Abstract

Object-Oriented modeling is a fast-growing area of modeling and simulation that provides a structured, computer-supported way of doing mathematical and equation-based modeling. Modelica is today the most promising modeling and simulation language in that it effectively unifies and generalizes previous object-oriented modeling languages and provides a sound basis for the basic concepts.

The Modelica modeling language and technology is being warmly received by the world community in modeling and simulation with major applications in virtual prototyping. 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 reusable components, its support for modeling and simulation of complex applications involving parts from several application domains, and many more useful facilities. To draw an analogy, Modelica is currently in a similar phase as Java early on, before the language became well known, but for virtual prototyping instead of Internet programming.

The tutorial presents an object-oriented component-based approach to computer supported mathematical modeling and simulation through the powerful Modelica language and its associated technology. Modelica can be viewed as an almost universal approach to high level computational modeling and simulation, by being able to represent a range of application areas and providing general notation as well as powerful abstractions and efficient implementations.

The tutorial gives an introduction to the Modelica language to people who are familiar with basic programming concepts. It gives a basic introduction to the concepts of modeling and simulation, as well as the basics of object-oriented component-based modeling for the novice, and an overview of modeling and simulation in a number of application areas.

The tutorial has several goals:

- Being easily accessible for people who do not previously have a background in modeling, simulation.
- Introducing the concepts of physical modeling, object-oriented modeling and component-based modeling and simulation.
- Giving an introduction to the Modelica language.
- Demonstrating modeling examples from several application areas.
- Giving a possibility for hands-on exercises.
Presenter's data

Peter Fritzson is a Professor and Director of the Programming Environment Laboratory (Pelab), at the Department of Computer and Information Science, Linköping University, Sweden. He holds the position of Director of Research and Development of MathCore Engineering AB. Peter Fritzson is chairman of the Scandinavian Simulation Society, secretary of the European simulation organization, EuroSim; and vice chairman of the Modelica Association, an organization he helped to establish. His main area of interest is software engineering, especially design, programming and maintenance tools and environments.

1. Useful Web Links

The Modelica Association Web Page

http://www.modelica.org

Modelica publications

http://www.modelica.org/publications.shtml

Modelica related research and the OpenModelica open source project at Linköping University with download of the OpenModelica system and link to download of MathModelica Lite.

http://www.ida.liu.se/~pelab/modelica/OpenModelica.html

The Proceedings of 5th International Modelica Conference, September 4-5, 2006, Vienna, Austria

http://www.modelica.org/events/Conference2006/

The Proceedings of 4th International Modelica Conference, March 7-8, 2005, Hamburg, Germany

http://www.modelica.org/events/Conference2005/


http://www.modelica.org/events/Conference2003/


http://www.modelica.org/events/Conference2002/

The Proceedings of Modelica Workshop, October 23 - 24, 2000, Lund University, Lund, Sweden

http://www.modelica.org/events/workshop2000/
2. 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 Clauß, 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 Prölß, 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
Principles of Object-Oriented Modeling and Simulation with Modelica

Peter Fritzson
Peter Bunus

Linköping University, Dept. of Comp. & Inform. Science
SE 581-83, Linköping, Sweden
{petfr,petbu}@ida.liu.se

Copyright © Peter Fritzson

Course Based on Recent Book, 2004

Peter Fritzson
Principles of Object Oriented Modeling and Simulation with Modelica 2.1
Wiley-IEEE Press
940 pages
Acknowledgements, Usage, Copyrights

• If you want to use the Powerpoint version of these slides in your own course, send an email to: peter.fritzson@ida.liu.se
• Thanks to Emma Larsdotter Nilsson for contributions to the layout of these slides
• Most examples and figures in this tutorial are adapted with permission from Peter Fritzson’s book "Principles of Object Oriented Modeling and Simulation with Modelica 2.1", copyright Wiley-IEEE Press
• Some examples and figures reproduced with permission from Modelica Association, Martin Otter, Hilding Elmqvist
• Modelica Association: www.modelica.org
• OpenModelica: www.ida.liu.se/projects/OpenModelica

Outline

• Introduction to Modeling and Simulation
• Modelica - The next generation modeling and Simulation Language
• Classes
• Components, Connectors and Connections
• Equations
• Discrete Events and Hybrid Systems
• Algorithm and Functions
• Modeling and Simulation Environments
• Demonstrations
Why Modeling & Simulation?

- Increase understanding of complex systems
- Design and optimization
- Virtual prototyping
- Verification

What is a system?

- A system is an object or collection of objects whose properties we want to study
- Natural and artificial systems
- Reasons to study: curiosity, to build it
Examples of Complex Systems

- Robotics
- Automotive
- Aircrafts
- Satellites
- Biomechanics
- Power plants
- Hardware-in-the-loop, real-time simulation

Experiments

An experiment is the process of extracting information from a system by exercising its inputs.

Problems
- Experiment might be too expensive
- Experiment might be too dangerous
- System needed for the experiment might not yet exist
Model concept

A *model* of a system is anything an *experiment* can be applied to in order to answer questions about that *system*

Kinds of models:

- **Mental model** – statement like “a person is reliable”
- **Verbal model** – model expressed in words
- **Physical model** – a physical object that mimics the system
- **Mathematical model** – a description of a system where the relationships are expressed in mathematical form – a *virtual prototype*
- **Physical modeling** – also used for mathematical models built/structured in the same way as physical models

Simulation

A *simulation* is an *experiment* performed on a *model*

Examples of simulations:

- **Industrial process** – such as steel or pulp manufacturing, study the behaviour under different operating conditions in order to improve the process
- **Vehicle behaviour** – e.g. of a car or an airplane, for operator training
- **Packet switched computer network** – study behaviour under different loads to improve performance
Reasons for Simulation

• Suppression of second-order effects
• Experiments are too expensive, too dangerous, or the system to be investigated does not yet exist
• The time scale is not compatible with experimenter (Universe, million years, …)
• Variables may be inaccessible.
• Easy manipulation of models
• Suppression of disturbances

Dangers of Simulation

Falling in love with a model
The Pygmalion effect (forgetting that model is not the real world, e.g. introduction of foxes to hunt rabbits in Australia)

Forcing reality into the constraints of a model
The Procrustes effect (e.g. economic theories)

Forgetting the model’s level of accuracy
Simplifying assumptions
Building Models Based on Knowledge

System knowledge
- The collected *general experience* in relevant domains
- The *system* itself

Specific or generic knowledge
- E.g. software engineering knowledge

Kinds of Mathematical Models
- Dynamic *vs.* Static models
- Continuous-time *vs.* Discrete-time dynamic models
- Quantitative *vs.* Qualitative models
Dynamic vs. Static Models

A **dynamic** model includes *time* in the model.
A **static** model can be defined *without* involving *time*.

Continuous-Time vs. Discrete-Time Dynamic Models

**Continuous-time** models may evolve their variable values *continuously* during a time period.
**Discrete-time** variables change values a *finite* number of times during a time period.
Quantitative vs. Qualitative Models

Results in qualitative data
Variable values cannot be represented numerically

Mediocre = 1, Good = 2, Tasty = 3, Superb = 4

Using Modeling and Simulation within the Product Design-V
Principles of Equation-Based Modeling

- Each icon represents a physical component i.e. Resistor, mechanical Gear Box, Pump
- Composition lines represent the actual physical connections i.e. electrical line, mechanical connection, heat flow
- Variables at the interfaces describe interaction with other component
- Physical behavior of a component is described by equations
- Hierarchical decomposition of components

Application Example – Industry Robot

Courtesy of Martin Otter
GTX Gas Turbine Power Cutoff Mechanism

Modelica – The Next Generation Modeling Language
Stored Knowledge

Model knowledge is stored in books and human minds which computers cannot access

“The change of motion is proportional to the motive force impressed“
– Newton

The Form – Equations

• Equations were used in the third millennium B.C.
• Equality sign was introduced by Robert Recorde in 1557

\[ 14.2c \rightarrow \frac{15.9}{2} \rightarrow 71.9 \]

Newton still wrote text (Principia, vol. 1, 1686)
“The change of motion is proportional to the motive force impressed”

CSSL (1967) introduced a special form of “equation”:

\[ \text{variable} = \text{expression} \]
\[ v = \text{INTEG(F)}/m \]

Programming languages usually do not allow equations!
## Modelica – The Next Generation Modeling Language

### Declarative language
- Equations and mathematical functions allow acausal modeling, high level specification, increased correctness

### Multi-domain modeling
- Combine electrical, mechanical, thermodynamic, hydraulic, biological, control, event, real-time, etc...

### Everything is a class
- Strongly typed object-oriented language with a general class concept, Java & MATLAB-like syntax

### Visual component programming
- Hierarchical system architecture capabilities

### Efficient, non-proprietary
- Efficiency comparable to C; advanced equation compilation, e.g. 300 000 equations, ~150 000 lines on standard PC

---

## Modelica – The Next Generation Modeling Language

### High level language
- MATLAB-style array operations; Functional style; iterators, constructors, object orientation, equations, etc.

### MATLAB similarities
- MATLAB-like array and scalar arithmetic, but strongly typed and efficiency comparable to C.

### Non-Proprietary
- Open Language Standard
- Both Open-Source and Commercial implementations

### Flexible and powerful external function facility
- LAPACK interface effort started
## Modelica Language Properties

- **Declarative** and **Object-Oriented**
- **Equation-based**: continuous and discrete equations
- **Parallel** process modeling of real-time applications, according to synchronous data flow principle
- **Functions** with algorithms without global side-effects (but local data updates allowed)
- **Type system** inspired by Abadi/Cardelli
- **Everything is a class** – Real, Integer, models, functions, packages, parameterized classes....

## Object Oriented

### Mathematical Modeling with Modelica

- The static *declarative structure* of a mathematical model is emphasized
- OO is primarily used as a *structuring concept*
- OO is *not* viewed as dynamic object creation and sending messages
- *Dynamic model* properties are expressed in a *declarative way* through equations.
- Acausal classes supports *better reuse of modeling and design knowledge* than traditional classes
Brief Modelica History

- First Modelica design group meeting in fall 1996
  - International group of people with expert knowledge in both language design and physical modeling
  - Industry and academia

- Modelica Versions
  - 1.0 released September 1997
  - 2.0 released March 2002
  - Latest version, 2.2 released March 2005

- Modelica Association established 2000
  - Open, non-profit organization

Modelica Conferences

- The 1st International Modelica conference October, 2000

- The 2nd International Modelica conference March 18-19, 2002

- The 3rd International Modelica conference November 5-6, 2003 in Linköping, Sweden

- The 4th International Modelica conference March 6-7, 2005 in Hamburg, Germany

- The 5th International Modelica conference planned September 4-5, 2006 in Vienna, Austria
Modelica Classes and Inheritance

Simplest Model – Hello World!

A Modelica “Hello World” model

Equation: $x' = -x$
Initial condition: $x(0) = 1$

```
class HelloWorld "A simple equation"
  Real x(start=1);
  equation
    der(x) = -x;
  end HelloWorld;
```

Simulation in OpenModelica environment

```
simulate(HelloWorld, stopTime = 2)
plot(x)
```
Another Example

Include algebraic equation
Algebraic equations contain no derivatives

Simulation in OpenModelica environment

Example class: Van der Pol Oscillator
Small Exercise

- Locate the HelloWorld model in DrModelica using OMNotebook!
- Simulate and plot the example. Do a slight change in the model, re-simulate and re-plot.

```modelica
class HelloWorld "A simple equation"
  Real x(start=1);
  equation
    der(x) = -x;
end HelloWorld;
```

```modelica
simulate(HelloWorld, stopTime = 2)
plot(x)
```

- Locate the VanDerPol model in DrModelica and try it!

