Modelica™ - A Unified Object-Oriented Language for Physical Systems Modeling

Tutorial

Version 1.4
December 15, 2000

by the

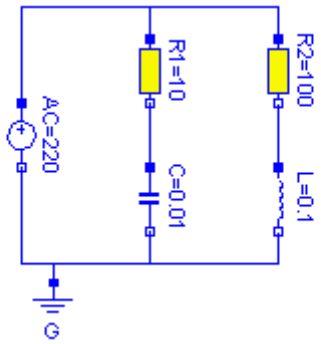
Modelica Association

Abstract:
This document is a tutorial for the Modelica language, version 1.4, which is developed by the Modelica Association, a non-profit organization with seat in Linköping, Sweden. Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical systems. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation and distribution of electric power. Models in Modelica are mathematically described by *differential, algebraic* and *discrete equations*. No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large models having more than hundred thousand equations. Modelica is suited and used for hardware-in-the-loop simulations and for embedded control systems. More information is available at
http://www.Modelica.org/

Contents

1. Modelica at a Glance

To give an introduction to Modelica we will consider modeling of a simple electrical circuit as shown below.



The system can be broken up into a set of connected electrical standard components. We have a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of these components are typically available in model libraries and by using a graphical model editor we can define a model by drawing an object diagram very similar to the circuit diagram shown above by positioning icons that represent the models of the components and drawing connections.

A Modelica description of the complete circuit looks like

```
model circuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  Ground    G;

equation
  connect (AC.p, R1.p);    // Capacitor circuit
  connect (R1.n, C.p);
  connect (C.n, AC.n);
  connect (R1.p, R2.p);    // Inductor circuit
  connect (R2.n, L.p);
  connect (L.n,  C.n);
  connect (AC.n, G.p);     // Ground
end circuit;
```

For clarity, the definition of the graphical layout of the composition diagram (here: electric circuit diagram) is not shown, although it is usually contained in a Modelica model as annotations (which are not processed by a Modelica translator and only used by tools). A composite model of this type specifies the topology of the system to be modeled. It specifies the components and the connections between the components. The statement

```
Resistor R1(R=10);
```

declares a component `R1` to be of class `Resistor` and sets the default value of the resistance, `R`, to 10. The connections specify the interactions between the components. In other modeling languages connectors are referred as cuts, ports or terminals. The language element **connect** is a special operator that generates equations taking into account what kind of quantities that are involved as explained below.

The next step in introducing Modelica is to explain how library model classes are defined.

A connector must contain all quantities needed to describe the interaction. For electrical components we need the quantities voltage and current to define interaction via a wire. The types to represent them are declared as

```
type Voltage = Real(unit="V");
type Current = Real(unit="A");
```

where `Real` is the name of a predefined variable type. A real variable has a set of attributes such as unit of measure, initial value, minimum and maximum value. Here, the units of measure are set to be the SI units.

In Modelica, the basic structuring element is a **class**. There are seven *restricted* classes with specific names, such as **model**, **type** (a class which is an extension of built-in classes, such as **Real**, or of other defined types), **connector** (a class which does not have equations and can be used in connections). For a valid model it is fully equivalent to, e.g., replace the **model**, and **type** keywords by the keyword **class**, because the restrictions imposed by such a specialized class are fulfilled by a valid model.

The concept of restricted classes is advantageous because the modeler does not have to learn several different concepts, but just one: the class concept. All properties of a class, such as syntax and semantic of definition, instantiation, inheritance, genericity are identical to all kinds of restricted classes. Furthermore, the construction of Modelica translators is simplified considerably because only the syntax and semantic of a **class** has to be implemented along with some additional checks on restricted classes. The basic types, such as `Real` or `Integer` are built-in **type** classes, i.e., they have all the properties of a class and the attributes of these basic types are just parameters of the class.

There are two possibilities to define a class: The standard way is shown above for the definition of the electric circuit (**model** circuit). A short hand notation is possible, if a new class is identical to an existing one and only the default values of attributes are changed. The types above, such as *Voltage*, are declared in this way.

A connector class is defined as

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

A connection **connect** `(Pin1, Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins such that they form one node. This implies two equations, namely `Pin1.v = Pin2.v` and `Pin1.i + Pin2.i = 0`. The first equation indicates that the voltages on both branches connected together are the same, and the second corresponds to Kirchhoff's current law saying

that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix **flow** is used. Similar laws apply to flow rates in a piping network and to forces and torques in mechanical systems.

When developing models and model libraries for a new application domain, it is good to start by defining a set of connector classes. A common set of connector classes used in all components in the library supports compatibility of the component models. In the Modelica Standard Library developed together with the Modelica Language, for many domains appropriate connector definitions are already available.

A common property of many electrical components is that they have two pins. This means that it is useful to define an "interface" model class,

```
partial model OnePort "Superclass of elements with two electrical pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

that has two pins, `p` and `n`, a quantity, `v`, that defines the voltage drop across the component and a quantity, `i`, that defines the current into the pin `p`, through the component and out from the pin `n`. The equations define generic relations between quantities of a simple electrical component. In order to be useful a constitutive equation must be added. The keyword **`partial`** indicates that this model class is incomplete. The key word is optional. It is meant as an indication to a user that it is not possible to use the class as it is to instantiate components. Between the name of a class and its body it is allowed to have a string. It is treated as a comment attribute and is meant to be a documentation that tools may display in special ways.

To define a model for a resistor we exploit OnePort and add a definition of parameter for the resistance and Ohm's law to define the behavior:

```
model Resistor "Ideal electrical resistor"
  extends OnePort;
  parameter Real R(unit="Ohm") "Resistance";
equation
    R*i = v;
end Resistor;
```

The keyword **parameter** specifies that the quantity is constant during a simulation run, but can change values between runs. A parameter is a quantity which makes it simple for a user to modify the behavior of a model.

A model for an electrical capacitor can also reuse the TwoPin as follows:

```
model Capacitor "Ideal electrical capacitor"
  extends OnePort;
  parameter Real C(unit="F") "Capacitance";
equation
  C*der(v) = i;
end Capacitor;
```

where **der**(v) means the time derivative of v. A model for the voltage source can be defined as

```
model VsourceAC "Sin-wave voltage source"
  extends OnePort;
  parameter Voltage VA = 220 "Amplitude";
  parameter Real f(unit="Hz") = 50  "Frequency";
  constant  Real PI=3.141592653589793;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;
```

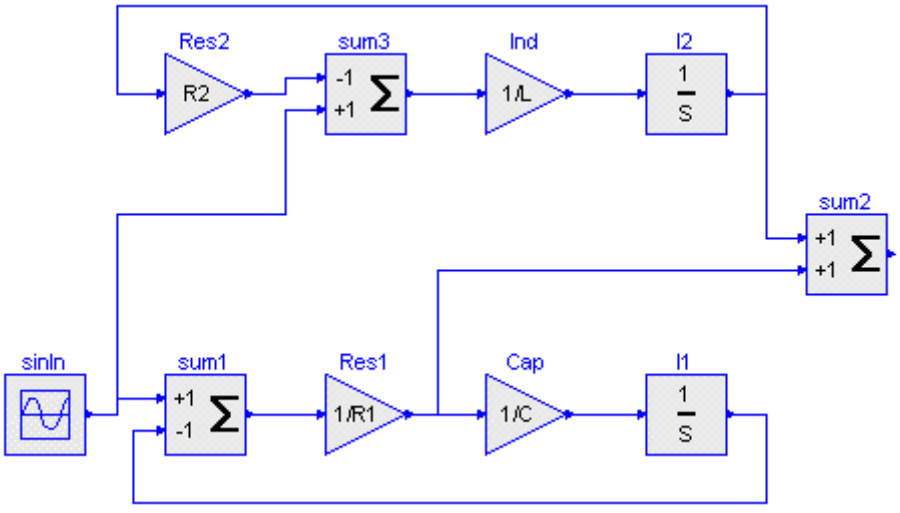
In order to provide not too much information at this stage, the constant PI is explicitly declared, although it is usually imported from the Modelica standard library (see appendix of the Language Specification). Finally, we must not forget the ground point.

```
model Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;
```

The purpose of the ground model is twofold. First, it defines a reference value for the voltage levels. Secondly, the connections will generate one Kirchhoff's current law too many. The ground model handles this by introducing an extra current quantity p.i, which implicitly by the equations will be calculated to zero.

Comparison with block oriented modeling

If the above model would be represented as a block diagram, the physical structure will not be retained as shown below. The block diagram is equivalent to a set of assignment statements calculating the state derivatives. In fact, Ohm's law is used in two different ways in this circuit, once solving for i and once solving for u.



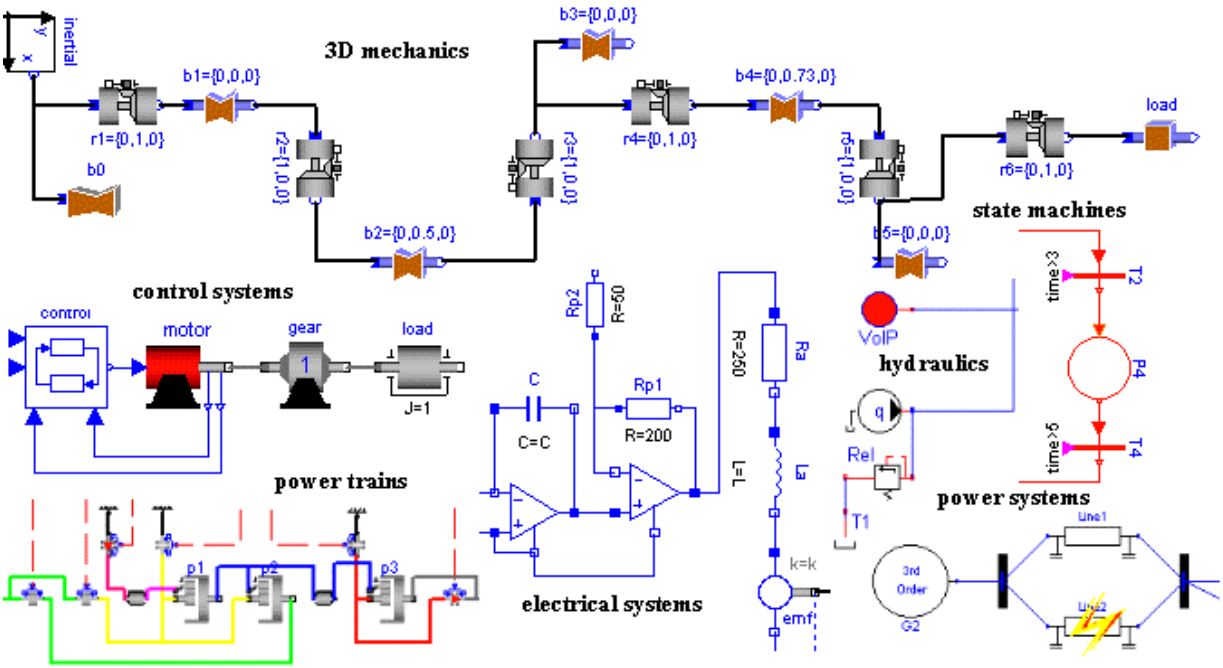
This example clearly shows the benefits of physically oriented, *non-causal modeling* compared to block oriented, causal modeling.

Modelica Libraries

In order that Modelica is useful for *model exchange*, it is important that libraries of the most commonly used components are available, ready to use, and sharable between applications. For this reason, the Modelica Association develops and maintains a growing *Modelica Standard Library*. Furthermore, other people and organizations are developing free and commercial Modelica libraries. For more information and especially for downloading the free libraries, see http://www.Modelica.org/library/library.html. Currently, component libraries are available in the following domains:

- About 450 type definitions, such as Angle, Voltage, Inertia.
- Mathematical functions such as sin, cos, ln
- Continuous and discrete input/output blocks, such as transfer functions, filters, sources.
- Electric and electronic components such as resistor, diode, MOS and BJT transistor.
- 1-dim. translational components such as mass, spring, stop.
- 1-dim. rotational components such as inertia, gearbox, planetary gear, bearing friction, clutch.
- 3-dim. mechanical components such as joints, bodies and 3-dim. springs.
- Hydraulic components, such as pumps, cylinders, valves.
- Thermo-fluid flow components, such as pipes with multi-phase flow, heat exchangers.
- 1-dim. thermal components, such as heat resistance and heat capacitance.
- Power system components such as generators and lines.
- Power train components such as driver, engine, torque converter, automatic gearboxes.

A screenshot of several examples built by available Modelica libraries is given below:



2. Modelica Language Overview

Modeling the dynamic behavior of physical systems implies that one is interested in specific properties of a limited class of systems. These restrictions give a means to be more specific then is possible when focusing on systems in general. Therefore, the physical background of the models should be reflected in Modelica.

Nowadays, physical systems are often complex and span multiple physical domains, whereas mostly these systems are computer controlled. Therefore, hierarchical models (i.e., models described as connected submodels) using properties of the physical domains involved should easily be described in Modelica. To properly support the modeler (i.e. to be able to perform automated modeling), these physical properties should be incorporated in Modelica in such a way, that checking consistency, like checking against basic laws of physics, can be programmed easily in the Modelica translators. Examples of physical properties are the physical quantity and the physical domain of a variable. This implies that a suitable representation for physical systems modeling is more than a set of pure mathematical differential equations.

2.1 Basic Language Elements

The language constructs will be developed gradually starting with small examples, and then extended by considering practical issues when modeling large systems.

Handling large models means careful structuring in order to reuse model knowledge. A model is built-up from

- basic components such as Real, Integer, Boolean and String
- structured components, to enable hierarchical structuring
- component arrays, to handle real matrices, arrays of submodels, etc
- equations and/or algorithms (= assignment statements)
- connections
- functions

Some means of declaring variable properties is needed, since there are different kinds of variables, Parameters should be given values and there should be a possibility to give initial conditions.

Basic declarations of variables can be made as follows:

```
Real u, y(start=1);
parameter Real T=1;
```

Real is the name of a predefined class or type. A Real variable has an attribute called `start` to give its initial value. A component declaration can be preceded by a *specifier* like **constant** or **parameter** indicating that the component is constant, i.e., its derivative is zero. The specifier parameter indicates that the value of the quantity is constant during simulation runs. It can be

modified when a component is reused and between simulation runs. The component name can be followed by a *modification* to change the value of the component or its attributes.

Equations are composed of expressions both on the left hand side and the right hand side like in the following filter equation.

```
equation
T*der(y) + y = u;
```

Time derivative is denoted by **der**( ).

2.2 Classes for Reuse of Modeling Knowledge

Assume we would like to connect two filters in series. Instead of repeating the filter equation, it is more convenient to make a definition of a filter once and create two instances. This is done by declaring a *class*. A class declaration contains a list of component declarations and a list of equations preceded by the keyword **equation**. An example of a low pass filter class is shown below.

```
class LowPassFilter
  parameter Real T=1;
  Real u, y(start=1);

equation
  T*der(y) + y = u;
end LowPassFilter;
```

The model class can be used to create two instances of the filter with different time constants and "connecting" them together as follows

```
class FiltersInSeries
  LowPassFilter F1(T=2), F2(T=3);

equation
  F1.u = sin(time);
  F2.u = F1.y;
end FiltersInSeries;
```

In this case we have used a *modification* to modify the time constant of the filters to T=2 and T=3 respectively from the default value T=1 given in the low-pass filter class. Dot notation is used to reference components, like u, within structured components, like F1. For the moment it can be assumed that all components can be reached by dot-notation. Restrictions of accessibility will be introduced later. The independent variable is referenced as **time**. It is available in all classes without declaration.

If the FiltersInSeries model is used to declare components at a higher hierarchical level, it is still possible to modify the time constants by using a hierarchical *modification:*

```
model ModifiedFiltersInSeries
  FiltersInSeries F12(F1(T=6), F2(T=11, k=2));   // alternative 1
  FiltersInSeries F34(F1.T=6, F2.T=11, F2.k=2);  // alternative 2
end ModifiedFiltersInSeries;
```

The class concept is similar as in programming languages. It is used for many purposes in Modelica, such as model components, connection mechanisms, parameter sets, input-output

blocks and functions. In order to make Modelica classes easier to read and to maintain, special keywords have been introduced for such special uses, **model**, **connector**, **record**, **block**, **function, type** and **package**. It should be noted though that the use of these keywords only apply certain restrictions, like records are not allowed to contain equations. However, for a valid model, the replacement of these keywords by **class** would give exactly the same model behavior. In the following description we will use the specialized keywords in order to convey their meaning.

Records

It is possible to introduce parameter sets as *record*s which is a restricted form of class which may not have any equations:

```
record FilterData
  Real T;
end FilterData;

record TwoFilterData
  FilterData F1, F2;
end TwoFilterData;

model ModifiedFiltersInSeries2
  TwoFilterData TwoFilterData1(F1(T=6), F2(T=11));

  FiltersInSeries F12=TwoFilterData1;
end ModifiedFiltersInSeries2;
```

The modification `F12=TwoFilterData1` is possible since all the components of `TwoFilterData1 (F1, F2, T) are` present in `FiltersInSeries`. More about type compatibility can be found in section 4.4.

Packages

Class declarations may be nested. One use of that is maintenance of the name space for classes, i.e., to avoid name clashes, by storing a set of related classes within an enclosing class. There is a special kind of class for that, called **package**. A package may only contain declarations of constants and classes. Dot-notation is used to refer to the inner class. Examples of packages are given in the appendix of the Language Specification, where the Modelica standard package is described which is always available for a Modelica translator.

Information Hiding

So far we have assumed all components to be accessible from the outside by dot-notation. To develop libraries in such a way is a bad principle. Information hiding is essential from a maintenance point of view.

Considering the FiltersInSeries example, it might be a good idea to just declare two parameters for the time constants, T1 and T2, the input, u and the output y as accessible from the outside. The realization of the model, using two instances of model LowPassFilter, is a protected detail. Modelica allows such information hiding by using the heading **protected**.

```
model FiltersInSeries2
  parameter Real T1=2, T2=3;
  input Real u;
  output Real y;

protected
  LowPassFilter F1(T=T1), F2(T=T2);

equation
  F1.u = u;
  F2.u = F1.y;
  y = F2.y;
end FiltersInSeries2;
```

Information hiding does not control interactive environments though. It is possible to inspect and plot protected variables. Note, that variables of a **protected** section of a class A can be accessed by a class which **extends** class A. In order to keep Modelica simple, additional visibility rules present in other object-oriented languages, such as *private* (no access by subtypes), are not used.

2.3 Connections

We have seen how classes can be used to build-up hierarchical models. It will now be shown how to define physical connections by means of a restricted class called **connector**.

We will study modeling of a simple electrical circuit. The first issue is then how to represent pins and connections. Each pin is characterized by two variables, voltage and current. A first attempt would be to use a connector as follows.

```
connector Pin
  Real v, i;
end Pin;
```

and build a resistor with two pins p and n like

```
model Resistor
  Pin p, n;        // "Positive" and "negative" pins.
  parameter Real R "Resistance";

equation
    R*p.i = p.v - n.v;
    n.i = p.i;     // Assume both n.i and p.i to be positive
                   // when current flows from p to n.
end Resistor;
```

A descriptive text string enclosed in " " can be associated with a component like R. A comment which is completely ignored can be entered after //. Everything until the end of the line is then ignored. Larger comments can be enclosed in /* ... */.

A simple circuit with series connections of two resistors would then be described as:

```
model FirstCircuit
  Resistor R1(R=100), R2(R=200);

equation
  R1.n = R2.p;
end FirstCircuit;
```

The equation R1.n = R2.p represents the connection of pin n of R1 to pin p of R2. The semantics of this equation on structured components is the same as

```
R1.n.v = R2.p.v
R1.n.i = R2.p.i
```

This describes the series connection correctly because only two components were connected. Some mechanism is needed to handle Kirchhoff's current law, i.e. that the currents of all wires connected at a node are summed to zero. Similar laws apply to flows in a piping network and to forces and torques in mechanical systems. The default rule is that connected variables are set equal. Such variables are called *across* variables. Real variables that should be summed to zero are declared with prefix **flow**. Such variables are also called *through* variables. In Modelica we assume that such variables are positive when the flow (or corresponding vector) is into the component.

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

It is useful to introduce *units* in order to enhance the possibility to generate diagnostics based on redundant information. Modelica allows deriving new classes with certain modified attributes. The keyword **type** is used to define a new class, which is derived from the built-in data types or defined records. Defining Voltage and Current as modifications of Real with other attributes and a corresponding Pin can thus be made as follows:

```
type Voltage = Real(unit="V");
type Current = Real(unit="A");

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

model Resistor
  Pin p, n;     // "Positive" and "negative" pins.
  parameter Real R(unit="Ohm")  "Resistance";

equation
    R*p.i = p.v - n.v;
    p.i + n.i = 0;     // Positive currents into component.
end Resistor;
```

We are now able to correctly connect three components at one node.

```
model SimpleCircuit
  Resistor R1(R=100), R2(R=200), R3(R=300);

equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit;
```

connect is a special operator that generates equations taking into account what kind of variables that are involved. The equations are in this case equivalent to

```
R1.p.v = R2.p.v;
R1.p.v = R3.p.v;
R1.p.i + R2.p.i + R3.p.i = 0;
```

In certain cases, a model library might be built on the assumption that only one connection can be made to each connector. There is a built-in function **cardinality**(c) that returns the number of connections that has been made to a connector c. It is also possible to get information about the direction of a connection by using the built-in function **direction**(c) (provided cardinality(c) == 1). For a connection, connect(c1, c2), direction(c1) returns -1 and direction(c2) returns 1. The cardinality and direction operators can be, e.g., used to build a library of bond graph components.

2.4 Partial Models and Inheritance

A very important feature in order to build reusable descriptions is to define and reuse *partial models*. Since there are other electrical components with two pins like capacitor and inductor we can define a generic component with one electrical port[1], model OnePort, having two pins as a base for all of these models.

```
partial model OnePort
  Pin p, n;
  Voltage v  "Voltage drop";

equation
  v = p.v - n.v;
  p.i + n.i = 0;
end TwoPin;
```

Such a partial model can be extended or reused to build a complete model like an inductor.

```
model Inductor "Ideal electrical inductance"
  extends OnePort;
  parameter Real L(unit="H") "Inductance";
equation
  L*der(i) = v;
end Inductor;
```

The facility is similar to inheritance in other languages. Multiple inheritance, i.e., several **extends** statements, is supported.

The type system of Modelica is greatly influenced by type theory (Abadi and Cardelli 1996), in particular their notion of subtyping. Abadi and Cardelli separate the notion of subclassing (the mechanism for inheritance) from the notion of subtyping (the structural relationship that determines type compatibility). The main benefit is added flexibility in the composition of types, while still maintaining a rigorous type system.

Inheritance is not used for classification and type checking in Modelica. An **extends** clause can

[1] In the Modelica Standard Library model OnePort is a component where the current flowing into the first pin is the same as the current flowing out of the second pin, where as model TwoPin is a component where this assumption does not necessarily hold, e.g., because the component consists internally of a non-trivial electrical circuit.

be used for creating a subtype relationship by inheriting all components of the base class, but it is not the only means to create it. Instead, a class A is defined to be a *subtype* of class B, if class A contains all the public components of B. In other words, B contains a *subset* of the components declared in A. This subtype relationship is especially used for class parameterization as explained in the next section.

Assume, for example, that a more detailed resistor model is needed, describing the temperature dependency of the resistance:

```
model TempResistor "Temperature dependent electrical resistor"
  extends OnePort;
  parameter Real R(unit="Ohm")               "Resistance for ref. Temp.";
  parameter Real RT(unit="Ohm/degC")=0     "Temp. dep. Resistance.";
  parameter Real Tref(unit="degC")=20     "Reference temperature.";
  Real Temp=20                             "Actual temperature";
equation
    v = p.i*(R + RT*(Temp-Tref));
end TempResistor;
```

It is not possible to extend this model from the ideal resistor model `Resistor` discussed in Chapter 2, because the equation of the `Resistor` class needs to be replaced by a new equation. Still, the `TempResistor` is a subtype of `Resistor` because it contains all the public components of `Resistor`.

2.5 Class Parameterization

We will now discuss a more powerful parameterization, not only involving values like time constants and matrices but also classes. (This section might be skipped during the first reading.) Assume that we have the description (of an incomplete circuit) as above.

