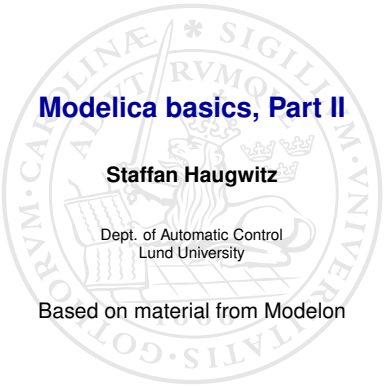


Modelica basics, Part II

Staffan Haugwitz

Dept. of Automatic Control
Lund University

Based on material from Modelon

Modelica Literature

- *Principles of object-oriented modeling and simulation with Modelica 2.1*, Peter Fritzson, 2004 Wiley
- Modelica Tutorial 1.4 available at www.modelica.org
- Right-click on any component from the standard library to get html-documentation.

- Flat Modelica (all code within the same model)
- Hierachical Modelica (modeling with components and object-oriented models)

Modelica Basics

package	A container for holding many models for later use. Structural unit for organizing libraries in a hierarchy.
model	A model of a physical component.
connector	A special class to define how models can interact with other models.
inheritance	A mechanism by which models can be re-used and specialized without having to copy the source code.
extends	The Modelica keyword to implement inheritance.
public	The part of the model that may be accessed and used from the outside
protected	The part of the model that is only useable by the class itself and its descendents.

Model management

- Use a package to to store all models for a certain process
- Menu File -> New Model to define new model
- “Extend from ...” to specialize or adapt a certain model
- “Duplicate class” to make a separate copy of a model for modifications that can not be realized by inheritance.
- “Save total” if you want to save the current model and all underlying code that this model uses by extension or components.

Simple Pendulum, (Flat Modelica)

$$\ddot{\theta} = -g/L \sin\theta$$

```
model SimplePendulum
  constant Real g = 9.81;
  parameter Real L = 1  ''Length of pendulum [m]'';
  Real Theta;
  Real ThetaDot;
equation
  ThetaDot = der(Theta);
  der(ThetaDot) = - g/L*sin(Theta);
end SimplePendulum;
```

Simple Pendulum, SI-units

$$\ddot{\theta} = -g/L \sin\theta$$

```
model SimplePendulum
  constant Modelica.SIunits.Acceleration g = 9.81;
  parameter Modelica.SIunits.Length L = 1;
  Modelica.SIunits.Angle Theta;
  Modelica.SIunits.AngularVelocity ThetaDot;
equation
  ThetaDot = der(Theta);
  der(ThetaDot) = - g/L*sin(Theta);
end SimplePendulum;
```

Simple Pendulum, Import Package SI-units

$$\ddot{\theta} = -g/L \sin\theta$$

```
model SimplePendulum
  package SI = Modelica.SIunits;
  constant SI.Acceleration g = 9.81;
  parameter SI.Length L = 1;
  SI.Angle Theta;
  SI.AngularVelocity ThetaDot;
equation
  ThetaDot = der(Theta);
  der(ThetaDot) = - g/L*sin(Theta);
end SimplePendulum;
```

Simple Pendulum, Attributes

```
model SimplePendulum
  package SI = Modelica.SIunits;
  constant SI.Acceleration g = 9.81;
  parameter SI.Length L (min = 0) = 1;
  SI.Angle Theta (start = 0.1,fixed = true);
  SI.AngularVelocity ThetaDot;
equation
  ThetaDot = der(Theta);
  der(ThetaDot) = - g/L*sin(Theta);
end SimplePendulum;
```

