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

TUTORIAL and RATIONALE

Version 1.2
June 15, 1999

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. Kågedahl[7], M. Klose[9], N. Loubere[3], S. E. Mattsson[1], P. Mostermann[11], H. Nilsson[7], M. Otter[11], P. Sahlin[12], A. Schneider[13], H. Tummescheit[10], H. Vangheluwe[15]

[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] University of Gent, Belgium

Contents

1. Introduction

There definitely is an interoperability problem amongst the large variety of modeling and simulation environments available today, and it gets more pressing every year with the trend towards ever more complex and heterogeneous systems to be simulated. The main cause of this problem is the absence of a state-of-the-art, standardized external model representation. Modeling languages, where employed, often do not adequately support the structuring of large, complex models and the process of model evolution in general. This support is usually provided by sophisticated graphical user interfaces - an approach which is capable of greatly improving the user's productivity, but at the price of specialization to a certain modeling formalism or application domain, or even uniqueness to a specific software package. It therefore is of no help with regard to the interoperability problem.

Among the recent research results in modeling and simulation, two concepts have strong relevance to this problem:

- *Object oriented modeling languages* already demonstrated how object oriented concepts can be successfully employed to support hierarchical structuring, reuse and evolution of large and complex models independent from the application domain and specialized graphical formalisms.

- *Non-causal modeling* demonstrated that the traditional simulation abstraction - the input/output block - can be generalized by relaxing the causality constraints, i.e., by not committing ports to an 'input' or 'output' role early, and that this generalization enables both more simple models and more efficient simulation while retaining the capability to include submodels with fixed input/output roles.

Examples of object-oriented and/or non-causal modeling languages include: ASCEND, Dymola, gPROMS, NMF, ObjectMath, Omola, SIDOPS+, Smile, U.L.M., ALLAN, and VHDL-AMS.

The combined power of these concepts together with proven technology from existing modeling languages justifies a new attempt at introducing interoperability and openness to the world of modeling and simulation systems.

Having started as an action within ESPRIT project "Simulation in Europe Basic Research Working Group (SiE-WG)" and currently operating as Technical Committee 1 within Eurosim and Technical Chapter on Modelica within Society for Computer Simulation International, a working group made up of simulation tool builders, users from different application domains, and computer scientists has made an attempt to unify the concepts and introduce a common modeling language. This language, called *Modelica*, is intended for modeling within many application domains (for example: electrical circuits, multi-body systems, drive trains, hydraulics, thermodynamical systems and chemical systems) and possibly using several formalisms (for example: ODE, DAE, bond graphs, finite state automata and Petri nets). Tools which might be general purpose or specialized to certain formalism and/or domain will store the models in the Modelica format in order to allow exchange of models between tools and between users. Much of the Modelica syntax will be hidden from the end-user because, in most cases, a graphical user interface will be used to build models by selecting icons for model components,

using dialogue boxes for parameter entry and connecting components graphically.

The work started in the continuous time domain since there is a common mathematical framework in the form of differential-algebraic equations (DAE) and there are several existing modeling languages based on similar ideas. There is also significant experience of using these languages in various applications. It thus seems to be appropriate to collect all knowledge and experience and design a new unified modeling language or neutral format for model representation. The short range goal was to design a modeling language for differential-algebraic equation systems with some discrete event features to handle discontinuities and sampled systems. The design should be extendible in order that the goal can be expanded to design a multi-formalism, multi-domain, general-purpose modeling language. This is a report of the design state as of December 1998, Modelica version 1.1.

The object-oriented, non-causal modeling methodology and the corresponding standard model representation, Modelica, should be compared with at least four alternatives. Firstly, established commercial *general purpose simulation tools*, such as ACSL, EASY5, SIMULINK, System Build and others, are continually developed and Modelica will have to offer significant practical advantages with respect to these. Secondly, *special purpose simulation programs* for electronics (Spice, Saber, etc), multibody systems (ADAMS, DADS, SIMPACK, etc), chemical processes (ASPEN Plus, SpeedUp, etc) have specialized GUI and strong model libraries. However, they lack the multi-domain capabilities. Thirdly, many industrial simulation studies are still done without the use of any general purpose simulation tool, but rather relying on *numerical subroutine libraries and traditional programming languages*. Based on experience with present tools, many users in this category frequently doubt that any general purpose method is capable of offering sufficient efficiency and robustness for their application. Forthly, an IEEE supported *alternative language standardization effort* is underway: VHDL-AMS.

Most engineers and scientists recognize the advantages of an expressive and standardized modeling language. Unlike a few years ago, they are today ready to sacrifice reasonable amounts of short-term advantages for the benefit of abstract things like potential abundance of compatible tools, sound model architecture, and future availability of ready-made model libraries. In this respect, the time is ripe for a new standardization proposal. Another significant argument in favor of a new modeling language lies in recent achievements by present languages using a *non-causal* modeling paradigm. In the last few years, it has in several cases been proved that non-causal simulation techniques not only compare to, but outperform special purpose tools on applications that are far beyond the capability of established block oriented simulation tools. Examples exist in multi-body and mechatronics simulation, building simulation, and in chemical process plant simulation. A combination of modern numerical techniques and computer algebra methods give rise to this advantage. However, these non-causal modeling and simulation packages are not general enough, and exchange of models between different packages is not possible, i.e. a new unified language is needed. Furthermore, text books promoting the object-oriented, non-causal methodology are now available, such as Cellier (1991), and university courses are given in many countries.

