MathModelica
An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming

A short version of this paper in Proceedings of the 2:nd International Modelica Conference, Munich, March 18-19, 2002, available at www.Modelica.org; This version at www.ida.liu.se/~pelab/modelica

Peter Fritzson[1], Johan Gunnarsson[2], Mats Jirstrand[2]
1) PELAB, Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, SE-581 83, Linköping, Sweden
petfr@ida.liu.se
2) MathCore AB, Wallenbergs gata 4, SE-583 35 Linköping, Sweden
{johan,mats}@mathcore.se

Abstract

MathModelica is an integrated interactive development environment for advanced system modeling and simulation. The environment integrates Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of program code, test cases, graphics, documentation, mathematical type setting, and symbolic formula manipulation provided via Mathematica. The user interface consists of a graphical Model Editor and Notebooks. The Model Editor is a graphical user interface in which models can be assembled using components from a number of standard libraries representing different physical domains or disciplines, such as electrical, mechanics, block-diagram and multi-body systems. Notebooks are interactive documents that combine technical computations with text, graphics, tables, code, and other elements. The accessible MathModelica internal form allows the user to extend the system with new functionality, as well as performing queries on the model representation and write scripts for automatic model generation. Furthermore, extensibility of syntax and semantics provides additional flexibility in adapting to unforeseen user needs.

1 Background

Traditionally, simulation and accompanying activities [Fritzson-92a] have been expressed using heterogeneous media and tools, with a mixture of manual and computer-supported activities:

- 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 problems in simulation development environments is that gathering and maintaining correct versions of all these components from various files and formats is difficult and error-prone.

Our vision of a solution to this set of problems is to provide integrated computer-supported modeling and simulation environments that enable the user to work effectively and flexibly with simulations. Users would then be able to 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.

1.1 Integrated Interactive Programming Environments

An integrated interactive modeling and simulation environment is a special case of programming environments with applications in modeling and simulation. Thus, it should fulfill the requirements both from general integrated environments and from the application area of modeling and simulation mentioned in the previous section.

The main idea of an integrated programming environment in general is that a number of

programming support functions should be available within the same tool in a well-integrated way. This means that the functions should operate on the same data and program representations, exchange information when necessary, resulting in an environment that is both powerful and easy to use. An environment is interactive and incremental if it gives quick feedback, e.g. without recomputing everything from scratch, and maintains a dialogue with the user, including preserving the state of previous interactions with the user. Interactive environments are typically both more productive and more fun to use.

There are many things that one wants a programming environment to do for the programmer, particularly if it is interactive. What functionality should be included? Comprehensive software development environments are expected to provide support for the major development phases, such as:

- requirements analysis,
- design,
- implementation,
- maintenance.

A programming environment can be somewhat more restrictive and need not necessarily support early phases such as requirements analysis, but it is an advantage if such facilities are also included. The main point is to provide as much computer support as possible for different aspects of software development, to free the developer from mundane tasks so that more time and effort can be spent on the essential issues. The following is a partial list of integrated programming environment facilities, some of which are already mentioned in [Sandewall-78], that should be provided for the programmer:

- Administration and configuration management of program modules and classes, and different versions of these.
- Administration and maintenance of test examples and their correct results.
- Administration and maintenance of formal or informal documentation of program parts, and automatic generation of documentation from programs.
- Support for a given programming methodology, e.g. top-down or bottom-up. For example, if a top-down approach should be encouraged, it is natural for the interactive environment to maintain successive composition steps and mutual references between those.
- Support for the interactive session. For example, previous interactions should be saved in an appropriate way so that the user can refer to previous commands or results, go back and edit those, and possibly re-execute.
- Enhanced editing support, performed by an editor that knows about the syntactic structure of the language. It is an advantage if the system allows editing of the program in different views. For example, editing of the overall system structure can be done in the graphical view, whereas editing of detailed properties can be done in the textual view.
- Cross-referencing and query facilities, to help the user understand interdependences between parts of large systems.
- Flexibility and extensibility, e.g. mechanisms to extend the syntax and semantics of the programming language representation and the functionality built into the environment.
- Accessible internal representation of programs. This is often a prerequisite to the extensibility requirement. An accessible internal representation means that there is a well-defined representation of programs that are represented in data structures of the programming language itself, so that user-written programs may inspect the structure and generate new programs. This property is also known as the principle of program-data equivalence.

1.2 Vision of Integrated Interactive Environment for Modeling and Simulation.

Our vision for the *MathModelica* integrated interactive environment is to fulfill essentially all the requirements for general integrated interactive environments combined with the specific needs for modeling and simulation environments, e.g.:

- Specification of requirements, expressed as documentation and/or mathematics;
- Design of the mathematical model;
- symbolic transformations of the mathematical model;
- A uniform general language for model design, mathematics, and transformations;
- Automatic generation of efficient simulation code;
- Execution of simulations;
- Evaluation and documentation of numerical experiments;
- Graphical presentation.

The design and vision of MathModelica is to a large extent based on our earlier experience in research and development of integrated incremental programming environments, e.g. the DICE system [Fritzson-83] and the ObjectMath environment [Fritzson-92b,Fritzson-95], and many years of intensive use of advanced integrated interactive environments such as the InterLisp system [Sandewall-78], [Teitelman-69,Teitelman-74], and *Mathematica* [Wolfram-88,Wolfram-97]. The InterLisp system was actually one of the first really powerful integrated environments, and still beats most current programming environments in terms of powerful facilities available to the programmer.

It was also the first environment that used graphical window systems in an effective way [Teitelman77], e.g. before the Smalltalk environment [Goldberg 89] and the Macintosh window system appeared.

Mathematica is a more recently developed integrated interactive programming environment with many similarities to InterLisp, containing comprehensive programming and documentation facilities, accessible intermediate representation with program-data equivalence, graphics, and support for mathematics and computer algebra. *Mathematica* is more developed than InterLisp in several areas, e.g. syntax, documentation, and pattern-matching, but less developed in programming support facilities.

1.3 Mathematica and Modelica

It turns out that the Mathematica is an integrated programming environment that fulfils many of our requirements. However, it lacks object-oriented modeling and structuring facilities as well as generation of efficient simulation code needed for effective modeling and simulation of large systems. These modeling and simulation facilities are provided by the object-oriented modeling language Modelica [MA-97a, MA-97b, MA-02a, MA-02b], [Tiller-01], [Elmqvist-99], [Fritzson-98].

Our solution to the problem of a comprehensive modeling and simulation environment is to combine Mathematica and Modelica into an integrated interactive environment called MathModelica. This environment provides an internal representation of Modelica that builds on and extends the standard Mathematica representation, which makes it well integrated with the rest of the Mathematica system.

The realization of the general goal of a *uniform general language* for model design, mathematics, and symbolic transformations is based on an integration of the two languages Mathematica and Modelica into an even more powerful language called the MathModelica language. This language is Modelica in Mathematica syntax, extended with a subset of Mathematica. Only the Modelica subset of MathModelica can be used for object-oriented modeling and simulation, whereas the Mathematica part of the language can be used for interactive scripting.

Mathematica provides representation of mathematics and facilities for programming symbolic transformations, whereas Modelica provides language elements and structuring facilities for object-oriented component based modeling, including a strong type system for efficient code and engineering safety. However, this language integration is not yet realized to its full potential in the current release of MathModelica, even though the current level of integration provides many impressive capabilities. Future improvements of the MathModelica language integration might include making the object-oriented facilities of Modelica available also for ordinary Mathematica programming, as well as making some of the Mathematica language constructs available also within code for simulation models.

The current MathModelica system builds on experience from the design of the ObjectMath [Fritzson-92b,Fritzson-95] modeling language and environment, early prototypes [Fritzson-98b], [Jirstrand-99], as well as on results from object-oriented modeling languages and systems such as Dymola [Elmqvist-78,Elmqvist-96] and Omola [Mattsson-93], [Andersson-94], which together with ObjectMath and a few other object-oriented modeling languages, e.g. [Sahlin-96], [Breunese-97], [Ernst-97], [Piela-91], [Oh-96], have provided the basis for the design of Modelica.

ObjectMath was originally designed as an object-oriented extension of Mathematica augmented with efficient code generation and a graphic class browser. The ObjectMath effort was initiated 1989 and concluded in the fall of 1996 when the Modelica Design Group was started, later renamed to Modelica Association. At that time, instead of developing a fifth version of ObjectMath, we decided to join forces with the originators of a number of other object-oriented mathematical modeling languages in creating the Modelica language, with the ambition of eventually making it an international standard. In many ways the MathModelica product can be seen as a logical successor to the ObjectMath research prototype.

2 The Modelica Language

The details of the MathModelica language as tentatively defined in the previous section will be described using an example of an electric circuit model that is given in the form of MathModelica expressions in this section. The subset of the MathModelica language described in this section is the part that corresponds to Modelica and can be used in the simulation models, not in general Mathematica programming. Note that here we only describe modeling in terms of textually programming MathModelica. The MathModelica environment also includes a graphical modeling tool and language based on MathModelica language, which is briefly described in Section 3 in this article. Visual constructs in the graphical environment have a one-to-one correspondence with constructs in the textual MathModelica language, or classes defined in MathModelica.

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 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 can be inherited from other classes. Equations can also be specified by the `connect` statement. The statement `connect(v1,v2);` expresses coupling between the variables `v1` and `v2`. These variables are instances of connector classes and are attributes of the connected object. 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. Powerful parametrization capabilities (which are advanced features of Modelica) are discussed in Section 2.10.

2.1 Connection Diagrams

As an introduction to Modelica we will present a model of a simple electrical circuit shown in Figure 1.

The circuit can be broken down into a set of standard connected electrical components. We have a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of such standard components are available in Modelica class libraries.

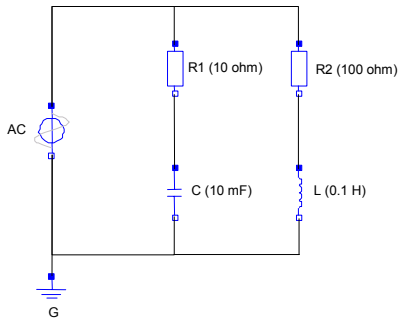


Figure 1. Connection diagram of the electric circuit.

A declaration like the one below specifies `R1` to be an object or instance of the 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:

```
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);
   connect(R1.n, C.p);
   connect(C.n, AC.n);
   connect(R1.p, R2.p);
   connect(R2.n, L.p);
   connect(L.n, C.n);
   connect(AC.n, G.p);
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 as 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 2.3.

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 Type Definitions

The Modelica language is a strongly typed language with both predefined and user-defined types. The built-in "primitive" data types support floating-point, integer, boolean, and string values. These primitive types contain data that Modelica understands directly. The type of every variable must be stated explicitly. The primitive data types of Modelica are listed in Table 1.

Type	Description
`Boolean`	either true or false
`Integer`	corresponding to the `C int` data type, usually 32-bit two's complement
`Real`	corresponding to the `C double` data type, usually 64-bit floating-point
`String`	string of 8-bit characters

Table 1. Predefined data types in Modelica

It is possible to define new user-defined types:

```
type name = type "optionaltextcomment";
```

An example is to define a temperature measured in Kelvin, K, which is of type `Real` with the minimum value zero;

```
type Temperature =
      Real(Unit="K", Min=0)
"temperature measured in Kelvin";
```

Below the user-defined types of `Voltage` and `Current` are defined.

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

This defines the symbol `Voltage` to be a specialization of the type `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"`. A corresponding definition is made for `Current` below.

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

In MathModelica, the basic structuring element is a class. The general keyword `class` is used for declaring classes. There are also 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` still keeps the model semantically equivalent to the original, because the restrictions imposed by such a specialized class are already fulfilled by a valid model. Other specific class categories are `model`, `record`, and `block`. Moreover, functions and packages are regarded as special kinds of restricted and enhanced classes, denoted by the keyword `function` for functions, and `package` for packages.

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 basic properties of a class, such as syntax and semantics 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 have been expressed using the keyword `type` which is equivalent to `class`, but limits the defined type to be an extension of a built-in type, a record type or an array type. Note however that the restricted classes that are packages and functions have some special properties that are not present in general classes.

2.3 Connector Classes

When developing models and model libraries for a new application domain, it is good to start by defining a set of connector classes which are used as templates for interfaces between model instances. A common set of connector classes used by all models in the library supports compatibility and connectability of the component models.

2.3.1 Pin

The following is a definition of an electrical connector class `Pin`, used as an interface class for electrical components. The voltage, *v*, is defined as an effort variable, and the current, *i*, as a flow variable. This implies that voltages will be set equal when two or more components are connected together, i.e. $v_1 = v_2 = \ldots = v_n$, and currents are summed to zero at the connection point, i.e. $i_1 + i_2 + \ldots + i_n = 0$.

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

Connection statements are used to connect instances of connector classes. A connection statement `connect(Pin1,Pin2);` with the instances `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, i.e. $v_1 = v_2 = \ldots = v_n$. The second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a connection point (assuming positive value while flowing into the component), i.e. $i_1 + i_2 + \ldots + i_n = 0$. 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.

2.4 Partial (Virtual) Classes

A useful strategy for reuse in object-oriented modeling is to try to capture common properties in superclasses which can be inherited by more specialized classes. For example, a common property of many electrical components such as resistors, capacitors, inductors, and voltage sources, etc., is that they have *two pins*. This means that it is useful to define a generic "template" class, or superclass, that captures the properties of all electric components with two pins. This class is *partial*, i.e. *virtual* in standard object-oriented terminology, since it does not specify all properties needed to instantiate the class.

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

The class (or model) `TwoPin` 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`. This can be summarized in the following points:

- Classes that inherit `TwoPin` have at least two pins, `p` and `n`.
- The voltage, `v`, is calculated as the potential at pin `p` minus the potential at pin `n`, i.e. `v = p.v - n.v;`.

- The current at the negative pin of a component equals the current at the positive pin, only with different sign, i.e. `p.i + n.i=0`;
- The current, `i`, through a component is defined as the current at the positive pin, i.e. `i = p.i;`.

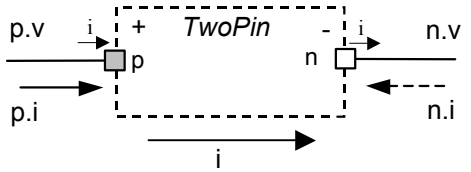


Figure 2. Structure of a TwoPin class with two pins

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.

The 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 dialogues and forms presenting details about class definitions.

2.5 Equations and Acausal Modeling

Acausal 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 equation-based models is unspecified and fixed only when the equation systems are solved. This is called acausal modeling.

The main advantage with acausal 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 specifying which variables are needed as outputs and which are external inputs to the simulated system.

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

Consider for example the constitutive equation from the `Resistor` class below:

```
R*i = v
```

This equation 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 consider the following equation from the class `TwoPin.`

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

This equation gives rise to one of the three assignment statements shown below,when the equation system is to be solved, depending on the data flow context where the equation appears:

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

2.6 Inheritance, Parameters and Constants

We will use the `Resistor` example below to explain *inheritance*, *parameters* and *constants*.

The `Resistor` inherits `TwoPin` using the `extends` statement. A model parameter, `R`, is defined for the resistance, and is used to state the constitutive equation for an ideal resistor, namely *Ohm's Law*: `v=R*i.` We add a definition of a parameter for the resistance and Ohm's law to define the behavior of the `Resistor` class in addition to what is inherited from `TwoPin`:

```
model Resistor
   "Ideal electrical resitor"
   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 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. There are also Modelica constants that never change and can be substituted inline, which are specified by the keyword `constant`. Additional examples of constants and parameters, whose default values are defined via a so-called declaration equations that appear in the declarations:

```
Constant  Real c0 = 2.99792458E8;
Constant  String  redcolor = "red";
Constant  Integer  population = 1234;
Parameter  Real  speed = 25;
```

There are several predefined constants in the `Modelica.Constants` package, e.g. Planck, Boltzmann, and molar gas constants. In contrast to constants, parameters can be defined via input to a model, thus a parameter can be declared *without* a declaration equation. For example:

```
parameter Real  mass,velocity;
```

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

Just like in C++, the parent class cannot be replaced in a subclass. In Modelica similar restrictions also apply to equations and connections.

In C++ and Java a virtual function can be replaced/specialized by a function with the same name in the child class. In Modelica 2.0 equations in `equation` section cannot be directly named (but indirectly using a local class for grouping a set of equations) and therefore we cannot directly 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.

2.7 Time and Model Dynamics

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

The classes defined below for electric voltage sources, capacitors, and inductors, have all dynamic time dependent behavior, and can also reuse the `TwoPin` superclass. In the differential equations in the classes `Capacitor` and `Inductor`, `v'` and `i'` denote the time derivatives of `v` and `i` respectively.

During system simulation the variables i and v evolve as functions of time. The differential equations solver will compute the values of $i(t)$ and $v(t)$ (t is time) so that $Cv'(t) = i(t)$ for all values of t.

2.7.1 VsourceAC

A class for the voltage source can be defined as follows. This `VsourceAC` class inherits `TwoPin` since it is an electric component with two connector attributes, `n` and `p`. A parameter, `VA`, is defined for the amplitude, and a parameter `f` for the frequency. Both are given default values, 220 V, and 50 Hz respectively, that however can easily be modified by the user when running simulations, e.g. through the graphical user interface. A constant `PI` is also declared using the value for π defined in the Modelica Standard Library, just to demonstrate the declaration of a constant.. The input voltage `v` is defined by $v = VA * \sin(2 * \pi * f * time)$. Note that `time` is a builtin Modelica primitive.

```
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;
```

2.7.2 Capacitor

The `Capacitor` inherits `TwoPin` using `extends`. A parameter, `C`, is defined for the capacitance, and is used to state the constitutive equation for an ideal capacitor, namely, $\frac{dv}{dt} = \frac{i}{C}$

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

2.7.3 Inductor

The `Inductor` inherits `TwoPin` using `extends`. A parameter, `L`, is defined for the inductance, and is used to state the constitutive equation for an ideal inductor, namely, $L * \frac{di}{dt} = v$

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

2.7.4 Ground

Finally, we define a `Ground` class which in the circuit model is instantiated as a ground point that serves as a reference value for the voltage levels.

