AN OVERVIEW OF THE MODELING LANGUAGE MODELICA

Sven Erik Mattsson[1], Hilding Elmqvist[2]

[1]Dept of Automatic Control, Lund University, Box 118, SE-221 00 Lund, Sweden, SvenErik@control.lth.se
[2]Dynasim AB, Research Park Ideon, SE-223 70 Lund, Sweden, Elmqvist@Dynasim.se

Abstract. A new language called Modelica™ for physical modeling has been developed in an international effort. The main objective is to make it easy to exchange models and model libraries. The design approach builds on non-causal modeling with true ordinary differential and algebraic equations and the use of object-oriented constructs to facilitate reuse of modeling knowledge.

Introduction

Mathematical modeling and simulation are emerging as key technologies in engineering. Relevant computerized tools, suitable for integration with traditional design methods are essential to meet future needs of efficient engineering. There is a large amount of simulation software on the market. However, languages and model representations are proprietary and developed for certain tools. Most general-purpose tools are based on the same modeling methodology, input-output blocks, as in the previous standardization effort, CSSL, from 1967 [6], when it was necessary to put more concern on computational aspects than on user aspects. The modeller has to perform the tedious work of transforming conservation laws and constitutive equation into an explicit ordinary differential equation, ODE, system. Domain-oriented packages are with few exceptions, only strong in one domain and are not capable of modeling components in other domains reasonably. This is a major disadvantage since technical systems are becoming more and more heterogeneous with components from many engineering domains.

Techniques for general-purpose physical modeling have been developed during the last decades, but did not receive much attention from the simulation market. The modern approaches build on non-causal modeling with true equations and the use of object-oriented constructs to facilitate reuse of modeling knowledge. There are already several modeling languages with such a support available from universities and small companies. There is also significant experience of using these languages in various applications.

In October 1996 an international effort started to unify the concepts of these languages in order to introduce common basic syntax and semantics and to design a new unified modeling language. The language is called Modelica[1]. The main objective is to make it easy to exchange models and model libraries and to allow users to benefit from the advances in object-oriented modeling methodology. In February 1997 the Modelica design effort became a Technical Committee within the Federation of European Simulation Societies, EUROSIM. The Modelica effort started in the continuous time domain since there is a common mathematical framework in the form of differential-algebraic equation (DAE) systems and there are several existing modeling languages based on similar ideas. Modelica version 1.0, was finished in September 1997. It is based on DAE systems with some discrete-event features to handle discontinuities and sampled systems.

The Modelica Design Group: *Manuel Alfonseca*, Universidad Autonoma de Madrid, Spain, *Bernhard Bachmann*, ABB Corporate Research, Baden-Dättwil, Switzerland, *Fabrice Boudaud* and *Alexandre Jeandel*, Gaz de France, *Jan Broenink*, University of Twente, The Netherlands, *Dag Brück* and *Hilding Elmqvist* (chairman), Dynasim AB, Lund, Sweden, *Thilo Ernst*, GMD-FIRST, Berlin, Germany, *Rüdiger Franke*, Technical University of Ilmenau, Germany, *Peter Fritzson*, Linköping University, Sweden, *Kaj Juslin*, VTT, Finland, *Matthias Klose*, Technical University of Berlin, Germany, *Sven Erik Mattsson*, Lund University, Sweden, *Pieter Mosterman* and *Martin Otter*, DLR Oberpfaffenhofen, Germany, *Per Sahlin*, BrisData AB, Stockholm, Sweden, *André Schneider* and *Peter Schwarz*, Fraunhofer Institute for Integrated Circuits, Dresden, Germany, *Hubertus Tummescheit*, GMD FIRST, Berlin, Germany, *Hans Vangheluwe*, University of Gent, Belgium.

[1]Modelica™ is a trade mark of the Modelica Design Group

Modelica Fundamentals

In order to give an introduction to Modelica we will consider modeling of a simple electrical circuit as defined in Figure 1. 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. Using a graphical model editor we can define a model by drawing an object diagram as shown in Figure 1, by positioning icons that represent the models of the components and drawing connections.

A Modelica model of the circuit is given in Figure 2. It is a composite model which specifies the topology of the system to be modeled in terms of components and connections between the components. The statement "`Resistor R1 (R=10);`" declares a component `R1` of class `Resistor` and sets the default value of the resistance `R` to 10.

Variables and connectors

Connections specify interactions between components. A connector must contain all quantities needed to describe the interaction. For electrical components we need the quantities voltage and current. A type for voltage quantities is declared as **`type`** `Voltage = Real(Unit = "V");` where `Real` is the name of a predefined type. A real variable has a set of attributes such as unit of measure, minimum value, maximum value and initial value.

To simplify the use of Modelica and to support compatibility, there is an extensive standard library of type definitions which always is available with a Modelica translator. The type definitions in this base library are based on ISO 1000 and its naming conventions for physical quantities. Several ISO names are long, which make them awkward in practical modeling work. For this reason, shorter alias-names are provided if necessary. The use of the name `ElectricPotential` repeatedly in a model becomes cumbersome and therefore the alternative `Voltage` is provided.