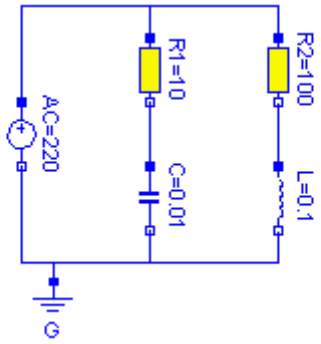
The next section will give an introduction to the basic concepts of Modelica by means of a small example. Requirements for this type of language are then discussed. Section 4 is the main section and it gradually introduces the constructs of Modelica and discusses the rationale behind them. It

is followed by an overview of present object-oriented equation based modeling languages that have been used as a basis for the Modelica language design. The design rationale from a computer science point of view is given in section 6. Syntax and detailed semantics as well as the Modelica standard library are presented in the appendices of the Language Specification.

---

2. 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 replace the **model**, **type**, and **connector** 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 modeller 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.

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 TwoPin "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 TwoPin;
```

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 TwoPin and add a definition of parameter for the resistance and Ohm's law to define the behavior:

```
model Resistor "Ideal electrical resistor"
  extends TwoPin;
  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 TwoPin;
  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 TwoPin;
  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;
```

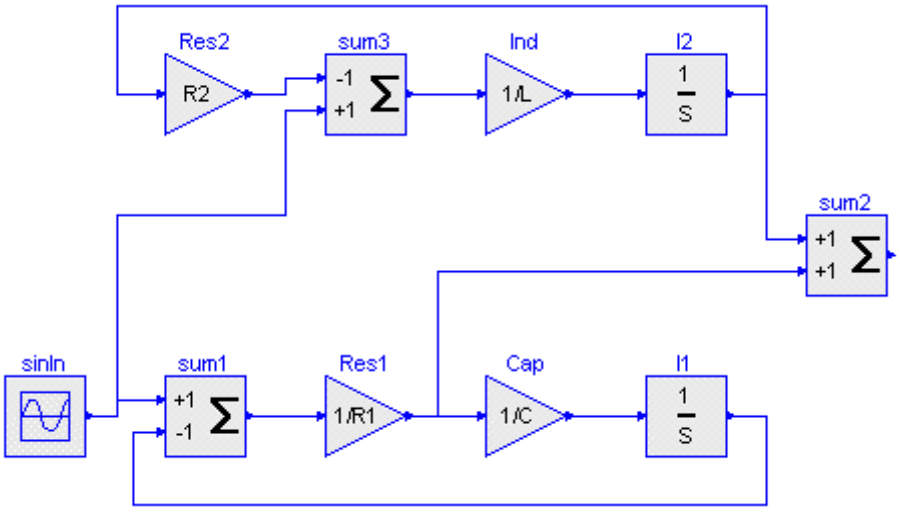
In order to provide not too much information at this stage, the constant `PI` is explicitly declared, although it is usually imported from the Modelica standard library (see appendix of the Language Specification). Finally, we must not forget the ground point.

```
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.

---

3. Requirements for Multi-domain Modeling

In this section, the most important requirements used for the Modelica language design are collected together.

The Modelica language should support both ODE and DAE (differential-algebraic equations) formulations of models. The mixture of DAE and discrete events should be possible and be defined in such a way that efficient simulation can be performed. Other data types than real, such as integer, Boolean and string should be supported. External functions written in common programming languages need to be supported in addition to a data type corresponding to external object references. It should be possible to express information about units used and minimum and maximum allowed values for a variable in order that a modeling tool might do consistency checking. It should be possible to parameterize models with both values of certain quantities and also with respect to model representation, i.e., allowing, for example, to select different levels of detail for a model. Component arrays and the connection of elements of such arrays should be supported. In order to allow exchange of models between different tools, also a certain standardization of graphical attributes for icon definitions and object diagrams should be done within the Modelica definition.

Certain modeling features will be added in later stages of the Modelica design. One example is to allow partial differential equations. More advanced discrete event modeling facilities will also be considered then, for example to allow queue handling and dynamical creation of model instances, see (Elmqvist, et.al. 1998).

Besides requirements for modeling in general, every discipline has its specific peculiarities and difficulties which often require special consideration. In the following sections, such requirements from multiple domains are presented.

Block Diagrams

Block diagrams consist of input/output blocks. For the definition of linear state space systems and transfer functions *matrices* and *matrix equations* are needed. This is most conveniently done with a MATLAB and/or Mathematica-like notation.

It is also important to support fixed and variable *time delays*. This could be done by calling an external function which interpolates in past values. However, if a delay is defined via a specific language construct, it is in principle possible to use a specific integrator to take care of the delay which can be done in a better numerical way than in the first case. Therefore, a delay operator should be defined in the language which leaves the actual implementation to the Modelica translator. Furthermore, interpolation in 1-, 2-, n-dimensional *tables* with fixed and variable grids has to be supported, because technical models often contain tables of measured data.

