Modelica – A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation

Peter Fritzson, Peter Bunus
PELAB - Programming Environment Laboratory
Department of Computer and Information Science,
Linköping University, SE-581 83, Linköping, Sweden
{petbu,petfr}@ida.liu.se

Abstract

Modelica is a general equation-based object-oriented language for continuous and discrete-event modeling of physical systems for the purpose of efficient simulation. The language unifies and generalizes previous object-oriented modeling languages. The Modelica modeling language and technology is being warmly received by the world community in modeling and simulation. It is bringing about a revolution in this area, based on its ease of use, visual design of models with combination of lego-like predefined model building blocks, its ability to define model libraries with re-usable components and its support for modeling and simulation of complex applications involving parts from several application domains. In this paper we present the Modelica language with emphasis on its language features and one of the associated simulation environments. Simulation models can be developed in an integrated problem-solving environment by using a graphical editor for connection diagrams. Connections are established just by drawing lines between objects picked from a class library. The principles of object oriented physical systems modeling and the multi-domain capabilities of the language are presented in the paper by several examples.

1. Introduction

Modelica is a new language for hierarchical object-oriented physical modeling which is developed through an international effort [6][3] [8][9]. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a *de facto* standard. The language has been designed to allow tools to generate efficient simulation code automatically with the main objective to facilitate exchange of models, model libraries and simulation specifications. It allows defining simulation models modularly and hierarchically and combining various formalisms expressible in the more general Modelica formalism. The multidomain capability of Modelica gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic etc, model components within the same application model. Compared to other equation based languages available today. Modelica is primarily a modeling language, sometimes called hardware description language, that allows the user to specify mathematical models of complex physical systems, e.g. for the purpose of computer simulation of dynamic systems where behavior evolves as a function of time. Modelica is also an object-oriented equation based programming language, oriented towards computational applications with high complexity requiring high performance. The four most important features of Modelica are:

- Modelica is primarily based on equations instead of assignment statements. This permits acausal modeling that gives better reuse of classes since equations do not specify a certain data flow direction. Thus a Modelica class can adapt to more than one data flow context.
- Modelica has multi-domain modeling capability, meaning that model components corresponding to physical objects from several different domains such as e.g. electrical, mechanical, thermodynamic, hydraulic, biological and control applications can be described and connected.
- Modelica is an object-oriented language with a general class concept that unifies classes, generics — known as templates in C++, and general subtyping into a single language construct. This facilitates reuse of components and evolution of models.
- Modelica has a strong software component model, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.

The reader of the paper is referred to [9][8] and [13] for a complete description of the language and its

functionality from the perspective of the motivations and design goals of the researchers who developed it.

In the following section, the object oriented mathematical modeling principle is briefly introduced together some Modelica language constructs. In Section 3 the continuous system modeling capabilities of the Modelica language are introduced with the help of a simple simulation example. Section 4 and Section 5 present some discrete Modelica language constructs and corresponding discrete and hybrid simple simulation examples. In Section 6 the Modelica package concept is briefly introduced and a mechanical example is provided from the Modelica Standard Multi-Body Library. Section 7 introduces MathModelica: a fully integrated program solving environment for full system modeling and simulation. Finally, future work and concluding remarks are given.

2. Object-Oriented Mathematical Modeling

Traditional object-oriented programming languages like Simula, C++, Java, and Smalltalk, as well as procedural languages such as Fortran or C, support programming with operations on stored data. The stored data of the program includes variable values and object data. The number of objects often changes dynamically. The Smalltalk view of object-orientation emphasizes sending messages between (dynamically) created objects.

The Modelica view on object-orientation is different since the Modelica language emphasizes *structured* mathematical modeling. Object-orientation is viewed as a structuring concept that is used to handle the complexity of large system descriptions. A Modelica model is primarily a declarative mathematical description, which simplifies further analysis. Dynamic system properties are expressed in a declarative way through equations.

The concept of *declarative* programming is inspired by mathematics where it is common to state or declare what *holds*, rather than giving a detailed stepwise *algorithm* on *how* to achieve the desired goal as is required when using procedural languages. This relieves the programmer from the burden of keeping track of such details. Furthermore, the code becomes more concise and easier to change without introducing errors.

Thus, the Modelica view of object-orientation, from the point of view of object-oriented mathematical modeling, can be summarized as follows:

- Object-orientation is primarily used as a structuring concept, emphasizing the declarative structure and reuse of mathematical models.
- Dynamic model properties are expressed in a declarative way through equations.
- An object is a collection of instance variables and equations that share a set of stored data.

- Object orientation is not viewed as dynamic message passing.

The declarative object-oriented way of describing systems and their behavior offered by Modelica is at a higher level of abstraction than the usual object-oriented programming since some implementation details can be omitted. For example, the users do not need to write code to explicitly transport data between objects through assignment statements or message passing code. Such code is generated automatically by the Modelica compiler based on the given equations.

Just as in ordinary object-oriented languages classes are blueprints for creating objects. Both variables and equations can be inherited between classes. Function definitions can also be inherited. However, specifying behavior is primarily done through equations instead of via methods. There are also facilities for stating algorithmic code including functions in Modelica, but this is an exception rather than the rule.

As briefly mentioned before, acausal modeling is a declarative modeling style meaning modeling based on equations instead of assignment statements. The main advantage 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 stating which variables are needed as *outputs,* and which are external *inputs* to the simulated system. The acausality of Modelica library classes makes these more reusable than traditional classes containing assignment statements where the `input-output` causality is fixed.

To illustrate the idea of acausal physical modeling we give an example of a simple electrical circuit, see Figure 1. The connection diagram of the electrical circuit shows how the components are connected and roughly corresponds to the physical layout of the electrical circuit on a printed circuit board. The physical connections in the real circuit correspond to the logical connections in the diagram. Therefore the term physical modeling is quite appropriate.
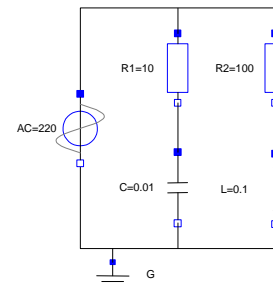
Figure 1. Connection diagram of the acausal simple circuit model.

The Modelica `SimpleCircuit` class below directly corresponds to the circuit depicted in the connection diagram of Figure 1. Each graphic object in the diagram corresponds to a declared instance in the simple circuit model. The model is acausal since no signal flow, i.e

cause-and-effect flow, is specified. Connections between objects are specified using the connect statement, which is a special syntactic form of equation that we will tell more about later.

```
model SimpleCircuit
  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 SimpleCircuit;
```

As a comparison we show the same circuit modeled using causal block oriented modeling (e.g using Simulink) depicted as a diagram in

Figure 2. Here the physical topology is lost – the structure of the diagram has no simple correspondence to the structure of the physical circuit board. This model is causal since the signal flow has been deduced and is clearly shown in the diagram.
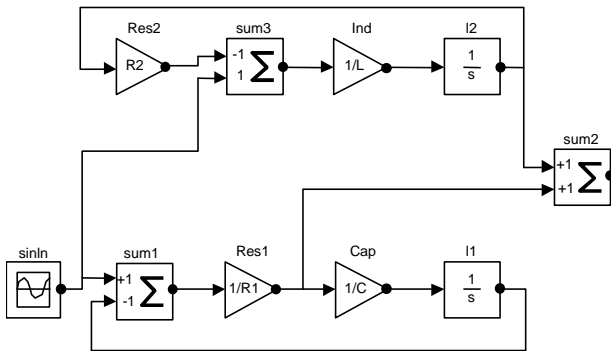

Figure 2. The simple circuit model using causal block oriented modeling with explicit signal flow.