Defining a set of connector classes is a good start when developing model libraries for a new application domain. It promotes compatibility of the component models. A connector class, `Pin`, with voltage and current for electrical connections is defined in Figure 3.

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 current sums to zero at a node. Similar laws apply to flow rates in a piping network and to forces and torques in a mechanical system. The sum-to-zero equations are generated when the prefix **`flow`** is used in the connector declarations. In Modelica it is assumed that the value is positive when the current or the flow is into the component.

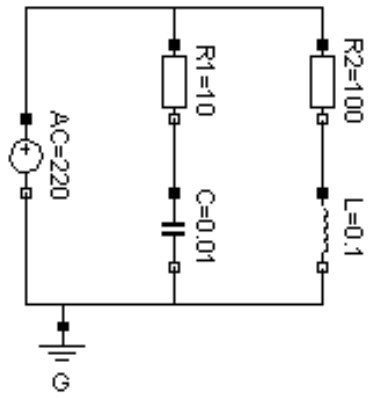

Figure 1 A simple electrical circuit.

```
model Circuit
  Resistor  R1 (R=10),   R2 (R=100);
  Capacitor C  (C=0.01);
  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);
end Circuit;
```

Figure 2 A Modelica model of the circuit in Figure 1.

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

Figure 3 A connector representing a pin.

```
partial model TwoPin
  Pin p, n;
  Voltage v;
equation
  v = p.v - n.v;   p.i + n.i = 0;
end TwoPin;
```

Figure 4 Shell model with two electrical pins.

Partial models and inheritance

A very important feature in order to build reusable descriptions is to define and reuse *partial models*. A common property of many electrical components is that they have two pins. This means that it is useful to define a "shell" model class `TwoPin`, that has two pins, `p` and `n`, and a quantity, `v`, that defines the voltage drop across the component. See Figure 4. The equations define common 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.

To define a model for a resistor, start from `TwoPin` and add a parameter for the resistance and Ohm's law to define the behavior. A model for a capacitor is defined similarly.

```
model Resistor "Ideal resistor"
  extends TwoPin;
  parameter Resistance R;
equation
  R*p.i = v;
end Resistor;
```

```
model Capacitor "Ideal capacitor"
  extends TwoPin;
  parameter Capacitance C;
equation
  C*der(v) = p.i;
end Capacitor;
```

A string between the name of a class and its body is treated as a comment attribute. Tools may display this documentation in special ways. The keyword **`parameter`** specifies that the quantity is constant during a simulation experiment, but can change values between experiments. A parameter is a quantity which makes it simple for a user to modify the behavior of a model. The expression **`der`**`(v)` means the time derivative of `v`.

Modelica's facility with `partial` and `extends` 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 [1], in particular the notion of subtyping (the structural relationship that determines type compatibility) which is different from subclassing (the mechanism for inheritance). 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. As an example, consider a varistor model

```
model Varistor
  extends TwoPin;
  parameter Resistance R;
  parameter Voltage    Uc "Cut-off voltage";
equation
  v = Uc*atanh(R/Uc*p.i);
end Varistor;
```

It is not possible to extend this model from the ideal resistor model `Resistor`, because the equation of the `Resistor` class needs to be replaced by a new equation. Still, `Varistor` is a subtype of `Resistor` because it contains all the public components of `Resistor`. This subtype relationship is especially used for class parameterization as explained in the next section, which discusses a more powerful parameterization, not only involving values like time constants and matrices but also classes.

Class Parameterization

Consider the model `Circuit` in Figure 2. Assume that we would like to exchange the resistor models with the varistor model, `Varistor`, while retaining the parameter values given for `R1.R` and `R2.R` and the circuit topology. This can be accomplished by redeclaring `R1` and `R2`:

```
model Circ2 = Circuit(redeclare Varistor R1, redeclare Varistor R2);
```

It is possible to replace the ideal resistor model, since `Varistor` is a subtype of `Resistor`. A value can be given to the parameter `Uc` in the redeclaration: `redeclare Varistor R1(Uc=100);`

This is a very strong modification of the circuit model and there is the issue of possible invalidation of the model. The keyword `redeclare` clearly marks such modifications. Furthermore, the modeller of `Circuit` is able to state that such modifications are not allowed by declaring a component as final: `final Resistor R2(R=100);` It is also possible to state that a parameter is frozen to a certain value, i.e., is not a parameter anymore: `Resistor R2(final R=100);`

To use another resistor model in `Circuit` we needed to know that there were two replaceable resistors and their names. To avoid this problem and prepare for replacement of a set of models, one can define a replaceable class, `ResistorModel` in the circuit model:

```
model Circuit2
replaceable model ResistorModel = Resistor;
protected
  ResistorModel R1(R=10), R2(R=100);
  // then as Circuit.
```

The replaceable model `ResistorModel` is declared to be of type `Resistor`. This means that it will be enforced that the actual class will be a subtype of `Resistor`, i.e., have compatible connectors and parameters. It must have `Pins p` and `n` and a parameter `R`. Default for `ResistorModel`, i.e., when no actual redeclaration is made, is `Resistor`. Setting `Resistormodel` to `Varistor` is done as

```
model MyCircuit2 = Circuit2(redeclare model ResistorModel = Varistor);
```

Replaceable classes are useful for medium parametrization. It makes it possible to separate medium properties from device properties in libraries for process components such as pumps, heat exchangers, chemical reactors etc. [4]. Another example is a controlled plant where some PID controllers are replaced with auto tuning controllers. It is of course possible to just replace those controllers in a graphical user environment, i.e., to create a new model. The problem with this solution is that two models must be maintained. Modelica has the capability to instead just substitute the model class of certain components using a language construct at the highest hierarchical level, so only one version of the rest of the model is needed.