- min,max,start,nominal, unit, quantity

Initial Conditions

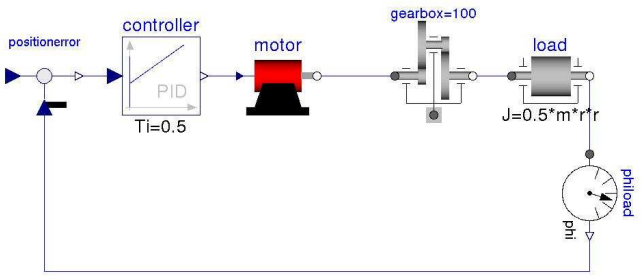
- Setting initial values for states and variables
- Important for highly nonlinear systems
- Staying in domain where the model is valid
- Start from steady-state

Initial conditions - Cartesian Coordinates

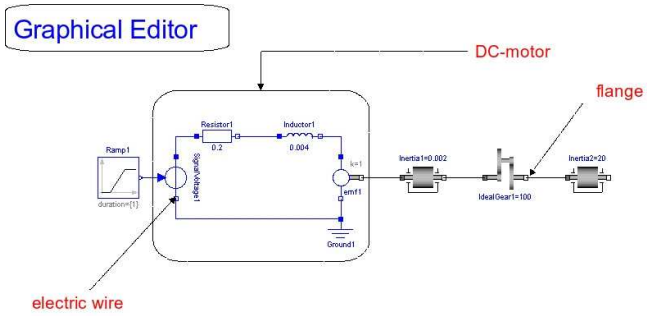
```
model PendulumInitialization
  extends SimplePendulum
  Real x(start=L); // fixed = false is default
  Real y;
initial equation
  y = -0.8*L;
  der(y) = 0;
equation
  x =  L*sin(phi);
  y = -L*cos(phi);
end PendulumInitialization;
```

- parameter SI.Angle Theta (start = 0.1); //approximate OK!
- parameter SI.Angle Theta (start = 0.1, fixed = true); // Too many initial conditions!
- May define system of equations to determine initial values, but they cannot be changed without re-compiling.

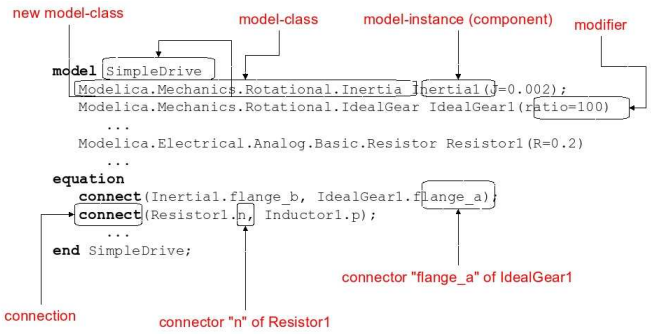
Hierarchical Modelica - Components



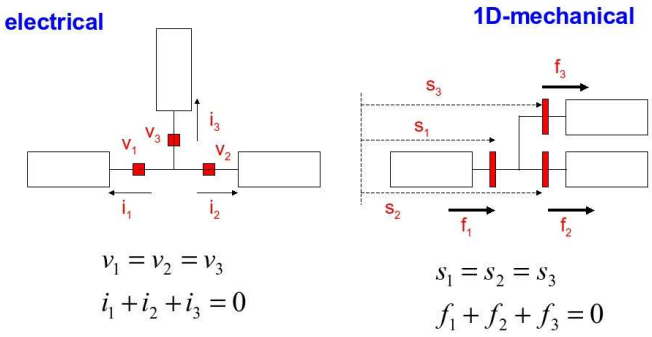
Hierarchical Modelica - Components



Connecting Components - Text Version



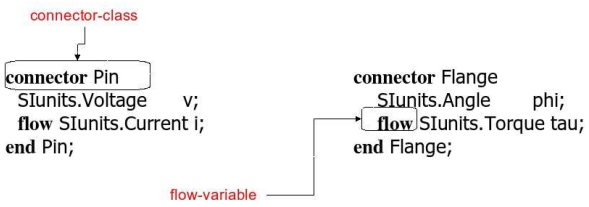
Variables with Connectors



connect(R1.p, R2.p);

