MathModelica—An Object-Oriented Mathematical Modeling and Simulation Environment

Peter Fritzson

MathModelica is an integrated interactive development environment for advanced object-oriented 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 typesetting, and symbolic formula manipulation provided via Mathematica. The user interface consists of a graphical Model Editor and Mathematica 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 multibody systems. The accessible MathModelica internal form allows the user to extend the system with new functionality, as well as to perform 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 [1] 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, and 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, and 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 environment with applications in modeling and simulation. Thus, it should fulfill the requirements from both general integrated programming environments and 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 and 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 a programming environment should accomplish 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
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 and 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 [2], 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 top-down or bottom-up programming methodology. 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 advantageous 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. This means that there is a well-defined representation of programs as data structures in 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, such as

- 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 [3] and the ObjectMath environment [4, 5]) and many years of intensive use of advanced integrated interactive environments such as the Interlisp system [2, 6, 7], and Mathematica [8, 9]. The Interlisp system was actually one of the first really powerful integrated environments and still exceeds 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 [10] (e.g., before the Smalltalk environment [11] and the Macintosh window system appeared).

Mathematica, which has been available since 1988, is an 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 so in programming support facilities.

1.3. Mathematica, Modelica, and MathModelica

It turns out that Mathematica is an integrated programming environment that fulfills 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 [12, 13, 14, 15, 16, 17, 18, 19].
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. 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 and symbolic transformations.

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 also available for ordinary Mathematica programming, as well as making some of the Mathematica language constructs available within code for simulation models.

The current MathModelica system builds on experience from the design of the ObjectMath [4, 5] modeling language and environment, early Modelica prototypes [20, 21], as well as results from object-oriented modeling languages and systems such as Dymola [22, 23] and Omola [24, 25], which together with ObjectMath and a few other object-oriented modeling languages (e.g., [26, 27, 28, 29, 30]) 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 in 1989 and concluded in the fall of 1996 when the Modelica Design Group was started and later renamed the 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 Mathematica-Style Modelica Language for Simulation

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

Modelica models are built from classes. Like other object-oriented languages, a class contains variables (i.e., class attributes representing data). The main difference compared to traditional object-oriented languages is that instead of functions (methods) 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 \texttt{Connect} equation construct. The equation (although it has function call syntax) \texttt{Connect[v1,v2]} expresses coupling between the variables \( v_1 \) and \( v_2 \). These variables are instances of connector classes and are attributes of the connected objects. This gives a flexible way of specifying topology of physical systems described in an object-oriented way using Modelica.

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

\section{2.1. Connection Diagrams}

As an introduction to the Modelica language 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.](image-url)
A declaration like the following specifies R1 to be an object or instance of the class `Resistor` and sets the default value of the resistance parameter, R, to 10, shown in *Mathematica*-style *Modelica* syntax.

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

A *Mathematica*-style *Modelica* description of the complete circuit appears as follows.

```
Model[Circuit,  /* Circuit model in Mathematica-style syntax */
       Resistor  R1[R=10];
       Capacitor C[C=0.01];
       Resistor R2[R=100];
       Inductor L[L=0.1]];
VsourceAC AC;
Ground G;
Equation[
   Connect[AC.p, R1.p];  (* Capacitor circuit*)
   Connect[R1.n, C.p];
   Connect[C.n, AC.n];
   Connect[R1.p, R2.p];  (* Inductor circuit*)
   Connect[R2.n, L.p];
   Connect[L.n, C.n];
   Connect[AC.n, G.p];   (* Ground *)
]
```

The same Circuit model in standard *Modelica*-style syntax appears in the following. The relation between these two syntactic forms is explained in Section 3.4, including a discussion regarding our choice of *Mathematica* syntax for certain *Modelica* language features. For the rest of this article, we will primarily use *Mathematica*-style syntax for examples and *Modelica* language constructs.

```
model Circuit  /* Circuit model in Modelica-style syntax */
   Resistor R1(R=10);
   Capacitor C(C=0.01);
   Resistor R2(R=100);
   Inductor L(L=0.1);
   VsourceAC AC;
   Ground G;
   equation
      connect(AC.p, R1.p);  // Capacitor circuit
      connect(R1.n, C.p);
      connect(C.n, AC.n);
      connect(R1.p, R2.p);  // Inductor circuit
      connect(R2.n, L.p);
      connect(L.n, C.n);
      connect(AC.n, G.p);   // Ground
end Circuit;
```

A composite model like the circuit model described earlier 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 Connect construct is a special equation form that generates standard 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 boolean, integer, real, 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Either True or False</td>
</tr>
<tr>
<td>Integer</td>
<td>Corresponding to the C int data type, usually 32-bit two’s complement</td>
</tr>
<tr>
<td>Real</td>
<td>Corresponding to the C double data type, usually 64-bit floating-point</td>
</tr>
<tr>
<td>String</td>
<td>String of 8-bit characters</td>
</tr>
<tr>
<td>enumeration</td>
<td>Enumeration literal constants of the corresponding enumeration type</td>
</tr>
</tbody>
</table>

Table 1. Predefined data types in Modelica.

It is possible to define new user-defined types.

Type[name, annotation, type]

Here is an example defining a temperature measured in Kelvin, K, which is of type Real with the minimum value zero.

Type[Temperature, "temperature measured in Kelvin", Real[{Unit => "K", Min => 0}]]

Here the user-defined types of Voltage and Current are defined.

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

This defines the symbol Voltage to be a specialization of the type Real that 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 previous case, the unit of measure of Voltage is changed to "V". Here is a corresponding definition made for Current.

Type[Current, Real[{Unit => "A"}]]
In *Modelica*, 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 `inoutBlock`. Moreover, functions and packages are regarded as special kinds of restricted and enhanced classes, denoted by the keywords `modelicaFunction` 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, and generic properties are identical to all kinds of restricted classes. Furthermore, the construction of *MathModelica* translators is simplified considerably because only the syntax and semantics of the `class` construct 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 that 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.

**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 a “potential” nonflow variable, and the current, `i`, as a flow variable by being prefixed by the keyword `flow`. This implies that voltages will be set equal when two or more components are connected (i.e., `v1 = v2 = ... = vn`) and currents are summed to zero at the connection point, `i1 + i2 + ... + in = 0`.

```plaintext
Connector[Pin, Voltage v; Flow Current i
}
```

Connect equations are used to connect instances of connector classes. A `connect` equation `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 standard equations, namely:

\[
\begin{align*}
\text{Pin1}.v & = \text{Pin2}.v \\
\text{Pin1}.i + \text{Pin2}.i & = 0
\end{align*}
\]

The first equation says that the voltages of the connected wire ends are the same (i.e., \(v_1 = v_2 = \cdots = v_n\)). 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), \(i_1 + i_2 + \cdots + i_n = 0\). 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 (Abstract) Classes

A useful strategy for reuse in object-oriented modeling is to try to capture common properties in superclasses that can be inherited by more specialized classes. For example, a common property of many electrical components such as resistors, capacitors, inductors, and voltage sources, 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., abstract in standard object-oriented terminology) since it does not specify all properties needed to instantiate the class.

```mathematica
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
  ]
]
```

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 a 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., \(i = p.i\)).
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 and 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 of 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 following Resistor class.

\[ R \cdot i = v \]

This equation can be used in three ways. The variable \( v \) can be computed as a function of \( i \), the variable \( i \) can be computed as a function of \( v \), or the resistance \( R \) can be computed as a function of \( v \) and \( i \), as shown in these three statements.

\[
\begin{align*}
i &= v/R \\
v &= R \cdot i \\
R &= v/i
\end{align*}
\]

In the same way, consider the following equation from the class TwoPin.

\[ v = p \cdot v - n \cdot v \]
This equation gives rise to one of the following three assignment statements, 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 following Resistor example to explain *inheritance, parameters, and constants*.

