

MathModelica

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

(Abridged Version*)

Peter Fritzson¹, Johan Gunnarsson², Mats Jirstrand²

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. These 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

* The complete version of the paper can be found at <http://www.mathcore.com> and <http://www.ida.liu.se/~pelab/modelica/>

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-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 a 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. 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.

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, 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 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 1 below:

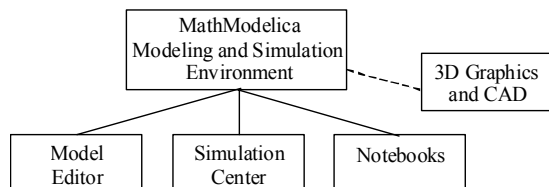


Figure 1. The MathModelica system architecture.

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.

Additionally, MathModelica is loosely coupled to two optional subsystems for 3D graphics visualization and automatic translation of CAD models to Modelica. [Bunus-00], [Engelson-99]. [Engelson-00]. 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.

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.

2.1 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 2. 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.

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.

2.2 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 3 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.

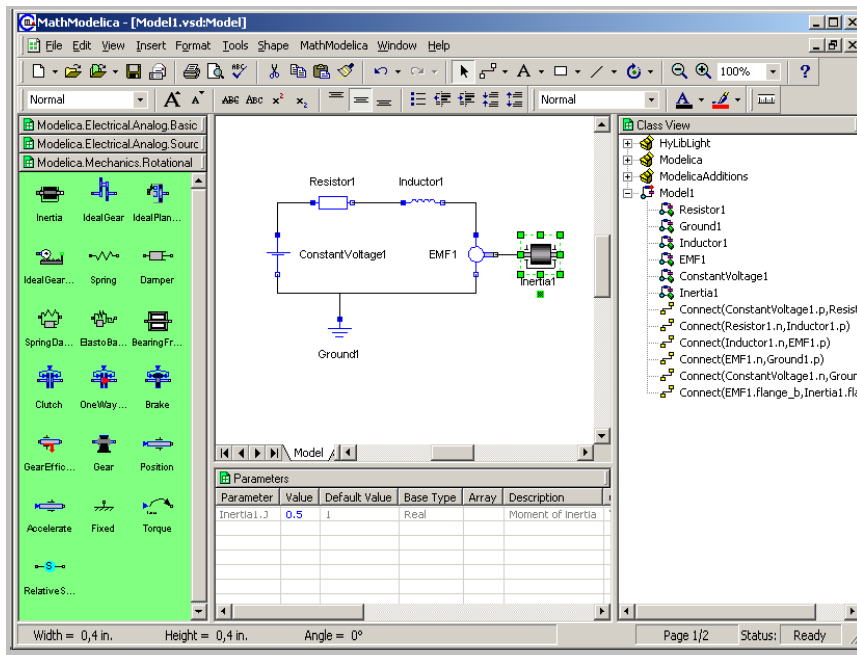


Figure 2. The Graphic Model Editor showing an electrical motor with the Inertia parameter J modified.

- 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 3, allows changing parameter values, whereas the Variables section allows modifying initial (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 no parameter influencing the equation structure has been changed. Such parameters are sometimes called structural parameters.

2.3 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 *Notebooks*, which is documents that may contain technical computations, text, and graphics. Hence, these documents are suitable to be used both as simulation scripting tools, 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 notebooks is shown in Figure 4.