If it is known that a component is an input/output block, *local analysis* of the equations is possible which improves the error diagnostics considerably. For example, it can be detected whether the number of unknown variables of the block matches the number of equations in the block. Therefore, it should be possible to state explicitly that a model component is an input/output block.

Multi-Body Systems

Multi-body systems are used to model 3-dimensional mechanical systems, such as robots, satellites and vehicles. Nearly all variables in multi-body system modeling are vectors or matrices and the equations are most naturally formulated as matrix equations. Therefore, support of matrices is essential. This should include the **cross** operator for the vector cross-product because this operation often occurs in mechanical equations. It is convenient to have multi-body objects with several interfaces, but without requiring that every interface has to be connected for a model. For example, revolute and prismatic joints should have an additional interface to attach a drive train to drive the joint.

Usually, multi-body algorithms are written in such a way that components cannot be connected together in an arbitrary way. To ensure that an erroneous connection cannot be created, it should be possible to define *rules* about the connection structure. Rules help to provide a meaningful error message as early as possible.

In order that Modelica will be attractive to use for modeling of multi-body systems, *efficiency* is crucial. It must be possible that Modelica generated code is as efficient as that of special purpose multi-body programs. For that, operators like **symmetric** and **orthogonal** are necessary in order to be able to state that a matrix is symmetric or orthogonal, respectively.

Electrical and Electronic Circuits

Models of different complexity to describe electrical components are often needed. Therefore, it should be easy to *replace* a specific model description of a component by another one in the model of an electrical circuit.

It might be advantageous to implement complicated elements, such as detailed transistor models, by procedural code. This may be either an external C or C++ function or a Modelica function. In any case, the model equations are already sorted and are not expanded, i.e., every instance uses the same "function call". This is especially important, if a large number of instances are present.

It is essential that SPICE net list descriptions of electrical circuits can be used within Modelica, because vendor models of electric hardware components are described in this format. It seems sufficient to provide the SPICE component models as classes in a Modelica library and to rely on an external tool which transforms a SPICE net list description into a composite Modelica model.

Besides non-linear simulation, *small signal analysis* is often needed for electrical circuits. This

implies linearization and frequency response calculation. Numerical linearization introduces unnecessary errors. For electrical circuits it is almost always possible to symbolically differentiate the components. Special language constructs are probably not needed because in principle Modelica translators can be realized which derive the (symbolically) linearized components automatically. Modern electric circuit programs use symbolic Jacobians to enhance the efficiency. Similar to linearization, it should be possible to compute the symbolic Jacobian from a Modelica model by symbolic differentiation. If a component is provided as external function, it should be possible to provide an external function for the corresponding Jacobian of the component as well.

Chemical and Thermodynamic Systems

Processing systems for chemical or energy production are often composed of complex structures. The modeling of these systems needs encapsulation of detailed descriptions and abstraction into hierarchies in order to handle the complexity. To increase the reuse of submodels in complex structures there is a need for an advanced concept of parameterization of submodels. Especially, component arrays and class parameters are needed. An example is a structure parameter for the change of the number of trays in a distillation plant.

In order to achieve a high degree of model reuse, all medium specific data and calculation should be encapsulated in a medium properties submodel. In most cases the thermodynamic properties of the medium will be calculated externally by one of the many available specialized software packages. Thus it is necessary to provide a calling interface to external functions in ordinary programming languages. Keeping in mind both efficient simulation and model reuse, there should be a uniform way how thermodynamic properties of different external packages can be accessed from Modelica models.

Many applications in process engineering and power plant simulation can only be captured adequately with distributed parameter models. A method of lines (MOL) grid discretisation (either finite difference, finite volume or finite element methods) is the state of the art of all but a few very specialized simulation packages for modeling partial differential equations (PDEs). Modelica is envisaged as a language that is both open to future advances in numerical techniques and as an exchange format for many existing software environments. Existing simulation environments should be able to simulate Modelica code after preprocessing to DAE form. Support for PDE is planned for future versions of Modelica.

Energy domain systems

Simulation in the energy domain sector is mainly used for improving or designing technical systems: boilers, kilns, HVAC systems, pressure governors, etc. The first characteristic of these systems is that they are complex and multi-domain. For example the building energy domain deals with all types of heat exchanges, with fluid flows, with combustion, with particle pollution, with system controls, automatons etc. Modelica needs to address all these issues. It stresses the need for non-causal hierarchical modeling. To a certain extent temperature distribution and PDE

are relevant for improvement studies. Matrices and PDE features are useful. Combustion models need to address thermodynamic tables by means of a suitable feature. But, the main requirements of this domain are linked with user-friendliness, reuse, documentation, capitalization for study efficiency and reproducibility. This means that it is necessary to isolate models, isolate numerical data, isolate validation runs, integrate validity checks (domains, constraints, units, etc.) and in order to produce automatic documentation include documentation features.

