MathModelica

– a new modeling and simulation environment for *Mathematica*

Mats Jirstrand and *Johan Gunnarsson*

MathCore AB
Mjärdevi Science Park
SE-583 30 Linköping
Sweden

Peter Fritzson

PELAB – Programming Environment Lab
Department of Computer and Information Science
Linköping University
SE-581 83 Linköping
Sweden

Third International Mathematica Symposium 1999,
23-25 August, RISC, Linz, Austria

Abstract

MathModelica is a *Mathematica* extension, which provides a modeling, and simulation environment for *Mathematica* based on the new standard of physical modeling languages called Modelica. Modelica is a new object-oriented multi-domain modeling language based on algebraic and differential equations. In this paper we present a language and an environment, MathModelica, that integrates different phases of the Modelica development lifecycle. This is achieved by using the *Mathematica* environment and its structured documents, “notebooks”. Simulation models are represented in the form of structured documents, which integrate source code, documentation and code transformation specifications, as well as providing control over simulation and result visualization.

Import and export of Modelica code between internal structured and external textual representation is supported. Mathematica is an interpreted language, which is suitable as a scripting language for controlling simulation and visualization. Mathematica also supports symbolic transformations on equations and algebraic expressions which is useful in building mathematical models.

1 Introduction

Integrated simulation environments are advantageous in order to work effectively and flexibly with simulations. Users prepare and run simulations as well as investigate simulation results. Several auxiliary activities accompany simulation experiments: requirements are specified, models are designed, documentation is associated with appropriate places in the models, input and output data as well as possible constraints on such data are documented and stored together with the simulation model. The user should be able to reproduce experimental results. Therefore input data and parts of output data as well as the experimenter's notes should be stored for future analysis.

Traditionally, simulation and accompanying activities have been expressed using heterogeneous media and tools:

- a simulation model is traditionally designed on paper using traditional mathematical notation;
- simulation programs are written in a low-level programming language and stored on text files;
- input and output data (if stored at all) are saved in proprietary formats needed for particular applications and numerical libraries;
- documentation is written on paper or in separate files that are not integrated with the program files;
- the graphical results are printed on paper or saved using proprietary formats.

When the result of the research and experiments, such as a scientific paper, is written, the user normally gathers together input data, algorithms, output data and its visualizations as well as notes and descriptions. One of the major problem in simulation development environments is that gathering correct versions of all these components from various files and formats is difficult and error-prone.

1.1 The Modelica Language

There is definitely an interoperability problem amongst the large variety of modeling and simulation environments available today [3] . The main cause of this problem is the absence of a state-of-the-art, standardized external model representation. Modeling languages often do not adequately support the structuring of large, complex models and the process of model evolution in general.

The language called Modelica [11] for hierarchical physical modeling is developed through an international effort. It is an object-oriented language [3] [6] for modeling of physical systems. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a de facto standard. It offers three important features: 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; 3) a general type system that unifies object-orientation, multiple inheritance, and templates within a single class construct.

Modelica models are built from classes. Like in other object-oriented languages, a class contains variables, i.e., class attributes representing data. The main difference compared to traditional object-oriented languages is that instead of functions (methods) the programmer uses equations to specify behavior. Equations can be written explicitly, like `a=b`, or can be inherited from other classes. Equations can also be specified by the `connect` statement. Equations here includes differential equations (ODE and DAE).

1.2 MathModelica - A *Mathematica* Extension

Our approach to the integration problem is based on the *Mathematica* environment and its programmable notebooks. Every notebook corresponds to one document (one file) and contains a tree structure of cells. A cell can include other cells and/or arbitrary text or graphics. In particular a cell can include a code fragment or a graph with computational results.

The contents of cells can be

- parts of models (a formal description that can be used for verification, compilation and execution of some simulation model);
- text/documentation (used as comments to executable, formal model specifications);
- dialogue forms for specification and modification of input data;
- result tables (the results can be immediately represented in table form);
- graphical result representation (with 2D vector and raster graphics as well as 3D vector and surface graphics);
- 2D graphs that are used for various model structure visualizations:
    - class diagrams
    - variable dependency diagrams
    - data structure diagrams

Apart from the *Mathematica* notebook interface, the *Mathematica* system with its kernel, programming language, and symbolic representation of code and mathematical expressions provides a powerful environment for the modeling and simulation technology as given by the Modelica language. Having a language for differential equations (DAEs) *Mathematica* ability to perform symbolic computations/transformations on expressions is very important. The *MathLink* interface that lets *Mathematica* communicate with other processes seamlessly can be used to, e.g., integrate external special purpose simulators, or graphical modeling editors. The functional and rule based language is not only good for expressing mathematica operations, but also good transformations of (code) formats to be interpreted, e.g., existing numerical data and tables that are needed to be integrated into a simulation.

MathModelica is the name of an extension of *Mathematica* that is an implementation of a modeling and simulation environment based on the Modelica language. The goal has been to integrate the Modelica language into the language of *Mathematica* as close as possible. This makes it possible for the user to utilize both *Mathematica* and the Modelica language without severe restrictions.

MathModelica means both a language and an environment. The MathModelica language is a language close to the *Mathematica* syntax with the extension of a syntax for data types, i.e., the data types of symbols and functions in *Mathematica* expressions can be specified in a convenient way. Section 2 will describe the language of MathModelica. The MathModelica environment means a collection of modeling tools (graph editors) and simulation engines where *Mathematica* is the center and user frontend. Section 3 will describe this environment briefly.

