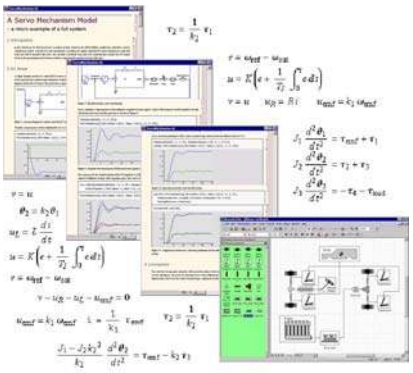
Principles of Object-Oriented Modeling and Simulation with Modelica



Peter Fritzson

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

Tutorial Presented by:

Håkan Lundvall

Linköping University
haklu@ida.liu.se

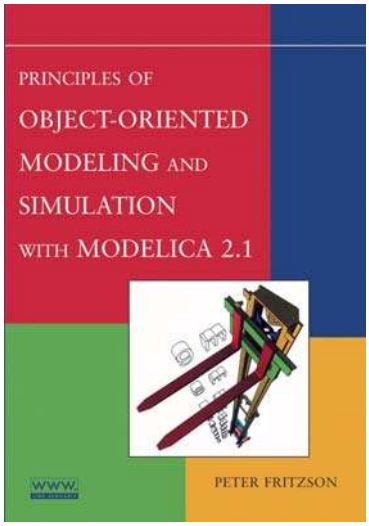
Jan Brugård

MathCore Engineering AB
Jan.Brugard@mathcore.com

pelab

MODELICA

Course Based on Recent Book, 2004



Peter Fritzson

Principles of Object Oriented Modeling and Simulation with Modelica 2.1

Wiley-IEEE Press

940 pages

Copyright © Peter Fritzson

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, and MathCore
- Modelica Association: www.modelica.org
- OpenModelica: www.ida.liu.se/projects/OpenModelica

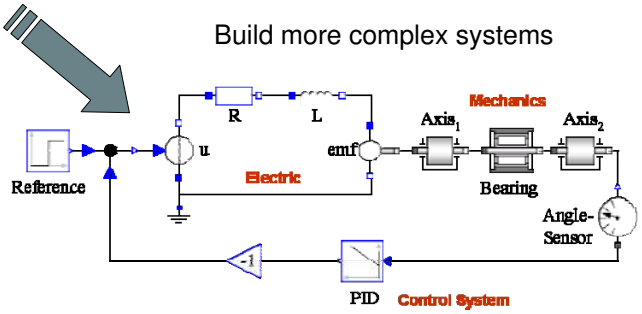
Content

- Modelica – The Next Generation Modeling Language
- OpenModelica Environment
- The Modelica Language – Modelica Classes and Inheritance
  - This section including hands-on exercises on textual modeling using the **OpenModelica** environment
- MathModelica Environment
- Components, Connectors and Connections – Modelica Libraries and Graphical Modeling
- Graphical Modeling Exercises
  - This section including hands-on exercises on graphical modeling using the **MathModelica** environment

Why Modeling & Simulation ?

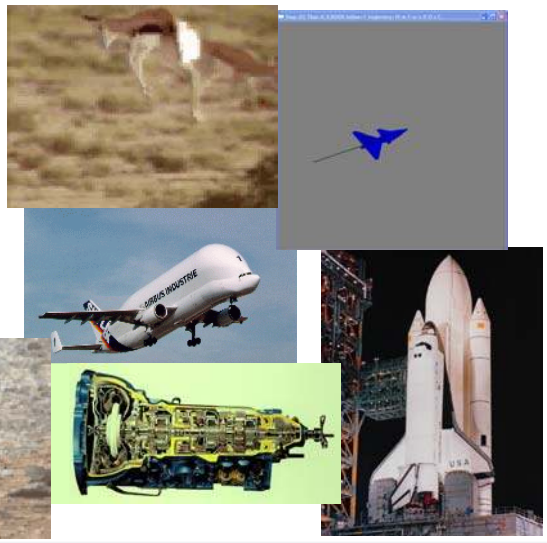
- Increase understanding of complex systems
- Design and optimization
- Virtual prototyping
- Verification

Build more complex systems



Examples of Complex Systems

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



Kinds of Mathematical Models

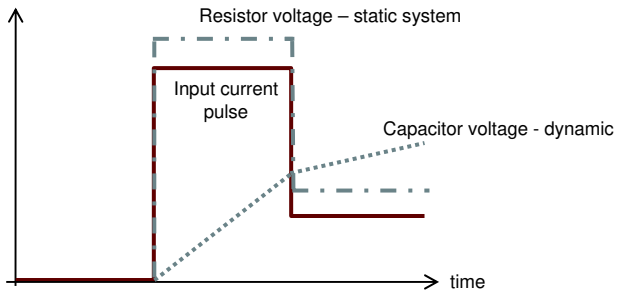
- Dynamic *vs.* Static models
- Continuous-time *vs.* Discrete-time dynamic models
- Quantitative *vs.* Qualitative models

 Copyright © Peter Fritzson

Dynamic vs. Static Models

A **dynamic** model includes *time* in the model

A **static** model can be defined *without* involving *time*

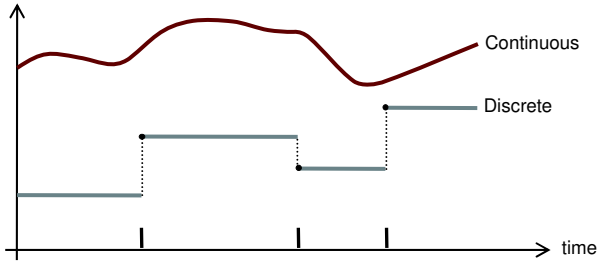


 Copyright © Peter Fritzson

Continuous-Time vs. Discrete-Time Dynamic Models

Continuous-time models may evolve their variable values *continuously* during a time period

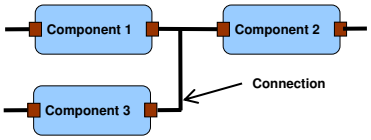
Discrete-time variables change values a *finite* number of times during a time period



 Copyright © Peter Fritzson

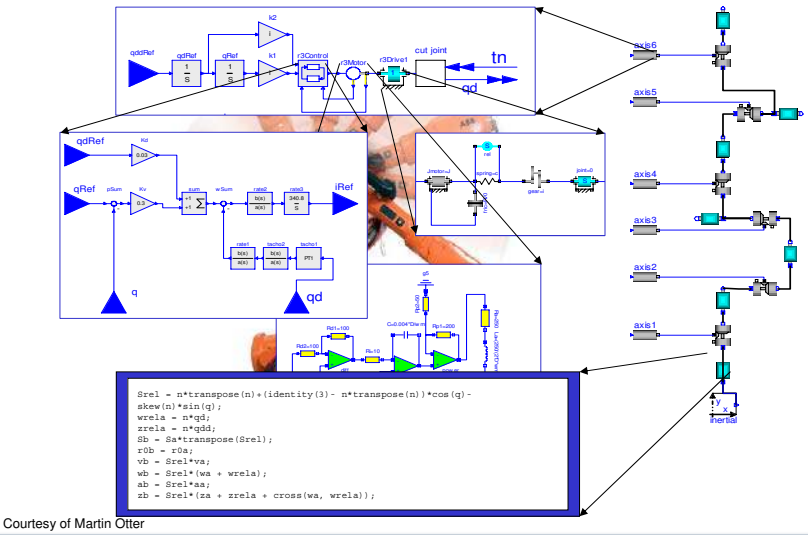
Principles of Graphical Equation-Based Modeling

- 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



 Copyright © Peter Fritzson

Application Example – Industry Robot

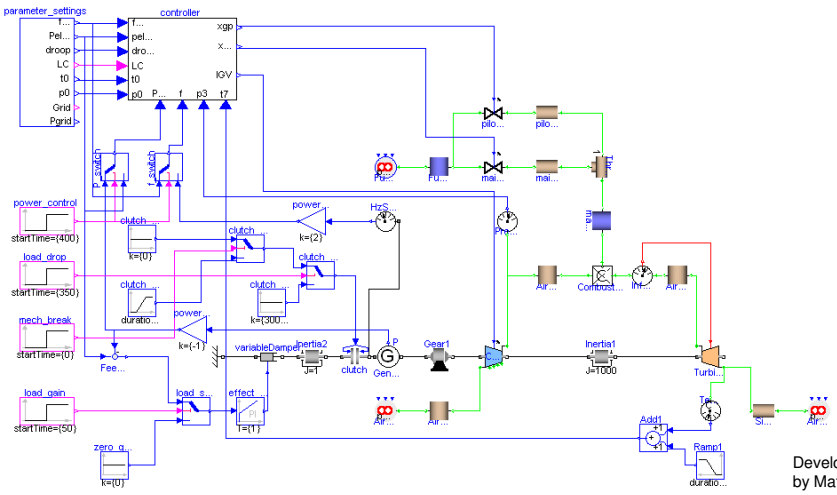


Courtesy of Martin Otter

 Copyright © Peter Fritzson

MODELICA pelab

GTX Gas Turbine Power Cutoff Mechanism



Developed by MathCore for Siemens

Courtesy of Siemens Industrial Turbomachinery AB

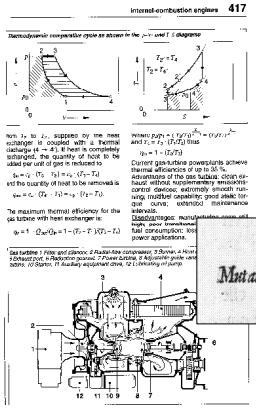
 Copyright © Peter Fritzson

MathCore MODELICA pelab

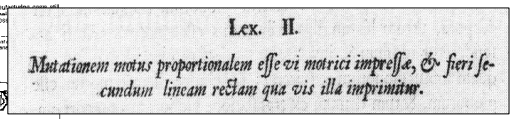
Modelica – The Next Generation Modeling Language