Bond graphs

Bond graphs (Karnopp and Rosenberg, 1968; Breedveld, 1985) are designed to model the *energy flow* of physical systems using a small set of unified modeling primitives, such as storage, transformation and dissipation of (free) energy. Bond graphs are in principle labeled and directed graphs, in which the vertices represent submodels and the edges represent an ideal energy connection between power ports. This connection is a point-to-point connection, i.e. only one bond can be connected to a power port. When preparing for simulation, the bonds are embodied as two-signal connections with opposite directions. This signal direction depends on both the internal description of the submodel and the structure of the bond graph where the submodel is used; it is an algorithmic process. Consequently, the model equations are non-causal. Within some submodel equations, the power directions of the connected bonds are used in generating the proper equations. As a consequence, it must be possible to define *rules* about the connection structure, especially that only *one-to-one* connections are possible. Furthermore, it must be possible to inquire the *direction* of a connection in a component, in order that the *positive* energy flow direction can be deduced. Since bond graphs can be mixed with block-diagram parts, bond-graph submodels can have power ports, signal inputs and signal outputs as their interfacing elements. Furthermore, aspects like the physical domain of a bond (energy flow) can be used to support the modeling process, and should therefore be incorporated in Modelica. Note that the power bonds can be *multi dimensional*, i.e., are composed of a matrix of single power bonds. This *multi-bond* feature is used to describe, e.g., 3D mechanical systems in an elegant and compact way.

Finite Automata and Extensions

Finite automata are used to model discrete systems, such as discrete control devices as well as switching structure of clutches or idealized thyristors. Several extensions are popular, e.g., Petri nets, grafcet and state charts. It seems more flexible and powerful to build component libraries of e.g., Petri nets and state charts, using basic Modelica language constructs instead of having direct built-in language elements.

---

4. Modelica Language Rationale and 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.

4.1 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**( ).

4.2 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**.

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));
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**, **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.

4.3 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. An example of the use of cardinality and direction is the bond graph components in the standard Modelica library.

4.4 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 TwoPin as a base for all of these models.

```
partial model TwoPin
  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 TwoPin;
  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 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 TwoPin;
  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`.

4.5 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
  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. We thus think such modifications should be clearly marked by the keyword **redeclare**. Furthermore, we think the modeller of the SimpleCircuit should be able to state that such modifications are not allowed by declaring a component as **final**.

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

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

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

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);
  final 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.

4.6 Matrices

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;
```

or

```
Real position[3], velocity[3], acceleration[3], transformation[3, 3];
```

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

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

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.

4.7 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 be 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
  constant 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.

Outputs, i.e., variables that appear on the left hand side of the equal sign, which are conditionally assigned, are *initialized* to their start value *whenever the algorithm is invoked.* Due to this feature it is impossible for a function 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 expressions (one output) or as equations (several outputs). 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);
```

Similar to Java, Modelica functions can have only **one** output argument. This is not a severe restriction, because a record can be returned in which the desired output arguments are collected together. Since the appropriate style of a Modelica function call with multiple arguments is not yet fully clear - use a functional style as Matlab or Mathematica do it, or use a procedural style, as C, C++ and Fortran do it - it is not supported in the current version of Modelica.

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. More details are discussed in the appendix of the Language Specification.

4.8 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 could 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;
```

Note, that the special notation, **pre**(x), is used to denote the value of the **discrete** state variable x before the sampling.

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 must have the variable prefix **discrete**.

For non-periodic sampling a somewhat more complex method for specifying the samplings would be used. The sequence of sampling instants could 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 would thus be.

```
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;
```

Indicator functions for efficient simulation

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. 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 should have 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 no**Event** as follows.

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

Iin some cases the event does not need to be triggered exactly when the condition becomes true. It might be sufficient to wait until the next step of the integration has been completed. Such events are sometimes called step events. An appropriate translator pragma for that would be to use a function **switch**(relation).

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.

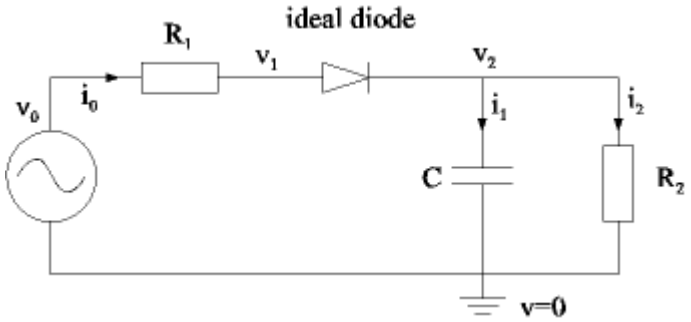


Figure 3. Rectifier circuit

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

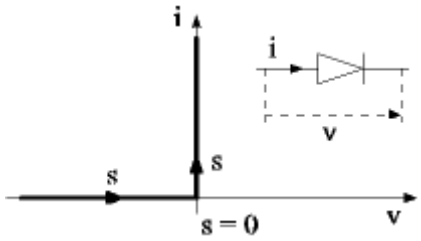


Figure 4. Characteristics of ideal diode

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 TwoPin;
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. 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.

4.9 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 a *standard package* of *predefined* quantity and connector types is provided in the form as shown in the example above. This will give some help in standardization of the interfaces of models. 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/s^2" 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
  parameter Real  freq (unit="Hz" );
  parameter Angle phi;

  replaceable type SineType = Real;
  parameter SineType Amplitude;
  output    SineType y;
  constant  Real PI=3.141592653589793;
equation
  y = Amplitude*sin(2*PI*freq*time + phi);
end SineSignal;

model Circuit
  SineSignal    sig(redeclare SineType = 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.