Variables and Constants

**Built-in primitive data types**

- **Boolean**: `true` or `false`
- **Integer**: Integer value, e.g. `42` or `−3`
- **Real**: Floating point value, e.g. `2.4e-6`
- **String**: String, e.g. “Hello world”
- **Enumeration**: Enumeration literal e.g. `ShirtSize.Medium`
Variables and Constants cont’

• Names indicate meaning of constant
• Easier to maintain code
• Parameters are constant during simulation
• Two types of constants in Modelica
  • constant
  • parameter

<table>
<thead>
<tr>
<th>Variables and Constants cont’</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant Real PI = 3.141592653589793;</td>
</tr>
<tr>
<td>constant String redcolor = &quot;red&quot;;</td>
</tr>
<tr>
<td>constant Integer one = 1;</td>
</tr>
<tr>
<td>parameter Real mass = 22.5;</td>
</tr>
</tbody>
</table>

Comments in Modelica

1) Declaration comments, e.g. Real x "state variable";

class VanDerPol "Van der Pol oscillator model"
Real x(start = 1) "Descriptive string for x"; // x starts at 1
Real y(start = 1) "y coordinate"; // y starts at 1
parameter Real lambda = 0.3;
equation
  der(x) = y; // This is the 1st diff equation //
  der(y) = -x + lambda*(1 - x*x)*y; /* This is the 2nd diff equation */
end VanDerPol;

2) Source code comments, disregarded by compiler
   2a) C style, e.g. /* This is a C style comment */
   2b) C++ style, e.g. // Comment to the end of the line...
A Simple Rocket Model

\[
\text{acceleration} = \frac{\text{thrust} - \text{mass} \cdot \text{gravity}}{\text{mass}}
\]

\[
\text{mass}' = -\text{massLossRate} \cdot \text{abs(\text{thrust})}
\]

\[
\text{altitude}' = \text{velocity}
\]

\[
\text{velocity}' = \text{acceleration}
\]

```modelica
class Rocket
  parameter String name;
  parameter 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;
```

New model declaration with comments about parameters and floating point type.

Celestial Body Class

A class declaration creates a type name in Modelica.

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

An instance of the class can be declared by prefixing the type name to a variable name.

```
... CelestialBody moon;
...```

The declaration states that `moon` is a variable containing an object of type `CelestialBody`. 
Moon Landing

\[ \text{apollo.gravity} = \frac{\text{moon.g} \cdot \text{moon.mass}}{(\text{apollo.altitude} + \text{moon.radius})^2} \]

Simulation of Moon Landing

\[
\text{simulate(MoonLanding, stopTime=230)} \\
\text{plot(apollo.altitude, xrange=[0,208])} \\
\text{plot(apollo.velocity, xrange=[0,208])}
\]

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. The rocket initially has a high negative velocity when approaching the lunar surface. This is reduced to zero at touchdown, giving a smooth landing.
Restricted Class Keywords

- The class keyword can be replaced by other keywords, e.g.: model, record, block, connector, function, ...
- Classes declared with such keywords have restrictions
- Restrictions apply to the contents of restricted classes

- Example: A model is a class that cannot be used as a connector class
- Example: A record is a class that only contains data, with no equations
- Example: A block is a class with fixed input-output causality

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

Modelica Functions

- Modelica Functions can be viewed as a special kind of restricted class with some extensions
- A function can be called with arguments, and is instantiated dynamically when called
- More on functions and algorithms later in Lecture 4

```function sum
input Real arg1;
input Real arg2;
output Real result;
algorithim
result := arg1+arg2;
end sum;
```
Inheritance

Data and behavior: field declarations, equations, and certain other contents are copied into the subclass

Inheriting definitions

Inheriting multiple identical definitions results in only one definition

Legal! Identical to the inherited field blue

Illegal! Same name, but different value

Inheriting multiple different definitions of the same item is an error
Inheritance of Equations

```plaintext
class Color
    parameter Real red=0.2;
    parameter Real blue=0.6;
    Real green;
    equation
        red + blue + green = 1;
end Color;
```

Color is identical to Color2

```plaintext
class Color2 // OK!
extends Color;
    equation
        red + blue + green = 1;
end Color2;
```

```
class Color3 // Error!
extends Color;
    equation
        red + blue + green = 1.0;
        // also inherited: red + blue + green = 1;
end Color3;
```

Same equation twice leaves one copy when inheriting

Color3 is overdetermined

Different equations means two equations!

Multiple Inheritance

Multiple Inheritance is fine – inheriting both geometry and color

```plaintext
class Color
    parameter Real red=0.2;
    parameter Real blue=0.6;
    Real green;
    equation
        red + blue + green = 1;
end Color;
```

```plaintext
class Point
    Real x;
    Real y, z;
end Point;
```

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

```plaintext
class ColoredPointWithoutInheritance
    Real x;
    Real y, z;
    parameter Real red = 0.2;
    parameter Real blue = 0.6;
    Real green;
    equation
        red + blue + green = 1;
end ColoredPointWithoutInheritance;
```

Equivalent to
Multiple Inheritance cont'

Only one copy of multiply inherited class `Point` is kept

```
class Point
  Real x;
  Real y;
end Point;
```

Diamond Inheritance

```
class VerticalLine extends Point;
  Real vlength;
end VerticalLine;
```

```
class HorizontalLine extends Point;
  Real hlength;
end HorizontalLine;
```

```
class Rectangle extends VerticalLine;
  extends HorizontalLine;
end Rectangle;
```

Simple Class Definition – Shorthand Case of Inheritance

**Example:**

```
class SameColor = Color;
```

Equivalent to:

```
inheritance
class SameColor
  extends Color;
end SameColor;
```

**Often used for introducing new names of types:**

```
type Resistor = Real;
```

```
connector MyPin = Pin;
```
### Inheritance Through Modification

- Modification is a concise way of combining inheritance with declaration of classes or instances.
- A modifier modifies a declaration equation in the inherited class.
- Example: The class `Real` is inherited, modified with a different `start` value equation, and instantiated as an `altitude` variable:

```plaintext
... Real altitude(start= 59404); ...
```

### The Moon Landing

#### Example Using Inheritance

```plaintext
model Body "generic body"
  Real   mass;
  String name;
end Body;

model CelestialBody extends Body;
  constant Real g = 6.672e-11;
  parameter Real radius;
end CelestialBody;

model Rocket "generic rocket class"
  extends Body;
  parameter Real massLossRate=0.000277;
  Real altitude(start= 59404);
  Real velocity(start= -2003);
  Real acceleration;
  Real thrust;
  Real gravity;
  equation
    thrust=mass*gravity;
    der(mass)= -massLossRate*abs(thrust);
    der(altitude)= velocity;
    der(velocity)= acceleration;
end Rocket;
```
The Moon Landing
Example using Inheritance cont’

```plaintext
model MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
  Rocket apollo(name="apollo13", mass(start=1038.358) );
  CelestialBody moon(mass=7.382e22, radius=1.738e6, name="moon");
  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 Landing;
```

Inheritance of Protected Elements

If an `extends`-clause is preceded by the `protected` keyword, all inherited elements from the superclass become protected elements of the subclass.

```plaintext
class Color
  Real red;
  Real blue;
  Real green;
  equation
    red + blue + green = 1;
end Color;

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

class ColoredPoint
  protected
    extends Color;
  public
    extends Point;
end ColoredPoint;

class ColoredPointWithoutInheritance
  Real x;
  Real y,z;
  protected
    Real red;
    protected Real blue;
    protected Real green;
  equation
    red + blue + green = 1;
end ColoredPointWithoutInheritance;
```

The inherited fields from `Point` keep their protection status since that `extends`-clause is preceded by `public`.

A protected element cannot be accessed via dot notation!
Advanced Topic

- Class parameterization

Generic Classes with Type Parameters

Formal class parameters are replaceable variable or type declarations within the class (usually) marked with the prefix `replaceable`.

Actual arguments to classes are modifiers, which when containing whole variable declarations or types are preceded by the prefix `redeclare`.

Equivalent to:

```plaintext
class C2 = class ColoredClass = BlueClass;

obj3 ColoredClass object
obj4 A red object
```

```plaintext
class C2 = class ColoredClass = GreenClass;

obj1 ColoredClass object
obj2 A yellow object
```
Class Parameterization when Class Parameters are Components

The class `ElectricalCircuit` has been converted into a parameterized generic class `GenericElectricalCircuit` with three formal class parameters $R_1$, $R_2$, $R_3$, marked by the keyword `replaceable`.

```plaintext
class ElectricalCircuit
    Resistor R1(R=100);
    Resistor R2(R=200);
    Resistor R3(R=300);
    Inductor L1;
    SineVoltage AC;
    Ground G;
    equation
        connect(R1.n,R2.n);
        connect(R1.n,L1.n);
        connect(R1.n,R3.n);
        connect(R1.p,AC.p);
        ....
    end ElectricalCircuit;
```

Class parameterization

```plaintext
class GenericElectricalCircuit
    replaceable Resistor R1(R=100);
    replaceable Resistor R2(R=200);
    replaceable Resistor R3(R=300);
    Inductor L1;
    SineVoltage AC;
    Ground G;
    equation
        connect(R1.n,R2.n);
        connect(R1.n,L1.n);
        connect(R1.n,R3.n);
        connect(R1.p,AC.p);
        ....
    end GenericElectricalCircuit;
```

Class Parameterization when Class Parameters are Components - cont'

A more specialized class `TemperatureElectricalCircuit` is created by changing the types of $R_1$, $R_3$ to `TempResistor`.

```plaintext
class TemperatureElectricalCircuit
    parameter Real Temp=20;
    extends GenericElectricalCircuit{
        redeclare TempResistor R1(RT=0.1, Temp=Temp),
        redeclare TempResistor R3(R=300);
    };
end TemperatureElectricalCircuit
```

We add a temperature variable `Temp` for the temperature of the resistor circuit and modifiers for $R_1$ and $R_3$ which are now `TempResistors`.

```plaintext
class ExpandedTemperatureElectricalCircuit
    parameter Real Temp;
    extends GenericElectricalCircuit{
        redeclare TempResistor R1(R=200, RT=0.1, Temp=Temp),
        replaceable Resistor R2,
        TempResistor R3(R=300);
    };
    equation
        ....
    end ExpandedTemperatureElectricalCircuit
```

```plaintext
class TemperatureElectricalCircuit = GenericElectricalCircuit (redeclare TempResistor R1
redeclare TempResistor R3);
```
Components, Connectors and Connections

Software Component Model

A component class should be defined *independently of the environment*, very essential for *reusability*. A component may internally consist of other components, i.e. *hierarchical* modeling. Complex systems usually consist of large numbers of *connected* components.
Connectors and Connector Classes

Connectors are instances of **connector classes**

- **Electrical connector**
  - **connector class**
  - **keyword** `flow` indicates that currents of connected pins sum to zero.
  - An instance `pin` of class `Pin`

- **Mechanical connector**
  - **connector class**
  - An instance `flange` of class `Flange`

The **flow prefix**

Two kinds of variables in connectors:
- **Non-flow variables** potential or energy level
- **Flow variables** represent some kind of flow

**Coupling**
- **Equality coupling**, for non-flow variables
- **Sum-to-zero coupling**, for flow variables

The value of a flow variable is **positive** when the current or the flow is into the component

**positive flow direction:**

\[ v \quad + \quad \text{pin} \]

\[ i \quad \text{pin} \]
Physical Connector
Classes Based on Energy Flow

<table>
<thead>
<tr>
<th>Domain Type</th>
<th>Potential</th>
<th>Flow</th>
<th>Carrier</th>
<th>Modelica Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electrical</td>
<td>Voltage</td>
<td>Current</td>
<td>Charge</td>
<td>Electrical, Analog</td>
</tr>
<tr>
<td>Translational</td>
<td>Position</td>
<td>Force</td>
<td>Linear momentum</td>
<td>Mechanical, Translational</td>
</tr>
<tr>
<td>Rotational</td>
<td>Angle</td>
<td>Torque</td>
<td>Angular momentum</td>
<td>Mechanical, Rotational</td>
</tr>
<tr>
<td>Magnetic</td>
<td>Magnetic potential</td>
<td>Magnetic flux rate</td>
<td>Magnetic flux</td>
<td></td>
</tr>
<tr>
<td>Hydraulic</td>
<td>Pressure</td>
<td>Volume flow</td>
<td>Volume</td>
<td>MyLibLight</td>
</tr>
<tr>
<td>Heat</td>
<td>Temperature</td>
<td>Heat flow</td>
<td>Heat</td>
<td>HeatFlow1D</td>
</tr>
<tr>
<td>Chemical</td>
<td>Chemical potential</td>
<td>Particle flow</td>
<td>Particles</td>
<td>Under construction</td>
</tr>
<tr>
<td>Pneumatic</td>
<td>Pressure</td>
<td>Mass flow</td>
<td>Air</td>
<td>PneuLibLight</td>
</tr>
</tbody>
</table>

Connects between connectors are realized as equations in Modelica