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

2.8 Definition and Simulation of the Complete Circuit Model

After all the component classes have been defined, it is possible to construct a circuit. First the components are *declared*, then the parameter values are *set*, and finally the components are *connected* together using `connect`.

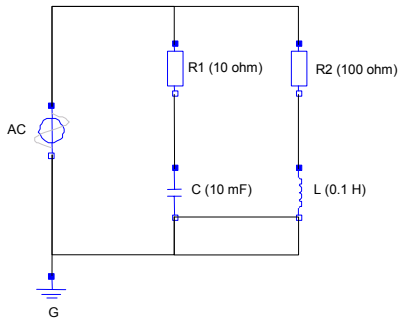


Figure 3. Diagram of the electric circuit, once again.

We show the `Circuit` model once more:

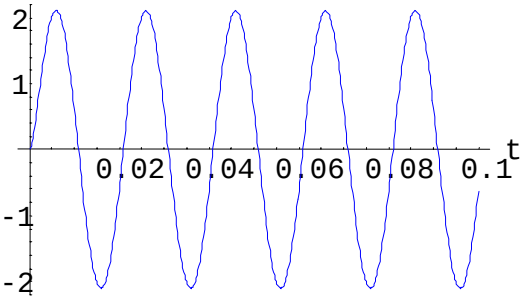
```
model Circuit
   Resistor R1(R = 10);
   Capacitor C(C = 0.01);
   Resistor R2(R = 100);
   Inductor L(L = 0.1);
   VsourceAC AC;
   Ground G;
equation
   connect(AC.p, R1.p);
   connect(R1.n, C.p);
   connect(C.n, AC.n);
   connect(R1.p, R2.p);
   connect(R2.n, L.p);
   connect(L.n, C.n);
   connect(AC.n, G.p);
end Circuit;
```

We simulate the model with the default initial values and parameter settings in the range $0 \le t \le 0.1$. The status bar in the lower left corner of the notebook shows the status of the simulation. Since this is the first time we simulate the circuit model `Simulate` will generate C-code and compile the code before the simulation.

```
Simulate[Circuit, {t, 0, 0.1}];
```

Let us plot the current in the inductor for the first 0.1 second.

```
PlotSimulation[{L.i[t]}, {t, 0, 0.1}];
```

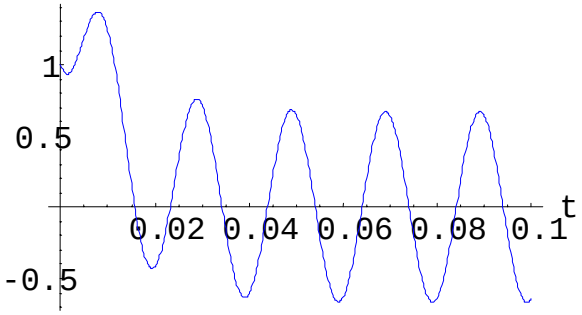


Note that the current starts at 0 Ampere, which is the default initial value. Let us change the initial values for the inductor current and the inductance using the options `InitialValues` and `ParameterValues` respectively. This time `Simulate` will use the compiled code from the previous simulation as we have only changed initial value and parameter value, and not the structure of the problem.

```
Simulate[Circuit, {t, 0, 0.1},
  InitialValues -> {L.i == 1},
  ParameterValues → {L.L == 1}];
```

A plot shows the result. Note the difference in initial current and also the difference in amplitude due to the changed inductance.

```
PlotSimulation[{L.i[t]}, {t, 0, 0.1}];
```



2.9 The Modelica Notion of Subtypes

The notion of subtyping in Modelica is influenced by a type theory of Abadi and Cardelli [Abadi-Cardelli-96]. The notion of inheritance in Modelica is independent from the notion of subtyping. According to the definition, a class A is a subtype of a class B if and only if the class A contains all the public variables declared in the class B, and the types of these variables are subtypes of types of corresponding variables in B. The main benefit of this definition is additional flexibility in the definition and usage of types. For instance, the class `TempResistor` is a subtype of `Resistor`, without being a subclass of `Resistor`.

```
model TempResistor
  extends TwoPin;
  parameter Real R
     "Resistance at reference Temp.";
  parameter Real RT=0
     "Temp. dependent Resistance.";
  parameter Real Tref=20
     "Reference temperature.";
  Real Temp=20
     "Actual temperature";
equation
    v = i*(R + RT*(Temp-Tref));
end TempResistor;
```

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

Note that `TempResistor` does not inherit the `Resistor` class. There are different definition for

evaluation of `v`. If equations are inherited from `Resistor` then the set of equations will become inconsistent in `TempResistor`, since there would be two definitions of `v`. For example, the specialized equation below from `TempResistor`:

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

and the general equation from class `Resistor`:

```
v=R*i
```

are incompatible. Modelica currently does not support explicitly named equations and replacement of equations, except for the cases when the equations are collected into a local class, or a declaration equation is present in a variable declaration.

2.10 Class Parametrization

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

- *Inheritance*. This is essentially "copying" class definitions and adding additional elements (variables, equations and functions) to the inheriting class.
- *Class parametrization* (also called generic classes or types). This mechanism replaces a generic type identifier in a whole class definition by an actual type.

In Modelica we can use redeclaration to control class parametrization. Assume that a library class is defined as follows:

```
model SimpleCircuit
   Resistor R1(R=100), R2(R=200),
            R3(R=300);
equation
   connect(R1.p, R2.p);
   connect(R1.p, R3.p);
end SimpleCircuit;
```

Assume also 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` for the previously mentioned temperature-dependent resistor model, `TempResistor`.

This can be accomplished by redeclaring `R1` and `R2` as in the following type definition which defines `RedefinedSimpleCircuit` to be a special variant of `SimpleCircuit`.

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

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 by a more specific *temperature dependent* model. Values of the additional parameters of `TempResistor` can also be added in the redeclaration:

```
redeclare TempResistor R1(RT=0.1,
                          Tref=20.0)
```

Replacing `Resistor` by `TempResistor` is a very strong modification. However, it should be noted that all equations that are defined in the previous `Circuit` example model are still valid.

2.11 Discrete and Hybrid Modeling

Macroscopic physical systems in general evolve continuously as a function of time, obeying the laws of physics. This includes the movements of parts in mechanical systems, current and voltage levels in electrical systems, chemical reactions, etc. Such systems are said to have continuous dynamics.

On the other hand, it is sometimes beneficial to make the approximation that certain system components display *discrete* behavior, i.e. changes of values of system variables may occur instantaneously and discontinuously. In the real physical system the change can be very fast, but not instantaneous. Examples are collisions in mechanical systems, e.g. a bouncing ball that almost instantaneously changes direction, switches in electrical circuits with quickly changing voltage levels, valves and pumps in chemical plants, etc. We talk about system components with discrete dynamics. The reason for making the discrete approximation is to simplify the mathematical model of the system, making the model more tractable and usually speeding up the simulation of the model several orders of magnitude.

Since the discrete approximation only can be applied to certain subsystems, we often arrive at system models consisting of interacting continuous and discrete components. Such a system is called a *hybrid system* and the associated modeling techniques *hybrid modeling*. The introduction of hybrid mathematical models creates new difficulties for their solution, but the disadvantages are far outweighed by the advantages.

Modelica provides two kinds of constructs for expressing hybrid models: *conditional expressions* or *conditional equations* to describe discontinuous and conditional models, and *when-equations* to express equations that are only valid at discontinuities, e.g. when certain conditions become true. For example, if-then-else conditional expressions allow modeling of phenomena with different expressions in different operating regions, as for the equation describing a limiter below.

```
y = if u > limit then limit else u;
```

A more complete example of a conditional model is the model of an ideal diode. The characteristic of a real physical diode is depicted in Figure 4, and the ideal diode characteristic in parameterized form is shown in Figure 5.

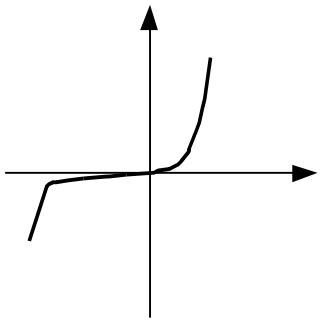


Figure 4. Real diode characteristic.

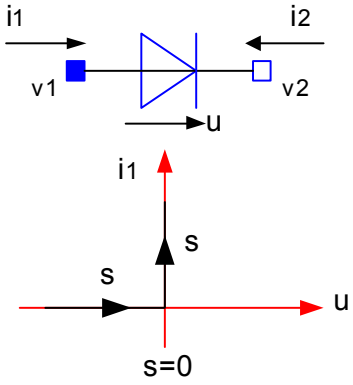


Figure 5. Ideal diode characteristic.

Since the voltage level of the ideal diode would go to infinity in an ordinary voltage-current diagram, a parameterized description is more appropriate, where both the voltage `u` and the current `i`, same as `i1`, are functions of the parameter `s`. When the diode is off no current flows and the voltage is negative, whereas when it is on there is no voltage drop over the diode and the current flows.

```
model Diode "Ideal diode"
   extends TwoPin;
   Real s;
   Boolean off;
equation
  off = s < 0;
  if off then v=s else v=0;
            // conditional equations
  i = if off then 0 else s;
           // conditional expression
end Diode;
```

`When`-equations have been introduced in Modelica to express instantaneous equations, i.e. equations that are valid only at certain points, e.g. at discontinuities, when specific conditions become `true`. The syntax of `when`-equations for the case of a vector of conditions is shown below. The equations in the `when`-equation are activated when at least one of the conditions become `true`. A single condition is also possible.

```
when {condition1, condition2, …} then
   <equations>...
end when;
```

A bouncing ball is a good example of a hybrid system for which the when-clause is appropriate when modeled. The motion of the ball is characterized by the variable `height` above the ground and the vertical `velocity`. The ball moves continuously between bounces, whereas discrete changes occur at bounce times, as depicted in Figure 6. When the ball bounces against the ground its velocity is reversed. An ideal ball would have an elasticity coefficient of 1 and would not lose any energy at a bounce. A more realistic ball, as the one modeled below, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.

```
model BouncingBall
    "Simple model of a bouncing ball"
   constant  Real g = 9.81
     "Gravity constant";
   parameter Real c = 0.9
     "Coefficient of restitution";
   parameter Real radius=0.1
     "Radius of the ball";
   Real height(start = 1)
     "Height of the ball center";
   Real velocity(start = 0)
     "Velocity of the ball";
equation
   der(height) = velocity;
   der(velocity) = -g;
   when height <= radius then
     reinit(velocity,-c*pre(velocity));
   end when;
end BouncingBall;
```

The bouncing ball model contains the two basic equations of motion relating height and velocity as well as the acceleration caused by the gravitational force. At the bounce instant the velocity is suddenly reversed and slightly decreased, i.e. `velocity`(after bounce) = `-c*velocity`(before bounce), which is accomplished by the special syntactic form of instantaneous equation: `reinit(velocity,-c*pre(velocity))`.

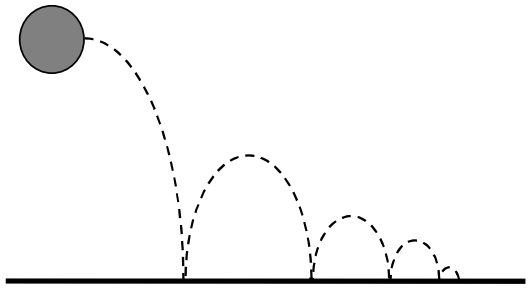


Figure 6. A bouncing ball.

Example simulations of the bouncing ball model are available in Section 4.

2.12 Discrete Events

In the previous section on hybrid modeling we briefly mentioned the notion of *discrete events*. But what is an *event*? Using every day language an event is simply *something that happens*. This is also true for events in the abstract mathematical sense. An event in the real world, e.g. a music performance, is always associated with a point in time. However, abstract mathematical events are not always associated with time but they are usually ordered, i.e. an event ordering is defined. By associating an event with a point in time, e.g. as in Figure 7 below, we will automatically obtain an ordering of events to form an event history. Since this is also the case for events in the real world we will in the following always associate a point in *time* to each event. However, such an ordering is *partial* since several events can occur at the same point in time. To achieve a total ordering we can use causal relationships between events, priorities of events, or if these are not enough simply pick an order based on some other event property.

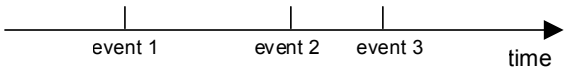


Figure 7. Events are ordered in time and form an event history.

The next question is whether the notion of event is a useful and desirable abstraction, i.e. do events fit into our overall goal of providing an object-oriented declarative formalism for modeling the world? There is no question that events actually exist, e.g. a cocktail party event, a car collision event, or a voltage transition event in an electrical circuit. A set of events without structure can be viewed as a rather low-level abstraction - an unstructured mass of small low-level items that just happen.

The trick to arrive at declarative models about *what is*, rather than imperative recipies of how things *are done*, is to focus on *relations* between events, and between events and other abstractions. Relations between events can be expressed using declarative formalisms such as equations. The object-oriented modeling machinery provided by Modelica can be used to bring a high-level model structure and grouping of state variables affected by events, relations between events, conditions for events, and behavior in the form of equations associated with events. This brings order into what otherwise could become a chaotic mess of low-level items.

Our abstract "mathematical" notion of event is an approximation compared to real events. For example, events in Modelica take *no time* - this is the most important abstraction of the *synchronous* principle to be described later. This abstraction is not completely correct with respect to our cocktail party event example since there is no question that a cocktail party actually takes some time. However, experience has shown that abstract events that take no time are more useful as a modeling primitive than events that have duration. Instead, our cocktail party should be described as a model class containing state variables such as the number of guests that are related by equations active at primitive events like opening the party, the arrival of a guest, ending the party, serving the drinks, etc.

To conclude, an event in Modelica is something that happens that has the following four properties:

- A *point* in time that is instantaneous, i.e. has zero duration.
- An *event condition* that switches from `false` to `true` for the event to happen.
- A set of *variables* that are associated with the event, i.e. are referenced or explicitly changed by equations associated with the event.
- Some *behavior* associated with the event, expressed as *conditional equations* that become active or are deactivated at the event. *Instantaneous equations* is a special case of conditional equations that are only active at events.

2.12.1 Discrete-time and Continuous-time Variables

The so called *discrete-time* variables in Modelica only change value at discrete points in time, i.e. at event instants, and keep their values constant between events. This is in contrast to *continuous-time* variables which may change value at any time, and usually evolve continuously over time. Figure 8 shows graphs of two variables, one continuous-time and one discrete-time.

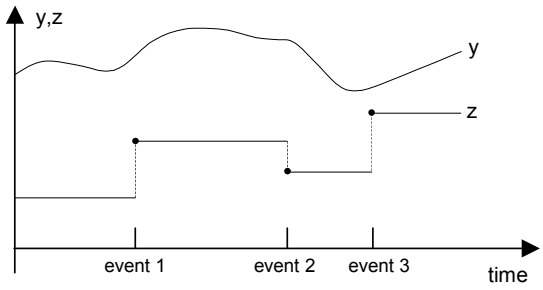


Figure 8. A discrete-time variable z changes value only at event instants, whereas continuous-time variables like y and z may change value both between and at events.

Note that discrete-time variables change their values at an event instant by solving the equations active at the event. The previous value of a variable, i.e. the value before the event, can be obtained via the `pre` function.

Variables in Modelica are discrete-time if they are declared using the `discrete` prefix, e.g. `discrete Real y`, or if they are of type `Boolean`, `Integer`, or `String`, or of types constructed from discrete types. A variable being on the left-hand side of an equation in a `when`-equation is also discrete-time. A `Real` variable *not* fulfilling the conditions for discrete-time is continuous-time. It is not possible to have continuous-time `Boolean`, `Integer`, or `String` variables.

3 The MathModelica Integrated Interactive Environment.

The MathModelica system consists of three major subsystems that are used during different phases of the modeling and simulation process, as depicted in Figure 9 below.

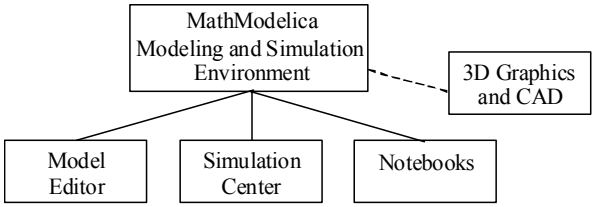


Figure 9. The MathModelica system architecture.

These subsystems are the following:

- The graphic *Model Editor* used for design of models from library components.
- The interactive *Notebook* facility, for literate programming, documentation, running simulations, scripting, graphics, and symbolic mathematics with Mathematica.
- The Simulation center, for specifying parameters, running simulations and plotting curves.

A menu palette enables the user to select whether to use the Notebook interface for editing and simulations, or the Model Editor combined with the Simulation Center graphical user interface.

Additionally, MathModelica is loosely coupled to two optional subsystems for 3D graphics visualization and automatic translation of CAD models to Modelica. In order to provide the best possible facilities available on the market for the user, MathModelica integrates and extends several professional software products that are included in the three subsystems. For example, the model editor is a customization and extension of the diagram and visualization tool Visio [Visio] from Microsoft, the simulation center includes simulation algorithms from Dynasim [Elmqvist-96], and the Notebook facility includes the technical computing system Mathematica [Wolfram-97] from Wolfram Research. Basing the Model Editor on Visio gives it properties such as power, flexibility, and customizability, but currently limits the system to MicroSoft Windows based platforms. However, a Model Editor with basic functionality for Unix platforms is under development.