```
model SimpleCircuit
  replaceable Resistor R1(R=100), R2(R=200), R3(R=300);

equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit;
```

Assume we would like to utilize the parameter values given for R1.R and R2.R and the circuit topology, but exchange Resistor with the temperature dependent resistor model, TempResistor, discussed above. This can be accomplished by redeclaring R1 and R2 as follows.

```
model RefinedSimpleCircuit
  Real Temp;
  extends SimpleCircuit(
    redeclare TempResistor R1(RT=0.1, Temp=Temp),
    redeclare TempResistor R2);
end RefinedSimpleCircuit;
```

Since TempResistor is a *subtype* of Resistor, it is possible to replace the ideal resistor model. Values of the additional parameters of TempResistor and definition of the actual temperature can be added in the redeclaration:

```
redeclare TempResistor R1(RT=0.1, Temp=Temp);
```

This is a very strong modification of the circuit model and there is the issue of possible invalidation of the model. For this reason, such modifications have to be marked by the keyword **redeclare**. Furthermore, the model developer has to explicitly allow such type of modification by declaring a component as replaceable: -

```
replaceable Resistor R3(R=300);
```

It is also possible to state that a parameter is frozen to a certain value, i.e., is not a parameter anymore:

```
Resistor R3(final R=300);
```

and can therefore no longer be changed by a modification (including a redeclaration). In some situations it may be necessary that the basic constraining type is different from the default type, for example,

```
model SimpleCircuit
  replaceable Resistor R1(R=100) extends OnePort;
  replaceable Resistor R2(R=200) extends OnePort;
  Resistor R3(R=300);

equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit;
```

Here, the resistors R1 and R2 can be replaced by any electrical component which is a subtype of model OnePort, as in the following redeclaration:

```
model RefinedSimpleCircuit2
   extends SimpleCircuit(redeclare Capacitor R1(C=0.001));
end RefinedSimpleCircuit2;
```

To use another resistor model in the model SimpleCircuit, we needed to know that there were two replaceable resistors and we needed to know their names. To avoid this problem and prepare for replacement of a set of models, one can define a *replaceable class*, ResistorModel. The actual class that will later be used for R1 and R2 must have Pins p and n and a parameter R in order to be compatible with how R1 and R2 are used within SimpleCircuit2. The replaceable model ResistorModel is declared to be a Resistor model. This means that it will be enforced that the actual class will be a subtype of Resistor, i.e., have compatible connectors and parameters. Default for ResistorModel, i.e., when no actual redeclaration is made, is in this case Resistor. Note, that R1 and R2 are in this case of class ResistorModel.

```
model SimpleCircuit2
  replaceable model ResistorModel = Resistor;

protected
  ResistorModel R1(R=100), R2(R=200);
  Resistor R3(final R=300);

equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit2;
```

Binding an actual model TempResistor to the replaceable model ResistorModel is done as follows.

```
model RefinedSimpleCircuit2 =
  SimpleCircuit2(redeclare model ResistorModel = TempResistor);
```

Another case where redeclarations are needed is extensions of interfaces. Assume we have a definition for a Tank in a model library:

```
connector Stream
  Real pressure;
  flow Real volumeFlowRate;
end Stream;

model Tank
  parameter Real Area=1;
  replaceable connector TankStream = Stream;
  TankStream Inlet, Outlet;
  Real level;

equation
  // Mass balance.
  Area*der(level) = Inlet.volumeFlowRate + Outlet.volumeFlowRate;
  Outlet.pressure = Inlet.pressure;
end Tank;
```

We would like to extend the Tank to model the temperature of the stream. This involves both extension to interfaces and to model equations.

```
connector HeatStream
  extends Stream;
  Real temp;
end HeatStream;

model HeatTank
  extends Tank(redeclare connector TankStream = HeatStream);
  Real temp;

equation
  // Energy balance.
  Area*Level*der(temp) = Inlet.volumeFlowRate*Inlet.temp +
    Outlet.volumeFlowRate*Outlet.temp;
  Outlet.temp = temp;  // Perfect mixing assumed.
end HeatTank;
```

The definition of HeatTank above is equivalent to the following definition (which has been automatically produced by a Modelica translator).

```
model HeatTankT
  parameter Real Area=1;

  connector TankStream
    Real pressure;
    flow Real volumeFlowRate;
    Real temp;
  end TankStream;

  TankStream Inlet, Outlet;
  Real level;
  Real temp;
equation
```

```
  Area*der(level) = Inlet.volumeFlowRate + Outlet.volumeFlowRate;
  Outlet.pressure = Inlet.pressure;
  Area*level*der(temp) = Inlet.volumeFlowRate*Inlet.temp +
    Outlet.volumeFlowRate*Outlet.temp;
  Outlet.temp = temp;
end HeatTankT;
```

Replaceable classes are also very convenient to separate fluid properties from the actual device where the fluid is flowing, such as a pump.

2.6 Matrices and Arrays

An array variable can be declared by appending dimensions after the class name or after a component name.

```
Real[3] position, velocity, acceleration;
Real[3,3] transformation;
Real[3,2,10] table;
```

or

```
Real position[3], velocity[3], acceleration[3], transformation[3, 3];
Real table[3,2,10];
```

It is also possible to make a matrix type

```
type Transformation = Real[3, 3];
Transformation transformation;
```

The following definitions are appropriate for modeling 3D motion of mechanical systems.

```
type Position = Real(unit="m");
type Position3 = Position[3];

type Force = Real(unit="N");
type Force3 = Force[3];

type Torque = Real(unit="N.m");
type Torque3 = Torque[3];
```

It is now possible to introduce the variables that are interacting between rigidly connected bodies in a free-body diagram.

```
connector MbsCut
  Transformation S  "Rotation matrix describing frame A"
                    " with respect to the inertial frame";
  Position3      r0 "Vector from the origin of the inertial"
                    " frame to the origin of frame A";
  flow Force3    f  "Resultant cut-force acting at the origin"
                    " of frame A";
  flow Torque3   t  "Resultant cut-torque with respect to the"
                    " origin of frame A";
end MbsCut;
```

Such a definition can be used to model a rigid bar as follows.

```
model Bar "Massless bar with two mechanical cuts."
  MbsCut a b;
  parameter    Position3 r = {0, 0, 0}
    "Position vector from the origin of cut-frame A"
```

```
    " to the origin of cut-frame B";

equation
  // Kinematic relationships of cut-frame A and B
  b.S   = a.S;
  b.r0  = a.r0 + a.S*r;

  // Relations between the forces and torques acting at
  //   cut-frame A and B
  zeros(3) = a.f + b.f;
  zeros(3) = a.t + b.t - cross(r, a.f);
  // The function cross defines the cross product
  //   of two vectors
end Bar;
```

Vector and matrix expressions are formed in a similar way as in Mathematica and Matlab. The operators +, -, * and / can operate on either scalars, vectors or two-dimensional matrices of type Real and Integer. Division is only possible with a scalar. An array expression is constructed as $\{expr_1, expr_2, ... expr_n\}$. A matrix (two dimensional array) can be formed as

$$[expr_{11}, expr_{12}, ... expr_{1n};\ expr_{21}, expr_{22}, ... expr_{2n};\ \ldots\ expr_{m1}, expr_{m2}, ... expr_{mn}]$$

i.e. with commas as separators between columns and semicolon as separator between rows. Indexing is written as A[i] with the index starting at 1. Submatrices can be formed by utilizing : notation for index ranges, A[i1:i2, j1:j2]. The then and else branches of if-then-else expressions may contain matrix expressions provided the dimensions are the same. There are several built-in matrix functions like zeros, ones, identity, transpose, skew (skew operator for 3 x 3 matrices) and cross (cross product for 3-dimensional vectors). For details about matrix expressions and available functions, see the Language Specification.

Matrix sizes and indices in equations must be constant during simulation. If they depend on parameters, it is a matter of "quality of implementation" of the translator whether such parameters can be changed at simulation time or only at compilation time.

Block Diagrams

We will now illustrate how the class concept can be used to model block diagrams as a special case. It is possible to postulate the data flow directions by using the prefixes **input** and **output** in declarations. This also allows checking that only one connection is made to an input, that outputs are not connected to outputs and that inputs are not connected to inputs on the same hierarchical level.

A matrix can be declared without specific dimensions by replacing the dimension with a colon: A[:, :]. The actual dimensions can be retrieved by the standard function **size**. A general state space model is an input-output **block** (restricted class, only inputs and outputs) and can be described as

```
block StateSpace
  parameter Real A[:, :],
                B[size(A, 1), :],
                C[:, size(A, 2)],
                D[size(C, 1), size(B, 2)]=zeros(size(C, 1), size(B, 2));
  input     Real u[size(B, 2)];
  output    Real y[size(C, 1)];
protected
  Real   x[size(A, 2)];

equation
  assert(size(A, 1) == size(A, 2), "Matrix A must be square.");
  der(x) = A*x + B*u;
  y      = C*x + D*u;
end StateSpace;
```

Assert is a predefined function for giving error messages taking a Boolean condition and a string as arguments. The actual dimensions of A, B and C are implicitly given by the actual matrix parameters. D defaults to a zero matrix:

```
block TestStateSpace
  StateSpace S(A = [0.12, 2;  3, 1.5], B = [2, 7;  3, 1], C = [0.1, 2]);

equation
  S.u = {time, sin(time)};
end TestStateSpace;
```

The **block** class is introduced to allow better diagnostics for pure input/output model components. In such a case the correctness of the component can be analyzed locally which is not possible for components where the causality of the public variables is unknown.

2.7 Repetition, Algorithms and Functions

Regular Equation Structures

Matrix equations are in many cases convenient and compact notations. There are, however, cases when indexed expressions are easier to understand. A loop construct, **for,** which allow indexed expressions will be introduced below.

Consider evaluation of a polynomial function

$$y = \sum_{i=0}^{n} c_i x^i$$

with a given set of coefficients c_i in a vector a[n+1] with $a[i] = c_{i-1}$. Such a sum can be expressed in matrix form as a scalar product of the form

```
a * {1, x, x^2, ... x^n}
```

if we could form the vector of increasing powers of x. A recursive formulation is possible.

```
xpowers[1] = 1;
xpowers[2:n+1] = xpowers[1:n]*x;
y = a * xpowers;
```

The recursive formulation would be expanded to

```
xpowers[1] = 1;
xpowers[2] = xpowers[1]*x;
xpowers[3] = xpowers[2]*x;
...
xpowers[n+1] = xpowers[n]*x;
y = a * xpowers;
```

The recursive formulation above is not so understandable though. One possibility would have been to introduce a special matrix operator for element exponentiation as in Matlab (.^). The readability does not increase much though.

Matrix equations like

```
xpowers[2:n+1] = xpowers[1:n]*x;
```

can be expressed in a form that is more familiar to programmers by using a for loop:

```
for i in 1:n loop
  xpowers[i+1] = xpowers[i]*x;
end for;
```

This for-loop is equivalent to n equations. It is also possible to use a block for the polynomial evaluation:

```
block PolynomialEvaluator
  parameter Real a[:];
  input  Real x;
  output Real y;

protected
  parameter Integer n = size(a, 1)-1;
  Real xpowers[n+1];

equation
  xpowers[1] = 1;
  for i in 1:n loop
    xpowers[i+1] = xpowers[i]*x;
  end for;
  y = a * xpowers;
end PolynomialEvaluator;
```

The block can be used as follows:

```
  PolynomialEvaluator polyeval(a={1, 2, 3, 4});
  Real p;
equation
  polyeval.x = time;
  p = polyeval.y;
```

It is also possible to bind the inputs and outputs in the parameter list of the invocation.