4.10 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).

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.

---

5. Overview of Present Languages

In this chapter an overview is given on the languages which have been used as starting point for

the Modelica design, i.e., Modelica builds upon the experience gained with these languages.

Since the definition of CSSL in 1967 (Strauss, 1967), most modeling languages are essentially block oriented with inputs and outputs and the mathematical models are defined as assignment statements for auxiliary variables and derivatives. Physical equations thus need to be transformed to a form suitable for calculations. The only aid in transforming the equations to an algorithm for calculating derivatives is automatic sorting of the equations.

The languages that form the base of Modelica, all have general equations, i.e. expression = expression, as the basic element. Hierarchical decomposition and reuse are typically supported by some kind of model class. Typically, the languages have provisions to describe physical connection mechanisms, i.e. to associate a set of variables with some kind of port. Such ports can be used at higher hierarchical levels when connecting submodels without having to deal with individual variables.

ASCEND

ASCEND (Advanced System for Computation in ENgineering Design) (http://www.cs.cmu.edu/~ascend/Home.html) was developed at Carnegie Mellon University, PA, USA to be a rapid model building environment for complex models comprising large sets of nonlinear algebraic equations (Piela 1989, Piela et.al. 1991). The language is textual. It supports quantity equations, single inheritance and hierarchical decomposition, but it does not have well defined submodel interfaces. The application domain is chemical process modeling. Later versions support dynamic continuous time modeling.

Dymola

Dymola (Dynamic Modeling Language) (http://www.dynasim.se/), as introduced already in 1978 (Elmqvist, 1978), is based on equations for non-causal modeling, model types for reuse and submodel invocation for hierarchical modeling. The Dymola translator utilizes graph theoretical methods for causality assignment, for sorting and for finding minimal systems of simultaneous equations. Computer algebra is used for solving for the unknowns and to make simplifications of the equations. Constructs for hybrid modeling, including instantaneous equations, was introduced in 1993 (Elmqvist et.al. 1993). Crossing functions for efficient handling of state events are automatically generated. A graphical editor is used to build icons and to make model compositions (Elmqvist et.al. 1996). Major application areas include multi-body systems, drive-trains, power electronics and thermal systems.

gPROMS

gPROMS (http://www.ps.ic.ac.uk/gPROMS/, Barton and Pantelides 1994, Oh and Pantelides 1996) is a general process modeling system. The language is a further development of SPEEDUP. Continuous parts of the process are modelled by DAE's. A task concept handles the discrete events. Continuous models and tasks are combined into a single entity called process. The gPROMS language has constructs for certain kinds for partial differential equations. The major application domain is chemical process modeling.

MOSES

MOSES (Modular Object-oriented Software Environment for Simulation) (http://www.elet.polimi.it/section/automeng/control/oo) is a prototype system for object-oriented modeling based on the experience with Omola. It consists of a "Model Definition Language" (MDL), a "Data Model" (DM) yielding minimum mismatch with MDL, and an object-oriented data base system based on GemStone to meet the hard data management problems involved in complex system modeling. Combined continuous and discrete-time (hybrid) systems are supported. The main application area is robotics.

NMF

The Neutral Model Format (NMF) (http://urd.ce.kth.se/, Sahlin et.al. 1996) is a language in the Dymola and Omola tradition and was first proposed as a standard to the building and energy systems simulation community in 1989. The language is formally controlled by a committee within ASHRAE (Am. Soc. for Heating, Refrigerating and Air-Conditioning Engineers). Several independently developed NMF tools and model libraries exist, and valuable lessons on language standardization and development of reusable model libraries have been learned. Salient features of NMF are: (1) good support for model documentation, (2) dynamical vector and parameter dimensions (a model can, e.g., calculate required spatial resolution for PDE), (3) full support for calls to foreign models (e.g. legacy or binary Fortran or C models) including foreign model event signals.

ObjectMath

ObjectMath (Object Oriented Mathematical Modeling Language), (http://www.ida.liu.se/labs/pelab/omath/, Fritzson et.al. 1995) is a high-level programming environment and modeling language designed as an extension to Mathematica. The language integrates object-oriented constructs such as classes, and single and multiple inheritance with computer algebra features from Mathematica. Both equations and assignment statements are included, as well as functions, control structures, and symbolic operations from standard Mathematica. Other features are parameterized classes, hierarchical composition and dynamic array dimension sizes for multi-dimensional arrays. The environment provides a class browser for the combined inheritance and composition graph and supports generation of efficient code in C++ or Fortran90. The user can influence the symbolic transformation of equations or expressions by manually specifying symbolic transformation rules, which also gives an opportunity to control the quality of generated code. The main application area so far has been in mechanical systems modeling and analysis.

Omola

Omola (http://www.control.lth.se/~cace/omsim.html, Andersson 1984, Mattsson et.al. 1993) is an object-oriented and equation based modeling language. Models can be decomposed hierarchically with well-defined interfaces that describe interaction. All model components are represented as classes. Inheritance and specialization support easy modification. Omola supports behavioral descriptions in terms of differential-algebraic equations (DAE), ordinary differential equations (ODE) and difference equations. The primitives for describing discrete events allow

implementation of high level descriptions as Petri nets and Grafcet. An interactive environment called OmSim supports modeling and simulation: graphical model editor, consistency analysis, symbolic analysis and manipulation to simplify the problem before numerical simulation, ODE and DAE solvers and interactive plotting. Applications of Omola and OmSim include chemical process systems, power generations and power networks.