2 MathModelia – the language

2.1 Syntax

A specific feature of *Mathematica* is that models (or code) are normally not written as free formatted text. Instead, *Mathematica* expressions (terms) are used. These can be conveniently written in a tree-like prefix form, or entered using standard mathematical notation. Every term is a number, an identifier or a form such as:

```
head[term₁,..., termₙ]
```

In order to satisfy this requirement, we designed the new MathModelica language. Note that MathModelica has the same abstract syntax and the same semantics as Modelica, but different concrete syntax. This means that essentially the same language constructs are written differently, as illustrated below.

The MathModelica language uses some *Mathematica* notation, such as:

```
term₁; ...; termₙ,
{term₁, ..., termₙ},
term₁ term₂,
term₁ == term₂
```

and arbitrary arithmetic expressions composed from terms. We will not present the complete syntax of MathModelica and it's relation to Modelica here, but we will use some examples.

Note! The syntax and command names pruposed in this paper are preliminary and can be changed in future versions of MathModelica.

2.1.1 Type Operator

Consider the Modelica code:

```
model FirstOrder
    Real x(start=1);
    parameter Real a=1;
equation
    der(x)=-a*x;
end FirstOrder;
```

The above example is a class definition of the model named `"FirstOrder"`. This model includes one dynamic variable `x` of type Real with the initial value (`start`) set to 1. The symbol `a` is declared as a parameter which means that its value must be a constant and given by the user. To simulate this model means to compute values of the variable `x` starting from the value 1 such that the equations of the model are satisfied. The dynamics (differential equations) of the model is given after the `equation` keyword. The operator `der` is derivation with respect to time. If time is represented by the symbol t, `der(x)` would mean $\frac{dx(t)}{dt}$. The model definition can contain any number of equations.

The MathModelica syntax of the Modelica example above is

```
Model[FirstOrder,
   Real x[{Start == 1}];
   Parameter Real a == 1;
   Equation[
        x' == -a x
    ]
]
```

Note that the structure of the MathModelica code is the same as for Modelica but the syntax is a valid *Mathematica* expression. A few things needs to be explained here:

- In Modelica the character '=' stand for an equation and not for assignment. Therefor '==' should be used in MathModelica which means `Equal[]`.
- Space in *Mathematica* means normally multiplication (`Times`). To provide a easy to write and read syntax for data types we have introduced a operator for prefixed attributes. In this case this operator is called `TypeMark`, and it applies in certain specific places in the MathModelica language where multiplication (`Times`) is forbidden and therefore introduces no ambiguities.

To illustrate how `TypeMark` works we can take a look at the `FullForm` of a simple declaration statement that is held to prevent any calculations. The constant `dpi` is declared and assigned the value of 2*3.14.

```
Declare[
          Constant Real dpi = 2 3.14
      ] // Hold // FullForm
```

```
Hold[Declare[Set[TypeMark[Constant, Real, dpi], Times[2, 3.14`]]]]
```

We see that spaces between the type keywords are correctly interpreted to `TypeMark`.

To use space as the type operator simplifies the writing of code including data types, especially when there is several hundred variables to declare in realistic modeling projects. Once the code is written we can ask *Mathematica* to list the code such that the type operator and multiplication are presented differently. This is possible by the command `MathModelicaForm` where the `TypeMark` operator is printed using '␣'. The command `GetModel[]` returns the model definition.

```
MathModelicaForm[GetModel[FirstOrder]]
```

```
Model[FirstOrder,
   Real ␣ x[{Start == 1}];
   Parameter ␣ Real ␣ a == 1;
   Equation[
      x′ == -a x
   ]
]
```

2.1.2 Field Selector

The second extension to the *Mathematica* language necessary for the MathModelica language is the field selector. Consider the record definition below.

```
Record[Person,
   String Name;
   Integer Age
]
```

This type definition creates the type `Person` including two fields: `Name` and `Age`. Declaration and initialization of a record variable can then be done as

```
Declare[
      Person p1
   ];
```

```
p1 = Person["John", 23];
```

The record variable `p1` will now contain the expression `Person["John",23]` where `Person[]` is a constructor for the record type.

To extract fields in a record or a general class, we use the `Member` function

```
Member[p1, Name]
```

```
John
```

which selects the field named `Name` and returns it's value. The infix operator symbol for the field selector is '.' in many common programming languages. To introduce '.' as the infix field selector in MathModelica is therefore natural, except for the fact that *Mathematica* uses '.' for `Dot` (tensor product). Still, the importance to have a infix operator for the field selector is extensive since the equations generated by the MathModelica environment will contain many variables which are represented by the `Member` function. To keep the format easy both to read and to write it is necessary to have a short one character long infix operator. Unfortunately there are no free one character operators that can be easily typed on the keyboard. The '.' is therefore chosen since it is possible for the MathModelica environment to distinguish between `Member` and `Dot`, by analyzing the type of the first argument of `Dot`. If the type is a class then `Dot` is converted to `Member`.

Using '.' on a record variable returns the field

```
p1.Name
```

```
John
```

whereas '.' on vectors returns the scalar product.

```
{1, 2}.{3, 4}
```

```
11
```

In StandardForm the `Member` and `Dot` command are distinguished since the `Member` function is printed using the '˯' ([Hacek]) character.

```
pi.Age . {a, b}
```

```
(p1˯Age).{a, b}
```

2.2 Circuit Example

The details of the MathModelica language will be described by an example of a circuit model that will be given in the form of MathModelica expressions in this section. Note that we here only describe the modeling in terms of programming MathModelica textually. The MathModelica environment also includes a graphical modeling paradigms also that is based on MathModelica language. The graphical environment has an one-to-one correspondence with the textual MathModelica language.