```
PolynomialEvaluator polyeval(a={1, 2, 3, 4}, x=time, y=p);
```

Regular Model Structures

The **for** construct is also essential in order to make regular connection structures for component arrays, for example:

```
Component components[n];
equation
  for i in 1:n-1 loop
    connect(components[i].Outlet, components[i+1].Inlet);
  end for;
```

Algorithms

The basic describing mechanism of Modelica are *equations* and not assignment statements. This gives the needed flexibility, e.g., that a component description can be used with different causalities depending on how the component is connected. Still, in some situations it is more convenient to use assignment statements. For example, it might be more natural to define a digital controller with ordered assignment statements since the actual controller will be implemented in such a way.

It is possible to call external functions written in other programming languages from Modelica and to use all the power of these programming languages. This can be quite dangerous because many difficult-to-detect errors are possible which may lead to simulation failures. Therefore, this should only be done by the simulation specialist if tested legacy code is used or if a Modelica implementation is not feasible. In most cases, it is better to use a Modelica **algorithm** which is designed to be much more secure than calling external functions.

The vector `xvec` in the polynomial evaluator above had to be introduced in order that the number of unknowns are the same as the number of equations. Such a recursive calculation scheme is often more convenient to express as an algorithm, i.e., a sequence of assignment statements, if-statements and loops, which allows multiple assignments:

```
algorithm
  y := 0;
  xpower := 1;
  for i in 1:n+1 loop
    y := y + a[i]*xpower;
    xpower := xpower*x;
  end for;
```

A Modelica algorithm is a function in the *mathematical sense*, i.e. without internal memory and side-effects. That is, whenever such an algorithm is used with the same inputs, the result will be exactly the same. If a function is called during *continuous* integration this is an absolute prerequisite. Otherwise the mathematical assumptions on which the integration algorithms are based on, would be violated. An internal memory in an algorithm would lead to a model giving different results when using different integrators. With this restriction it is also possible to symbolically form the Jacobian by means of automatic differentiation. This requirement is also present for functions called only at **event** instants (see below). Otherwise, it would not be possible to restart a simulation at any desired time instant, because the simulation environment does not know the actual value of the internal algorithm memory.

In the **algorithm** section, ordered assignment statements are present. To distinguish from equations in the **equation** sections, a special operator, :=, is used in assignments (i.e. given causality) in the **algorithm** section. Several assignments to the same variable can be performed

in one algorithm section. Besides assignment statements, an algorithm may contain if-then-else expressions, if-then-else constructs (see below) and loops using the same syntax as in an equation-section.

Variables that appear on the left hand side of the assignment operator, which are conditionally assigned, are *initialized* to their start value (for algorithms in functions, the value given in the binding assignment) *whenever the algorithm is invoked.* Due to this feature it is impossible for a function or an algorithm section to have a memory. Furthermore, it is guaranteed that the output variables always have a well-defined value.

Within an equation section of a class, algorithms are treated as a sets of equations. Especially, algorithms are sorted together with all other equations. For the sorting process, the calling of a function with *n* output arguments is treated as *n* implicit equations, where **every** equation depends on all output and on all input arguments. This ensures that the implicit equations remain together during sorting (and can be replaced by the algorithm invocation afterwards), because the implicit equations of the function form one algebraic loop.

In addition to the for loop, there is a while loop which can be used within algorithms:

```
while condition loop
  { algorithm }
end while;
```

Functions

The polynomial evaluator above is a special input-output block since it does not have any states. Since it does not have any memory, it would be possible to invoke the polynomial function as a function, i.e. memory for variables are allocated temporarily while the algorithm of the function is executing. Modelica allows a specialization of a class called *function* which has only public inputs and outputs, one algorithm and no equations.

The polynomial evaluation can thus be described as:

```
function PolynomialEvaluator2
  input  Real a[:];
  input  Real x;
  output Real y;

protected
  Real   xpower;

algorithm
  y := 0;
  xpower := 1;
  for i in 1:size(a, 1) loop
    y := y + a[i]*xpower;
    xpower := xpower*x;
  end for;
end PolynomialEvaluator2;
```

A function declaration is similar to a class declaration but starts with the **function** keyword. The input arguments are marked with the keyword **input** (since the causality is input). The result argument of the function is marked with the keyword **output**.

No internal states are allowed, i.e., the der- and pre- operators are not allowed. Any class can be used as an input and output argument. All public, non-constant variables of a class in the output argument are the outputs of a function.

Instead of creating a polyeval object as was needed for the block PolynomialEvaluator:

```
PolynomialEvaluator polyeval(a={1, 2, 3, 4}, x=time, y=p);
```

it is possible to invoke the function as usual in an expression.

```
p = PolynomialEvaluator2(a={1, 2, 3, 4}, x=time);
```

It is also possible to invoke the function with positional association of the actual arguments:

```
p = PolynomialEvaluator2({1, 2, 3, 4}, time);
```

A function can have several output arguments:

```
function Circle
  input  Real angle;
  input  Real radius;
  output Real x;
  output Real y;
algorithm
  x = radius*Modelica.Math.cos(phi);
  y = radius*Modelica.Math.sin(phi);
end Circle;
```

Such a function is called in the following way:

```
(x,y) = Circle(1.2, 2);
```

i.e., the actual values of all input variables are provided in the list after the function name and the return values of this function are provided at the left hand side of the equal sign enclosed in parentheses.

External functions

It is possible to call functions defined outside of the Modelica language. The body of an external function is marked with the keyword **external**:

```
function log
  input Real x;
  output Real y;
external
end log;
```

There is a "natural" mapping from Modelica to the target language and its standard libraries. The C language is used as the least common denominator.

The arguments of the external function are taken from the Modelica declaration. If there is a scalar output, it is used as the return type of the external function; otherwise the results are returned through extra function parameters. Arrays of simple types are mapped to an argument of the simple type, followed by the array dimensions. Storage for arrays as return values is allocated by the calling routine, so the dimensions of the returned array is fixed. It is possible to specify exactly the order of the arguments for the external C-function as well as the name of the C-

function, to define the allowed dependencies between array dimensions, and to provide internal work arrays. These features are demonstrated by the following quite complicated function interface:

```
function BilinearSampling
  "Slicot function for Discrete-time <--> continuous-time
   systems conversion by a bilinear transformation."
  input  Real alpha=1, beta=1;
  input  Real A[:, size(A, 1)], B[size(A, 1), :],
              C[:, size(A, 1)], D[size(C, 1), size(B, 2)];
  input  Boolean isContinuous = true;
  output Real Ares[size(A, 1), size(A, 2)]=A, // Ares is in-out
              Bres[size(B, 1), size(B, 2)]=B,
              Cres[size(C, 1), size(C, 2)]=C,
              Dres[size(D, 1), size(D, 2)]=D;
  output Integer info;
protected
  Integer iwork[size(A, 1)]; // Work arrays
  Real    dwork[size(A, 1)];
  String  c2dstring=if isContinuous then "C" else "D";
external "C" ab04md(c2dstring,size(A,1),size(B,2),size(C,1),
      alpha,beta,Ares,size(Ares,1),Bres,size(Bres,1),
      Cres,size(Cres,1),Dres,size(Dres,1),
      iwork,dwork,size(dwork,1),info);
end BilinearSampling;
```

It is expected that an external C-function is available which has the following prototype:

```
void ab04md(const char *, size_t, size_t, size_t, double, double,
            double *, size_t, double *, size_t, double *, size_t,
            double *, size_t, int *, double *, size_t, int *);
```

and the Modelica translator maps a function call of BilinearSampling to a function call of the C-function ab04md. Within Modelica, this function is called as:

```
  parameter Real alpha=1, beta=1;
  parameter Real A[:,:] = [0, 1; 2, 4], B[:,:]=...;
            Real Ares[size(A, 1), size(A, 2)], Bres ...;
equation
  (Ares,Bres,Cres,Dres,info) = BilinearSampling(alpha,beta,A,B,C,D,true);
```

More details, especially the exact mapping of the Modelica types to C and Fortran 77 types, are discussed in the appendix of the Modelica Language Specification.

2.8 Hybrid Models

Modelica can be used for mixed continuous and discrete models. For the discrete parts, the synchronous data flow principle with the single assignment rule is used. This fits well with the continuous DAE with equal number of equations as variables. Certain inspiration for the design has been obtained from the languages Signal (Gautier, et.al., 1994) and Lustre (Halbwachs, et.al. 1991).

Discontinuous Models

If-then-else expressions allow modeling of a phenomena with different expressions in different

operating regions. A limiter can thus be written as

```
y = if      u > HighLimit then HighLimit
    else if u < LowLimit  then LowLimit else u;
```

This construct might introduce discontinuities. If this is the case, appropriate information about the crossing points should be provided to the integrator. The use of crossing functions is described later.

More drastic changes to the model might require replacing one set of equations with another depending on some condition. It can be described as follows using vector expressions:

```
zeros(3) = if cond_A then
  { expression_A1l - expression_A1r,
    expression_A2l - expression_A2r }
else if cond_B then
  { expression_B1l - expression_B1r,
    expression_B2l - expression_B2r }
else
  { expression_C1l - expression_C1r,
    expression_C2l - expression_C2r };
```

The size of the vectors must be the same in all branches, i.e., there must be *equal number of expressions (equations)* for all conditions.

It should be noted that the order of the equations in the different branches is important. In certain cases systems of simultaneous equations will be obtained which might not be present if the ordering of the equations in one branch of the if-construct is changed. In any case, the model remains valid. Only the efficiency might be unnecessarily reduced.

Conditional Models

It is useful to be able to have models of different complexities. For complex models, *conditional components* are needed as shown in the next example where the two controllers are modeled itself as subcomponents:

```
block Controller
  input  Boolean simple=true;
  input  Real e;
  output Real y;
protected
  Controller1 c1(u=e, enable=simple);
  Controller2 c2(u=e, enable=not simple);
equation
  y = if simple then c1.y else c2.y;
end Controller;
```

Attribute `enable` is built-in Boolean input of every block with default equation "enable=true". It allows enabling or disabling a component. The enable-condition may be time and state dependent. If enable=false for an instance, its equations are not evaluated, all declared variables are held constant and all subcomponents are disabled. Special consideration is needed when enabling a subcomponent. The reset attribute makes it possible to reset all variables to their Start-values before enabling. The reset attribute is propagated to all subcomponents. The previous controller example could then be generalized as follows, taking into account that the Boolean variable simple could vary during a simulation.

```
block Controller
  input  Boolean simple=true;
  input  Real e
  output Real y
protected
  Controller1 c1(u=e, enable=simple, reset=true);
  Controller2 c2(u=e, enable=not simple, reset=true);
equation
  y = if simple then c1.y else c2.y;
end Controller;
```

Discrete Event and Discrete Time Models

The actions to be performed at events are specified by a when-statement.

```
when condition then
  equations
end when;
```

The equations are active instantaneously when the condition *becomes* true. It is possible to use a vector of conditions. In such a case the equations are active whenever *any* of the conditions becomes true.

Special actions can be performed when the simulation starts and when it finishes by testing the built-in predicates **initial**() and **terminal**( ). A special operator **reinit**(state, value) can be used to assign new values to the continuous states of a model at an event.

Let's consider discrete time systems or sampled data systems. They are characterized by the ability to periodically sample continuous input variables, calculate new outputs influencing the continuous parts of the model and update discrete state variables. The output variables keep their values between the samplings. We need to be able to activate equations once every sampling. There is a built-in function **sample**(Start, Interval) that is true when **time**=Start + n*Interval, n>=0. A discrete first order state space model can then be written as

```
block DiscreteStateSpace
  parameter Real a, b, c, d;
  parameter Real Period=1;
  input  Real u;
  discrete output Real y;
protected
  discrete Real x;

equation
  when sample(0, Period) then
    x = a*pre(x) + b*u;
    y = c*pre(x) + d*u;
  end when;
end DiscreteStateSpace;
```

The special notation **pre**(x) is used to denote the left limit of the **discrete** state variable x at an event instant, whereas variable x always denotes the right limit.

In this case, the first sampling is performed when simulation starts. With Start > 0, there would not have been any equation defining x and y initially. All variables being defined by when-statements hold their values between the activation of the equations and have the value of their

start-attribute before the first sampling, i.e., they are discrete state variables and may optionally have the variable prefix **discrete**. Boolean, Integer, and String variables are always discrete-time variables, i.e., these variables change their value only at an event instant.

For non-periodic sampling a somewhat more complex method for specifying the samplings can be used. The sequence of sampling instants can be calculated by the model itself and kept in a discrete state variable, say NextSampling. We would then like to activate a set of equations once *when* the condition **time**>= NextSampling *becomes* true. An alternative formulation of the above discrete system is thus.