Potential and Flow Variables in Connectors

Potential-Variable	Connected variables are identical
Flow-Variable	Connected variables fulfil the zero-sum equation

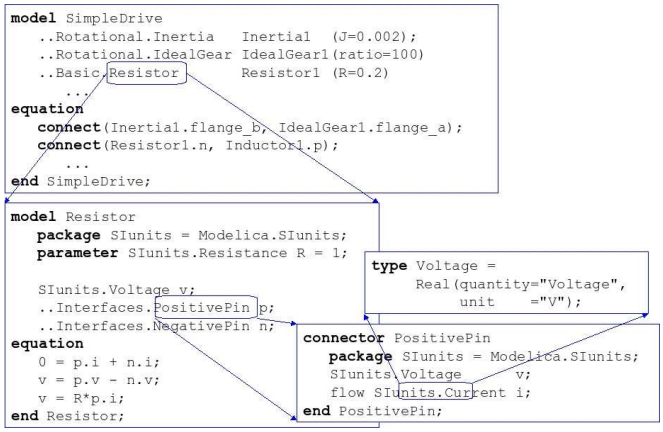


Consequence: flow-variables like mass flows **change sign** from one side of a 2-element connection to the other!

Model of Resistor


```
model Resistor
   package SIunits     = Modelica.SIunits;
   package Interfaces = Modelica.Electrical.Analog.Interfaces;
   parameter SIunits.Resistance R = 1 "Resistance";
   SIunits.Voltage         v "Spannungsabfall über Element";
   Interfaces.PositivePin p;
   Interfaces.NegativePin n;
equation
   0 = p.i + n.i;
   v = p.v - n.v;
   v = R*p.i;
end Resistor;
```

Hierarchical Modelica - Summary



Inheritence

Major benefit of object-oriented programming is the ability to extend the behavior and properties of an existing class.

```
partial model TwoPin
Pin     p, n;
Voltage v;
Current i;
equation
v = p.v - n.v;
0 = p.i + n.i;
i = p.i;
end TwoPin
```

Re-use Code with Inheritence, Resistor

```
model Resistor
extends TwoPin;
parameter Real R (unit = ''Ohm'') ''Resistence'';
equation
R*i = v;
end Resistor
```

```
model Resistor
Pin     p, n;
Voltage v;
Current i;
parameter Real R (unit = ''Ohm'') ''Resistence'';
equation
v = p.v - n.v;
0 = p.i + n.i;
i = p.i;
R*i = v;
end Resistor
```

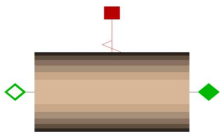
Re-use Code with Inheritence, Capacitor

```
model Capacitor
extends TwoPin;
parameter Real C (unit = ''F'') ''Capacitance'';
equation
C*der(v) = i;
end Capacitor
```

Multiple Inheritence

A pipe model

- Connectors (static/dynamic)
- Geometry
- Flow characteristics
- Medium



Pipe

Pipe Model - Multiple Inheritence

```
model SimplePipeDS
  "Distributed pipe model without heat res, static"
  extends Icons.MultiStatic.PipeAdiab(nspecies=NumberOfSpecies);
  replaceable parameter Data.Pipes.BaseGeometry geo;
  replaceable parameter Data.Pipes.BaseFlowChar char;
  extends PartialComponents.ControlVolumes.Volume2PortDS_pTX(
    nspecies=NumberOfSpecies,
    V=ones(n)*geo.V/n,
    L=geo.L,
    A=geo.A,
    Dhyd=geo.Dhyd,
    alpha=geo.pipeangle,
    redeclare model PressureLoss = PressureDrop.PressureLossD (d
        mdot0=char.mdot0),
    redeclare model Medium = ThisMedium);
equation
  connect(q, heat.q);
end SimplePipeDS;
```

Multiple Inheritence

+ Simplifies re-use of existing code.