2.1.   Modelica Classes

Modelica, like any object-oriented computer language, provides the notions of classes and objects, also called instances, as a tool for solving modeling and programming problems. Every object in Modelica has a class that defines its data and behavior. A class has three kinds of members:

- *Fields* are data variables associated with a class and its instances. Fields store results of computations

caused by solving the equations of a class together with equations from other classes.

- *Equations* specify the behavior of a class. The way in which the equations interact with equations from other classes determines the solution process, i.e. program execution.
- *Classes* can be members of other classes.

Here is the declaration of a simple class that might represent a point in a three-dimensional space:

```
class Point
 "point in a three-dimensional space"
  public Real x;
  Real y, z;
end Point;
```

2.2.   Inheritance

One of the major benefits of object-orientation is the ability to extend the behavior and properties of an existing class. The original class, known as the *superclass* or *base class*, is extended to create a more specialized version of that class, known as the *subclass* or *derived class*. In this process, the behavior and properties of the original class in the form of field declarations, equations, and other contents is reused, or inherited, by the subclass.

Let us regard an example of extending a simple Modelica class, e.g. the class `Point` introduced previously. First we introduce two classes named `ColorData` and `Color`, where ‑`Color` inherits the data fields to represent the color from class `ColorData` and adds an equation as a constraint. The new class `ColoredPoint` inherits from multiple classes, i.e. uses multiple inheritance, to get the position fields from class `Point` and the color fields together with the equation from class `Color`.

```
record ColorData
  Real red;
  Real blue;
  Real green;
end ColorData;

class Color
  extends ColorData;
equation
  red + blue + green = 1;
end Color;

class Point
  public Real x;
  Real y, z;
end Point;

class ColoredPoint
  extends Point;
  extends Color;
end ColoredPoint;
```

2.3. Modelica Equations

As we already stated, Modelica is primarily an equation-based language in contrast to ordinary programming languages where assignment statements proliferate. Equations are more flexible than assignments since they do not prescribe a certain data flow direction. This is the key to the physical modeling capabilities and increased reuse potential of Modelica classes.

Thinking in equations is a bit unusual for most programmers. In Modelica the following holds:

- Assignment statements in conventional languages are usually represented as equations in Modelica.
- Attribute assignments are represented as equations.
- Connections between objects generate equations.

Equations are more powerful than assignment statements. For example, consider a resistor equation where the resistance R multiplied by the current i is equal to the voltage v:

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

This equation can be used in three ways corresponding to three possible assignment statements: computing the current from the voltage and the resistance, computing the voltage from the resistance and the current, or computing the resistance from the voltage and the current. This is expressed in the following three assignment statements:

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

2.4. Components

Components are connected via the *connection mechanism* realized by the Modelica language, which can be visualized in connection diagrams. A *component framework* realizes components and connections, and insures that communication works over the connections. For systems composed of *acausal* components the direction of data flow, i.e. the *causality*, is initially unspecified for connections between those components. Instead the causality is automatically deduced by the compiler when needed. Components have well-defined *interfaces* consisting of ports, also known as *connectors*, to the external world. These concepts are illustrated in Figure 3.
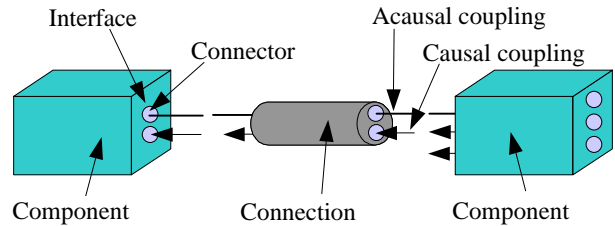

Figure 3. Connecting two components within a component framework.

Modelica uses a slightly different terminology compared to most literature on software component systems: *connector* and *connection*, rather than *port* and *connector* respectively in software component literature.

In the context of Modelica *class libraries* software components are Modelica classes. However when building particular models, components are *instances* of those Modelica classes. A component class should be defined *independently of the environment* where it is used, which is essential for its *reusability*. This means that in the definition of the component including its equations, only local variables and connector variables can be used. No means of communication between a component and the rest of the system, apart from going via a connector, is allowed. A component may internally consist of other connected components, i.e. *hierarchical* modeling.

2.5. Connectors and Connector Classes

Modelica *connectors* are instances of *connector classes*, i.e. classes with the keyword `connector` or classes with the `class` keyword that fulfill the constraints of `connector` restricted classes. Such *connectors* declare variables that are part of the communication *interface* of a component defined by the connectors of that component. Thus, connectors specify the interface for interaction between a component and its surroundings.
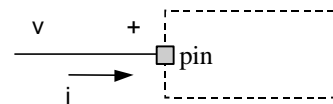

Figure 4. A component with an electrical pin connector; i.e. an instance of a `Pin`

For example, class `Pin` is a connector class that can be used to specify the external interface for electrical components that have pins as interaction points.

```
connector Pin
  Voltage        v;
  flow Current   i;
end Pin;
Pin  pin; // An instance pin of class Pin
```

Since Modelica is a multidomain language, connector classes can be formulated for a number of different application domains. The `Flange` connector class below, analogous to `Pin`, is used to describe interfaces for one-dimensional interaction in the mechanical domain by specifying the position `s` and force `f` at a point of interaction.
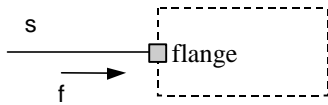
Figure 5. A component with a mechanical flange connector.

```
connector Flange
  Position   s;
  flow Force f;
end Flange;
Flange flange; // An instance flange of
                   class Flange
```
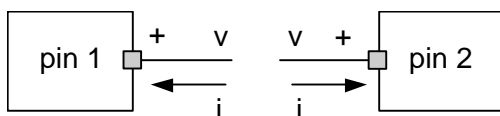
2.6.   Connections

Connections between components can be established between connectors of equivalent type. Modelica supports equation-based acausal connections, which means that connections are realized as equations. For acausal connections, the direction of data flow in the connection need not be known. Additionally, causal connections can be established by connecting a connector with an `input` attribute to a connector declared as `output`.

Two types of coupling can be established by connections depending on whether the variables in the connected connectors are non-flow (default), or declared using the prefix `flow`:

- Equality coupling, for non-flow variables, according to Kirchhoff's first law.
- Sum-to-zero coupling, for flow variables, according to Kirchhoff's current law.

For example, the keyword `flow` for the variable `i` of type `Current` in the `Pin` connector class indicates that all currents in connected pins are summed to zero, according to Kirchhoff's current law.

Figure 6. Connecting two components that have electrical pins.

Connection statements are used to connect instances of connection classes. A connection statement `connect(pin1,pin2)` with `pin1` and `pin2` of connector class `Pin`, connects the two pins so that they form one node. This produces two equations, namely:

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

The first equation says that the voltages of the connected wire ends are the same. The second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix `flow` is used. Similar laws apply to flows in piping networks and to forces and torques in mechanical systems.

3.   Continuous Time Simulation

As an introduction to the Modelica continuous time simulation capabilities we will present a model of a rocket landing on the moon surface adapted from [4].