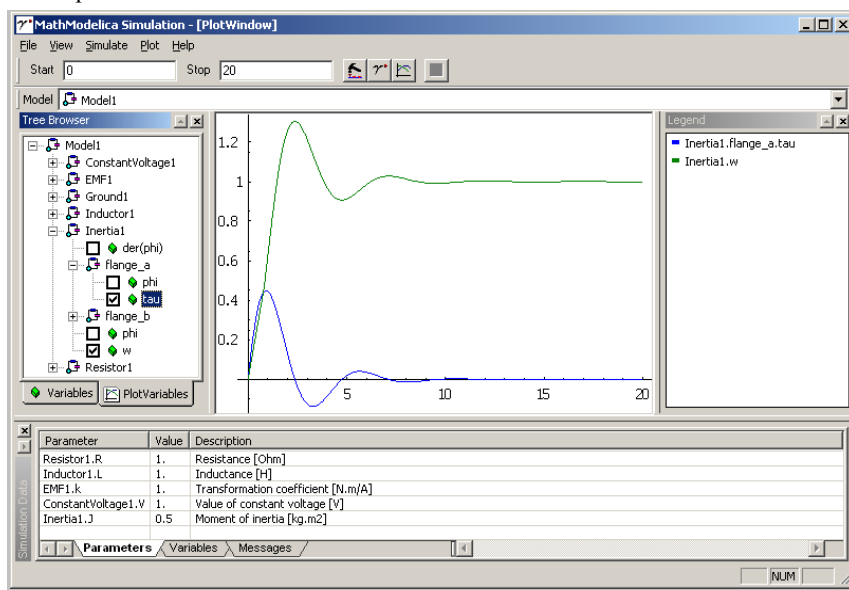


Figure 3. The Simulate window with plots of the signals Inertia1.flange_a.tau and Inertia1.w .

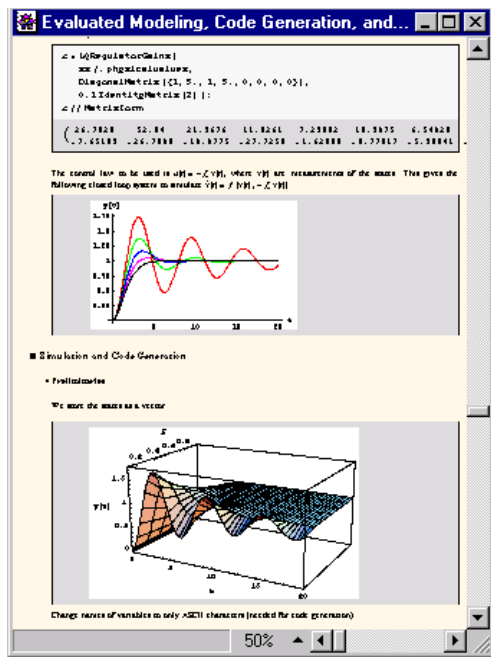


Figure 4. Examples of MathModelica notebooks..

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 implementations 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.

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 functions 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

2.3.1 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.

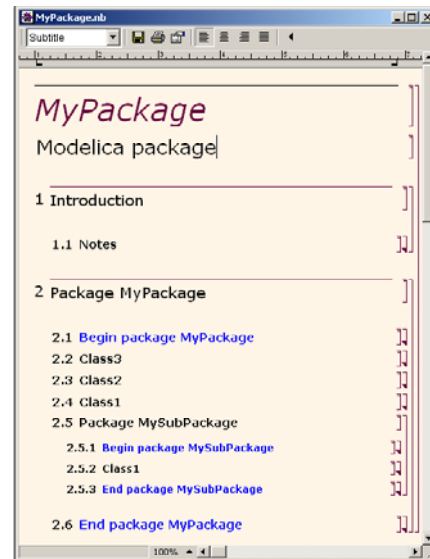


Figure 5. The package Mypackage in a notebook

In the MathModelica system, Modelica packages including documentation and test cases are primarily stored as notebooks, e.g. as in Figure 4. 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 5, 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;

```

2.3.2 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.
- *Mathematical* 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.

2.3.3 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 Traditional Form, 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 standard form 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 traditional form.

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

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.

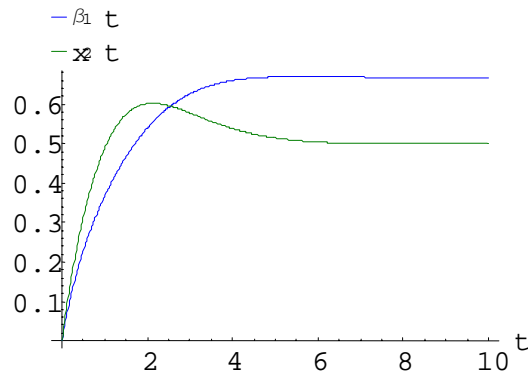
```
Model [SimpleDAE,
Real beta_1;
Real x_2;
Equation [
  beta_1' / (1 + (beta_1')^2) + sin[x_2'] / (1 + (beta_1')^2) + beta_1 x_2 + beta_1 == 1;
  sin[beta_1'] - x_2' / (1 + (beta_1')^2) - 2 beta_1 x_2 + beta_1 == 0;
]]
```

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[{beta_1[t], x_2[t]}, {t, 0, 10}];
```



2.4 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 is 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.

2.4.1 Scripting for Extension of Functionality

An interactive scripting language is a common way of providing extensibility of 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.

