Modelica – A Language for Equation-Based Physical Modeling and High Performance Simulation

Peter Fritzson

PELAB, Dept. of Computer and Information Science
Linköping University, S-58183, Linköping, Sweden
petfr@ida.liu.se

Abstract. A new language called Modelica for hierarchical physical modeling is developed through an international effort. Modelica 1.0 [http://www.Dynasim.se/Modelica] was announced in September 1997. It is an object-oriented language for modeling of physical systems for the purpose of efficient simulation. The language unifies and generalizes previous object-oriented modeling languages and techniques.

Compared to the widespread simulation languages available today this language offers two important advances: 1) non-causal modeling based on differential and algebraic equations; 2) multidomain modeling capability, i.e. it is possible to combine electrical, mechanical, thermodynamic, hydraulic etc. model components within the same application model.

A class in Modelica may contain variables (i.e. instances of other classes), equations and local class definitions. The multi-domain capability is partly based on a notion of connectors, which are classes just like any other entity in Modelica.

Simulation models can be developed using a graphical editor for connection diagrams. Connections are established just by drawing lines between objects picked from a class library. The Modelica model is translated into a set of constants, variables and equations. Equations are sorted and converted to assignment statements when possible. Strongly connected sets of equations are solved by calling a symbolic and/or numeric solver. The C/C++ code generated from Modelica models is quite efficient.

High performance parallel simulation code can be obtained either at the coarse-grained level by identifying fairly independent submodels which are simulated in parallel, or at the fine-grained level by parallelizing on clustered expression nodes in the equation graph. Preliminary results using the coarse-grained approach have been obtained in an application on simulating an autonomous aircraft watching car traffic.

1 Introduction

The use of computer simulation in industry is rapidly increasing. This is typically used to optimize products and to reduce product development cost and time. Whereas in the past it was considered sufficient to simulate subsystems separately, the current trend is to simulate increasingly complex physical systems composed of subsystems from multiple domains such as mechanic, electric, hydraulic, thermodynamic, and control system components.

1.1 Background

Many commercial simulation software packages are currently available. The market is divided into distinct domains, such as packages based on block diagrams (block-oriented tools, such as SIMULINK, System Build, ACSL), electronic programs (signal-oriented tools, such as SPICE, Saber), multibody systems (ADAMS, DADS, SIMPACK), and others. With very few exceptions, all simulation packages are strong only in one domain and are not capable of modeling components from other domains in a reasonable way. However, this is a prerequisite to be able to simulate modern products that integrate components from several domains such as for example electric, mechanic, hydraulic and control.

1.2 Problems

To summarize the current situation, there are at least three serious problems:

- High performance simulation of complex multi-domain systems is needed. Current widespread methods cannot cope with serious multi-domain modeling and simulation in an economical way.
- Simulated systems are increasingly complex. Thus, system modeling has to be based primarily on combining reusable components instead of re-inventing the wheel. A better technology is needed in creating easy-to-use reusable components.
- It is hard to achieve truly reusable components in object-oriented programming and modeling.

1.3 Proposed Solution

The goal of the Modelica project[8] is to provide practically usable solutions to these problems, based on techniques for mathematical modeling of reusable components.

Several first generation object-oriented mathematical modeling languages and simulation systems (ObjectMath [4], Dymola [2], Omola [1], NMF, gPROMS, Allan, Smile, etc.) have been developed during the past few years. These languages were applied in areas such as robotics, vehicles, thermal power plants, nuclear power plants, airplane simulation, real-time simulation of gear boxes, etc.

Several applications have shown that object-oriented modeling techniques is not only comparable to, but outperform special purpose tools on applications that are far beyond the capacity of established block-oriented simulation tools.

However, the situation of a number of different incompatible object-oriented modeling and simulation languages was not satisfactory. Therefore in the fall of 1996 a group of researchers (see Sect. 3.5) from universities and industry started work towards standardization and making this object-oriented modeling technology widely available.

The new language is called Modelica and designed for modeling dynamic behavior of engineering systems, intended to become a *de facto* standard.

Modelica is superior to current technology mainly for the following reasons:

- *Object-oriented modeling.* This technique makes it possible to create physically relevant and easy-to-use model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains. More details on object-orientation in Modelica can be found in [5] and [8].
- *Non-causal modeling.* Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation. However, for interfacing with traditional software, algorithm sections with assignments as well as external functions/procedures are also available in Modelica.
- *Physical modeling of multiple domains.* Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. In Modelica the structure of the model becomes more natural in contrast to block-oriented modeling tools. For application engineers, such "physical" components are particularly easy to combine into simulation models using a graphical editor.