```
block DiscreteStateSpace2
  parameter Real a, b, c, d;
  parameter Real Period=1;
  input  Real u;
  discrete output Real y;
protected
  discrete Real x, NextSampling(start=0);

equation
  when time >= pre(NextSampling) then
    x = a*pre(x) + b*u;
    y = c*pre(x) + d*u;
    NextSampling = time + Period;
  end when;
end DiscreteStateSpace2;
```

The built-in operator **edge**(v), for discrete-time variable v, is defined as "v **and not pre**(v)", i.e., it is **true** at the time instant when v changes its value and otherwise it is false. With this operator the precise meaning of a when clause

```
when condition then
   v2 = f1(..);
   v3 = f2(..);
end when
```

can be defined as:

```
   Boolean b(start = <condition using start values>);
equation
   b  = condition;
   v2 = if edge(b) then f1(..) else pre(v2);
   v3 = if edge(b) then f2(..) else pre(v3);
```

In other words, at the time instant when b changes its value from **false** to **true**, the two equations are activated. At all other time instants, v2 and v3 hold their previous value. The **pre**-value of b at the initial time is determined by evaluating the condition with the start values of all variables appearing in this condition. If this evaluates to **true**, the when-clause equations are not activated at the initial time. As sketched in this example, **when**-clause equations are mapped to equations which can be sorted together with all other discrete and continuous equations. In order that this is possible, there is the restriction that equations in when-clauses do not have the general form "expr1 = expr2", but the restricted form "v1 = expr", i.e., a single variable or array at the left hand side of the equality sign. The reason for this restriction becomes apparent in the following example:

```
   Real x, y;
equation
   x + y = 5;
   when condition then
      2*x + y = 7;         // error: not valid Modelica
   end when;
```

When the equations of the when-clause are not activated it is not clear which variable to hold constant, either x or y. A corrected version of this example is:

```
   Real x,y;
equation
   x + y = 5;
   when condition then
      y = 7 - 2*x;        // fine
   end when;
```

Here, variable y is held constant when the when-clause is de-activated and x is computed from the first equation using the value of y from the previous event instant.

When-clauses in equation sections can have only one branch. However, in algorithm sections, **elsewhen** branches are possible. This is useful, in order to define priorities between discrete actions, such as:

```
   Boolean open;
algorithm
   when h1 < hmax then
      open := true;
   elsewhen pushbutton then
      open := false;
   end when;
```

Here, the condition "h1 < hmax" has higher priority as the condition "pushbutton", if both conditions become true at the same event instant. Similarly as for when-clauses in equation sections, the precise meaning of this when-clause in an algorithm can be expressed as:

```
   Boolean open(start = false);
   Boolean b1  (start = h1.start < hmax);
   Boolean b2  (start = pushbutton.start);
algorithm
   open := pre(open);
   b1   := h1 < hmax;
   b2   := pusbutton;
   if edge(b1) then
      open := true;
   elseif edge(b2) then
      open := false;
   end when;
```

Note, that this is a conceptual mapping and a Modelica translator may perform it more efficiently. In general, all discrete-time variables which are potentially assigned in an algorithm section (such as variable "open" above) are initialized with their "pre"-value when the algorithm section is entered, whereas all continuous-time variables which are potentially assigned in an algorithm section are initialized with their "start"-value.

The condition of a when-clause may be a vector expression. In this case the when-clause is activated whenever one of the elements of the vector condition becomes true. Example:

```
model vectorwhen
   parameter Real A=1.5, w=6;
   Real    u1, u2;
   Boolean b1, b2;
equation
   u1 = A*Modelica.Math.sin(w*time);
   u2 = A*Modelica.Math.cos(w*time);
   when u1 > 0 or u2 > 0 then
      b1 = not pre(b1);
   end when;
   when {u1 > 0, u2 > 0} then   // vector condition
      b2 = not pre(b2);
   end when;
end vectorwhen;
```

The two when clauses are not equivalent as can be seen when applying the discussed mapping rule:

```
b1 = if edge(u1 > 0  or u2 > 0)      then not pre(b1) else pre(b1);
b2 = if edge(u1 > 0) or edge(u2 > 0) then not pre(b2) else pre(b2);
```

If the conditions used in if-the-else expressions contain relations with dynamic variables, the corresponding derivative function f might not be continuous and have as many continuous partial derivatives as required by the integration routine in order for efficient simulation. Every change of such a relation triggers an event in Modelica in order to efficiently and reliably handle such a discontinuity. Modern integrators have indicator functions for such discontinuous events. For a relation like v1 > v2, a proper indicator function is v1 - v2.

If the resulting if-then-else expression is smooth, the modeller has the possibility to give this extra information to the integrator in order to avoid event handling and thus enhance efficiency. This can be done by embedding the corresponding relation in a function **noEvent** as follows.

```
y = if      noEvent(u > HighLimit) then HighLimit
    else if noEvent(u < LowLimit)  then LowLimit else u;
```

The **noEvent**() operator can only be applied in Real equations, but not in Boolean, Integer or String equations, in order that Boolean, Integer and String variables can change their value only at event instants, i.e., they are always discrete-time variables. (for the exact formulation of this restriction, see section 3.4.8 in the Modelica Language Specification).

Synchronization and event propagation

Propagation of events can be done by the use of Boolean variables. A Boolean equation like

```
Out.Overflowing = Height > MaxLevel;
```

in a level sensor might define a Boolean variable, Overflowing, in an interface. Other components, like a pump controller might react on this by testing Overflowing in their corresponding interfaces

```
Pumping = In.Overflowing or StartPumping;
DeltaPressure = if Pumping then DP else 0;
```

A connection like

```
connect(LevelSensor.Out, PumpController.In);
```

would generate an equation for the Boolean component StartPump

```
LevelSensor.Out.StartPump = PumpController.In.StartPump;
```

For simulation, this equations needs to be solved for PumpController.In.StartPump. Boolean equations always needs to have a variable in either the left hand part or the right hand part or in both in order to be solvable.

An event (a relation becoming true or false) might involve the change of continuous variables. Such continuous variables might be used in some other relation, etc. Propagation of events thus might require evaluation of both continuous equations and conditional equations.

Ideal switching devices

Consider the rectifier circuit of Figure 3. We will show an appropriate way of modeling an ideal diode.

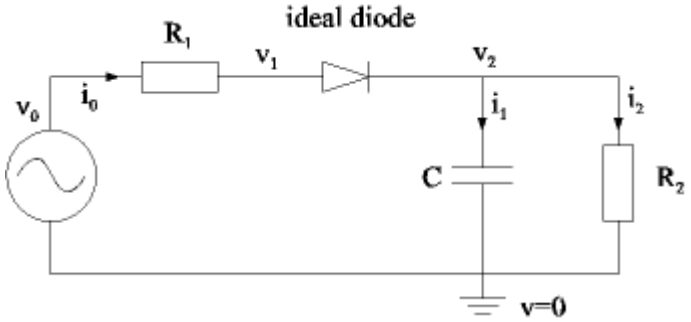


Figure 3. Rectifier circuit

The characteristics of the ideal diode is shown in Figure 4.

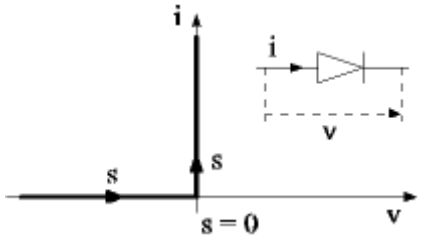


Figure 4. Characteristics of ideal diode

It is not possible to write i as a function of v or vice versa because the ideal characteristics. However, for such planar curves a parametric form can be used

x = f(s)
y = g(s)

where s is a scalar curve parameter. The ideal diode can then be described as

```
    i = if s < 0 then s else 0;
    v = if s < 0 then 0 else s;
```

The complete model of the ideal diode is then

```
model IdealDiode "Ideal electrical diode"
  extends OnePort;
protected
  Real s;
equation
   i = if s < 0 then s else 0;
   v = if s < 0 then 0 else s;
end IdealDiode;
```

This technique is also appropriate to model ideal thyristors, hysteresis and ideal friction.

Conditional Equations with Causality Changes

The following example models a breaking pendulum - a simple variable structure model. The number of degrees-of-freedom increases from one to two when the pendulum breaks. The example shows the needs to transfer information from one set of state variables (phi, phid) to another (pos, vel) at an event. Consider the following description with a *parameter* Broken.

```
model BreakingPendulum
  parameter Real m=1, g=9.81, L=0.5;

  parameter Boolean Broken;
  input Real    u;
  Real pos[2], vel[2];
  constant Real PI=3.141592653589793;
  Real phi(start=PI/4), phid;

equation
   vel = der(pos);

   if not Broken then
    // Equations of pendulum
    pos  = {L*sin(phi), -L*cos(phi)};
    phid = der(phi);
    m*L*L*der(phid) + m*g*L*sin(phi) = u;

  else;
    // Equations of free flying mass
    m*der(vel) = m*{0, -g};
  end if;
end BreakingPendulum;
```

This problem is non-trivial to simulate if Broken would be a dynamic variable because the defining equations of the absolute position "pos" and of the absolute velocity "vel" of the mass change causality when the pendulum breaks. When "Broken=false", the position and the velocity are calculated from the Pendulum angle "phi" and Pendulum angular velocity "phid". After the Pendulum is broken, the position and velocity are state variables and therefore known quantities in the model.

As already mentioned, conditional equations with dynamic conditions are presently not supported because it is not yet clear in which way a translator can handle such a system automatically in an efficient way. It might be that a translator pragma is needed to guide the translation process. It is possible to simulate variable causality systems, such as the breaking pendulum, by reformulating the problem into a form where no causality change takes place using conditional block models:

```
record PendulumData
   parameter Real m, g, L;
end PendulumData;


partial model BasePendulum
   PendulumData p;
   input   Real u;
   output  Real pos[2], vel[2];
end BasePendulum;


block Pendulum
   extends BasePendulum;
   constant Real PI=3.141592653589793;
   output   Real phi(start=PI/4), phid;
equation
   phid = der(phi);
   p.m*p.L*p.L*der(phid) + p.m*p.g*p.L*sin(phi) = u;

   pos = {p.L*sin(phi), -p.L*cos(phi)};
   vel = der(pos);
end Pendulum;


block BrokenPendulum
   extends BasePendulum;
equation
   vel = der(pos);
   p.m*der(vel) = p.m*{0, -p.g};
end BrokenPendulum;


model BreakingPendulum2
   extends BasePendulum(p(m=1, g=9.81, L=0.5));
   input Boolean Broken;
protected
   Pendulum       pend (p=p, u=u, enable=not Broken);
   BrokenPendulum bpend(p=p, u=u, enable=Broken);
equation
   when Broken then
      reinit(bpend.pos, pend.pos);
      reinit(bpend.vel, pend.vel);
   end when;
   pos = if not Broken then pend.pos else bpend.pos;
   vel = if not Broken then pend.vel else bpend.vel;
end BreakingPendulum2;
```

This rewriting scheme is always possible and results in a larger model. It has the drawback that

the same physical variable is represented by several model variables. In some cases, such as for the breaking pendulum, it is possible to avoid this drawback:

```
model BreakingPendulum3
  parameter Real m=1, g=9.81;

  input Boolean Broken;
  input Real    u;
  Real pos[2], vel[2];
  constant Real PI=3.141592653589793;
  Real phi(start=PI/4), phid;
  Real L(start=0.5), Ldot;

equation
  pos = {L*sin(phi), -L*cos(phi)};
  vel = der(pos);
  phid = der(phi);
  Ldot = der(L);

  zeros(2) = if not Broken then {
    // Equations of pendulum
    m*der(phid) + m*g*L*sin(phi) – u,
    der(Ldot)}
  else
    // Equations of free flying mass
    m*der(vel) - m*{0, -g};
end BreakingPendulum3;
```

The trick was to use complete polar coordinates including the length, L and to give a differential equation for L in the non Broken mode. If the derivatives of some variables are not calculated during the "not Broken"-phase, the variables "pos" and "vel" can be considered as algebraic variables. A simulator thus has the possibility to remove them from the set of active state variables.

2.9 Physical Fields

Modeling of physical fields, such as an environment with fixed temperature and pressure, or an electrical or a gravity field, is possible with the already introduced language elements. However, for bigger systems modeling becomes tedious and inconvenient, because lumped physical field models lead to *1* to *n* or *n* to *n* connections between all *n* components influenced by a field. For *1* to *n* connections, Modelica offers a convenient modeling mechanism with the **inner** and **outer** language elements. Basically, these two elements are used in the following way:

```
model Component
  outer Real T0;     // temperature T0 defined outside of Component
  Real T;
equation
  T = T0;
end Component;

model Environment
  inner Real T0;          // actual environment temperature T0
  Component c1, c2;       // c1.T0=c2.T0=T0
  parameter Real a=1;
```

```
equation
  T0 = Modelica.Math.sin(a*time);
end Environment;

model SeveralEnvironments
  Environment e1(a=1), e2(a=2)
end SeveralEnvironments
```

If a variable or a component is declared as **outer**, as in model Component, the actual instance is defined outside of the defining class and is determined by searching the *object hierarchy* upwards until a corresponding declaration with the **inner** prefix is found. In the example this declaration is found in model Environment. Therefore, all declarations "outer Real T0" of all objects and subobjects within Environment are just references to the only "real" instance declared as "inner Real T0". This feature allows to have several environments in parallel, as is shown in model SeveralEnvironments.

The **inner** and **outer** prefixes can be applied to every type of component. It is especially useful for **connectors**, in order to define implicitly physical connections between all objects and their environment. Assume for example, that the heat flow of all components of an electrical circuit board to the fixed-temperature environment shall be modeled. This requires to first introduce a connector for 1-dimensional heat flow:

```
connector HeatCut
   Modelica.SIunits.Temp_K            T "temperature in [K]";
   flow Modelica.SIunits.HeatFlowRate q "heat flux";
end HeatCut;
```

All components which generate heat need a reference to the environment heat connector:

```
model Component
  outer HeatCut environment;  // reference to environment
  HeatCut heat;               // heat connector of component
    ...
equation
  connect(heat, environment);
    ...
end Component;
```

Note, that "**outer** HeatCut environment" is a reference to the declaration of connector environment defined outside of this component and that the heat connector of this Component is connected to the environment connector.

```
model TwoComponents
  Component Comp[2];
end TwoComponents;

model CircuitBoard
  inner HeatCut environment;
  Component     comp1;
  TwoComponents comp2;
end CircuitBoard;
```

The components can be used in several levels until the environment of the circuit board is reached where the "inner" declaration of the heat connector is present. All instances of model Component which are used inside model CircuitBoard automatically connect their heat flow connector to the environment connector, i.e., there is a 1 to n connection of the "inner HeatCut

environment" to the heat flow connectors of all components.

In some cases the exact nature of the field is unknown, when the components are defined. In such a case inner/outer functions can be utilized as demonstrated by the model of a particle moving in an a-priori unknown gravity field:

A generic gravity field shall be defined by the partial function gravityInterface

```
partial function gravityInterface
  input  Real r[3] "position";
  output Real g[3] ”gravity acceleration";
end gravityInterface;
```

where only the interface of the function is defined. Since it is a partial function, this function cannot be called and can only be used as superclass for other functions, such as:

```
function uniformGravity
  extends gravityInterface;
algorithm
  g := {0, -9.81, 0};
end uniformGravity;

function pointGravity
   extends gravityInterface;
   parameter Real k=1;
protected
   Real n[3]
algorithm
   n := -r/sqrt(r*r);
   g := k/(r*r) * n;
end pointGravity;
```

The idea is to utilize the partial function "gravityInterface" when defining the particle model which shall move in a gravity field

```
model Particle
   parameter Real m = 1;
   outer function gravity = gravityInterface;
   Real r[3](start = {1,1,0}) "position";
   Real v[3](start = {0,1,0}) "velocity";
equation
     der(r) = v;
   m*der(v) = m*gravity(r);
end Particle;
```

and to define the function which is actually used to compute the gravity acceleration at an outer level using an **inner** function definition:

```
model Composite1
   inner function gravity = pointGravity(k=1);
   Particle p1, p2(r(start={1,0,0}));
end Composite1;

model Composite2
   inner function gravity = uniformGravity;
   Particle p1, p2(v(start={0,0.9,0}));
end Composite2;
```

```
model system
   Composite1 c1;
   Composite2 c2;
end system;
```

As can be seen by this example, different fields of this nature can be handled at the same time.

2.10 Library Construction

Modelica has a sophisticated scheme to handle large model libraries in a convenient and practical way. Basically, component models are stored in hierarchically structured packages. Example:

```
package Modelica
   package Mechanics
      package Rotational
         model Inertia    // Modelica.Mechanics.Rotational.Inertia
           ...
         end Inertia;
         model Torque
            ...
         end Torque;
      end Rotational;
   end Mechanics;
end Modelica;
```

From the outside of package Modelica, components can be accessed via dot-notation, e.g., model Inertia is uniquely identified as Modelica.Mechanics.Rotational.Inertia.

Name look-up in hierarchically structured classes

Within a hierarchical package (or in general in any class, such as a model or a block), the first part of a name is searched recursively in upper hierarchies until this name is found. The rest of the name is located in the hierarchy below this name. For example:

```
package Modelica
  package Blocks
     package Interfaces
        connector InPort
           ...
        end InPort;
     end Interfaces;
  end Blocks;

  package Mechanics
    package Rotational
      package Interfaces
        connector Flange_a
           ...
        end Flange_a;
      end Interfaces

      model Inertia
        Interfaces.Flange_a a1;  // Modelica.Blocks.Interfaces.Flange_a
        Modelica.Mechanics.Rotational.Interfaces.Flange_a a2;
      end Inertia;
```

```
      model Torque
         Interfaces.Flange_a      a;
         Blocks.Interfaces.InPort inPort;  // Modelica.Blocks...
          ...
      end Torque;
      ...
    end Rotational;
  end Mechanics;
end Modelica;
```

Both definitions of a1 and a2 refer to the same connector, namely Modelica.Mechanics.Rotational.Interfaces.Flange_a. In all cases, the first part of the names (here: "Interfaces", "Modelica" and "Blocks") are searched in upper hierarchies until they are found.

Encapsulated classes and import statement

Copying or moving package Modelica.Mechanics.Rotational from the example above to another location may not work as expected, because elements of this package may access packages outside of package Rotational, such as "*Blocks.Interfaces.InPort* inPort": In the new location a package "Blocks" is most likely not available and therefore references to this package cannot be resolved, leading to an error. To improve this situation, in Modelica 1.4 "*self-contained classes*" have been introduced which are defined with the class-prefix **encapsulated**. The recommended way to structure libraries is shown in the following example, taken from the Modelica Standard Library:

```
encapsulated package Modelica
  encapsulated package Blocks
    package Interfaces
      connector InPort
        ...
      end InPort;
    end Interfaces;

    package Continuous
      model Integrator
        Interfaces.InPort inPort;
          ...
      end Integrator;
        ...
    end Continuous

    package Examples
       encapsulated model Example1
          import Modelica.Blocks;
          Blocks.Integrator int1;      // Modelica.Blocks.Integrator
          Modelica.Blocks.Integrator int2;  // error, Modelica unknown
             ...
       end Example1;
         ...
    end Examples;
  end Blocks;

  encapsulated package Mechanics
```

```
    encapsulated package Rotational
      import Modelica.Blocks.Interfaces;
      model Torque
         Interfaces.InPort inPort; // Modelica.Blocks.Interfaces.InPort
          ...
      end Torque;
      ...
    end Rotational;
  end Mechanics;
end Modelica;
```

The **encapsulated** prefix stops name look-up from lower to upper hierarchies. As a result, within such a "self-contained unit" it is no longer possible to access classes outside of this unit in an uncontrolled way. Instead, such classes have to be made explicitly available with the **import** statement. Consequently, when moving or copying an **encapsulated** class, at most the **import** statements in this class have to be changed, and nothing else. If an **encapsulated** class does not contain another **encapsulated** class, not even import statements have to be changed in such a situation.

In the example above, the typical usage of the **encapsulated** prefix is demonstrated. Package Modelica.Blocks is constructed as a "self-contained unit". Since this unit is already quite large, it is further hierarchically structured with subpackages, such as "Interfaces", "Continuous", "Examples". These subpackages are not defined as **encapsulated**, because (1) there are many dependencies between these subpackages and it is more convenient to access the different parts of package Blocks directly and because (2) it is not very useful to cut-off parts of this package and use it somewhere else, i.e., the library developer does not see, e.g., "Modelica.Blocks.Continuous" as a self-contained unit. On the other hand, the models within the Modelica:Blocks.Examples subpackage are declared as **encapsulated**, because a modeller may wish to copy such an example model out of the library and use or modify it.

The **import** statement can be used in three different variants:

```
import Modelica.Mechanics.Rotational;    // access by Rotational.Torque
import R = Modelica.Mechanics.Rotational;// access by R.Torque
import Modelica.Mechanics.Rotational.*;  // access by Torque
```

In all cases, the first part of the name defined in the import statement is located within the top-level classes and no hierarchical search in upper hierarchical levels is performed. Note, that import statements are not inherited in order to not introduce hidden dependencies. The third, "unqualified", form for import statements should be avoided, because, e.g., the later addition of a model "Gearbox" to package Modelica.Mechanics.Rotational may give rise to a name conflict, if the same name is already used in the user model where this import statement is present. Therefore, by just getting a new version of the Modelica Standard Library, existing user models may no longer work. This cannot happen with the first two variants. The "unqualified" variant is useful, to arrive at the standard notation for some basic mathematical constants and functions, such as:

```
model SineVoltageSource
   import Modelica.Constants.*;   // to access Modelica.Constants.pi
   import Modelica.Math.*;        // to access Modelica.Math.sin
   extends Modelica.Electrical.Analog.Interfaces.OnePort;
   parameter Real A=220 "amplitude";
   parameter Real f=50  "frequency";
```

```
equation
   v = A*sin(2*pi*f*time);
end SineVoltageSource;
```

Mapping of class names to names in the file system

Classes are usually stored in the file system (or in databases etc.). In order that a Modelica environment can locate a desired class uniquely in the file system without any additional information, it is precisely defined how Modelica classes have to be stored. Therefore, when a referenced class, such as Modelica.Mechanics.Rotational.Inertia, is not yet available in the "workspace" of the simulation environment, the tool can locate this class in the file system and can load it automatically. In order that this is possible, Modelica classes have to be stored in one of the following ways:

- The complete class is stored in one file, where the file name is the class name with extension ".mo". Examples:

```
file   : Modelica.mo
content: encapsulated package Modelica
            encapsulated package Blocks
              ...
            end Blocks;
          ...
          end Modelica;

file   : robot.mo
content: model robot
           ...
          end robot;
```

- The class is stored in directories and files, such that
  (1) a directory name corresponds to the name of the class stored in this directory,
  (2) every directory has a file package.mo in which at least the exact class declaration is present, such as "package Modelica end Modelica;",
  (3) file names correspond to the class name stored in the file together with the extension ".mo" and
  (4) the first statement in a file or in package.mo is a "within name" statement which defines the full name of the class in the outer hierarchy. Examples:

```
directory: .../library
                /Modelica
                    package.mo
                    /Blocks
                        package.mo
                        Continuous.mo
                        Interfaces.mo
                        /Examples
                            package.mo
                            Example1.mo
                    /Mechanics
                        package.mo
                        Rotational.mo

file   : .../library/Modelica/Blocks/Examples/package.mo:
```

```
content: within Modelica.Blocks;
         package Examples "examples of package Modelica.Blocks";
         end Examples;

file   : .../library/Modelica/Mechanics/Rotational.mo
content: within Modelica.Mechanics;
         encapsulated package Rotational //Modelica.Mechanics.Rotational
           package Interfaces
             connector Flange_a;
                ...
             end Flange_a;
               ...
           end Interfaces;
           model Inertia
              ...
           end Inertia;
         end Rotational;
```

The top-level classes, such as "Modelica", are located in all directories defined in the environment variable *MODELICAPATH*, which contains a semicolon-separated list of directory names. For example, the first part of the class name A.B.C (i.e., A) is located by searching the ordered list of directories in MODELICAPATH. If no directory contains A the lookup fails. If A has been found in one of the directories, the rest of the name is located in A; if that fails, the entire lookup fails without searching for A in any of the remaining directories in MODELICAPATH.

2.11 Units and Quantities

The built-in "Real" type of Modelica has additional attributes to define *unit* properties of variables:

```
type Real
  parameter StringType quantity    = "";
  parameter StringType unit        = "" "unit in equations";
  parameter StringType displayUnit = "" "default display unit";
  ...
end Real;

// define quantity types
type Force = Real(final quantity="Force", final unit="N");
type Angle = Real(final quantity="Angle", final unit="rad",
                        displayUnit="deg");

// use the quantity types
Force f1   , f2  (displayUnit="kp");
Angle alpha, beta(displayUnit="rad");
```

The *quantity* attribute defines the category of the variable, like Length, Mass, Pressure. The *unit* attribute defines the unit of a variable as utilized in the equations. That is, all equations in which the corresponding variable is used are only correct, provided the numeric value of the variable is given with respect to the defined unit. Finally, *displayUnit* gives the default unit to be used in tools based on Modelica for interactive input and output. If, for example, a parameter value is input via a menu, the user can select the desired unit from a list of units, using the "displayUnit" value as default. When generating Modelica code, the tool makes the conversion to the defined

"unit" and stores the used unit in the "displayUnit" field. Similarly, a simulator may convert simulation results from the "unit" into the "displayUnit" unit before storing the results on file. All of these actions are optional. If tools do not support units, or a specific unit cannot be found in the unit database, the value of the "unit" attribute could be displayed in menus, plots etc.

The *quantity* attribute is used as grouping mechanism in an interactive environment: Based on the quantity name, a list of units is displayed which can be used as *displayUnit* for the underlying physical quantity. The quantity name is needed because it is in general not possible to determine just by the *unit* whether two different units belong to the same physical quantity. For example,

```
type Torque = Real(final quantity="MomentOfForce", final unit="N.m");
type Energy = Real(final quantity="Energy"       , final unit="J");
```

the units of type Torque and type Energy can be both transformed to the same *base units*, namely "kg.m2/s2". Still, the two types characterize different physical quantities and when displaying the possible displayUnits for torque types, unit "J" should not be in such a list. If only a unit name is given and no quantity name, it is not possible to get a list of displayUnits in a simulation environment.

Together with Modelica the *standard package Modelica.SIunits* of *predefined* quantity and connector types is provided in the form as shown in the example above. This package is based on the corresponding ISO norm. This will give some help in standardization of the interfaces of models. The grammar for unit expressions, such as "N.m/s2" is defined in the Modelica Language Specification and follows an ISO recommendation. Note, that the prefix **final** defines that the quantity and unit values of the predefined types cannot be modified.

Conversion between units is **not** supported within the Modelica language. This simplifies a Modelica translator considerably, especially because a unit-database with its always incomplete collection of units is not needed, see e.g. (Cardarelli 1997). As a consequence, the semantics of a correct Modelica model is independent of the unit attributes and the Modelica translator can ignore them during code generation. Especially, the unit attributes need *not* be checked for a connection, i.e., connected variables may have different quantities and units.

Much more support on units and quantities will be given by tools based on Modelica. This will be considered as "quality of implementation". An object-diagram editor may, for example, support automatic unit conversion when two interfaces are connected. As a general rule it will always be allowed to connect any variable to a variable which has no quantity and unit associated with it. Furthermore, a Modelica translator may optionally check equations on correct dimensionality (this will produce only warning messages, i.e., code will be produced anyway). The equation "f=m*a" would, for example, produce a warning, if "f" is given in "N.m" because then the units are not compatible to each other. The variables in the equations may have non-SI units. Therefore, for example, the compiler will not detect that "f=m*a" is an error, if the units "N" for "f", "g" for "m" and "m/s2" for "a" are used. Dimension checking is done by transforming the "quantity" information into one of the seven base "quantities" (like "Mass", "Length").

Usually, units are associated with types. There are however elements where instances may have a different unit by redefinition of the quantity type. Example:

```
type Voltage = Real(final quantity="Voltage", final unit="V");
```

```
model SineSignal
  import Modelica.Constants.*;
  import Modelica.Math.*;
  import SI=Modelica.SIunits;
  parameter Real freq (unit="Hz");
  parameter SI.Angle phi;

  replaceable type SineType = Real;
  parameter SineType Amplitude;
  output    SineType y;
equation
  y = Amplitude*sin(2*pi*freq*time + phi);
end SineSignal;

model Circuit
  import SI=Modelica.SIunits;
  SineSignal    sig(redeclare SineType = SI.Voltage);
  VoltageSource Vsource;
    ...
equation
  connect(sig.y, Vsource.in);
end Circuit;
```

In a block diagram library there is a general sine signal generator. When it is used to generate a voltage sine for a voltage source, the output of the signal generator should have a unit of "V". This can be accomplished by having the type of the amplitude and of the output as a replaceable type which can be changed appropriately when this signal generator is instantiated.

2.12 Annotations for Graphics and Documentation

In addition to the mathematical model with variables and equations, additional information is needed for example to represent icons, graphical layout, connections and extended documentation. Graphically representing models as interconnected submodels displayed as icons, supports their quick understanding. As most contemporary tools provide facilities to build models graphically, Modelica has language constructs to represent icons, graphical layout and the connections between submodels.

Modelica supports property lists for the various components. Such lists can be used to store graphical, documentation and tool related annotations. Each component can have a list designated by the keyword **annotation**. The value of such annotations can be according to any class, i.e., it can be created using a class modification. The strong type checking is abandoned in this case because of the need for various modeling tools to use different kinds of annotations. Since such annotation values are normally generated and read by tools, i.e., not directly edited by humans, there is a reduced need for having redundant type information. However, in order that graphical and documentation information can be exchanged between tools, a minimum set of annotation components are specified.

Graphical representation of models

Graphical annotation information is given in three separate contexts:

- Annotations associated with a component, typically to specify position and size of the component.
- Annotations of a class to specify the graphical representation of its icon (see above), diagram, and common properties such as the local coordinate system.
- Annotations associated with connections, i.e., route, color of connection line, etc.

The example below shows the use of such graphical attributes to define a resistor.

```
model Resistor
  Pin p annotation (extent=[-110, -10; -90, 10]);
  Pin n annotation (extent=[ 110, -10;  90, 10]);

  parameter R "Resistance in [Ohm]";

equation
  R*p.i = p.v - n.v;
  n.i = p.i;

public
  annotation (Icon(
    Rectangle(extent=[-70, -30; 70, 30], style(fillPattern=1)),
    Text(extent=[-100, 55; 100, 110], string="%name=%R"),
    Line(points=[-90, 0; -70, 0]),
    Line(points=[70, 0; 90, 0])
    ));
end Resistor;
```

The resistor has two pins, and we specify two opposite corners of the extent of their graphical representation. An icon of the Resistor is defined by a rectangle, a text string and two lines. For the rectangle we specify additional style attributes for fill pattern.

The extent specified for a component is used to scale the icon image. The icon is drawn in the master coordinate system specified in the component's class. The icon is scaled and translated so the coordinate system is mapped to the region defined in the component declaration.

The attribute set to represent component positions, connections and various graphical primitives for building icons is shown below. The attribute structures are described through Modelica classes. Points and extents (two opposite points) are described in matrix notation.

```
type Point = Real[2];      // {x, y}

type Extent = Real[2,2];   // [x1, y1; x2, y2]

record CoordinateSystem    // Attribute to class
  Extent extent;
  Point grid;
  Point size;
end CoordinateSystem;

record Placement           // Attribute for component
  Extent extent;
  Real rotation;
end Placement;
```

```
record Style
  Integer color[3], fillColor[3];    // RGB
  Integer pattern, fillPattern, thickness, gradient,
      smooth, arrow, textStyle;
  String font;
end Style;

record Route                // Attribute for connect
  Point points[:];
  Style style;
  String label;
end Route;

// Definitions for graphical elements
record Line = Route;

record Polygon = Route;

record GraphicItem
  Extent extent;
  Style style;
end GraphicItem;

record Rectangle = GraphicItem;

record Ellipse = GraphicItem;

record Text
  extends GraphicItem;
  String string;
end Text;

record BitMap
  extends GraphicItem;
  String URL;              // Name of bitmap file
end BitMap;
```

The graphical unit of the master coordinate system used when drawing lines, rectangles, text etc. is the baseline spacing of the default font used by the graphical tool, typically 12 points for a 10 point font (note: baseline spacing = space between text lines).

Definition of menu lists

If replaceable models are used, it is convenient in a graphical user environment to present the user a list of alternative models which can be used for redeclaration. This can be accomplished with the predefined annotation "*choices*":

```
replaceable model MyResistor=Resistor
  annotation(choices(
     choice(redeclare MyResistor=lib2.Resistor(a={2}) "Resistor 1"),
     choice(redeclare MyResistor=lib2.Resistor2 "Resistor 2")));

replaceable Resistor Load(R=2) extends OnePort
  annotation(choices(
     choice(redeclare lib2.Resistor Load(a={2}) "Resistor"),
```

```
      choice(redeclare Capacitor Load(L=3) "Capacitor")));

  replaceable FrictionFunction a(func=exp) extends Friction
    annotation(choices(
      choice(redeclare ConstantFriction a(c=1) "Constant Friction"),
      choice(redeclare TableFriction a(table="…")"Table-Friction"),
      choice(redeclare FunctionFriction a(func=exp) "Exp-Friction"))));
```

The "choices" annotation contains modifiers on choice, where each of them indicates a suitable redeclaration or modifications of the element. The string comments on the choice declaration can be used as textual explanations of the choices. A graphical user environment could display the string comments in a list and after selection of one of the alternatives by the user, the redeclare statement of the corresponding choice is executed.

This annotation is not restricted to replaceable elements but can also be applied to non-replaceable elements, enumerated types, and simple variables. Example:

```
type controllerType=Integer(min=1,max=3)
   annotation(choices(
               choice=1 "P",
               choice=2 "PI",
               choice=3 "PID"));

model test
  parameter controllerType c;
    ...
end test;

test t;
```

When displaying the parameter menu for model t, a Modelica environment could present not a value field for parameter c, but a choice selecting menu with the entries "P", "PI" and "PID" as choices. After the selection, the graphical environment transforms the choice into the Integer values 1, 2 or 3.Documentation of models

In practical modeling studies, documenting the model is an important issue. It is not only for writing a report on the modeling work, but also to record additional information which can be consulted when the model is reused. This information need not necessarily be completely structured and standardized in the sense that Modelica language constructs are available for all aspects. The following aspects should typically be recognized:

History information

> Major milestones, like creation, important changes, release into public accessibility should be recorded. Information to store are the author, date and a brief description. This functionality is comparable with version control of software, using tools such as SCCS or RCS. If a specific modeling procedure is used, the mile stones of such a procedure can be recorded in this part.

References to literature

> References to external documents and/or scientific literature for understanding the model, its context and/or underlying theory should be mentioned here. The format can be like a

literature reference list in an scientific article.

Validation information

This concerns the reference (model or measurement data) to which the model is validated and criteria for validation. Also the simulation experiments used for the validation should be mentioned.

Explanation and sketches

A brief text describing the model or device, a kind of 'manual page' of the model. Schematic drawings or sketches can be incorporated for better understanding.

User advice

This extension of the explanation part, concerns additional remarks giving hints for reuse of the model.

Basic documentation functionality is available in Modelica. This consists of an annotation attribute **`Documentation`** which is further structured into key/text pairs.

