial conditions. The approach is to select continuous time states with inactive start values and make their start values active by turning their fixed attribute to **true** to get a structurally well posed initialization problem. A message informing about the result of such a selection can be obtained.

6. Function Applications

In Modelica 1.4, a function application can have *either* positional *or* named *input* arguments. In Modelica 2, a function application may have optional *positional input arguments* followed by zero, one or more *named input arguments*. Arguments not explicitly present get the default value supplied in the function declaration. This feature is useful to make the same function fit for beginners and expert users. For example, a function **RealToString** may be defined as follows to convert a Real number into a String representation:

```
function RealToString
  input  Real number;
  input  Real precision = 6;
  input  Real minLength = 0;
  output String string;
algorithm
   ...
end RealToString;
```

Argument "number" is the number to be converted, "precision" is the number of significant digits in the String representation and "minLength" is the minimum length of the String in which the number is stored right justified. Since positional, named and default arguments are allowed, the following function applications are equivalent:

```
RealToString(2.0);
RealToString(2.0, 6, 0);
RealToString(2.0, 6);
RealToString(2.0, precision=6);
RealToString(2.0, minLength=0);
```

Note, that the following call leads to an error

```
RealToString(2.0, 6, precision=4);
```

since argument 2 is defined twice. This function may be used to conveniently build up a message string, such as

```
Variable "mass" (= -10.4562) shall
be non-negative.
```

via the function call

```
assert(v>=0,"Variable \"mass\" (="
      + RealToString(v) + " shall be "
      + "non-negative.\n"
```

As before, only positional output arguments of a function application are possible. However, output arguments shall be omitted, if the corresponding variables has attribute enable=**false** in the function declaration. This makes it possible to avoid dummy output arguments in the function application which are not used in the calling function. For example, a function to compute eigenvalues and optionally right and left eigenvectors may be defined in Modelica as:

```
function eigen
  parameter Integer n = size(A,1);
  input  Real    A[:,n];
  input  Boolean getREV = false;
  input  Boolean getLEV = false;
  output Real eigenValues[n,2];
  output Real REV[n,n](enable=getREV);
  output Real LEV[n,n](enable=getLEV);
algorithm
  // compute eigenvalues
  if getREV then
    // compute right eigenvectors
  end if;
  if getLEV then
    // compute left eigenvectors
  end if;
end eigen;
```

This function may be called to calculate only the eigenvalues of a matrix or to just determine whether a matrix has only stable eigenvalues:

```
  ev = eigen(A);
  b  = isStable(eigen(A));          //
```

to calculate eigenvalues and right eigenvectors:

```
  (ev, REV) = eigen(A, getREV=true);
```

to calculate additionally also the left eigenvectors:

```
  (ev, REV, LEV) = eigen(A, getREV=true,
                            getLEV=true);
```

7. Record Constructor

In Modelica 2, the missing constructor for the record data type is introduced. It is defined as a function with the same name and the same scope as the corresponding record containing all modifiable components of the record as input arguments and a record instance as output argument. Since a record constructor is just a function, it can be used at all places, where a function call is allowed. For example, with the following record declaration

```
  record Complex "Complex number"
    Real re "real part";
    Real im "imaginary part";
  end Complex;
```

a Complex data type is defined and implicitly its constructor function

```
  function Complex
    input  Real re "real part";
    input  Real im "imaginary part";
    output Complex out(re=re,im=im);
  end Complex;
```

Additionally, functions are needed, operating on this data type, such as:

```
  function add "Add Comp. numbers"
    input  Complex u, v;
    output Complex w(re=u.re + v.re,
                     im=u.im + v.im);
  end add;
```

The record constructor allows, e.g., to avoid the usage of unnecessary auxiliary variables:

```
    Complex c1, c2;
  equation
    c2 = add(c1, Complex(sin(time),
                         cos(time));
```

Note, that the second argument of the function application uses the record constructor to construct a temporary instance of type Complex.