SIDOPS+

SIDOPS+ (http://www.rt.el.utwente.nl/proj/modsim/modsim.htm) supports nonlinear multidimensional bond-graph and block-diagram models, which can contain continuous-time parts and discrete-time parts(Breunese and Broenink, 1997). The language has facilities for automated modeling support like polymorphic modeling (separation of the interface and the internal description), multiple representations (component graphs, physical concepts like bond graphs or ideal physical models and (acausal) equations or assignment statements), and support for reusability (e.g. documentation fields, physical types). Currently, SIDOPS+ is mainly used in the field of mechatronics and (neural) control. It is the model description language of the package 20-SIM (Broenink, 1997). SIDOPS+ is the third generation of SIDOPS which started as a model description language for single-dimensional bond-graph and block-diagram models.

Smile

Smile (http://www.first.gmd.de/smile/smile0.html) is an object-oriented and equation-based modeling and simulation environment. The object-oriented and imperative features of Smile's model *description language* are very similar to Objective-C. Equations may either be specified symbolically or as procedures; external modules can be integrated. Smile also has a dedicated *experiment description language*. The system consists of translators for the above-mentioned languages, a simulation engine offering several numeric solvers, and components for interactive experimenting, visualization, and optimization. Smile's main application domain traditionally has been the simulation of solar energy equipment and power plants (Tummescheit and Pitz-Paal, 1997), but thanks to its object-oriented modeling features it is applicable to other classes of complex systems as well. An extension of Smile to support Modelica is planned (Ernst, et.al., 1997).

U.L.M. - Allan

The goal of ALLAN (Pottier, 1983; Jeandel 1997) is to free engineers from computer science and numerical aspects, and to work towards capitalization and reuse of models. This means non-causal and hierarchical modeling. A graphical representation of the model is associated to the textual representation and can be enhanced by a graphical editor. A graphical interface is used for hierarchical model assembly. The discrete actions at the interrupts in continuous behavior are managed by events. Automatons (synchronous or asynchronous) are available on events. FORTRAN or C code can be incorporated in the models. Two translators toward the NEPTUNIX and ADASSL (modified DASSLRT) solvers are available. Main application domains are energy systems, car electrical circuits, geology and naval design.

The language U.L.M. has been designed in 1993 with the same features as the ALLAN language in a somewhat different implementation (Jeandel, 1996). It is a model exchange language linked

to ALLAN. All aspects of modeling are covered by the textual language. There is an emphasis on the separation of the model structure and the model numerical data for reuse purposes. It also has an interesting feature on model validation capitalization.

VHDL-AMS

VHDL-AMS (http://www.vhdl.org/analog, IEEE, 1997) is an extension to the discrete circuit modeling language VHDL for combined continuous and discrete models. Structuring is done by means of entities and architectures. An entity defines the external view of a component including its parameters (generics), its discrete signal interface and its continuous interface (ports). The architecture associated with an entity describes the implementation which may contain equations (DAE's). VHDL-AMS is a large and rich modeling language targeted mainly at the application domain of electronics hardware. Several extensions of VHDL towards full object orientation have been proposed (see e.g. Benzakki, et.al., 1997), but the continous modeling extensions of VHDL-AMS were not yet taken into account in this work.

---

6. Design Rationale

As already pointed out in the beginning of this chapter, Modelica is an object oriented, equation-based, declarative data-oriented modeling language for non-causal modeling of physical systems. In this section we give a short rationale of the language from a computer science point of view by explaining some of the design principles and decisions behind the language in its current form.

The following are a set of general principles and design goals that have been applied more or less consistently during the design of the Modelica language. We give several examples how these goals have influenced the current design.

- Engineering tool

  The Modelica language is designed to be an engineering tool for modeling of realistic physical systems, usually with the aim of simulating, optimizing or controlling such systems. Thus, the language has to fulfill the requirements of engineering, such as allowing efficient implementation, coping with large physical systems composed of different kinds of subsystems.

- Reliability and correctness

  The language as an engineering tool should support the construction of reliable and correct software. This goal is rather fundamental to the overall design of Modelica. For example, readability of system models is important since this contributes to reliability in engineering, even at the cost of more verbose code. This is the main reason for having named parameter passing in Modelica, also present in Ada. The strong typing in Modelica has been introduced to provide partial verification of internal consistency. The declarative and functional style of Modelica helps avoid certain errors and enhances code reuse.

- Coping with system evolution

  Large software systems are always evolving, e.g. by adding new functionality, adapting to new hardware, enhancing performance, etc. Most large software systems are always in a transitional situation where most things work and a few things do not work. We say that an evolving system is reliable if it does not break too often or too extensively in spite of change. The strong type system of Modelica is one way of controlling system evolution, by partially verifying system models at each stage. The Modelica class and package concepts, integrated with the type system, provide a module mechanism to control system complexity.

