Modelica Tutorial for Beginners

Multi-domain Modeling and Simulation

Hubertus Tummescheit† and Bernhard Bachmann‡
† United Technologies Research Center
‡ University of Applied Sciences Bielefeld

Outline

- Introduction
  - Industrial Application Examples
  - Composition Diagram versus Block Diagram
  - The Modelica Association

- Modeling with Modelica
  - Flat and Hierarchical Models
  - Special Model Classes
  - Matrices, Arrays and Arrays of Components
  - Physical Fields
  - Hybrid Modeling
**Space-Robotic**

D2 Mission 1995 (Robot in Space Shuttle is controlled from Earth, 7s for signal transmission)

**Industrial Robots**

New drive trains, Service-Robots, cooperation with KUKA

**Control Design**

for fly-by-wire, automatic landing, etc.; based on optimizing of parameter

**Automobile**

Modeling, simulation of mechatronical components

---

**Test of Protection Devices in Power Systems**

[Diagram of power system components and connections]
Modelica Design Effort

Modeling und simulation of multi-domain physical systems

- to beat the modeling complexity

Example: Dynamics of an air plane

- Mechanics (3D-Mechanics, Drive Trains)
- Aerodynamics
- Thermo-fluid dynamics (Turbine engine)
- Hydraulics
- Electrics
- Control systems
- Discrete Control
Example: Vehicle Dynamics using MBS-library

BMW 3-series chassis driving over icy patch on the road. Off-center additional weight on the roof.

154 States
5245 non-trivial variables
Largest linear system 478, reduced to 30 by tearing

The property to “figure out” how to use a component optimally in different environments is a condition for re-usable, object-oriented model libraries, like the VehicleDynamics library:

- Symbolic capabilities condition for scalability to complex models
- Based on a few, very general component models to build complex sub-systems, e.g. McPherson suspension.

Component diagrams generalize Block diagrams
=> The next generation of simulation tools
Modeling Knowledge

Where to find?

- Books
- Experts

Main Idea:

Computer based storage of modeling knowledge

Component Diagrams

- Each icon represents a physical component.
i.e.: electrical Resistance, mechanical Gearbox, Pump

- Composition lines are the actual physical connections.
i.e.: electrical line, mechanical connection, heat flow between two components

- Variables at the interfaces describe interaction with other components

- Physical behavior of a component is described by equations

- Hierarchical decomposition of components
Example: Industrial Robot

Modeling and Simulation with Modelica tools

Modelica model (Text file: test1.mo)

model test1
    Modelica.Mechanics.Rotational.Inertia J1(J=0.002)
    ...

Graphical Editor

C-function

void disblock(double *x,
...

compile + link

Simulator
Modelica Language Design Goals

- Unify object-oriented modeling languages
  - Dymola, gPROMS, NMF, ObjectMath, Omola, Smile, U.L.M., ...
- Allow reuse of physical models
  - Electrical motor, robot arm, ...
- Combine components of different engineering disciplines
  - Electrics, mechanics, thermo-dynamics, hydraulics, ...
- Description using differential- und algebraic equations
  - Declarative instead of procedural
- Achieve efficient simulation code
  - Event handling, ideal devices, etc.
- Develop component libraries

Modelica Association

- Chairman: Martin Otter, DLR, Munich, Germany
- Vice-Chairman: Peter Fritzson, Linköping University, Sweden
- Secretary: Hilding Elmqvist, Dynasim AB, Lund, Sweden (former Chairman)
- Treasurer: Michael Tiller, Ford Motor Company, Dearborn, U.S.A.

- Peter Aronsson, MathCore, Linköping, Sweden
- Bernhard Bachmann, University of Applied Sciences, Germany
- Peter Beater, Universität Paderborn, Germany
- Dag Brück, Dynasim AB, Lund, Sweden
- Peter Bunus, Linköping University, Sweden
- Vadim Engelson, Linköping University, Sweden
- Thilo Ernst, GMD-FIRST, Berlin, Germany
- Jorge Ferreira, Universidade de Aveiro, Portugal
- Rüdiger Franke, ABB Corporate Research Ltd, Heidelberg, Germany
- Pavel Grosman, BrisData AB, Stockholm, Sweden
- Johan Gunnarsson, MathCore, Linköping, Sweden
- Mats Jirstrand, MathCore, Linköping, Sweden
- Kaj Justlin, VTT, Finland
- Clemens Klein-Robbenhaard, GMD Köln, Germany
- Sven Erik Mattsson, Dynasim AB, Lund, Sweden
- Henrik Nilsson, Linköping University, Sweden
- Hans Olsson, Dynasim AB, Lund, Sweden
- Tommy Persson, Linköping University, Sweden
- Per Sahlén, BrisData AB, Stockholm, Sweden
- Levon Saldamli, Linköping University, Sweden
- Andre Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
- Peter Schwarz, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
- Hubertus Tummescheit, Lund University, Sweden
- Hans-Jürg Wiesmann, ABB Corporate Research Ltd, Baden, Switzerland
Status of Modelica

- Design started September 1996
- Modelica Version 1.0 - September 1997
- Modelica Version 1.2 - June 1999
- Modelica Version 1.3 - December 1999
- Modelica Version 1.4 - December 2000
- Modelica Version 2.0 - March 2002
- Modelica Version 2.1 - October 2003
  - > 35 Modelica-Design-Group Meetings (each 3 days)
  - > 25 members of the “Modelica Association”
  - > 200 members of the “Modelica Interest group”
- Libraries and tools are available

Modeling with Modelica

- Flat und hierarchical models
  - data types, attributes, components, interface-variables, package-concept, inheritance
- Special model classes
  - constraints on variables and parameters
    - input, output, final, protected
  - equations versus Algorithms
  - class types in Modelica
    - type, connector, model, block, function, package
- Matrices, arrays and arrays of components
  - definition, index sets, for-in-loop, array-functions
- Physical fields
  - global variables, inner, outer
- Hybrid modeling
  - events, if-then-else, when-statement
Simple Modelica-Model (Flat)