MathModelica models are built from classes. Like in other object-oriented languages, 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 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 MathModelica.

In the following sections we introduce some basic and distinctive syntactical and semantic features of MathModelica, such as connectors, encapsulation of equations, inheritance, declaration of parameters and constants. Powerful parametrization capabilities (which are advanced features of MathModelica) are discussed in Section 2.4.

2.2.1 Connection Diagram

As an introduction to Modelica we will present a model of a simple electrical circuit as shown in Figure 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.

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

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

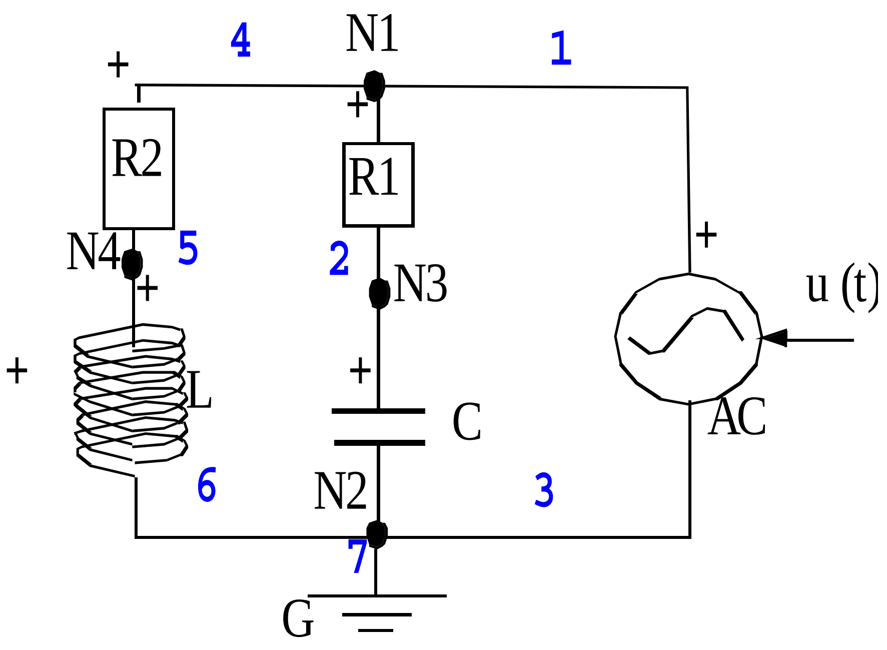


Figure 1

A MathModelica description of the complete circuit appears as follows:

```
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"
   ]
]
```

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 Section 2.2.3.

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.

2.2.2 Type Definitions

The MathModelica language is a typed language where new types can be defined. Here the type of voltage and current are defined.

```
Type[Voltage, Real[{Unit == "V"}]]
```

This defines the symbol `Voltage` to be the type of a `Real` which is a basic predefined type. Each type (including the basic types) has a collection of default attributes such as unit of measure, initial value, minimum and maximum value. These default attributes can be changed when declaring a new type. In the case above the unit of measure of `Voltage` is changed to `"V"`. The corresponding definition is also made for the current.

```
Type[Current, Real[{Unit == "A"}]]
```

In MathModelica, 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). For a valid model replacing the type and connector keywords by the keyword `Class` is fully equivalent, because the restrictions imposed by such a specialized class are fulfilled by a valid model. Other specific class categories are `Model`, `Record`, and `InOutBlock`.

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

2.2.3 Connector Classes

A connector class is defined as follows:

```
Connector[Pin,
   Voltage v;
   Flow Current i
]
```

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 (in this case one electrical connection). 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 in the declaration. 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.

2.2.4 Virtual (Partial) 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,

```
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
      ]
   ]
```

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`.

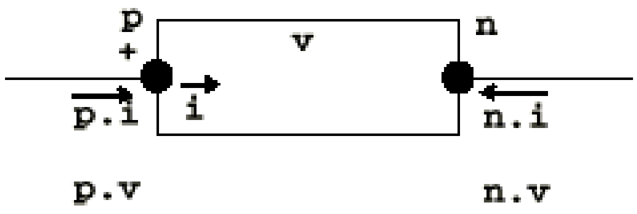


Figure 2

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 keyword 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.

String after the class name is a comment that is a part of the language, i.e., these comments are associated with the definition and are normally displayed by dialogs and forms presenting details about class definitions.

2.2.5 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 fixed only when the 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 MathModelica (Modelica) library classes makes these more reusable than traditional classes containing assignment statements where the input-output causality is fixed.