Stored Knowledge

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

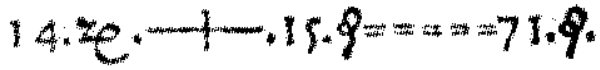


"The change of motion is proportional to the motive force impressed"
– Newton


Lex. II.
Mutationem motus proportionalem esse vi motrici impressæ, & fieri secundum lineam rectam qua vis illa imprimitur.

The Form – Equations

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



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":

```
variable = expression
v = INTEG(F)/m
```

Programming languages usually do not allow equations!

 Copyright © Peter Fritzson

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

 Copyright © Peter Fritzson

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

 Copyright © Peter Fritzson

Modelica Acausal Modeling

- What is *acausal* modeling/design?
- Why does it increase *reuse*?

  The acausality makes Modelica library classes *more reusable* than traditional classes containing assignment statements where the input-output causality is fixed.

- Example: a resistor *equation*:

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

- can be used in three ways:

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

 Copyright © Peter Fritzson

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
  - Next version, 3.0 to be released fall 2007
- Modelica Association established 2000
  - Open, non-profit organization

 Copyright © Peter Fritzson

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 September 4-5, 2006 in Vienna, Austria
- Coming: 6th International Modelica conference March 3-4, 2008 in Bielefeld, Germany

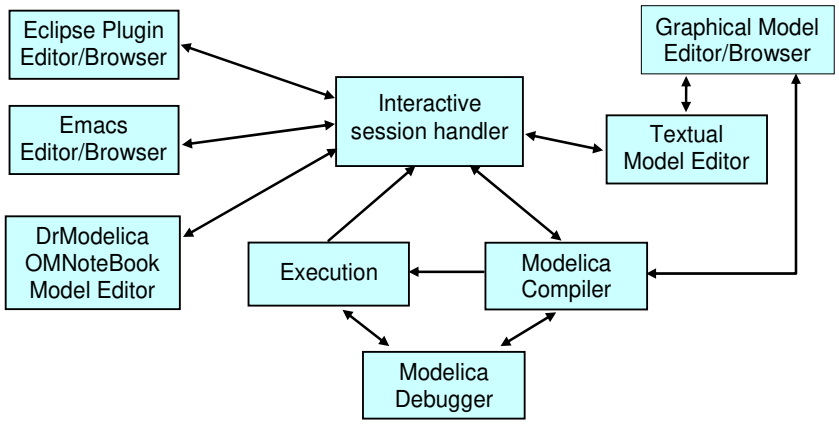
 Copyright © Peter Fritzson

OpenModelica Environment

OpenModelica

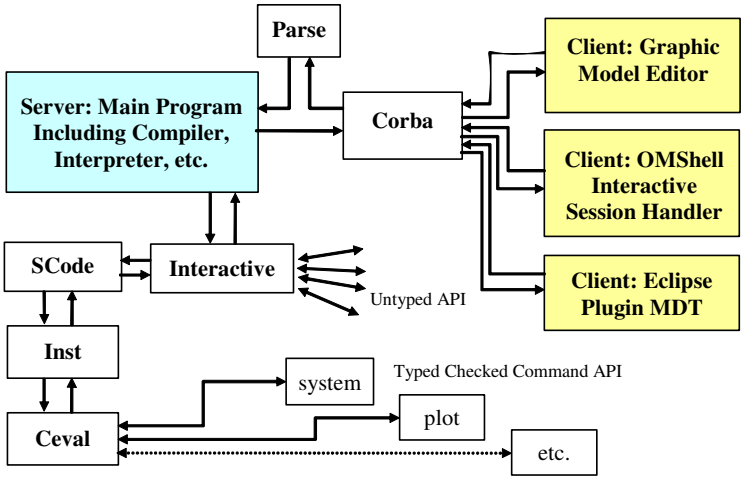
- Goal: comprehensive modeling and simulation environment for research, teaching, and industrial usage
- Free, open-source for both academic and commercial use
- Available under Berkley New BSD open source license
- The OpenModelica compiler (OMC) implemented in MetaModelica, a slightly extended Modelica
- Invitation for open-source cooperation around OpenModelica, tools, and applications

OpenModelica Environment Architecture



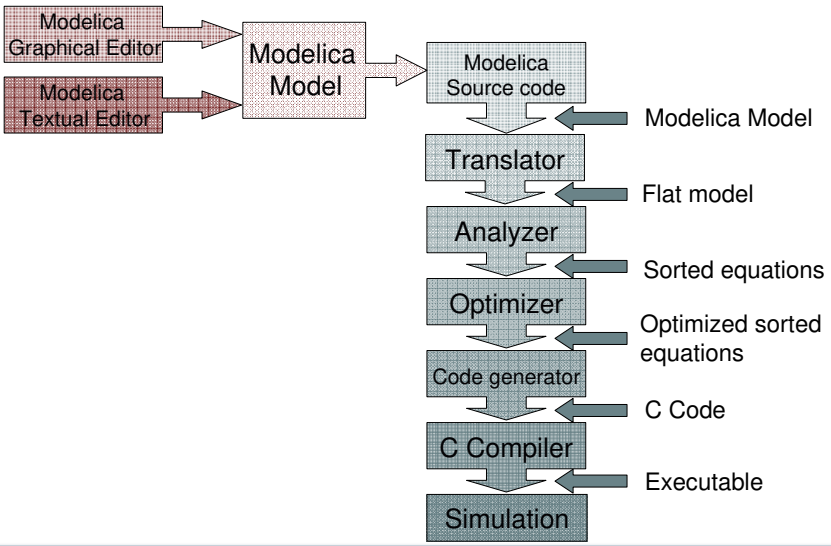
 Copyright © Peter Fritzson

OpenModelica Client-Server Architecture



 Copyright © Peter Fritzson

The Compiler (OMC) Translates Model to Simulation Code



 Copyright © Peter Fritzson

Released in OpenModelica 1.4.3

- OpenModelica compiler/interpreter – OMC
- Interactive session handler – OMShell
- OpenModelica Notebook with DrModelica – OMNotebook
- OpenModelica Eclipse plugin MDT
- Available: MathModelica Lite graphic editor (not part of OpenModelica)

 Copyright © Peter Fritzson

Platforms

- All OpenModelica GUI tools (OMShell, OMNotebook, ...) are developed on the Qt4 GUI library, portable between Windows, Linux, Mac
- Both compilers (OMC, MMC) are portable between the three platforms

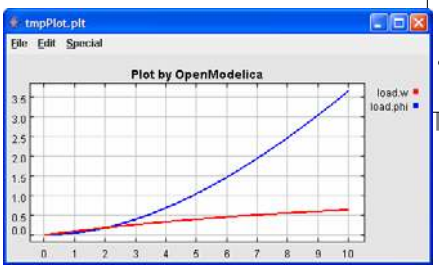
- Windows – currently main development and release platform
- Linux – available
- Mac – recently available

 Copyright © Peter Fritzson

Interactive Session Handler – on dcmotor Example
(Session handler called OMShell – OpenModelica Shell)

```
>>simulate(dcmotor,startTime=0.0,stopTime=10.0)
>>plot({load.w,load.phi})
```

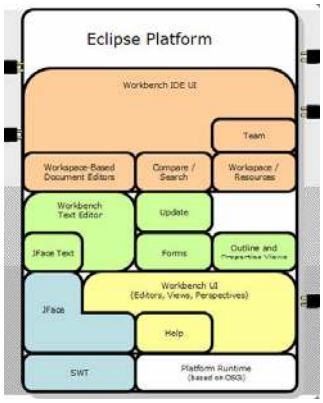
```
model dcmotor
  Modelica.Electrical.Analog.Basic.Resistor r1(R=10);
  Modelica.Electrical.Analog.Basic.Inductor i1;
  Modelica.Electrical.Analog.Basic.EMF emf1;
  Modelica.Mechanics.Rotational.Inertia load;
  Modelica.Electrical.Analog.Basic.Ground g;
  Modelica.Electrical.Analog.Sources.ConstantVoltage v;
equation
  connect(v.p,r1.p);
  connect(v.n,g.p);
  connect(r1.n,i1.p);
  connect(i1.n,emf1.p);
  connect(emf1.n,g.p);
  connect(emf1.flange_b,load.flange_a);
end dcmotor;
```