Version 1:

\[ m \ddot{s} = f \]

- graphical information
- parameters (changeable before start of simulation)
- new model
- floating point number
- name + default-value
- comment (display in dialogue)
- mathematical equation, (non-causal)
- differentiation with regards to time

```
model MovingMass1
  parameter Real m = 2
  parameter Real f = 6
  Real s
  Real v
annotation(Diagram(Rectangle(extent=...)))

equation
  v = der(s);
  m*der(v) = f;
end MovingMass1;
```

Pre-Defined Basic-Data Types in Modelica

- **Real** floating point variable, i.e. 1.0, -2.3e-5
- **Integer** integer variable, i.e. 1, 4, -333
- **Boolean** boolean variable, i.e. false, true
- **String** string, i.e. "from file:"
Cause of Failure of the Mars Climate Orbiter on September 23, 1999

Mars Climate Orbiter Failure Board Release Report, Nov. 10, 1999:
... "The root cause of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software, as NASA has previously announced," said Arthur Stephenson, chairman of the Mars Climate Orbiter Mission Failure Investigation Board.

Simple Modelica-Model (Flat)

Version 2:

\[ m \cdot \dot{s} = f \]

\begin{verbatim}
model MovingMass2
    parameter Real m(min=0,unit="kg") = 2;
    parameter Real f(unit="N") = 6;
    Real s;
    Real v;
    annotation(Diagram(Rectangle(extent=..))..);

equation
    v = der(s);
    m*der(v) = f;
end MovingMass2;
\end{verbatim}
**SI-Base Units**

Each physical unit can be calculated based on the 7 SI-base units:

\[ \text{kg, m, s, A, K, mol, cd} \]

Comparison in equations:

Two physical variables are comparable, if the units with regards to the 7 SI-base units are identical.

Example:

<table>
<thead>
<tr>
<th>Type</th>
<th>Unit</th>
<th>in SI-base units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moment</td>
<td>Nm</td>
<td>kgm²/s²</td>
</tr>
<tr>
<td>Energy</td>
<td>J</td>
<td>kgm²/s²</td>
</tr>
</tbody>
</table>

**Realization of Units in Modelica**

attributes of Real variables:

- **quantity**: type of physical quantity
- **unit**: unit of variable, used within equations
- **displayUnit**: unit, used for visualization

Example:

```
quantity = "Torque"  unit = "N.m"  displayUnit = "deg"
quantity = "Energy"  unit = "J"     displayUnit = "deg"
```

Syntax of unit-expressions, Examples:

- kg.m²/s², kg.m/(s.s), rad/s, 1/s, s⁻¹
More Attributes of Real Variables:

- \( \text{min} \) minimal value of quantity
- \( \text{max} \) maximal value of quantity
- \( \text{start} \) start value of state variables. i.e.: \( m \cdot \dot{v} = f, \ v(t_0) = 3 \)
- \( \text{nominal} \) nominal value can be used for scaling purposes in numerical routines

Example:

```modelica
parameter Real m(min=0, quantity="mass", unit="kg") = 2;
Real v(quantity="velocity", unit="m/s", start=3);
```

Pre-defined variable types:

e.g. variables with a given set of attributes

Pre-Defined Variable Types

Examples of different variable types:

```modelica
type Angle = Real(quantity = "Angle", unit = "rad", displayUnit = "deg");
type Torque = Real(quantity = "Torque", unit = "N.m");
type Mass = Real(quantity = "Mass", unit = "kg", min=0);
type Velocity = Real(quantity = "Velocity", unit = "m/s");
```

Use of variable types:

```modelica
parameter Mass m = 2;
Velocity v(start=3);
```
The Modelica Library Modelica.Slunits includes all 450 ISO-standard units in form of pre-defined variable types!

Use of the Modelica.Slunits Library

Variant 1 (full name):

```modelica
parameter Modelica.SIunits.Mass m = 2;
Modelica.SIunits.Velocity v(start=3);
```

Variant 2 (short name):

```modelica
package SI = Modelica.SIunits; // Alias-Name
parameter SI.Mass m = 2;
SI.Velocity v(start=3);
```
Simple Modelica-Model (Flat)

Version 3:

\[ m \cdot \ddot{s} = f \]

```
model MovingMass3

package SIunits = Modelica.SIunits;

parameter SIunits.Mass m = 2 "mass of block";
parameter SIunits.Force f = 6 "force to pull block";
SIunits.Position s "position of block";
SIunits.Velocity v "velocity of block";
annotation(Diagram(Rectangle{extent=..}..));
equation
  v = der(s);
  m*der(v) = f;
end MovingMass3;
```

This is the preferred style of modeling!

Hierarchical Modelica Model
(Model of a simple drive train)

Graphical Editor

DC-motor

flange

electric wire

motor-inertia + ideal gear box + load-inertia
Part of drive train model (without graphics-information)

new model-class
model SimpleDrive
Modelica.Mechanics.Rotational.Inertial[Inertial](J=0.002);
Modelica.Mechanics.Rotational.IdealGear IdealGear1[ratio=100];
...  
Modelica.Electrical.Analog.Basic.Resistor Resistor1(R=0.2)
equation
  connect(Inertial.flange_b, IdealGear1.flange_a);
  connect(Resistor1.n, Inductor1.p);
...
end SimpleDrive;

Variables within interfaces (connector variables)
electrical 1D-mechanical

\[ v_1 = v_2 = v_3 \]
\[ i_1 + i_2 + i_3 = 0 \]
\[ s_1 = s_2 = s_3 \]
\[ f_1 + f_2 + f_3 = 0 \]

connect (R1.p, R2.p);  
connect (ml.flange_a, m2.flange_a);
### Two kinds of variables in connectors