Record constructors are very useful in situations where previously replaceable records have been needed (which are much less convenient to utilize). For example, a data sheet library of motors shall be constructed. The motor model consists essentially of two parts, one part containing all the data defining a particular motor as a record, e.g.,

```
  record MotorData
    parameter Real inertia;
    parameter Real nominalTorque;
    parameter Real maxTorque;
    parameter Real maxSpeed;
      ...
  end MotorData;
```

and the motor model utilizing the motor data

```
  model Motor
     MotorData data;
     // connector definitions
  equation
     ...
  end Motor;
```

When using a motor, specific values of the motor data could be given in the usual way:

```
  model Robot1
    Motor m1(data(inertia       = 0.001,
                  nominalTorque = 10,
                  maxTorque     = 20,
                  maxSpeed      = 3600));
    Motor m2(data(...));
      ...
  end Robot1;
```

When using the same motor type several times, it is better to define the motor data just ones, i.e., build up a **data sheet library** by **modifications** of the default values of the basic MotorData record:

```
package Motors
  record M103 = MotorData(
    inertia       = 0.001,
    nominalTorque = 10,
    maxTorque     = 20,
    maxSpeed      = 3600);

  record M104 = MotorData(
    inertia       = 0.0015,
    nominalTorque = 15,
    maxTorque     = 22,
    maxSpeed      = 3600);
  ...
end Motors;
```

Whenever one of the motors of package Motors is needed, it can be accessed by using the corresponding **record constructor**:

```
model Robot2
  Motor m1(data = Motors.M103());
  Motor m2(data = Motors.M104(
                    inertia=0.0012));
    ...
end Robot2;
```

It is still possible to override parameters in such a definition, see declaration of m2, by calling the record constructor function with appropriate positional or (preferably) named arguments.

8. Iterators

Modelica 2 introduces several enhancements to support more powerful expressions, especially in declarations, in order to avoid inconvenient local function definitions:

Deduction of Ranges

In all iterators, e.g., in for-loop, the expression to define the range of the iteration need not to be given if the *iterator variables* appear *as array indices*. In such cases the iteration range is deduced from the dimension sizes of the corresponding arrays. Example:

```
for i loop
  A[i] = B[i]^2;
end for;
```

A nested for loop

```
for i in 1:size(A,1) loop
  for j in 1:size(A,2) loop
    A[i,j] = B[i,j]^2;
  end for;
end for;
```

may be abbreviated as

```
for i in 1:size(A,1),
    j in 1:size(A,2) loop
    A[i,j] = B[i,j]^2;
end for;
```

or even shorter by automatic deduction of ranges

```
for i,j loop
  A[i,j] = B[i,j]^2;
end for;
```

Reduction Operators

An expression

```
function(expression for iterators);
```

is a **reduction-expression**. Currently, only the function names **sum**, **product**, **min**, and **max** can be used. The result is constructed by evaluating "expression" for each value of the iterator variable and computing the sum, product, minimum, or maximum of the computed elements. Examples:

```
sum(i for i in 1:10);
```

is the same as

$$\sum_{i=1}^{10} i = 1+2+...+10=55$$

A Modelica translator may transform this operation into:

```
algorithm
  result := 0;
  for i in 1:10 loop
    result := result + i;
  end for;
```

The sum of elements could also be defined as

```
sum(1:10);
```

using the built-in operator **sum**(). However, when summing up complex expressions or non-scalar expressions the reduction-expression can be made more readable than finding the appropriate vectorized expressions. As an example consider summing the squares instead:

```
sum(i^2 for i in 1:10);
```

The sum of squared elements could also be defined as

```
sum(diagonal(1:10)^2);
```

but even though it is slightly shorter it is not as readable.
Other examples are:

```
product(a[i,1]*s + a[i,2] for i);
```

is the same as

$$\prod_{i=1}^{n}(a_{i1}s + a_{i2}) = (a_{11}s + a_{12})\cdot(a_{21}s + a_{22})\cdot\ldots$$