- Harder to get an overview, as the code is scattered among many different submodels.

Matrices and Arrays in Modelica 1

Declaration of multidimensional arrays:

```
parameter Real v[3] = {1, 2, 3};
```

{} is array constructor and generates the dimensions. Allows for initialization of arrays with arbitrary dimension. In example:

```
parameter Real m1[2,3] = {{11,12,13}, {21,22,23}};
```

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

```
parameter Real m2[2,3] = [11, 12, 13; 21, 22, 23];
```

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

[...] generates matrices **Matlab** compatible.
In general: [...] generates a Matrix, therefore, [v] is a 3 x 1 Matrix.

```
parameter Real m3[3,3] = [m1; transpose([v])];
```

Matrices and Arrays in Modelica 2

":" in the declaration section is used, when the sizes of the array is undefined

```
parameter Real v[:];  // Size not defined yet
parameter Real A[:,:];
```

Access to matrix elements:

```
M2[2,3]  // element [2,3] of Matrix M2
```

Vector constructor normally used to generate an indices-vector:

```
1:4     // generates {1,2,3,4}
1:2:7   // generates {1,3,5,7}
```

Extraction mechanism of sub-matrices like in Matlab:

```
M2[2:4,3]   // generates {M2[2,3], M2[3,3], M2[4,3]}
```

Matrices and Arrays in Modelica 3

Addition/Subtraction: element-wise

```
  Real A1[3,2,4], A2[3,2,4], A3[3,2,4];
equation
  A1 = A2 + A3;
```

Scalar Multiplication: element-wise

```
  Real A1[3,2,4], A2[3,2,4], A3[3,2,4];
  Real p1, p2;
equation
  A1 = p1*A2 + p2*A3;
```

Matrices and Arrays in Modelica 4

Matrix Multiplication:

```
Real A[3,4], B[4,5], C[3,5]
Real v1[3], v2[4], s1, s2[1,1];
equation
// Vector*Vector = Scalar
  s1 = v1*v1;

// Matrix * Matrix = Matrix
  A  = B*C;
  s2 = transpose([v1])*[v1];
  s1 = scalar(s2);

// Matrix*Vector = Vector
  v1 = A*v2;
```

```
s := 0;
for i in 1:size(v1,1) loop
  s := s + v1[i]*v1[i];
end for;
```

```
for i in 1:size(A,1) loop
  v1[i] := 0;
  for j in 1:size(A,2) loop
    v1[i] := v1[i] + A[i,j]*v2[j];
  end for;
end for;
```

Matrices and Arrays in Modelica 5

Modelica	*Erklärung*
ndims(A)	Returns the number of dimensions k of array expression A, with k >= 0.
size(A,i)	Returns the size of dimension i of array expression A where i shall be > 0 and <= ndims(A).
size(A)	Returns a vector of length ndims(A) containing the dimension sizes of A.
scalar(A)	Returns the single element of array A. size(A,i) = 1 is required for 1 <= i <= ndims(A).
vector(A)	Returns a 1-vector, if A is a scalar and otherwise returns a vector containing all the elements of the array, provided there is at most one dimension size > 1.
matrix(A)	Returns promote(A,2), if A is a scalar or vector and otherwise returns the elements of the first two dimensions as a matrix. size(A,i) = 1 is required for 2 < i <= ndims(A).
transpose(A)	Permutes the first two dimensions of array A. It is an error, if array A does not have at least 2 dimensions.
outerproduct(v1,v2)	Returns the outer product of vectors v1 and v2 (= matrix(v)*transpose(matrix(v))).
identity(n)	Returns the n x n Integer identity matrix, with ones on the diagonal and zeros at the other places.
diagonal(v)	Returns a square matrix with the elements of vector v on the diagonal and all other elements zero.
zeros(n_1,n_2,n_3,...)	Returns the n_1 x n_2 x n_3 x ... Integer array with all elements equal to zero (n_i >= 0).
ones(n_1,n_2,n_3,...)	Return the n_1 x n_2 x n_3 x ... Integer array with all elements equal to one (n_i >=0).
fill(s,n_1,n_2,n_3, ...)	Returns the n_1 x n_2 x n_3 x ... array with all elements equal to scalar expression s which has to be a subtype of Real, Integer, Boolean or String (n_i >= 0). The returned array has the same type as s.