<table>
<thead>
<tr>
<th>Potential-Variable</th>
<th>Flow-Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected variables are identical</td>
<td>Connected variables fulfil the zero-sum equation</td>
</tr>
</tbody>
</table>

```modelica
connector Pin
  SIunits.Voltage v;
  SIunits.Angle phi;
  flow SIunits.Current i;
end Pin;

connector Flange
  flow SIunits.Torque tau;
end Flange;
```

### Interface variables within the Modelica standard library

<table>
<thead>
<tr>
<th>Type</th>
<th>Potential variable</th>
<th>Flow variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>electric</td>
<td>$V$</td>
<td>$i$ current</td>
</tr>
<tr>
<td>translatorial</td>
<td>$s$</td>
<td>$f$ force</td>
</tr>
<tr>
<td>rotatorial</td>
<td>$\phi$</td>
<td>$\tau$ torque</td>
</tr>
<tr>
<td>hydraulic</td>
<td>$p$ pressure</td>
<td>$\dot{V}$ flow rate</td>
</tr>
<tr>
<td>thermal</td>
<td>$T$ temperature</td>
<td>$\dot{Q}$ heat flow</td>
</tr>
<tr>
<td>chemical</td>
<td>$\mu$ chem. potential</td>
<td>$N$ Current of particles</td>
</tr>
</tbody>
</table>

```modelica
connector XXX
  Real PotentialVariable
  flow Real FlowVariable
end XXX;
```
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;
```

Summary

```model SimpleDrive
  .Rotational.Inertia Inertial (J=0.002);
  .Rotational.IdealGear IdealGear1(ratio=100)
  .BasicResistor Resistor1 (R=0.2)
  ...
  equation
    connect(Inertial.flange_b, IdealGear1.flange_a);
    connect(Resistor1.n, Inductor1.p);
    ...
end SimpleDrive;
```

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

```type Voltage =
  Real(quantity="Voltage",
        unit ="V");
```

```connector PositivePin
package SIunits = Modelica.SIunits;
SIunits.Voltage v;
flow SIunits.Current i;
end PositivePin;
```
The package concept in Modelica

Modelica models are structured in hierarchical libraries (packages)

```modelica
package Modelica
  package Mechanics
    package Rotational
      model Inertia
        ...
      end Inertia;
      model Torque
        ...
      end Torque;
      ...
    end Rotational;
  end Mechanics;
...
end Modelica;
```

The package concept in Modelica

Name-lookup within a package

```
encapsulated package Modelica
  package Mechanics
    package Rotational
      package Interfaces
        connector Flange_a
        ...
      end Interfaces
      model Inertia
        Interfaces.Flange_a flange_a;
      end Inertia;
      ...
    end Rotational;
  end Mechanics;
...
end Modelica;
```

Name lookup stops at "encapsulated"

```
equivalent definitions
```
The package concept in Modelica

Storage of a hierarchical package within one file

File: Modelica.mo

```modelica
package Modelica
package Mechanics
package Rotational
    model Inertia
    ...
    end Inertia;

    model Torque
    ...
    end Torque;
    ...
    end Rotational;
end Mechanics;
...
end Modelica;
```

---

The package concept in Modelica

Storage of a hierarchical package distributed within different files and directories

File: ...\Modelica\Mechanics\Rotational.mo

```modelica
...\Modelica\Blocks
...\Modelica\Electrical
...\Modelica\Mechanics
    package.mo
    Rotational.mo
    Translational.mo
```

File: ...\Modelica\Mechanics\package.mo

```modelica
package Rotational
    model Inertia
    ...
    end Inertia;

    model Torque
    ...
    end Torque;
    ...
    end Rotational;
end Mechanics;
```

Modelica is case-sensitive!

Each package-directory must include a file package.mo that contains additional information to the package (i.e. annotations to \Mechanics)
Partial Models and Inheritance

package Electrical
package SIunits = Modelica.SIunits;

= abbreviation
(see: type Force = Real (unit="N")

connector Pin
SIunits.Voltage  v;
flow SIunits.Current i;
end PositivePin;

partial model TwoPin
Pin p,n;
SIunits.Current i;
SIunits.Voltage u;

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

model Capacitor

extends TwoPin;

parameter SIunits.Capacitance C;

equation
C*der(u) = i;
end Capacitor;

end Electrical;

Previous Model is identical to:

package Electrical
package SIunits = Modelica.SIunits;

connector Pin
SIunits.Voltage  v;
flow SIunits.Current i;
end PositivePin;

model Capacitor
Pin p,n;
SIunits.Current i;
SIunits.Voltage u;

parameter SIunits.Capacitance C;

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

end Electrical;

partial model TwoPin
Pin p,n;
SIunits.Current i;
SIunits.Voltage u;

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

model Capacitor
extends TwoPin;

parameter SIunits.Capacitance C;

equation
C*der(u) = i;
end Capacitor;

Advantage of extends:
Common properties are defined only once!
Constraints on variables and parameters

Restriction of outputs
i.e. no connections to other outputs.
(Same yields for input)

connector OutPort
  parameter Integer n=1;
  output Real signal[n];
end OutPort;

model TorqueSensor
  SUnits.Torque tau;
  Rotational.Interfaces.Flange_a flange_a;
  Rotational.Interfaces.Flange_b flange_b;
  Blocks.Interfaces.OutPort outPort (final n=1);
  equation
    flange_a.phi = flange_b.phi;
    tau = flange_a.tau;
    tau = -flange_b.tau;
    tau = outPort.signal[1];
end TorqueSensor;

Efficient and reliable modeling

\[ y = \frac{k}{T \cdot s + 1} \cdot u \rightarrow T \cdot \dot{y} + y = k \cdot u \]

block FirstOrder
  parameter Real k=1 "gain";
  parameter Real T=0.01 "time constant";
  Blocks.Interfaces.InPort inPort (final n=1);
  Blocks.Interfaces.OutPort outPort (final n=1);
  protected
    Real y = outPort.signal[1];
    equation
      T*der(y) + y = k*u;
  end FirstOrder;
Comparison of equations and algorithms

Variant 1:

block Limiter
  parameter Real uMax= 1 "maximum value";
  parameter Real uMin=-1 "minimum value";
  Blocks.Interfaces.InPort inPort (final n=1);
  Blocks.Interfaces.OutPort outPort(final n=1);
protected
  Real u = inPort.signal[1];
  Real y = outPort.signal[1];
equation
  y = if u > uMax then uMax else if u < uMin then uMin else u;
end Limiter;

Comparison of equations and algorithms

Variant 2:

block Limiter
  parameter Real uMax= 1 "maximum value";
  parameter Real uMin=-1 "minimum value";
  Blocks.Interfaces.InPort inPort (final n=1);
  Blocks.Interfaces.OutPort outPort(final n=1);
protected
  Real u = inPort.signal[1];
  Real y = outPort.signal[1];
algorithm
  if u > uMax then y := uMax;
  elseif u < uMin then y := uMin;
  else y := u;
  end if;
end Limiter;

if-block (as in C, Fortran, etc.)

all assignments in an algorithm-section will be executed in the given order

Sunday, October 12, 2003
Multi-domain Modeling and Simulation with Modelica

22
Variant 3:

```modelica
function Limiter
  input Real uMax=2, uMin=-2;
  input Real u;
  output Real y;
  algorithm
    if u > uMax then
      y := uMax;
    elseif u < uMin then
      y := uMin;
    else
      y := u;
    end if;
end Limiter;

model test
  Real x0, x1, x2;
  equation
    x1 = Limiter(1, -1, x0);  // (means: uMax=1, uMin=-1, u=x0)
    x2 = Limiter(u=x0);       // (means: uMax=2, uMin=-2, u=x0)
end test;
```

Functions in Modelica

- **Function**, as in C or Fortran
- default-value

Different class-types in Modelica

- **type** class to define variable types
- **connector** class to define interfaces
- **model** class to define model components
- **block** model, for which all public variables are input, output, parameter or constant.
- **function** block with algorithm-section and function-call syntax
- **package** class to define libraries.
**Arrays and Matrices**

**Declaration of multidimensional arrays:**

```modelica
parameter Real v[3] = (1, 2, 3);
```

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

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

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

[...] generates matrices Matlab compatible.

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

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

---

**Array access operator**

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

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

**Access** to matrix elements:


**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]}
Matrix operations

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;

Matrix operations

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;

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;

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

for i in 1:size(s1,1) loop
for j in 1:size(s2,2) loop
s1[i,j] := s1[i,j] + s2[i,j]*v2[j];
end for;
end for;
Example: Transfer function

\[ y = \frac{b_2 s^2 + b_1 s + b_0}{a_2 s^2 + a_1 s + a_0} u \]

Transformation in \textbf{controller canonical form} (for n=m=5):

\[
\begin{align*}
\mathbf{x} &= \begin{bmatrix}
\frac{a_2}{a_1} & \frac{a_3}{a_1} & \frac{a_4}{a_1} & \frac{a_5}{a_1} \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{bmatrix} \cdot \mathbf{x} + \begin{bmatrix}
\frac{1}{a_1} \\
0 \\
0 \\
0 \\
\end{bmatrix} u \\
y &= \begin{bmatrix} b_2 - \frac{b_1}{a_1} & b_3 - \frac{b_2}{a_1} - \frac{b_0}{a_1} & b_4 - \frac{b_1}{a_1} & b_5 - \frac{b_2}{a_1} - \frac{b_0}{a_1} \end{bmatrix} \cdot \mathbf{x} + \frac{b_1}{a_1} u
\end{align*}
\]

\textbf{Example: Transfer function}

partial block SISO "Single Input/Single Output block"

\begin{verbatim}
Modelica.Blocks.Interfaces.InPort  inPort (final n=1) "input";
Modelica.Blocks.Interfaces.OutPort outPort(final n=1) "output";
Real u = inPort.signal[1];
Real y = outPort.signal[1];
end SISO;

block TransferFunction
  extends SISO;
  parameter Real b[1] = {1};
  parameter Real a[1] = {1, 1};
protected
  constant Integer na = size(a, 1);
  constant Integer nb(max=na) = size(b, 1);
  constant Integer n = na-1
  Real x[n] "State vector";
  Real b0[n] = vector( [zeros(na-nb); b] );
equation
  der(x[2:n]) = x[1:n-1];
  a[1]*\text{der}(x[1]) + a[2:na]*x = u;
  y = (b0[2:na] - b0[1]/a[1]*a[2:na])*x + b0[1]/a[1]*u;
end TransferFunction;
\end{verbatim}
For-loop, indexing and Arrays

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;

Pre-defined Array-Functions

<table>
<thead>
<tr>
<th>Modelica</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndims(A)</td>
<td>Returns the number of dimensions k of array expression A, with k =&gt; 0.</td>
</tr>
<tr>
<td>size(A,i)</td>
<td>Returns the size of dimension i of array expression A where i shall be &gt; 0 and &lt;= ndims(A).</td>
</tr>
<tr>
<td>scalar(A)</td>
<td>Returns the single element of array A. size(A,i) = 1 is required for 1 &lt;= i &lt;= ndims(A).</td>
</tr>
<tr>
<td>vector(A)</td>
<td>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(A) = 1.</td>
</tr>
<tr>
<td>matrix(A)</td>
<td>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 &lt;= i &lt;= ndims(A).</td>
</tr>
<tr>
<td>transpose(A)</td>
<td>Permutes the first two dimensions of array A. It is an error, if array A does not have at least 2 dimensions.</td>
</tr>
<tr>
<td>outerproduct(v1,v2)</td>
<td>Returns the outer product of vectors v1 and v2 (= matrix(v1)*transposed(matrix(v2))).</td>
</tr>
<tr>
<td>identity(n)</td>
<td>Returns the n x n Integer identity matrix, with ones on the diagonal and zeros at the other places.</td>
</tr>
<tr>
<td>diagonal(v)</td>
<td>Returns a square matrix with the elements of vector v on the diagonal and all other elements zero.</td>
</tr>
<tr>
<td>zero(n1,n2,...)</td>
<td>Returns the n1 x n2 x ... Integer array with all elements equal to zero (n1 =&gt; 0).</td>
</tr>
<tr>
<td>ones(n1,n2,...)</td>
<td>Returns the n1 x n2 x ... Integer array with all elements equal to one (n1 =&gt; 0).</td>
</tr>
<tr>
<td>fill(x,n1,n2,...)</td>
<td>Returns the n1 x n2 x ... array with all elements equal to scalar expression x which has to be a subtype of Real, Integer, Boolean or String (n1 =&gt; 0). The returned array has the same type as x.</td>
</tr>
</tbody>
</table>
### Pre-defined Array-Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linspace(x1,x2,n)</td>
<td>Returns a Real vector with n equally spaced elements, such that v[i] = x1 + ((x2-x1)/(n-1))i for 1 &lt;= i &lt;= n. It is required that n &gt;= 2.</td>
</tr>
<tr>
<td>min(A)</td>
<td>Returns the smallest element of array expression A.</td>
</tr>
<tr>
<td>max(A)</td>
<td>Returns the largest element of array expression A.</td>
</tr>
<tr>
<td>sum(A)</td>
<td>Returns the sum of all the elements of array expression A.</td>
</tr>
<tr>
<td>product(A)</td>
<td>Returns the product of all the elements of array expression A.</td>
</tr>
<tr>
<td>symmetrize(A)</td>
<td>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[i,j] = A[i,j], if i &lt;= j, B[i,j] = A[j,i], if i &gt; j.</td>
</tr>
<tr>
<td>cross(x,y)</td>
<td>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]; ]).</td>
</tr>
<tr>
<td>skew(x)</td>
<td>Returns the 3 x 3 skew symmetric matrix associated with a 3-dim-vector, i.e., skew(x) = skew(x)*y; skew(x) = [0, -x[3], x[2]; x[3], 0, -x[1]; -x[2], x[1], 0].</td>
</tr>
</tbody>
</table>

### Arrays of components

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