 Copyright © Peter Fritzson

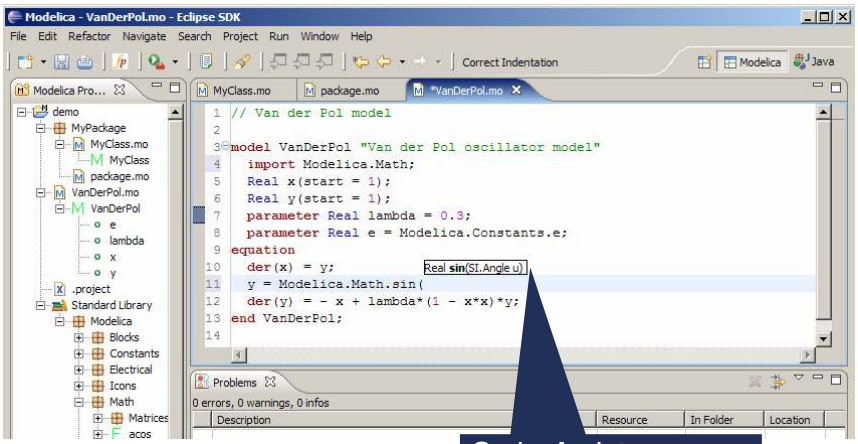
OpenModelica MDT – Eclipse Plugin

- Browsing of packages, classes, functions
- Automatic building of executables; separate compilation
- Syntax highlighting
- Code completion,
  Code query support for developers
- Automatic Indentation
- Debugger
  (Prel. version for algorithmic subset)



 Copyright © Peter Fritzson

OpenModelica Eclipse Plugin – Usage Example



Code Assistance on function calling.

 Copyright © Peter Fritzson

The Modelica Language – Modelica Classes and Inheritance

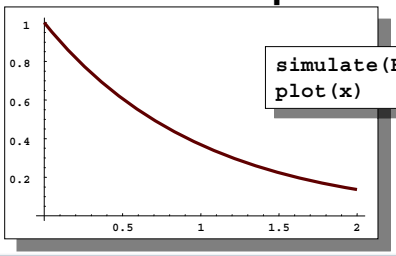
Simplest Model – Hello World!

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)
```

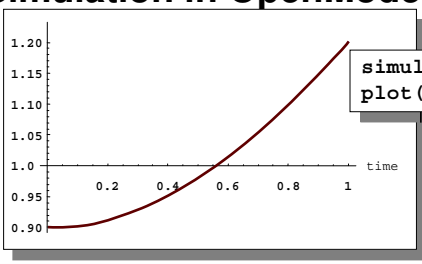
Another Example

Include algebraic equation

Algebraic equations contain no derivatives

```
class DAEexample
  Real x(start=0.9);
  Real y;
equation
  der(y)+(1+0.5*sin(y))*der(x)
    = sin(time);
  x - y = exp(-0.9*x)*cos(y);
end DAEexample;
```

Simulation in OpenModelica environment



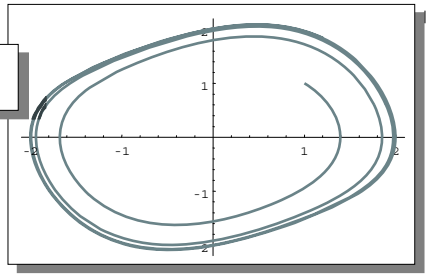

```
simulate(DAEexample, stopTime = 1)
plot(x)
```

 Copyright © Peter Fritzson

Example class: Van der Pol Oscillator

```
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;
```

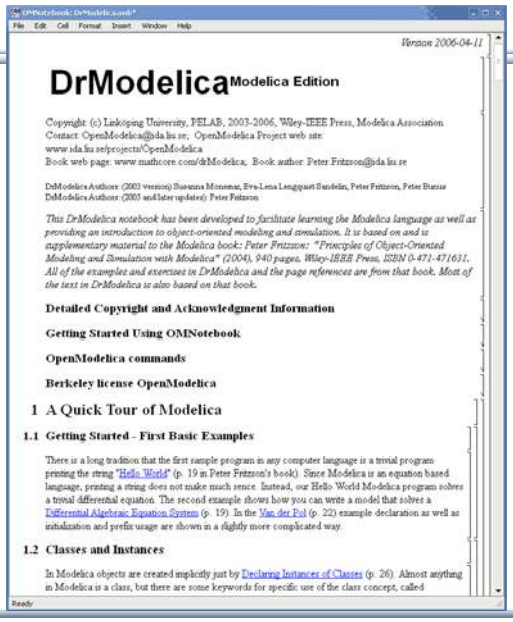
```
simulate(VanDerPol,stopTime = 25)
plotParametric(x,y)
```



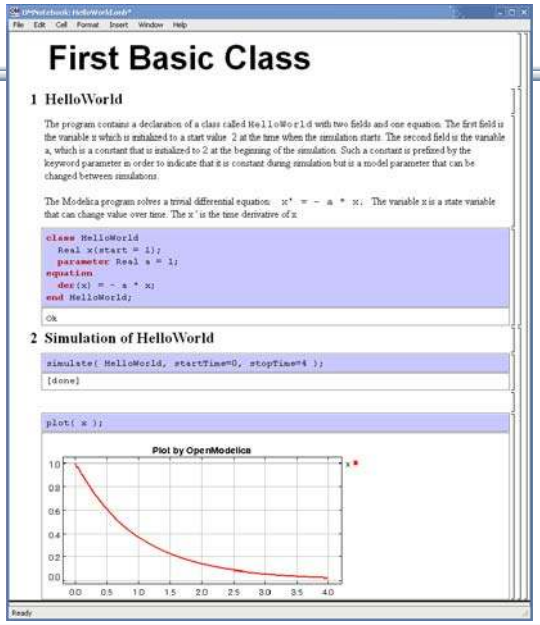
 Copyright © Peter Fritzson

OpenModelica OMNotebook Electronic Notebook with DrModelica

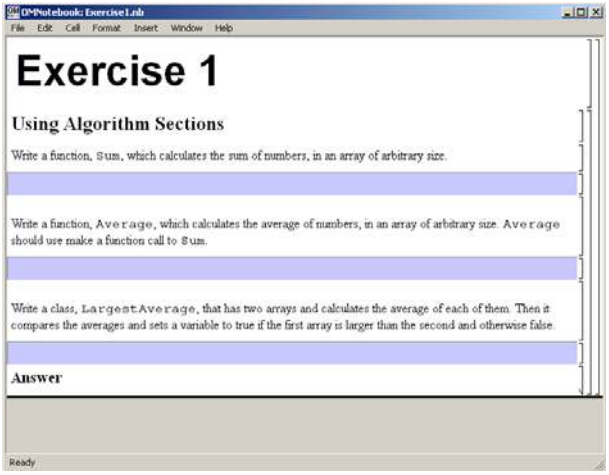
- Primarily for teaching
- Interactive electronic book
- Platform independent



Cells with both Text and Graphics



Exercises and Answers in OMNotebook DrModelica



 Copyright © Peter Fritzson

Some OMNotebook Commands
(see also OpenModelica Users Guide)

- Shift-return (evaluated a cell)
- File Menu (open, close, etc.)
- Text Cursor (vertical), Cell cursor (horisontal)
- Cell types: text cells & executable code cells
- Copy, paste, group cells
- Copy, pase, group text
- Command Completion (shift-tab)

 Copyright © Peter Fritzson

Small Exercises (1.1 and 1.2)

- Locate the VanDerPol model in DrModelica (link from Section 2.1), using OMNotebook!
- **Exercise 1.1**: Simulate and plot VanDerPol. Do a slight change in the model, re-simulate and re-plot.
- Open the Exercises Modelica Tutorial Introduction found in the Tutorial directory.
- **Exercise 1.2**. Simulate and plot the HelloWorld example. Do a slight change in the model, re-simulate and re-plot. Try command-completion, val( ), etc.

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

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

 Copyright © Peter Fritzson

OMNotebook Cell Editing Exercises (optional)

- Select and copy a cell (tree to the right)
- Position the horizontal cell cursor: first click between cells; then click once more on top of the horizontal line
- Paste the cell

- Note: You can find most Users Guide examples in the UsersGuideExamples.onb in the testmodels directory

 Copyright © Peter Fritzson

Modelica Language Continued
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**

 Copyright © Peter Fritzson

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`

```
constant  Real    PI=3.141592653589793;
constant  String  redcolor = "red";
constant  Integer one = 1;
parameter Real    mass = 22.5;
```

 Copyright © Peter Fritzson

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;
```