Here is a simple class called `CelestialBody` that can be used to store data related to celestial bodies such as the earth, the moon, asteroids, planets, comets, and stars:

```
class CelestialBody
  constant  Real   g = 6.672e-11;
  parameter Real   radius;
  parameter String name;
  Real             mass;
end CelestialBody;
```

Equations is the primary means of specifying the behavior of a class in Modelica, even though algorithms and functions are also available. The way in which the equations interact with equations from other classes determines the solution process, i.e. program execution, where successive values of variables are computed over time. This is exactly what happens during dynamic system simulation. During solution of time dependent problems, the variables store results of the solution process at the current time instant.

The class `Rocket` embodies the equations of vertical motion for a rocket which is influenced by an external gravitational force field `gravity`, and the force `thrust` from the rocket motor, acting in the opposite direction to the gravitational force, as in the expression for acceleration below:

$$acceleration = \frac{thrust - mass * gravity}{mass}$$

The following three equations are first-order differential equations stating well-known laws of motion between altitude, vertical velocity, and acceleration:

$$mass' = -massLossRate * abs(thrust)$$

$$altitude' = velocity$$

$velocity' = acceleration$

All these equations appear in the class `Rocket` below, where the mathematical notation (') for derivative has been replaced by the pseudo function `der()` in Modelica. The derivative of the rocket `mass` is negative since the rocket fuel mass is proportional to the amount of `thrust` from the rocket motor.

```
class Rocket "rocket class"
  parameter String name;
  Real mass(start=1038.358);
  Real altitude (start= 59404);
  Real velocity(start= -2003);
  Real acceleration;
  Real thrust; // Thrust force on rocket
  Real gravity;// Gravity forcefield
  parameter Real massLossRate=0.000277;
equation
  (thrust-mass*gravity)/mass=acceleration;
  der(mass)  = -massLossRate * abs(thrust);
  der(altitude) = velocity;
  der(velocity) = acceleration;
end Rocket;
```

The following equation, specifying the strength of the gravitational force field, is placed in the class `MoonLanding` in the next section since it depends on both the mass of the rocket and the mass of the moon:

$$gravity = \frac{moon.g * moon.mass}{(apollo.altitude + moon.radius)^2}$$

The amount of thrust to be applied by the rocket motor is specific to a particular class of landings, and therefore also belongs to the class `MoonLanding`:

$$thrust = if\ (time < thrustDecreaseTime)\ then$$
$$force1$$
$$else\ if\ (time < thrustEndTime)\ then$$
$$force2$$
$$else\ 0$$

Members of a Modelica class can have two levels of visibility: `public` or `protected`. The default is `public` if nothing else is specified, e.g. regarding the variables `force1` and `force2` in the class `MoonLanding` below. The `public` declaration of `force1`, `force2`, `apollo`, and `moon` means that any code with access to a `MoonLanding` instance can read or update those values. The other possible level of visibility, specified by the keyword `protected` – e.g. for the variables `thrustEndTime` and `thrustDecreaseTime`, means that only code inside the class as well as code in classes that inherit this class are allowed access. However, only code inside the class is allowed access to the *same* instance of a protected variable.

Note that an occurrence of one of the keywords `public` or `protected` means that all member declarations following that keyword assume the corresponding visibility until another occurrence of one of those keywords.

The variables `thrust`, `gravity` and `altitude` belong to the `apollo` instance of the `Rocket` class and are therefore prefixed by `apollo` in references such as `apollo.thrust`. The gravitational constant `g`, the `mass`, and the `radius` belong to the particular celestial body called `moon` on which surface the `apollo` rocket is landing.

```
class MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
protected
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
public
  Rocket          apollo(name="apollo13");
  CelestialBody moon (name="moon",
                      mass=7.382e22,
                      radius=1.738e6);
equation
  apollo.thrust =
      if (time<thrustDecreaseTime) then
            force1
      else if (time<thrustEndTime) then
            force2
      else 0;
  apollo.gravity = moon.g*moon.mass
      /(apollo.altitude + moon.radius)^2;
end MoonLanding;
```

We simulate the `MoonLanding` model during the time interval {0, 230} by the following command, using the MathModelica simulation environment:

```
Simulate[MoonLanding, {t, 0, 230}]
```

Since the solution for the altitude of the Apollo rocket is a function of time, it can be plotted in a diagram, see Figure 7. It starts at an altitude of 59404 (not shown in the diagram) at time zero, gradually reducing it until touchdown at the lunar surface when the altitude is zero. Note that the MathModelica `PlotSimulation` command, by default, if nothing else is specified, refers to the results of the most recent simulation.
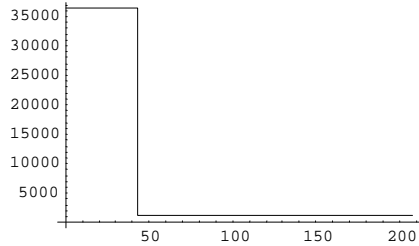
```
PlotSimulation[apollo.altitude[t],
                      {t, 0, 208}]
```
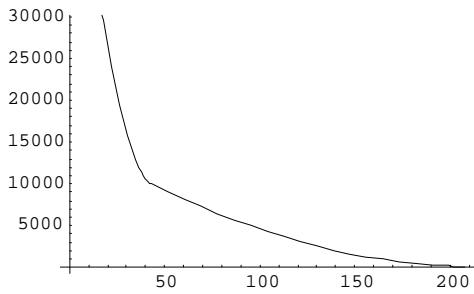
Figure 7. Altitude of the Apollo Rocket over the lunar surface.

The thrust force from the rocket is initially high but is reduced to a low level after 43.2 seconds, i.e. the value of the simulation parameter `thrustDecreaseTime`, as shown in Figure 8.
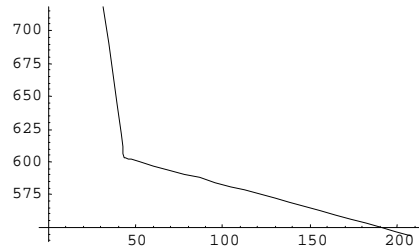
```
PlotSimulation[apollo.thrust[t],
                        {t, 0, 208}]
```


Figure 8. Thrust from the rocket motor, with an initial high thrust f1 followed by a lower thrust f2.
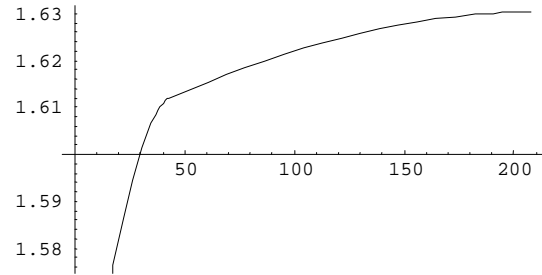
The mass of the rocket decreases from initially 1038.358 to around 540 as the fuel is consumed, see Figure 9.

```
PlotSimulation[apollo.mass[t], {t, 0, 208}]
```


**Figure 9. Rocket mass decreases when the fuel is consumed.**

```
PlotSimulation[apollo. gravity [t],
                        {t, 0, 208}]
```


Figure 10. Gradually increasing gravity when the rocket approaches the lunar surface.