```modelica
for i in 1:9 loop
  connect(R[i].p, R[i+1].n); // serial connection
end for;
```

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

**Example:**

**electrical line with losses**
Arrays of components

model ULine "Lossy RC Line"
  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

Modeling of physical fields

Modeling of physical fields, such as

- gravitation,
- electrical field,
- (constant) temperature or pressure of the environment

can be done in Modelica with the

**inner/outer** language element

(= more selective than **global variables**)
The inner / outer concept

A component c with the `outer` prefix in an object B refers to a component with the same name and having the `inner` prefix in an object A, provided B is contained in the hierarchy of A.

Example:

```model Component
  outer Real T0;
  Real T;
  equation
  T = T0;
end Component;

model Environment
  inner Real T0;
  Component c1, c2; // c1.T0=c2.T0=T0
  parameter Real a=1;
  equation
  T0 = Modelica.Math.sin(a*time);
end Environment;

model SeveralEnvironments
  Environment e1(a=1), e2(a=2)
end SeveralEnvironments```

The inner / outer concept

A component c with the `outer` prefix in an object B refers to a component with the same name and having the `inner` prefix in an object A, provided B is contained in the hierarchy of A.

Example:

```model Component
  outer Real T0;
  Real T;
  equation
  T = T0;
end Component;

model Environment
  inner Real T0;
  Component c1, c2; // c1.T0=c2.T0=T0
  parameter Real a=1;
  equation
  T0 = Modelica.Math.sin(a*time);
end Environment;

model SeveralEnvironments
  Environment e1(a=1), e2(a=2)
end SeveralEnvironments```
Example: Heat exchange

Physical connection between all components and their environment, such as heat exchange implies many explicit connections.

Example: Heat exchange

Also a connector can have the prefix outer and is then a reference to the corresponding inner connector. A connection to the connector declared outer is therefore implicitly a connection to the global inner connector.

A new component in the hierarchy gets automatically connected.
Model of point mass in gravitational field

All kinds of objects can be declared as inner/outer. For example, functions and connectors can be used.

Point mass (equations independent of environment)

\[
\begin{align*}
\text{parallel field} & \quad \text{central field} \\
\end{align*}
\]

Model of point mass in gravitational field

```model Particel
parameter Real \( m = 1 \);
outer function gravity = gravityInterface;
Real \( r[3]\)(start = \( \{1,1,0\}\)) "position";
Real \( v[3]\)(start = \( \{0,1,0\}\)) "velocity";
equation
\[
\text{der}(r) = v;
\]
\[
m \cdot \text{der}(v) = m \cdot \text{gravity}(r);
\]
end Particle;

partial function gravityInterface
input Real \( r[3] \) "position";
output Real \( g[3] \) "gravity acceleration";
end gravityInterface;

function uniformGravity
extends gravityInterface;
algorithm
\[
g := \{0, -9.81, 0\};
\]
end uniformGravity;

function pointGravity
extends gravityInterface;
parameter Real \( k=1 \);
protected
Real \( n[3] \);
algorithm
\[
n := -r/\text{sqrt}(r \cdot r);
g := k/(r \cdot r) \cdot n;
\]
end pointGravity;
```

Sunday, October 12, 2003
Multi-domain Modeling and Simulation with Modelica

64
Use of point masses in different gravitational fields