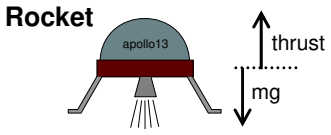
2) Source code comments, disregarded by compiler

2a) C style, e.g. `/* This is a C style comment */`

2b) C++ style, e.g. `// Comment to the end of the line…`

 Copyright © Peter Fritzson

A Simple Rocket Model



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

$$mass' = -massLossRate \cdot \mathrm{abs}(thrust)$$

$$altitude' = velocity$$

$$velocity' = acceleration$$

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

new model

parameters (changeable before the simulation)

floating point type

differentiation with regards to time

declaration comment

start value

name + default value

mathematical equation (acausal)

 Copyright © Peter Fritzson

Celestial Body Class

A class declaration creates a *type name* in Modelica

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

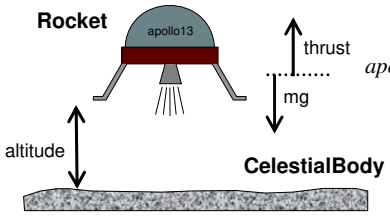
An *instance* of the class can be declared by *prefixing* the type name to a variable name

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

The declaration states that **`moon`** is a variable containing an object of type **`CelestialBody`**

MODELICA pelab

Moon Landing



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

only access inside the class

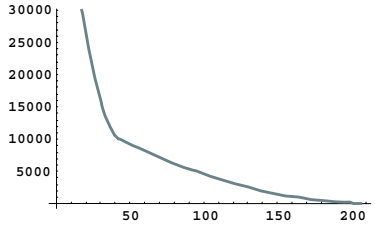
access by dot notation outside the class

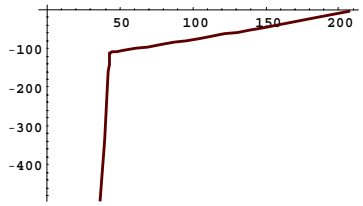
```
class MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
protected
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
public
  Rocket         apollo(name="apollo13");
  CelestialBody   moon(name="moon",mass=7.382e22,radius=1.738e6);
equation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


MODELICA pelab

Simulation of Moon Landing

```
simulate(MoonLanding, stopTime=230)
plot(apollo.altitude, xrange={0,208})
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

 Copyright © Peter Fritzson

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;
```

 Copyright © Peter Fritzson

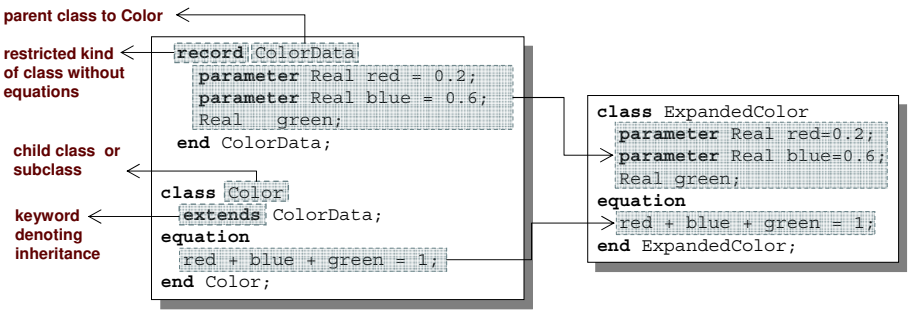
Modelica Functions

- 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

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

 Copyright © Peter Fritzson

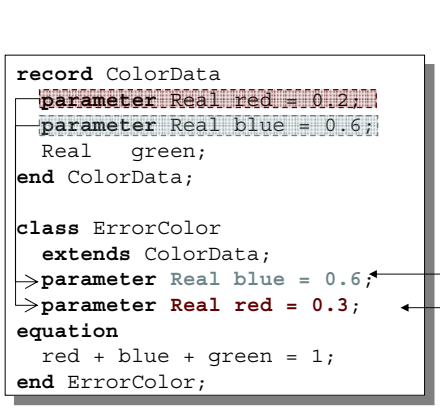
Inheritance



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

 Copyright © Peter Fritzson

Inheriting definitions



Legal!
Identical to the inherited field blue

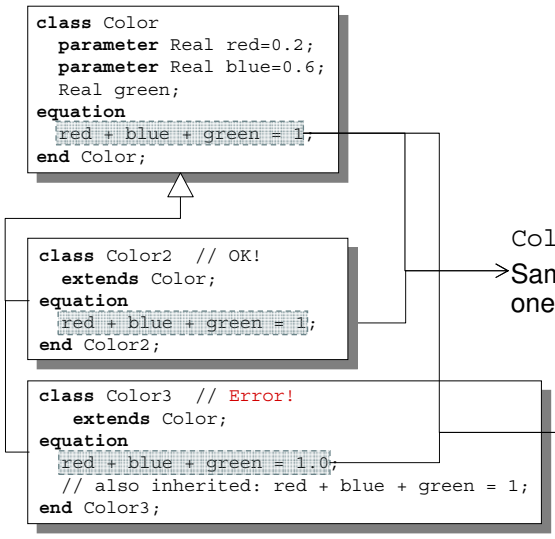
Illegal!
Same name, but different value

Inheriting multiple identical definitions results in only one definition

Inheriting multiple different definitions of the same item is an error

 Copyright © Peter Fritzson

Inheritance of Equations



`Color` is identical to `Color2`

Same equation twice leaves one copy when inheriting

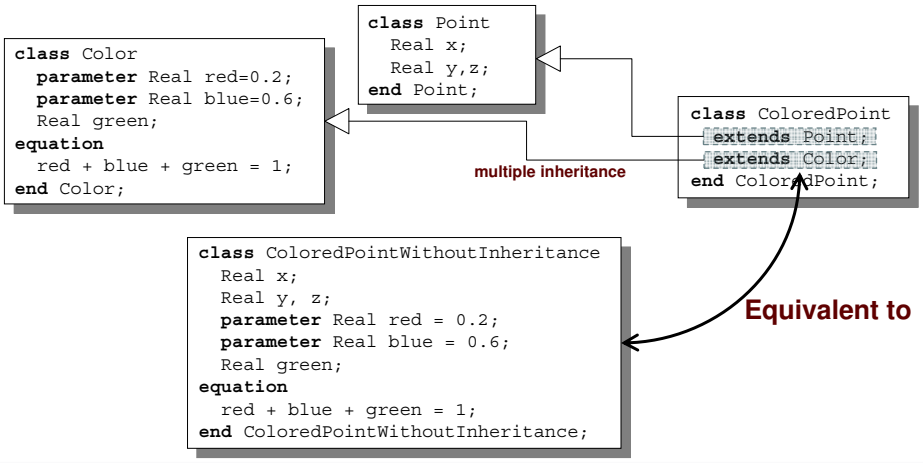
`Color3` is overdetermined

Different equations means two equations!

 Copyright © Peter Fritzson

Multiple Inheritance

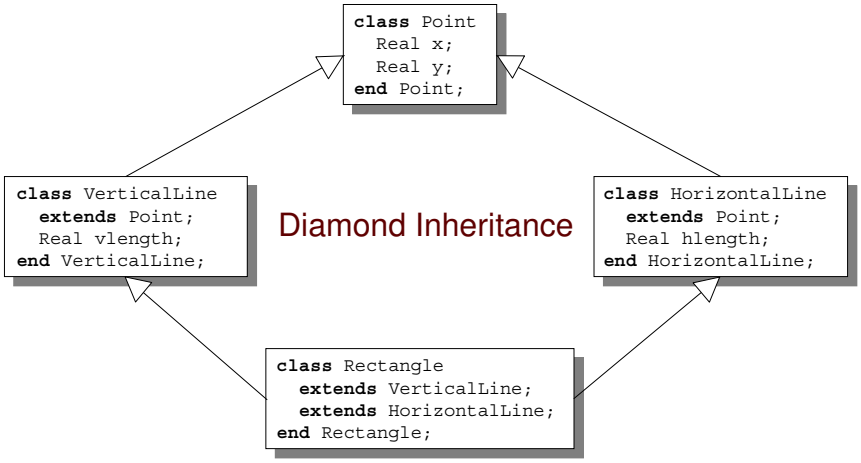
Multiple Inheritance is fine – inheriting both geometry and color



 Copyright © Peter Fritzson

Multiple Inheritance cont'

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



 Copyright © Peter Fritzson

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;
```

 Copyright © Peter Fritzson

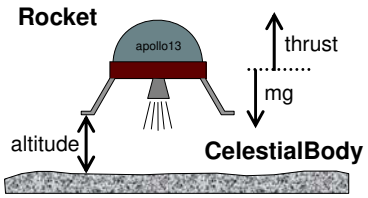
Inheritance Through Modification

- 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:

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

 Copyright © Peter Fritzson

The Moon Landing Example Using Inheritance




```
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= mass*acceleration;
  der(mass)= -massLossRate*abs(thrust);
  der(altitude)= velocity;
  der(velocity)= acceleration;
end Rocket;
```

 Copyright © Peter Fritzson

The Moon Landing Example using Inheritance cont'

inherited parameters

```
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;
```

 Copyright © Peter Fritzson

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

```
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;
```

Equivalent to

```
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!

 Copyright © Peter Fritzson

Do Exercise 1.4 on Model with Simple Systems of Equations

$$\dot{x} = 2 * x * y - 3 * x$$
$$\dot{y} = 5 * y - 7 * x * y$$
$$x(0) = 2$$
$$y(0) = 3$$

 Copyright © Peter Fritzson

Algorithms and Functions

(Only If there is enough time, otherwise skip to page 71, Components, Connectors, Graphical modeling

 Copyright © Peter Fritzson

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;
  ...
```

 Copyright © Peter Fritzson

Iteration Using for-statements in Algorithm Sections

```
for <iteration-variable> in <iteration-set-expression> loop
   <statement1>
   <statement2>
   ...
end for
```

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

```
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;
```

A simple `for`-loop summing the five elements of the vector `z`, within the class `SumZ`

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

```
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
```

 Copyright © Peter Fritzson

Iterations Using while-statements in Algorithm Sections

```
while <conditions> loop
  <statements>
end while;
```

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

```
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;
```

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 eps.