For example the equation from resistor class below:

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

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;
```

2.2.6 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:

```
Model[Resistor, "Ideal electrical resistor",
   Extends[TwoPin];
   Parameter Real R[{unit == "ohm"}]; "Resistance";
   Equation[
      R i == v
   ]
]
```

The keyword `Parameter` specifies that the variable is constant during a simulation run, but can change values between runs. 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 MathModelica constant never changes and can be substituted inline.

The keyword `Extends` specifies the parent class. All variables, equations and connects are inherited from the parent. Multiple inheritance is supported in MathModelica.

Just like in 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 the equations cannot be named and therefore we cannot replace equations. 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.

An innovation of MathModelica is that the type of a variable of the parent class can be replaced. We describe this in more detail in Section 2.4.

2.2.7 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 can be defined as:

```
Model[VsourceAC, "Sine-wave voltage source",
   Extends[TwoPin];
   Parameter Real VA == 220; "Amplitude [V]";
   Parameter Real f == 50; "Frequency [Hz]";
   Protected[
      Constant Real PI == 3.141592
   ];
   Equation[
      v == VA * Sin[2 PI f Time]
   ]
]
```

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

```
Model[Capacitor, "Ideal electrical capacitor",
   Extends[TwoPin];
   Parameter Real C[{unit == "F"}]; "Capacitance";
   Equation[
      C v' == i
   ]
]
```

```
Model[Inductor, "Ideal electrical inductor",
   Extends[TwoPin];
   Parameter Real L[{unit == "H"}]; "Inductance";
   Equation[
      L i' == v
   ]
]
```

where `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 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

```
Model[Ground, "Ground",
    Pin p;
    Equation[
        p.v == 0
    ]
]
```

2.2.8 The Complete Circuit Model

Finally we let the MathModelica system print out the complete code for the circuit model. Each `Class`, `Model`, `Type`, and `Connector` definition above stores the definitions in a symbol table. The command `ListSymbolTable` generates a list of the class definitions available in the symbol table. `MakeModel` package these definitions into a single model. The complete model in the `MathModelicaFullForm` format is stored in variable `m`.

```
m = MakeModel[ListSymbolTable[]];
```

The `MathModelicacFullForm` format is converted to input form and then the command `MathModelicaForm` is used to pretty print the code.

```
MathModelicaFullFormToInputForm[m] // MathModelicaForm
```

```
ModelicaModel[
    Model[Capacitor, Ideal electrical capacitor,
        Extends[TwoPin];
        Parameter ␣ Real ␣ C[{unit == F}]; Capacitance;
        Equation[
            C v′ == i
        ]
    ];
    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
        ]
    ];
    Type[Current,
        Real[{unit == A}]
    ];
    Model[Ground, Ground,
        Pin ␣ p;
        Equation[
            p.v == 0
        ]
    ];
    Model[Inductor, Ideal electrical inductor,
        Extends[TwoPin];
        Parameter ␣ Real ␣ L[{unit == H}]; Inductance;
        Equation[
            L i′ == v
        ]
    ];
    Connector[Pin,
        Voltage ␣ v ;
        Flow ␣ Current ␣ i
    ];
    Model[Resistor, Ideal electrical resistor,
        Extends[TwoPin];
        Parameter ␣ Real ␣ R[{unit == ohm}]; Resistance;
        Equation[
            R i == v
        ]
    ];
    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
        ]
    ];
    Type[Voltage,
        Real[{unit == A}]
    ];
    Model[VsourceAC, Sine-wave voltage source,
        Extends[TwoPin];
        Parameter ␣ Real ␣ VA == 220; Amplitude [V];
        Parameter ␣ Real ␣ f == 50; Frequency [Hz];
        Protected[
            Constant ␣ Real ␣ PI == 3.14159
        ];
        Equation[
            v == VA Sin[((2 PI) f) Time]
        ]
    ]
]
```

2.3 The MathModelica Notion of Subtypes

The notion of subtyping in MathModelica is influenced by type theory of Abbadi and Cardelli [1] . The notion of inheritance in MathModelica is separated from the notion of subtyping. According to the definition, a class A is a subtype of class B if class A contains all the public variables declared in the class B, and types of these variables are subtypes of types of corresponding variables in B. The main benefit of this definition is additional flexibility in the composition of types. For instance, the class `TempResistor` is a subtype of `Resistor`.

```
Model[TempResistor,
   Extends[TwoPin];
   Parameter Real {R, RT, Tref};
   Real T;
   Equation[
      v == i (R + RT * (T - Tref));
   ]
]
```

Subtyping is used for example in class instantiation, redeclarations and function calls. If variable a is of type A, and A is a subtype of B, then `a` can be initialized by a variable of type B. Redeclaration is discussed in the next section.

Note that TempResistor does not inherit the `Resistor` class. There are different equations for evaluation of `v`. If equations are inherited from `Resistor` then the set of equations will become inconsistent in `TempResistor`, since MathModelica currently does not support named equations and replacement of equations. For example, the specialized equation below from `TempResistor`:

```
v=i*(R+RT*(T-Tref))
```

and the general equation from class `Resistor`

```
v=R*i
```

are inconsistent.

2.4 Class Parametrization

A distinctive feature of object-oriented programming languages and environments is ability to fetch classes from standard libraries and reuse them for particular needs. Obviously, this should be done without modification of the library codes. The two main mechanisms that serve for this purpose are:

- *inheritance*. It is essentially “copying” class definition and adding more elements (variables, equations and functions) to it.
- *class parametrization* (also called generic classes or types). It is replacing a generic type identifier in whole class definition by an actual type.

In MathModelica we have a new way to control class parametrization. Assume that a library class is defined as

```
Model[SimpleCircuit,
   Resistor {R1[{R == 100}], R2[{R == 200}], R3[{R == 300}]};
   Equation[
      Connect[R1.p, R2.p];
```

```
        Connect[R1.p, R3.p]
    ]
]
```

Assume that in our particular application we would like to reuse the definition of `SimpleCircuit`: we want to use 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.