2.4.2 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.

2.4.3 Mathematica vs Modelica syntax.

In order 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.

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

3 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. For the PDEs see the long version of the paper.

3.1 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.

3.1.1 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. Some of the contents of such descriptor is 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 is loaded into the system in an load-by-need manner, and represented as an InterPolatingFunction.

3.1.2 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.

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

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

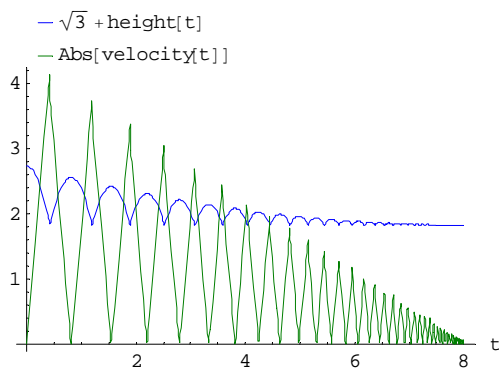


Figure 6. Plotting arbitrary functions in the same diagram.

3.1.3 ParametricPlotSimulation

Parametric plots can be done using `ParametricPlotSimulation`.

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

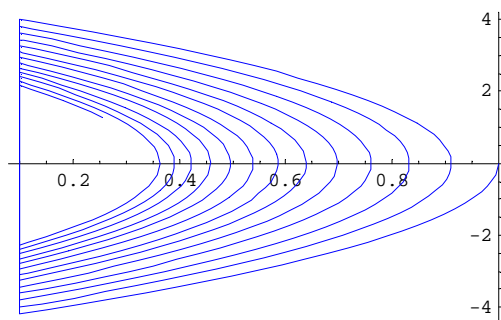


Figure 7. A parametric plot.

3.1.4 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  $\alpha$  == 0.2;
Parameter Real  $\beta$  == 0.2;
Parameter Real  $\gamma$  == 8;
Real x[{Start == 1}];
Real y[{Start == 3}];
Real z[{Start == 0}];
Equation[
x' == -y - z;
y' == x +  $\alpha$  y;
z' ==  $\beta$  + x z -  $\gamma$  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},
NumberOfIntervals -> 1000];
```

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

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

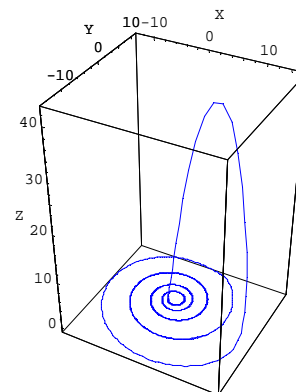


Figure 8. 3-D parametric plot of curve with many data points from the Rossler attractor simulation.

3.2 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:

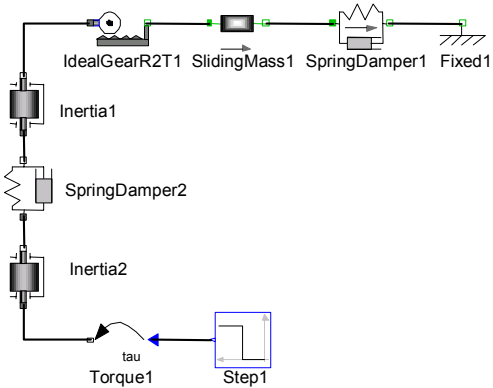


Figure 9. A LinearActuator model containing a spring damped connection to an anchoring point.

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];
```

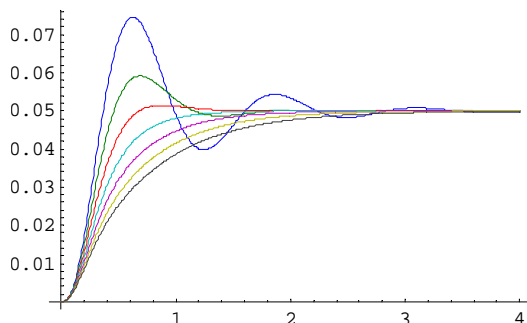


Figure 10. Plots of step responses from seven simulations of the linear actuator with different damping coefficients.