```
annotation (Documentation(
  key1 = "Text string",
  key2 = "Text string"
  ));
```

Currently, no further detail on structuring information is given. The information is given as plain text in the appropriate category. It is likely that companies have their own way of documenting their models and experiments, so that different ways of filling in the documentation information are needed. In the Modelica Standard Library, the annotation "info" is used to define the documentation. For example, the documentation of this package is defined as:

```
annotation ( Documentation( info = "
<HTML>
<p>
Package <b>Modelica</b> is a <b>standardized</b> and <b>pre-defined</b>
package that is developed together with the Modelica language from the
Modelica Association, see
<a href=\"http://www.Modelica.org\">http://www.Modelica.org</a>.
It is also called <b>Modelica Standard Library</b>.
It provides constants, types, connectors, partial models and model
components in various disciplines.</p>
   ...
<HTML>" ) );
```

Within the documentation HTML-tags may be present. When extracting the documentation from a model, a tool shall store this part of the documentation as HTML-file which can be displayed by appropriate browsers such as Netscape or Microsoft Explorer in a nice way.

3. Examples

Modelica has been used to model various kinds of systems. Otter et.al., 1997 describes modeling of automatic gearboxes for the purpose of real-time simulation. Such models are non-trivial

because of the varying structure during gear shift utilizing clutches, free wheels and brakes. Mattsson, 1997 discusses modeling of heat exchangers. Class parameters of Modelica are used for medium parameterization and regular component structures are used for discretization in space of the heat exchanger. Tummescheit et.al., 1997 discusses thermodynamical and flow oriented models. Broenink, 1997 describes a Modelica library with bond graph models for supporting the bond graph modeling methodology. Franke, 1998 models a central solar heating plant using Modelica. Mosterman et.al., 1998 describes a Petri-Net library written in Modelica. In October 2000, the first workshop on Modelica with 85 participants took place in Lund, Sweden. All papers and posters of this workshop are available in pdf-format from http://www.Modelica.org/modelica2000/proceedings.

4. Conclusions

An overview of the most important language constructs as well as a series of useful modeling examples has been given for Modelica, version 1.4, which was released in December 15, 2000. Since the first release of Modelica 1.0 in September 1997, all necessary ingredients for modeling and simulation of industrial applications with Modelica have been developed and put together: The language, libraries, tools and successful applications. More detailed information and the most actual status can be found at

http://www.Modelica.org/

Therefore, it can be expected that Modelica will have a significant impact in the modeling and simulation community in the future. The Modelica Association is very interested to further adapt the Modelica language to the needs of the end-users and to further develop the Modelica libraries. Interested simulation professionals who would like to contribute are encouraged and invited to participate at the design meetings (see the Modelica homepage for details where and when design meetings take place).

5. References

Abadi M., and **L. Cardelli**: *A Theory of Objects*. Springer Verlag, ISBN 0-387-94775-2, 1996.

Broenink J.F.: "Bond-Graph Modeling in Modelica". *ESS'97 - European Simulation Symposium*, Oct., 1997.

Cardarelli F.: *Scientific Unit Conversion*. Springer Verlag, 1997.

Cardelli L.: "Types for Data-Oriented Languages (Overview)", in J. W. Schmidt, S. Ceri and M. Missikof (Eds.): *Advances in Database Technology* - EDBT'88, Lecture Notes in Computer Science n. 303, Springer-Verlag, 1988.

Cardelli L.: "Typeful Programming", in E. J. Neuhold and M. Paul (Eds.): *Formal description of Programming Concepts*, Springer-Verlag, 1991. Also published as SRC Research Report 45, Digital Equipment Corporation.

Elmqvist H., F.E. Cellier, and **M. Otter**: ‘‘Object-oriented modeling of hybrid systems.” In

Proceedings of European Simulation Symposium, ESS'93. The Society of Computer Simulation, October 1993.

Franke R.: "Modeling and Optimal Design of a Central Solar Heating Plant with Heat Storage in the Ground Using Modelica", Eurosim '98 Simulation Congress, Helsinki, Finland, April 14-15, 1998.

Kågedal D.: "A Natural Semantics specification for the equation-based modeling language Modelica," LiTH-IDA-Ex-98/48, Linköping University, Sweden, 1998.

Mattsson S.E.: "On Modelling of Heat Exchangers in Modelica". *ESS'97 - European Simulation Symposium*, Oct., 1997.

Mosterman P. J., M. Otter, H. Elmqvist: "Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica", Summer Computer Simulation Conference -98 , Reno, Nevada, USA, July 19-22, 1998.

Otter M., C. Schlegel, and **H. Elmqvist**: "Modeling and Realtime Simulation of an Automatic Gearbox using Modelica". *ESS'97 - European Simulation Symposium*, Oct., 1997.

Tummescheit H., T. Ernst and **M. Klose**: "Modelica and Smile - A Case Study Applying Object-Oriented Concepts to Multi-facet Modeling". *ESS'97 - European Simulation Symposium*, Oct., 1997.

Tummescheit H., and **R. Pitz-Paal**: "Simulation of a solar thermal central receiver power plant". *Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Vol. 6, Berlin, Germany, pp. 671-676, 1997.

6. Revision history

This section describes the history of the Modelica Language Design, and its contributors. The current version of this document is available from http://www.modelica.org/.

This document was written together with the Modelica Language Specification 1 as one document, and in version 1.1 they were revised and separated into two documents.

For Modelica 1.4, the tutorial has been updated with the most important language elements introduced since version 1.1 and therefore reflects the current status of the Modelica language. This document was edited by M. Otter and H. Olsson.

Modelica 1.4

Modelica 1.4 was released December 15, 2000. The Modelica Association was formed in Feb. 5, 2000 and is now responsible for the design of the Modelica language.

Contributors to the Modelica Language, version 1.4

Bernhard Bachmann, University of Applied Sciences, Bielefeld, Germany
Peter Bunus, Linköping University, Linköping, Sweden

Dag Brück, Dynasim, Lund, Sweden
Hilding Elmqvist, Dynasim, Lund, Sweden
Vadim Engelson, Linköping University, Sweden
Jorge Ferreira, University of Aveiro, Portugal
Peter Fritzson, Linköping University, Linköping, Sweden
Pavel Grozman, Equa, Stockholm, Sweden
Johan Gunnarsson, MathCore, Linköping, Sweden
Mats Jirstrand, MathCore, Linköping, Sweden
Clemens Klein-Robbenhaar, Germany
Pontus Lidman, MathCore, Linköping, Sweden
Sven Erik Mattsson, Dynasim, Lund, Sweden
Hans Olsson, Dynasim, Lund, Sweden
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
Tommy Persson, Linköping University, Sweden
Levon Saldamli, Linköping University, Sweden
André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Michael Tiller, Ford Motor Company, Detroit, U.S.A.
Hubertus Tummescheit, Lund Institute of Technology, Sweden
Hans-Jürg Wiesmann, ABB Corporate Research Ltd., Baden, Switzerland

Contributors to the Modelica Standard Library

Peter Beater, University of Paderborn, Germany
Christoph Clauß, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Michael Tiller, Ford Motor Company, Detroit, U.S.A
Hubertus Tummescheit, Lund Institute of Technology, Sweden

Main Changes in Modelica 1.4

- Removed declare-before-use rule. This simplifies graphical user environments, because there exists no order of declarations when components are graphically composed together.

- Refined package concept by introducing encapsulated classes and import mechanism. Encapsulated classes can be seen as "self-contained units": When copying or moving an encapsulated class, at most the import statements in this class have to be changed and nothing else.

- Refined when-clause: The nondiscrete keyword is removed, equations in when-clauses must have a unique variable name on the left hand side variable and the exact mapping of when-clauses to equations is defined. As a result, when-clauses are now precisely defined without referring to a sorting algorithm and it is possible to handle algebraic loops between when-clauses with different conditions and between when-clauses and the continuous-time part of a model. The discrete keyword is now optional, simplifying the

library development because only one type of connector is needed and not several types which do contain or do not contain the discrete prefix on variables. Additionally, when-clauses in algorithm sections may have elsewhen clauses which simplifies the definition of priorities between when-clauses

- For replaceable declarations: allowed constraining clauses, and annotations listing suitable redeclarations. This allows a graphical user environment to automatically built-up meaningful choice menus.

- Functions can specify their derivative. This allows, e.g., the application of the Pantelides algorithm to reduce the index of a DAE also for external functions.

- New built-in operator "**rem**" (remainder) and the built-in operators **div**, **mod**, **ceil**, **floor**, **integer**, previously only allowed to be used in when-clauses can now be used everywhere, because state events are automatically generated when the result value of one of these operator changes discontinuously ().

- Quantity attribute also for base types Boolean, Integer, String (and not only for Real), in order to allow abstracted variables to refer to physical quantities (e.g. Boolean i(quantity="Current") is true if current is flowing and is false if no current is flowing).

- Several minor enhancements, such as usage of dot-notation in modifications (e.g.: `"A x(B.C=1,B.D=2)"` is the same as `"A x(B(C=1,D=2));")`.

- final keyword also allowed in declaration, to prevent modification. Example

```
model A
   Real x[:];
   final Integer n=size(x,1);
 end A;
```

- Internally restructured specification.

Modelica 1.4 is backwards compatible with Modelica 1.3, with the exception of (1) some exotic cases where different results are achieved with the removed "declare-before-use-rule" and the previous declaration order, (2) when-clauses in equations sections, which use the general form "expr1 = expr2" (now only "v=expr" is allowed + some special cases for functions), (3) some exotic cases where a when-clause may be no longer evaluated at the initial time, because the initialization of the when-condition is now defined in a more meaningful way (before Modelica 1.4, every condition in a when-clause has a "previous" value of false), and (4) models containing the nondiscrete keyword which was removed.

Modelica 1.3 and older versions.

Modelica 1.3 was released December 15, 1999.

Contributors up to Modelica 1.3

The following list contributors and their affiliations at the time when Modelica 1.3 was released.

H. Elmqvist[1],

B. Bachmann[2], F. Boudaud[3], J. Broenink[4], D. Brück[1], T. Ernst[5], R. Franke[6], P. Fritzson[7], A. Jeandel[3], P. Grozman[12], K. Juslin[8], D. Kågedal[7], M. Klose[9], N. Loubere[3], S. E. Mattsson[1], P. J. Mostermann[11], H. Nilsson[7], H. Olsson[1], M. Otter[11], P. Sahlin[12], A. Schneider[13], M. Tiller[15], H. Tummescheit[10], H. Vangheluwe[16]

[1] Dynasim AB, Lund, Sweden
[2] ABB Corporate Research Center Heidelberg
[3] Gaz de France, Paris, France
[4] University of Twente, Enschede, Netherlands
[5] GMD FIRST, Berlin, Germany
[6] ABB Network Partner Ltd. Baden, Switzerland
[7] Linköping University, Sweden
[8] VTT, Espoo, Finland
[9] Technical University of Berlin, Germany
[10] Lund University, Sweden
[11] DLR Oberpfaffenhofen, Germany
[12] Bris Data AB, Stockholm, Sweden
[13] Fraunhofer Institute for Integrated Circuits, Dresden, Germany
[14] DLR, Cologne, Germany
[15] Ford Motor Company, Detroit, U.S.A.
[16] University of Gent, Belgium

Main changes in Modelica 1.3

Modelica 1.3 was released December 15, 1999.

- Defined connection semantics for inner/outer connectors.
- Defined semantics for protected element.
- Defined that least variable variability prefix wins.
- Improved semantic definition of array expressions.
- Defined scope of for-loop variables.

Main changes in Modelica 1.2

Modelica 1.2 was released June 15, 1999.

- Changed the external function interface to give greater flexibility.
- Introduced inner/outer for dynamic types.
- Redefined final keyword to only restrict further modification.
- Restricted redeclaration to replaceable elements.
- Defined semantics for if-clauses.
- Defined allowed code optimizations.
- Refined the semantics of event-handling.
- Introduced fixed and nominal attributes.

- Introduced terminate and analysisType.

Main Changes in Modelica 1.1

Modelica 1.1 was released in December 1998.

Major changes:

- Specification as a separate document from the rationale.
- Introduced prefixes discrete and nondiscrete.
- Introduced pre and when.
- Defined semantics for array expressions.
- Introduced built-in functions and operators (only connect was present in Modelica 1.0).

Modelica 1.0

Modelica 1, the first version of Modelica, was released in September 1997, and had the language specification as a short appendix to the rationale.