```
connect(connector1,connector2)
```

The two arguments of a connect-equation must be references to connectors, either to be declared directly within the same class or be members of one of the declared variables in that class.

Pin pin1,pin2;
//A connect equation
//in Modelica:
connect(connector1,connector2);

Corresponds to

```
pin1.v = pin2.v;
pin1.i + pin2.i =0;
```
Connection Equations

Pin pin1,pin2;
// A connect equation
// in Modelica
connect(pin1,pin2);
connect(pin1,pin3); ...
connect(pin1,pinN);

Multiple connections are possible:
connect(pin1,pin2); connect(pin1,pin3); ...
connect(pin1,pinN);

Each primitive connection set of nonflow variables is used to generate equations of the form:

\[ V_1 = V_2 = V_3 = \ldots V_n \]

Each primitive connection set of flow variables is used to generate sum-to-zero equations of the form:

\[ i_1 + i_2 + \ldots (-i_k) + \ldots i_n = 0 \]

Acausal, Causal, and Composite Connections

Two basic and one composite kind of connection in Modelica

- Acausal connections
- Causal connections, also called signal connections
- Composite connections, also called structured connections, composed of basic or composite connections
Common Component Structure

The base class TwoPin has two connectors p and n for positive and negative pins respectively

Partial model TwoPin

Voltage v
Current i

equation
v = p.v - n.v;
0 = p.i + n.i;
i = p.i;
end TwoPin;

// TwoPin is same as OnePort in
// Modelica.Electrical.Analog.Interfaces

Electrical Components

model Resistor "Ideal electrical resistor"
extends TwoPin;
parameter Real R;
equation
R*i = v;
end Resistor;

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

model Capacitor "Ideal electrical capacitor"
extends TwoPin;
parameter Real C;
equation
i=C*der(v);
end Inductor;
Electrical Components cont'

```plaintext
model Source
    extends TwoPin;
    parameter Real A, w;
    equation
        v = A*sin(w*time);
end Resistor;
```

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

Resistor Circuit

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

Corresponds to

```plaintext
R1.p.v = R2.p.v;
R1.p.v = R3.p.v;
R1.p.i + R2.p.i + R3.p.i = 0;
```
An Oscillating Mass Connected to a Spring

```model Oscillator
  Mass  mass1(L=1, s(start=-0.5));
  Spring spring1(srel0=2, c=10000);
  Fixed fixed1(s0=1.0);
  equation
    connect(spring1.flange_b, fixed1.flange_b);
    connect(mass1.flange_b, spring1.flange_a);
  end Oscillator;
```

Graphical Modeling Using Drag and Drop Composition

Courtesy MathCore Engineering AB
Exercise

- Locate the Oscillator model in DrModelica using OMNotebook!
- Simulate and plot the example. Do a slight change in the model e.g. different elasticity c, re-simulate and re-plot.

- Draw the Oscillator model using the graphic connection editor e.g. using the library Modelica. Mechanical.Translational
- Including components SlidingMass, Force, Blocks.Sources.Constant

- Simulate and plot!
Signal Based Connector Classes

connector InPort "Connector with input signals of type Real"
  parameter Integer n=1 "Dimension of signal vector";
  input Real signal[n] "Real input signals";
end InPort;

connector OutPort "Connector with output signals of type Real"
  parameter Integer n=1 "Dimension of signal vector";
  output Real signal[n] "Real output signals";
end OutPort;

fixed causality

partial block MISO
  "Multiple Input Single Output continuous control block"
  parameter Integer nin=1 "Number of inputs";
  InPort inPort(n=nin) "Connector of Real input signals";
  OutPort outPort(n=1) "Connector of Real output signal";
protected
  Real u[:] = inPort.signal "Input signals";
  Real y = outPort.signal[1] "Output signal";
end MISO; // From Modelica.Blocks.Interfaces

multiple input single output block

Connecting Components from Multiple Domains

• Block domain
• Mechanical domain
• Electrical domain

model Generator
  Modelica.Mechanics.Rotational.Inertia iner;
  Modelica.Electrical.Analog.Basic.EMF emf(k=-1);
  Modelica.Electrical.Analog.Basic.Inductor ind(L=0.1);
  Modelica.Electrical.Analog.Basic.Resistor R1,R2;
  Modelica.Blocks.Sources.Exponentials ex(riseTime={2},riseTimeConst={1});
equation
  connect(ac.flange_b, iner.flange_a);
  connect(iner.flange_b, emf.flange_b);
  connect(emf.p, ind.p);
  connect(ind.n, R1.p);
  connect(emf.n, G.p);
  connect(R1.n, R2.n);
  connect(R2.p, vsens.n);
  connect(ex.outPort, ac.inPort);
end Generator;
A DC motor can be thought of as an electrical circuit which also contains an electromechanical component.

```
model DCMotor
  Resistor R(R=100);
  Inductor L(L=100);
  VsourceDC DC(f=10);
  Ground G;
  EMF emf(k=10,J=10, b=2);
  Inertia load;
  equation
    connect(DC.p,R.n);
    connect(R.p,L.n);
    connect(L.p, emf.n);
    connect(emf.p, DC.n);
    connect(DC.n,G.p);
    connect(emf.flange,load.flange);
end DCMotor;
```

**Exercise**

- Draw the `DCMotor` model using the graphic connection editor using models from the following Modelica libraries:
  Mechanics.Rotational,
  Electrical.Analog.Basic,
  Electrical.Analog.Sources

- Simulate it for 15s and plot the variables for the outgoing rotational speed on the inertia axis and the voltage on the voltage source (denoted u in the figure) in the same plot.
Hierarchically Structured Components

An inside connector is a connector belonging to an internal component of a structured component class.

An outside connector is a connector that is part of the external interface of a structured component class, is declared directly within that class.

partial model PartialDCMotor
  InPort inPort;  // Outside signal connector
  RotFlange_b rotFlange_b;  // Outside rotational flange connector
  Inductor inductor1;
  Resistor resistor1;
  Ground ground1;
  EMF emf1;
  SignalVoltage signalVoltage1;
equation
  connect(inPort, signalVoltage1.inPort);
  connect(signalVoltage1.n, resistor1.p);
  connect(resistor1.n, inductor1.p);
  connect(signalVoltage1.p, ground1.p);
  connect(ground1.p, emf1.n);
  connect(inductor1.n, emf1.p);
end PartialDCMotor;

Hierarchically Structured Components cont'

model DCMotorCircuit2
  Step step1;
  PartialDCMotor partialDCMotor1;
  Inertia inertial1;
equation
  connect(step1.outPort, partialDCMotor1.inPort);
  connect(partialDCMotor1.rotFlange_b, inertial1.rotFlange_a);
end DCMotorCircuit2;
Connection Restrictions

- Two _acausal_ connectors can be connected to each other
- An _input_ connector can be connected to an _output_ connector or vice versa
- An _input_ or _output_ connector can be connected to an _acausal_ connector, i.e. a connector without input/output prefixes
- An _outside_ _input_ connector behaves approximately like an _output_ connector internally
- An _outside_ _output_ connector behaves approximately like an _input_ connector internally

A circuit consisting of four connected components C1, C2, C3, and C4 which are instances of the class C
Connector Restrictions cont’

A circuit in which the middle components C2 and C3 are placed inside a structured component M1 to which two outside connectors M1.u and M1.y have been attached.

Parameterization and Extension of Interfaces

The Tank model has an external interface in terms of the connectors inlet and outlet.

External interfaces to component classes are defined primarily through the use of connectors.
We would like to extend the Tank model to include temperature-dependent effects, analogous to how we extended a resistor to a temperature-dependent resistor.

```model HeatTank
  extends Tank
  redeclare connector TankStream = HeatStream;
  Real temp;
  equation
    // Energy balance for temperature effects
    Area*level*der(temp) = inlet.volumeFlowRate*inlet.temp + outlet.volumeFlowRate*outlet.temp;
    outlet.temp = temp; // Perfect mixing assumed.
  end HeatTank;
end
```

```connector HeatStream
  extends Stream;
  Real temp;
end HeatStream;
```

Cardinality-dependent Connection Equations

In certain cases there is a need to let the behavior of a model be dependent on the number of connections to certain connectors of the model. This can be achieved by using a built-in function `cardinality()` that returns the number of connections that have been made to a connector. (if-equations, see Lecture 4)

```model CardinalityResistor
  extends TwoPin;
  parameter Real R(unit="Ohm") "Resistance";
  equation
    // Handle cases if pins are not connected
    if cardinality(p) == 0 and cardinality(n) == 0 then
      p.v = 0; n.v = 0;
    elseif cardinality(p) == 0 then
      p.i = 0;
    elseif cardinality(n) == 0 then
      n.i = 0;
    end if
    // Resistor equation
    v = R*i;
  end CardinalityResistor;
```
Arrays of Connectors

Part built up with a for-equation (see Lecture 4)

```
model ArrayOfLinks
  constant Integer n=10 "Number of segments (>0)";
  parameter Real[3,n] r={fill(1,n),zeros(n),zeros(n)};
  ModelicaAdditions.MultiBody.Parts.InertialSystem InertialSystem1;
  ModelicaAdditions.MultiBody.Parts.BoxBody[n] boxBody(r = r, Width=fill(0.4,n));
  equation
    connect(InertialSystem1.frame_b, spherical[1].frame_a);
    connect(spherical[1].frame_b, boxBody[1].frame_a);
    for i in 1:n-1 loop
      connect(boxBody[i].frame_b, spherical[i+1].frame_a);
      connect(spherical[i+1].frame_b, boxBody[i+1].frame_a);
    end for;
end ArrayOfLinks;
```
Equations, Algorithms, and Functions

Equations

Usage of Equations

In Modelica equations are used for many tasks

- The main usage of equations is to represent relations in mathematical models.
- Assignment statements in conventional languages are usually represented as equations in Modelica.
- Attribute assignments are represented as equations.
- Connections between objects generate equations.
Equation Categories

Equations in Modelica can informally be classified into three different categories

- **Normal equations** (e.g., $expr1 = expr2$) occurring in equation sections, including `connect` equations and other equation types of special syntactic form

- **Declaration equations**, (e.g., Real $x = 2.0$) which are part of variable, parameter, or constant declarations

- **Modifier equations**, (e.g. $x($unit="V")$) which are commonly used to modify attributes of classes.

Constraining Rules for Equations

**Single Assignment Rule**

The total number of “equations” is identical to the total number of “unknown” variables to be solved for

**Synchronous Data Flow Principle**

- All variables keep their actual values until these values are explicitly changed
- At every point in time, during “continuous integration” and at event instants, the active equations express relations between variables which have to be fulfilled concurrently
  
  Equations are not active if the corresponding `if`-branch or `when`-equation in which the equation is present is not active because the corresponding branch condition currently evaluates to `false`

- Computation and communication at an event instant does not take time
Declaration Equations

It is also possible to specify a declaration equation for a normal non-constant variable:

```
Real speed = 72.4;
```

Modifier Equations

Modifier equations occur for example in a variable declaration when there is a need to modify the default value of an attribute of the variable. A common usage is modifier equations for the start attribute of variables.