Matrices

Modeling of, for example, multi-body systems, control systems and approximations to partial differential equations is done conveniently by utilizing *matrix equations*. Multi-dimensional matrices and the usual matrix operators and matrix functions are thus supported in Modelica. It is also possible to have arrays of components and to define regular connection patterns. A typical usage is the modeling of a distillation column which consists of a set of trays connected in series.

A matrix variable can be declared by appending dimensions after the name: `Real S[3, 3];` It is also possible to declare matrix types: `type Transformation = Real[3, 3];`.

To describe interaction between rigidly connected bodies in a 3D free-body diagram, define

```
connector MbsCut
  Transformation S  "Rotation matrix for frame A relative to the inertial frame";
  Position3      r0 "Vector from the origin of the inertial frame to A's origin";
  flow Force3   f  "Resultant cut-force acting at A's origin";
  flow Torque3  t  "Resultant cut-torque with respect to A's origin";
end MbsCut;
```

A model for a rigid bar can be defined as

```
model Bar "Massless bar with two mechanical cuts."
  MbsCut a b;
  parameter Position3 r[3] = [0, 0, 0]  "Position vector from a to b";
equation
  b.S = a.S;     b.r0  = a.r0 + a.S*r;   // Kinematic relations between a and b
  0 = a.f + b.f;                         // Force balance
  0 = a.t + b.t - cross(r, a.f);         // Torque balance
end Bar;
```

Other Modeling Features

Modelica supports *hybrid modeling*. Realistic physical models often contain discontinuities, discrete events or changes of structure. Examples are relays, switches, friction, impact, sampled data systems etc. Modelica has introduced special language constructs allowing a simulator to introduce efficient handling of such events.

Algorithms and functions are supported in Modelica for modeling parts of a system in procedural programming style. Constructs for including *graphical annotations* are available in order that also icons and model diagrams become portable. An extensive *Modelica base library* containing standard variable and connector types promotes reuse by standardizing on interfaces.

Conclusions

A short overview of Modelica has been given. The expressive modeling power of Modelica is large. Model classes and their instantiation form the basis of hierarchical modeling, connectors and connections correspond to physical connections of components. At the lowest level, equations are used to describe the relation between the quantities of the model.

Modelica has been used to model various kinds of systems. Modeling of automatic gearboxes for the purpose of real-time simulation is described in [5]. Such models are non-trivial because of the varying structure during gear shift utilizing clutches, free wheels and brakes. Modeling of heat exchangers is discussed in [4]. Class parameters of Modelica are used for medium parametrization and regular component structures are used for discretization in space of the heat exchanger. Thermodynamical and flow oriented models are also discussed in [3]. A Modelica library supporting the bond graph modeling methodology is described in [2].

Several Modelica tools and model libraries are under development. There are discussions to extend Modelica to support, for example, partial differential equations and discrete event models.

More information, including modeling requirements, rationale and definition of Modelica and future developments is available on WWW at `URL: http://www.Dynasim.se/Modelica/`.

Acknowledgements. The authors would like to thank the other members of the Modelica Design Group for inspiring discussions and their contributions to the Modelica design.

References

[1] M. ABADI and L. CARDELLI. *A Theory of Objects*. Springer-Verlag, 1996.

[2] J. F. BROENINK. "Bond-graph modeling in Modelica." In *Proceedings of the 1997 European Simulation Symposium (ESS'97)*, Passau, Germany, October 1997. The Society for Computer Simulation.

[3] T. ERNST, M. KLOSE, and H. TUMMESCHEIT. "Modelica and Smile — A case study applying object-oriented concepts to multi-facet modeling." In *Proceedings of the 1997 European Simulation Symposium (ESS'97)*, Passau, Germany, October 1997. The Society for Computer Simulation.

[4] S. E. MATTSSON. "On modeling of heat exchangers in Modelica." In *Proceedings of the 1997 European Simulation Symposium (ESS'97)*, Passau, Germany, October 1997. The Society for Computer Simulation.

[5] M. OTTER, C. SCHLEGEL, and H. ELMQVIST. "Modeling and realtime simulation of an automatic gearbox using Modelica." In *Proceedings of the 1997 European Simulation Symposium (ESS'97)*, Passau, Germany, October 1997. The Society for Computer Simulation.

[6] J. C. STRAUSS (ED.). "The SCi continuous system simulation language (CSSL)." *Simulation*, **9**, pp. 281–303, 1967.