The rocket initially has a high negative velocity when approaching the lunar surface. This is reduced to zero at touchdown giving a smooth landing, as shown in Figure 11.

```
PlotSimulation[apollo.velocity[t],
                        {t, 0, 208}]
```
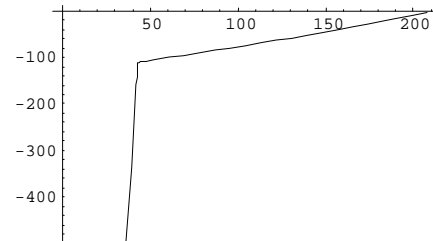

Figure 11. Vertical velocity relative to the lunar surface.

4. Discrete Event Modeling

Physical systems evolve continuously over time, whereas certain man-made systems evolve by discrete steps between states. These discrete changes, or *events*, when something happens, occur at certain points in time. We talk about *discrete event dynamic systems* as opposed to *continuous dynamic systems* directly based on the physical laws derived from first principles, e.g. from conservation of energy, matter, or momentum. One of the key issues concerning the use of events for modeling is how to express behavior associated with events. The traditional imperative way of thinking about a behavior at events is a piece of code that is activated when an event condition becomes `true` and then executes certain actions, e.g. as in the `when`-statement skeleton below:

```
when (event_conditions) then
  event-action1;
  event-action2;
  ...
end when;
```

On the other hand, a declarative view of behaviour can be based on equations. *When-clauses* are used to express

equations that are only valid (become active) at events, e.g. at discontinuities or when certain conditions become true. The conditional *equations* automatically contain only *discrete-time* expressions since they become active only at event instants, i.e. at discrete points in time and may change.

```
when <conditions> then
  <equations>
end when;
```

For example, the two equations in the `when`-clause below become active at the event instant when the `Boolean` expression `x > 2` becomes `true`.

```
when x > 2 then
  y1 = sin(x);
  y3 = 2*x + y1+y2;
end when;
```

If we instead use a `Boolean` vector expression containing three elements as the *conditions*, then the two equations will be activated at event instants when either of the three conditions: `x > 2`, `sample(0,2)`, or `x < 5` becomes `true`. Typically each condition in the vector gives rise to its own event instants.

```
when {x > 2, sample(0,2), x < 5} then
  y1 = sin(x);
  y3 = 2*x + y1+y2;
end when;
```

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 12 shows a graph of a discrete-time variable, and Figure 13 two evolving continuous-time variables.
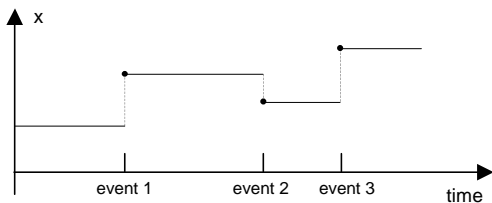
Figure 12. A discrete-time variable x changes value only at event instants.

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 assigned in a when-statement or being on the left-

hand side of an equation in a when-clause is discrete-time. A Real variable *not* assigned in any when-statement or being on the left-hand side of an equation in a when-clause is continuous-time. It is not possible to have continuous-time `Boolean`, `Integer`, or `String` variables.
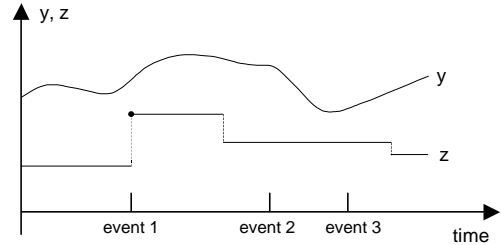

Figure 13. Continuous-time variables like y and z may change value both between and at events.

In Figure 13 above, both `y` and `z` are classified as continuous-time variables and are allowed to change at any point in time. The `y` variable is a continuous function of time whereas `z` is discontinuous at several points in time. Two of those points where `z` is discontinuous are not associated with any events, which will create some extra work and possibly numerical difficulties for the numerical equation solver. In this case it would have been better to have `z` as a discrete-time variable with events at all its points of discontinuity.

4.1. Event-Related Built-in Functions and Operators

Modelica provides a number of built-in functions and operators related to events and time, which are quite useful in discrete and hybrid modeling. Some functions generate events, some can be used to express conditions for triggering events, one function prevents events, and some operators and functions provide access to variable values before the current event as well as causing discontinuous change of variable values at an event.

The function `sample`(*first,interval*) returns `true` and can be used to trigger triggers events at time instants *first* + i*interval (i=0,1,...), where *first* is the time of the first event and *interval* is the time interval between the periodic events. It is typically used in models of periodically sampled systems.
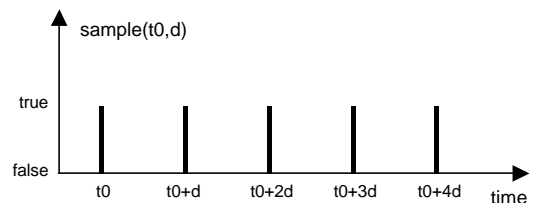
Figure 14. The call sample(t0,d) returns true and triggers events at times t0+i*d, where i=0,1,etc.

The function call `pre(y)` gives the "predecessor value" of `y`. For a discrete-time variable `y` at an event `pre(y)` is the value of `y` immediately preceding the event, as depicted in Figure 15 below. The argument `y` must be a variable and not a general expression
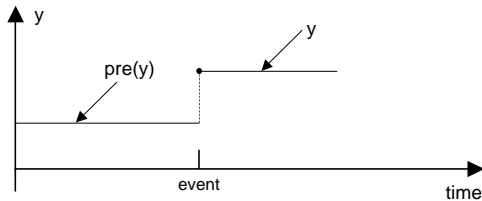
Figure 15. At an event, `pre(y)` gives the previous value of `y` immediately before the event, except for event iteration when the value is from the previous iteration.

The `pre` operator can only be applied to a variable `y`, e.g. as in the expression `pre(y)`, if the following three conditions are fulfilled:

- The variable `y` has type `Boolean`, `Integer`, `Real`, or `String`, or a subtype of those.
- The variable y is a discrete-time variable.
- The operator is *not* used within a function

At the initial time when simulation starts, `pre(y) = y.start`, i.e. it is identical to the start attribute value of its argument. Certain Modelica variables can be declared as constants or simulation parameters by using the prefixes `constant` and `parameter` respectively. Such variables do not change value during simulations. For a constant or parameter variable `c`, it is always the case that `c = pre(c)`.

4.2.   A Simple Discrete Simulation Model.

As a very simple example of discrete time simulation we can consider a simple server model with queue. We start by first defining a model which generates customers at random time moments. This model calls the function `normalvariate` which is a random number generator function used to generate the time delay until the next customer arrival.

```
model CustomerGeneration
  Random.discreteConnector dOutput;
  parameter Real mean = 0;
  parameter Real stDeviation = 1;
  discrete Real normalDelta;
  discrete Real
      nextCustomerArrivalTime(start=0);
  discrete Random.Seed
      randomSeed(start={23,87,187});
equation
  when pre(nextCustomerArrivalTime)<=time
    then (normalDelta,randomSeed)=
```
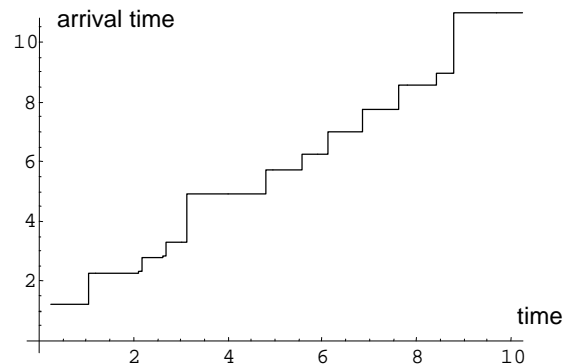
```
      normalvariate(mean,
                stDeviation,
                pre(randomSeed));
    nextCustomerArrivalTime =
      pre(nextCustomerArrivalTime) +
      abs(normalDelta);
  end when;
  dOutput.dcon = (nextCustomerArrivalTime
        <>pre(nextCustomerArrivalTime);
 end CustomerGeneration;

  Simulate[CustomerGeneration,{t,0,10}]
  PlotSimulation[nextCustomerArrivalTime[t
                ],{t,0,10}]
```


Figure 16. Customers generated at random time moments.