```
Real speed(start=72.4);
```

Modifier equations also occur in type definitions:

```
type Voltage = Real(unit="V", min=-220.0, max=220.0);
```
Kinds of Normal Equations in Equation Sections

Kinds of equations that can be present in equation sections:
- equality equations
- connect equations
- assert and terminate
- reinit

model MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
  Rocket apollo(name="apollo13", mass(start=1038.358) );
  CelestialBody moon(mass=7.382e22, radius=1.738e6, name="moon");
equation
  if (time<thrustDecreaseTime) then
    apollo.thrust = force1;
  elseif (time<thrustEndTime) then
    apollo.thrust = force2;
  else
    apollo.thrust = 0;
  end if
  apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
end Landing;

Equality Equations

expr1 = expr2;
(out1, out2, out3,...) = function_name(in_expr1, in_expr2, ...);

class EqualityEquations
  Real x,y,z;
equation
  (x, y, z) = f(1.0, 2.0); // Correct!
  (x+1, 3.0, z/y) = f(1.0, 2.0); // Illegal!
end EqualityEquations;
Repetitive Equations

The syntactic form of a \texttt{for}-equation is as follows:

\begin{verbatim}
for <iteration-variable> in <iteration-set-expression> loop
  <equation1>
  ...
  <equation2>
end for;
\end{verbatim}

Consider the following simple example with a \texttt{for}-equation:

\begin{verbatim}
class FiveEquations
  Real[5] x;
  equation
    for i in 1:5 loop
      x[i] = i+1;
    end
  end FiveEquations;
end

class FiveEquationsUnrolled
  Real[5] x;
  equation
    x[1] = 2;
    x[2] = 3;
    x[3] = 4;
    x[4] = 5;
    x[5] = 6;
  end FiveEquationsUnrolled;
end
\end{verbatim}

Both classes have equivalent behavior!

In the class on the right the \texttt{for}-equation has been unrolled into five simple equations.

connect-equations

In Modelica \texttt{connect}-equations are used to establish connections between components via connectors:

\begin{verbatim}
connect(connector1,connector2)
\end{verbatim}

Repetitive \texttt{connect}-equations

\begin{verbatim}
class RegComponent
  Component components[n];
  equation
    for i in 1:n-1 loop
      connect(components[i].outlet,components[i+1].inlet);
    end
  end RegComponent;
\end{verbatim}
Conditional Equations: if-equations

**if-equations** for which the conditions have higher variability than constant or parameter must include an else-part

Each then-, elseif-, and else-branch must have the same number of equations

```model MoonLanding
  parameter Real force1 = 36350;
  ...
  Rocket apollo(name="apollo13", mass=start=1038.358);
  CelestialBody moon(mass=7.382e22, radius=1.738e6);
  equation
    if (time<thrustDecreaseTime) then
      apollo.thrust = force1;
    elseif (time<thrustEndTime) then
      apollo.thrust = force2;
    else
      apollo.thrust = 0;
    end if;
    apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
  end
end MoonLanding;
```

Conditional Equations: when-equations

```model MoonLanding
  parameter Real force1 = 36350;
  ...
  Rocket apollo(name="apollo13", mass=start=1038.358);
  CelestialBody moon(mass=7.382e22, radius=1.738e6);
  equation
    when (time<thrustDecreaseTime) then
      apollo.thrust = force1;
    elseif (time<thrustEndTime) then
      apollo.thrust = force2;
    else
      apollo.thrust = 0;
    end when;
    apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
  end
end MoonLanding;
```

**when-equations** in when-equations are instantaneous equations that are active at events when <conditions> become true

Events are ordered in time and form an event history:

- An event is a point in time that is instantaneous, i.e., has zero duration
- An event condition switches from false to true in order for the event to take place
Conditional Equations: when-equations cont'

when-equations are used to express instantaneous equations that are only valid (become active) at events, e.g. at discontinuities or when certain conditions become true.

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

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

when initial() then
  ... // Equations to be activated at the beginning of a simulation
end when;

when terminal() then
  ... // Equations to be activated at the end of a simulation
end when;

Restrictions on when-equations

Form restriction

Modelica restricts the allowed equations within a when-equation to: variable = expression, if-equations, for-equations,...

In the WhenNotValid model when the equations within the when-equation are not active it is not clear which variable, either x or y, that is a “result” from the when-equation to keep constant outside the when-equation.

A corrected version appears in the class WhenValidResult below

A corrected version appears in the class WhenValidResult below

A corrected version appears in the class WhenValidResult below

model WhenValidResult
  Real x,y;
  equation
    x + y = 5; // Equation to be used to compute x.
  when sample(0,2) then
    y = 7; // Correct, y is a result variable from the when;
  end when;
end WhenValidResult;

model WhenNotValid
  Real x,y;
  equation
    x + y = 5;
  when sample(0,2) then
    // Error: not valid Modelica
    // Equation to be used to compute x.
    y = 7; // Error: not valid Modelica
  end when;
end WhenNotValid;
Restrictions on *when*-equations cont’

Restriction on nested *when*-equations

```model ErrorNestedWhen

Real x,y1,y2;

equation

when x > 2 then

  when y1 > 3 then // Error!
    y2 = sin(x); // when-equations should not be nested
  end when;
end when;

end ErrorNestedWhen;
```

*when*-equations cannot be nested!

---

Restrictions on *when*-equations cont’

Single assignment rule: same variable may not be defined in several *when*-equations.

A conflict between the equations will occur if both conditions would become true at the same time instant.

```model DoubleWhenConflict

Boolean close; // Error: close defined by two equations!

equation

... when condition1 then // First equation
  close = true;
end when;
...

when condition2 then //Second equation
  close = false;
end when;

end DoubleWhenConflict
```
Restrictions on *when*-equations cont’

Solution to assignment conflict between equations in independent *when*-equations:

• Use `elseif` to give higher priority to the first *when*-equation

```
model DoubleWhenConflictResolved
    equation
        ... 
        when condition1 then
            close = true;  // First equation has higher priority!
        elseif condition2 then
            close = false;  // Second equation
        end when;
    end DoubleWhenConflictResolved
```

Restrictions on *when*-equations cont’

Vector expressions

The equations within a *when*-equation are activated when any of the elements of the vector expression becomes true

```
model VectorWhen
    Boolean close;
    equation
        when [condition1,condition2] then
            close = true;
        end when;
    end DoubleWhenConflictResolved
```
assert-equations

assert(assert-expression, message-string)

assert is a predefined function for giving error messages taking a Boolean condition and a string as an argument

The intention behind assert is to provide a convenient means for specifying checks on model validity within a model

class AssertTest
parameter Real lowlimit = -5;
parameter Real highlimit = 5;
Real x;
equation
assert(x >= lowlimit and x <= highlimit, "Variable x out of limit");
end AssertTest;

terminate-equations

The terminate-equation successfully terminates the current simulation, i.e. no error condition is indicated

model MoonLanding
parameter Real force1 = 36350;
parameter Real force2 = 1308;
parameter Real thrustEndTime = 210;
parameter Real thrustDecreaseTime = 43.2;
Rocket apollo(name="apollo13", mass(start=1038.358) );
CelestialBody moon(mass=7.382e22,radius=1.738e6,name="moon");
equation
apollo.thrust = if (time<thrustDecreaseTime) then force1
else if (time<thrustEndTime) then force2
else 0;
apollo.gravity = moon.g * moon.mass /(apollo.height + moon.radius)^2;
when apollo.height < 0 then // termination condition
terminate("The moon lander touches the ground of the moon");
end when;
end MoonLanding;
Algorithms and Functions

Algorithm Sections

Whereas equations are very well suited for physical modeling, there are situations where computations are more conveniently expressed as algorithms, i.e., sequences of instructions, also called statements.

```
algorithm
    <statements>
    <some keyword>
```

Algorithm sections can be embedded among equation sections:

```
equation
    x = y^2;
    z = w;
algorithm
    x1 := z+x;
    x2 := y-5;
    x1 := x2+y;
equation
    u = x1+x2;
...
```
### Iteration Using for-statements in Algorithm Sections

The general structure of a `for`-statement with a single iterator:

```plaintext
for <iteration-variable> in <iteration-set-expression> loop
    <statement1>
end for
```

A simple `for`-loop summing the five elements of the vector \( z \) within the class `SumZ`:

```plaintext
class SumZ
    parameter Integer n = 5;
    Real[n] z(start = [10, 20, 30, 40, 50]);
    Real sum;
    algorithm
    sum := 0;
    for i in 1:n loop
        // 1:5 is {1, 2, 3, 4, 5}
        sum := sum + z[i];
    end for;
end SumZ;
```

Examples of `for`-loop headers with different range expressions:

```plaintext
for k in 1:10+2 loop  // k takes the values 1, 2, 3, ..., 12
for i in [1, 3, 6, 7] loop  // i takes the values 1, 3, 6, 7
for r in 1.0 : 1.5 : 5.5 loop  // r takes the values 1.0, 2.5, 4.0, 5.5
for <iteration-variable> in <iteration-set-expression-
```

### Iterations Using while-statements in Algorithm Sections

The general structure of a `while`-loop with a single iterator:

```plaintext
while <conditions> loop
    <statement1>
end while
```

The example class `SumSeries` shows the `while`-loop construct used for summing a series of exponential terms until the loop condition is violated, i.e., the terms become smaller than \( \text{eps} \):

```plaintext
class SumSeries
    parameter Real eps = 1.E-6;
    Integer i;
    Real sum;
    Real delta;
    algorithm
    i := 1;
    delta := \exp(-0.01*i);
    while delta >= eps loop
        sum := sum + delta;
        i := i+1;
        delta := \exp(-0.01*i);
    end while;
end SumSeries;
```
if-statements

The general structure of if-statements. The elseif-part is optional and can occur zero or more times whereas the optional else-part can occur at most once.

The if-statements used in the class SumVector perform a combined summation and computation on a vector v.

class SumVector
    parameter Real v[5] = {100,200,-300,400,500};
    parameter Integer n = size(v,1);
    algorithm
        for i in 1:n loop
            if v[i] > 0 then
                sum := sum + v[i];
            elseif v[i] > -1 then
                sum := sum + v[i] - 1;
            else
                sum := sum - v[i];
            end if;
        end for;
end SumVector;

when-statements

when-statements are used to express actions (statements) that are only executed at events, e.g. at discontinuities or when certain conditions become true.

There are situations where several assignment statements within the same when-statement is convenient.

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

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

when [x > 2, sample(0.2), x < 5] then
    y1 := sin(x);
    y3 := 2*x + y1 + y2;
end when;

Algorithm and equation sections can be interleaved.
Function Declaration

The structure of a typical function declaration is as follows:

```
function <functionname>  
  input TypeI1 in1;  
  input TypeI2 in2;  
  input TypeI3 in3;  
  output TypeO1 out1;  
  output TypeO2 out2;  
protected  
  <local variables>  
algorithm  
  <statements>  
end <functionname>;
```

All internal parts of a function are optional, the following is also a legal function:

```
function <functionname>  
end <functionname>;
```

Modelica functions are **declarative mathematical functions**:

- Always return the same result(s) given the same input argument values

Function Call

Two basic forms of arguments in Modelica function calls:

- **Positional** association of actual arguments to formal parameters
- **Named** association of actual arguments to formal parameters

Example function called on next page:

```
function PolynomialEvaluator  
  input Real A[:];    // array, size defined at function call time  
  input Real x := 1.0;  // default value 1.0 for x  
  output Real sum;  
protected  
  Real xpower;          // local variable xpower  
algorithm  
  sum := 0;  
  xpower := 1;  
  for i in 1:size(A,1) loop  
    sum := sum + A[i]*xpower;  
    xpower := xpower*x;  
  end for;  
end PolynomialEvaluator;
```

The function `PolynomialEvaluator` computes the value of a polynomial given two arguments: a coefficient vector `A` and a value of `x`. 
Positional and Named Argument Association

Using *position*al association, in the call below the actual argument \( \{1,2,3,4\} \) becomes the value of the coefficient vector \( A \), and 21 becomes the value of the formal parameter \( x \).

```plaintext
... algorithm ...
  p := polynomialEvaluator({1,2,3,4},21)
```

The same call to the function `polynomialEvaluator` can instead be made using named association of actual parameters to formal parameters.