 Copyright © Peter Fritzson

if-statements

```
if <condition> then
  <statements>
elseif <condition> then
  <statements>
else
  <statementss>
end if
```

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
  Real sum;
  parameter Real v[5] = {100,200,-300,400,500};
  parameter Integer n = size(v,1);
algorithm
  sum := 0;
  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;
```


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`.

 Copyright © Peter Fritzson

Positional and Named Argument Association

Using *positional* 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`.

```
...
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.

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

 Copyright © Peter Fritzson

External Functions

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

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

The body of an external function is marked with the keyword **`external`**

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:

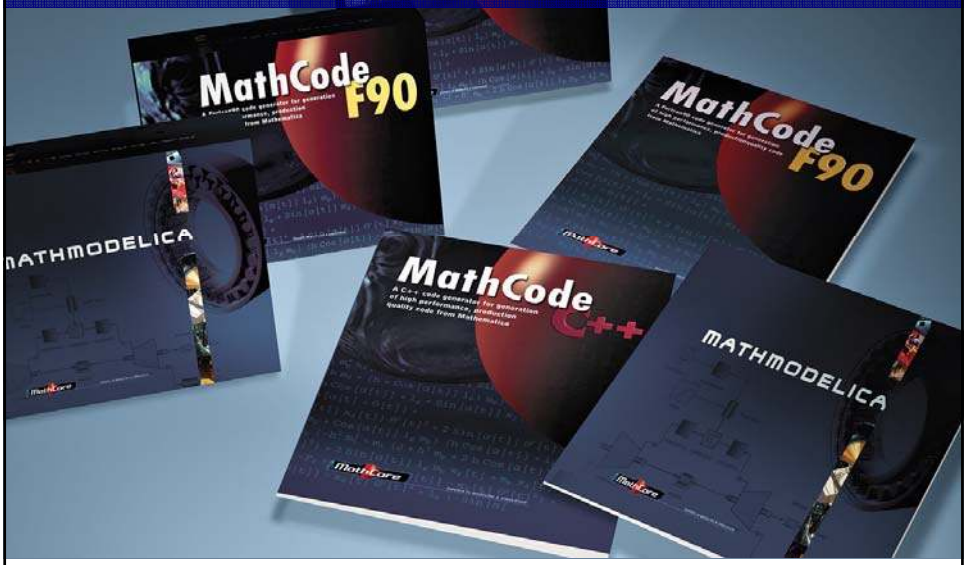
```
function polynomialMultiply
  input  Real a[:], b[:];
  output Real c[:] := zeros(size(a,1)+size(b, 1) - 1);
external "FORTRAN 77"
end polynomialMultiply;
```

 Copyright © Peter Fritzson

If enough time, Do Exercise 1.5 on Functions and algorithms

 Copyright © Peter Fritzson

MathModelica Environment



 Copyright © Peter Fritzson

MathModelica Editions

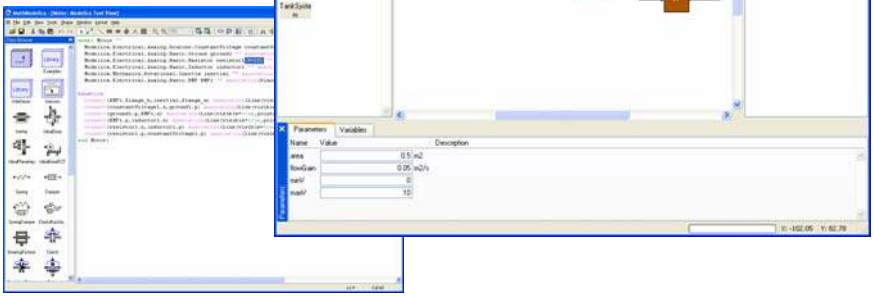
	MathModelica Lite	MathModelica System Designer	MathModelica System Designer Professional
Libraries (kernel)	✓ partial	✓	✓
Model Editor	✓ partial	✓	✓
Simulation Center	⊘	✓	✓
Mathematica connection	⊘	⊘	✓
MathCore Premier Service	⊘	✓	✓

 Copyright © Peter Fritzson

MathModelica® Lite™

Ideal for small student projects. It is free for academic and personal use, but not available for industry.

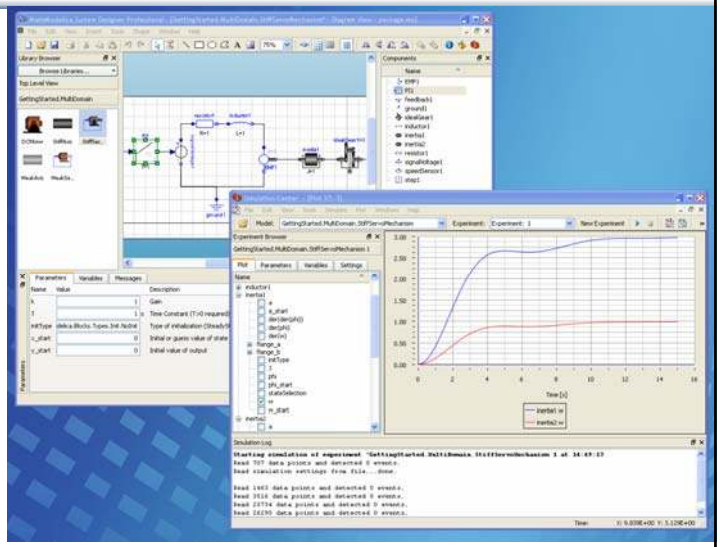
Runs with, but is not part of OpenModelica



 Copyright © Peter Fritzson

MathModelica® System Designer™

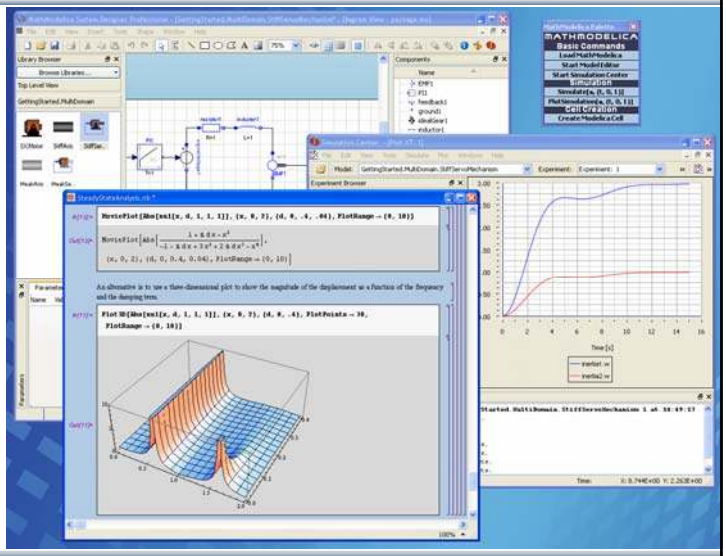
Suited for **modeling and simulation projects** in industry and academia.



 Copyright © Peter Fritzson

MathModelica® System Designer Professional™

Targeted at **research in industry and academia**, offering unparalleled possibilities for analyzing results.