```
Type[RedefinedSimpleCircuit,
    SimpleCircuit[{
            Redeclare[TempResistor R1],
            Redeclare[TempResistor R2]
        }]
]
```

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

```
Redeclare[TempResistor R1[{RT==0.1, Tref==20.0}]]
```

This is a very strong modification but it should be noted that all equations that could be defined in `SimpleCircuit` are still valid.

3 MathModelica – the environment

3.1 MathModelicaFullForm

The MathModelica syntax presented so far is the syntax that can be given as input (InputForm) to the MathModelica system, this syntax is also used when code is printed in StandardForm where indentations and the special character for the type operator are used.

Internally in the MathModelica system uses another format called MathModelicaFullForm. This format is the abstract syntax [2] of the MathModelica language where all the elements of the language are separated canonically to be easy to extract and compare for the functions operating on the MathModelica language. See also the semantic implementation of [9].

The following simple constant declaration

```
Declare[
    Constant Real[2, 2] unitarr = {{1, 0}, {0, 1}}; "2D Identity"
]
```

is stored internally in the MathModelicaFullForm format as

```
GetMathModelicaFullForm[unitarr]
```

```
Hold[Declaration[TYPE[Real, {2, 2}, {Constant}, {}],
      VariableComponent[unitarr, ValueBinding[{{1, 0}, {0, 1}}],
         {}, {}, StringRows[2D Identity]]]]
```

A declaration of a global variable is represented by the `Declaration` node in the abstract syntax. This node has two arguments: the type and the component. The type is represented by the `TYPE` node which stores the name, array dimension, type attributes (`Constant`) and type modifications (which is empty in this case). The component argument contains a `VariableComponent` including the name of the variable, the initialization (`ValueBinding`), and in the end the comment string (`StringRows`) that is associated with the variable.

If we instead declare the type of a function we will get a similar expression in MathModelicaFullForm. (The syntax for function declarations have been introduced by the MathCode C++ system. [10])

```
Declare[
   foo[Real x_] → Real[3]
]
```

This declaration specifies that the *Mathematica* function foo[x] has the `Real` type for the input argument `x`, and a `Real` vector for the return value. The MathModelicaFullForm is in this case:

```
GetMathModelicaFullForm[foo]
```

```
Hold[Declaration[TYPE[FunctionType[{TYPE[Real, {}, {}, {}]},
            {TYPE[Real, {3}, {}, {}]}, {}], {}, {}, {}],
      FunctionComponent[foo, {x}, {Null}, foo[x_], Null, Null]]]
```

The function declaration will also create a `Declaration` node with two arguments: type and component. In this case the type expression (`TYPE`) has a `FunctionType` node as it's first argument instead of a name of a type (like `Real` in the array declaration above). The `FunctionType` node stores a list of the input arguments types and another list with the type of the output values. Note that these type are also represented by the `TYPE` node, i.e. any type can be built of nested `TYPE` expressions. The component is in this case a `FunctionComponent` which stores the function name, input argument symbol names (the formal parameters), output names, and the pattern of the function among other things not discussed here.

There are several goals behind the design of the MathModelicaFullForm format:

- Abstract Syntax: The format separates the different constructions in the language systematically making the navigation of types and code easier.

- The preserving the syntax structure of the Modelica or MathModelica code. This means that the mapping from Modelica to MathModelicaFullForm should be injective, and that transformations from Modelica to MathModelicaFullForm into MathModelica input form should be reversible.

- Symbol table format. The MathModelicaFullForm should be possible to use in the symbol table. Specially the representation of types with the `TYPE` node should be ready for efficient type inference, i.e., deriving the types of general expressions.
- Internal standard: The MathModelicaFullForm format should be used by all the components in the MathModelica system. Therefore it must be easily parsed and unparsed. By generating the FullForm format of MathModelicaFullForm we get a pure tree syntax of the format which is very easy for external programs to parse. The unparsing (e.g., to Modelica) is easy and can be done by simple table driven unparsers if the MathModelicaFullForm is has a well designed abstract syntax.

3.2 Typed Pattern Matching

The type system in MathModelica is mainly used for generating the simulation code, whereas the *Mathematica* computation is not affected of the type of a symbol or a function. However, the types can be used in pattern matching.

Assume we have a list of equations including typed variables, and that some of these variables have the type `Angle`. In particular we are interested in the expressions in the equations of the form `Sin[exp]` where the `exp` is of the type `Angle`. Assume that the values of the angle expressions are always close to zero, then to improve simulation performance we could replace each Sin expression with it's third order approximation.

This can be done by the following rule.

```
r1 = TypedPattern[Sin[Angle x_]] → Normal[Series[Sin[x], {x, 0, 3}]]
```

```
HoldPattern[Sin[x_?(TypeQ[Angle])]] → x - x^3/6
```

The `TypedPattern` command works as the inbuilt `HoldPattern` except that type information are extracted and the pattern is rewritten using a predicate test called `TypeQ`.

3.3 Simulation

In Section 2.2 the MathModelica model of the circuit was defined. This circuit model will here be simulated.

To simulate a MathModelica model a sequence of transformations must be done. The MathModelica code (or MathModelicaFullForm format) is the starting point and an executable binary file is the final result of the transformation. The simulation is performed by executing the binary file which generates a data file with the simulation data, that is then loaded into *Mathematica.* We will here present each of these steps to perform the a simulation. Note that the whole sequence is normally automated but is here manually done for illustration.

In Section 2.2.8, the MathModelicaFullForm model was stored in the the variable `m`. The first step is to export the MathModelicaFullForm format to the Modelica format since the external simulation engine supports Modelica.