Based on the previously defined model, we can now design a server model which through one of its input ports will accept the arriving customers from the `CustomerGeneration` model.

```
model ServerWithQueue
  Random.discreteConnector dInput;
  parameter Real serveTime=0.1;
  discrete Real nrOfArrivedCustomers;
  discrete Real nrOfServedCustomers;
  discrete Real
      nrOfCustomersInQueue(start=0);
  discrete Boolean
      readyCustomer(start=false);
  discrete Boolean
      serveCustomer(start=false);
  Real resetTime;

equation
  when dInput.dcon then
    nrOfArrivedCustomers =
      pre(nrOfArrivedCustomers)+1;
  end when;

  when readyCustomer then
    nrOfServedCustomers =
      pre(nrOfServedCustomers)+1;
  end when;

  when (pre(nrOfCustomersInQueue)==0 and
        dInput.dcon) or
```

```
      (pre(nrOfCustomersInQueue)>=1 and
           pre(readyCustomer))then
    serveCustomer=true;
    resetTime=time;
  end when;

  readyCustomer = (serveCustomer and
       (serveTime < time-resetTime));
  nrOfCustomersInQueue =
     nrOfArrivedCustomers -
     nrOfServedCustomers;
end ServerWithQueue;
```

The nrOfCustomersInQueue represents the current number of customers in the system. The event generated by the when dInput.dcon statement represents the arrival of a new customer through the input port of the server model. The event readyCustomer represents completed service of a customer. To serve a customer if takes serveTime time. The simulation of the whole system can be done by defining a test model which connects the CustomerGeneration model with the ServerWithQueue model.

```
model testServer1
   CustomerGeneration  customer;
   ServerWithQueue server(serveTime=0.4);
equation
   connect(customer.dOutput,server.dInput);
end testServer1;
```
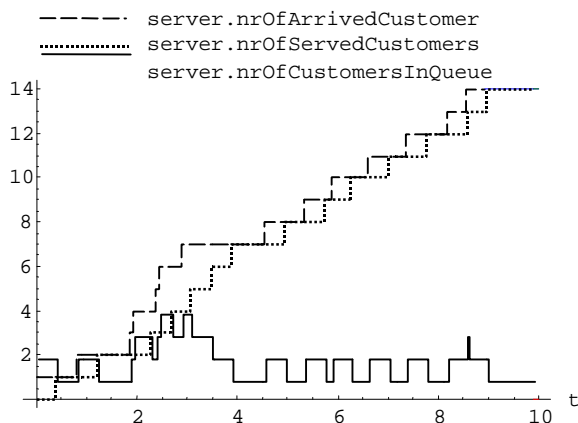
Using the MathModelica simulation environment we simulate the model for 10sec with the time necessary to serve a customer `serveTime=0.4sec`.

```
Simulate[testServer1,{t,0,10}]
PlotSimulation[
   {server.nrOfArrivedCustomers[t],
    server.nrOfServedCustomers[t],

server.nrOfCustomersInQueue[t]},{t,0,10}]
```

Figure 17. Number of arrived customers, number of served customers, and number of customers waiting in the queue.

5. Hybrid Modeling

As we have mentioned previously physical systems evolve continuously over time, whereas certain man-made systems evolve by discrete steps between states.

A *hybrid* system contains both discrete parts and continuous parts. A typical example is a digital computer interacting with the external physical world, as depicted schematically in Figure 18. This is however a very simplified picture since the discrete parts usually are distributed together with physical components in many technical systems.

This subsection gives an introduction to modeling and simulating discrete event dynamic systems and hybrid dynamic systems using Modelica.
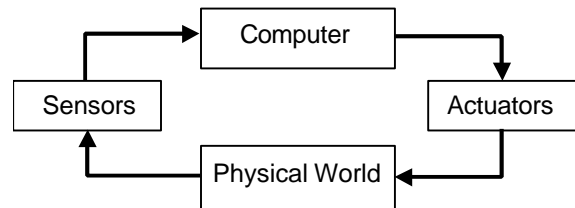
Figure 18. A hybrid system where a discrete computer senses and controls the continuous physical world.

5.1.   Bouncing Ball Simulation

A bouncing ball is a good example of a hybrid system for which the `when`-clause is appropriate as a modeling construct. The motion of the ball is characterized by the variable `height` above the ground and the vertical velocity v. The ball moves continuously between bounces, whereas discrete changes occur at bounce times, as depicted in Figure 19 below. When the ball bounces against the ground its velocity is reversed. An ideal ball would have an elasticity coefficient of 1 and would not lose any energy at a bounce. A more realistic ball, as the one modeled below, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.
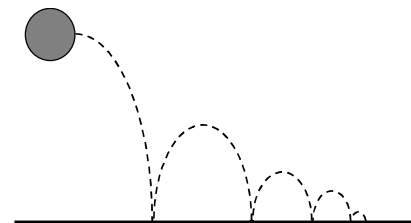
Figure 19. A bouncing ball

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. v(after bounce) = -c*v(before bounce), which is accomplished by the special syntactic form of instantaneous equation: `reinit(v,-c*v)`.

```
model BouncingBall "Bouncing ball model"
  parameter Real g=9.81;
  parameter Real c=0.90;
              // elasticity constant of ball
  Real height(start=0);
              // height above ground
  Real v(start=10);     // velocity
equation
  der(height) = v;
  der(v)      = -g;
  when height<0 then
    reinit(v, -c*v);
  end when;
end BouncingBall;
```

The `reinit(v, -c*v)` equation effectively "removes" the previous equation defining statevariable and "adds" a new equation `statevariable = valueexpression` that gives a new definition to `statevariable`. Neither the number of variables nor the number of equations is changed. The single assignment rule of Modelica is therefore not violated. The "single assignment rule" roughly means that the number of equations is the same as the number of variables.

6. Modelica Packages and Libraries

6.1.   Modelica Packages

Packages in Modelica may contain definitions of constants and classes including all kinds of restricted classes, functions, and subpackages. By the term *subpackage* we mean that the package is declared inside another package, no inheritance relationship is implied. Parameters and variables cannot be declared in a package. The definitions in a package should be related in some way, which is the main reason they are placed in a particular package. Packages are useful for number of reasons:

- Definitions that are related to some particular topic are typically grouped into a package. This makes those definitions easier to find and the code more understandable.
- Packages provide encapsulation and coarse grained structuring that reduces the complexity of large systems. An important example is the use of packages for construction of (hierarchical) class libraries.
- Name conflicts between definitions in different packages are eliminated since the package name is implicitly prefixed to names of definitions declared in a package.
- Information hiding and encapsulation can be supported to some extent by declaring `protected` classes, types, and other definitions that are only available inside the package and therefore inaccessible to outside code.
- Modelica defines a method for locating a package by providing a standard mapping of package names to storage places, typically file or directory locations in a file system.
- Identification of packages. A package stored separately, e.g. on a file, can be (uniquely) identified.