- Generality, uniformity

  The design of Modelica emphasizes generality and uniformity. This makes the language easier to learn, yet powerful. Therefore the concepts of model, type, connector, block, package and function in Modelica have been designed to be just restricted versions of the general class concept. A general static and strong type system designed by Luca Cardelli (Cardelli 1988, Cardelli 1991), has been adopted for Modelica. This type system integrates object orientation with multiple inheritance, subtyping, and parametric polymorphism - the latter also known as generics in Ada and templates in C++. Another example of uniformity and generality is that named and positional parameter passing is available for both class specialization and function calls in Modelica.

- Declarativity and referential transparency

  Most high level specification languages are declarative, including Modelica, since this allows expressing properties of systems without specifying in detail how, or in what order, such properties should be realized. For example, Modelica views object orientation as a declarative structuring concept for mathematical modeling in contrast to the non-declarative view of languages like SmallTalk, which regard object orientation as message passing between (dynamically) created objects. Modelica functions are declarative and encourages a functional programming style. They are essentially side effect free mathematical functions. The body of a function is called an algorithm section. From the equation point of view, such an algorithm section can be regarded as a strongly connected set of equations.

- Adherence to common de facto language standards

  Modelica tries to be somewhat compatible with several existing common programming languages, since this makes Modelica easier to learn and to use for engineers. For example, Modelica has adopted some of the Java syntax and the UniCode character standard, and uses the Matlab notation for matrix operations.

- High level of abstraction

  Since Modelica is a specification language, it is designed to allow abstraction from unnecessary detail. The language obtains its strong abstraction power by being based on equations integrated with object oriented structuring concepts and object connection

mechanisms.

- Code reuse

  Code reuse is a desirable but hard-to-reach goal for software development. Modelica contributes to this goal in several ways. Its non-causal equation-based modeling style permits model components to be reused in different contexts, automatically adapting to the data flow order in specific simulation applications, i.e. the Modelica compiler automatically arranges equations for solution with particular inputs or outputs. Object orientation and polymorphism significantly enhances the potential for reuse of Modelica model components.

- Mathematical foundation

  The Modelica language has a strong mathematical foundation in the sense that a Modelica model is expanded (from a semantic point of view) into a set of differential-algebraic equations. Thus, Modelica is primarily equation-based. Equations can be conditional, to represent discrete-event features and enable hybrid modeling.

---

7. 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.

---

8. Conclusions

The Modelica effort has been described and a definition of Modelica has been given. Version 1.0 was finished in September 1997. For Modelica 1.1, as defined in this report (together with the Language Specification) from December 1998, the semantic specification was considerably enhanced, especially for redeclarations, array language elements, hybrid features, lexical scoping and library support. Furthermore, a partial formal specification of the language semantics was developed (Kågedal, 1998). More than 20 papers have been written about various aspects of Modelica. See the URL below.

The design of standard function and model libraries is in progress. There is ongoing work to write books on the Modelica language and on Modelica model libraries. Several Modelica tools are also under development. There are discussions to extend the Modelica design into, for

example, handling partial differential equations and discrete event models, see Elmqvist et.al. 1998.

More information and the most actual status of the Modelica effort can be found at

```
URL: http://www.Modelica.org
```

---

9. Acknowledgments

The authors are thankful for all the feedback that has been obtained from various people that have reviewed different versions of the design.

---

10. References

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

Andersson M.: *Object-Oriented Modeling and Simulation of Hybrid Systems.* PhD thesis ISRN LUTFD2/TFRT--1043--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, December 1994.

Barby J.A.: "The need for a unified modeling language and VHDL-A". In *Proceedings of the 1996 IEEE International Symposium on Computer-Aided Control System Design*, pp. 258--263, Dearborn, Mi, USA, September 1996.

Barton P.I., and **C.C. Pantelides**: "Modeling of combined discrete/continuous processes". *AIChE J.*, **40**, pp. 966--979, 1994.

Benzakki J., and **Djafri B.**: "Object-Oriented Extensions to VHDL - the LaMI proposal". Proc. CHDL'97, pp. 334-347.

Biersack M., V. Friesen, S. Jähnichen, M. Klose, and **M. Simons**: "Towards an architecture for simulation environments." In **Vren and Birta**, Eds., *Proceedings of the Summer Computer Simulation Conference (SCSC'95)*, pp. 205--212. The Society for Computer Simulation, 1995.

Breedveld P.C: "Multiple bond graph elements in physical systems theory". *Journal of the Franklin Institute*, vol. 319, no. 1/2 pp. 1-36, 1985.

Breunese A.P.J., and **J.F. Broenink**: *Modeling mechatronic systems using the SIDOPS+ language*, Proceedings of ICBGM'97, 3rd International Conference on Bond Graph Modeling and Simulation, Phoenix, Arizona, January 12-15, 1997, SCS Publishing, San Diego, California, Simulation Series, Vol.29, No.1, ISBN 1-56555-050-1, pp 301-306.

Broenink J.F: *Modeling, Simulation and Analysis with 20-SIM*, Journal A, (Benelux quarterly journal on automatic control), Vol 38 no 3, 1997. See also http://www.rt.el.utwente.nl/20sim.

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.

Cellier F.E: *Continuous system modeling*. Springer Verlag, ISBN 0 387 97502 0, 1991.