A key aspect of MathModelica is that the modeling and simulation is done within an environment that also provides a variety of technical computations. This can be utilized both in a preprocessing stage in the development of models for subsystems as well as for postprocessing of simulation results such as signal processing and further analysis of simulated data.

3.1 Graphic Model Editor.

The MathModelica Model Editor is a graphical user interface for model diagram construction by "drag-and-drop" of model classes from the Modelica Standard Library or from user defined component libraries, visually represented as graphic icons in the editor. A screen shot of the Model Editor is shown in Figure 10. In the left part of the window three library packages have been opened, visually represented as overlapping windows containing graphic icons. The user can drag models from these windows (called stencils in Visio terminology) and drop them on the drawing area in the middle of the tool.

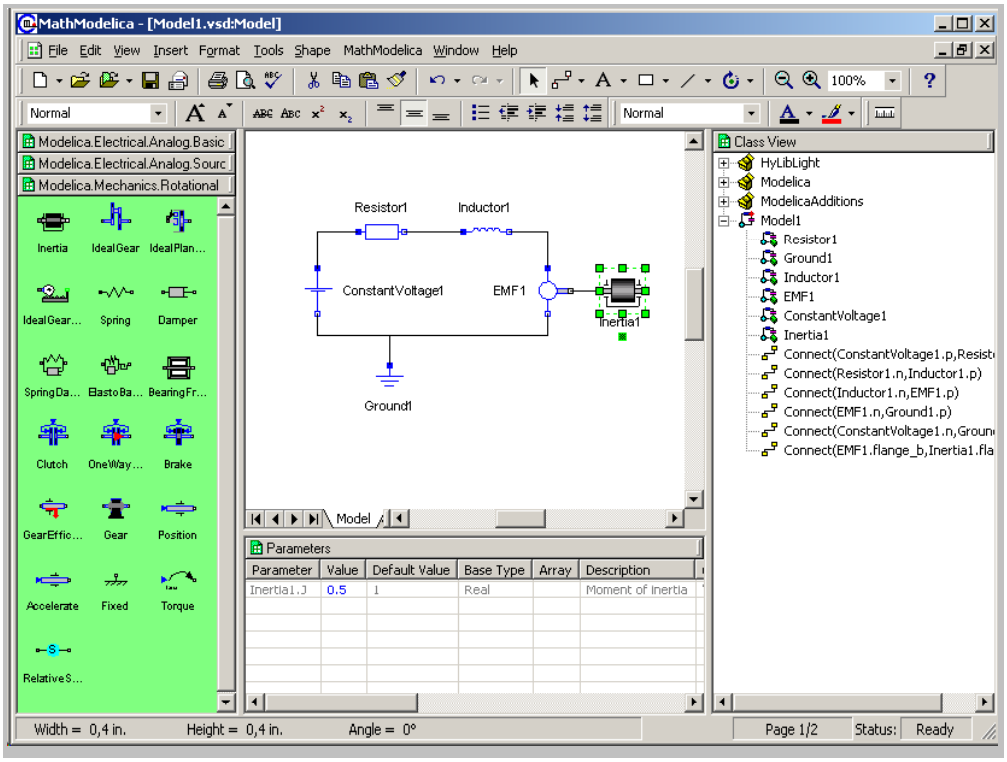


Figure 10. The Graphic Model Editor showing an electrical motor with the Inertia parameter J modified**.**

The Model Editor is an extension of the Microsoft Visio software for diagram design and schematics. This means that the user has access not only to a well developed and user friendly graph drawing application, but also to a vast array of professional design features to make graphical representations of developed models visually attractive. Since Modelica classes often represent physical objects it is of great value to have a sufficiently rich graphical description of these classes.

The Model Editor can be viewed as a user interface for graphical programming in Modelica. Its basic functionality consists of selection of components from libraries, connection of components in model diagrams, and entering parameter values for different components

For large and complex models it is important to be able to intuitively navigate quickly through component hierarchies. The Model Editor supports such navigation in several ways. A model diagram can be browsed and zoomed.

The Model Editor is well integrated with Notebooks. A model diagram stored in a notebook is a tree-structured graphical representation of the Modelica code of the model, which can be converted into textual form by a command.

3.2 Simulation Center.

The simulation center is a subsystem for running simulations, setting initial values and model parameters, plot results, etc. These facilities are accessible via a graphic user interface accessible through the simulation window, e.g. see Figure 11 below. However, remember that it is also possible to run simulations from the textual user interface available in the notebooks. The simulation window consists of five areas or subwindows with different functionality:

- The uppermost part of the simulation window is a control panel for starting and running simulations. It contains two fields for setting start and stop time for simulation, followed by `Build`, `Run Simulation`, `Plot`, and `Stop` buttons.
- The left subwindow in the middle section shows a tree-structure view of the model selected and compiled for simulation, including all its submodels and variables. Here, variables can be selected for plotting.
- The center subwindow is used for diagrams of plotted variables.
- The right subwindow in the middle section contains the legend for the plotted diagram, i.e. the names of the plotted variables.
- The subwindow at the bottom is divided into three sections: `Parameters`, `Variables`, and `Messages`, of which only one at a time is visible. The `Parameters` section, shown in Figure 11, allows changing parameter values, whereas the `Variables` section allows modifying intial (start) values, and the `Message` section to view possible messages from the simulation process.

If a model parameter or initial value has been changed, it is possible to rerun the simulation without rebuilding the executable code if the changed parameter does not influence the equation structure. Structure changing parameters are sometimes called *structural parameters*.

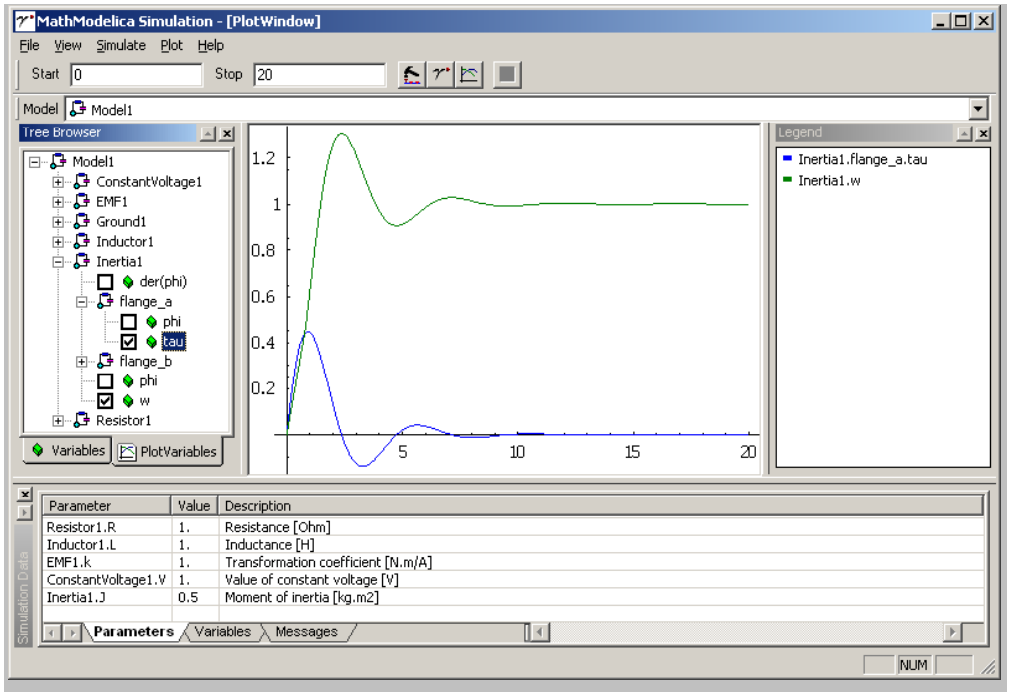


Figure 11. The Simulate window with plots of the signals `Inertia1.flange_a.tau` and `Inertia1.w` selected in the subwindow menus.

3.3 Interactive Notebooks with Literate Programming.

In addition to purely graphical programming of models using the Model Editor, MathModelica also provides a text based programming environment for building textual models using Modelica. This is done using Mathematica Notebooks, which are documents that may contain technical computations and text, as well as graphics. Hence, these documents are suitable to be used for simulation scripting, model documentation and storage, model analysis and control system design, etc. In fact, this article is written as such a notebook and in the live version the examples can be run interactively. A sample notebook is shown in Figure 12.

The MathModelica Notebook facility is actually an interactive WYSIWYG (What-You-See-Is-What-You-Get) realization of Literate Programming, a form of programming where programs are integrated with documentation in the same document, originally proposed in [Knuth-84]. A noninteractive prototype implementation of Literate Programming in combination with the document processing system LaTex has been realized [Knuth-94]. However, MathModelica is one of very few interactive WYSIWYG systems so far realized for Literate Programming, and to our knowledge the only one yet for Literate Programming in Modeling, which also might be called Literate Modeling.

Integrating Mathematica with MathModelica does not only give access to the Notebook interface but also to thousands of available functions and many application packages, as well as the ability of communicating with other programs and import and export of different data formats. These capabilities make MathModelica more of a complete workbench for the innovative engineer than just a modeling and simulation tool. Once a model has been developed there is often a need for further analysis such as linearization, sensitivity analysis, transfer function computations, control system design, parametric studies, Monte Carlo simulations, etc.

In fact, the combination of the ability of making user defined libraries of reusable components in Modelica and the Notebook concept of living technical documents provides an integrated approach to model and documentation management for the evolution of models of large systems

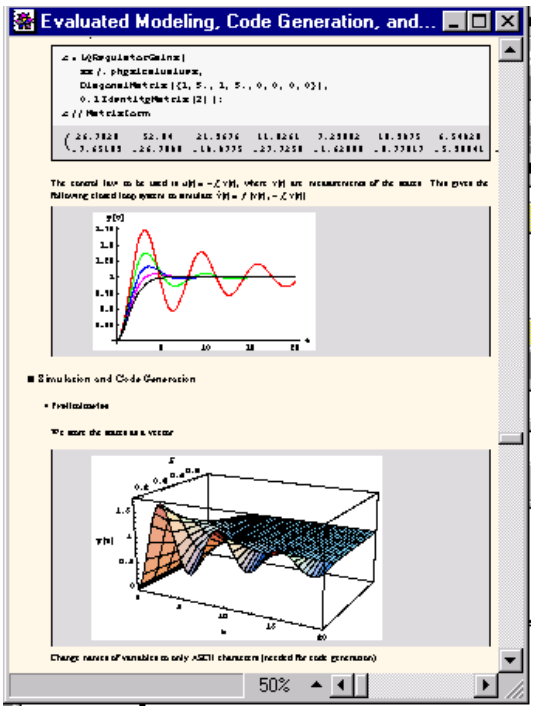


Figure 12. Example of MathModelica notebook.

3.3.1 Tree Structured Hierarchical Document Representation.

Traditional documents, e.g. books and reports, essentially always have a hierarchical structure. They are divided into sections, subsections, paragraphs, etc. Both the document itself and its sections usually have headings as labels for easier navigation. This kind of structure is also reflected in MathModelica notebooks. Every notebook corresponds to one document (one file) and contains a tree structure of cells. A cell can have different kinds of contents, and can even contain other cells. The notebook hierarchy of cells thus reflects the hierarchy of sections and subsections in a traditional document.

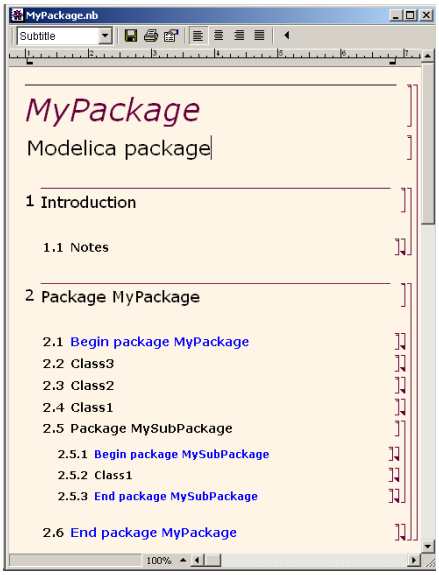


Figure 13. The package `Mypackage` in a notebook

In the MathModelica system, Modelica packages including documentation and test cases are primarily stored as notebooks, e.g. as in Figure 12. Those cells that contain Modelica model classes intended to be used from other models, e.g. library components or certain application models, should be marked as exports cells. This means that when the notebook is saved, such cells are automatically exported into a Modelica package file in the standard Modelica textual representation (`.mo` file) that can be processed by any Modelica compiler and imported into other models. For example, when saving the notebook `MyPackage.nb` of Figure 13, a file `MyPackage.mo` would be created with the following contents:

```
package MyPackage
  model class3
  ...
  end class3;
  model class2 ...
  model class1 ...
  package MySubPackage
    model class1
    ...
    end class1;
  end MySubPackage;
end MyPackage;
```

3.3.2 Program Cells, Documentation Cells, and Graphic Cells.

A notebook 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 for example be one of the following forms:

- Model classes and parts of models, i.e. formal descriptions that can be used for verification, compilation and execution of simulation models.
- *Mathematica*l formulas in the traditional mathematical two dimensional syntax.
- Text/documentation, e.g. used as comments to executable formal model specifications.
- Dialogue forms for specification and modification of input data.
- Result tables. The results can be automatically represented in (live) tables, which can even be automatically updated after recomputation.
- Graphical result representation, e.g. with 2D vector and raster graphics as well as 3D vector and surface graphics.
- 2D structure graphs, that for example are used for various model structure visualizations such as connection diagrams and data structure diagrams.

A number of examples of these different forms of cells are available throughout this paper.

3.3.3 Mathematics with 2D-syntax, Greek letters, and Equations

MathModelica uses the syntactic facilities of Mathematica to allow writing formulas in the standard mathematical notation well-known, e.g. from textbooks in mathematics and physics. Certain parts of the Mathematica language syntax are however a bit unusual compared to many common programming languages. The reason for this design choice is to make it possible to use traditional mathematical syntax. The following three syntactic features are unusual:

- Implied multiplication is allowed, i.e. a space between two expressions, e.g. `x` and `f(x)`, means multiplication just as in mathematics. A multiplication operator * can be used if desired, but is optional.
- Square brackets are used around the arguments at function calls. Round parentheses are only used for grouping of expressions. The exception is `TraditionalForm`, see below.
- Support for two-dimensional mathematical syntactic notation such as integrals, division bars, square roots, matrices, etc.

The reason for the unusual choice of square brackets around function arguments is that the implied multiplication makes the interpretation of round parenthesis ambiguous. For example, `f(x+1)` can be interpreted either as a function call to `f` with the argument `x+1`, or f multiplied by `(x+1)`. The integral in the cell below contains examples of both implied multiplication and two-dimensional integral syntax. The cell style is called MathModelica input form (called `StandardForm` in Mathematica) and is used for mathematics and Modelica code in Mathematica syntax:

$$\int \frac{x\, f[x]}{1 + x^2 + x^3}\, dx$$

There is also a purely textual input form using a linear sequence of characters. This is for example used for entering Modelica models in the standard Modelica syntax, and is currently the only cell format in MathModelica that can interpret standard Modelica syntax. However, all mathematics can also be represented in this syntax. The above example in this textual format appears as follows:

```
Integrate[(x*f[x])/(1 + x^2 + x^3), x]
```

Finally, there is also a cell format called `TraditionalForm` which is very close to traditional mathematical syntax, avoiding the square brackets. The above-mentioned syntactic ambiguities can be avoided if the formula is first entered using one of the above input forms, and then converted to `TraditionalForm`.

$$\int \frac{x\, f(x)}{x^3 + x^2 + 1}\, dx$$

The MathModelica environment allows easy conversion between these forms using keyboard or menu commands. Below we show a small example of a Modelica model class `SimpleDAE` represented in the Mathematica style syntax of Modelica that allows greek characters and two dimensional syntax. The apostrophe (') is used for the derivatives just as in traditional mathematics, corresponding to the Modelica `der()` operator.

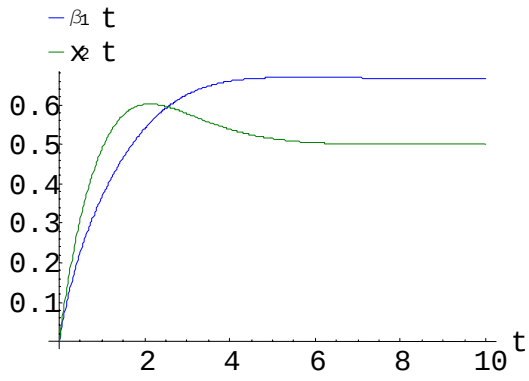
```
Model[SimpleDAE,
 Real β1;
 Real x2;
 Equation[
   β1'/(1 + (β1')^2) + sin[x2']/(1 + (β1')^2) + β1 x2 + β1 == 1;
   sin[β1'] - x2'/(1 + (β1')^2) - 2 β1 x2 + β1 == 0;
 ]]
```

We simulate the model for ten seconds by giving a `Simulate` command:

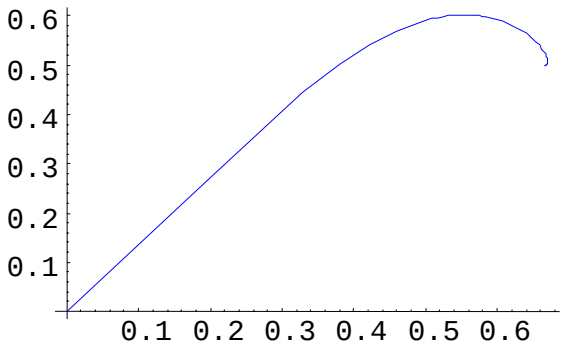
```
Simulate[SimpleDAE,{t,0,10}];
```

We use the command `PlotSimulation` for plotting the solutions for the two state variables, which of course both are functions of time, here denoted by `t` in Mathematica syntax:

```
PlotSimulation[{β1[t], x2[t]}, {t, 0, 10}];
```



A phase plane plot appears as follows:



3.4 Environment and Language Extensibility

Programming environments need to be flexible to adapt to changing user needs. Without flexibility, a programming tool will become too hard to use for practical needs, and stopped to be used. Adaptability and flexibility are especially important for integrated environments, since they need to interact with a number of external tools and data formats, contain many different functions, and usually need to add new ones.