```model Compositel
inner function gravity =
  pointGravity(k=1);
Particle p1, p2(r=start={1,0,0});
end Compositel;
```

```model Composite2
inner function gravity =
  uniformGravity;
Particle p1, p2(v=start={0,0.9,0});
end Composite2;
```

```model system
Composite1 c1;
Composite2 c2;
end system;
```

Hybrid Modeling

**Goal:**
Modeling and simulation of discontinuous and/or non-differentiable systems.

Examples for „**simple Discontinuities**“:
- Discontinuous input functions (i.e. „step functions“)
- Sample system (digital controller)
- Hysteresis

Examples for „**Systems with variable structure**“:
- ideal Diode
- ideal Thyristor
- Coulomb friction
- Clutch based on Coulomb friction
Pre-defined discontinuous components

i.e. **Sample systems:**
ModelicaAdditions.Blocks.Discrete

Each block has a **continuous input and output signal.** This signal is **sampled** within each block based on the corresponding sample time.

Therefore, the components can be easily mixed with „continuous“ blocks.

---

Pre-defined discontinuous components

Example: **Clutch** and **Brake** from the Modelica.Mechanics.Rotational library

The input signal defines the pressing force of the clutch and the break

Clutch\_mode and Brake\_mode

= 2: clutch/brake is not active
= 1: forward sliding
= 0: stuck (no relative motion)
=-1: backward sliding
Automatic gear box with 6 clutches (friction elements)

Modelica model

Example:

Modelica:

\[ y = \text{if } u > 0 \text{ then } 1 \text{ else } -1; \]

continuation of branch for switching point detection

Modeling of events

\[ y = y(t^+) \]

\[ \text{pre}(y) = y(t^-) \]
### Modeling of events

Relations, such as $u > 0$, automatically trigger state or time events to handle discontinuities in a numerical sound way.

This feature can be switched off with the `noEvent()` Operator:

$$
y = \begin{cases} 
  u^2 & \text{if } \text{noEvent}(u >= 0) \text{ then } u^2 \text{ else } u^3; \\
  1/u & \text{if } \text{noEvent}(u > \text{eps}) \text{ then } 1/u \text{ else } 1/\text{eps};
\end{cases}
$$

At a discontinuous point yields:

- $y$ is the **right limit**
- $\text{pre}(y)$ is the **left limit**

### Example: noEvent-Operator

```model
twoPoint
  parameter Real w=5, A=1.5;
  Real u, y1, y2;
  equation
    u = A*Modelica.Math.sin(w*time);
    y1 = if u > 0 then 1 else -1;
    y2 = if noEvent(u > 0) then 1 else -1;
end twoPoint;
```

Integrator = DASSL, 50 output points
Discrete variables and the when-statement

Additional equations can be declared at an event using the when-statement. These equations are de-activated during the continuous integration.

\[
\text{when <condition> then}
\begin{align*}
& \text{<equations>}
\end{align*}
\text{end when;}
\]

When <condition> is true, <equations> are calculated.

Example: when-Operator

```
model whendemo
  parameter Real A=1.5, w=4;
  Real u;
  Boolean b;
  equation
    u = A*Modelica.Math.sin(w*time)
    when u > 0 then
      b = not pre(b);
    end when;
  end whendemo;
```

![Graph of u and b](image)
Hybrid Operators in Modelica

<table>
<thead>
<tr>
<th>Modelica</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial()</td>
<td>Returns true at the simulation start (where time is equal to time.start).</td>
</tr>
<tr>
<td>terminal()</td>
<td>Returns true at the end of a succesful simulation</td>
</tr>
<tr>
<td>noEvent(expr)</td>
<td>Real elementary relations within expr are taken literally i.e., no state or time event is triggered.</td>
</tr>
<tr>
<td>sample(start,interval)</td>
<td>Returns true and triggers time events at time instants “start + i*interval” (i=0,1,...). During continuous integration the operator returns always false. The starting time “start” and the sample interval “interval” need to be parameter expressions and need to be a subtype of Real or Integer.</td>
</tr>
<tr>
<td>pre(v)</td>
<td>Returns the “left limit” y(t-0) of variable y(t) at a time instant t.</td>
</tr>
<tr>
<td>edge(b)</td>
<td>Is expanded into “(b and not pre(b))” for Boolean variable b.</td>
</tr>
<tr>
<td>change(v)</td>
<td>Is expanded into “(v&lt;&gt;pre(v))”</td>
</tr>
<tr>
<td>reinit(x, expr)</td>
<td>Reinitializes state variable x with expr at an event instant. Argument x need to be (a) a subtype of Real and (b) the der-operator need to be applied to it. expr need to be an Integer or Real expression. The reinit operator can only be applied once for the same variable x.</td>
</tr>
</tbody>
</table>

Re-Initialization of states (Bouncing Ball)