1.4 Modelica view of object-oriented mathematical modeling

Traditional procedural languages such as Fortran or C, and object-oriented languages like C++, Java, Smalltalk and Simula support programming with operations on state. The state of the program includes variable values and object data. The number of objects changes dynamically. The Smalltalk view of object orientation is of sending messages between (dynamically) created objects. The Modelica view is different since the Modelica language emphasizes *structured* mathematical modeling. Object-orientation is primarily viewed as a structuring concept that is used to handle the complexity of large system descriptions. A Modelica model is primarily a declarative mathematical description, which allows analysis and equational reasoning. Dynamic system properties are expressed in a declarative way through equations.

1.5 Traditional High-Performance Software

High-performance software is traditionally developed in languages such as Fortran, C or C++, often combined with parallel libraries e.g. MPI, ScalaPack, etc. Although this paper primarily presents new aspects of Modelica examplified through non-causal classes using equations, it is possible to write algorithmic code in Modelica, and to integrate traditional software components with Modelica code. In fact, Modelica's strong view of software components [11] opens up new possibilities for component based software design and system integration.

2 A Modelica overview

Modelica programs are built from *classes*. Like in other object-oriented languages, a class contains variables, i.e. class attributes representing data. The main difference compared with traditional object-oriented languages is that instead of functions (methods) we primarily use *equations* to specify behavior. Equations can be written explicitly, like `a=b`, or be inherited from other classes. Equations can also be specified by the `connect` statement. The statement `connect(v1, v2)` expresses coupling between variables *v1* and *v2*. These variables are called *connectors* and belong to the connected objects. This gives a flexible way of specifying topology of physical systems described in an object-oriented way using Modelica.

In the following sections we introduce some basic and distinctive syntactical and semantic features of Modelica, such as connectors, encapsulation of equations, inheritance, declaration of parameters and constants.

2.1 Modelica model of an electric circuit

As an introduction to Modelica we will present a model of a simple electrical circuit as shown in Fig. 1. The system can be broken 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 such components are available in Modelica class libraries.

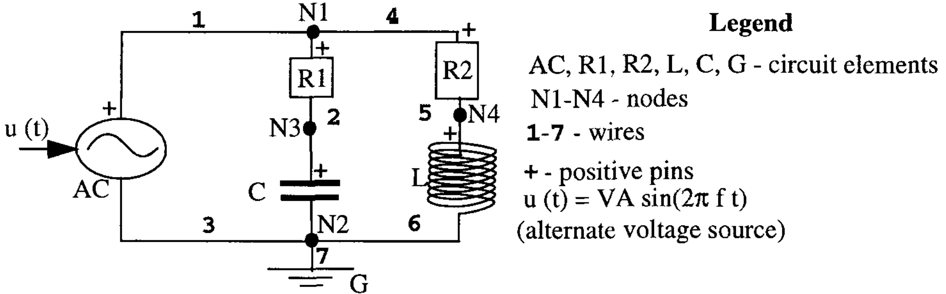

Fig. 1. A connection diagram of the simple electrical circuit example. The numbers of wires and nodes are used for reference in Table 1.

A declaration like the one below specifies `R1` to be of class `Resistor` and sets the default value of the resistance, `R`, to `10`.

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

A Modelica description of the complete circuit appears as follows:

```
class 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);     // Wire 1
    connect (R1.n, C.p);      // Wire 2
    connect (C.n,  AC.n);     // Wire 3
    connect (R1.p, R2.p);     // Wire 4
    connect (R2.n, L.p);      // Wire 5
    connect (L.n,  C.n);      // Wire 6
    connect (AC.n, G.p);      // Wire 7
  end circuit;
```

A composite model like the circuit model described above specifies the system topology, i.e. the components and the connections between the components. The connections specify interactions between the components. In some previous object-oriented modeling languages connectors are referred to cuts, ports or terminals. The keyword `connect` is a special operator that generates equations taking into account what kind of interaction is involved as explained in Sect. 2.3.

Variables declared within classes are public by default, if they are not preceded by the keyword `protected` which has the same semantics as in Java. Additional `public` or `protected` sections can appear within a class, preceded by the corresponding keyword.