There are two major ways to extend a programming environment:

- Extension of functionality, e.g. through user-defined commands, user-extensible menus, and a scripting languages for programmability.
- Extension of language and notation, e.g. by facilities to add new syntactic constructs and new notation, or extend the meaning of existing ones.

Mathematica has been designed from the start to be an inherently extensible environment, which is what is used in MathModelica. Almost anything can be redefined, extended, or added.

3.4.1 Scripting for Extension of Functionality

An interactive scripting language is a common way of providing extensibility and flexibility in functionality. The MathModelica environment primarily uses the Mathematica language and its interpreter as a scripting language, as can be seen from a number of examples in this paper. Another possibility would be to use the Modelica language itself as a scripting language, e.g. by providing an interpreter for the algorithmic and expression parts of the language. This can easily be realized in MathModelica since the intermediate form has been designed to be compatible with Mathematica, and we already have Modelica input cells: just use Modelica input cells also for commands, which are sent to the Mathematica interpreter instead of the simulator.

3.4.2 Extensible Syntax and Semantics

As was already apparent in the section on mathematical syntax, MathModelica provides a Mathematica-like input syntax for Modelica in addition to the usual Modelica syntax. One reason is to give support for mathematical notation, as explained previously. Another reason is to provide user extensible syntax.

This is easy since syntactic constructs in Mathematica apart from the operators use a simple prefix syntax: a keyword followed by square brackets surrounding the contents of the construct, i.e. the same syntax as for function calls. If there is a need to add a new construct no changes are needed in the parser, and no reserved words need to be added. Just define a Mathematica function to do the desired symbolic or numeric processing.

The other major class of syntactic constructs are operators. There are special facilities in Mathematica to add new operators by defining their priority, operator syntax, and internal representation. It is also possible to extend the meaning of existing operators like +, *, -, etc. However, it is not possible to just use any Mathematica function or operator without a Modelica definition within a Modelica class. For this to work, a MathModelica/Modelica definition of the function or operator must be provided.

3.4.3 Mathematica vs Modelica syntax.

In order to to show the difference between the standard Modelica textual syntax and the extensible Mathematica-like syntax, we first show a simple model in a Modelica-style input cell:

```
model secondordersystem
   Real x(start=0);
   Real xdot(start=0);
   parameter Real a=1;
equation
   xdot=der(x);
   der(xdot)+a*der(x)+x=1;
end secondordersystem;
```

The same model in the Mathematica-like Modelica syntax appears below. Note the use of the simple prefix syntax: a keyword followed by square brackets surrounding the contents of the construct. All reserved words, predefined functions, and types in MathModelica start with an upper-case letter just as in Mathematica. Equation equality is represented by the == operators since = is the assignment operator in Mathematica. The derivative operator is the mathematical apostrophe (') notation rather than der(). The semicolon (;) is a sequencing operator to group more than one declaration, statement, or expression together.

Note that the `Start` attribute values below (`start` in Modelica standard syntax), e.g. for `x` and `xdot`, are defined using declaration modifier equations `Start==0`. These `start` attributes are used as hints for the initial conditions when the simulation starts. The simulator is free to deviate somewhat from these hints if needed to obtain a consistent set of initial values by solving an equation system for the initial values. However, if the attribute `fixed` is `true` (default `false`), then the initial variable value is required to be the `start` attribute value.

```
Model[secondordersystem,
  Real x[{Start == 0}];
  Real xdot[{Start == 0}];
  Parameter Real a == 1;
Equation[
  xdot == x';
  xdot' + a*x' + x == 1
 ]
]
```

3.5 Simulation, Translation, and Graphic Animation of CAD Models

The Model Editor provides an easy-to-use high level user interface that works quite well for most application areas. However, for certain application areas such as design of 3D mechanical parts and assemblies of such parts the two-dimensional user interface of the Model Editor is not very intuitive and sometimes hard to use. On the other hand, tools with 3D interactive user interfaces for design of mechanical systems already exist. These are known as CAD systems for mechanical applications.

For these reasons we have developed an integration mechanism, i.e. a translator, between existing CAD systems and the MathModelica environment. A CAD system is used as the interactive user interface to design the geometry, constraints, and connection structure of the mechanical application. This design is then automatically translated into a mechanical Modelica model for dynamic simulation. The generated Modelica model consists of connected instances of classes from the Modelica MBS (Multi Body System) library [Otter-95], [Otter-96]. Such translators integrated with the simulation environment have so far been developed for the two CAD systems SolidWorks [Engelson-99] and AutoDesk's Mechanical Desktop [Bunus-00].

We have also developed an OpenGL based 3D visualizer and animation system called MVIS (Modelica VISualizer) [Engelson-00] that provides online dynamic display of the mechanical assembly during simulation, or offline display based on saved state information for each time step.

Both the MVIS visualizer and the CAD translators are separate subsystems which communicate with the rest of the MathModelica environment using files and other means. They are not yet official parts of the MathModelica product release, and are therefore indicated by a dotted line in the previously presented MathModelica structure diagram. The interplay between the simulation environment and the CAD environment is shown in Figure 14 below.

Both translators are implemented as CAD system plug-ins that extract geometry, mass, inertia, and constraints information, and translate this information to Modelica source code. This code is combined with other code fragments, e.g. control system models, and simulated.

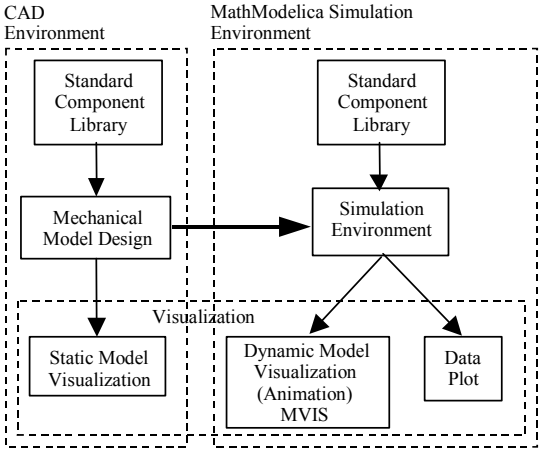


Figure 14. Functional structure of the information flow between the MathModelica simulation environment, the MIVS 3D visualizer and the CAD environment.

The output can subsequently be visualised as a data plot of the system variables and/or as a 3D or 2D dynamic model animation. The 3D visualisations are scenes that display the geometry of the parts in motions prescribed by the simulation results. The graphical user interface of the CAD model and the output visualisation capabilities of the simulation environment make it easy to describe and modify model geometry as well as examine analysis results at the same time. A more detailed picture of the translation and visualization mechanisms including associated data flows it is shown Figure 15 below.

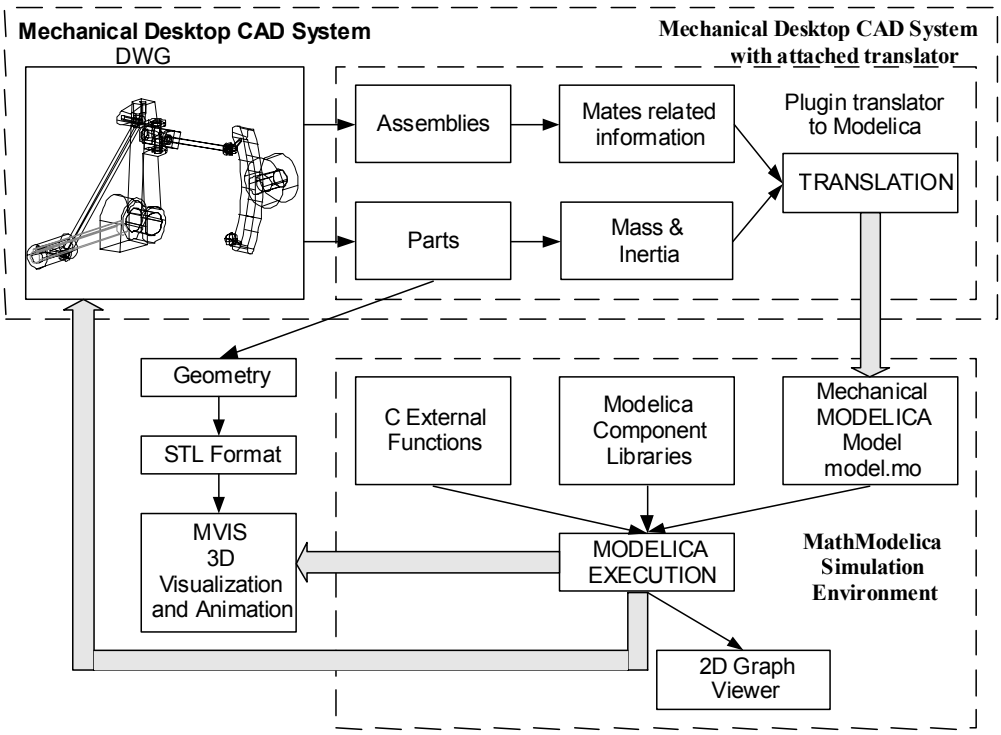


Figure 15. The path from a static CAD model to a dynamic system simulation and visualization.

Figure 15 above shows a mechanical model designed in the AutoCAD Mechanical Desktop Environment serving as the starting point of the specification of the virtual prototype with dynamic properties. The model is first saved in the DWG format, which contains all the information, including connections and mates constraints, related to the geometrical properties of the parts and the whole mechanical assembly.

The geometry of each part is exported to the STL file format [3Dsystems-00] for use by the MVIS visualizer. At the same time, the mass and inertia of the parts are extracted together with mates information from the mechanical assembly. The translator uses this information to generate a corresponding set of Modelica class instances coupled by connections. This automatically generated Modelica file is processed by the MathModelica simulation environment. The simulation code can be enhanced by adding components from other Modelica libraries or by adding externally defined C code. In this phase electrical, control, or hydraulics components can be added to the generated mechanical model, providing multi-domain simulation. The translated CAD model contains a set of dynamic equations of motion. The solution of these equations during simulation computes the dynamic response to a given set of initial conditions including force and/or torque loads, which might even be functions of time.

One might ask why develop yet another tool for multibody simulation of mechanical systems? There are already commercially available MBS simulation packages like ADAMS or Working Model 3D. Many CAD systems are integrated with some multibody simulation tool. However, the primary limitation of these environments is the difficulty of integrating multi-domain simulation within the same environment. Usually an interface to common simulation tools, like MATLAB and Simulink, is provided, but such solutions are not very flexible and do not give good performance because of the loose integration. By contrast, the MathModelica environment provides solutions to both of the following two major requirements:

- The need to integrate multi-domain simulation in the same environment.
- The generation of quality documentation coupled to the design and code.

The main advantage of the MathModelica solution is that multidomain modeling and simulation is available in an integrated way in the same environment. This is provided in a way that is both very flexible and gives very efficient simulations, which for example is needed for tightly interacting system components like controllers embedded in mechanical systems. The control algorithms can for example be tested in parallel with the design of the mechanical parts of the system.

4 Application Examples

This section gives a number of application examples of the use of the Mathmodelica environment. The intent is to demonstrate the power of integration and interactivity - the interplay between the object-oriented modeling and simulation capabilities of Modelica integrated with the powerful scripting facilities of Mathematica within MathModelica. This includes the

representation of simulation results as 1D and 2D interpolating functions of time being combined with arithmetic operations and functions in expressions, advanced plotting facilities, and computational capabilities such as design optimization, fourier analysis, and solution of time-dependent PDEs.

4.1 Advanced Plotting and Interpolating Functions

This section illustrates the flexible usage of simulation results represented as interpolating functions, both for further computations that may include simulation results in expressions, and for both simple and advanced plotting. The simple bouncing ball model below from [MA-02a] is used in the simulation and plotting examples.

```
model BouncingBall
  "Simple model of a bouncing ball"
   constant  Real g = 9.81
      "Gravity constant";
   parameter Real c = 0.9
      "Coefficient of restitution";
   parameter Real radius=0.1
      "Radius of the ball";
   Real height(start = 1)
      "Height of the ball center";
   Real velocity(start = 0)
      "Velocity of the ball";
equation
   der(height) = velocity;
   der(velocity) = -g;
   when height <= radius then
     reinit(velocity,-c*pre(velocity));
   end when;
end BouncingBall;
```

4.1.1 Interpolating Function Representation of Simulation Results

The following simulation of the above `BouncingBall` model is done for a short time period using very few points:

```
res1=Simulate[BouncingBall,{t,0,0.5},
              NumberOfIntervals->10]
```

```
<SimulationData: BouncingBall: 2002-2-
26 10:48:10 : {0., 0.5} : 15 data
points : 1 events : 7 variables>
{c, g, height, radius, velocity,
height' velocity'}
```

The results returned by `Simulate` are represented by an access descriptor or handle. Note that the output also mentions the parameters `c` and `g` in the "variables" list even though their values are constant and not generated by the simulation. Some of the contents of such descriptor are shown as the result of the above call to `Simulate`. At this stage the simulation data is stored on disk and referenced by `res1` which acts as a handle to the simulation data. When one of the variables from the last simulation is referenced, e.g. `height`, `radius`, etc., the data for that variable are loaded into the system in a load-by-need manner, and represented as an ordinary Mathematica `InterpolatingFunction`.

Working with simulation result datasets in Mathematica is made in a very convenient way using the Mathematica `InterpolatingFunction` mechanism that encapsulates the data into a function object and provides interpolation so that the data acts as a regular function. The mechanism of loading simulation data into the system and representing it as a function object is performed by the function `VariableTrajectory`, which can be called explicitly as below, but is called automatically on any variable from the last simulation when referenced.

```
h=VariableTrajectory[height]
```

The expression returned from the `VariableTrajectory` is an anonymous function object having one formal parameter (for the time `t`) and consists of a body containing an expression that computes a value of the variable for the given time. In this case the body consist of a Mathematica `Which` expression that switches between two (or more) `InterpolatingFunction` objects dependent on whether the time is less than the event time point at 0.428 or not. The `InterpolatingFunction` objects uses interpolation order 3 as default but can be altered by using options. Pure discrete data, i.e data changing only at event points, is encapsulated by one `InterpolatingFunction` object with zero interpolation order to get a piecewise constant behavior in the interpolation. This is more efficient than using `Which` statements. The system will automatically choose the most efficient representation of these two alternatives.

The interpolation function can now be used in any computation in Mathematica. In this case we just evaluate the derivative at the time 0.2:

```
h'[0.2]
-1.962
```

In the case above Mathematica made the differentiation of the `InterpolatingFunction` object `h`. Normally the derivatives of the simulation variables are also available in the simulation data.

As previously mentioned, to help the MathModelica user, variables of the most recent simulation are always accessible directly. In this case the function `VariableTrajectory` is automatically applied. Therefore, instead of assigning the variable `h` as above, one can write the following and get the same result:

```
height'[0.2]
-1.962
```

Note, to keep variable values from previous simulations accessible, one should use `VariableTrajectory` on the appropriate variable, specifying the desired descriptor, e.g. `res1`:

```
h=VariableTrajectory[height,
            SimulationResult->res1];
```

Now perform a new simulation, with the result denoted by `res2`:

```
res2=Simulate[BouncingBall,{t,0,0.5},
      NumberOfIntervals->10,
      ParameterValues->c==0.95];
```

Having the previous height curve represented as the function object `h` we can easily compute the difference of the curves between the simulations, e.g. using a plot expression `height[t]-h[t]`.

4.1.2 PlotSimulation

First we simulate the bouncing ball for eight seconds and store the results in the variable `res1` for subsequent use in the plotting examples.

```
res1=Simulate[BouncingBall,{t,0,8}];
```

The command `PlotSimulation` is used for simple standard plots. If nothing else is specified, i.e. by the optional `SimulationResult` parameter, the command refers to the results from the last simulation. In the diagram below the height above ground of the ball from the bouncing ball model simulation is plotted for the first eight seconds of simulation. The optional parameter `PlotJoined` has been set to `False` to create a dotted plot:

```
PlotSimulation[height[t],{t,0,8},
      PlotJoined->False];
```

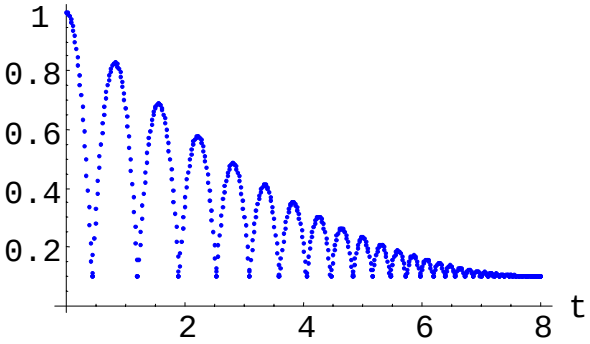


Figure 16. Dotted plot of bouncing ball example model.