Elmqvist H.: *A Structured Model Language for Large Continuous Systems*. PhD thesis, ISRN LUTFD2/TFRT--1015--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 1978.

Elmqvist H., D. Brück, and **M. Otter**: *Dymola --- User's Manual*. Dynasim AB, Research Park Ideon, Lund, Sweden, 1996.

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.

Elmqvist H., S.E. Mattsson, and **M. Otter.:** "Modelica - An International Effort to Design an Object-Oriented Modeling Language", Summer Computer Simulation Conference -98 , Reno, Nevada, USA, July 19-22, 1998.

Ernst T., S. Jähnichen, and **M. Klose:** "The Architecture of the Smile/M Simulation Environment". *Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Vol. 6, Berlin, Germany, pp. 653-658, 1997

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.

Fritzson P., L. Viklund, D. Fritzson, and **J. Herber**: "High-level mathematical modeling and programming" *IEEE Software*, **12:3**, July 1995.

Gautier T., P. Le Guernic, and O. Maffeis.: "For a New Real-Time Methodology", Publication Interne No. 870, Institut de Recherche en Informatique et Systemes Aleatoires, Campus de Beaulieu, 35042 Rennes Cedex, France, 1994.

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

IEEE: "Standard VHDL Analog and Mixed-Signal Extensions". Technical Report IEEE 1076.1, IEEE, March 1997.

Jeandel A., F. Boudaud, Ph. Ravier, and **A. Buhsing**: "U.L.M: Un Langage de Modélisation, a modelling language". In *Proceedings of the CESA'96 IMACS Multiconference*. IMACS, Lille,

France, July 1996.

Jeandel A., Ph. Ravier, and **A. Buhsing**: "U.L.M.: Reference guide". Technical Report M DéGIMA.1205, Gaz de France, 1995.

Jeandel A., **F. Boudaud**., and **E. Larivière**: "ALLAN.Simuation release 3.1 description" M.DéGIMA.GSA1887. GAZ DE FRANCE, DR, Saint Denis La plaine, FRANCE, 1997.

Karnopp D.C., and **R.C. Rosenberg**: *Analysis and simulation of multiport systems - the bond graph approach to physical system dynamics.* MIT Press, Cambridge, MA, USA, 1968.

Kloas M., V. Friesen, and **M. Simons**: "Smile - A simulation environment for energy systems." In **Sydow**, Ed., *Proceedings of the 5th International IMACS-Symposium on Systems Analysis and Simulation (SAS'95)*, volume 18--19 of *Systems Analysis Modelling Simulation*, pp. 503--506. Gordon and Breach Publishers, 1995.

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., M. Andersson, and **K. J. Åström**: "Object-oriented modelling and simulation". In **Linkens**, Ed., *CAD for Control Systems*, chapter 2, pp. 31--69. Marcel Dekker Inc, New York, 1993.

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

Mosterman P. J., and **G. Biswas**: "A Formal Hybrid Modeling Scheme for Handling Discontinuities in Physical System Models". Proceedings of AAAI-96, pp. 905-990, August 2.-4., Portland, OR, 1996 (http://www.vuse.vanderbilt.edu/~pjm/papers/aaai96/p.html).

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.

Oh M., and **C.C. Pantelides**: "A modelling and simulation language for combined lumped and distributed parameter systems". *Computers and Chemical Engineering*, **20**, pp. 611--633, 1996.

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

Piela P.C.: *ASCEND: An Object-Oriented Environment for Modeling and Analysis.* PhD thesis EDRC 02-09-89, Engineering Design Research Center, Carnegie Mellon Univeristy, Pittsburgh, PA, USA, 1989.

Piela P.C., T.G. Epperly, K.M. Westerberg, and **A.W. Westerberg**: "ASCEND: An object-oriented computer environment for modeling and analysis: the modeling language". *Computers and Chemical Engineering*, **15:1**, pp. 53--72, 1991.

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

Pottier M: "Extensions et applications envisageables des procédures complémentaires établies pour accéder au progiciel ASTEC 3 : ALLAN 6" Technical report M.D6 n°4034. GAZ DE FRANCE, DETN, Saint Denis La plaine, FRANCE, 1983.

Sahlin P., A. Bring, and **E.F. Sowell**: "The Neutral Model Format for building simulation, Version 3.02". Technical Report, Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden, June 1996.

Strauss J.C., D.C. Augustin, M.S. Fineberg, B.B. Johnson, R.N. Linebarger, and **F.H. Sanson**: *The SCI Continuous System Simulation Language (CSSL).* Simulation, December 1967.

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.

Vangheluwe H. L., Eugène J.H. Kerckhoffs, and **Ghislain C. Vansteenkiste**: "Simulation for the Future: Progress of the ESPRIT Basic Research working group 8467". In **Bruzzone and Kerckhoffs**, Eds., *Proceedings of the 1996 European Simulation Symposium (Genoa)*, pp. XXIX -- XXXIV. Society for Computer Simulation International (SCS), October 1996.

Viklund L.,and **P. Fritzson**: "ObjectMath --- An object-oriented language and environment for symbolic and numerical processing in scientific computing". *Scientific Programming*, **4**, pp. 229--250, 1995.