 Copyright © Peter Fritzson

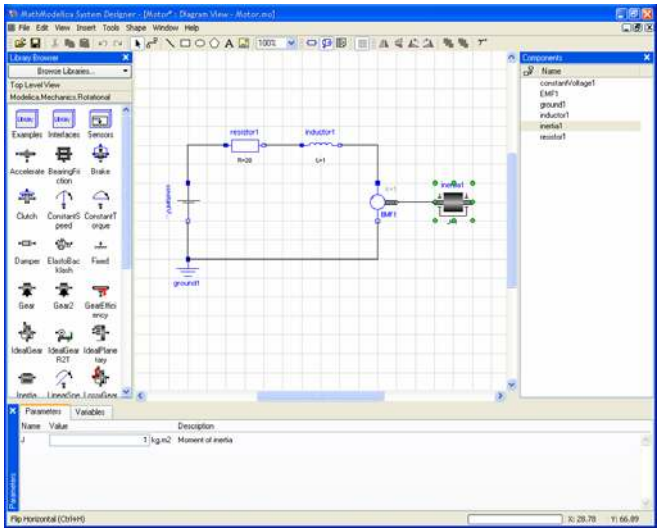
Components, Connectors and Connections – Modelica Libraries and Graphical Modeling

 Copyright © Peter Fritzson

Live example of electric circuit

 Copyright © Peter Fritzson

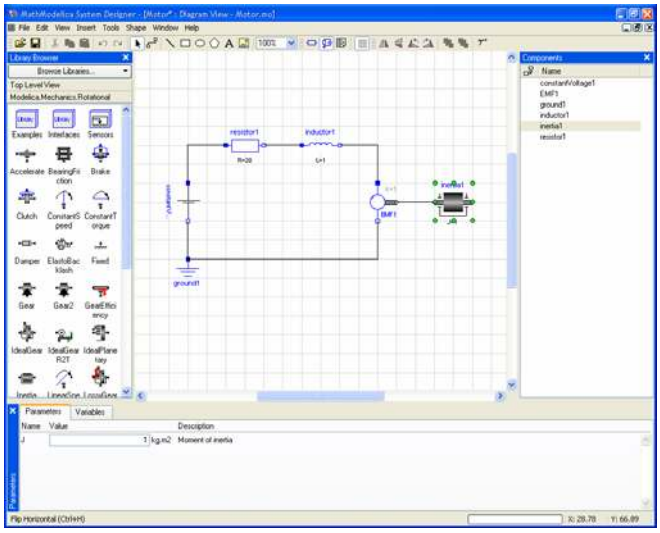
Graphical Modeling
Using Drag and Drop Composition



Courtesy
MathCore
Engineering AB

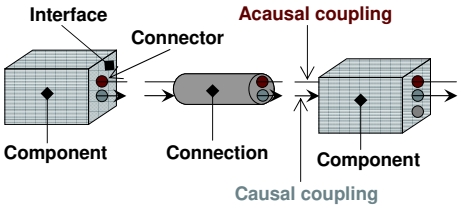
 Copyright © Peter Fritzson

Completed DCMotor using Graphical Composition



Courtesy MathCore Engineering AB

Software Component Model



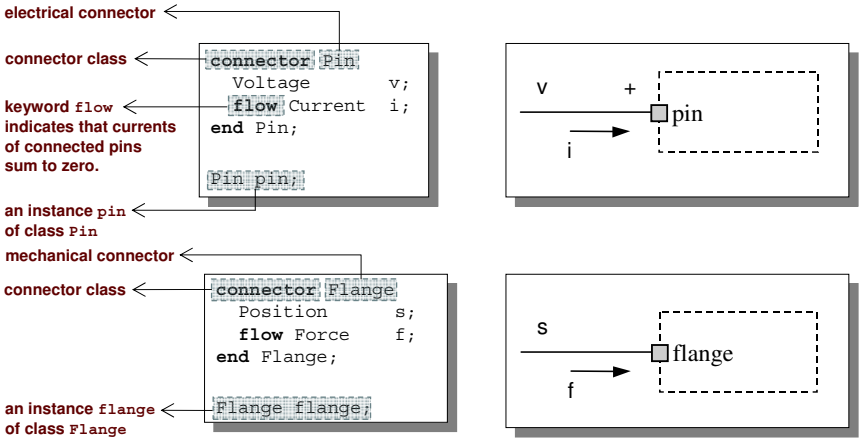
A component class should be defined *independently of the environment,* very essential for *reusability*

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*



 Copyright © Peter Fritzson

The `flow` prefix

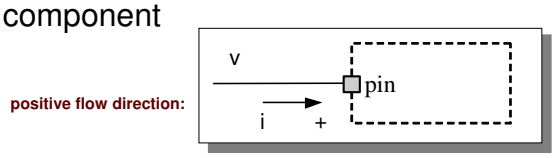
Two kinds of variables in connectors:

- *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



 Copyright © Peter Fritzson

Physical Connector Classes Based on Energy Flow

Domain Type	Potential	Flow	Carrier	Modelica Library
Electrical	Voltage	Current	Charge	`Electrical.Analog`
Translational	Position	Force	Linear momentum	`Mechanical.Translational`
Rotational	Angle	Torque	Angular momentum	`Mechanical.Rotational`
Magnetic	Magnetic potential	Magnetic flux rate	Magnetic flux	
Hydraulic	Pressure	Volume flow	Volume	`HyLibLight`
Heat	Temperature	Heat flow	Heat	`HeatFlow1D`
Chemical	Chemical potential	Particle flow	Particles	Under construction
Pneumatic	Pressure	Mass flow	Air	`PneuLibLight`

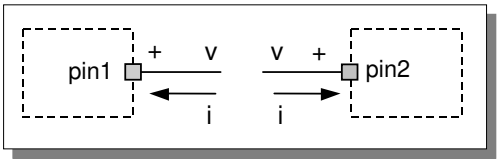
 Copyright © Peter Fritzson

`connect`-equations

Connections 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(pin1,pin2);
```

Corresponds to

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

 Copyright © Peter Fritzson

Connection Equations

```
Pin pin1,pin2;
//A connect equation
//in Modelica
connect(pin1,pin2);
```

Corresponds to

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

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:

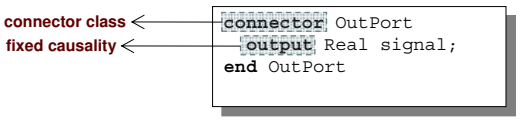
$$i_1 + i_2 + \ldots (-i_k) + \ldots i_n = 0$$

 Copyright © Peter Fritzson

Acausal, Causal, and Composite Connections

Two *basic* and one *composite* kind of connection in Modelica

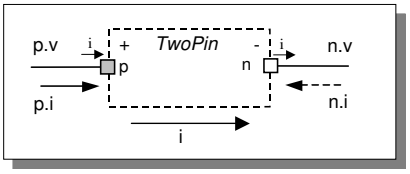
- *Acausal connections*
- *Causal connections*, also called *signal* connections
- *Composite connections*, also called structured connections, composed of basic or composite connections



 Copyright © Peter Fritzson

Common Component Structure

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



partial class (cannot be instantiated)

positive pin

negative pin

```
partial model TwoPin
  Voltage     v
  Current     i
  Pin p;
  Pin n;
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
```

```
connector Pin
  Voltage        v;
  flow Current   i;
end Pin;
```

electrical connector class

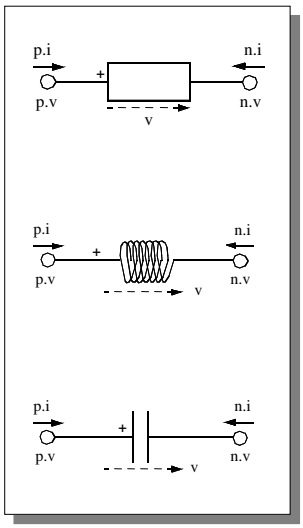
 Copyright © Peter Fritzson

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;
```

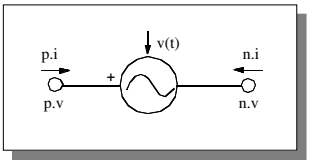
```
model Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real C ;
equation
  i=C*der(v);
end Capacitor;
```



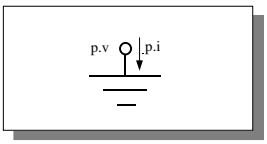
 Copyright © Peter Fritzson

Electrical Components cont'

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

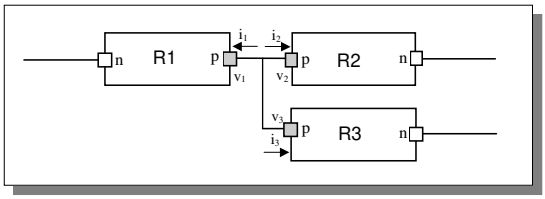



```
model Ground
   Pin p;
equation
   p.v = 0;
end Ground;
```