`PlotSimulation` can also handle expressions containing simulated results. When this is done, a warning is returned to emphasize that interpolation is performed which could result in a slightly less accurate plot.

```
PlotSimulation[e^-Cos[height[t]], {t, 0, 8}];
```

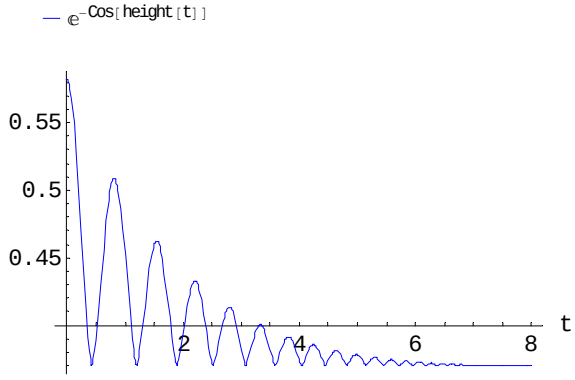


Figure 17. Plot of expression involving interpolated function of simulation result.

Plotting several arbitrary functions can be done using a list of function expressions instead of a single expression:

```
PlotSimulation[{height[t] + √3,
   Abs[velocity[t]]}, {t, 0, 8}];
```

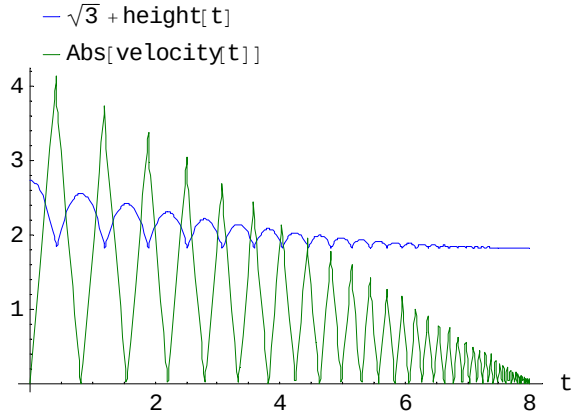


Figure 18. Plotting arbitrary functions in the same diagram.

Now we simulate the bouncing ball again but with different value of the coefficient of restitution, `c`, which is changed to `0.95`. The result is stored in `res2`.

```
res2 = Simulate[BouncingBall, {t, 0, 8},
   ParameterValues → c == 0.95];
```

The optional argument `SimulationResult` specifies which simulation data to use. In this case we will use `res1` and `res2`. The two plots are stored as two graphics objects `gr1` and `gr2`, which are displayed together using the Mathematica command `Show`:

```
gr1 = PlotSimulation[height[t], {t, 0, 8},
    SimulationResult → res1,
    DisplayFunction → Identity];
gr2 = PlotSimulation[height[t], {t, 0, 8},
    SimulationResult → res2,
    DisplayFunction → Identity];
Show[GraphicsArray[{gr1, gr2}],
  DisplayFunction → $DisplayFunction];
```

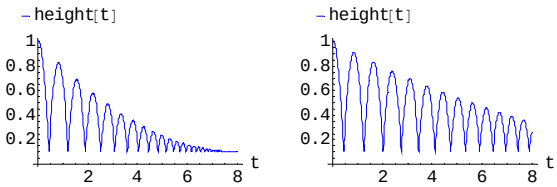


Figure 19. Parallel display of two diagrams.

It is possible to plot variables with the same name from several different simulations together. This is specified by an array value for the optional argument `SimulationResult`:

```
PlotSimulation[height[t], {t, 0, 8},
  SimulationResult → {res1, res2}];
```

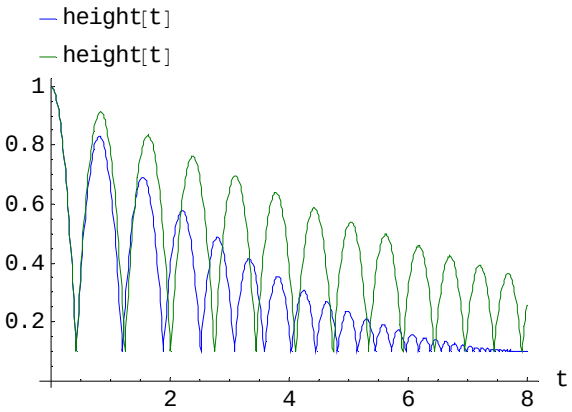


Figure 20. Plot of variables from several simulations in the same diagram.

The plot colors are specified by the `$PlotSimulationColors` predefined variable.

```
PlotSimulation[{height[t], height[t]^2,
    height[t]^3, height[t]^4, height[t]^5,
    height[t]^6, height[t]^7}, {t, 0, 2}];
```

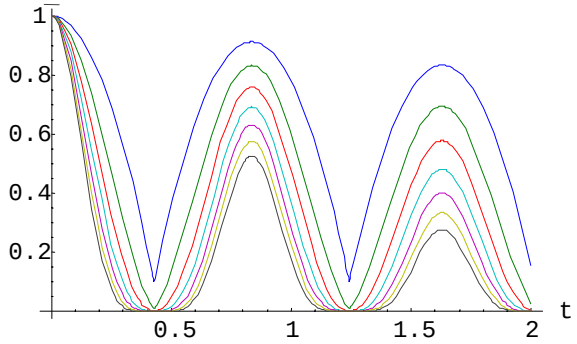


Figure 21. Plot of multiple curves with different colors.

4.1.3 ParametricPlotSimulation

Parametric plots can be done using `ParametricPlotSimulation`.

```
ParametricPlotSimulation[
  {height[t], velocity[t]},
  {t, 0, 8}];
```

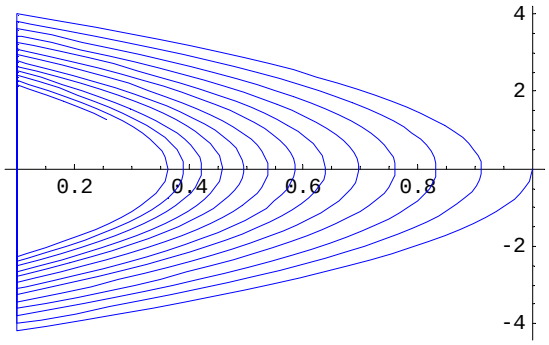


Figure 22. A parametric plot.

In the same way as for `PlotSimulation` the `ParametricPlotSimulation` function can handle several plots:

```
ParametricPlotSimulation[
  {{height[t], velocity[t]},
   {velocity[t], height[t]}},
  {t, 0, 8}];
```

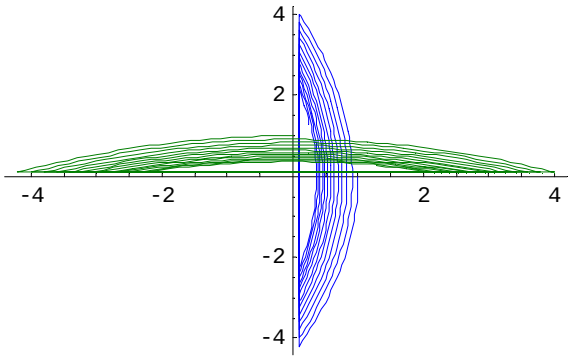


Figure 23. Multiple parametric plots in the same diagram.

`ParametricPlotSimulation` can also handle results form different simulations and plot only the actual data points:

```
ParametricPlotSimulation[{height[t],
   velocity[t]}, {t, 0, 3},
  PlotJoined → False,
  SimulationResult → {res1, res2}];
```

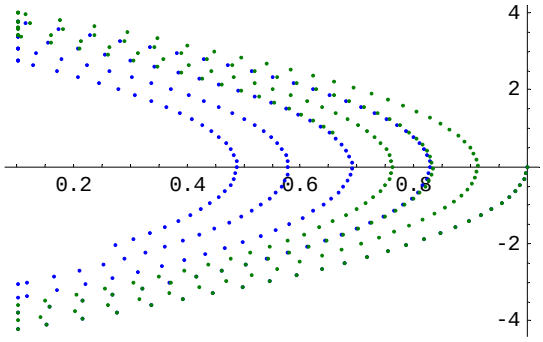


Figure 24. Parametric plots of data points from different simulations in the same diagram.

4.1.4 ParametricPlotSimulation3D

In this example we are going to use the Rossler attractor to show the `ParametricPlotSimulation3D` command. The Rossler attractor is named after Otto Rossler from his work in chemical kinetics. The system is described by three coupled non-linear differential equations:

$$\frac{dx}{dt} = -y - x$$
$$\frac{dy}{dt} = x + \alpha y$$
$$\frac{dz}{dt} = \beta + (x - \gamma)z$$

Here α, β and γ are constants. The attractor never forms limit circles nor does it ever reach a steady state. The model is shown in Mathematica syntax, enabling the use of greek characters:

```
Model[Rossler, "Rossler attractor",
 Parameter Real α == 0.2;
 Parameter Real β == 0.2;
 Parameter Real γ == 8;
 Real x[{Start == 1}];
 Real y[{Start == 3}];
 Real z[{Start == 0}];
 Equation[
  x' == -y - z;
  y' == x + α y;
  z' == β + x z - γ z
 ]
]
```

The model is simulated using different initial values. Changing these can considerably influence the appearance of the attractor.

```
Simulate[Rossler, {t, 0, 40},
  InitialValues → {x == 2, y == 2.5, z == 0}];
```

The Rossler attractor is easy to plot using `ParametricPlotSimulation3D`:

```
ParametricPlotSimulation3D[
  {x[t], y[t], z[t]},
  {t, 0, 40},
  AxesLabel → {X, Y, Z}];
```

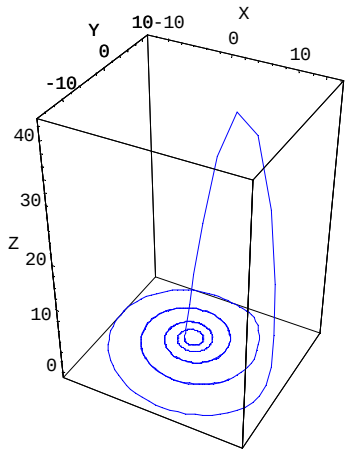


Figure 25. 3-D parametric plot of interpolated curve from the Rossler attractor simulation.

The plot does not look smooth at some areas, especially for high values of Z. Let us take a look at the actual data points of the simulation:

```
ParametricPlotSimulation3D[
  {x[t], y[t], z[t]},
  {t, 0, 40}, PlotJoined → False,
  AxesLabel → {X, Y, Z}];
```

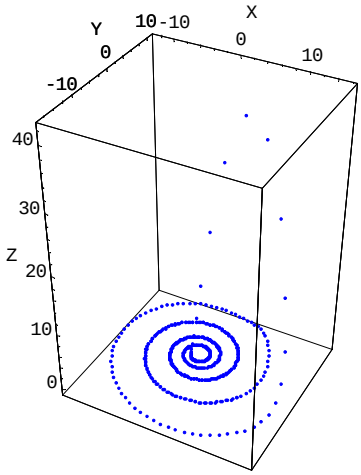


Figure 26. 3-D parametric plot of actual data points from the Rossler attractor simulation.

There seem to be few data points outside the "rings". This can be fixed by adding data points during simulation. The default value is 500. Let us try 1000 data points:

```
Simulate[Rossler, {t, 0, 40},
  InitialValues → {x == 2, y == 2.5, z == 0},
  NumberOfIntervals → 1000];
```

Now the plot looks smoother:

```
ParametricPlotSimulation3D[
  {x[t], y[t], z[t]},
  {t, 0, 40},
  AxesLabel → {X, Y, Z}];
```

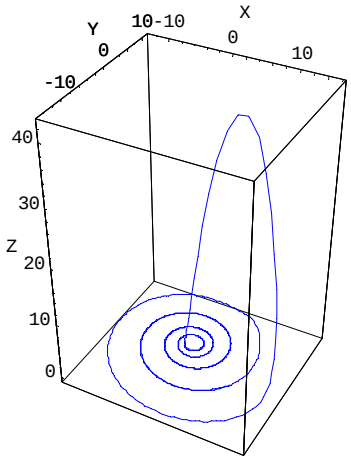


Figure 27. 3-D parametric plot of curve with many data points from the Rossler attractor simulation.

4.2 Design Optimization

This is an example of how the powerful scripting language of MathModelica can be utilized to solve non-trivial optimization problems that contain dynamic simulations. First we will define a Modelica model of a linear actuator with spring damped stopping and then a first order system. Using MathModelica scripting we will then find a damping for the translational spring-damper such that the step response is as "close" as possible to the step response from a first order system.

Consider the following model of a linear actuator with a spring damped connection to an anchoring point:

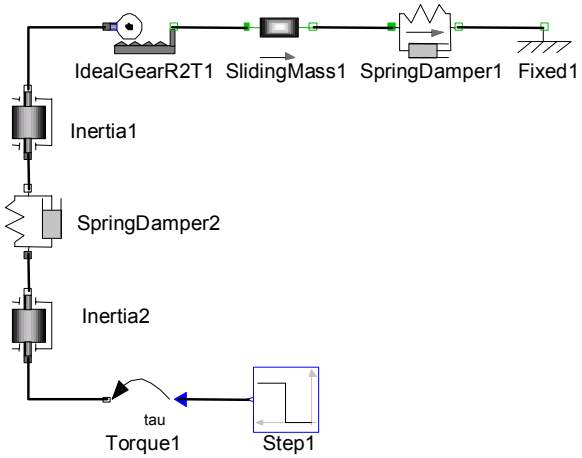


Figure 28. A `LinearActuator` model containing a spring damped connection to an achoring point.

```
model LinearActuator
  import
     Modelica.Mechanics.Translational;
  import Modelica.Mechanics.Rotational;
  import Modelica.Blocks.Sources;
  Translational.SlidingMass
     SlidingMass1(m=0.5);
  Translational.SpringDamper
     SpringDamper1(d=3,c=20);
  Translational.Fixed  Fixed1;
  Rotational.IdealGearR2T
     IdealGearR2T1 ;
  Rotational.Inertia
     Inertia1(J=0.1) ;
  Rotational.SpringDamper
     SpringDamper2(c=15,d=2);
  Rotational.Inertia
     Inertia2(J=0.1) ;
  Rotational.Torque Torque1;
  Sources.Step  Step1;
equation
  connect(Inertia1.flange_b,
             IdealGearR2T1.flange_a);
  connect(IdealGearR2T1.flange_b,
             SlidingMass1.flange_a);
  connect(SlidingMass1.flange_b,
             SpringDamper1.flange_a);
  connect(SpringDamper1.flange_b,
             Fixed1.flange_b);
  connect(Inertia1.flange_a,
             SpringDamper2.flange_b);
  connect(SpringDamper2.flange_a,
             Inertia2.flange_b);
  connect(Inertia2.flange_a,
             Torque1.flange_b);
  connect(Torque1.inPort,
             Step1.outPort)
end LinearActuator;
```

We simulate a step response and store the result in `res0`.

```
res0 = Simulate[LinearActuator,
   {t, 0, 5}];
```

```
PlotSimulation[SlidingMass1.s[t],
  {t, 0, 5}];
```

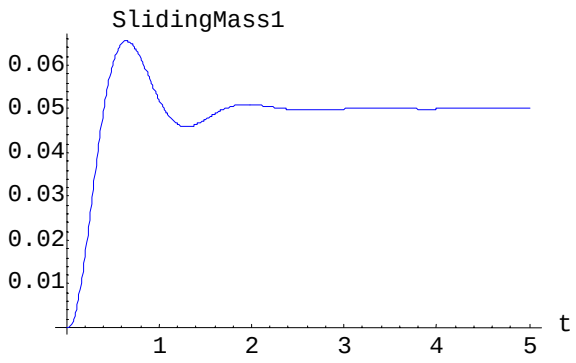


Figure 29. Plot of step response from the linear actuator.

Assume that we have some freedom in choosing the damping in the translational spring-damper. A number of simulation runs show what kind of behavior we have for different values of the dampingparameter `d`. The Mathematica `Table[]` function is used in `Simulate[]` to collect the results into an array `res`. This array then contains the results from simulations of `LinearActuator` with a damping of 2 to 14 with a step size of 2, i.e. seven simulations are performed.

```
res = Table[Simulate[LinearActuator,
    {t, 0, 4},
    ParameterValues →
     {SpringDamper1.d == s}],
   {s, 2, 15, 2}];

PlotSimulation[SlidingMass1.s[t],
  {t, 0, 4},
  SimulationResult → res,
  Legend → False];
```

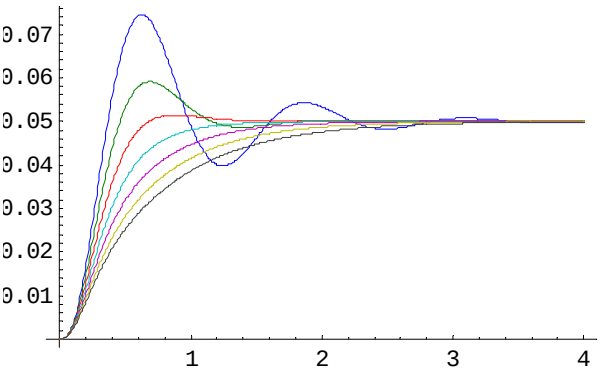


Figure 30. Plots of step responses from seven simulations of the linear actuator with different camping coefficients.