As an example, consider the package `ComplexNumbers` below which contains a data structure declaration, the record `Complex`, and associated operations such as `Add`, `Multiply`, `MakeComplex`, `etc`. The package is declared as `encapsulated` which is the recommended software engineering practice to keep the system well structured as well as being easier to understand and maintain.

```
encapsulated package ComplexNumbers

record Complex
  Real re;
  Real im;
end Complex;

function Add
  input  Complex x;
  input  Complex y;
  output Complex z;
algorithm
  z.re := x.re + y.re;
  z.im := x.im + y.im
end Add;

function Multiply
  input  Complex x;
  input  Complex y;
  output Complex z;
algorithm
  z.re := x.re*y.re - x.im*y.im;
  z.im := x.re*y.im + x.im*y.re;
end Multiply;

function MakeComplex
  input  Real    x;
  input  Real    y;
  output Complex z;
algorithm
  z.re := x;
  z.im := y;
end MakeComplex;

// Declarations of Subtract, Divide,
// RealPart, ImaginaryPart, etc.
// (not shown here)
end ComplexNumbers;
```

The example below presents a way how one can make use of the package ComplexNumbers, where both the type Complex and the operations Multiply and Add are referenced by prefixing them with the package name ComplexNumbers.

```
class ComplexUser
  ComplexNumbers.Complex  a(x=1.0, y=2.0);
  ComplexNumbers.Complex  b(x=1.0, y=2.0);
  ComplexNumbers.Complex  z,w;
equation
  z = ComplexNumbers.Multiply(a,b);
  w = ComplexNumbers.Add(a,b);
end ComplexUser;
```

A well-designed package structure is one the most important aspects that influences the complexity, understandability, and maintainability of a large software systems. There are many factors to consider when designing a package, e.g.:

- The name of the package.
- Structuring of the package into subpackages.
- Reusability and encapsulation of the package.
- Dependences on other packages.

Modelica defines a standard mapping of hierarchical package structures onto file systems or other storage mechanisms such as databases, which provides the user with a simple and unambiguous way of locating a package. The Modelica library path mechanism makes it possible to make multiple packages and package hierarchies simultaneously available for lookup.

All these mechanisms give the user a considerable flexibility and power in structuring a package. This freedom should however be used in an appropriate way.

The name of a package should be descriptive and relate to the topic of the definitions in the package. The name can be chosen to be unique within a project, within an organization, or sometimes in the whole world, depending on the typical usage scope of the package.

Since classes can be placed in packages, and packages is a restricted form of class, Modelica allows packages to contain *subpackages*, i.e. packages declared within some other package. This implies that a package name can be composed of several simple names appended through dot notation, e.g. "*Root package*"." *Package level two*"." *Package level three*", etc. Typical examples can be found in the Modelica standard library, where all level zero subpackages are placed within the root package called Modelica. This is an extensive library containing predefined subpackages from several application domains as well as subpackages containing definitions of common constants and mathematical functions. A few examples of names of subpackages in this library follow here:

```
Modelica.Mechanics.Rotational.Interfaces
Modelica.Electrical.Analog.Basic
Modelica.Blocks.Interfaces
```

6.2.   Modelica Libraries

The equation-based foundation of the Modelica language enables simulation in an extremely broad range of scientific and engineering areas. For this reason an extensive Modelica base library is under continuous development and improvement being an intrinsic part of the Modelica effort (see www.modelica.org). Some of the model libraries include application areas such as mechanics, electronics, hydraulics and pneumatics. These libraries are primarily intended to tailor Modelica toward a specific domain by giving modelers access to common model elements and terminology from that domain.

We briefly present the multi-body system library together with a simple modeling example. The MBS (Multi Body System) library has been developed in [11], An overview can be found in [10]. Our modeling example consists of a mass hanging from a spring in a gravity field. When the spring-mounted body is disturbed from its equilibrium position, its ensuing motion in the absence of any imposed external forces is called free vibration. However, in the real world, every mechanical system possesses some inherent degree of friction which will act as a consumer of mechanical energy. Therefore, we should add to our system a viscous damper for the purpose of limiting or retarding the vibration. A schematic diagram of the system under consideration is shown in Figure 20.
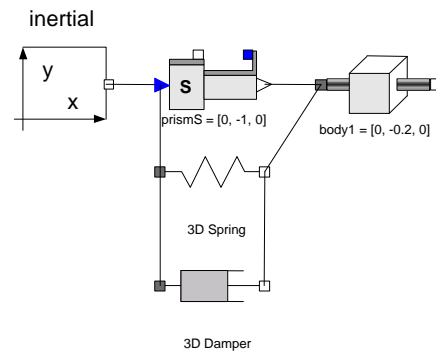
Figure 20. Schematic diagram of a damped free vibration mass system

The Modelica code for the model considered above is shown below:

```
import Modelica.Aditions.MultiBody;
model Model1
  Parts.InertialSystem inertial;
  Joints.Prismatic prismS(n={0,-1,0});
  Parts.Body2 body(r={0,-1,0},m=1);
  Forces.Spring spring1(c=300);
  Forces.Damper damper1(d=2);

equation

connect(inertial.frame_b,prismS.frame_a;
connect(prismS.frame_b,body.frame_a);
```

```
connect(prismS.frame_b,spring1.frame_b;
connect(spring1.frame_a,prismS.frame_a;
connect(spring.frame_b,damper1.frame_b);
connect(damper1.frame_a,spring1.frame_a);
end Model1;
```

An instance of the `Inertial` class defines the global coordinate system and gravitational forces (the inertial frame). All parameter vectors and tensors are given in the home position of the multi-body system with respect to the inertial frame. One instance of class `Inertial` must always be present for every multi-body model. All other objects are in some way connected to the inertia system, either directly or through other objects.

Every basic mechanical component from the MBS library has at least one or two interfaces to connect the element rigidly to an other mechanical elements. A distinguishing feature of multi-body systems is the presence of joints, which impose different types of kinematic constrains between the various bodies of the kinematic chain. The motions between links of the mechanism must to be constrained to produce the proper relative motion, i.e. the motion chosen by the designer for the particular task to be performed.

A `Prismatic` joint has been introduced in order to produce the relative motion in the Y-direction. The relative motion direction is specified by the parameter vector `n=[0,-1,0]` which is the axis of translation resolved in `frame_a`.
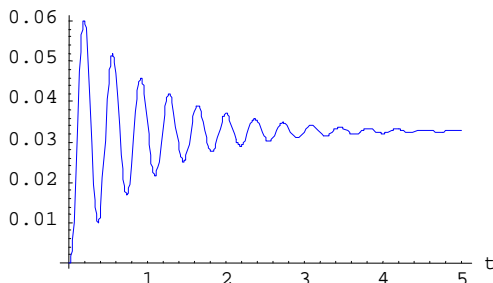

Figure 21. Plot of Mass1.s(t) absolute position of the center of the mass1 component.