```
ExportModelica["circuit.mo", m];
```

The resulting file is

```
!! circuit.mo
```

```
model Capacitor "Ideal electrical capacitor"
   extends TwoPin;
   parameter Real C(Unit="F") "Capacitance";

equation
   C*(der(v))=i;
end Capacitor;
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;
type Current = Real(Unit="A");
model Ground "Ground"
   Pin p;

equation
   p.v=0;
end Ground;
model Inductor "Ideal electrical inductor"
   extends TwoPin;
   parameter Real L(Unit="H") "Inductance";

equation
   L*(der(i))=v;
end Inductor;
connector Pin
   Voltage v;
   flow Current i;
end Pin;
model Resistor "Ideal electrical resistor"
   extends TwoPin;
   parameter Real R(Unit="ohm") "Resistance";

equation
   R*i=v;
end Resistor;
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;
type Voltage = Real(Unit="A");
model VsourceAC "Sine-wave voltage source"
   extends TwoPin;
   parameter Real VA=220 "Amplitude [V]";
   parameter Real f=50 "Frequency [Hz]";
   protected
      constant Real PI=3.141592;

equation
   v=VA*(sin(2*PI*f*Time));
end VsourceAC;
```

The command `OpenModel` will make the external simulation engine to load the Modelica file.

```
OpenModel["circuit.mo"]
```

The command `InstantiateModel` instantiate a model object of the type `Circuit`. Each component in that model class will also be instatiated.

```
InstantiateModel["Circuit"];
```

The command `TranslateModel` starts a sequence of transformations:

- The set of differential equations is extracted from the Modelica classes into one set of equations. This process is called flattening of the Modelica model, since it is equivalent to write the complete set of equations for the model into one single class.
- After flattening, all the equations are sorted. Simplification algorithms can eliminate many of them. If two syntactically equivalent equations appear only one copy of the equations is kept. Then they can be converted to assignment statements. If a strongly connected set of equations appears, these can be transformed by a symbolic solver. The symbolic solver 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 it is linked with a numeric solver.

```
TranslateModel["Circuit"];
```

The Modelica technology gives a high level modeling paradigm in which compilers, and algebraic transformation can gain performance in the simulation runs, by reducing the number of equations, and solving equations symbolically. In the MathModelica environment the user can improve the symbolic transformation using domain specific knowledge to derive transformation rules that are applied on the equations before the simulation code is generated. The MathModelica environment is an open system where the user has access to the different layers of transformations.

The final transformation produces a C code for the mode in the file "dsmodel.c".

```
!! dsmodel.c
```

```
#include <matrixop.h>
/* Prototypes for functions used in model */
/* Codes used in model */
/*
*/
/*
*/
/* DSblock model generated by Dymola from Modelica model. */

/* DSblock C-code: */

#include <moutil.c>

/* Define variable names. */

#define Sections_
#define R1_p_direction  Variable(0)
#define R1_n_direction  Variable(1)
#define R1_R  Variable(2)
#define C_p_direction  Variable(3)
#define C_n_direction  Variable(4)
#define C_C  Variable(5)
#define R2_p_direction  Variable(6)
#define R2_n_direction  Variable(7)
#define R2_R  Variable(8)
#define L_p_direction  Variable(9)
#define L_n_direction  Variable(10)
#define L_L  Variable(11)
#define AC_p_direction  Variable(12)
#define AC_n_direction  Variable(13)
#define AC_VA  Variable(14)
#define AC_f  Variable(15)
#define AC_PI  Variable(16)
#define G_p_direction  Variable(17)
#define C_n_v  Variable(18)
#define AC_n_v  Variable(19)
#define L_n_v  Variable(20)
#define G_p_v  Variable(21)
#define AC_p_v  Variable(22)
#define R1_p_i  Variable(23)
#define R1_n_v  Variable(24)
#define R1_n_i  Variable(25)
#define R1_v  Variable(26)
#define C_p_i  Variable(27)
#define C_n_i  Variable(28)
#define R2_p_i  Variable(29)
#define R2_n_v  Variable(30)
#define R2_n_i  Variable(31)
#define R2_v  Variable(32)
#define L_n_i  Variable(33)
#define L_v  Variable(34)
#define AC_p_i  Variable(35)
#define AC_n_i  Variable(36)
#define AC_v  Variable(37)
#define G_p_i  Variable(38)
#define C_v   State(0)
#define der_C_v  Derivative(0)
#define L_i   State(1)
```

```
#define der_L_i  Derivative(1)
#define CPUClk  Output(0)

TranslatedEquations
CPUClk = CurrentClockTime;

InitialSection
R1_p_direction = 1;
R1_n_direction = (-1);
R1_R = 10;
C_p_direction = 1;
C_n_direction = (-1);
C_C = 0.01;
R2_p_direction = 1;
R2_n_direction = (-1);
R2_R = 100;
L_p_direction = 1;
L_n_direction = (-1);
L_L = 0.1;
AC_p_direction = (-1);
AC_n_direction = 1;
AC_VA = 220;
AC_f = 50;
AC_PI = 3.141592;
G_p_direction = 1;
C_n_v = 0;
AC_n_v = 0;
L_n_v = 0;
G_p_v = 0;