The Resistor inherits TwoPin using the Extends construct (inspired from `extends` in Java). A model parameter, \( R \), is defined for the resistance and is used to state the constitutive equation for an ideal resistor, namely \( \text{Ohm's Law}: v = R * i \). 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.

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

The keyword Parameter specifies that the variable is constant during a simulation run, but can change values between runs. This means that a model 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 by changing the parameter value. There are also Modelica constants that never change and can be substituted inline, which are specified by the keyword Constant. Here are additional examples of constants and parameters with default values defined via so-called declaration equations that appear in declarations.

```plaintext
Constant Real c0 = 2.99792458 10^8;
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 later via input to a model. Thus a parameter can be declared without a declaration equation. For example:

```plaintext
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 equations in an 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 into a class, equations from those classes are copied into the class. This makes the equation-based semantics of the child classes consistent with the semantics of the parent class since the equation constraints of the parent class are fulfilled.

2.7. Time and Model Dynamics

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

The following classes defined for electric voltage sources, capacitors, and inductors have dynamic time-dependent behavior and can also reuse the TwoPin superclass. In the differential equations in the classes Capacitor and Inductor, the forms \( v' \) 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 \( C v'(t) = i(t) \) for all values of \( t \).

**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 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 \( \pi \) defined in the Modelica Standard Library, just to demonstrate the declaration of a constant. The input voltage \( v \) is defined by \( v = VA \cdot \sin(2 \pi f \cdot time) \). Note that time is a built-in Modelica primitive.

```model
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]
  ]
]
```
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, \( \frac{dv}{dt} = \frac{i}{C} \).

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

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, \( L \frac{di}{dt} = v \).

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

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.

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

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 using Connect.
Figure 3. Connection diagram of the electric circuit (repeat of Figure 1).

Here we reproduce the Circuit model.

```mathematica
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]
  ]
]
```

**Simulation**

We simulate the model with the default initial values and parameter settings in the range $0 \leq t \leq 0.1$. 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.

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

Let us plot the current in the inductor for the first 0.1 seconds.
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 and parameter values, and not the structure of the problem.

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

Here is a plot that shows the result. Note the differences in initial current and amplitude due to the changed inductance.

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

2.9. The MathModelica Notion of Subtypes

The notion of subtyping in Modelica is influenced by a type theory of Abadi and Cardelli [31]. The notion of inheritance in Modelica is independent of 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.

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

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 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 definitions for the 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 following specialized equation from TempResistor:

\[ v = i * (R + RT * (T - Tref)) \]

and the general equation from class Resistor:

\[ v = R * i \]

are incompatible. MathModelica currently does not support explicitly named equations and replacement of equations, except for the cases when the equations are collected into local class, or when a declaration equation occurs as part of a variable declaration.

## 2.10. Class Parameterization

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]
]
]

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 that defines RedefinedSimpleCircuit to be a special variant of SimpleCircuit.

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-time dynamics.

On the other hand, it is sometimes beneficial to make the approximation that certain system components display discrete-time behavior (i.e., changes of values of system variables over time may occur instantaneously and discontinuously). In a 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. 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 by several orders of magnitude.
Since the discrete approximation can only 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 and conditional equations—to describe discontinuous and conditional models. When-equations are a particular kind of conditional equation, here instantaneous equations, that express equations that are only valid at instants in time—at discontinuities—when certain conditions become true. If[cond, then-part, else-part] is the Modelica form of conditional expressions that allows modeling of phenomena with different expressions in different operating regions, as seen in the following equation describing a limiter.

\[ y = \text{If}[u > \text{limit}, \text{limit}, u] \]

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.](image4.png)

![Figure 5. Ideal diode characteristic.](image5.png)
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 \( v \) and the current \( i \), here the same as \( i_1 \), are functions of the parameter \( s \). When the diode is \textit{off}, no current flows and the voltage is negative; whereas, when it is \textit{on}, there is no voltage drop over the diode and the current flows.

```mathematica
Model[Diode, "Ideal diode",
   Extends[TwoPin];
   Reals s;
   Boolean off;
   Equation[
      off == s < 0;
      If[off, (* two conditional equations *)
         u == s,
         u == 0
      ];
      i = If[off, 0, s] (* conditional expression *)
   ]
]
```

When-equations have been introduced in \textit{MathModelica} 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 \textit{When}-equations for the case of a vector of conditions is shown as follows. The equations in the \textit{When}-equation are activated when at least one of the conditions become True. A single condition is also possible.

```mathematica
when[ {condition1, condition2, \ldots},
   <equations>
   ]
```

A bouncing ball is a good example of a hybrid system for which the \textit{When}-equation is appropriate when modeled. The motion of the ball is characterized by the variable \textit{height} above the ground and the vertical \textit{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 following modeled one, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.
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., \( \text{velocity (after bounce)} = -c \times \text{velocity (before bounce)} \)) which is accomplished by the special syntactic form of instantaneous equation: \( \text{Reinit[velocity, -c \ Pre[velocity]]} \).

Example simulations of the bouncing ball model can be found 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 everyday 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, as in Figure 7, 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 \textit{time} to each event. However, such an ordering is \textit{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.

\begin{center}
\begin{tabular}{c}
\text{event 1} & \text{event 2} & \text{event 3} \\
\end{tabular}
\end{center}

\textbf{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, such as a cocktail party event, a car collision event, or a voltage transition event in an electrical circuit, actually exist. 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 arriving at declarative models about what is, rather than imperative recipes 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 an event is an approximation compared to real events. For example, events in Modelica take \textit{no time}—this is the most important abstraction of the \textit{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 \textit{point} in time that is instantaneous, i.e., has zero duration.
- An \textit{event condition} that switches from \texttt{False} to \texttt{True} for the event to happen.
- A set of \textit{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 are a special case of conditional equations that are only active at events.

**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.

![Image of two variables, one continuous-time and one discrete-time](image)

**Figure 8.** Example graphs of continuous-time and discrete-time variables.

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.
These subsystems are the following.

- The graphic Model Editor used for designing models from library components.
- The interactive Notebook facility used for literate programming, documentation, running simulations, scripting, graphics, and symbolic mathematics with Mathematica.
- The Simulation Center used 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 also integrates and extends several professional software products that are included in the three subsystems. For example, one version of the simulation kernel includes simulation algorithms from Dynasim [23, 32], and the notebook facility includes the technical Mathematica computing system [9].

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 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.](image)

The Model Editor can be viewed as a user interface for graphical programming in Modelica. Its basic functionality consists of selecting components from libraries, connecting 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 with a graphical user interface for running simulations, setting initial values and model parameters, plot results, etc. These facilities are accessible via a graphic user interface accessible through the simula-
tion window (see Figure 11). However, remember that it is also possible to run simulations from the textual user interface available in the notebooks. The simulation window currently (MathModelica in 2005) 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 times 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 initial (start) values, and the Message section allows viewing possible messages from the simulation process.

![Simulation Window](image)

**Figure 11.** The simulation window with plots of the signals Inertia1.flange_a.tau and Inertia1.w selected in the menus.
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*.

### 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 number of sample notebooks are shown in Figure 12.

The *Mathematica* notebook facility is actually an interactive What-You-See-Is-What-You-Get (WYSIWYG) realization of *Literate Programming*, a form of programming where programs are integrated with documentation in the same document, originally proposed in [33]. A noninteractive prototype implementation of Literate Programming in combination with the document processing system *LaTeX* has been realized [34]. 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*.

![Figure 12. Examples of MathModelica notebooks.](image)
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 data in different 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.