```plaintext
... algorithm ...
  p := polynomialEvaluator(A={1,2,3,4},x=21)
```

Functions with Multiple Results

```plaintext
function PointOnCircle "Computes cartesian coordinates of point"
input Real angle "angle in radians";
input Real radius;
output Real x;    // 1:st result formal parameter
output Real y;    // 2:nd result formal parameter
algorithm
  x := radius * cos(phi);
  y := radius * sin(phi);
end PointOnCircle;
```

Example calls:

```plaintext
(out1,out2,out3,...) = function_name(in1, in2, in3, in4, ...); // Equation
(out1,out2,out3,...) := function_name(in1, in2, in3, in4, ...); // Statement
(px,py) = PointOnCircle(1.2, 2);  // Equation form
(px,py) := PointOnCircle(1.2, 2); // Statement form
```

Any kind of variable of compatible type is allowed in the parenthesized list on the left hand side, e.g. even array elements:

```plaintext
[arr[1],arr[2]] := PointOnCircle(1.2, 2);
```
External Functions

It is possible to call functions defined outside the Modelica language, implemented in C or FORTRAN 77.

```modelica
function polynomialMultiply
    input Real a[:], b[:];
    output Real c[:]; = zeros(size(a,1)+size(b,1) - 1);
    external
end polynomialMultiply;
```

If no language is specified, the implementation language for the external function is assumed to be C. The external function `polynomialMultiply` can also be specified, e.g. via a mapping to a FORTRAN 77 function:

```modelica
function polynomialMultiply
    input Real a[:], b[:];
    output Real c[:]; = zeros(size(a,1)+size(b,1) - 1);
    external "FORTRAN 77"
end polynomialMultiply;
```
Discrete Events and Hybrid Systems

Events

Events are ordered in time and form an event history

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

Initialization actions are triggered by `initial()`

![Diagram of initial() event]

Actions at the end of a simulation are triggered by `terminal()`

![Diagram of terminal() event]

**Terminating a Simulation**

There `terminate()` function is useful when a wanted result is achieved and it is no longer useful to continue the simulation. The example below illustrates the use:

```model terminationModel
Real y;
equation
y = time;
when y > 5
then terminate("The time has elapsed 5s");
end
end terminationModel;
```

```
simulate(terminationModel, startTime = 0, stopTime = 10)
```

Simulation ends before reaching time 10.
Generating Repeated Events

The call `sample(t0,d)` returns true and triggers events at times `t0+i*d`, where `i=0,1,...`

```
class SamplingClock
  parameter Modelica.SIunits.Time  first,interval;
  Boolean clock;
  equation
    clock = sample(first,interval);
    when clock then
      ...
  end when;
end SamplingClock;
```

Expressing Event Behavior in Modelica

*if-equations, if-statements, and if-expressions* express different behavior in different operating regions

```
if <condition> then
  <equations>
else <condition> then
  <equations>
else
  <equations>
end if;
```

*when-equations* become active at events

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

```
model Diode "Ideal diode"
  extends TwoPin;
  Real s;
  Boolean off;
  equation
    off = s < 0;
    if off then
      v=s
    else
      v=0;
    end if;
    i = if off then 0 else s;
end Diode;
```
## Event Priority

Erroneous multiple definitions, single assignment rule violated

```model WhenConflictX
  discrete Real x;
  equation
  when time>=2 then // When A: Increase x by 1.5 at time=2
    x = pre(x)+1.5;
  end when;
  when time>=1 then // When B: Increase x by 1 at time=1
    x = pre(x)+1;
  end when;
end WhenConflictX;
```

Using event priority to avoid erroneous multiple definitions

```model WhenPriorityX
  discrete Real x;
  equation
  when time>=2 then // Higher priority
    x = pre(x)+1.5;
  elsewhen time>=1 then // Lower priority
    x = pre(x)+1;
  end when;
end WhenPriorityX;
```

## Obtaining Predecessor Values of a Variable Using pre()

At an event, `pre(y)` gives the previous value of `y` immediately before the event, except for event iteration of multiple events at the same point in time when the value is from the previous iteration.

- The variable `y` has one of the basic types `Boolean`, `Integer`, `Real`, `String`, or `enumeration`, a subtype of those, or an array type of one of those basic types or subtypes.
- The variable `y` is a discrete-time variable.
- The `pre` operator can *not* be used within a function.
Detecting Changes of Boolean Variables Using \( \text{edge()} \) and \( \text{change()} \)

Detecting changes of boolean variables using \( \text{edge()} \)

The expression \( \text{edge}(b) \) is true at events when \( b \) switches from false to true

Detecting changes of discrete-time variables using \( \text{change()} \)

The expression \( \text{change}(v) \) is true at instants when \( v \) changes value

Creating Time-Delayed Expressions

Creating time-delayed expressions using \( \text{delay()} \)

In the expression \( \text{delay}(v, d) \) \( v \) is delayed by a delay time \( d \)
A Sampler Model

```
model Sampler
  parameter Real sample_interval = 0.1;
  Real x(start=5);
  Real y;
  equation
    der(x) = -x;
    when [sample(0, sample_interval)] then
      y = x;
  end when;
end Sampler;
```

```
simulate(Sampler, startTime = 0, stopTime = 10)
plot({x,y})
```

Discontinuous Changes to Variables at Events via When-Equations/Statements

The value of a discrete-time variable can be changed by placing the variable on the left-hand side in an equation within a when-equation, or on the left-hand side of an assignment statement in a when-statement.

The value of a continuous-time state variable can be instantaneously changed by a reinit-equation within a when-equation.

```
model BouncingBall "the bouncing ball model"
  parameter Real g=9.81; //gravitational acc.
  parameter Real c=0.90; //elasticity constant
  Real x(start=0),y(start=10);
  equation
    der(x) = y;
    der(y)=-g;
    when x<0 then
      reinit(y, -c*y);
    end when;
end BouncingBall;
```
A Mode Switching Model Example

Elastic transmission with slack

DC motor transmission with elastic backlash

A finite state automaton

SimpleElastoBacklash model

---

A Mode Switching Model Example cont’

```plaintext
partial model SimpleElastoBacklash

Boolean backward, slack, forward;  // Mode variables
parameter Real b            "Size of backlash region";
parameter Real c = 1.e5     "Spring constant (c>0), N.m/rad";
parameter Real phi_rel0 = 0 "Angle when spring exerts no torque";
parameter Real phi_rel      "Relative rotation angle betw. flanges";
parameter Real phi_dev      "Angle deviation from zero-torque pos";
parameter Real tau          "Torque between flanges";

equation
phi_rel   = flange_b.phi - flange_a.phi;
phi_dev   = phi_rel - phi_rel0;
backward  = phi_rel < -b/2;    // Backward angle gives torque tau<0
forward   = phi_rel > b/2;     // Forward angle gives torque tau>0
slack     = not (backward or forward); // Slack angle gives no torque
tau      = if forward then
            c*(phi_dev - b/2)            // Backward angle gives positive driving torque
          else if backward then
            c*(phi_dev + b/2)     // Forward angle gives negative braking torque
          else
            0;                   // Slack angle gives zero torque
end SimpleElastoBacklash
```

---
A Mode Switching Model Example cont'

Relative rotational speed between the flanges of the Elastobacklash transmission

We define a model with less mass in inertia2 (J=1), no damping d=0, and weaker string constant c=1e-5, to show even more dramatic backlash phenomena.

The figure depicts the rotational speeds for the two flanges of the transmission with elastic backlash.

Water Tank System with PI Controller

```model TankPI
    LiquidSource source(flowLevel=0.02);
    Tank tank(area=1);
    PIContinuousController piContinuous(ref=0.25);
    equation
        connect(source.qOut, tank.qIn);
        connect(tank.tActuator, piContinuous.cOut);
        connect(tank.tSensor, piContinuous.cIn);
    end TankPI;
```

```model Tank
    ReadSignal tOut;  // Connector, reading tank level
    ActSignal tInp;  // Connector, actuator controlling input flow
    parameter Real flowVout = 0.01;  // [m^3/s]
    parameter Real area = 0.5;       // [m^2]
    parameter Real flowGain = 10;   // [m^2/s]
    Real h(start=0);                 // tank level [m]
    Real qIn;                        // flow through input valve [m^3/s]
    Real qOut;                       // flow through output valve [m^3/s]
    equation
        der(h)=(qIn-qOut)/area;           // mass balance equation
        qOut=if time>100 then flowVout else 0;
    end Tank;
```
Water Tank System with PI Controller – cont’

```model BaseController
    parameter Real Ts(unit = "s") = 0.1 "Time period between discrete samples";
    parameter Real K = 2 "Gain";
    parameter Real T(unit = "s") = 10 "Time constant";
    ReadSignal cin;
    AntSignal cOut;
    parameter Real ref;
    Real error;
    Real outCtr;
    equation
        error = ref - cin.val;
        cOut.act = outCtr;
    end BaseController;
```

```model PIdiscreteController
    extends BaseController(K = 2, T = 10);
    discrete Real x;
    equation
        when sample(0, Ts) then
            x = pre(x) + error * Ts / T;
            outCtr = K * (x + error);
        end when;
    end PIdiscreteController;
```

```model PIDcontinuousController
    extends BaseController(K = 2, T = 10);
    Real x;
    Real y;
    equation
        der(x) = error/T;
        y = T*der(error);
        outCtr = K * (error + x + y);
    end PIDcontinuousController;
```

Concurrency and Resource Sharing

Dining Philosophers Example

```model DiningTable
    parameter Integer n = 5 "Number of philosophers and forks";
    parameter Real sigma = 5 " Standard deviation for the random function";
    // Give each philosopher a different random start seed
    // Comment out the initializer to make them all hungry simultaneously.
    Philosopher phil[n](startSeed=[1:n,1:n,1:n], sigma=fill(sigma,n));
    Mutex mutex[n];
    Fork fork[n];
    equation
        for i in 1:n loop
            connect(phil[i].mutexPort, mutex.port[i]);
            connect(phil[i].right, fork[i].left);
            connect(fork[i].right, phil[mod(i, n) + 1].left);
        end for;
    end DiningTable;
```
Packages

Packages for Avoiding Name Collisions

- Modelica provide a safe and systematic way of avoiding name collisions through the package concept
- A package is simply a container or name space for names of classes, functions, constants and other allowed definitions
Packages as Abstract Data Type: Data and Operations in the Same Place

Keywords denoting a package
encapsulated makes package dependencies (i.e., imports) explicit

encapsulated package ComplexNumber

record Complex
  Real re;
  Real im;
end Complex;

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

function multiply
  input Complex x,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;

end ComplexNumbers

Usage of the ComplexNumber package

class ComplexUser

ComplexNumbers.Complex a(re=1.0, im=2.0);
ComplexNumbers.Complex b(re=1.0, im=2.0);

zeq := ComplexNumbers.multiply(a,b);
w := ComplexNumbers.add(a,b);
end ComplexUser

Accessing Definitions in Packages

- Access reference by prefixing the package name to definition names

- Shorter access names (e.g., Complex, multiply) can be used if definitions are first imported from a package (see next page).
Importing Definitions from Packages

The four forms of import are exemplified below assuming that we want to access the addition operation (add) of the package Modelica.Math.ComplexNumbers

```
import Modelica.Math.ComplexNumbers;      //Access as ComplexNumbers.add
import Modelica.Math.ComplexNumbers.add;  //Access as add
import Modelica.Math.ComplexNumbers.*    //Access as add
import Co = Modelica.Math.ComplexNumbers //Access as Co.add
```

Qualified Import

The qualified import statement

```
import <packagename>;
```

imports all definitions in a package, which subsequently can be referred to by (usually shorter) names

```
simplepackagename . definitionname,
```

where the simple package name is the packagename without its prefix.

```
encapsulated package ComplexUser1
import Modelica.Math.ComplexNumbers;
class User
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 User;
end ComplexUser1;
```

This is the most common form of import that eliminates the risk for name collisions when importing from several packages.
Single Definition Import

The single definition import of the form
import <packagename>.<definitionname>;
allows us to import a single specific definition (a constant or class but not a subpackage) from a package and use that definition referred to by its definitionname without the package prefix.