7. The MathModelica Simulation Environment.

MathModelica[16] is an integrated problem-solving environment (PSE) for full system modeling and simulation using Modelica. The environment integrates Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of code and documentation, and symbolic formula manipulation provided via *Mathematica*. These capabilities give the environment considerable power, combined with easy of use. Import and export of Modelica code between internal structured and external textual representation is supported by MathModelica. For example, the system is typically used for the creation of "virtual prototypes", computer simulations that behave like a real physical object or system, e.g., an electrical network, a turbo engine, an aircraft, or an industrial robot.
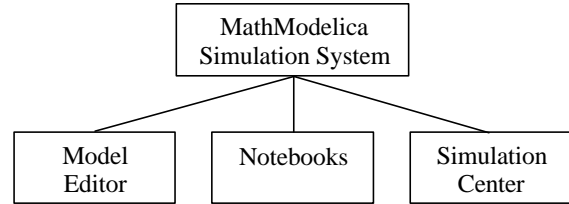
Figure 22. The MathModelica Environment structure.

The environment uses extensively the principles of literate programming [7], and integrates most activities needed in simulation design: modeling, symbolic processing, transformation and formula manipulation, storage of simulation models, version control, input and output data visualization, storage and generation of documentation. The *Mathematica*[14] system is part of the MathModelica environment. The interpreted *Mathematica* language integrates several features into a unified integrated environment: numerical and symbolic calculations, functional, procedural, rule-based and graphical programming. Additionally, the language incorporates many features of traditional mathematical notation and the goal of the language is to provide a precise and consistent way to specify computations.

The *Mathematica* system is divided into two distinct parts: the computer algebra engine ("kernel") that receives and evaluates all expressions sent to it and the user interface ("front-end"). The front-end provides the programming interface to the user and is concerned with such issues as how input is entered and how computation results are displayed to the user.

MathModelica uses Mathematica's front-end documents that are called notebooks, Figure 23. They combine text, executable commands, numerical results, graphics, and sound in a single document. A notebook provides users with a medium in which they can document their solution along with the computation itself. To provide these powerful integrated capabilities without reinventing the wheel, MathModelica integrates and extends a number of software products such as the technical computing system *Mathematica* [14] from Wolfram Research, the diagram and visualization tool Visio from Microsoft, and Dymola the simulation engine from Dynasim[15] grouped in three modules: *Notebooks, Model Editor,* and *Simulation Center*, as it is shown in Figure 22.
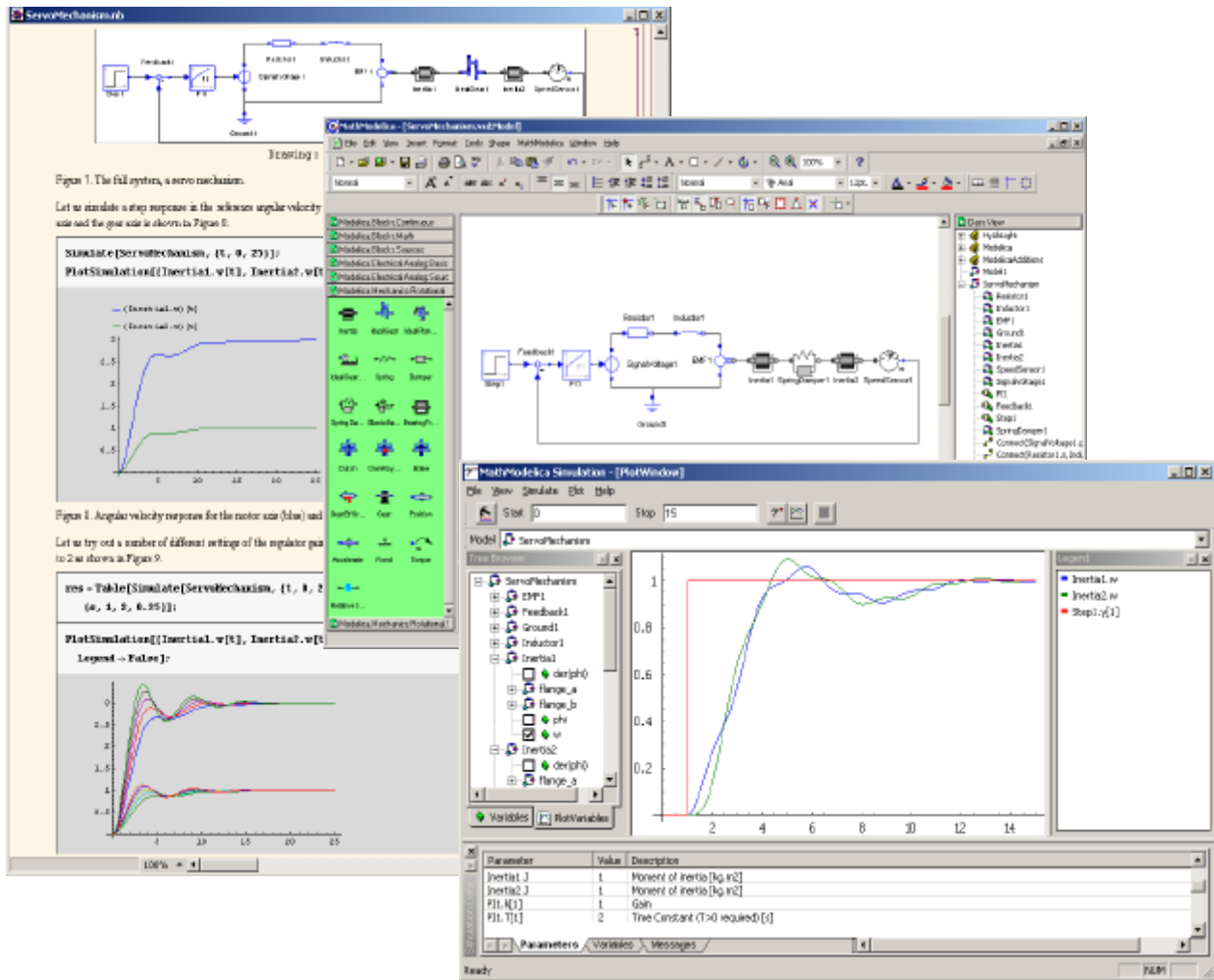
Figure 23. The MathModelica Simulation Environment: *Notebook*, *Model Editor* and *Simulation Center* screen captures.

7.1. The Model Editor

The MathModelica *Model Editor* is a graphical user interface for model diagram construction by "drag-and-drop" of ready made components from model libraries graphically represented by Visio stencils. The libraries correspond to the physical domains represented in the Modelica Standard Library or components from user defined component libraries. A screen shot of the *Model Editor* is shown in the middle of Figure 23. The *Model Editor* is an extension of the Microsoft software Visio for diagram design and schematics. This means that the user has access to a well-developed and user-friendly application but also a vast array of design features to make graphical representations of developed models look professional. Since Modelica components often represents

physical objects it is of great value to have a sufficiently rich graphical description of these objects.

The *Model Editor* can be viewed as a user interface for graphical programming in Modelica. Its basic functionality is the selection of components from ready-made libraries, connection of components in model diagrams, and to 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 this in several ways. A model diagram can be browsed and zoomed. Aggregate components can be opened in separate windows by double-clicking on their icons.

Parameter values of a component can be accessed by right-clicking on a component and open the Custom

Properties form. The complete textual Modelica model is stored in a so-called *Notebook*, which also can be opened by right-clicking a component.