2.2 Library classes

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

A connector must contain all quantities needed to describe an interaction. For electrical components we need the variables `voltage` and `current` to define interaction via a wire. The types to represent those can be declared as:

```
class Voltage = Real;
class Current = Real;
```

where `Real` is the name of a predefined variable type. A `Real` variable has a set of default attributes such as unit of measure, initial value, minimum and maximum value. These default attributes can be changed when declaring a new class, for example:

```
class Voltage = Real(unit="V", min=-220.0, max=220.0);
```

In Modelica, the basic structuring element is a `class`. There are seven restricted class categories with specific keywords, such as `type` (a class that is an extension of built-in classes, such as `Real`, or of other defined types) and `connector` (a class that does not have equations and can be used in connections). In any model the `type` and `connector` keywords can be replaced by the `class` keyword giving a semantically equivalent model. Other specific class categories are `model`, `package`, `function` and `record`

of which `model` and `record` can be replaced by `class`.

The idea of restricted classes is advantageous because the modeler does not have to learn several different concepts, except for one: the class concept. All properties of a class, such as syntax and semantics of definition, instantiation, inheritance, and generic properties are identical to all kinds of restricted classes. Furthermore, the construction of Modelica translators is simplified considerably because only the syntax and semantics of a class have 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. The previous two definitions can be expressed as follows using the keyword `type` which is equivalent to `class`, but limiting the defined type to be extension of a built-in type, record or array.

```
type Voltage = Real;
type Current = Real;
```

2.3 Connector classes

A connector class is defined as in the example below:

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

Connection statements are used to connect instances of connection classes. A connection statement `connect(Pin1,Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins so that they form one node. This implies two equations, namely:

```
Pin1.v = Pin2.v
Pin1.i + Pin2.i = 0
```

The first equation says that the voltages of the connected wire ends are the same. The second equation 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.

2.4 Virtual classes

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,

```
class TwoPin          "Superclass of elements
                       with two electric 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 out from pin n (Fig. 2).

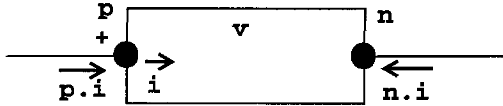

Fig. 2. Generic TwoPin model.

The equations define generic relations between quantities of a simple electrical component. In order to be useful a constitutive equation must be added.

2.5 Equations and non-causal modeling

Non-causal modeling means modeling based on equations instead of assignment statements. Equations do not specify which variables are inputs and which are outputs, whereas in assignment statements variables on the left-hand side are always outputs (results) and variables on the right-hand side are always inputs. Thus, the causality of equations-based models is unspecified and becomes fixed only when the corresponding equation systems are solved. This is called non-causal modeling.

The main advantage with non-causal modeling is that the solution direction of equations will adapt to the data flow context in which the solution is computed. The data flow context is defined by telling which variables are needed as *outputs* and which are external *inputs* to the simulated system.

The non-causality of Modelica library classes makes these more reusable than traditional classes containing assignment statements where the input-output causality is fixed.

For example, regard the equation below from the Resistor class:

```
R*i = v;
```

which can be used in two ways. The variable v can be computed as a function of i, or the variable i can be computed as a function of v as shown in the two assignment statements below:

```
i := v/R;
v := R*i;
```

In the same way the following equation from the class TwoPin:

```
v = p.v - n.v
```

can be used in three ways:

```
v   := p.v - n.v;
p.v := v + n.v;
n.v := p.v - v;
```

2.6 Inheritance, parameters and constants

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

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

The keyword `parameter` specifies that the variable is constant during a simulation run, but can have its value initialized before a run. This means that `parameter` is a special kind of constant, which is implemented as a static variable that is initialized once and never changes its value during a specific execution. A `parameter` is a variable that makes it simple for a user to modify the behavior of a model.

A Modelica `constant` never changes and can be substituted inline.

The keyword `extends` specifies the parent class. Variables, equations and `connects` are inherited from the parent. Multiple inheritance is supported in Modelica.

Analogously to C++, variables, equations and connections of the parent class cannot be *removed* in the subclass.

In C++ a virtual function can be *replaced* by a function with the same name in the child class. In Modelica 1.0 equations cannot be named and therefore in general inherited equations cannot be replaced/specialized in a child class[1]. However, equations may be replaced in the special case where they have a single identifier on When classes are inherited, equations are accumulated. This makes the equation-based semantics of the child classes consistent with the semantics of the parent class.

2.7 Time and model dynamics

Dynamic systems are models where behavior evolves as a function of *time*. We use a predefined variable `time` which steps forward during system simulation.

A class for the voltage source in our circuit example can thus be defined as:

```
class VsourceAC "Sin-wave voltage source"
  extends TwoPin;
```

1. In the ObjectMath language, one of the precursors to Modelica, equations in general can be named and thus specialized through inheritance.