```
import ComplexNumbers.multiply;
import ComplexNumbers.add;
class User
    Complex a(x=1.0, y=2.0);
    Complex b(x=1.0, y=2.0);
    Complex z,w;
equation
    z = multiply(a,b);
    w = add(a,b);
end User;
```

There is no risk for name collision as long as we do not try to import two definitions with the same short name.

Unqualified Import

The unqualified import statement of the form
import packagename.*;
imports all definitions from the package using their short names without qualification prefixes.
Danger: Can give rise to name collisions if imported package is changed.

```
class ComplexUser3
    Complex a(x=1.0, y=2.0);
    Complex b(x=1.0, y=2.0);
    Complex z,w;
equation
    z = multiply(a,b);
    w = add(a,b);
end ComplexUser3;
```

This example also shows direct import into a class instead of into an enclosing package.
Renaming Import

The renaming import statement of the form:
import <shortpackagename> = <packagename>;
imports a package and renames it locally to shortpackagename.
One can refer to imported definitions using shortpackagename as
a presumably shorter package prefix.

```plaintext
class ComplexUser4
  Co.Complex a(x=1.0, y=2.0);
  Co.Complex b(x=1.0, y=2.0);
  Co.Complex z,w;
  equation
    z = Co.multiply(a,b);
    w = Co.add(a,b);
  end ComplexUser4;
```

This is as safe as qualified import but gives more
concise code

Package and Library Structuring

A well-designed package structure is one of the most
important aspects that influences the complexity,
understandability, and maintainability of 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.
- Dependencies on other packages.
Subpackages and Hierarchical Libraries

The main use for Modelica packages and subpackages is to structure hierarchical model libraries, of which the standard Modelica library is a good example.

```modelica
encapsulated package Modelica  // Modelica
encapsulated package Mechanics  // Modelica.Mechanics
      ...
    end Inertia;
      ...
    end Torque;
    ...
  end Rotational;
  ...
end Mechanics;
...
end Modelica;
```

Ecapsulated Packages and Classes

An encapsulated package or class prevents direct reference to public definitions outside itself, but as usual allows access to public subpackages and classes inside itself.

- Dependencies on other packages become explicit – more readable and understandable models!
- Used packages from outside must be imported.

```modelica
encapsulated model TorqueUserExample1
  import Modelica.Mechanics.Rotational;  // Import package Rotational
  Rotational.Torque t2;                 // Use Torque, OK!
  Modelica.Mechanics.Rotational.Inertia w2;
  //Error! No direct reference to the top-level Modelica package ...
  // to outside an encapsulated class
end TorqueUserExample1;
```
**within Declaration for Package Placement**

Use *short names* without dots when declaring the package or class in question, e.g. on a separate file or storage unit. Use `within` to specify within which package it is to be placed.

```modelica
within Modelica.Mechanics;
  import ...;
  connector Flange_a;
  ...
  end Flange_a;
  ...
end Interfaces;
model Inertia ...
  end Inertia;
  ...
end Rotational;
```

The `within` declaration states the prefix needed to form the fully qualified name.

The subpackage `Rotational` declared within `Modelica.Mechanics` has the fully qualified name `Modelica.Mechanics.Rotational`, by concatenating the `packageprefix` with the short name of the package.

---

**Mapping a Package Hierarchy into a Directory Hirarchy**

A Modelica package hierarchy can be mapped into a corresponding directory hierarchy in the file system.

```
C:\library
  \Modelica
    package.mo
  \Blocks
    package.mo
    Continuous.mo
    Interfaces.mo
  \Examples
    package.mo
    Example1.mo
  \Mechanics
    package.mo
    Rotational.mo
  ...
```

The `within Modelica.Mechanics` declares a package hierarchy that can be mapped into a directory hierarchy.
Mapping a Package Hierarchy into a Directory Hierarchy

It contains an empty Modelica package declaration since all subpackages under Modelica are represented as subdirectories of their own. The empty within statement can be left out if desired.

The subpackage Rotational stored as the file Rotational.mo. Note that Rotational contains the subpackage Interfaces, which also is stored in the same file since we chose not to represent Rotational as a directory.

C:\library
\Modelica
  \Blocks
    package.mo
    Continuous.mo
    Interfaces.mo
  \Examples
    \Mechanics
      package.mo
      ...
  ...

within
  encapsulated package Modelica
    "Modelica root package";
  end Modelica;

within Modelica.Blocks;
  encapsulated package Examples
    "Examples for Modelica.Blocks";
    import ...;
  end Examples;
  model Example1
    "Usage example 1 for Modelica.Blocks";
    ...;
  end Example1;

within Modelica.Mechanics;
  encapsulated package Rotational
  end Rotational;

  ...
end Rotational;

...
Modelica Libraries

Modelica Standard Library

*Modelica Standard Library* (called Modelica) is a standardized predefined package developed by Modelica Association.

It can be used freely for both commercial and noncommercial purposes under the conditions of *The Modelica License*.

Modelica libraries are available online including documentation and source code from http://www.modelica.org/library/library.html.
Modelica Standard Library cont’

Modelica Standard Library contains components from various application areas, with the following sublibraries:

- **Blocks**: Library for basic input/output control blocks
- **Constants**: Mathematical constants and constants of nature
- **Electrical**: Library for electrical models
- **Icons**: Icon definitions
- **Math**: Mathematical functions
- **Mechanics**: Library for mechanical systems
- **Media**: Media models for liquids and gases
- **Siunits**: Type definitions based on SI units according to ISO 31-1992
- **Stategraph**: Hierarchical state machines (analogous to Statecharts)
- **Thermal**: Components for thermal systems
- **Utility**: Utilities Utility functions especially for scripting

---

**Modelica.Blocks**

This library contains input/output blocks to build up block diagrams.

![Library Diagram](Image)

Example:

![Block Diagram](Image)
Modelica.Constants

A package with often needed constants from mathematics, machine dependent constants, and constants of nature.

Examples:

constant Real pi = 2*Modelica.Math.asin(1.0);

constant Real small = 1.e-60 "Smallest number such that small and ~small are representable on the machine";

constant Real G(final unit="m3/(kg.s2)") = 6.673e-11 "Newtonian constant of gravitation";

constant Real h(final unit="J.s") = 6.62606876e-34 "Planck constant";

constant Modelica.SIunits.CelsiusTemperature T_zero = -273.15 "Absolute zero temperature";

Modelica.Electrical

Electrical components for building analog, digital, and multiphase circuits

Examples:
**Modelica.Icons**

Package with icons that can be reused in other libraries

Examples:

![Info](image1)
![Library](image2)
![Library2](image3)
![Example](image4)

![RotationalSensor](image5)
![TranslationalSensor](image6)
![GearIcon](image7)
![MotorIcon](image8)

---

**Modelica.Math**

Package containing basic mathematical functions:

- \( \sin(u) \): sine
- \( \cos(u) \): cosine
- \( \tan(u) \): tangent  \((u \text{ shall not be: } \ldots, -\pi/2, \pi/2, 3\pi/2, \ldots)\)
- \( \arcsin(u) \): inverse sine \((-1 \leq u \leq 1)\)
- \( \arccos(u) \): inverse cosine \((-1 \leq u \leq 1)\)
- \( \arctan(u) \): inverse tangent
- \( \arctan2(u_1, u_2) \): four quadrant inverse tangent
- \( \sinh(u) \): hyperbolic sine
- \( \cosh(u) \): hyperbolic cosine
- \( \tanh(u) \): hyperbolic tangent
- \( \exp(u) \): exponential, base \(e\)
- \( \log(u) \): natural (base \(e\)) logarithm \((u > 0)\)
- \( \log_{10}(u) \): base 10 logarithm \((u > 0)\)
**Modelica.Mechanics**

Package containing components for mechanical systems

Subpackages:
- Rotational 1-dimensional rotational mechanical components
- Translational 1-dimensional translational mechanical components
- MultiBody 3-dimensional mechanical components

**Modelica.SIunits**

This package contains predefined types based on the international standard of units:

- ISO 31-1992 “General principles concerning quantities, units and symbols”
- ISO 1000-1992 “SI units and recommendations for the use of their multiples and of certain other units”.

A subpackage called **NonSIunits** is available containing non SI units such as Pressure_bar, Angle_deg, etc.
Modelica.Thermal

Subpackage **FluidHeatFlow** with components for heat flow modeling.

Sub package **HeatTransfer** with components to model 1-dimensional heat transfer with lumped elements

Example:

![Diagram of heat transfer system](image)

ModelicaAdditions Library (OLD)

**ModelicaAdditions** library contains additional Modelica libraries from DLR. This has been largely replaced by the new release of the Modelica 2.1 libraries.

Sublibraries:

- **Blocks** Input/output block sublibrary
- **HeatFlow1D** 1-dimensional heat flow (replaced by **Modelica.Thermal**)
- **Multibody** Modelica library to model 3D mechanical systems
- **PetriNets** Library to model Petri nets and state transition diagrams
- **Tables** Components to interpolate linearly in tables
**ModelicaAdditions.Multibody (OLD)**

This is a Modelica library to model 3D Mechanical systems including visualization.

New version has been released (March 2004) that is called Modelica.Mechanics.MultiBody in the standard library.

**Improvements:**
- Easier to use
- Automatic handling of kinematic loops.
- Built-in animation properties for all components.

---

**MultiBody (MBS) - Example Kinematic Loop**

Old library (cutjoint needed)  
New library (no cutjoint needed)
MultiBody (MBS) - Example Animations

ModelicaAdditions.PetriNets

This package contains components to model Petri nets

Used for modeling of computer hardware, software, assembly lines, etc
Power System Stability - ObjectStab

The ObjectStab package is a Modelica Library for Power Systems Voltage and Transient stability simulations.

Thermo-hydraulics Library - ThermoFluid

ThermoFluid is a Modelica base library for thermo-hydraulic models:

- Includes models that describe the basic physics of flows of fluid and heat, medium property models for water, gases and some refrigerants, and also simple components for system modeling.
- Handles static and dynamic momentum balances.
- Robust against backwards and zero flow.
- The discretization method is a first-order, finite volume method (staggered grid).
Vehicle Dynamics Library - VehicleDynamics

This library is used to model vehicle chassis

Some Other Free Libraries

- **ExtendedPetriNets** Petri nets and state transition diagrams (extended version)
- **QSSFluidFlow** Quasi Steady-State Fluid Flows
- **SystemDynamics** System Dynamics Formalism
- **Atplus** Building Simulation and Building Control (includes Fuzzy Control library)
- **ThermoPower** Thermal power plants
- **WasteWater** Library for biological wastewater treatment plants
- **SPICELib** Support modeling and analysis capabilities of the circuit simulator PSPICE

Read more about the libraries at www.modelica.org/library/library.html
**Hydraulics Library HyLib**

- Licensed Modelica package developed by Peter Beater
- More than 90 models for
  - Pumps
  - Motors and cylinders
  - Restrictions and valves
  - Hydraulic lines
  - Lumped volumes and sensors
- Models can be connected in an arbitrary way, e.g. in series or in parallel.
- **HyLibLight** is a free subset of HyLib
- More info: www.hylib.com

---

**HyLib - Example**

Hydraulic drive system with closed circuit
Pneumatics Library PneuLib

- Licensed Modelica package developed by Peter Beater
- More than 80 models for
  - Cylinders
  - Motors
  - Valves and nozzles
  - Lumped volumes
  - Lines and sensors
- Models can be connected in an arbitrary way, e.g. in series or in parallel.
- PneuLibLight is a free subset of HyLib.
- More info: www.pneulib.com

PneuLib - Example

Pneumatic circuit with multi-position cylinder, booster and different valves
Powertrain Library - Powertrain

• Licensed Modelica package developed by DLR
• Speed and torque dependent friction
• Bus concept
• Control units
• Animation

Some Modelica Applications
Example Fighter Aircraft Library

Custom made library, Aircraft*, for fighter aircraft applications

- Six degrees of freedom (6 DOF)
- Dynamic calculation of center of gravity (CoG)
- Use of Aerodynamic tables or mechanical rudders

*Property of FOI (The Swedish Defence Institute)

Aircraft with Controller

- Simple PID
- Controls alpha and height
Example Aircraft Animation

Animation of fighter aircraft with controller

Example Gas Turbine

42 MW gas turbine (GTX 100) from Siemens Industrial Turbomachinery AB, Finspång, Sweden
Example Gas Turbine

Example Gas Turbine – Load Rejection

Rotational speed (rpm) of the compressor shaft

- Load rejection
- Generator
- Switch pilot to main fuel
Example Gas Turbine – Load Rejection

Percentage of fuel valve opening (red = pilot, blue = main)

Generated power to the simulated electrical grid
Modeling and Simulation Environments

The Translation Process

Modelica Graphical Editor → Modelica Model → Modelica Source code

Translator → Flat model
Analyzer → Sorted equations
Optimizer → Optimized sorted equations
Code generator → C Code
C Compiler → Executable
Simulation
Commercial Environments – Dymola from Dynasim

3D Animations

Model diagrams

Equation editor

Courtesy of Dynasim AB, Sweden

Commercial Environments – MathModelica System Designer from MathCore

MathModelica Graphic editor

Simulation Center

Courtesy of Mathcore Engineering AB
OpenModelica Environment

The goal of the OpenModelica project is to:
- Create a complete Modelica modeling, compilation and simulation environment.
- Provide free software distributed in binary and source code form.
- Provide a modeling and simulation environment for research and industrial purposes.
- Develop a formal semantics of Modelica

Features of currently available implementation:
- Command shell environment allows to enter and evaluate Modelica declarations, expressions, assignments, and function calls.
- Modelica functions are implemented, including array support.
- Modelica equations are implemented, but with certain limitations.
- Packages, inheritance, modifiers, etc. are implemented.
- etc.
  
  http://www.ida.liu.se/~pelab/modelica/OpenModelica.html

OpenModelica Environment Architecture

http://www.ida.liu.se/projects/OpenModelica
OMNotebook Electronic Book with Modelica
Exercises and OMShell Interactive Shell

First Basic Class

1 HelloWorld

2 Simulations of HelloWorld

Examples of Applications
(usually using commercial tools)
Example - Modeling of a Wheel Loader Lifter

Simulation of a Wheel Loader Lifter
Modelica Simulation of AirCraft Dynamics

Developed by MathCore for the Swedish Defense Research Institute (FOI)

Modelica AirCraft Component Library

Model Structure – Using a Modelica AirCraft Component Library developed by MathCore for the Swedish Defense Research Institute (FOI)

Courtesy of Swedish Defense Research Institute (FOI)
PathWays in a Biochemical System

Examples of Modelica Research

- PDEs in Modelica
- Debugging
- Parallelization
- Language Design for Meta Programming
- Variant Handling
- Biochemical modeling
Extending Modelica with PDEs for 2D, 3D flow problems

class PDEModel
    HeatNeumann h_iso;
    Dirichlet h_heated(g=50);
    HeatRobin h_glass(h_Heat=30000);
    HeatTransfer ht;
    Rectangle2D dom;
equation
    dom.eq = ht;
    dom.left.bc = h_glass;
    dom.top.bc = h_iso;
    dom.right.bc = h_iso;
    dom.bottom.bc = h_heated;
end PDEModel;

Automatic Generation of Parallel Code from Modelica Equation-Based Models

Clustered Task Graph

Thermofluid Pipe Application
### Conclusions

Modelica has a good chance to become the next generation computational modeling language

Two complete commercial Modelica implementations currently available (MathModelica, Dymola), and an open source implementation (OpenModelica) under development
Contact

www.ida.liu.se/projects/OpenModelica
   Download OpenModelica and drModelica, book chapter

www.mathcore.com
   MathModelica Tool

www.mathcore.com/drModelica
   Book web page, Download book chapter

www.modelica.org
   Modelica Association

petfr@ida.liu.se
OpenModelica@ida.liu.se
Biological Models
Population Dynamics
Predator-Prey

Some Well-known Population Dynamics Applications

• Population Dynamics of Single Population

• Predator-Prey Models (e.g. Foxes and Rabbits)
Population Dynamics of Single Population

- \( P \) – population size = number of individuals in a population
- \( \dot{P} \) – population change rate, change per time unit
- \( g \) – growth factor of population (e.g. % births per year)
- \( d \) – death factor of population (e.g. % deaths per year)

\[
\text{growthrate} = g \cdot P \\
\text{deathrate} = d \cdot P \\
\dot{P} = \text{growthrate} - \text{deathrate}
\]

Exponentially increasing population if \((g-d)>0\)
Exponentially decreasing population if \((g-d)<0\)

Population Dynamics Model

- \( g \) – growth rate of population
- \( d \) – death rate of population
- \( P \) – population size

\[
\dot{P} = \text{growthrate} - \text{deathrate}
\]

class PopulationGrowth
  parameter Real g = 0.04 "Growth factor of population";
  parameter Real d = 0.0005 "Death factor of population";
  Real P(start=10) "Population size, initially 10";
  equation
    \( \text{der}(P) = (g-d) \cdot P; \)
end PopulationGrowth;
Simulation of PopulationGrowth