As usual, a vector of values may be given as an iterator:

```
sum(i^2 for i in {1,3,7,6})
```

Gives $\sum_{i\in\{1\ \ 3\ \ 7\ \ 6\}} i^2 = 1+9+49+36=95$

```
max(i^2 for i in {3,7,6})
```

results in 49

Iterator Array Construction

In a similar way as a reduction operator, the construction

```
{expression for iterators};
```

with *n* iterators generates an array with *n* dimensions. The array is constructed by evaluating the expression for every iterator value and collecting the results to a corresponding array. Examples:

```
{i^2 for i in 1:5}
```

results in the vector

```
{1, 4, 9, 16, 25}
```

An (n,m) array having the same value v for all elements may be constructed as

```
{v for i in 1:n, j in 1:m}
```

which is the same as "fill(v,n,m)". The special matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

may be created with

```
{if i==j then i else 0
       for i in 1:n, j in 1:n}
```

9. External Utility Functions

Modelica 1.4 has already a convenient and simple to use interface for external C and FORTRAN procedures which allows to pass nearly all data types of Modelica. The only exceptions have been String types which could not be returned. In Modelica 2, the following utility functions can be called in external C functions:

```
void  ModelicaMessage
      (const char* string)
void  ModelicaFormatMessage
      (const char* string, ...)
void  ModelicaError
      (const char* string)
void  ModelicaFormatError
      (const char* string, ...)
char* ModelicaAllocateString
      (size_t len)
char*
ModelicaAllocateStringWithErrorReturn
      (size_t len)
```

ModelicaMessage and **ModelicaFormatMessage** output a string to the message window of the Modelica environment. The latter with the same format control as the C-function printf. In both cases linefeeds need to be explicitly defined in the string by "\n". Similarly, **ModelicaError** and **ModelicaFormatError** output an error to the error window of the Modelica environment. Contrary to the first two functions, these functions never return to the calling function, but handle the error similarly to an assert in the Modelica code. Example for usage:

```
ModelicaFormatError(
   "\"%s\" cannot be copied to \"%s\""
   ":\n%s", oldFile, newFile,
   strerror(errno));
```

Here, an error message is printed if a file cannot be copied. The error message of the operating system containing the source of the error is included at the end of the message by a call to the C function strerror(...).

ModelicaAllocateString allocates memory for a Modelica string which is used as return argument of an external Modelica function. If an error occurs, this function does not return. The Modelica environment is responsible to free this memory when appropriate. In a similar way **ModelicaAllocateStringWithErrorReturn** allocates string memory, but returns in case of error. This allows the external function to close files and free other open resources in case of error. Due to these two functions, Modelica Strings can now also be returned from external Modelica functions. For example, with the following external Modelica interface

```
function blanks
   input  Integer n(min=0);
   output String blankString;
   external "C"
      blankString = blanks(n);
end blanks;
```

a string containing n blanks shall be returned. An implementation of this function in C could be accomplished in the following way:

```
#include "ModelicaUtilities.h"

const char* blanks(int n) {
  /* Create string with n blanks */
  char *c = ModelicaAllocateString(n);
  int i;
  for(i=0; i<n; ++i)
    c[i]=' ';
  c[n]='\0';
  return c;
};
```

Note, that it is not necessary to check in the C-function that the input argument "n" is not negative, because this is already defined in the Modelica interface and therefore the Modelica environment is responsible to check this property. Furthermore, it needs not to be checked whether memory could be allocated, because ModelicaAllocateString will not return in such a case but will raise an exception in the Modelica run-time environment and will jump to a place where execution can continue, e.g., after terminating the simulation.

Note that the newly introduced enumeration types can also be used as input and output arguments in external functions. They are mapped to `int` in C and `INTEGER` in FORTRAN. The first value in an enumeration type is hereby mapped to 1, the second to 2, etc.