Now assume that we would like to choose the damping `d` so that the resulting system behaves as closely as possible to the following first order system response, obtained by solving a first order ODE using NDSolve:

```
res1 = NDSolve[{0.2 y'[t] + y[t] == 0.05,
    y[0] == 0}, {y}, {t, 0, 4}];
```

We make a comparison with the step response we simulated first (`d=2`) and the first order system.

```
PlotSimulation[{SlidingMass1.s[t],
   y[t] /. res1}, {t, 0, 4},
  SimulationResult → res0];
```

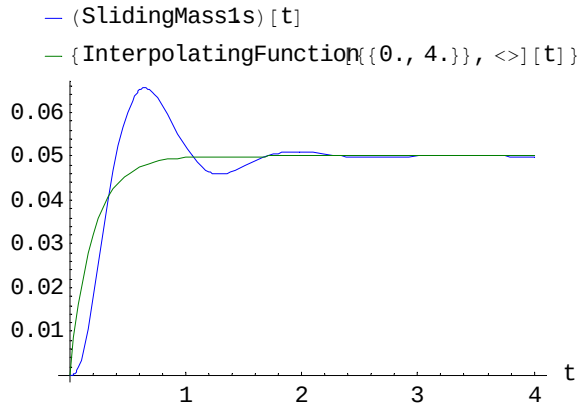


Figure 31. Comparison plot between the step response of the linear actuator and a first-order system.

Now, let us make things a little more automatic. Simulate and compute the integral of the square error from `t=0` to `t=4`.

```
res = Simulate[LinearActuator, {t, 0, 4},
   ParameterValues → {SpringDamper1.d == 3}];

NIntegrate[First[(y[t] /. res1) -
    SlidingMass1.s[t]]^2, {t, 0, 4}]
```

```
0.000162505
```

We define a function, `f(a)`, doing the same thing as above, but for different spring-damper parameters `d=a`.

```
f[a_] := Module[{res, t},
  res = Simulate[
        LinearActuator, {t, 0, 4},
        ParameterValues →
         {SpringDamper1.d == a}
      ];
  NIntegrate[First[(y[t] /. res1) -
      SlidingMass1.s[t]]^2, {t, 0, 4}]]
```

and tabulate some results for $2 \le d = a \le 10$

```
res2 = Table[{a, f[a]}, {a, 2, 10, .5}]
```

```
{{2,0.000317667},
{2.5,0.000221484},{3.,0.000162505},
```

```
{3.5,0.000125513},{4.,0.000102749},
{4.5,0.0000898832},{5.,0.000084067},
{5.5,0.0000836711},{6.,0.0000874586},
{6.5,0.0000945743},{7.,0.000104399},
{7.5,0.000116464},{8.,0.000130394},
{8.5,0.000145922},{9.,0.000162819},
{9.5,0.000180894},{10.,0.000200008}}
```

The tabulated values are interpolated using an interpolating function object. The default interpolation order is 3.

```
fpre = Interpolation[res2];

Plot[fpre[a], {a, 2, 10}];
```

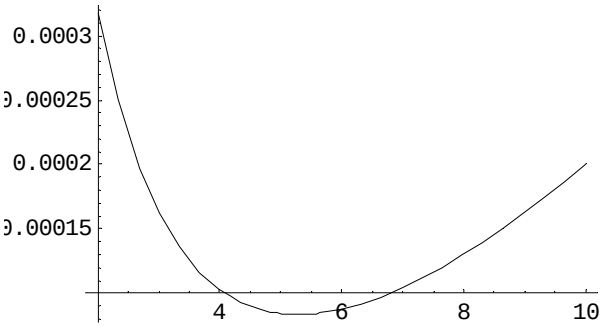


Figure 32. Plot of the error function for finding a minimum deviation from the desired step response.

The minimizing value of a can be computed using FindMinimum:

```
FindMinimum[fpre[s], {s, 4}]

{0.0000832564 , {s → 5.28642 }}
```

A simulation with the optimal parameter value

```
Simulate[LinearActuator, {t, 0, 4},
  ParameterValues →
   {SpringDamper1.d == 5.28642}];
```

A plot comparing the first and second order system response together with a plot of the squared error amplified with a factor 100.

```
PlotSimulation[{SlidingMass1.s[t],
   y[t] /. res1,
   100 (SlidingMass1.s[t] -
       (y[t] /. res1))^2}, {t, 0, 4},
  Legend → False];
```

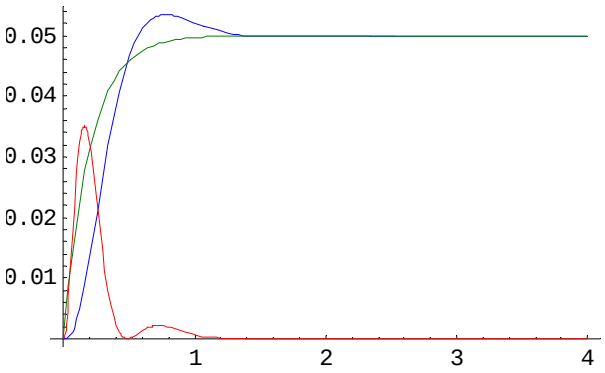


Figure 33. Comparison plot of the first and second order system step responses together with the squared error.

4.3 Fourier Analysis of Simulation Data

Consider a weak axis excited by a torque pulse train. The axis is modeled by three segments joined by two torsion springs. The following diagram is imported from the MathModelica Model Editor where the model was defined.

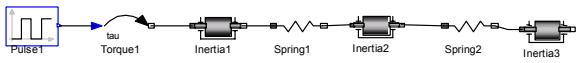


Figure 34. A WeakAxis model excited by a torque pulse train.

The corresponding Modelica code:

```
model WeakAxis
  Modelica.Mechanics.Rotational.Torque
      Torque1;
  Modelica.Mechanics.Rotational.Inertia
      Inertia1;
  Modelica.Mechanics.Rotational.Spring
      Spring1(c=0.7);
  Modelica.Mechanics.Rotational.Inertia
      Inertia2;
  Modelica.Mechanics.Rotational.Spring
      Spring2(c=1);
  Modelica.Mechanics.Rotational.Inertia
      Inertia3;
  Modelica.Blocks.Sources.Pulse
      Pulse1(width={1},period={200});
equation
  connect(Pulse1.outPort,
          Torque1.inPort);
  connect(Torque1.flange_b,
          Inertia1.flange_a);
   connect(Inertia1.flange_b,
           Spring1.flange_a);
   connect(Spring1.flange_b,
           Inertia2.flange_a);
   connect(Inertia2.flange_b,
           Spring2.flange_a);
   connect(Spring2.flange_b,
           Inertia3.flange_a);
end WeakAxis;
```

We simulate the model during 200 seconds:

```
Simulate[WeakAxis , {t, 0, 200}];
```

The plot of the angular velocity of the rightmost axis segment appears as follows:

```
PlotSimulation[{Inertia3.w[t],
   Torque1.τ[t]}, {t, 0, 200}];
```

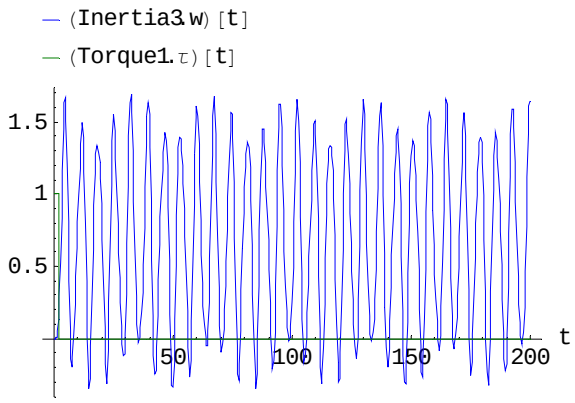


Figure 35. Plot of the angular velocity of the rightmost axis segment of the WeakAxis model.

Now, let us sample the interpolated function `Inertia3.w` using a sample frequency of 4Hz, and put the result into an array using the Mathematica `Table` array constructor:

```
data1 = Table[Inertia3.w[t],
   {t, 0, 200, .25}];
```

Compute the absolute values of the discrete Fourier transform of data1 with the mean removed:

```
fdata1 = Abs[Fourier[data1 -
     MeanValue[data1]]];
```

Plot the first 80 points of the data.

```
ListPlot[fdata1[[Range[80]]],
  PlotStyle → {Red, PointSize[0.015]}];
```

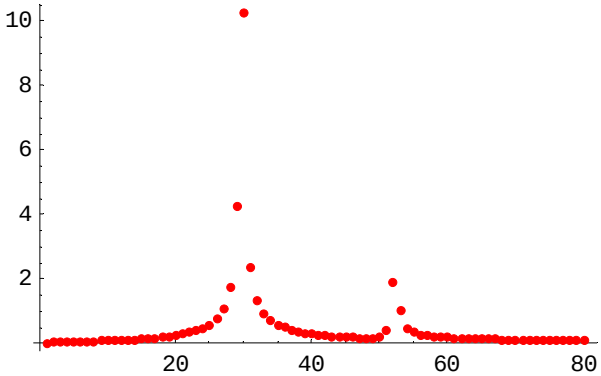


Figure 36. Plot of the data points of the Fourier transformed angular velocity.

It is easy to write a function `FourierPlot` that repeats the above operations. `FourierPlot` also scales the axes such that amplitude of trigonometric components are plotted against frequency (Hz).

```
FourierPlot[Inertia3.w[t], {t, 0, 200},
  0.5, PlotJoined → True, PlotStyle → Red]
```

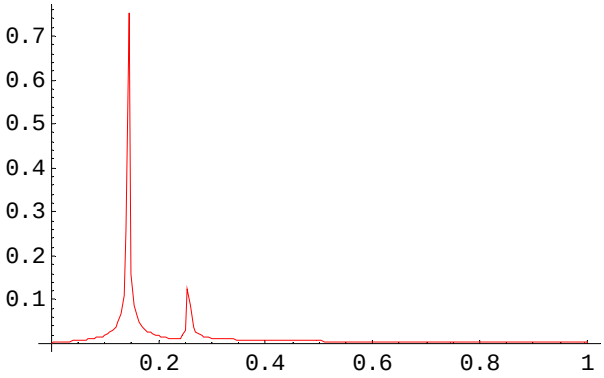


Figure 37. Plot of the data points of the Fourier transformed angular velocity.

It can be shown that the frequencies of the eigenmodes of the system is given by the imaginary parts of the eigenvalues of the following matrix (c_1 and c_2 are the spring constants)

```
1/(2 π) Eigenvalues[{{0, 1, 0, 0, 0, 0},
    {-c1, 0, -c1, 0, 0, 0},
    {0, 0, 0, 1, 0, 0},
    {-c1, 0, -c1 - c2, 0, -c2, 0},
    {0, 0, 0, 0, 0, 1},
    {0, 0, -c2, 0, -c2, 0}} /.
  {c1 → 0.7, c2 → 1}] // Chop
```

```
{0.256077 i, -0.256077 i,
 0.143343 i, -0.143343 i, 0, 0}
```

These values, 0.256077, 0.143344, fit very well with the peaks in the above diagram.

4.4 Solution and 2D-Interpolation of Discretized PDEs

Currently Modelica cannot handle partial differential equations directly since there is only the notion of differentiation with respect to time built into the language. However, in many cases derivatives with respect to other variables such as for example spatial dimensions can be handled by simple discretizations schemes easily implemented using the array capabilities in Modelica. Here we will give an example of how the one dimensional wave equation can be represented in Modelica and how MathModelica can be used for simulation and display of the results, as well as representing the result as a 2D interpolating function.

The one dimensional wave equation is given by a partial differential equation of the following form:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{\partial^2 p}{\partial x^2}$$

where `p = p[x, t]` is a function of both space and time. As a physical example let us consider a duct of length 10 where we let $-5 \le x \le 5$ describe its spatial dimension. Since Modelica only can handle time as the independent variable we need to discretize the problem in the spatial dimension and approximate the spatial derivatives using difference approximations using the approximation

$$\frac{\partial^2 p}{\partial x^2} \approx \frac{p_{i-1} + p_{i+1} - 2\,p_i}{\Delta x^2}$$

Utilizing this approach a Modelica model of a duct whose pressure dynamics is given by the wave equation can be written as follows:

```
model WaveEquationSample
  import Modelica.SIunits;
  parameter SIunits.Length L=10
      "Length of duct";
  parameter Integer n=30
      "Number of sections";
  parameter SIunits.Length dL=L/n
      "Section length";
  parameter SIunits.Velocity c=1;
  SIunits.Pressure
      p[n](start=initialPressure(n));
  Real dp[n](start=fill(0,n));
equation
  p[1]=exp(-(-L/2)^2);
  p[n]=exp(-(L/2)^2);
  dp=der(p);
  for i in 2:n-1 loop
    der(dp[i])=
      c^2*(p[i+1]-2*p[i]+p[i-1])/dL^2;
  end for;
end WaveEquationSample;
```

Here we are using a Modelica function `initialPressure` (defined below) to specify the initial value of the pressure along the duct. Assume that we would like an initial pressure profile in the duct of the form e^{-x^2} ,i.e.,

```
Plot[e^-x^2, {x, -5, 5}];
```

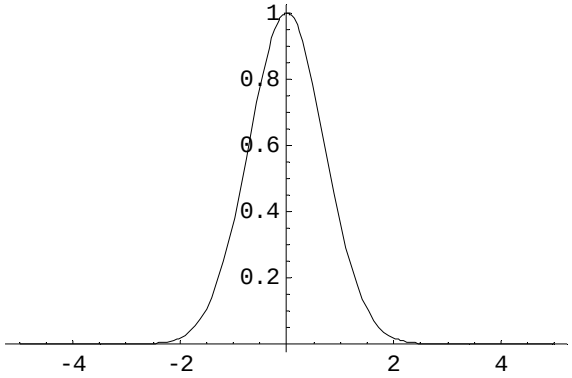


Figure 38. The initial pressure profile of the duct.

```
function initialPressure
  input Integer n;
  output Real p[n];
protected
  parameter Modelica.SIunits.Length
      L=10;
algorithm
 for i in 1:n loop
    p[i]:=exp(-(-L/2+(i-1)/(n-1)*L)^2);
 end for;
end initialPressure;
```

We simulate the `WaveEquationSample` model:

```
Simulate[WaveEquationSample , {t, 0, 10}];
```

The result is packed into a 2D interpolation function:

```
Plot3D[intp[x, t], {x, -5, 5}, {t, 0, 10},
  PlotPoints → 30, PlotRange → {-1, 1},
  AxesLabel → {"x", "t", "p[x,t]"}];
```

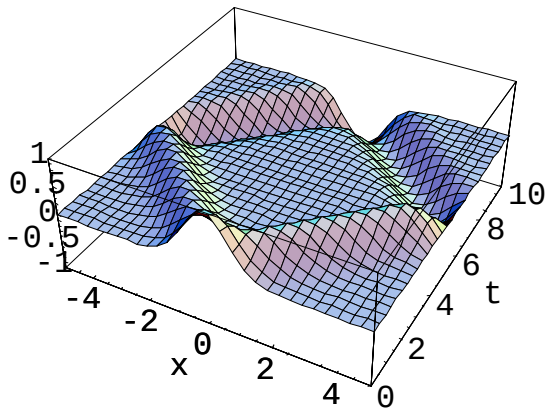


Figure 39. A 3D plot of the pressure distribution in the duct.

Let us also plot the wave as a function of position for a fixed point in time

```
Plot[intp[x, 1.2], {x, -5, 5}];
```

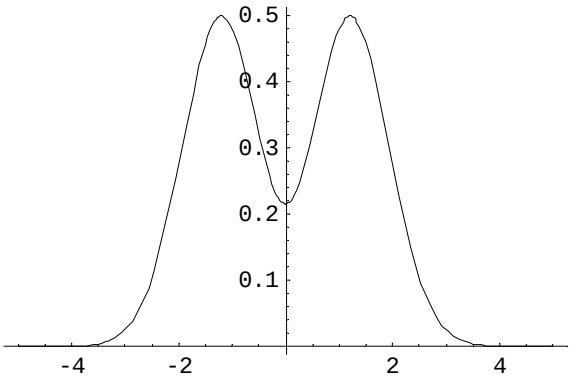


Figure 40. The wave as a function of position at time=1.2.

5 Using the Symbolic Internal Representation

In order to satisfy the requirement of a well integrated environment and language, the new MathModelica internal representation was designed with a Mathematica compatible version of the syntax. Note that the Mathematica version of the syntax has the same internal abstract syntax tree representation and the same semantics as Modelica, but different concrete syntax. Which syntax to use, the standard Modelica textual syntax, or the Mathematica-style syntax for Modelica is however largely a matter of taste.

The fact that the Modelica abstract syntax tree representation is compatible with the Mathematica standard representation means that a number of symbolic operations such as simplifying model equations, performing Laplace transformations, and performing queries on code as well as automatically constructing new code is available to the user. The capability of automatically generating new code is especially useful in the area of model diagnosis, where there is often a need for generating a number of erroneous models for diagnosis based on corresponding fault scenarios.

5.1 Mathematica Compatible Internal Form

An inherent property of Mathematica is that code or models are normally not written as free formatted text. Instead, Mathematica expressions (also called terms) are used, internally represented as abstract syntax trees. 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:

$$head[term_1,\ldots,term_n]$$

For example, an expression: `a+b` is represented as `Plus[a,b]` in prefix form, also called `FullForm syntax.` A while loop is represented as the term `While[test,body].`

In order to satisfy the requirement of a well integrated environment, we designed the new MathModelica internal representation with a Mathematica compatible version of the concrete syntax, called `MathModelicaForm`. Note that `MathModelicaForm` has the same abstract syntax trees and the same semantics as `ModelicaForm` representing standard Modelica, but different concrete syntax. This means that essentially the same language constructs are written differently, as illustrated below. Since the same internal representation is used, a cell expressed in `ModelicaForm` can easily be converted to `MathModelicaForm` or vice versa by just calling `GetDefinition` with a different value of the `Format` parameter.

The Mathematica language syntax uses some special operators, see below, and arbitrary arithmetic expressions composed from terms.