```modelica
simulate(PopulationGrowth, stopTime=100)
plot(P)
```

*Exponentially increasing population if (g-d)>0*

```
Exponentially increasing population if (g-d)>0
```

Population Growth Exercise!!

- Locate the PopulationGrowth model in DrModelica
- Change the initial population size and growth and death factors to get an exponentially decreasing population

```modelica
simulate(PopulationGrowth, stopTime=100)
plot(P)
```

*Exponentially decreasing population if (g-d)<0*

```
class PopulationGrowth
  parameter Real g = 0.04  "Growth factor of population";
  parameter Real d = 0.0005  "Death factor of population";
  Real           P(start=10) "Population size, initially 10";
  equation
    der(P) = (g-d)*P;
end PopulationGrowth;
```
Population Dynamics with both Predators and Prey Populations

- Predator-Prey models

Predator-Prey (Foxes and Rabbits) Model

- $R = \text{rabbits} = \text{size of rabbit population}$
- $F = \text{foxes} = \text{size of fox population}$
- $\dot{R} = \text{der(rabbits)} = \text{change rate of rabbit population}$
- $\dot{F} = \text{der(foxes)} = \text{change rate of fox population}$
- $g_r = g_r = \text{growth factor of rabbits}$
- $d_f = d_f = \text{death factor of foxes}$
- $d_{rf} = d_{rf} = \text{death factor of rabbits due to foxes}$
- $g_{fr} = g_{fr} = \text{growth factor of foxes due to rabbits and foxes}$

\[
\dot{R} = g_r \cdot R - d_{rf} \cdot F \cdot R \\
\dot{F} = g_{fr} \cdot d_{rf} \cdot R \cdot F - d_f \cdot F
\]

\[
\text{der(rabbits)} = g_r \cdot \text{rabbits} - d_{rf} \cdot \text{rabbits} \cdot \text{foxes}; \\
\text{der(foxes)} = g_{fr} \cdot d_{rf} \cdot \text{rabbits} \cdot \text{foxes} - d_f \cdot \text{foxes};
\]
## Predator-Prey (Foxes and Rabbits) Model

```
class LotkaVolterra

    parameter Real g_r = 0.04    "Natural growth rate for rabbits";
    parameter Real d_rf = 0.0005  "Death rate of rabbits due to foxes";
    parameter Real d_f = 0.09     "Natural death rate for foxes";
    parameter Real g_fr = 0.1     "Efficiency in growing foxes from rabbits";

    Real rabbits(start=700)      "Rabbits, (R) with start population 700";
    Real foxes(start=10)          "Foxes, (F) with start population 10";

equation

    der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
    der(foxes) = g_fr*d_rf*rabbits*foxes - d_f*foxes;

end LotkaVolterra;
```

---

## Simulation of Predator-Prey (LotkaVolterra)

```
simulate(LotkaVolterra, stopTime=3000)
plot({rabbits, foxes}, xrange={0,1000})
```

---

![Simulation graph](image)
Exercise of Predator-Prey

- Locate the LotkaVolterra model in DrModelica
- Change the death and growth rates for foxes and rabbits, simulate, and observe the effects

```plaintext
simulate(LotkaVolterra, stopTime=3000)
plot({rabbits, foxes}, xrange={0,1000})
```

```plaintext
class LotkaVolterra
  parameter Real g_r = 0.04    "Natural growth rate for rabbits";
  parameter Real d_rf = 0.0005  "Death rate of rabbits due to foxes";
  parameter Real d_f = 0.09    "Natural death rate for foxes";
  parameter Real g_fr = 0.1    "Efficiency in growing foxes from rabbits";
  Real rabbits(start=700) "Rabbits, (R) with start population 700";
  Real foxes(start=10)    "Foxes, (F) with start population 10";
  equation
    der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
    der(foxes)   = g_fr*d_rf*rabbits*foxes - d_f*foxes;
end LotkaVolterra;
```
Model Design

Modeling Approaches

- Traditional state space approach
- Traditional signal-style block-oriented approach
- Object-oriented approach based on finished library component models
- Object-oriented flat model approach
- Object-oriented approach with design of library model components
Modeling Approach 1

Traditional state space approach

---

Traditional State Space Approach

- Basic structuring in terms of subsystems and variables
- Stating equations and formulas
- Converting the model to state space form:
  \[ \dot{x}(t) = f(x(t), u(t)) \]
  \[ y(t) = g(x(t), u(t)) \]
Difficulties in State Space Approach

- The system decomposition does not correspond to the "natural" physical system structure
- Breaking down into subsystems is difficult if the connections are not of input/output type.
- Two connected state-space subsystems do not usually give a state-space system automatically.

Modeling Approach 2

Traditional signal-style block-oriented approach
Physical Modeling Style (e.g. Modelica) vs signal flow Block-Oriented Style (e.g. Simulink)

Modelica: Physical model – easy to understand

Block-oriented: Signal-flow model – hard to understand for physical systems

Traditional Block Diagram Modeling

- Special case of model components: the causality of each interface variable has been fixed to either input or output

Typical Block diagram model components:

- Integrator
- Adder
- Multiplier
- Function
- Branch Point

Simulink is a common block diagram tool
Physical Modeling Style (e.g. Modelica) vs signal flow Block-Oriented Style (e.g. Simulink)

Modelica:
Physical model – easy to understand

Block-oriented:
Signal-flow model – hard to understand for physical systems

Example Block Diagram Models

Electric

Control

Rotational Mechanics
Properties of Block Diagram Modeling

- The system decomposition topology does not correspond to the "natural" physical system structure
- Hard work of manual conversion of equations into signal-flow representation
- Physical models become hard to understand in signal representation
- Small model changes (e.g. compute positions from force instead of force from positions) requires redesign of whole model
- Block diagram modeling works well for control systems since they are signal-oriented rather than "physical"

Object-Oriented Modeling Variants

- Approach 3: Object-oriented approach based on finished library component models
- Approach 4: Object-oriented flat model approach
- Approach 5: Object-oriented approach with design of library model components
Object-Oriented Component-Based Approaches in General

• Define the system briefly
  • What kind of system is it?
  • What does it do?

• Decompose the system into its most important components
  • Define communication, i.e., determine interactions
  • Define interfaces, i.e., determine the external ports/connectors
  • Recursively decompose model components of “high complexity”

• Formulate new model classes when needed
  • Declare new model classes.
  • Declare possible base classes for increased reuse and maintainability

Top-Down versus Bottom-up Modeling

• Top Down: Start designing the overall view. Determine what components are needed.

• Bottom-Up: Start designing the components and try to fit them together later.
Approach 3: Top-Down Object-oriented approach using library model components

- Decompose into subsystems
- Sketch communication
- Design subsystems models by connecting library component models
- Simulate!

Decompose into Subsystems and Sketch Communication – DC-Motor Servo Example

Controller → Electrical Circuit → Rotational Mechanics

The DC-Motor servo subsystems and their connections
Modeling the Controller Subsystem

Modeling the controller

Modeling the Electrical Subsystem

Modeling the electric circuit
Modeling the Mechanical Subsystem

Modeling the mechanical subsystem including the speed sensor.

Object-Oriented Modeling from Scratch

- Approach 4: Object-oriented flat model approach
- Approach 5: Object-oriented approach with design of library model components
Example: OO Modeling of a Tank System

• The system is naturally decomposed into components

Object-Oriented Modeling

Approach 4: Object-oriented flat model design
Tank System Model FlatTank – No Graphical Structure

- No component structure
- Just flat set of equations
- Straightforward but less flexible, no graphical structure