OutputSection

DynamicsSection
AC_v = 220*sin(314.1592*Time);
AC_p_v = AC_v;
R1_n_v = C_v;
R1_v = AC_p_v-R1_n_v;
R1_p_i = 0.1*R1_v;
R1_n_i =  -R1_p_i;
C_p_i =  -R1_n_i;
der_C_v = 100.0*C_p_i;
R2_n_i =  -L_i;
R2_p_i =  -R2_n_i;
R2_v = 100*R2_p_i;
R2_n_v = AC_p_v-R2_v;
L_v = R2_n_v;
der_L_i = 10.0*L_v;

AcceptedSection
C_n_i =  -C_p_i;
L_n_i =  -L_i;
AC_p_i =  -(R1_p_i+R2_p_i);
AC_n_i =  -AC_p_i;
G_p_i =  -(L_n_i+C_n_i+AC_n_i);

DefaultSection
EndTranslatedEquations

#include <dsblock6.c>
```

```
DeclareVariable("R1_p_direction")
DeclareConstant(1)
DeclareVariable("R1_n_direction")
DeclareConstant((-1))
DeclareVariable("R1_R")
DeclareConstant(10)
DeclareVariable("C_p_direction")
DeclareConstant(1)
DeclareVariable("C_n_direction")
DeclareConstant((-1))
DeclareVariable("C_C")
DeclareConstant(0.01)
DeclareVariable("R2_p_direction")
DeclareConstant(1)
DeclareVariable("R2_n_direction")
DeclareConstant((-1))
DeclareVariable("R2_R")
DeclareConstant(100)
DeclareVariable("L_p_direction")
DeclareConstant(1)
DeclareVariable("L_n_direction")
DeclareConstant((-1))
DeclareVariable("L_L")
DeclareConstant(0.1)
DeclareVariable("AC_p_direction")
DeclareConstant((-1))
DeclareVariable("AC_n_direction")
DeclareConstant(1)
DeclareVariable("AC_VA")
DeclareConstant(220)
DeclareVariable("AC_f")
DeclareConstant(50)
DeclareVariable("AC_PI")
DeclareConstant(3.141592)
DeclareVariable("G_p_direction")
DeclareConstant(1)
DeclareVariable("C_n_v")
DeclareConstant(0)
DeclareVariable("AC_n_v")
DeclareConstant(0)
DeclareVariable("L_n_v")
DeclareConstant(0)
DeclareVariable("G_p_v")
DeclareConstant(0)
DeclareVariable("AC_p_v")
DeclareVariable("R1_p_i")
DeclareVariable("R1_n_v")
DeclareVariable("R1_n_i")
DeclareVariable("R1_v")
DeclareVariable("C_p_i")
DeclareVariable("C_n_i")
DeclareVariable("R2_p_i")
DeclareVariable("R2_n_v")
DeclareVariable("R2_n_i")
DeclareVariable("R2_v")
DeclareVariable("L_n_i")
DeclareVariable("L_v")
DeclareVariable("AC_p_i")
DeclareVariable("AC_n_i")
DeclareVariable("AC_v")
DeclareVariable("G_p_i")
DeclareState("C_v", "der_C_v", 0, 0.0)
DeclareState("L_i", "der_L_i", 1, 0.0)
DeclareOutput("CPUClk")

DeclareAlias("R1_p_v", " ", "AC_p_v", 1)
DeclareAlias("R2_p_v", " ", "AC_p_v", 1)
DeclareAlias("R1_i", " ", "R1_p_i", 1)
DeclareAlias("C_p_v", " ", "R1_n_v", 1)
DeclareAlias("C_i", " ", "C_p_i", 1)
DeclareAlias("R2_i", " ", "R2_p_i", 1)
DeclareAlias("L_p_v", " ", "R2_n_v", 1)
DeclareAlias("L_p_i", " ", "L_i", 1)
DeclareAlias("AC_i", " ", "AC_p_i", 1)
#define NX_    2
#define NX2_   0
#define NU_    0
#define NY_    1
#define NW_    39
#define NP_    0
#define NRel_  0
#define NCons_ 0
#define NA_    9

#include <dsblock5.c>
```

The initial values can be taken from the model definition. If necessary, the user specifies the parameter values. Numeric solvers for differential equations (such as LSODE, part of ODEPACK [8]) give the user possibility to ask about the value of specific variable in a specific time moment. 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.

The command `SimulateModel` will compile and link the C code file and execute the resulting binary file. The parameter values are taken form the default settings given in the Modelica model and in the defaults of the simulation engine. The parameter values can be changed between the simulation runs.

```
SimulateModel["Circuit"]
```

We can check that the simulation data in produced.

```
FileNames["*.mat"]
```

```
{DSRES.MAT}
```

The simulation data is load back into *Mathematica*.

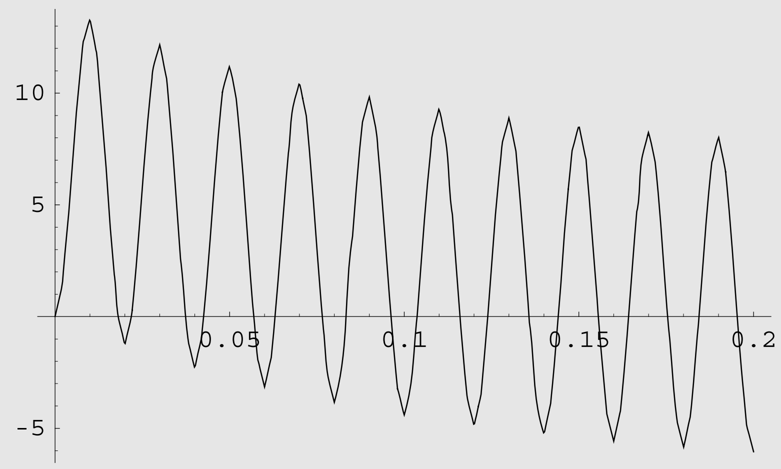
```
(simdata = GetSimulationData["dsres.mat"]) // Short
```

```
{{CPUClk → InterpolatingFunction[{{0., 1.}}, <>],
      ≪49≫, AC.i → InterpolatingFunction[{{0., 1.}}, <>]}}
```

The simulation data is automatically converted to rules of InterpolatingFunctions, in the same format as NDSolve produce.

A simple plot of one of the variables in the simulation data in the time interval [0, 0.2].