**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 Mathematica notebooks used in the MathModelica system.

![Figure 13. The MyPackage package in a notebook.](image)

In the MathModelica system, Modelica packages including documentation and test cases are primarily stored as notebooks as in Figure 13. 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 export 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 textual contents in Modelica syntax.

```modelica
package MyPackage
model Class3
...
end Class3;
model Class2
...
end Class2
model Class1
...
end Class1
package MySubPackage
model Class1
...
end Class1;
end MySubPackage;
end MyPackage;
```

This can be expressed in the Mathematica style of Modelica syntax as:

```mathematica
Package[MyPackage,
    Model[Class3, ...];
    Model[Class2, ...];
    Model[Class1, ...];
    Package[MySubPackage,
        Model[Class1, ...];
    ]
]
```

**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* used as comments to executable formal model specifications.

- *Dialogue forms* for specification and modification of input data.

- *Result tables* whose results can be automatically represented in (live) tables, which can even be automatically updated after recomputation.
Graphical result representation 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 article.

**Mathematics with 2D-Syntax, Greek Letters, and Equations**

MathModelica uses the syntactic facilities of Mathematica to allow writing formulas in the standard well-known mathematical notation 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, such as $x$ and $f(x)$, means multiplication just as in traditional mathematics notation). 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, which is discussed later in this section.

- 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 following cell contains examples of both implied multiplication and two-dimensional integral syntax. The cell style is called MathModelica input form (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. For example, this is 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 previous 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 aforemen-
tioned syntactic ambiguities can be avoided if the formula is first entered using one of the earlier input forms and then converted to TraditionalForm.

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

The MathModelica environment allows easy conversion between these forms using keyboard characters or menu items. In the following, we show a small example of a Modelica model class called 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 and in a MathModelica derivative with respect to time, corresponding to the Modelica \texttt{der()} operator. The initial conditions of the variables, if not explicitly specified in the declarations, are the default start-attribute values of the variables, which is zero for Real variables. Initial conditions for variables and derivatives can also be given in a special \	exttt{InitialEquation} section.

\begin{verbatim}
Model[SimpleDAE,
Real \( \beta_1 \);
Real \( x_2 \);
Equation[
\( \frac{\beta_1 }{1 + (\beta_1 )^2} \cdot \frac{\sin(x_2 )}{1 + (\beta_1 )^2} + \beta_1 \cdot x_2 + \beta_1 = 1; \)
\( \frac{\sin(\beta_1 )}{1 + (\beta_1 )^2} - \frac{x_2 }{1 + (\beta_1 )^2} - 2 \cdot \beta_1 \cdot x_2 + \beta_1 = 0; \)
]
\end{verbatim}

We simulate the model for ten seconds by giving a \texttt{Simulate} command.

\begin{verbatim}
Simulate[SimpleDAE, \{t, 0, 10\}];
\end{verbatim}

We use the command \texttt{PlotSimulation} for plotting the solutions for the two state variables, which of course are both functions of time, here denoted by \( t \) in Mathematica syntax.

\begin{verbatim}
PlotSimulation[\{\( \beta_1 \cdot [t] \), \( x_2 \cdot [t] \}\}, \{t, 0, 10\}]
\end{verbatim}
A phase plane plot appears as follows.

\[
\text{ParametricPlotSimulation}[\{\dot{x_1}[t], x_2[t]\}, \{t, 0, 10\}]
\]

3.4. Environment and Language Extensibility

Programming environments need to be flexible to adapt to changing user needs. Without flexibility, a programming tool becomes too hard to use for practical needs and therefore obsolete. 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 language for programmability).
- Extension of language and notation (e.g., by facilities to add new syntactic constructs and new notation or extending 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.

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 article. Another possibility is 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 for commands, which are sent to the Mathematica interpreter instead of the simulator.
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.

Mathematica versus 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 Modelica 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 later. Note the use of the simple prefix syntax: a keyword followed by square brackets surrounding the contents of the construct. All reserved words, and by convention also predefined functions, and types in MathModelica start with an upper-case letter just as in Mathematica. Equation equality is represented by the == operator 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 combine more than one declaration, statement, or expression.

Note that the Start attribute values, 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
  ]
]
```

**Choice of Mathematica Representation of Modelica Syntax**

In order to represent *Modelica* in a *Mathematica*-compatible way and thereby potentially extend the *Mathematica* language, certain choices were made during the design and implementation of the *MathModelica* environment. Some of these design tradeoffs were not easy, and a number of variants have been implemented and tried out. Here we give a short motivation for our design in a few tricky cases.

- **Type prefixes.** In declarations type prefixes are separated from the declared entity using one or more space characters. This happens to be standard notation in common programming languages such as Java, C++, or C. This choice is unambiguous also in *Mathematica* since the usual *Mathematica* interpretation of space as a multiplication operator is not valid in a type declaration and is instead interpreted as a type prefix operation. Such type prefixes are represented using the `Prefixed[]` node that is always unparsed into space. Space is still parsed as implied multiplication in the usual expression context. Our implementation prevents the usual evaluation of declarations, which must be prohibited anyway regardless of the syntax, and converts multiplications in the type declaration prefix parts to `Prefixed[]` nodes.

  Some people have suggested using the *Mathematica* `@` operator to represent such prefixes. This is however ambiguous since `@` is equivalent to function application. At unparsing it would be hard to know whether `Real@x` should be unparsed as `Real@x` or `Real[x]`. Moreover, the ambiguity is really serious and unsolvable in some cases such as array declarations, for example, the array declarations inside `WaveEquation`'s `Sample` and `initialPressure` in Section 4.4.

- **Dot notation.** Accessing the member of an object using dot notation (.), for example, `person.length` is an almost universally accepted notation in object-oriented programming languages. However, this clashes with the (rather rare) use of . in *Mathematica* for inner product as the `Dot[]` operator. We experimented with several different operators, including the use of backtick (``). However, all these attempts eventually failed because of inappropriate priority and precedence of other operators. For example, assuming `persons` is an array of person records, `persons[[2]].length` accesses the `length` field of the second record.
However, when using backtick, `persons[[2]]` length is incorrectly parsed into implied multiplication of persons[[2]] and `length` due to the high priority of backtick. Therefore, MathModelica redefines . to be parsed into Member[] nodes, that is, a.b means Member[a,b]. To perform dot product between a and b, use the FullForm Dot[a,b].

- **Underscore.** Most programming languages, including Modelica, allow underscore in identifiers. This is not possible in Mathematica since underscore is mapped to the Blank[] pattern-matching operator. Our solution is to use the Mathematica `\[UnderBracket1` character (___), which is very similar to underscore in appearance. For example, see its use in the design optimization model in Section 4.2 with the `idealGear`.R2T1.flange___b variable.

### 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 the 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 or 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 MultiBody System (MBS) library [35, 36]. Such translators integrated with the simulation environment have so far been developed for the two CAD systems SolidWorks [37] and AutoDesk’s Mechanical Desktop [38].

We have also developed an OpenGL-based 3D visualizer and animation system called MVIS (Modelica VISualizer) [39] that provides an online dynamic display of the mechanical assembly during simulation, or an offline display based on saved state information for each time step.

Both the MVIS visualizer and the CAD translators are separate subsystems that 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.
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. The output can subsequently be visualized as a data plot of the system variables and/or as a 3D or 2D dynamic model animation. The 3D visualizations 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 visualization 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 is shown in Figure 15.

Figure 15 shows a mechanical model designed in the AutoDesk’s 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 [40] for use by the MVIS visualizer. At the same time, the mass and inertia of the parts are extracted 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 multidomain 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.

We might ask why develop yet another tool for multibody simulation of mechanical systems? There are already commercially available MBS simulation packages like ADAMS or WorkingModel3D. Many CAD systems are integrated with some multibody simulation tool. However, the primary limitation of these environments is the difficulty of integrating multidomain 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:

---

**Figure 15.** The path from a static CAD model to a dynamic system simulation and visualization.
The need to integrate multidomain 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) for further computations that may include simulation results in expressions, and for both simple and advanced plotting. The following simple bouncing ball model [12] is used in the simulation and plotting examples.