The *Model Editor* is well integrated with *Notebook*s. A model diagram in a notebook is a graphical representation of the Modelica code, which can be converted into textual form by a command. Double-clicking on a model diagram in a notebook will open the Model Editor and the model can be graphically edited.

The combination of the ability of making user defined libraries of reusable components in Modelica and the *Notebook* concept of living technical documents supports model management and evolution of models of large systems. MathModelica is also an open environment both with respect to the ability of communicating with other programs as well as import and export of different data formats. A key aspect of MathModelica is that the modeling and simulation activities are 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 post processing of simulation results such as signal processing and further analysis of simulated data with the help of the environment component called *Simulation Center* shown in the lower right corner of Figure 23.

The symbolic computer-algebra capabilities can be applied to equations from Modelica models. For example, below a list of equations is extracted from a Modelica model, a new equation is appended to the list, and *Mathematica* pattern matching and transformation rules are used to transform the list of equations to a list of equations in *Laplace* form, which can be *Laplace* transformed, to be used as input for further analysis or a new Modelica-based simulation.

```
eqnlist = GetFlatEquations[Model1];
AppendTo[eqnlist,Inertia.flange_b.tau ==0];
laplaceeqns = eqnlist /.
      {1/der[a_] :> a/s, der[b_] :> s*b};
```

8. Future Work

The work by the Modelica Association on the further development of Modelica and tools is continuing. Several companies and research institutes are pursuing or planning development of tools based on Modelica. In our case we focus on interactive implementation of Modelica integrated with Mathematica to provide integration of model design, coding, documentation, simulation and graphics.

8.1. Partial Differential Equation Extension

In order to specify models containing partial differential equations, the concept of geometric domains and domain boundaries are needed in the Modelica language, as well as multi-variable functions and equations that are defined over such domains and boundaries. Furthermore, the connection mechanism need to be generalized to use domains or domain boundaries as connection regions between components with separate domains[11].

8.2. Debugging of Simulation Models

Determining the cause of errors in models of physical systems is hampered by the limitations of the current techniques of debugging declarative equation based languages. Improved debugging and verification tools need to be developed in order to statically detect a broad range of errors without having to execute the simulation model. Since the simulation system execution is expensive, static debugging tools should greatly reduce the number of test cases usually needed to validate a simulation model. The user should be presented to the original source code of the program and not burdened with understanding the intermediate code or the numerical artifacts for solving the underlying system of equations. When a simulation model fails to compile, the errors should be reported consistent with the user's perception of the simulation model or source code and possible repair strategies should also be provided [2].

8.3. Parallel Simulation

To deal with the growing complexity of modeled systems in the Modelica language, the need for parallelisation becomes increasingly important in order to keep simulation time within reasonable limits. A Modelica compiler usually performs many optimizations to reduce the number of equations and to increase the efficiency of the simulation code. However, for large and complex models there are also opportunities to successfully exploit parallelism in the simulation code. Current research is on an automatic parallelisation tool for Modelica models that reads the automatic generated simulation code produced by the standard Modelica compiler and by using specially adapted scheduling algorithms produces parallel simulation code for execution in a multiprocessor environment [1].

Acknowledgements

Sweden, at Feb. 5, 2000 and is responsible for the design of the Modelica language (see www.modelica.org).

Contributors to the Modelica Language, version 1.4

Bernhard Bachmann, *Fachhochschule Bielefeld, Germany*
Peter Bunus, *Linköping University, Linköping, Sweden*
Dag Brück, *Dynasim, Lund, Sweden*
Hilding Elmqvist, *Dynasim, Lund, Sweden*
Vadim Engelson, *Linköping University, Sweden*
Jorge Ferreira, *University of Aveiro, Portugal*
Peter Fritzson, *Linköping University, Linköping, Sweden*
Pavel Grozman, *Equa, Stockholm, Sweden*
Johan Gunnarsson, *MathCore, Linköping, Sweden*
Mats Jirstrand, *MathCore, Linköping, Sweden*
Clemens Klein-Robbenhaar, *Germany*
Pontus Lidman, MathCore, *Linköping, Sweden*
Sven Erik Mattsson, *Dynasim, Lund, Sweden*
Hans Olsson, Dynasim, *Lund, Sweden*
Martin Otter, *German Aerospace Center, Oberpfaffenhofen, Germany*
Tommy Persson, *Linköping University, Sweden*
Levon Saldamli, *Linköping University, Sweden*
André Schneider, *Fraunhofer Institute for Integrated Circuits, Dresden, Germany*
Michael Tiller, *Ford Motor Company, Detroit, U.S.A.*
Hubertus Tummescheit, *Lund Institute of Technology, Sweden*
Hans-Jürg Wiesmann, ABB Corporate Research Ltd., Baden, Switzerland

Contributors to the Modelica Standard Library, version 1.4

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
Hubertus Tummescheit, Lund Institute of Technology, Sweden

References

[1] Aronsson, P.; Fritzson, P. "Parallel Code Generation in MathModelica, an Object Oriented Component Based Simulation Environment" In *Proceedings of Workshop on Parallel/High Performance Object-Oriented Scientific Computing* (POOSC'01 14-18 Oct 2001, Tampa, Florida, USA)

[2] Bunus, P. and Fritzson, P. "Applications of Graph Decomposition Techniques to Debugging Declarative Equation Based Languages". In *Proceedings of the 42nd SIMS Conference (Porsgrunn Norway, Oct 8-9,2001)*

[3] Elmqvist, H.; S. E. Mattsson and M. Otter. 1999. "Modelica - A Language for Physical System Modeling, Visualization and Interaction." In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design* (Hawaii, Aug. 22-27).

[4] Francois E. Cellier. *Continuous System Modeling*. Springer Verlag, New York 1991.

[5] Fritzson P.; V. Engelson; J. Gunnarsson. 1998. "An Integrated Modelica Environment For Modeling, Documentation and Simulation." In *Proceedings of the 1998 Summer Computer Simulation Conference* (Reno, Nevada, USA, July 19-22).

[6] Fritzson P and V. Engelson. "Modelica, A Unified Object-Oriented Language for System Modeling and Simulation." In *Proceedings of ECOOP'98 (the 12th European Conference on Object-Oriented Programming)*(Brussels, Jul. 20-24,1998).

[7] Knuth, D.E.. "Literate Programming" The Computer Journal, NO27(2) (May 1984): 97-111.

[8] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 1.4* (December 15, 2000).

[9] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification Version 1.4.* (December 15, 2000*).*

[10] Otter, M.; Elmqvist H.; F. E. Cellier.:. Modeling of Multibody Systems with the Object-oriented Modeling Language Dymola. *Nonlinear Dynamics*, 9:91-112, Kluwer Academic Publishers. 1996.

[11] Otter, M.: O*bjektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter.* Dissertation, Fortshrittberichte VDI, Reihe 20, Nr 147. 1995

[12] Saldamli, L. and Fritzson, P. A Modelica-based Language for Object-Oriented Modeling with Partial Dirential Equations. In A. Heemink, L. Dekker, H. De Swaan Arons, I. Smith, and T. van Stijn, editors, *Proceedings. of the 4th International EUROSIM Congress* (Delft, The Netherlands, June 2001).

[13] Tiller M. Michael. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publisher 2001.

[14] Wolfram S. The Mathematica Book. Wolfram Media Inc. (February). (1996).

[15] www.dynasim.se Dynasim AB, Sweden

[16] www.mathcore.se MathCore AB, Sweden