10. External Objects

Formally, external functions in Modelica 1.4 need to be functions in the mathematical sense, i.e., they do not have a memory and therefore return exactly the same result if the function is called with the same input arguments. In Modelica 2.0, additionally **external objects** are supported in C, i.e., several functions may operate on a C data structure which is passed between function calls and represents an "object memory". Example:

A table data structure may be defined in such a way, that the table data is read in a user defined format from file. Furthermore, the table is interpolated in a user defined manner in the Modelica model utilizing the last used table interval for efficiently finding the current interval, i.e., an internal memory is needed. This requires the following Modelica definition:

```
class MyTable
  extends ExternalObject;

  function constructor
    input  String  fileName;
    input  String  tableName;
    output MyTable table;
    external "C" table =
        initMyTable(fileName, tableName);
  end constructor;

  function destructor
    input  MyTable table;
    external "C" closeMyTable(table);
  end destructor;
end MyTable;
```

That is, a Modelica class has to be defined as a direct subclass of the new predefined class "**ExternalObject**". This class shall contain exactly two function definitions, called "constructor" and "destructor" (and no other elements). The constructor function is called once before the first use of the object. For each completely constructed object (here: instance of MyTable), the destructor is called once, after the last use of the object, even if an error occurs. These two functions are always called implicitly and it is not allowed to call them explicitly. The MyTable Modelica class can be used in a Modelica model in the following way:

```
model test
  MyTable table1=MyTable(
             "testTables.txt", "table1");
  MyTable table2=table1; //copy of table1
  input  Real u1, u2;
  output Real y1, y2;
equation
  y1 = interpolateMyTable(table1, u1);
```

```
  y2 = interpolateMyTable(table2, u2);
end test;
```

In the declaration of "MyTable" either the MyTable constructor is called using the class-name as a function name, or a copy of another object of the same type is constructed (see table2). The objects may then be used in other external Modelica functions. Here, a special external interpolation function is used:

```
function interpolateMyTable
  input  MyTable table;
  input  Real  u;
  output Real  y;
  external "C" y =
             interpolateMyTable(table, u);
end interpolateMyTable;
```

The three external functions defined above may be implemented in C in the following way:

```
typedef struct {
  double* array;
  int     nrow;
  int     ncol;
  int     type; /* interpolation type */
  int     lastIndex; /* for search */
} MyTable;

void* initMyTable(char* fileName,
                  char* tableName) {
  MyTable* table=malloc(sizeof(MyTable));
  if ( table == NULL ) ModelicaError(
                    "Not enough memory");
  // read table from file and store
  // all data in *table
  return (void*) table;
};

void closeMyTable(void* object) {
  MyTable* table = (MyTable*) object;
  if ( object == NULL ) return;
  free(table->array);
  free(table);
}

double interpolateMyTable(void* object,
                          double u) {
  MyTable* table = (MyTable*) object;
  double y;
  // Interpolate using "*table" data
  return y;
};
```

The external object interface allows, for example, convenient implementations of

- user-defined table data structures,
- access to property databases,
- sparse matrix handling with specially defined data structures to store sparse matrices,
- hardware interfaces, since the constructor and destructor are called exactly once, even in case of error, so that the resources of the hardware are initialized and freed correctly in all situations (once the hardware is initialized, i.e., the Modelica object constructed, it is guaranteed that the destructor is called exactly once for this object when the object, i.e., the hardware, is no longer needed or when an error occurs).

11. Graphical Appearance

The graphical appearance of Modelica object diagrams has been defined informally up to Modelica version 1.4 in the respective tutorial. In Modelica 2, the graphical appearance is formally defined in the Modelica specification with several improvements, especially based on the new enumeration features. In this section the most important properties are sketched. Note, that all graphical information is defined with the **annotation**(...) language element and annotations are defined to have no effect on the result of a simulation. Therefore, annotations can be ignored when generating simulation code.

A graphical representation of a class consists of two abstraction layers, **icon layer** and **diagram layer**. The icon representation visualizes the component by hiding hierarchical details. The hierarchical decomposition is described in the diagram layer showing icons of sub-components.