```model bouncingBall
  parameter Real e=0.7;
  parameter Real g=9.81;
  Real h(start=1);
  Real v;
  equation
    der(h) = v;
    der(v) = -g;
  when h <= 0 then
    reinit(v, -e*pre(v));
  end when;
end bouncingBall;
```
Model instantiation gives \textbf{implicit} DAE
(Differential Algebraic Equation system)

\[
F(t, \frac{dx}{dt}, x, w, p, u, y) = 0
\]

What are known variables depend on problem formulation
- known forces and torque, unknown positions
- known positions, velocities and accelerations, unknown required force and torques

Direct use of DAE solver not feasible:
- dimension of \(w\) (auxiliary variables) high
- large Jacobian gives inefficient simulation

Example - Simple Circuit

\[
F(t, \frac{dx}{dt}, x, w, p, u, y) = 0
\]
Sorting of Equations

Original
\[ R_1.v = AC.Vp - R_1.vn \]
\[ R_1.R*R_1.i = R_1.v \]
\[ R_2.v = AC.Vp - L.vp \]
\[ R_2.R*L_1.i = R_2.v \]
\[ C.v = R_1.vn - G.Vp \]
\[ C.C*der(C.v) = R_1.i \]
\[ L.v = G.Vp - G.Vp \]
\[ B.a*der(L_1.i) = L.Vp - G.Vp \]
\[ AC.v = AC.Vp - G.Vp \]
\[ AC.Vp = AC.Vp - G.Vp \]
\[ AC.Vp*\sin(2*PI*AC.freq*time) \]
\[ G.Vp = 0 \]
\[ C.v = R_1.vn - G.Vp \]
\[ R_1.i = (R_1.i + L.i) \]
\[ AC.Vp = AC.Vp - G.Vp \]
\[ C.C*der(C.v) = R_1.i \]
\[ G.i = AC.i + R_1.i + L.i \]
\[ R_2.R*L_1.i = R_2.v \]
\[ R_2.v = AC.Vp - L.Vp \]
\[ L.v = L.Vp - G.Vp \]
\[ L.L*der(L_1.i) = L.Vp - G.Vp \]

Sorted
\[ G.Vp = 0 \]
\[ AC.Vp = G.Vp \]
\[ AC.VA*\sin(2*PI*AC.freq*time) \]
\[ C.v = R_1.vn - G.Vp \]
\[ R_1.v = AC.Vp - R_1.vn \]
\[ R_1.R*R_1.i = R_1.v \]
\[ R_2.R*L_1.i = R_2.v \]
\[ G.i = AC.i + R_1.i + L.i \]
\[ AC.i = (R_1.i + L.i) \]
\[ AC.v = AC.Vp - G.Vp \]
\[ der(C.v) = R_1.i / C.C \]
\[ G.i = AC.i + R_1.i + L.i \]
\[ R_2.v = R_2.R * L.i \]
\[ R_2.v = AC.Vp - R_2.v \]
\[ L.v = L.Vp - G.Vp \]
\[ der(L_1.i) = L.Vp - G.Vp / L.L \]

\[ f(t, \text{der}(x), w, p, u, y) = 0; \quad x = [C.v, L.i] \]

\[ dx \]

\[ dt = f(t, x, p, u) \]

Sunday, October 12, 2003
Multi-domain Modeling and Simulation with Modelica

Solving Equations

\[ G.Vp = 0 \]
\[ AC.Vp - G.Vp = AC.VA* \]
\[ AC.VA*\sin(2*PI*AC.freq*time) + G.Vp \]
\[ C.v = R_1.vn - G.Vp \]
\[ R_1.v = AC.Vp - R_1.vn \]
\[ R_1.R*R_1.i = R_1.v \]
\[ AC.i = (R_1.i + L.i) \]
\[ AC.v = AC.Vp - G.Vp \]
\[ der(C.v) = R_1.i / C.C \]
\[ G.i = AC.i + R_1.i + L.i \]
\[ R_2.R*L_1.i = R_2.v \]
\[ R_2.v = AC.Vp - L.Vp \]
\[ L.v = L.Vp - G.Vp \]
\[ L.L*der(L_1.i) = L.Vp - G.Vp \]

\[ dx \]

\[ dt = f(t, x, p, u) \]

Sunday, October 12, 2003
Multi-domain Modeling and Simulation with Modelica
Summary - Simple Circuit - ODE

ODE:

G: \[ G \cdot V_p = 0 \]
AC: \[ AC \cdot V_p = AC \cdot VA \times \sin(2 \pi F T \cdot \text{freq} \times \text{time}) + G \cdot V_p \]
C: \[ R_1 \cdot V_n = G \cdot V_p + C \cdot V \]
R1: \[ R_1 \cdot V = AC \cdot V_p - R_1 \cdot V_n \]
\[ R_1 \cdot i = R_1 \cdot v / R_1 \cdot R \]

Circuit: \[ AC \cdot i = -(R_1 \cdot i + L \cdot i) \]
AC: \[ AC \cdot v = AC \cdot V_p - G \cdot V_p \]
C: \[ \text{der}(C \cdot v) = R_1 \cdot i / C \cdot C \]
Circuit: \[ G \cdot i = AC \cdot i + R_1 \cdot i + L \cdot i \]
R2: \[ R2 \cdot V = R2 \cdot R \cdot L \cdot i \]
L: \[ L \cdot V_p = AC \cdot V_p - R2 \cdot V \]
L: \[ L \cdot V = V_p - G \cdot V_p \]
\[ \text{der}(L \cdot i) = (L \cdot V_p - G \cdot V_p) / L \cdot L \]

\[ \frac{dx}{dt} = f(t, x, p, u) \]

Data flow:

Structural Processing

- Conversion to explicit ODE form
  \[ \frac{dx}{dt} = f(t, x, p, u) \]
  \[ y = g(t, x, p, u) \]

- Graph theoretical methods used (bipartite graph)
  for assigning causalities and sorting equations
  (strongly connected components, Tarjan)

- Gives sequence of assignments statements
  (solver does not handle w)
  and simultaneous systems of equations (algebraic loops)
  - finding minimal loops
- Jacobian - Block Lower Triangular
- Tearing used to reduce sparse matrices
Symbolic Formula Manipulation

Formula manipulation
- abstract syntax tree for expressions
- algebraic transformation rules recursively applied to tree, such as:

\[(a + bx) - (c + dx) \rightarrow a - c + (b - d)x\]

Example of manipulations
- solving linear equations and certain non-linear equations
- finding matrix coefficients for linear systems of equations
- solving small linear systems of equations
- finding Jacobian for nonlinear systems of equations

Specialized computer algebra algorithms needed
- high capacity (> 100 000 equations)