```
model FlatTank
  // Tank related variables and parameters
  parameter Real flowLevel(unit="m3/s")=0.02;
  parameter Real area(unit="m2") =1;
  parameter Real flowGain(unit="m2/s") =0.05;
  Real h(start=0,unit="m") "Tank level";
  Real qInflow(unit="m3/s") "Flow through input valve";
  Real qOutflow(unit="m3/s") "Flow through output valve";
  // Controller related variables and parameters
  parameter Real K=2 "Gain";
  parameter Real T(unit="s")= 10 "Time constant";
  parameter Real minV=0, maxV=10; // Limits for flow output
  Real ref = 0.25 "Reference level for control";
  Real error "Deviation from reference level";
  Real outCtrl "Control signal without limiter";
  Real x; "State variable for controller";
  equation
    assert(minV>=0,"minV must be greater or equal to zero");
    der(h) = (qInflow-qOutflow)/area;   // Mass balance equation
    qInflow = if time>150 then 3*flowLevel else flowLevel;
    qOutflow = LimitValue(minV,maxV,-flowGain*outCtrl);
    error  = ref-h;
    der(x) = error/T;
    outCtrl = K*(error+x);
end FlatTank;
```

Simulation of FlatTank System

- Flow increase to flowLevel at time 0
- Flow increase to 3*flowLevel at time 150

```
simulate(FlatTank, stopTime=250)
plot(h, stopTime=250)
```
Object-Oriented Modeling

• Approach 5: Object-oriented approach with design of library model components

Object Oriented Component-Based Approach
Tank System with Three Components

• Liquid source
• Continuous PI controller
• Tank

```model TankPI
  LiquidSource    source(flowLevel=0.02);
  PIContinuousController piContinuous(ref=0.25);
  Tank            tank(area=1);
equation
  connect(source.qOut, tank.qIn);
  connect(tank.tActuator, piContinuous.cOut);
  connect(tank.tSensor, piContinuous.cIn);
end TankPI;
```
Tank model

- The central equation regulating the behavior of the tank is the mass balance equation (input flow, output flow), assuming constant pressure.

```model Tank
  ReadSignal tSensor "Connector, sensor reading tank level (m)";
  ActSignal tActuator "Connector, actuator controlling input flow";
  LiquidFlow qIn "Connector, flow (m3/s) through input valve";
  LiquidFlow qOut "Connector, flow (m3/s) through output valve";
  parameter Real area(unit="m2") = 0.5;
  parameter Real flowGain(unit="m2/s") = 0.05;
  parameter Real minV=0, maxV=10; // Limits for output valve flow
  Real h(start=0.0, unit="m") "Tank level";
  equation
    assert(minV>0,"minV - minimum Valve level must be > 0 ");
    der(h) = (qIn.lflow-qOut.lflow)/area; // Mass balance
    equation
      qOut.lflow = LimitValue(minV,maxV,-flowGain*tActuator.act);
      tSensor.val = h;
  end Tank;
```

Connector Classes and Liquid Source Model for Tank System

```connector ReadSignal "Reading fluid level"
  Real val(unit="m");
end ReadSignal;

connector ActSignal "Signal to actuator for setting valve position"
  Real act;
end ActSignal;

connector LiquidFlow "Liquid flow at inlets or outlets"
  Real lflow(unit="m3/s");
end LiquidFlow;

model LiquidSource
  LiquidFlow qOut;
  parameter flowLevel = 0.02;
  equation
    qOut.lflow = if time>150 then 3*flowLevel else flowLevel;
end LiquidSource;
```
Continuous PI Controller for Tank System

- error = (reference level – actual tank level)
- T is a time constant
- x is controller state variable
- K is a gain factor

\[ dx = \frac{\text{error}}{T} \]
\[ \text{outCtr} = K \times (\text{error} + x) \]

Integrating equations gives
Proportional & Integrative (PI)

\[ \text{outCtr} = K \times \text{error} + \int \frac{\text{error}}{T} \, dt \]

The Base Controller – A Partial Model

- error = difference between reference level and actual tank level from cIn connector

partial model BaseController
parameter Real Ts(unit="s")=0.1
    "Ts - Time period between discrete samples - discrete sampled";
parameter Real K=2  "Gain";
parameter Real T=10(unit="s")  "Time constant - continuous";
ActSignal cOut "Control to actuator, connector";
ReadSignal cIn  "Input sensor level, connector";
parameter Real ref  "Reference level";
Real error "Deviation from reference level";
Real outCtr "Output control signal";
equation
    error  = ref-cIn.val;
    cOut.act  = outCtr;
end BaseController;
Simulate Component-Based Tank System

• As expected (same equations), TankPI gives the same result as the flat model FlatTank

```plaintext
simulate(TankPI, stopTime=250)
plot(h, stopTime=250)
```

Flexibility of Component-Based Models

• Exchange of components possible in a component-based model

• Example:
  Exchange the PI controller component for a PID controller component
Tank System with Continuous PID Controller
Instead of Continuous PI Controller

- Liquid source
- Continuous PID controller
- Tank

### Continuous PID Controller

- \( \text{error} = (\text{reference level} - \text{actual tank level}) \)
- \( T \) is a time constant
- \( x, y \) are controller state variables
- \( K \) is a gain factor

\[
\frac{dx}{dt} = \frac{error}{T} \\
y = T \frac{derror}{dt} \\
outCtr = K \cdot (error + x + y)
\]

Integrating equations gives Proportional & Integrative & Derivative (PID)

\[
outCtr = K \cdot (error + \int \frac{error}{T} \, dt + T \frac{derror}{dt})
\]

**model PIDcontinuousController**

extends BaseController(K=2,T=10);

Real x; // State variable of continuous PID controller

Real y; // State variable of continuous PID controller

equation

der(x) = error/T;
y = T*der(error);
outCtr = K*(error + x + y);
end PIDcontinuousController;

---

Copyright © Peter Fritzson
Simulate TankPID and TankPI Systems

- TankPID with the PID controller gives a slightly different result compared to the TankPI model with the PI controller

```plaintext
simulate(compareControllers, stopTime=250)
plot({tankPI.h, tankPID.h})
```

Two Tanks Connected Together

- Flexibility of component-based models allows connecting models together

```plaintext
model TanksConnectedPI
    LiquidSource  source(flowLevel=0.02);
    Tank          tank1(area=1), tank2(area=1.3);
    PIcontinuousController piContinuous1(ref=0.25), piContinuous2(ref=0.4);
    equation
        connect(source.qOut, tank1.qIn);
        connect(tank1.tActuator, piContinuous1.cOut);
        connect(tank1.tSensor, piContinuous1.cIn);
        connect(tank1.qOut, tank2.qIn);
        connect(tank2.tActuator, piContinuous2.cOut);
        connect(tank2.tSensor, piContinuous2.cIn);
end TanksConnectedPI;
```
Simulating Two Connected Tank Systems

- Fluid level in tank2 increases after tank1 as it should
- Note: tank1 has reference level 0.25, and tank2 ref level 0.4

```model TanksConnectedPI
    parameter Real Ts(unit = "s") = 0.1 "Time period between discrete samples";
    parameter Real K = 2 "Gain";
    parameter Real T(unit = "s") = 10 "Time constant";
    Real ref "Reference level";
    Real error "Deviation from reference level";
    Real outCtr "Output control signal";

    equation
        error = ref - cIn.val;
        outCtr = K * (error + x + y);
    end TanksConnectedPI;
```

Exchange: Either PI Continous or PI Discrete Controller

```partial model BaseController
    parameter Real Ts(unit = "s") = 0.1 "Time period between discrete samples";
    parameter Real K = 2 "Gain";
    parameter Real T(unit = "s") = 10 "Time constant";
    parameter Real ref "Reference level";
    Real error "Deviation from reference level";
    Real outCtr "Output control signal";

    equation
        error = ref - cIn.val;
        outCtr = K * (error + x);
    end BaseController;

model PIdiscreteController
    extends BaseController(K = 2, T = 10);
    discrete Real x;

    equation
        when sample(0, Ts) then
            x = pre(x) + error * Ts / T;
            outCtr = K * (x + error);
    end when;
    end PIdiscreteController;

model PIDcontinuousController
    extends BaseController(K = 2, T = 10);
    Real x;
    Real y;

    equation
        der(x) = error / T;
        y = T * der(error);
        outCtr = K * (error + x + y);
    end PIDcontinuousController;
```
Exercises

- Replace the PI continuous controller by the PI discrete controller and simulate. (see also the book, page 461)
- Create a tank system of 3 connected tanks and simulate.

Principles for Designing Interfaces – i.e., Connector Classes

- Should be easy and natural to connect components
  - For interfaces to models of physical components it must be physically possible to connect those components
- Component interfaces to facilitate reuse of existing model components in class libraries
- Identify kind of interaction
  - If there is interaction between two physical components involving energy flow, a combination of one potential and one flow variable in the appropriate domain should be used for the connector class
  - If information or signals are exchanged between components, input/output signal variables should be used in the connector class
- Use composite connector classes if several variables are needed
Simplification of Models

• When need to simplify models?
  • When parts of the model are too complex
  • Too time-consuming simulations
  • Numerical instabilities
  • Difficulties in interpreting results due to too many low-level model details

• Simplification approaches
  • Neglect small effects that are not important for the phenomena to be modeled
  • Aggregate state variables into fewer variables
  • Approximate subsystems with very slow dynamics with constants
  • Approximate subsystems with very fast dynamics with static relationships, i.e. not involving time derivatives of those rapidly changing state variables
Exercises Using OpenModelica and MathModelica Lite

Version 2006-09-17

Peter Fritzson
PELAB – Programming Environment Laboratory
SE-581 83 Linköping, Sweden

1 Simple Textual Modelica Modeling Exercises

1.1 Try DrModelica with VanDerPol
Locate the VanDerPol model in DrModelica (link from Section 2.1), run it, change it slightly, and re-run it.

1.2 HelloWorld
Simulate and plot the following example with one differential equation and one initial condition. Do a slight change in the model, re-simulate and re-plot.

```model HelloWorld "A simple equation"
Real x(start=1);

equation
  der(x) = -x;
end HelloWorld;
```

1.3 BouncingBall
Locate the BouncingBall model in one of the hybrid modeling sections of DrModelica (e.g. Section 2.9), run it, change it slightly, and re-run it.

1.4 A Simple Equation System
Make a Modelica model that solves the following equation system with initial conditions:

\[
\begin{align*}
  x &= 2 + xy - 3 + x \\
  y &= 5 + y - 2 + x + y \\
  x(0) &= 2 \\
  y(0) &= 3
\end{align*}
\]

1.5 Functions and Algorithm Sections
a) Write a function, `sum`, which calculates the sum of Real numbers, for a vector of arbitrary size.

b) Write a function, `average`, which calculates the average of Real numbers, in a vector of arbitrary size. The function `average` should make use of a function call to `sum`. 
2 Graphical Design using MathModelica Lite

2.1 Simple DC-Motor

Make a simple DC-motor using the Modelica standard library that has the following structure:

![Diagram of a simple DC-motor](image)

Save the model from the graphic editor, load it and simulate it (using OMShell or OMNotebook) for 15s and plot the variables for the outgoing rotational speed on the inertia axis and the voltage on the voltage source (denoted $u$ in the figure) in the same plot.

Hint: if you have difficulty finding the names of the variables to plot, you can flatten the model by calling instantiateModel, which exposes all variable names.

2.2 DC-Motor with Spring and Inertia

Add a torsional spring to the outgoing shaft and another inertia element. Simulate again and see the results. Adjust some parameters to make a rather stiff spring.

![Diagram of a DC-motor with spring and inertia](image)

2.3 DC-Motor with Controller (Extra)

Add a PI controller to the system and try to control the rotational speed of the outgoing shaft. Verify the result using a step signal for input. Tune the PI controller by changing its parameters in MathModelica Lite.