```plaintext
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{
    height' := velocity;
    velocity' := -g;
    When[height ≤ radius,
      Reinit[velocity, -c Pre[velocity]]
    ]
  }
]```

Peter Fritzson
Interpolating Function Representation of Simulation Results

The following simulation of the previous BouncingBall model is done for a short time period using very few points.

```math
res1 = Simulate[BouncingBall, \{t, 0, 0.5\}, NumberOfIntervals → 10]
<SimulationData: BouncingBall : 2005-10-21 16:32:
  17.5845168 : \{c, Derivative[1], Derivative[1], g, height, radius, 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 a descriptor are shown as the result of the previous 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 done 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 follows, but is called automatically on any variable from the last simulation when referenced.

```math
h = VariableTrajectory[height]
  Function[\{$'t$\120667\}, Which[\{$'t$\120667 < 0.428353, InterpolatingFunction[\{0., 0.428353\}, \{\}]],
                \{$'t$\120667 ≥ 0.428353, InterpolatingFunction[\{0.428353, 0.5\}, \{\}]]\]]
```

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 consists 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 use interpolation order 3 as the 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.

```math
h'[0.2]
```

-1.962
In the previous case *Mathematica* made the differentiation of the Interpolating Function 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 \( \text{VariableTrajectory} \) is automatically applied. Therefore, instead of assigning the variable \( h \) as done earlier, we can write the following and get the same result.

\[
\text{height}'[0.2] = -1.962
\]

Note, to keep variable values from previous simulations accessible, we should use \( \text{VariableTrajectory} \) on the appropriate variable, specifying the desired descriptor (e.g., \( \text{res1} \)).

\[
\text{h = VariableTrajectory}[\text{height, SimulationResult} \rightarrow \text{res1}];
\]

Now we perform a new simulation, with the result denoted by \( \text{res2} \).

\[
\text{res2 = Simulate}[\text{BouncingBall, \{t, 0, 0.5\}},
\text{NumberOfIntervals} \rightarrow 10, \text{ParameterValues} \rightarrow c \rightarrow 0.95];
\]

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 \( \text{height}[t] - h[t] \)).

\section*{PlotSimulation}

First, we simulate the bouncing ball for eight seconds and store the results in the variable \( \text{res1} \) for subsequent use in the plotting examples.

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

The command \( \text{PlotSimulation} \) is used for simple standard plots. If nothing else is specified (i.e., by the optional \( \text{SimulationResult} \) parameter), the command refers to the results from the most recent simulation. In the following diagram 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 \( \text{PlotJoined} \) has been set to \text{False} to create a dotted plot.
PlotSimulation[height[t], {t, 0, 8}, PlotJoined → False]

![Graph](image1)

**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[Exp[-Cos[height[t]]], {t, 0, 8}]

![Graph](image2)

**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] + \sqrt{3}, Abs[velocity[t]]}, {t, 0, 8}]

Figure 18. Plot of arbitrary functions in the same diagram.

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

\[
res2 = \text{Simulate}[\text{BouncingBall}, \{t, 0, 8\}, \text{ParameterValues} \rightarrow c \rightarrow 0.95];
\]

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 \(Show\).

\[
gr1 = \text{PlotSimulation}[\text{height}[t], \{t, 0, 8\}, \text{SimulationResult} \rightarrow res1, \text{DisplayFunction} \rightarrow \text{Identity}];
gr2 = \text{PlotSimulation}[\text{height}[t], \{t, 0, 8\}, \text{SimulationResult} \rightarrow res2, \text{DisplayFunction} \rightarrow \text{Identity}];
\]

\(\text{Show}[\text{GraphicsArray}[[\text{gr1}, \text{gr2}]], \text{DisplayFunction} \rightarrow \$\text{DisplayFunction}]\)

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.

\[
\text{PlotSimulation}[\text{height}[t], \{t, 0, 8\}, \text{SimulationResult} \to \{\text{res1}, \text{res2}\}]
\]

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

The plot colors are specified by the $\text{PlotSimulationColors}$ predefined variable.

\[
\text{PlotSimulationColors}
\]

\[
\{\text{RGBColor}[0, 0, 1], \text{RGBColor}[0, 0.5, 0], \text{RGBColor}[1, 0, 0], \\
\text{RGBColor}[0, 0.75, 0.75], \text{RGBColor}[0.75, 0, 0.75], \\
\text{RGBColor}[0.75, 0.75, 0], \text{RGBColor}[0.25, 0.25, 0.25]\}
\]

Here we check the color scheme of $\text{PlotSimulationColors}$.

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

![Figure 21. Plot of multiple curves with different colors.]

**ParametricPlotSimulation**

Parametric plots can be done using ParametricPlotSimulation.

\[
\text{ParametricPlotSimulation}[\{\text{height}[t], \text{velocity}[t]\}, \{t, 0, 8\}]
\]

**Figure 22.** A parametric plot.

In the same way as PlotSimulation, the ParametricPlotSimulation function can handle several plots.

\[
\text{ParametricPlotSimulation[
    \{\{\text{height}[t], \text{velocity}[t]\}, \{\text{velocity}[t], \text{height}[t]\}\}, \{t, 0, 8\}]
\]

**Figure 23.** Multiple parametric plots in the same diagram.

ParametricPlotSimulation can also handle results from 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.

**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’s work in chemical kinetics. The system is described by three coupled nonlinear differential equations.

\[
\begin{align*}
\frac{dx}{dt} &= -y - x \\
\frac{dy}{dt} &= x + \alpha y \\
\frac{dz}{dt} &= \beta + (x - \gamma)z.
\end{align*}
\]

Here \(\alpha\), \(\beta\), and \(\gamma\) 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 \(x[\{\text{Start} == 1\}]\);
  Real \(y[\{\text{Start} == 3\}]\);
  Real \(z[\{\text{Start} == 0\}]\);
  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.

\[
\text{Simulate[Rossler, \{t, 0, 40\}, InitialValues \rightarrow \{x = 2, y = 2.5, z = 0\}];}
\]

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

\[
\text{ParametricPlotSimulation3D[}
\{x[t], y[t], z[t]\}, \{t, 0, 40\}, \text{AxesLabel \rightarrow \{X, Y, Z\}}
\]

Figure 25. 3D 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. 3D 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.
4.2. Design Optimization

Let us describe an example of how the powerful Mathematica scripting language available within MathModelica can be utilized to solve nontrivial 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 anchoring point.

Here is the corresponding model code in Mathematica-style syntax (the squares contain graphical annotations for icons, lines, and so on).

\[
\text{Model[LinearActuator,}
\]
\[
\quad\text{Modelica.Mechanics.Translational.SlidingMass}
\]
\[
\quad\text{slidingMass1[m == 0.5]}\]
\[
\quad\text{Modelica.Mechanics.Translational.SpringDamper}
\]
\[
\quad\text{springDamper1[d == 3, c == 20]}\]
\[
\quad\text{Modelica.Mechanics.Translational.Fixed fixed1}\]
\[
\quad\text{Modelica.Mechanics.Rotational.IdealGearR2T idealGearR2T1}\]
\[
\quad\text{Modelica.Mechanics.Rotational.Inertia inertia1[J == 0.1]}\]
\[
\quad\text{Modelica.Mechanics.Rotational.SpringDamper}
\]
\[
\quad\text{springDamper2[c == 15, d == 2]}\]
\[
\quad\text{Modelica.Mechanics.Rotational.Inertia inertia2[J == 0.1]}\]
\[
\quad\text{Modelica.Mechanics.Rotational.Torque torque1}\]
\[
\quad\text{Modelica.Blocks.Sources.Step step1}\]
\[
\]
As a comparison, we also show the model code in Modelica textual syntax:

```model LinearActuator
  Modelica.Mechanics.Translational.SlidingMass slidingMass1(m=0.5);
  Modelica.Mechanics.Translational.SpringDamper springDamper1(d=3, c=20);
  Modelica.Mechanics.Translational.Fixed fixed1;
  Modelica.Mechanics.Rotational.IdealGearR2T idealGearR2T1;
  Modelica.Mechanics.Rotational.Inertia inertial1(J=0.1);
  Modelica.Mechanics.Rotational.SpringDamper springDamper2(c=15, d=2);
  Modelica.Mechanics.Rotational.Inertia inertial2(J=0.1);
  Modelica.Mechanics.Rotational.Torque torque1;
  Modelica.Blocks.Sources.Step step1;

  equation
    connect(inertial1.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(inertial1.flange_a, springDamper2.flange_b);
    connect(springDamper2.flange_a, inertial2.flange_b);
    connect(inertial2.flange_a, torque1.flange_b);
    connect(torque1.tau, step1.y);
end LinearActuator;
```

The small square boxes in the code represent graphical annotations about the visual appearance of the components of the model, as seen in Figure 28. These annotations are hidden behind boxes; otherwise, the code becomes harder to read.

Here 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 the step response from the linear actuator.](image-url)
Assume that we have some freedom in choosing the damping in the translational spring-damper. A number of simulation runs shows what kind of behavior we have for different values of the damping parameter \( d \). The Mathematica Table function is used with Simulate to collect the results into an array \( \text{res} \). This array then contains the results from simulations of \text{LinearActuator} with a damping of 2 to 14 with a step size of 2 (i.e., seven simulations are performed).

\[
\text{res} = \text{Table}[\text{Simulate}[\text{LinearActuator}, \{t, 0, 4\}, \text{ParameterValues} \to \{\text{springDamper1.$d$} = s\}], \{s, 2, 15, 2\}];
\]

\[
\text{PlotSimulation}[\text{slidingMass1.$s$}[t], \{t, 0, 4\}, \text{SimulationResult} \to \text{res}, \text{Legend} \to \text{False}]
\]

Figure 30. 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 the following first-order system response, obtained by solving a first-order ODE using \text{NDSolve}:

\[
\text{Clear}[y]
\]

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

We make a comparison with the step response we simulated first (\( d = 2 \)) and the first-order system.
Now, let us make things a little more automatic. Simulate and compute the integral of the square error from $t = 0$ to $t = 4$.

$$
\text{res} = \text{Simulate}[\text{LinearActuator},
\{t, 0, 4\}, \text{ParameterValues} \rightarrow \{\text{springDamper1.d} = 3\};
\text{NIntegrate}[\text{First}[\{y(t)/.\text{res1}\] - \text{slidingMass1.s}[t]\}^2, \{t, 0, 4\}]
$$

0.000162508

Here we define a function, $f[a]$, doing the same thing as previously, but for different spring-damper parameter values $d = a$

$$
f[a_] := \text{Module}[\{\text{res}, t\}, 
\text{res} = \text{Simulate}[\text{LinearActuator},
\{t, 0, 4\}, \text{ParameterValues} \rightarrow \{\text{springDamper1.d} = a\};
\text{NIntegrate}[\text{First}[\{y(t)/.\text{res1}\] - \text{slidingMass1.s}[t]\}^2, \{t, 0, 4\}]]
$$

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

$$
\text{res2} = \text{Table}[\{a, f[a]\}, \{a, 2, 10, .5\}]
$$

{{2., 0.000317676}, {2.5, 0.000221485}, {3., 0.000162508}, {3.5, 0.000125518}, {4., 0.000102754}, {4.5, 0.000089821}, {5., 0.0000840739}, {5.5, 0.0000836789}, {6., 0.0000874671}, {6.5, 0.0000945833}, {7., 0.000104409}, {7.5, 0.000116474}, {8., 0.000130405}, {8.5, 0.000145933}, {9., 0.000162832}, {9.5, 0.000180907}, {10., 0.000200013}}

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

$$
f_{\text{pre}} = \text{Interpolation}[\text{res2}];
$$
The minimizing value of \( a \) can be computed using `FindMinimum`.

\[
\text{FindMinimum}[f_{pre}[a], \{a, 4\}]
\]

\[
\{0.0000832678, \{a \to 5.28665\}\}
\]

Here is a simulation with the optimal parameter value.

\[
\text{Simulate}[\text{LinearActuator}, \{t, 0, 4\},
\text{ParameterValues} \rightarrow \{\text{springDamper1.d} = 5.28665\}] ;
\]

Here is a plot comparing the first- and second-order system response with a plot of the squared error amplified by a factor of 100.

\[
\text{PlotSimulation}[\{\text{slidingMass1.s}[t], \text{y}[t] / . \text{res1},
100 (\text{slidingMass1.s}[t] - (\text{y}[t] / . \text{res1}))^2\}, \{t, 0, 4\}, \text{Legend} \rightarrow \text{False}]
\]

**Figure 33.** Comparison plot of the first- and second-order system step responses 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.

![Diagram of a WeakAxis model excited by a torque pulse train.](image)

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

Here is the corresponding Modelica code in Mathematica syntax.

```mathematica
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.y, torque1.tau]; 
  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;
```

The model code in Modelica syntax.

```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.y, torque1.tau); 
  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;
```
Here we simulate the model during 200 seconds.

\begin{verbatim}
Simulate[WeakAxis, {t, 0, 200}];
\end{verbatim}

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

\begin{verbatim}
PlotSimulation[{inertia3.w[t], torque1.t[t]}, {t, 0, 200}]
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure35.png}
\caption{Plot of the angular velocity of the rightmost axis segment of the WeakAxis model.}
\end{figure}

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.

\begin{verbatim}
data1 = Table[inertia3.w[t], {t, 0, 200, .25}];
\end{verbatim}

Here we compute the absolute values of the discrete Fourier transform of data1 with the mean value removed.

\begin{verbatim}
fdata1 = Abs[Fourier[data1 - Mean[data1]]];
\end{verbatim}

Here we plot the first 80 points of the data.

\begin{verbatim}
ListPlot[fdata1[[Range[80]]],
PlotStyle -> {Hue[1.0], PointSize[0.015]}]
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure36.png}
\caption{Plot of the data points of the Fourier transformed angular velocity.}
\end{figure}
It is easy to write a function \texttt{FourierPlot} that repeats the previous operations. \texttt{FourierPlot} also scales the axes such that amplitudes of trigonometric components are plotted against frequency (Hz).

\[
\text{FourierPlot}[\text{signal}_1, \{t, \text{tmin}, \text{tmax}\}, \text{T}, \text{options}___] := \\
\text{Module}[\{\text{data1}, n, \text{data2}, \text{fdata1}, \text{fdata2}, f\}, \\
\text{data1} = \text{Table[signal}, \{t, \text{tmin}, \text{tmax}, \text{T}\}]; \\
\text{n} = \text{Length[\text{data1}]; data2 = data1} - \text{Mean[\text{data1}];}
\]

\[
\text{fdata1} = \frac{2}{\sqrt{n}} \text{Abs[Fourier[\text{data2}]]}; \\
\text{fdata2} = \text{Drop[\text{fdata1}, -\text{Round[n/2.]}]]; \\
\text{f} = \text{Range[0, 1/(2+\text{T}), (1/(2 \times \text{T}) - 0)/\text{Round[n/2.]}]}; \\
\text{ListPlot[Transpose[\{f, \text{fdata2}\}], \text{options}] ]}
\]

\text{FourierPlot[inertia3.w[t]}, \{t, 0, 200\}, 0.5, \\
\text{PlotRange} \rightarrow \text{All, PlotJoined} \rightarrow \text{True, PlotStyle} \rightarrow \text{Hue[1.0]}]

![Figure 37. Plot of the curve of the Fourier transformed angular velocity.]

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

\[
\frac{1}{2\pi} \text{Eigenvalues}\left[\begin{array}{cccccc}
0 & 1 & 0 & 0 & 0 & 0 \\
-c_1 & 0 & -c_1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
-c_1 & 0 & -c_1 & -c_2 & 0 & -c_2 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & -c_2 & 0 & -c_2 & 0
\end{array}\right] / \{c_1 \rightarrow 0.7, c_2 \rightarrow 1\}\]

\text{Chop}

\{0.256077\,i, -0.256077\,i, 0.143344\,i, -0.143344\,i, 0, 0\}

These values, 0.256077, 0.143344, fit very well with the peaks in the previous 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 two-dimensional 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 \(-5 \leq x \leq 5\) describe its spatial dimension. Since Modelica can only 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

\[
\frac{\partial^2 p}{\partial x^2} \approx \frac{p_{i-1} + p_{i+1} - 2p_i}{\Delta x^2}.
\]

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