```
  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;
```

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

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

The notation `der(v)` means the time derivative of `v`.

During system simulation the variables `i` and `v` evolve as functions of time. The solver of differential equations (see Sect. 3.2) computes the values of $i(t)$ and $v(t)$ (t is `time`) so that $C\,v'(t)=i(t)$ for all values of t.

Finally, we define the ground point as a reference value for the voltage levels:

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

More details on other Modelica constructs are presented in [8].

3 Implementation

3.1 Flattening of equations

Classes, instances and equations are `translated` into flat set of equations, constants and variables (see Table 1). As an example, we translate the `circuit` model from Sect. 2.1.

The equation `v=p.v-n.v` is defined by the class `TwoPin`. The `Resistor` class inherits the `TwoPin` class, including this equation. The `circuit` class contains a variable `R1` of type `Resistor`. Therefore, we include this equation instantiated for `R1` as `R1.v=R1.p.v-R1.n.v` into the set of equations.

The wire labelled `1` is represented in the model as `connect(AC.p, R1.p)`. The variables `AC.p` and `R1.p` have type `Pin`. The variable `v` is a *non-flow* variable representing voltage potential. Therefore, the equality equation `AC.p.v=R1.p.v` is generated. Equality equations are always generated when non-flow variables are connected.

Table 1: Equations generated from the simple circuit model

AC	0=AC.p.i+Ac.n.i AC.v=Ac.p.v-AC.n.v AC.i=AC.p.i AC.v=AC.VA* sin(2*PI*AC.f*time);	L	0=L.p.i+L.n.i L.v=L.p.v-L.n.v L.i=L.p.i L.v = L.L*L.der(i)
R1	0=R1.p.i+R1.n.i R1.v=R1.p.v-R1.n.v R1.i=R1.p.i R1.v = R1.R*R1.i	G	G.p.v = 0

R2	0=R2.p.i+R2.n.i
	R2.v=R2.p.v-R2.n.v
	R2.i=R2.p.i
	R2.v = R2.R*R2.i
		wires	R1.p.v=AC.p.v // wire 1
			C.p.v=R1.n.v // wire 2
			AC.n.v=C.n.v // wire 3
			R2.p.v=R1.p.v // wire 4
			L.p.v=R2.n.v // wire 5
			L.n.v=C.n.v // wire 6
			G.p.v= AC.n.v // wire 7

C	0=C.p.i+C.n.i C.v=C.p.v-C.n.v C.i=C.p.i C.i = C.C*C.der(v)	flow at node	0=AC.p.i+R1.p.i+R2.p.i // 1 0=C.n.i+G.i+AC.n.i+L.n.i // 2 0=R1.n.i+ C.p.i // 3 0 =R2.n.i + L.p.i // 4

Notice that another wire (labelled 4) is attached to the same pin, `R1.p`. This is represented by an additional connect statement: `connect(R1.p.R2.p)`. The variable `i` is declared as a `flow` variable. Thus, the equation `AC.p.i+R1.p.i+R2.p.i=0` is generated. Zero-sum equations are always generated when connecting flow variables, corresponding to Kirchhoff's current law.

The complete set of equations generated from the `circuit` class (see Table 1) consists of 32 differential-algebraic equations. These include 32 variables, as well as `time` and several parameters and constants.

3.2 Solution and simulation

After flattening, all the equations are sorted. Simplification algorithms remove most equations. If two syntactically equivalent equations appear, only one copy of the equations is kept.

Then independent equations are converted to assignment statements. If a strongly connected set of equations appears, this set is transformed by a symbolic solver which performs a number of algebraic transformations to simplify the dependencies between the variables. It can also solve a system of differential equations if it has a symbolic solution. Finally, C/C++ code is generated, and linked with a numeric solver.

The initial values can be taken from the model definition. If necessary, the user specifies the parameter values (Sect. 2.6). Numeric solvers for differential equations (such as LSODE, part of ODEPACK[7]) give the user possibility to ask about the value of specific variable at a specific moment in time. As the result a function of time, e.g.

`R2.v(t)` can be computed for a time interval $[t_0, t_1]$ and displayed as a graph or saved in a file. This data presentation is the final result of system simulation.

In most cases (but not always) the performance of generated simulation code (including the solver) is similar to hand-written C code. Often Modelica is more efficient than straightforwardly written C code, because additional opportunities for symbolic optimization are used.

3.3 Current status and future work