Matrices and Arrays in Modelica 6

Modelica	*Erklärung*
linspace(x1,x2,n)	Returns a Real vector with n equally spaced elements, such that v=linspace(x1,x2,n), v[i] = x1 + (x2-x1)*(i-1)/(n-1) for 1 <= i <= n. It is required that n >= 2.
min(A)	Returns the smallest element of array expression A.
max(A)	Returns the largest element of array expression A.
sum(A)	Returns the sum of all the elements of array expression A.
product(A)	Returns the product of all the elements of array expression A.
symmetric(A)	Returns a matrix where the diagonal elements and the elements above the diagonal are identical to the corresponding elements of matrix A and where the elements below the diagonal are set equal to the elements above the diagonal of A, i.e., B := symmetric(A) -> B[i,j] := A[i,j], if i <= j, B[i,j] := A[j,i], if i > j.
cross(x,y)	Returns the cross product of the 3-dim-vectors x and y, i.e. cross(x,y) = vector([x[2]*y[3]-x[3]*y[2]; x[3]*y[1]-x[1]*y[3]; x[1]*y[2]-x[2]*y[1]]);
skew(x)	Returns the 3 x 3 skew symmetric matrix associated with a 3-dim-vector, i.e., cross(x,y) = skew(x)*y; skew(x) = [0, -x[3], x[2]; x[3], 0, -x[1]; -x[2], x[1], 0];

Matrices and Arrays in Modelica 7

```
block PolynomialEvaluator
   parameter Real a[:];
   input     Real x;
   output    Real y;
protected
   parameter n = size(a, 1)-1;
   Real xpowers[n+1];
equation
   xpowers[1] = 1;
   for i in 1:n loop
      xpowers[i+1] = xpowers[i]*x;
   end for;
   y = a * xpowers;
end PolynomialEvaluator;
```

Matrices and Arrays in Modelica 8

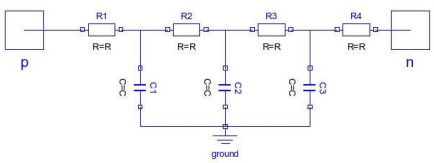
Arrays cannot only consist of Real variables, but of any model class. In example:

```
Modelica.Electrical.Analog.Basic.Resistor R[10] // 10 resistors

for i in 1:9 loop
   connect(R[i].p, R[i+1].n);       // connecting the resistors
end for;
```

This can be utilized to discretize simple partial differential equations in a modular way.

Example: electrical line with losses



Matrices and Arrays in Modelica 9

```
model ULine "Lossy RC Line"
  Modelica.Electrical.Analog.Interfaces.Pin p, n;
  parameter Integer N(final min=1) = 1 "Number of lumped segments";
  parameter Real r = 1 "Resistance per meter";
  parameter Real c = 1 "Capacitance per meter";
  parameter Real L = 1 "Length of line";
protected
  ..Electrical.Analog.Basic.Resistor  R[N + 1](R=r*length/(N + 1));
  ..Electrical.Analog.Basic.Capacitor C[N]    (C=c*length/(N + 1));
  ..Electrical.Analog.Basic.Ground    g;
equation
  connect(p, R[1].p);
  for i in 1:N loop
    connect(R[i].n, R[i + 1].p);
    connect(R[i].n, C[i].p);
    connect(C[i].n, g);
  end for;
  connect(R[N + 1].n, n);
end ULine
```