```mathematica
Model[WaveEquationSample,
    ModelicaImport[Modelica.SIunits];
Parameter SIunits.Length L == 10 "Length of duct";
Parameter Integer n == 30 "Number of sections";
Parameter SIunits.Length dL == L/FractionBarExt/FractionBarExt/FractionBarExt/FractionBarExt/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] == \[Exp\][-(\(\frac{L}{2}\))^2];
    p[n] == \[Exp\][-(\(\frac{L}{2}\))^2];
    dp == p';
    Do[
        dp[i+1] == \[FractionBar\][c^2 ((p[i+1] - 2 p[i] + p[i-1]))/dL^2], {i, Range[2, n-1]}
    ];
]
```

Here we are using a Modelica function `initialPressure` (defined as follows) 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 $e^{-x^2}$ (Figure 38).

```mathematica
Plot[e^{-x^2}, {x, -5, 5}, PlotRange -> All]
```

![Figure 38. The initial pressure profile of the duct.](image)

To provide an initial value profile (the start value of the vector variable $p$) for the pressure, we need to define a function that returns a sampled version of this profile. In principle this function could have been expressed in ordinary Mathematica function syntax with type extensions as in the MathCode system [41], but here we are using a more Modelica-like Mathematica syntax that is close to the syntax used for declaration of classes.

```modelica
ModelicaFunction[initialPressure,
  Input Integer n;
  Output Real p[n];
  Protected[
    Parameter Modelica.SIunits.Length L == 10
  ];
  Algorithm[
    Do[p[i] = exp[-(L/2 + (i-1)*L/n)^2], {i, Range[1, n]}]
  ]
]
```

Here we simulate the WaveEquationSample model.

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

The result is packed into a 2D interpolation function.

```mathematica
intp = Interpolation[Flatten[Table[
  {-5. + (k-1)/29*10, t, p[k][t]}, {k, 1, 30}, {t, 0, 10, .2}], 1]];
```

A 3D plot of the pressure distribution in the duct can be obtained as follows.
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.

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

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:

\[ \text{head} \left[ \text{term}_1, \ldots, \text{term}_n \right] \]

For example, an expression such as \( a + b \) is represented as \( \text{Plus}[a,b] \) in prefix form, also called FullForm syntax. A while loop is represented as the term \( \text{While}[\text{test}, \text{body}] \).

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 in the following. 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 and arbitrary arithmetic expressions composed from terms.

\[
\begin{align*}
\text{term}_1 ; \ldots ; \text{term}_n, & \quad \text{// Sequencing operator} \\
\{\text{term}_1, \ldots, \text{term}_n\}, & \quad \text{// Array/list constructor} \\
\text{term}_1 \text{ term}_2, & \quad \text{// Implied multiplication by space instead of \( * \)} \\
\text{term}_1 = \text{term}_2 & \quad \text{// Equation equality}
\end{align*}
\]

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 inside a model Arr.
This definition is stored internally in the MathModelicaFullForm format, which can be retrieved by calling the function `GetDefinition` that 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 except the array/list nodes shown as {...} instead of the prefix form List[...].

```
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 of 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.

- **Preservation of the syntactic structure of both Modelica and Mathematica code.** This means that the mapping from Modelica to MathModelicaFullForm 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 that 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 previous 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 parts of the MathModelica system.

Here we show a small model SecondOrderSystem in the different representations. First, the Modelica model is in the Mathematica-style Modelica syntax.

\[
\begin{align*}
\text{Model[SecondOrderSystem, } & \\
\text{Real } x[\{\text{Start }== 0\}]; & \\
\text{Real } xdot[\{\text{Start }== 0\}]; & \\
\text{Parameter Real } a == 1; & \\
\text{Equation[} & \\
\text{xdot }== x'; & \\
\text{xdot' + a }x' + x == 1 & \]
\end{align*}
\]

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

\[
\begin{align*}
\text{GetDefinition[SecondOrderSystem, Format } & \rightarrow \text{ModelicaForm]} \\
\text{model SecondOrderSystem} & \\
\text{Real } x(\text{start }== 0); & \\
\text{Real } xdot(\text{start }== 0); & \\
\text{parameter Real } a==1; & \\
\text{equation} & \\
\text{xdot }== \text{der}(x); & \\
\text{der}(xdot)+a*\text{der}(x)+x == 1; & \\
\text{end SecondOrderSystem;} & \\
\end{align*}
\]

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 returned in a Hold node to prevent symbolic evaluation.

\[
\begin{align*}
\text{GetDefinition[SecondOrderSystem, Format } & \rightarrow \text{MathModelicaForm]} \\
\text{Hold[Model[SecondOrderSystem, } & \\
\text{Real } x[\{\text{Start }== 0\}]; & \\
\text{Real } xdot[\{\text{Start }== 0\}]; & \\
\text{Parameter Real } a == 1; & \\
\text{Equation[} & \\
\text{xdot }== x'; & \\
\text{xdot' + a }x' + x == 1 & \]
\end{align*}
\]

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 (not done here).

\begin{verbatim}
GetDefinition[SecondOrderSystem, Format → MathModelicaFullForm]
Hold[SetType[SecondOrderSystem,
  TYPE[Model[Declaration[TYPE[Real, {}, {}],
    VariableComponent[x, Null, {}, {ElementModification[
      start, EqualBinding[0, {}, {}, Null]}
    ];
  Declaration[TYPE[Real, {}, {}, {}], VariableComponent[xdot, Null,
    {}, {ElementModification[start, EqualBinding[0, {}, {}, Null]}
    ];
  Declaration[TYPE[Real, {}, {Parameter}, {}], VariableComponent[a, EqualBinding[1, {}, {}], Null]];
  Equation[
    xdot == x';
    xdot' + a x' + x == 1
  ]], {}, {}, {}, {}, Null, Null]]
\end{verbatim}

\section{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.