 Copyright © Peter Fritzson

Resistor Circuit




```
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

```
R1.p.v = R2.p.v;
R1.p.v = R3.p.v;
R1.p.i + R2.p.i + R3.p.i = 0;
```

 Copyright © Peter Fritzson

Modelica Standard Library Used for Graphical Modeling

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.

 Copyright © Peter Fritzson

Modelica Standard Library cont'

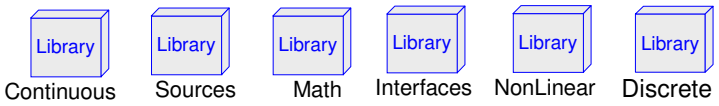
Modelica Standard Library contains components from various application areas, with the following sublibraries:

- 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 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

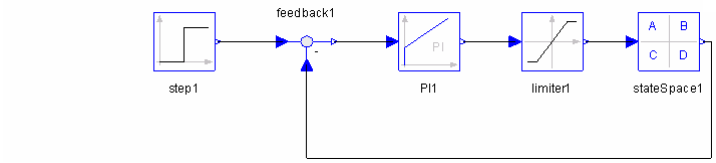
 Copyright © Peter Fritzson

Modelica.Blocks

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



Example:

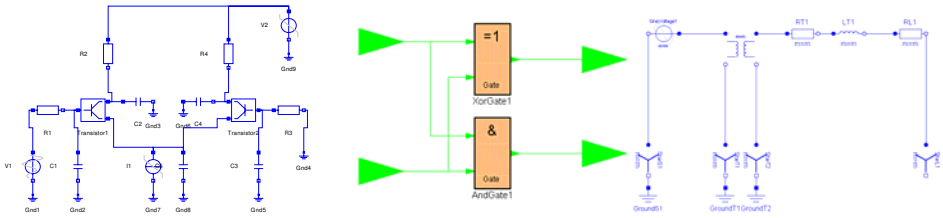


 Copyright © Peter Fritzson

Modelica.Electrical

Electrical components for building analog, digital, and multiphase circuits

Examples:



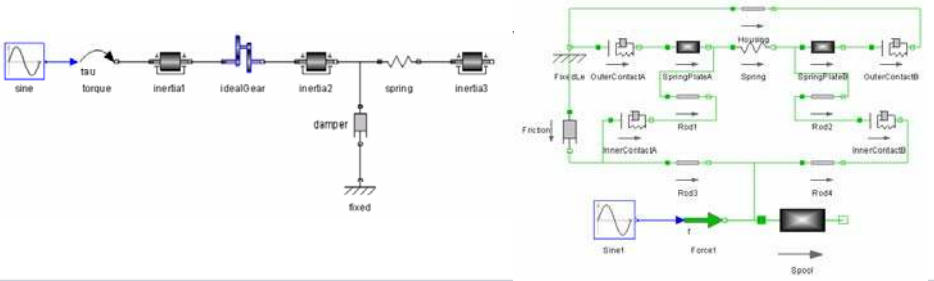
 Copyright © Peter Fritzson

Modelica.Mechanics

Package containing components for mechanical systems

Subpackages:

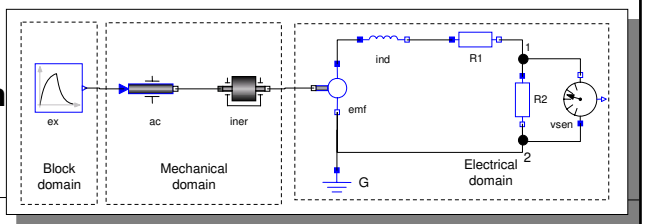
- Rotational — 1-dimensional rotational mechanical components
- Translational — 1-dimensional translational mechanical components
- MultiBody — 3-dimensional mechanical components



 Copyright © Peter Fritzson

Example of Connecting Components from Multiple Domains

- **Block domain**
- **Mechanical domain**
- **Electrical domain**



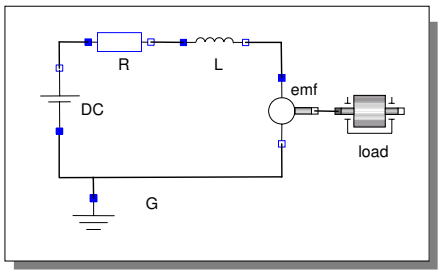

```
model Generator
  Modelica.Mechanics.Rotational.Accelerate ac;
  Modelica.Mechanics.Rotational.Inertia iner;
  Modelica.Electrical.Analog.Basic.EMF emf(k=-1);
  Modelica.Electrical.Analog.Basic.Inductor ind(L=0.1);
  Modelica.Electrical.Analog.Basic.Resistor R1,R2;
  Modelica.Electrical.Analog.Basic.Ground G;
  Modelica.Electrical.Analog.Sensors.VoltageSensor vsens;
  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(emf.n, R2.n); connect(R1.n, R2.p); connect(R2.p, vsens.n);
  connect(R2.n, vsens.p); connect(ex.outPort, ac.inPort);
end Generator;
```

 Copyright © Peter Fritzson

Simple Modelica DCMotor Model Multi-Domain (Electro-Mechanical)

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;
```



 Copyright © Peter Fritzson

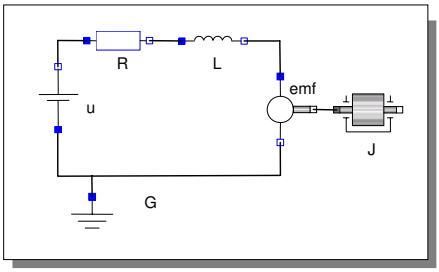
Graphical Modeling Exercises

 Copyright © Peter Fritzson

Exercise 2.1

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

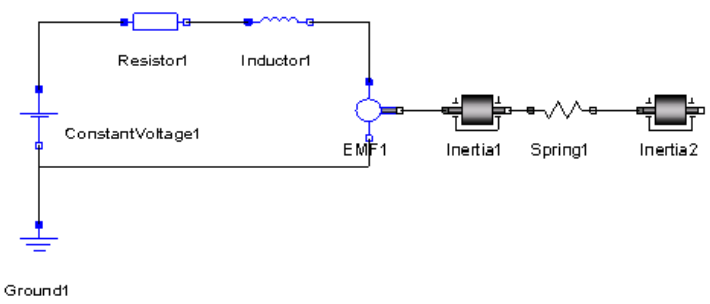
- Simulate it for 15s and plot the variables for the outgoing rotational speed on the inertia axis and the voltage on the voltage source (denoted u in the figure) in the same plot.



 Copyright © Peter Fritzson

Optional Exercise 2.2

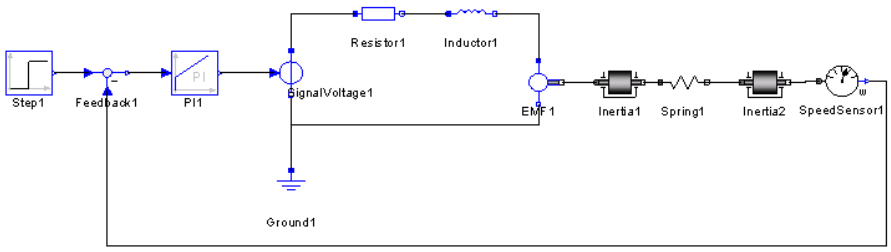
- If there is enough time: Add a torsional spring to the outgoing shaft and another inertia element. Simulate again and see the results. Adjust some parameters to make a rather stiff spring.



 Copyright © Peter Fritzson

Optional Exercise 2.3

- If there is enough time: 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.



 Copyright © Peter Fritzson

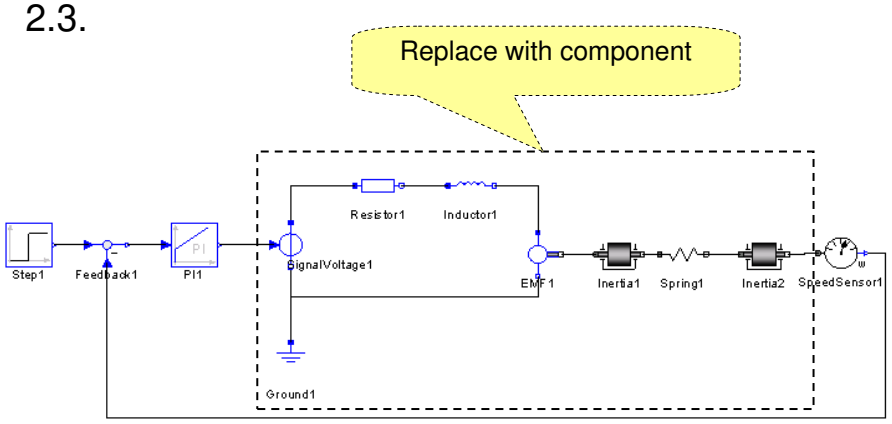
Live example

- Building a component with icon

 Copyright © Peter Fritzson

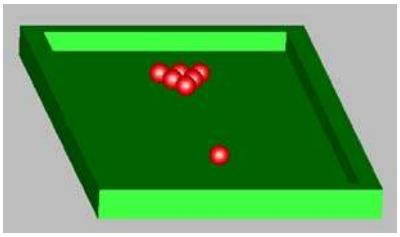
Optional Exercise 2.4

- Make a component of the model in Exercise 2.2, and use it when building the model in exercise 2.3.