As a result from 8 meetings in the period September 1996 - September 1997 the first full definition of Modelica 1.0 was announced in September 1997 [8, 3]. The work by the Modelica Committee on the further development of Modelica and tools is continuing. Current issues include definition of Modelica standard libraries, efficient compilation of Modelica, parallel simulation, experimentation environments. The next revision, Modelica 1.1, is expected during the fall of 1998, including among other things more precise semantic definitions, standard libraries, etc.

3.4 Conclusion

A new object-oriented language Modelica designed for physical modeling takes some distinctive features of object-oriented and simulation languages. It offers the user a tool for expressing non-causal relations in modeled systems. Modelica is able to support physically relevant and intuitive model construction in multiple application domains. Non-causal modeling based on equations instead of procedural statements enables adequate model design and a high level of reusability.

Traditional high-performance computational software written in languages such as Fortran, C, or C++, possibly including parallel libraries, is viewed as external functions or computational blocks by Modelica and can be linked together with compiled Modelica code. Furthermore, Modelica's strong component-based view of software facilitates integration and reuse of software components in both traditional languages and in Modelica.

There is a fairly straightforward algorithm translating Modelica classes, instances and connections into a flat set of equations. Further symbolic optimizations reduces the size of the equation system and prepares for generation of efficient code. A choice of several numeric DAE solvers are available for linking with the generated code for computation of simulation results. Experience shows that Modelica is an adequate tool for design of simulation models in several different application domains, as well as for complex multi-domain models.

3.5 Acknowledgments

The Modelica definition has been developed by the Eurosim Modelica technical committee (Hilding Elmqvist, Francois Boudaud, Jan Broenink, Dag Brück, Thilo Ernst, Peter Fritzson, Alexandre Jeandel, Kaj Juslin, Matthias Klose, Sven-Erik Mattson, Martin Otter, Per Sahlin, Hubertus Tummescheit, Hans Vangheluwe) under the chairman-

ship of Hilding Elmqvist. The work by Linköping University has been supported by the Wallenberg foundation as part of the WITAS project.

References

[1] Mats Andersson: *Object-Oriented Modeling and Simulation of Hybrid Systems.* Ph.D. thesis ISRN LUTFD2/TFRT--1043--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, December 1994.

[2] Hilding Elmqvist, Dag Brück, and Martin Otter: *Dymola — User's Manual.* Dynasim AB, Research Park Ideon,Lund, Sweden, 1996, http://www.Dynasim.se

[3] Hilding Elmqvist, Sven-Erik Mattsson: Modelica - The Next Generation Modeling Language - An International Design Effort. In *Proceedings of First World Congress of System Simulation,* Singapore, September 1-3 1997.

[4] Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber. High-Level Mathematical Modelling and Programming, *IEEE Software,* 12(4):77-87, July 1995, http://www.ida.liu.se/labs/pelab/omath

[5] Peter Fritzson, Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation, In *Proceedings of ECOOP-98,* Brussels, July 1998, LNCS 1445, Springer Verlag.

[6] Dag Fritzson, Patrik Nordling. Solving Ordinary Differential Equations on Parallel Computers Applied to Dynamic Rolling Bearing Simulation. In *Parallel Programming and Applications,* P. Fritzson, L. Finmo, eds., IOS Press, 1995

[7] A.C. Hindmarsh. ODEPACK, A Systematized Collection of ODE Solvers, *Scientific Computing,* R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, 1983 (Vol. 1 of IMACS Transactions on Scientific Computation), pp. 55-64, also http://www.netlib.org/odepack/index.html

[8] Modelica Home Page. http://www.Dynasim.se/Modelica

[9] ObjectMath Home Page. http://www.ida.liu.se/labs/pelab/omath

[10] Martin Otter, C. Schlegel, and Hilding Elmqvist. Modeling and Real-time Simulation of an Automatic Gearbox using Modelica. In Proceedings of ESS'97—European Simulation Symposium, Passau, Oct. 19-23, 1997.

[11] Clemens Szyperski. *Component Software—Beyond Object-Oriented Programming.* Addison-Wesley, 1997.