\textbf{Definition and Simulation of Model1}

The example class Model1 has been drawn in the graphic Model Editor and imported into the following notebook.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{model1.png}
\caption{Connection diagram of Model1.}
\end{figure}

Here is the model code.
Model[Model1, 
	;
Modelica.Electrical.Analog.Basic.Inductor Inductor1;
Modelica.Electrical.Analog.Basic.Ground Ground1;
Modelica.Electrical.Analog.Basic.EMF EMF1;
	ConstantVoltage1;
Modelica.Mechanics.Rotational.Spring Spring1[{c == 0.5'}];
Modelica.Mechanics.Rotational.Inertia Inertia1;
Modelica.Mechanics.Rotational.Inertia Inertia2;
Equation[
  Connect[ConstantVoltage1.p, Resistor1.p];
  Connect[Resistor1.n, Inductor1.p];
  Connect[Inductor1.n, EMF1.p];
  Connect[EMF1.n, Ground1.p];
  Connect[ConstantVoltage1.n, Ground1.p];
  Connect[EMF1.flange_b, Inertia1.flange_a];
  Connect[Inertia1.flange_b, Spring1.flange_a];
  Connect[Spring1.flange_b, Inertia2.flange_a]
]
]

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

![Figure 42. Plot of the current-voltage product over Resistor1 in Model1.](image)

The second plot is parametric, where we plot the Resistor1 current against its derivative.
Some Symbolic Computations

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

```mathematica
eqn = GetFlatEquations[Model1];
Length[eqn]

Length[GetFlatVariables[Model1]]
```

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.

```mathematica
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, we obtain the connector variables from the flattened model.

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

```mathematica
```
Here we use the `Eliminate` function for symbolic elimination of some variables from the system of equations.

\[
eqn2 = \text{Eliminate}\left[\text{eqn}, \text{connvars}\right]
\]

\[
\begin{align*}
der[\text{Inertia1}.\phi] &= \text{Inertia1}.w \&\& \text{der}[\text{Inertia1}.w] = \text{Inertia1}.a \&\& 
der[\text{Inertia2}.\phi] &= \text{Inertia2}.w \&\& \text{der}[\text{Inertia2}.w] = \text{Inertia2}.a \&\& 
\text{ConstantVoltage1}.i &= -(\text{Resistor1}.i) \&\& 
\text{ConstantVoltage1}.v &= (\text{EMF1}.k) (\text{EMF1}.w) + \der[\text{Inductor1}.i] (\text{Inductor1}.L) + (\text{Resistor1}.i) (\text{Resistor1}.R) \&\& 
\text{ConstantVoltage1}.V &= (\text{EMF1}.k) (\text{EMF1}.w) + \der[\text{Inductor1}.i] (\text{Inductor1}.L) + (\text{Resistor1}.i) (\text{Resistor1}.R) \&\& 
\text{EMF1}.i &= \text{Resistor1}.i \&\& \text{EMF1}.v = (\text{EMF1}.k) (\text{EMF1}.w) \&\& 
\text{Inductor1}.i &= \text{Resistor1}.i \&\& 
\text{Inductor1}.v &= \der[\text{Inductor1}.i] (\text{Inductor1}.L) \&\& \text{(Inertia1.a)} (\text{Inertia1}.J) = (\text{EMF1}.k) (\text{Resistor1}.i) + \text{Spring1}.\tau \&\& 
\text{Inertia1}.\phi &= \text{Inertia2}.\phi - \text{Spring1}.\phi_{\text{rel}} \&\& 
\text{Inertia2}.a &= -(\text{Spring1}.\tau) \&\& 
\text{Resistor1}.v &= (\text{Resistor1}.i) (\text{Resistor1}.R) \&\& 
\text{Spring1}.c &= -(\text{Spring1}.\phi_{\text{rel}}) - \text{Spring1}.\tau \&\& 
\text{Inertia2}.\text{flange}_bh.\phi &= \text{Inertia2}.\phi \&\& 
\text{Inertia2}.\text{flange}_bh.\tau &= 0 \&\& 
der^{(-1)} \left[\text{EMF1}.w\right] &= \text{Inertia2}.\phi - \text{Spring1}.\phi_{\text{rel}}
\end{align*}
\]

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: \(\text{der}^{(-1)} [a_] \rightarrow \frac{a}{s}\), \(\text{der}[b_] \rightarrow b\). Note that \(\text{der}^{(-1)}\) is the inverse of taking a derivative (i.e., an integration operation). Note also that the second rule contains an implied multiplication.
Here we introduce short names for the model parameter to obtain a more concise symbolic notation.

\[
\text{shortnames} = \\
\{ \text{Resistor1.} \rightarrow \text{R}, \text{Inductor1.} \rightarrow \text{L}, \text{EMF1.} \rightarrow \text{k}, \text{Inertia1.} \rightarrow \text{J1}, \\
\text{Spring1.} \rightarrow \text{c1}, \text{Spring1.} \rightarrow \text{rel0} \rightarrow \text{0}, \text{Inertia2.} \rightarrow \text{J2} \};
\]

Then we derive the relation between \( \text{Inertia2.} \) and the input voltage. \[
\text{eq4} = \frac{\text{Eliminate[eq3, Complement[GetFlatNonConnectionVariables[Model1], \{\text{Inertia2.}\}]]}}{. \text{shortnames}} \\
\begin{align*}
(k \cdot c_1 (\text{ConstantVoltage1.}).V) &= \\
&= k^2 \cdot c_1 (\text{Inertia2.w}) + R \cdot s \cdot c_1 \cdot J_1 (\text{Inertia2.w}) + L \cdot s^2 \cdot c_1 \cdot J_1 (\text{Inertia2.w}) + \\
&= k^2 \cdot s^2 \cdot J_2 (\text{Inertia2.w}) + R \cdot s \cdot c_1 \cdot J_2 (\text{Inertia2.w}) + L \cdot s^2 \cdot c_1 \cdot J_2 (\text{Inertia2.w}) + \\
&= R \cdot s^3 \cdot J_1 J_2 (\text{Inertia2.w}) + L s^4 \cdot J_1 J_2 (\text{Inertia2.w}) && \text{s} \neq 0
\end{align*}
\]

The transfer function \( H \) is obtained by symbolically solving for \( \text{Inertia2.w} \) in the equation system \( \text{eq4} \) and using the obtained solution on a form \( \text{Inertia2.w} \rightarrow \text{expr} \) to eliminate \( \text{Inertia2.w} \), thus obtaining \( H \).

\[
H[\_] = \text{First}[\text{Inertia2.w} / \text{ConstantVoltage1.} \cdot \text{V}. \text{Solve[eq4, Inertia2.w]}]
\]

\[
(k \cdot c_1) / (k^2 \cdot c_1 + R \cdot s \cdot c_1 \cdot J_1 + L \cdot s^2 \cdot c_1 \cdot J_1 + \\
k^2 \cdot s^2 \cdot J_2 + R \cdot s \cdot c_1 \cdot J_2 + L s^2 \cdot c_1 \cdot J_2 + R s^3 \cdot J_1 J_2 + L s^4 \cdot J_1 J_2)
\]

**A Root Locus Computation**

Here a list of model parameter values is defined for subsequent use.
Here we compute the poles of the transfer function to obtain the root locus.

\[
\text{poles} = \text{CharacteristicRoots}[H[s], s]
\]

\[
\{\text{Root}[k^2 c_1 + R C J_1 #1 + R C J_2 #1 + L C J_1 #1^2 +
\quad k^2 J_2 #1^2 + L C J_2 #1^2 + R J J_1 J_2 #1^4 & , 1],
\text{Root}[k^2 c_1 + R C J_1 #1 + R C J_2 #1 + L C J_1 #1^2 + k^2 J_2 #1^2 +
\quad L C J_2 #1^2 + R J J_1 J_2 #1^4 & , 2],
\text{Root}[k^2 c_1 + R C J_1 #1 + R C J_2 #1 + L C J_1 #1^2 + k^2 J_2 #1^2 +
\quad L C J_2 #1^2 + R J J_1 J_2 #1^4 & , 3],
\text{Root}[k^2 c_1 + R C J_1 #1 + R C J_2 #1 + L C J_1 #1^2 + k^2 J_2 #1^2 +
\quad L C J_2 #1^2 + R J J_1 J_2 #1^4 & , 4]\}
\]

\[
N[\text{poles} /. \text{parametervalues}]
\]

\[
\{-0.395123 - 0.506844 \text{i}, -0.395123 + 0.506844 \text{i},
-0.104877 - 1.55249 \text{i}, -0.104877 + 1.55249 \text{i}\}
\]

Here is a root locus plot that substitutes values for the model parameters.

\[
\text{ParametricPlotComplexPlane}[\text{CharacteristicRoots}[H[s], s] /.\n\{R \rightarrow 1, L \rightarrow 1, c_1 \rightarrow 0.7, J_1 \rightarrow 1, J_2 \rightarrow JJ, k \rightarrow 1\}, \{JJ, 0.1, 2\}]
\]

Figure 44. Root locus plot over the complex plane.

5.4. Queries and Automatic Generation of Models

The following 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 parame-
ter 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.

\[
\text{CircuitTemplateFn}[\text{name}_, \{\text{type1}_, \text{type2}_, \text{type3}_\}] := \{
\begin{aligned}
\text{Model}[\text{name}, \\
\text{type1 a;}
\text{type2 b;}
\text{type3 c;}
\text{Modelica.Electrical.Analog.Basic.Ground g;}
\text{Equation[} \\
\text{Connect}[\text{g.p, a.p}];
\text{Connect}[\text{a.n, b.p}];
\text{Connect}[\text{b.p, c.p}];
\text{Connect}[\text{b.n, g.p}];
\text{Connect}[\text{c.n, g.p}]
\text{]} \\
\}
\end{aligned}
\]

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 extend 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 that 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 to load the Modelica.Electrical.Analog.Basic package into the internal symbol table.

\[
\text{GetDefinition[Modelica.Electrical.Analog.Basic]};
\]

Then we perform the actual query.
types = Select[
  ListModelNames[Modelica.Electrical.Analog.Basic],
  Function[modelName,
    Not[FreeQ[GetDefinition[modelName, Format -> MathModelicaFullForm],
        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.), whose prefixes are not shown for
brevity, of the four types {Inductor,Capacitor,Conductor,Resistor} are
computed by taking a generalized outer product of the three

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}];

Here 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 -> MathModelicaForm]
Model[foo53,
  Modelica.Electrical.Analog.Basic.Resistor a;
  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]
]
]

Now we create a Total model within which all 64 generated models will be instantiated. First, create the an empty model.

    Model[Total, 
      ];

Then we use the Within statement to move the current scope inside the model and 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 should 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[]

Here we retrieve the generated model Total, where we have abbreviated the output to save some space.

    GetDefinition[Total, Format -> MathModelicaForm]
Finally, we simulate the Total model to verify that the test models are semantically correct.

\[ \text{Simulate}[\text{Total}, \{t, 0, 1\}]; \]

## 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 following example is from an experiment in extending Modelica with partial differential equations [42]. 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.