 Copyright © Peter Fritzson

Optional Extra Material
Discrete Events and Hybrid Systems

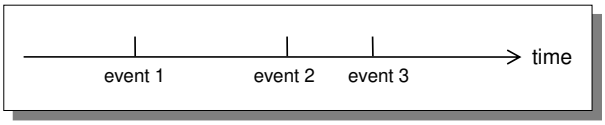


Picture: Courtesy Hilding Elmqvist

 Copyright © Peter Fritzson

Events

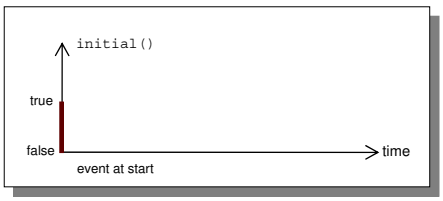
Events are ordered in time and form an event history



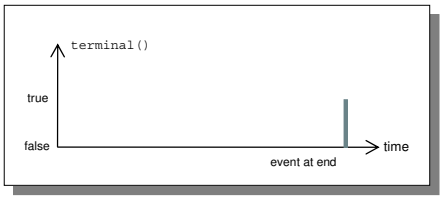
- A *point* in time that is instantaneous, i.e., has zero duration
- An *event condition* that switches from false to true in order for the event to take place
- A set of *variables* that are associated with the event, i.e. are referenced or explicitly changed by equations associated with the event
- 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()`

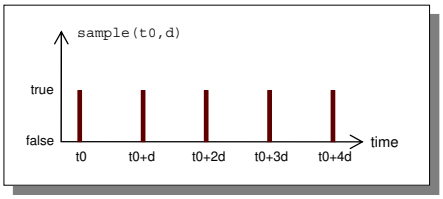


Actions at the end of a simulation are triggered by `terminal()`



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;
```

 Copyright © Peter Fritzson

Expressing Event Behavior in Modelica

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

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

```
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;
```

when-equations become active at events

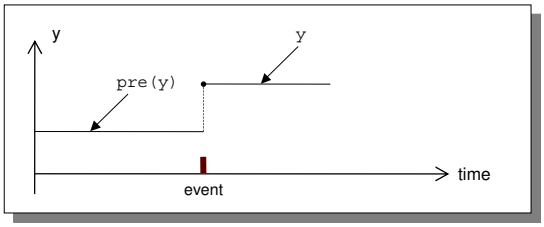
```
when <conditions> then
  <equations>
end when;
```

```
equation
  when x > y.start then
    ...
```

 Copyright © Peter Fritzson

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

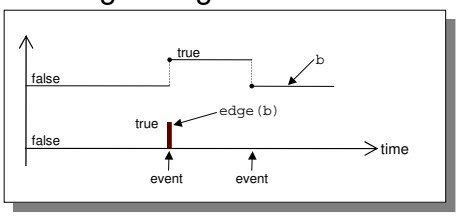


- The variable *y* has one of the basic types `Boolean`, `Integer`, `Real`, `String`, or `enumeration`, a subtype of those, or an array type of one of those basic types or subtypes
- The variable y is a discrete-time variable
- The `pre` operator can *not* be used within a function

 Copyright © Peter Fritzson

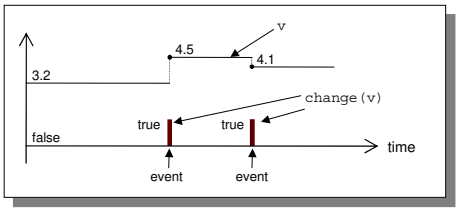
Detecting Changes of Boolean Variables Using `edge()` and `change()`

Detecting changes of boolean variables using `edge()`



The expression `edge(b)` is true at events when b switches from false to true

Detecting changes of discrete-time variables using `change()`

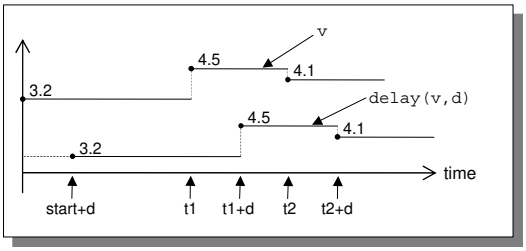


The expression `change(v)` is true at instants when v changes value

 Copyright © Peter Fritzson

Creating Time-Delayed Expressions

Creating time-delayed expressions using `delay()`



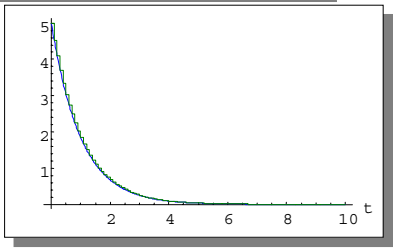
In the expression `delay(v,d)` *v* is delayed by a delay time *d*

 Copyright © Peter Fritzson

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})
```



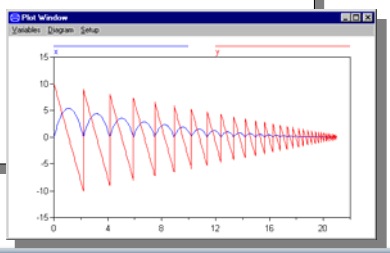
 Copyright © Peter Fritzson

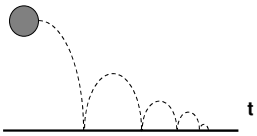
Discontinuous Changes to Variables at Events via When-Equations/Statements

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

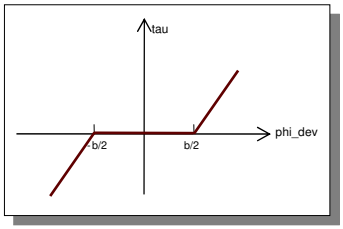
```
model BouncingBall "the bouncing ball model"
  parameter Real g=9.18;   //gravitational acc.
  parameter Real c=0.90;   //elasticity constant
  Real x(start=0),y(start=10);
equation
  der(x) = y;
  der(y)=-g;
  when x<0 then
    reinit(y, -c*y);
  end when;
end BouncingBall;
```



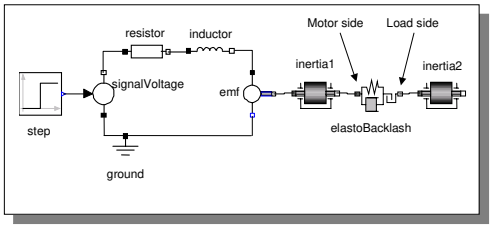


A Mode Switching Model Example

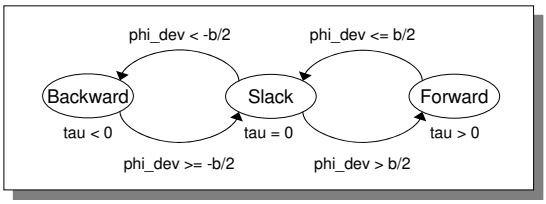
Elastic transmission with slack



DC motor transmission with elastic backlash



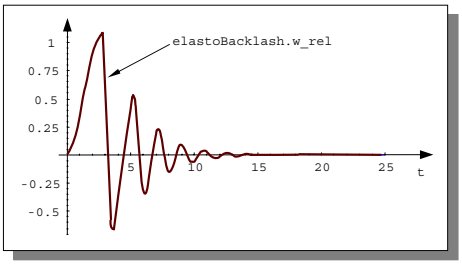
A finite state automaton `SimpleElastoBacklash` model



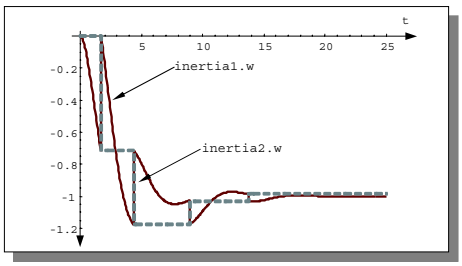
A Mode Switching Model Example cont'

```
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";
  Flange_a        flange_a           "(left) driving flange - connector";
  Flange_b        flange_b           "(right) driven flange - connector";
  parameter Real  phi_rel0 = 0       "Angle when spring exerts no torque";
  Real         phi_rel               "Relative rotation angle betw. flanges";
  Real         phi_dev               "Angle deviation from zero-torque pos";
  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                        // Forward angle gives
          c*(phi_dev - b/2)                    // positive driving torque
        else (if backward then                 // Backward angle gives
                c*(phi_dev + b/2)              // negative braking torque
              else                             // Slack gives
                0);                            // zero torque
end SimpleElastoBacklash
```


A Mode Switching Model Example cont'



Relative rotational speed between the flanges of the `Elastobacklash` transmission

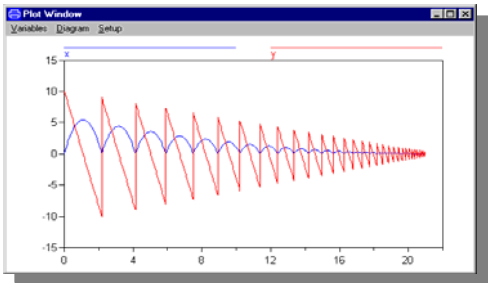


We define a model with less mass in `inertia2(J=1)`, no damping `d=0`, and weaker string constant `c=1e-5`, to show even more dramatic backlash phenomena

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

Optional Exercise 1.5

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



 Copyright © Peter Fritzson