$term_1;\ldots;term_n$ *//sequencing operator*
$\{term_1;\ldots;term_n\}$ *//array/list constructor*
$term_1\ term_2$ *//Implied multiplication by space instead of **
$term_1 == term_2$ *// Equation equality*

Internally the MathModelica system uses the `MathModelicaFullForm` format. This format is the abstract syntax of the MathModelica language where all the elements of the language have been defined to be easy to extract and compare for the functions operating on the MathModelica language representation, as well as achieving a high degree of compatibility with both Modelica and Mathematica.

The following is a simple constant declaration:

```
model Arr
  constant Real
       unitarr[2,2] = {{1,0},{0,1}}
       "2D Identity";
end Arr;
```

This definition is stored internally in the `MathModelicaFullForm` format which can be retrieved by calling the function `GetDefinition` which returns the internal abstract syntax tree representation of the model:

```
ff2 = GetDefinition[Arr,
  Format → MathModelicaFullForm]
```

The tree is wrapped into the node `Hold[]` to prevent symbolic evaluation of the model representation while we are manipulating it. All nodes are shown in prefix form excepts the array/list nodes shown as {...} instead of the prefix form `List[...]` for arrays.

```
Hold[SetType[Arr,
  TYPE[Model[Declaration
     [TYPE[Real, {2, 2}, {Constant}, {}],
      VariableComponent[unitarr,
       ValueBinding[{{1, 0}, {0, 1}}],
       {}, {}, Null]
       ];
       "2D  Identity"
      ], {}, {}, {}
  ], {}, Null, Null
 ]
]
```

A declaration of a variable such as `unitarr` is represented by the `Declaration` node in the abstract syntax. This node has two arguments: the type and the variable instance. 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 instance argument contains a `VariableComponent` including the name of the variable, the initialization (`ValueBinding`), at the end the comment string that is associated with the variable.

There are several goals behind the design of the `MathModelicaFullForm` format, which are fulfilled in the current system:

- *Abstract syntax*. The format systematically sorts out the different constructs in the language making the navigation of types and code easier.
- Preserving the syntactic structure of both Modelica and Mathematica code. This means that the mapping from Modelica to `MathModelica-FullForm` format should be injective, e.g. the source code can be recreated from the intermediate form, and that transformations from Modelica via `MathModelicaFullForm` into Mathematica style Modelica form should be reversible.
- *Explicit semantic structure*. The format has reserved fixed attribute positions for certain kinds of semantic information, to simplify semantic analysis and queries. There is also a *canonical subset* of the format which is even simpler for semantic analysis, but does not always recreate exactly the same source code since the same declaration often can be stated in several ways.
- *Symbol table and type representation* format. The `MathModelicaFullForm` format should be possible to use in the symbol table, e.g. to represent types. Types are represented by anonymous type expressions such as the `TYPE` node in the above example. Anonymous means that the type representation is separate from the entity having the type.
- *Internal standard*.

  The `MathModelicaFullForm` format should be used by all the components in the MathModelica system.

Below we show a small model `secondordersystem` in the different representations. First the textual Modelica model:

```
model secondordersystem
   Real x(start=0);
   Real xdot(start=0);
   parameter Real a=1;
equation
   xdot=der(x);
   der(xdot)+a*der(x)+x=1;
end secondordersystem;
```

This model is parsed and stored internally as Modelica in the abstract syntax `MathModelicaFullForm` representation. This can be retrieved and unparsed into standard Modelica as follows:

```
GetDefinition[secondordersystem,
 Format → ModelicaForm]
```

```
model secondordersystem
  Real x(start=0);
  Real xdot(start=0);
  parameter Real a=1;
equation
  xdot = der(x);
  der(xdot)+a*der(x)+x = 1;
end secondordersystem;
```

We retrieve the model once more, but specify unparsing into the Mathematica style Modelica syntax by setting the format argument to `MathModelicaForm`. The model is enclosed in a Hold node to prevent symbolic evaluation.

```
GetDefinition[secondordersystem,
 Format → MathModelicaForm]
```

```
Hold[Model[secondordersystem ,
  Real x[{Start == 0}];
  Real xdot[{Start == 0}];
  Parameter Real a == 1;
  Equation[
   xdot == x′;
   xdot′ + a x′ + x == 1
  ]
 ]]
```

Finally we obtain the `MathModelicaFullForm` of the model. Certain operators like `{}, ==, +,` etc. are still unparsed as infix notation. To get the prefix notation `List[], Equal[], Plus[],` etc. for these operators, use the `FullForm` command on the result.

```
GetDefinition[secondordersystem,
 Format → MathModelicaFullForm]
```

```
Hold[SetType[secondordersystem , TYPE[Model[
    Declaration[TYPE[Real, {}, {}, {}],
     VariableComponent[x, Null, {},
      {ElementModification[start, EqualBinding[0],
        {}, {}, Null]}, Null]];
    Declaration[TYPE[Real, {}, {}, {}],
     VariableComponent[xdot, Null, {},
      {ElementModification[start, EqualBinding[0],
        {}, {}, Null]}, Null]];
    Declaration[TYPE[Real, {}, {Parameter}, {}],
     VariableComponent[a, EqualBinding[1], {}, {},
      Null]];
    Equation[
     xdot == x′;
     xdot′ + a x′ + x == 1
    ]], {}, {}, {}], {}, Null, Null]]
```

5.2 Extracting and Simplifying Model Equations

This section will illustrate a few user-accessible symbolic operations on equations, such as obtaining the system of equations and the set of variables from a Modelica model, and symbolically simplifying this system of equations with the intention of performing symbolic Laplace transformation.

5.2.1 Definition and Simulation of `Model1`

The example class Model1 has been drawn in the graphic model editor and imported into the notebook below:

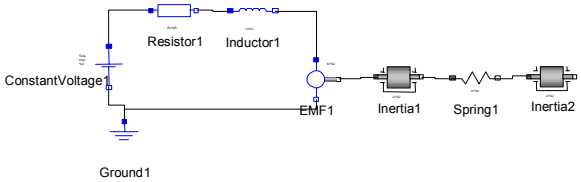


Figure 41. Connection diagram of Model1.

We simulate the model, smooth the result, and make two plots, where the first is a plot of the product of the voltage and current over `Resistor1`:

```
res0 = Simulate[Model1, {t, 0, 25},
   ParameterValues → {Resistor1.R == 0.9}];

res1 = SmoothInterpolation[res0];

PlotSimulation[{(Resistor1.v)[t] *
    (Resistor1.i)[t]}, {t, 0, 10}];
```

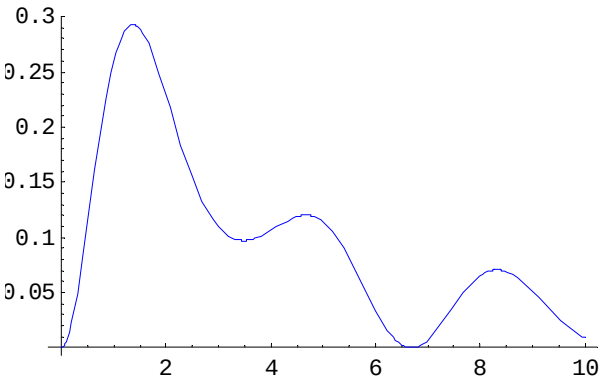


Figure 42. Plot of the current-voltage product over Resistor1 in Model1.

The second plot is parametric where we plot the `Resistor1` current against its derivative for both the original result and the smoothed version:

```
ParametricPlotSimulation[
  {(Resistor1.i)[t],
   (Resistor1.i)'[t]}, {t, 0, 25},
  SimulationResult → {res0, res1}];
```

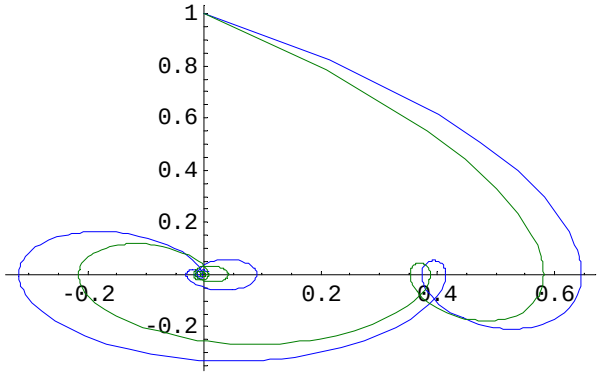


Figure 43. Parametric plots of the Resistor1 current against its derivative, both original and smoothed.

5.2.2 Some Symbolic Computations

Now, flatten `Model1` and extract the model equations and the model variables as lists, and compute the lengths of these lists:

```
eqn = GetFlatEquations[Model1];

Length[eqn]
```

48

```
Length[GetFlatVariables[Model1]]
```

49

There is one equation less than the number of variables. Therefore, add an equation for zero torque on the right flange to the equation system:

```
eqn = Append[eqn,
   Inertia2.flange_b.tau == 0];
```

We would like to simplify the equations by eliminating the connector variables before further symbolic processing. First obtain the connector variables from the flattened model:

```
connvars = GetFlatConnectionVariables
                [Model1]
```

```
{Resistor1.p.v, Resistor1.p.i,
 Resistor1.n.v, Resistor1.n.i,
 Inductor1.p.v, Inductor1.p.i,
 Inductor1.n.v, Inductor1.n.i,
 Ground1.p.v, Ground1.p.i,
 EMF1.p.v, EMF1.p.i, EMF1.n.v,
 EMF1.n.i, EMF1.flange_b.phi,
 EMF1.flange_b.tau,
 ConstantVoltage1.p.v,
 ConstantVoltage1.p.i,
 ConstantVoltage1.n.v,
 ConstantVoltage1.n.i,
 Spring1.flange_a.phi,
 Spring1.flange_a.tau,
 Spring1.flange_b.phi,
 Spring1.flange_b.tau,
 Inertia1.flange_a.phi,
 Inertia1.flange_a.tau,
 Inertia1.flange_b.phi,
 Inertia1.flange_b.tau,
 Inertia2.flange_a.phi,
 Inertia2.flange_a.tau}
```

Use the `Eliminate` function for symbolic elimination of some variables from the system of equations.

```
eqn2 = Eliminate[eqn, connvars]
```

:

```
der[Inertia1.phi] == Inertia1.w &&
 der[Inertia1.w] == Inertia1.a &&
 der[Inertia2.phi] == Inertia2.w &&
 der[Inertia2.w] == Inertia2.a &&
 ConstantVoltage1.i == -(Resistor1.i) &&
 ConstantVoltage1.v ==
   (EMF1.k) (EMF1.w) +
   der[Inductor1.i] (Inductor1.L) +
   (Resistor1.i) (Resistor1.R) &&
 ConstantVoltage1.V ==
  (EMF1.k) (EMF1.w) +
   der[Inductor1.i] (Inductor1.L) +
   (Resistor1.i) (Resistor1.R) &&
 EMF1.i == Resistor1.i &&
 EMF1.v == (EMF1.k) (EMF1.w) &&
 Inductor1.i == Resistor1.i &&
 Inductor1.v == der[Inductor1.i]
   (Inductor1.L) &&
 (Inertia1.a) (Inertia1.J) ==
   (EMF1.k) (Resistor1.i) +
   Spring1.tau &&
 Inertia1.phi == Inertia2.phi -
   Spring1.phi_rel &&
 (Inertia2.a) (Inertia2.J) == -
   (Spring1.tau) &&
 Resistor1.v == (Resistor1.i)
  (Resistor1.R) &&
 (Spring1.c) (Spring1.phi_rel0) ==
   (Spring1.c) (Spring1.phi_rel) -
   Spring1.tau &&
 Inertia2.flange_b.phi == Inertia2.phi &&
 Inertia2.flange_b.tau == 0 &&
 der^(-1)[EMF1.w] == Inertia2.phi -
   Spring1.phi_rel
```

5.3 Symbolic Laplace Transformation and Root Locus Computation

We would now like to perform a Laplace transformation of the symbolic equation system obtained in the previous section. This can be done by the application of two transformation rules: $der^{(-1)}[a\_] \to \frac{a}{s}$, $der[b\_] \to sb$. Note that $der^{(-1)}$ is the inverse of taking a derivative, i.e. an integration operation. Note also that the second rule contains an implied multiplication.

```
eq3 = eqn2 /. {der^(-1)[a_] → a/s, der[b_] → s b}
```

```
s (Inertia1.phi) == Inertia1.w&&
 s (Inertia1.w) == Inertia1.a&&
 s (Inertia2.phi) == Inertia2.w&&
 s (Inertia2.w) == Inertia2.a&&
 ConstantVoltage1i == - (Resistor1.i) &&
 ConstantVoltage1v ==
  (EMF1.k) (EMF1.w) +
   s (Inductor1.i) (Inductor1.L) +
   (Resistor1.i) (Resistor1.R) &&
 ConstantVoltage1V ==
  (EMF1.k) (EMF1.w) +
   s (Inductor1.i) (Inductor1.L) +
   (Resistor1.i) (Resistor1.R) &&
 EMF1i == Resistor1.i&&
 EMF1v == (EMF1.k) (EMF1.w) &&
 Inductor1i == Resistor1.i&&
 Inductor1v == s (Inductor1.i) (Inductor1.L) &&
 (Inertia1.a) (Inertia1.J) ==
  (EMF1.k) (Resistor1.i) +
   Spring1tau &&
 Inertia1phi == Inertia2.phi -
   Spring1phi_rel &&
 (Inertia2.a) (Inertia2.J) == - (Spring1.tau) &&
 Resistor1v == (Resistor1.i) (Resistor1.R) &&
 (Spring1.c) (Spring1.phi_rel0) ==
  (Spring1.c) (Spring1.phi_rel) -
   Spring1tau &&
 Inertia2flange_b.phi == Inertia2.phi&&
 Inertia2flange_b.tau == 0&&
 EMF1.w / s == Inertia2.phi - Spring1.phi_rel
```

Introduce short names for the model parameter to obtain a more concise symbolic notation:

```
shortnames =
  {Resistor1.R → R, Inductor1L → L,
   EMF1k → k, Inertia1.J → J1,
   Spring1c → c1, Spring1.phi_rel0 → 0,
   Inertia2J → J2};
```

Derive the relation between `Inertia2.w` and the input voltage

```
eq4 =
 Eliminate[eq3,
   Complement[
    GetFlatNonConnectionVariables[Model1],
    {Inertia2.w}]] /. shortnames
```

```
(k c1 (ConstantVoltage1.V) ==
   k^2 c1 (Inertia2.w) +
    R s c1 J1 (Inertia2.w) +
    L s^2 c1 J1 (Inertia2.w) +
    k^2 s^2 J2 (Inertia2.w) +
    R s c1 J2 (Inertia2.w) +
    L s^2 c1 J2 (Inertia2.w) +
    R s^3 J1 J2 (Inertia2.w) +
    L s^4 J1 J2 (Inertia2.w)) && s ≠ 0
```

The transfer function H is obtained by symbolically solving for `Inertia2.w` in the equation system `eq4`, and using the obtained solution on a `form Inertia2.w -> expr` to eliminate `Inertia2.w`, thus obtaining `H`:

```
H[s_] = First[Inertia2.w / ConstantVoltage1.V /.
   Solve[eq4, Inertia2.w]]
```

```
(k c1) / (k^2 c1 + R s c1 J1 + L s^2 c1 J1 +
        k^2 s^2 J2 + R s c1 J2 + L s^2 c1 J2 +
        R s^3 J1 J2 + L s^4 J1 J2)
```

5.3.1 A Root Locus Computation

A list of model parameter values is defined for subsequent use:

```
parametervalues = {R → 1, L → 1, c1 → 1,
                   J1 → 1, J2 → 1, k → 1};
```

We compute the poles of the transfer function to obtain the root locus:

```
N[poles /. parametervalues]
```

```
{-0.395123 - 0.506843 i,
 -0.395123 + 0.506843 i,
 -0.104876 - 1.552491 i,
 -0.104876 + 1.552491 i}
```

A root locus plot is given below, substituting values for the model parameters:

```
ParametricPlotComplexPlane[
  CharacteristicRoots[H[s], s] /.
   {R → 1, L → 1, c1 → 0.7,
    J1 → 1, J2 → JJ, k → 1},
  {JJ, 0.1, 2}];
```

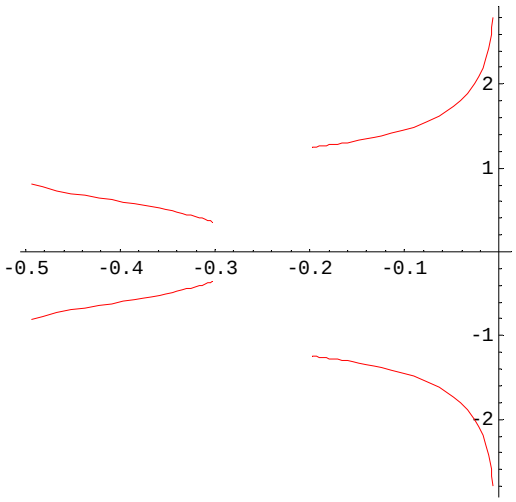


Figure 44. Root locus plot over the complex plane.

5.4 Queries and Automatic Generation of Models

This example of advanced scripting shows how the easily accessible internal representation in the form of abstract syntax trees can be used for automatic generation of models. The `CircuitTemplateFn` is a function returning a symbolic representation of a model. This function has two formal pattern parameters where the second one specifies an internal structure. The first parameter is `name_`, which matches symbolic names. The underscore in `name_` is not part of the parameter identifier itself, it is just a short form of the syntax `name:_`, which means that `name` will match any item.