```
Plot[C.p.v[t] /. simdata, {t, 0., 0.2}];
```



3.4 Graphical Model Editors and Visualization

3.4.1 Model Editor

A 2D graphical model editor can be used to define a model by drawing and editing an object diagram very similar to the circuit diagram shown in Figure 1. Such a model editor is developed for MathModelica. The user can place icons that represent the components and connect those components by drawing lines between their iconic representations. For clarity, attributes of the MathModelica definition for the graphical layout of the composition diagram (here: an electric circuit diagram) are not shown in examples. These attributes are usually contained in a MathModelica model as annotations (which are largely ignored by the MathModelica translator and used by graphical tools).

3.4.2 Dynamic Simulation Visualization

Annotations in the MathModelica language can also be used to store the 3D graphical representations of the physical objects that are modeled, e.g., when modeling mechanical systems the graphical models are stored together with the dynamical models. This can be used to do a physical visualization using the simulation data to steer the animation [4] .

3.5 Computational Steering

The set of simulation commands provided by the MathModelica environment can be used to as a scripting language together with the *Mathematica* language. Except from controlling the simulation the commands provides the features:

- Setting up schedules for simulation runs and generating simulation reports using *Mathematica* notebooks.
- Starting simulation runs on remote machines.
- Management of parallel simulations.
- Management for simulation results and model libraries.

The major advantage with the notebook concept is that the same interface can be used for both storing and document model, and for simulation scripts and logging the simulation result.

3.6 Codegeneration and Interfacing External Code

The *Mathematica* application MathCode C++ [10] generated C++ code from *Mathematica* code. This is an essential component in the MathModelica environment since it provides tools for having general function call in the simulation models. Any MathCode C++ supported function or algorithm in *Mathematica* can be used seamlessly in the MathModelica models. The C++ generated code from MathCode C++ is them automatically linked to the simulation engine.

MathCode C++ also provides tools for interfacing external (object code) in Fortran, C, and C++ into C++ code produced by MathCode C++. In this way external code implementing models (like old Fortran model libraries) can be integrated into the MathModelica models.

4 Conclusions

The MathModelica is extension of the Modelica language targeted for work within the Mathematica environment. This language is object-oriented (the programs consist of collections of classes). The language is equation based: instead of traditional functions and procedures we use non-causal equations which specify algebraic and differential relations between numerical variables. Input-output causality is not specified, and therefore these equations can be used in multiple ways.

The environment integrates most activities needed in simulation design and use: documentation, modeling (coding), symbolic processing and transformation of formulas, input and output data visualization. This advanced programming environment can be applied in various simulation applications.

Acknowledgement

The authors would like to thank the other members of the Modelica Design Group for their contribution of the Modelica design.

Johan Gunnarsson would like to thank Kestrel Institute for an inspiring environment during the development of MathModelica.

This work was supported by the Swedish National Board for Industrial and Technical Development (NUTEK), which is gratefully acknowledged.

5 References

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

2. Aho A., Sethi R., and Ullman J., *Compilers – Principles Techniques and Tools*, Addison – Wesley, 1986

3. Elmqvist H. and Mattsson S. E., *An Introduction to the Physical Modeling Language Modelica*, ESS'97 European Simulation Symposium, Passau, Germany, October 19-22, 1997

4. Engelson V., Fritzson P., and Fritzson D., *Generating Efficient 3D Graphics Animation Code with OpenGL from Object Oriented Models in Mathematica*, In Proceedings of the Second International *Mathematica* Symposium, Rovaniemi, Finland, July, 1997.

5. Fritzson P., *Static and Strong Typing for Extended Mathematica*, In Proceedings of the Second International *Mathematica* Symposium, Rovaniemi, Finland, July, 1997.

6. Fritzson P. and Engelson V., *Modelica – A Unified Object-Oriented Language for System Modeling and Simulation*, in Proceedings of ECOOP-98, Brussels, July 1998.

7. Fritzson P., Viklund L., Fritzson D., and Herber J., *High-level Mathematica Modeling and Programming*, IEEE Software, 12:3, July 1995.

8. Hindmarsh A.C., *ODEPACK, A Systematized Collection of ODE Solvers*, Scientific Computing, R.S. Stepleman et al. (eds.), North-Holland, Amsterdam, 1983

9. Kågedal D. and Fritzson P., *Generating a Modelica Compiler from Natural Semantic Specification*, In Proceedings of Summer Computer Simulation Conference -98 , Reno, Nevada, USA, July 19-22, 1998

10. MathCode C++ – A C++ Code Generator for *Mathematica*, MathCore AB, http://www.mathcore.com

11. Modelica, *Language Design for Multi-Domain Modeling*, Modelica Design Group, http://www.modelica.org.

6 Init