```plaintext
Class[TestModel,
    Parameter Real xc \equiv 0;
    Parameter Real yc \equiv 0;
    Parameter Real r \equiv 1;
    Parameter Real delta \equiv 0;
    Domain Circle dom\{\{xc \equiv xc, yc \equiv yc, r \equiv r\}\};
    Space Real u;
    Boundary[
        w[\{dom, 0\}] \equiv \text{finit}[\text{dom}.\text{x}, \text{dom}.\text{y}];
        \partial, w[\{dom, 0\}] \equiv 0;
        w[\{dom.lefthalf, time\}] \equiv \text{delta};
        \partial, w[\{dom.lefthalf, time\}] \equiv 0;
        \partial, w[\{dom.lefthalf, time\}] \equiv 0;
        w[\{dom.righthalf, time\}] \equiv \text{delta};
        \partial, w[\{dom.righthalf, time\}] \equiv 0;
        \partial, w[\{dom.righthalf, time\}] \equiv 0;
    ];
    Equation[
        \partial_{(t, 2)} w[\{dom, time\}] \equiv \partial_{(x, 2)} w[\{dom, time\}] + \partial_{(y, 2)} w[\{dom, time\}];
    ];
];
```

Here is a plotted result of a solution at a specific time instant from running the prototype simulator on this model.
6. Conclusion

This article 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 based on *Mathematica* capabilities 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 integrated with *Mathematica* 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.

Acknowledgments

I would like to thank the entire *MathModelica* team from MathCore AB and the Dynola team at Dynasim, as well as my colleagues at PELAB—the Programming Environment Laboratory, without whom this work would not have been possible. I would specially also like to thank Peter Bunus for inspiration and help in preparing this article.
I would like to acknowledge the following members of the *MathModelica* team for contributions to the design and implementation of the *MathModelica* system, as well as providing several examples in this article: Peter Aronsson, Jan Brugård, Peter Bunus, Jonas Elbornsson, Vadim Engelssohn, Daniel Hedberg, Andreas Idebrant, Lucian Popescu, and Yelena Turetskaya. I also thank Mikael Adlers, Henrik Johansson, Andreas Karström, and Pontus Lidman, for contributions to an early version of *MathModelica*, and Levon Saldamli for the PDE example in Section 5.5. Special thanks to Mats Jirstrand and Johan Gunnarsson who made substantial contributions to the early version of *MathModelica*. Moreover, Jirstrand wrote most of sections 4.2, 4.3, and 5.3, and Gunnarsson most of Section 5.4 in this article.

The author would also like to thank the other members of the Modelica Association for inspiring discussions and contributions to the *Modelica* language design. The most recent version at the time of this writing, *Modelica* 2.2, was released February 2, 2005. The Modelica Association was formed in Linköping Sweden, on February 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 2.2**

Bernhard Bachmann, University of Applied Sciences, Bielefeld, Germany  
John Batteh, Ford Motor Company, Dearborn, MI, U.S.A.  
Dag Brück, Dynasim, Lund, Sweden  
Francesco Casella, Politecnico di Milano, Milano, Italy  
Christoph Claß, Fraunhofer Institute for Integrated Circuits, Dresden, Germany  
Jonas Eborn, Modelon AB, Lund, Sweden  
Hilding Elmqvist, Dynasim, Lund, Sweden  
Rüdiger Franke, ABB Corporate Research, Ladenburg, Germany  
Peter Fritzson, Linköping University, Sweden  
Anton Haumer, Technical Consulting & Electrical Engineering, St. Andrae-Woerdern, Austria  
Christian Kral, arsenal research, Vienna, Austria  
Sven Erik Mattsson, Dynasim, Lund, Sweden  
Chuck Newman, Ford Motor Company, Dearborn, MI, U.S.A.  
Hans Olsson, Dynasim, Lund, Sweden  
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany  
Markus Plainer, arsenal research, Vienna, Austria  
Adrian Pop, Linköping University, Sweden  
Katrin Pröß, Technical University Hamburg-Harburg, Germany  
André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany  
Christian Schweiger, German Aerospace Center, Oberpfaffenhofen, Germany  
Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.  
Hubertus Tummescheit, Modelon AB, Lund, Sweden  
Hans-Jürg Wiesmann, ABB Switzerland Ltd., Corporate Research, 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—the Programming Environment Laboratory, Department of Computer
and Information Science, Linköping University, preceding the development of the
MathModelica system. These include NUTEK—The Swedish Board for Technical Development, for example, 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


About the Author

Peter Fritzson, Ph.D., is a professor and director of the Programming Environment Laboratory (PELAB) at the Department of Computer and Information Science, Linköping University, Sweden. He currently holds positions as Chairman at MathCore Engineering, Chairman of the Scandinavian Simulation Society, Secretary of EuroSim, and Vice Chairman of the Modelica Association (www.modelica.org), an organization he helped to establish. He was previously affiliated with Sun Microsystems. Professor Fritzson has published twelve books and over a hundred scientific papers.

Peter Fritzson
PELAB—Programming Environment Laboratory
Department of Computer and Information Science
Linköping University, SE-581 83, Linköping, Sweden
petfr@ida.liu.se
www.ida.liu.se/labs/pelab
Modelica book home page with MathModelica examples:
www.mathcore.com/DrModelica
Free OpenModelica implementation:
www.ida.liu.se/labs/pelab/modelica/OpenModelica.html