Now assume that we would like to choose the damping d so that the resulting system behaves as closely as possible to a certain first order system response.,

We simulate for different values of d and interpolate the result

```
f_pre = Interpolation[res2];
Plot[f_pre[a], {a, 2, 10}];
```

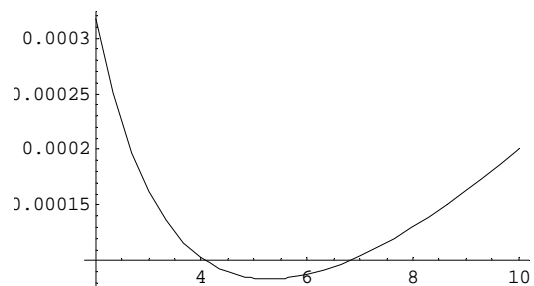


Figure 11. Plot of the error function for finding a minimum deviation from the desired step response.

The minimizing value of a can be computed using `FindMinimum`:

```
FindMinimum[f_pre[s], {s, 4}]
{0.0000832564 , {s -> 5.28642}}
```

3.3 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.

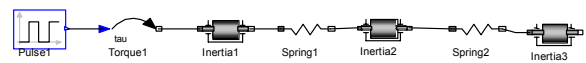


Figure 12. A WeakAxis model excited by a torque pulse train.

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.tau[t]}, {t, 0, 200}];
```

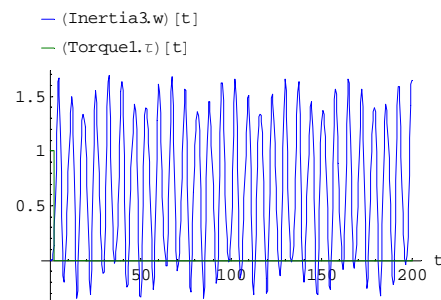


Figure 13. Plot of the angular velocity of the rightmost axis segment of the WeakAxis model.

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:

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

Plot the 80 first points of the data.

```
ListPlot[fdatal[[Range[80]]],
  PlotStyle -> {Red, PointSize[0.015]}];
```

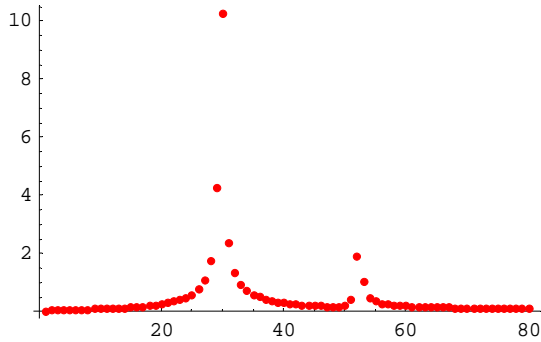


Figure 14. Plot of the data points of the Fourier transformed angular velocity.

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)

$$\frac{1}{2\pi} \text{Eigenvalues} \left[\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -c_1 & 0 & -c_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -c_1 & 0 & -c_1 - c_2 & 0 & -c_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -c_2 & 0 & -c_2 & 0 \end{pmatrix} \right].$$

$\{c_1 \rightarrow 0.7, c_2 \rightarrow 1\}$ // 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 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.

4.1 Mathematica Compatible Internal Form

An inherent property of Mathematica is that models or code is 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:

$$\text{head}[term_1, \dots, term_n]$$

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 syntax. Note that MathModelica has the same abstract syntax trees and the same semantics as Modelica, but different concrete syntax. This means that essentially the same language constructs are written differently, as illustrated below.

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

```
term1;...;termn //sequencing operator
{term1;...;termn} //array/list constructor
term1 term2 //Implied multiplication by space
instead of *
term1 == term2 //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:

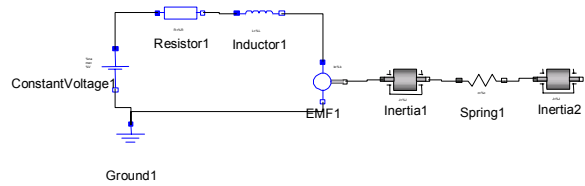
- *Abstract syntax.* The format systematically sorts out the different constructs in the language making the navigation of types and code easier.
- Preserving the syntactic structure of both `Modelica` and `Mathematica` code. This means that the mapping from `Modelica` to `MathModelica-FullForm` format 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.

4.2 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.

4.2.1 Definition and Simulation of Model1

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



Fi

Figure 15. Connection diagram of Model1.