The second pattern parameter is the list `{type1_,type2_,type3_}`, internally containing the three pattern parameters `type1_`, `type2_`, `type3_`. This second parameter will therefore only match lists of length 3, thereby binding the pattern variables `type1`, `type2`, and `type3` to the three type names presumably occurring in the list at pattern matching. For example, matching `{type1_,type2_,type3_}` against the list `{Capacitor, Conductor, Resistor}` will bind the variable `type1` to `Capacitor`, `type2` to `Conductor`, and `type3` to `Resistor`.

```
CircuitTemplateFn[name_,
  {type1_, type2_, type3_}] := (
  Model[name,
   type1 a;
   type2 b;
   type3 c;
   Modelica.Electrical.Analog.Basic.Ground g;
   Equation[
    Connect[g.p, a.p];
    Connect[a.n, b.p];
    Connect[b.p, c.p];
    Connect[b.n, g.p];
    Connect[c.n, g.p]
   ]
  ])
```

The aim of this exercise is to automatically generate models based on this template for all combinations of the types that extend the type `OnePort` in the library package `Modelica.Electrical.Analog.Basic`.

First we need to extract all the types that extends the type `OnePort` in the library package `Modelica.Electrical.Analog.Basic`. This is done by performing a query operation on the internal form using the Select function which has two arguments: the list to be searched, and a predicate function returning true or false. Only the elements for which the predicate is true are returned. In this case the query is performed on the list of model names in the package `Modelica.Electrical.Analog.Basic`. This list is returned by the function `ListModelNames`.

First we call `GetDefinition` below to load the `Modelica.Eletrical.Analog.Basic` package into the internal symbol table:

```
GetDefinition[Modelica.Electrical.Analog.Basic];
```

Then we perform the actual query:

```
types=Select[
  ListModelNames[
   Modelica.Electrical.Analog.Basic
  ],
  Function[
    modelName,
    Not[
      FreeQ[
         GetDefinition[
           modelName,
           Format->MathModelicaFullForm
         ],
         HoldPattern[
            Extends[
              TYPE[Modelica.Electrical.
                   Analog.Interfaces.
                   OnePort,{},{},{}
              ]
            ]
          ]
       ]
     ]
```

```
  ]
]
```

```
{Modelica.Electrical.Analog.Basic.Inductor,
 Modelica.Electrical.Analog.Basic.Capacitor,
 Modelica.Electrical.Analog.Basic.Conductor,
 Modelica.Electrical.Analog.Basic.Resistor}
```

All 64 three-type combinations, e.g. `{Inductor,Inductor,Inductor}`, `{Inductor,Inductor,Capacitor}`, etc., their prefixes not shown for brevity, of these 4 types are computed by taking a generalized outer product of the three types lists, which is flattened.

```
typecombinations =
  Flatten[Outer
    [List, types, types, types],
   2];

Length[typecombinations]
```

64

We generate a list of 64 synthetic model names by concatenating the string "`foo`" with numbers, using the Mathematica string concatenation operation "<>":

```
names = Table[ToExpression[
   "foo" <> ToString[i]], {i, 64}]
```

```
{foo1, foo2, foo3, foo4, foo5, foo6,
 foo7, foo8, foo9, foo10, foo11, foo12,
 foo13, foo14, foo15, foo16, foo17, foo18,
 foo19, foo20, foo21, foo22, foo23, foo24,
 foo25, foo26, foo27, foo28, foo29, foo30,
 foo31, foo32, foo33, foo34, foo35, foo36,
 foo37, foo38, foo39, foo40, foo41, foo42,
 foo43, foo44, foo45, foo46, foo47, foo48,
 foo49, foo50, foo51, foo52, foo53, foo54,
 foo55, foo56, foo57, foo58, foo59, foo60,
 foo61, foo62, foo63, foo64}
```

Here all 64 test models are created by the call to `MapThread` which applies `CircuitTemplateFn` to each combination.

```
MapThread[CircuitTemplateFn,
  {names, typecombinations}];
```

We retrieve the definition of one of the automatically generated models, `foo53`, and unparse it from its internal representation to the Modelica textual form:

```
GetDefinition[foo53, Format → ModelicaForm]
```

```
model foo53
    Modelica.Electrical.Analog.
                Basic.Resistor a;
    Modelica.Electrical.Analog.
                Basic.Capacitor b;
    Modelica.Electrical.Analog.
                Basic.Inductor c;
    Modelica.Electrical.Analog.
                Basic.Ground g;
equation
  connect(g.p,a.p);
  connect(a.n,b.p);
  connect(b.p,c.p);
  connect(b.n,g.p);
  connect(c.n,g.p);
end foo53;
```

We are now creating a `Total` model within which all 64 generated models will be instantiated. First create an empty model:

```
Model[Total,
 ];
```

Then use the `Within` statement to move the current scope inside the model and then make a declaration, i.e. instantiation of the first test model:

```
Within[Total]

Declare[foo1 m1]
```

Since we are free to use Mathematica scripting we had better use a loop for the 63 remaining declarations:

```
Do[
  With[{type = names[[i]],
    instanceName =
     ToExpression["m" <> ToString[i]]},
   Declare[type instanceName]
  ],
  {i, 2, Length[names]}
 ];
```

Finally we move the scope back to the global scope.

```
EndWithin[]
```

Retrieve the generated model `Total`, where we have abbreviated the output to save some space.

```
GetDefinition[Total, Format → ModelicaForm]
```

```
model Total
   foo64 m64;
   foo63 m63;
   foo62 m62;
   foo61 m61;
   foo60 m60;
   ...
   foo3 m3;
   foo2 m2;
   foo1 m1;
end Total;
```

Finally, simulate the `Total` model to verify that the test models are semantically correct.

```
Simulate[Total, {t, 0, 1}];
```

5.5 Language Extension Example for PDEs.

As previously stated, the uniform prefix syntax makes it easy to experiment with language extensions since both the syntax and the internal representation are obtained automatically. The example below is from an experiment in extending Modelica with partial differential equations [Saldamli-01]. Here we have added a new restricted class called `Domain`, the prefix `Space`, and a new kind of equation section called `Boundary` containing equations that specify boundary conditions.

```
Class[TestModel,
  Parameter Real xc == 0;
  Parameter Real yc == 0;
  Parameter Real r = 1;
  Parameter Real delta = 0;
  Domain Circle dom[{xc == xc, yc == yc, r == r}];
  Space Real u;
  Boundary[
   u[[{dom, 0}]] == finit[dom.x, dom.y];
   ∂t u[[{dom, 0}]] == 0;
   u[[{dom.lefthalf, time}]] == delta;
   ∂x u[[{dom.lefthalf, time}]] == 0;
   ∂y u[[{dom.lefthalf, time}]] == 0;
   u[[{dom.righthalf, time}]] == delta;
   ∂x u[[{dom.righthalf, time}]] == 0;
   ∂y u[[{dom.righthalf, time}]] == 0;
  ];
  Equation[
   ∂{t,2} u[[{dom, time}]] == ∂{x,2} u[[{dom, time}]]
       + ∂{y,2} u[[{dom, time}]];
  ];
 ];
```

A plotted result of a solution at a specific time instant from running the prototype simulator on this model:

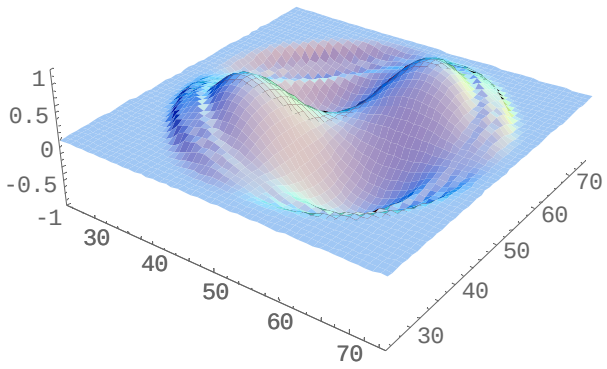


Figure 45. Plot of PDEs solution at a specific time instant.

6 Conclusion

This paper has presented a number of important issues concerning integrated interactive programming environments, especially with respect to the MathModelica environment for object-oriented modeling and simulation. We have especially emphasized environment properties such as integration and extensibility.

One of the current strong trends in software systems is the gradual unification of documents and software. Everything will eventually be integrated into a uniform, perhaps XML-based, representation. The integration of documents, model code, graphics, etc. in the MathModelica environment is one strong example of this trend.

Another important aspect is extensibility. Experience has shown that tools with built-in extensibility mechanisms can cope with unforeseen user needs to a great extent, and therefore often have a substantially longer effective usage lifetime.

The MathModelica system is currently one of the best existing examples of advanced integrated extensible environments. However, as most systems, it is not perfect. There are still a number of possible future improvements in the system including enhanced programmability and extensibility.

Acknowledgements

We thank the entire MathModelica team from MathCore AB and the Dymola team at Dynasim, as well as our colleagues at PELAB - the Programming Environment Laboratory, without whom this work would not have been possible. We also would like to thank Peter Bunus for inspiration and help in preparing this paper.

Acknowledgements to the following individuals in the MathModelica team for contributions to the design and implementation of the MathModelica system: Andreas Karström, Pontus Lidman, Henrik Johansson, Yelena Turetskaya, Mikael Adlers, Peter Aronsson, Vadim Engelsson, and to Jan Brugård and Andreas Idebrant for contributions to the MathModelica documentation including a number of the examples used in this paper. Thanks to Kristina Swenningsson for creating a nice working athmosphere at MathCore AB.

Modelica Association Members

The authors would also like to thank the other members of the Modelica Association (see below) for inspiring discussions and contributions to the Modelica language design. Modelica 2.0 was released March 15, 2002. The Modelica Association was formed in Linköping Sweden, at Feb. 5, 2000 and is responsible for the design of the Modelica language (see www.modelica.org).

Special thanks to Hilding Elmqvist, who was the first chairman of the Modelica Association, and to

Martin Otter, second chairman of the Modelica Association.

Contributors to the Modelica Language, version 2.0

Peter Aronsson, MathCore, Linköping
Bernhard Bachmann , University of Applied Sciences, Bielefeld
Peter Beater, University of Paderborn, Germany
Dag Brück, Dynasim, Lund, Sweden
Peter Bunus, Linköping University, Sweden
Hilding Elmqvist, Dynasim, Lund, Sweden
Vadim Engelson, Linköping University, Sweden
Peter Fritzson, Linköping University, Sweden
Rüdiger Franke, ABB Corporate Research, Ladenburg
Pavel Grozman, Equa, Stockholm, Sweden
Johan Gunnarsson, MathCore, Linköping
Mats Jirstrand, MathCore, Linköping
Sven Erik Mattsson, Dynasim, Lund, Sweden
Hans Olsson, Dynasim, Lund, Sweden
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
Levon Saldamli, Linköping University, Sweden
Michael Tiller, Ford Motor Company, Detroit, U.S.A.
Hubertus Tummescheit, Lund Institute of Technology, Sweden
Hans-Jürg Wiesmann, ABB Corporate Research Ltd., Baden, Switzerland

Contributors to the Modelica Standard Library, version 2.0

Peter Beater, University of Paderborn, Germany
Christoph Clauß, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany

Funding Organizations

Several funding organizations have over the years contributed to research on object-oriented modeling language design and simulation tool technology at PELAB, Department of Computer and Information Science, Linköping University, preceding the development of the MathModelica system. These include NUTEK - The Swedish Board for Technical Development, e.g within the Modelica Tools project, TFR - the previous Swedish Council for Technical research, SSF - the Swedish Strategic Research Foundation under the ECSEL project, WITAS – the Wallenberg Laboratory for Information Technology and Autonomous Systems at Linköping University, and EU under the projects GIPE-II, PREPARE, and Europort. We also acknowledge support from SKF AB.

References

[3DSystems-00] 3Dsystems Inc. Stereo Lithography Interface Specification - The STL format. 2000.

[Abadi-Cardelli-96] Martin Abadi and Luca Cardelli, *A Theory of Objects*, Springer Verlag, ISBN 0-387-94775-2, 1996.

[Andersson-94] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.

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

[Bunus-00] Peter Bunus, Vadim Engelson, Peter Fritzson. Mechanical Models Translation and Simulation in Modelica. In *Proceedings of Modelica Workshop 2000*. Lund University, Lund, Sweden, O ct 24-26, 2000.

[Elmqvist-78] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Elmqvist-96] Hilding Elmqvist, Dag Bruck, Martin Otter. *Dymola - User's Manual*. Dynasim AB, Research Park Ideon, Lund, 1996.

[Elmqvist-99] Hilding Elmqvist, Sven-Erik Mattsson and Martin Otter. Modelica - A Language for Physical System Modeling, Visualization and Interaction. In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design*, Hawaii, Aug. 22-27, 1999.

[Engelson-99] Vadim Engelson, Håkan Larsson, Peter Fritzson. 1999. A Design, Simulation and Visualization Environment for Object-Oriented Mechanical and Multi-Domain Models in Modelica. In *Proceedings of the IEEE International Conference on Information Visualization*, pp 188-193, London, July 14-16, 1999.

[Engelson-00] Vadim Engelson. *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing*. Ph.D. Thesis, Dept. of Computer and Information Science, Linköping University, Linköping, Sweden. 2000.

[Ernst-97] Thilo Ernst, Stephan Jähnichen, and Matthias Klose: The Architecture of the Smile/M Simulation Environment. *Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, Vol. 6, Berlin, Germany, pp. 653-658, 1997.

[Fritzson-83] Peter Fritzson. Symbolic Debugging through Incremental Compilation in an Integrated Environment. *The Journal of Systems and Software*, 3, 285-294, (1983).

[Fritzson-92a] Peter Fritzson, Dag Fritzson. The Need or High-Level Programming Support in Scientific Computing - Applied to Mechanical Analysis. *Computers and Structures*, Vol. 45, No. 2, pp. 387-295, 1992.

[Fritzson-92b]Peter Fritzson, Lars Viklund, Johan Herber, Dag Fritzson: Industrial Application of Object-Oriented Mathematical Modeling and Computer Algebra in Mechanical Analysis, In *Proc. of TOOLS EUROPE'92*, Dortmund, Germany, March 30 - April 2, 1992. Published by Prentice Hall.

[Fritzson-95] Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber. High Level Mathematical Modeling and Programming in Scientific Computing. *IEEE Software*, pp. 77-87, July 1995.

[Fritzson-98] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECOOP'98 , Brussels, Belgium, July 20-24, 1998.

[Fritzson-98b] Peter Fritzson, Vadim Engelson, Johan Gunnarsson. An Integrated Modelica Environment for Modeling, Documentation and Simulation. In *Proceedings of Summer Computer Simulation Conference '98*, Reno, Nevada, USA, July 19-22, 1998.

[Goldberg-89] Adele Goldberg and David Robson, *Smalltalk-80, The Language*. Addison-Wesley, 1989

[Jirstrand-99] Mats Jirstrand, Johan Gunnarsson, and Peter Fritzson. MathModelica - a new modeling and simulation environment for Modelica. In *Proceedings of the Third International Mathematica Symposium*, IMS'99, Linz, Austria, Aug, (1999).

[Knuth-84] Donald E. Knuth. Literate Programming. *The Computer Journal*, NO27(2) (May): 97-111. (1984)

[Knuth-94] Donald E. Knuth, Silvio Levy. The Cweb System of Structured Documentation /Version 3.0. Addison-Wesley Pub Co; 1994.

[MA-97a] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 1.0*, Sept 1997.

[MA-97b] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 1.0*, Sept 1997.

[MA-02a] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 2.0*, March 2002.

[MA-02b] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 2.0*, February 2002.

[Mattsson-93] Sven-Erik Mattsson, Mats Andersson, and Karl-Johan Åström. Object-oriented modelling and simulation. In Linkens, Ed., *CAD for Control Systems*, chapter 2, pp. 31-69. Marcel Dekker Inc, New York, 1993.

[Oh-96] Min Oh,3 and C.C. Pantelides. A modelling and simulation language for combined lumped and distributed parameter systems. *Computers and Chemical Engineering*, 20, pp. 611--633, 1996.

[Piela-91] Piela P.C., T.G. Epperly, K.M. Westerberg, and A.W. Westerberg. ASCEND: An object-oriented computer environment for modeling and analysis: the modeling language. *Computers and Chemical Engineering*, 15:1, pp. 53--72, 1991.

[Otter-95] Martin Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*, Dissertation, Fortshrittberichte VDI, Reihe 20, Nr 147. 1995.

[Otter-96] Martin Otter, Hilding Elmqvist, Francois E. Cellier. Modeling of Multibody Systems with the Object-oriented Modeling Language Dymola. *Nonlinear Dynamics*, 9:91-112, Kluwer Academic Publishers. 1996.

[Sahlin-96] Per Sahlin, A. Bring, and E.F. Sowell. The Neutral Model Format for building simulation, Version 3.02. Technical Report, Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden, June 1996.

[Saldamli-01] Levon Saldamli, Peter Fritzson. A Modelica-Based Language for Object-Oriented Modeling with Partial Differential Equations. In *Proceedings of the 4th International EuroSim Congress*, Delft, the Netherlands, June 26-29, 2001.

[Sandewall-78] Erik Sandewall. Programming in an Interactive Environment: the "LISP" Experience. *Computing Surveys*, Vol. 10, No. 1, March 1978.

[Teitelman-69] Warren Teitelman. Toward a Programming Laboratory. In *Proc. of First Int. Jt. Conf. on Artificial Intelligence*, 1969.

[Teitelman-74] Warren Teitelman. *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, CA, 1974.

[Teitelman-77] Teitelman, W. A display oriented programmer's assistant. Computer, 39-50. (1977, *August 22-25*)

[Tiller-01] Michael M. Tiller. I*ntroduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[Visio] http://www.microsoft.com/office/visio/

[Wolfram-88] Stephen Wolfram. *Mathematica System for Doing Mathematics by Computer*. Addison-Wesley, 1988.

[Wolfram-97] Stephen Wolfram. *The Mathematica Book*, Wolfram Media, 1997.