Icon and diagram layer are described by different coordinate systems which means that the shape and size of the two layers are independent from each other. This is different to previous versions of Modelica where only one coordinate system is defined for both layers. As a result, in Modelica 2 it is easier to arrive at nice looking drawings, because connectors may have different sizes in the icon and diagram layer and because a resizing of the diagram or the icon layer does not influence the size of the corresponding other layer. All size information, e.g., the size of icons and diagrams, the thickness of a line or the size of a font, is defined with the predefined type DrawingUnit:

```
type DrawingUnit =
              Real(final unit="mm");
```

The interpretation of "unit" in "mm" is with respect to printer output in natural size (not zoomed). Therefore, a rectangle with width=20 DrawingUnits, height = 10 DrawingUnits and line thickness of 0.5 DrawingUnits will be output as a

rectangle with 20 mm width, 10 mm height and 0.5 mm line thickness on a printer. The representation on screen is not formally defined. It is typically a direct mapping of "mm" to "pixels", e.g., 1 mm in "natural size" is typically mapped to 4 pixels. On high resolution screens, this mapping may be different.

The properties of graphical objects are mostly defined with enumerations, e.g.,

```
type LinePattern = enumeration(
            None, Solid, Dash, Dot,
               DashDot, DashDotDot);
```

Colors are defined as RGB values

```
type Color=Integer[3](min=0,max=255)
```

There is a set of predefined graphical primitives - Line, Polygon, Rectangle, Ellipse, Text, Bitmap - which may have graphical properties such as lineColor, fillColor, linePattern, fillPattern, borderPattern, lineThickness. For Text primitives, the font name and the font size can be defined. All graphical primitives are placed by defining the placement of the corresponding object coordinate system together with additional attributes to scale, rotate, flip the object.

Note: a Modelica tool is free to define and use other graphical attributes, in addition to those defined in the Modelica specification. The only requirement is that any tool must be able to save files with all attributes intact, including those that are not used. To ensure this, annotations shall be represented with constructs according to the Modelica grammar.

12. Outlook

We have this far described the status of Modelica 2.0. Some minor extensions have not been mentioned, such as the "smooth" operator and the "elseif" clause of if-expressions. In the near future we can also expect the Modelica 2.0 libraries, and in particular the blocks library, redesigned as described above. In addition a ModelicaFunctions library with matrix operations (linear algebra) will be made available and the new rules for variable number of input and output arguments will make it possible to provide one function easily usable both by experts and novices.

The ModelicaFunctions can also be used interactively, as well as other functions and we expect more use of Modelica scripts and potentially a formal definition of such scripts, and API-functions to access model properties from scripts. Other free libraries are also under development, e.g., for 1-dim. heat transfer and for 3-dim. vehicle dynamics.

From the language point of view some areas where improvements are needed is already clear, e.g., enumerations (as described above), impulses (eliminating the need for the reinit-operator [5]), heterogeneous arrays and PDEs (automatic discretization). More advanced use of the language and construction of large libraries and models will probably help in discovering areas where the specification can be made clearer and where further enhancement of the language is needed to better support the growing number of users of Modelica.

Bibliography

[1] Dymola, *Dynasim AB,* Lund, Sweden, version 5.0, http://www.dynasim.se.

[2] Mattsson S.E., Elmqvist H., and Olsson H.: Means to Control the Selection of States in Modelica (white paper of Dynasim), Nov. 2001.

[3] Mattsson S.E., Elmqvist H., Otter M., and Olsson H.: Initialization of Hybrid Differential-Algebraic Equations. Modelica 2002, Oberpfaffenhofen, pp. 9-15, March 18.-19., 2002.

[4] Mattsson S.E., Olsson H. and Elmqvist H: Dynamic Selection of States in Dymola. Modelica'2000, Oct. 2000.

[5] Mattsson S.E., Olsson H. and Elmqvist H: Varying structure Hybrid DAE (white paper of Dynasim), Jun. 2001.