- appropriate heuristics

Higher index DAE's

- Constraints on differentiated variables
- Dependent initial conditions
- Reduced degree-of-freedom

- Example: capacitors in parallel, rigidly connected masses

- Cannot solve for all derivatives
- Differentiate certain equations symbolically
  algorithm by Pantelides
- Automatic state variable selection
Capacitors in Parallel

\[ P_I_0 = 3.14159265358979; \]
\[ V_{g,u,v} = A_0 \sin(6.28318530717959*f_0*Time) + v_0; \]
\[ V_{g,p,v} = V_{g,u,v}; \]
\[ R_{l,n,v} = C_2_v; \]
\[ R_{l,v} = V_{g,p,v} - R_{l,n,v}; \]
\[ V_{g,n,l} = R_{l,v}/R_{l,R}; \]
\[ C_{1,der,v} = V_{g,n,l}/(C_2+C_1); \]

**InitialSection**

\[ P_I_0 = 3.14159265358979; \]
\[ V_{g,n,v} = 0; \]
\[ G_p,v = 0; \]
\[ C_{1,n,v} = 0; \]
\[ C_{2,n,v} = 0; \]

**DynamicsSection**

\[ V_{g,u,v} = A_0 \sin(6.28318530717959*f_0*Time) + v_0; \]
\[ V_{g,p,v} = V_{g,u,v}; \]
\[ R_{l,n,v} = C_2_v; \]
\[ R_{l,v} = V_{g,p,v} - R_{l,n,v}; \]
\[ V_{g,n,l} = R_{l,v}/R_{l,R}; \]
\[ C_{1,der,v} = V_{g,n,l}/(C_2+C_1); \]

**AcceptedSection**

\[ C_{1,i} = C_1*C_{1,der,v}; \]
\[ C_{2,i} = V_{g,n,i} - C_{1,i}; \]
\[ R_{l,n,der,v} = C_{1,der,v}; \]
\[ C_{2,der,v} = C_{1,der,v}; \]

Simplifications of equations

- General library models
- Needs specialization in its environment
- Example: 3D mechanical model constrained to move in 2D
  \[ \text{AxisOfRotation} = \{0, 0, 1\} \]

- Manipulations:
  - substitute constants and fixed parameters
  - partial evaluation of expressions:
    \[ a \times \text{expr} = 0, \ \text{expr/expr} = 1, \text{etc} \]

- Reduction in number of arithmetic operations:
  - typically a factor of 10
Singular systems - high index DAE

**DAE:**

\[ 0 = f(x, x, y, t) \]

with **singular** Jacobi-Matrix

\[ \left| \frac{\partial f}{\partial x} \frac{\partial f}{\partial y} \right| = 0 \]

can not be algebraically transformed to state space form, because:

There are constrains between differentiated variables \( x \), such that all \( x \)'s are not independent (can not be given independent initial conditions).

---

**Dummy Derivative Method:**

(1) Search **subsets** of equations, which have a **singular** Jacobian Matrix. Sufficient Condition:

number of equations > number of unknowns

(2) **Differentiate** the equation subsets and add the resulting equations to the DAE.

(3) From the **singular** subset of equations select **Dummy-derivatives** \( x^d \) until these equations are **regular**. (that means: treat them as unknown algebraic Variables (like \( y \)); before, \( x^d \) has been assumed known).

(4) Analyze the **complete DAE** again, that means repeat from point 1 until the Jacobian matrix of the DAE is **regular**.
remark:
A singular DAE can not be transformed into explicit state space form, when the DAE does not have a unique solution. Example:

\[
\begin{align*}
0 &= f_1(\dot{x}, x, y_2) \\
0 &= f_2(y_1) \\
0 &= f_3(y_1)
\end{align*}
\]

3 equations for the 3 unknown variables \(x, y_1, y_2\). If the last two functions are identical, there is an infinite number of solutions, else there is a contradiction for calculating \(y\) and there is no solution. Such a DAE is called structurally inconsistent. (This property is recognized by Dymola during translation).

Test (singular systems)

Analyze the following systems:

- Write down the number of local states for each component
- Which constraint conditions exist (Write down equations)?
- How many states exist in the total system?
Number of states

\[ J_1 \phi = i * J_2 \phi \]
\[ J_1 w = i * J_2 w \]

Number of states

\[ J_1 \phi = i1 * J_2 \phi \]
\[ J_2 \phi = i2 * J_3 \phi \]
\[ J_1 w = i1 * J_2 w \]
\[ J_2 w = i2 * J_3 w \]

Number of states

\[ w_0 \phi = (w_1 \phi + w_2 \phi) / 2 \]
\[ s_0 \phi = (s_1 \phi + s_2 \phi) / 2 \]

Number of states

\[ s_0 \phi = (s_1 \phi + s_2 \phi) / 2 \]
\[ w_0 \phi = (w_1 \phi + w_2 \phi) / 2 \]
Number of states

C1.v = C2.v

1

C1.v = C2.v

2

C1.v = C2.v

1

C1.v = constVolt.v

0

C1.v = C2.v

1

tempSource.T = heatCapacitance1.T

1

1
Summary

- **Introduction**
  - Industrial Application Examples
  - Composition Diagram versus Block Diagram

- **Modeling with Modelica**
  - Flat and Hierarchical Models
  - Special Model classes
  - Matrices, Arrays and Arrays of components
  - Physical Fields
  - Hybrid Modeling
  - Symbolic processing