We simulate the model, smooth the result, and make a plot.

```

res0 = Simulate[Model1, {t, 0, 25},
  ParameterValues -> {Resistor1.R == 0.9}];
res1 = SmoothInterpolation[res0];

```

The 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}];

```

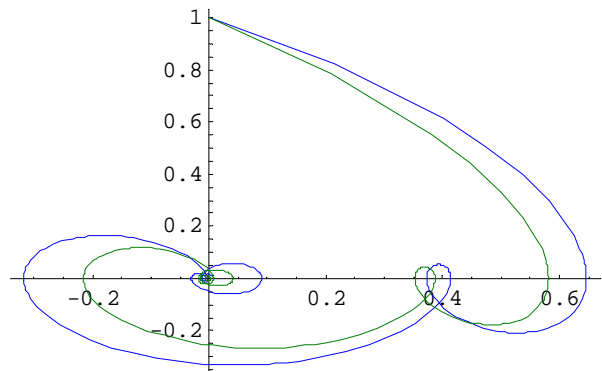


Figure 16. Parametric plots of the Resistor1 current against its derivative, both original and smoothed.

4.2.2 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
           [Modell1]
{Resistor1.p.v, Resistor1.p.i,
 Resistor1.n.v, Resistor1.n.i,
 ... ..,
 Inertia2.flange_a.tau}

```

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

```

eqn2 = Eliminate[eqn, connvars]
der[Inertial.phi] == Inertial.w &
der[Inertial.w] == Inertial.a &&
... ..
Inertia2.flange_b.tau == 0 &
der(-1)[EMF1.w] == Inertia2.phi -
  Spring1.phi_rel

```

4.3 Symbolic Laplace Transformation.

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: $der^{(-1)}[a_]\rightarrow\frac{a}{s}$, $der[b_]\rightarrow sb$.

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 (Inertial.phi) == Inertial.w &
s (Inertial.w) == Inertial.a &&
... ..
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, Inductor1.L -> L,
 EMF1.k -> k, Inertial.J -> J1,
 Spring1.c -> c1, Spring1.phi_rel0 -> 0,
 Inertia2.J -> J2};

```

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

```

eq4 =
Eliminate[eq3,
Complement[
GetFlatNonConnectionVariables[Modell1],
{Inertia2.w}]] /. shortnames
(k c1 (ConstantVoltage1.V) ==
 k2 c1 (Inertia2.w) +
... ..
 R s3 J1 J2 (Inertia2.w) +
 L s4 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`:

$$H[s_]=\text{First}\left[\frac{\text{Inertia2.w}}{\text{ConstantVoltage1.V}}/\text{Solve}[\text{eq4},\text{Inertia2.w}]\right]$$

$$(k c_1) / (k^2 c_1 + R s c_1 J_1 + L s^2 c_1 J_1 + k^2 s^2 J_2 + R s c_1 J_2 + L s^2 c_1 J_2 + R s^3 J_1 J_2 + L s^4 J_1 J_2)$$

4.4 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`.

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.Electrical.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,
  ... ..
  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 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;
```

5 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 would like to thank Peter Bunus for inspiration and great help in MicroSoft Word formatting and conversion from notebook format when preparing this paper, and Dan Costello for Word advice. Acknowledgements to the following individuals for contributions 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 atmosphere at MathCore AB. Acknowledgements also to the members of the Modelica Association for creating the

Modelica language, and to EU under the RealSim project for supporting part of the development of MathModelica.

References

- [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.
- [Bunus-00] Peter Bunus, Vadim Engelson, Peter Fritzon. Mechanical Models Translation and Simulation in Modelica. In *Proceedings of Modelica Workshop 2000*. Lund University, Lund, Sweden, Oct 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 Fritzon. 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.
- [Fritzon-83] Peter Fritzon. Symbolic Debugging through Incremental Compilation in an Integrated Environment. *The Journal of Systems and Software*, 3, 285-294, (1983).
- [Fritzon-92a] Peter Fritzon, Dag Fritzon. 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.
- [Fritzon-92b] Peter Fritzon, Lars Viklund, Johan Herber, Dag Fritzon: 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.
- [Fritzon-95] Peter Fritzon, Lars Viklund, Dag Fritzon, Johan Herber. High Level Mathematical Modeling and Programming in Scientific Computing. *IEEE Software*, pp. 77-87, July 1995.
- [Fritzon-98] Peter Fritzon 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.
- [Fritzon-98b] Peter Fritzon, 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 Fritzon. 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-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.
- [Otter-95] Martin Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*, Dissertation, Fortschrittberichte 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.
- [Saldamli-01] Levon Saldamli, Peter Fritzon. 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. *Introduction 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.