Modelica Tutorial for Beginners

Multi-domain Modeling and Simulation

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

Outline

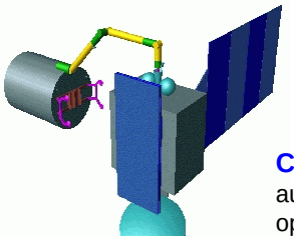
- Introduction
  - Industrial Application Examples
  - Composition Diagram versus Block Diagram
  - The Modelica Association
- Modeling with Modelica
  - Flat and Hierarchical Models
  - Special Model Classes
  - Matrices, Arrays and Arrays of Components
  - Physical Fields
  - Hybrid Modeling

DLR - Institute of Robotics and Mechatronics

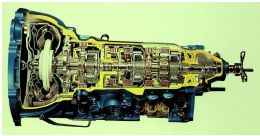
Space-Robotic

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

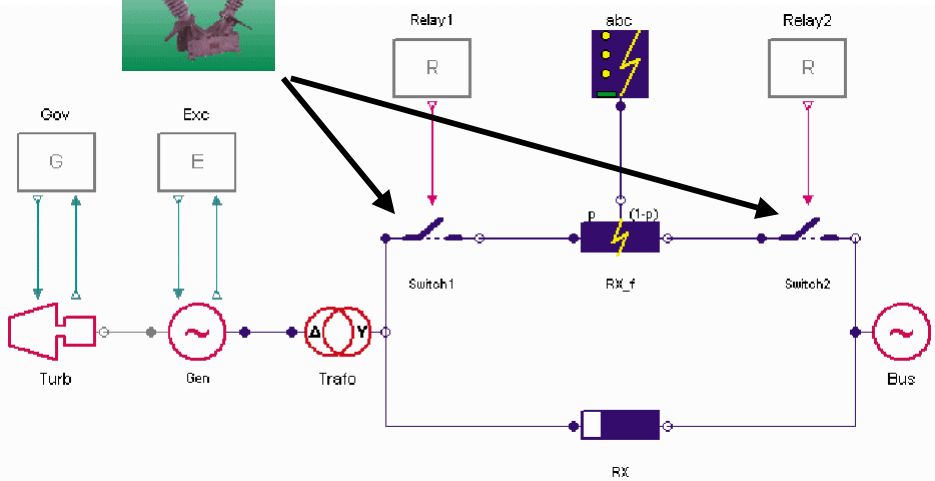


Control Design for fly-by-wire, automatic landing, etc.; based on optimizing of parameter

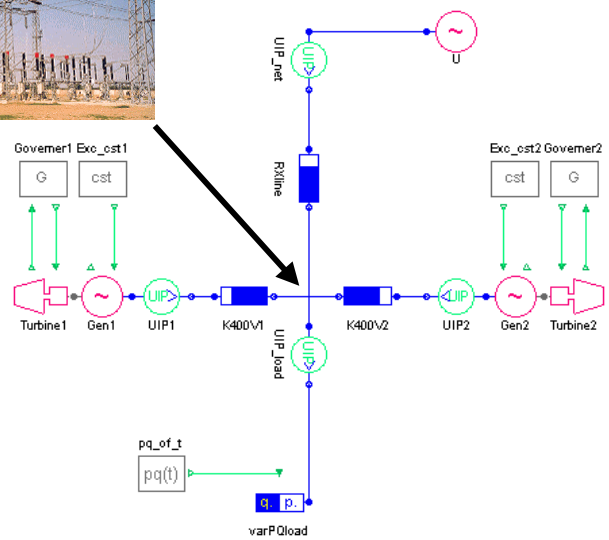


Automobile
Modeling, simulation of mechatronical components

Test of Protection Devices in Power Systems



Power Flow Analysis in Power Systems



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

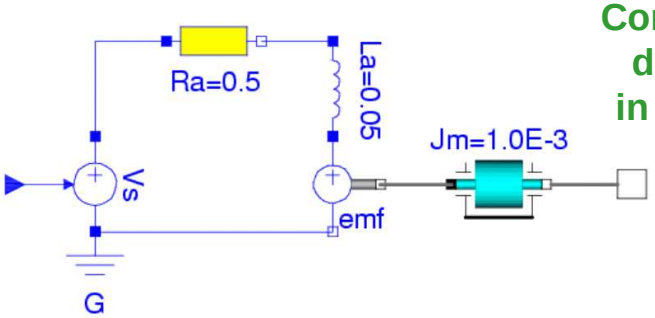
BMW 3-series chassis driving over icy patch on the road. Off-center additional weight on the roof.

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

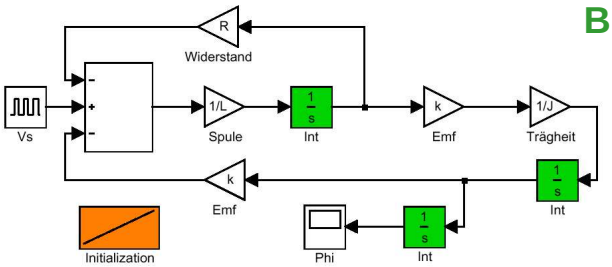
The property to “figure out” how to use a component optimally in different environments is a condition for re-usable, object-oriented model libraries, like the VehicleDynamics library

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

Component diagram in Dymola



Block diagram in Simulink



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

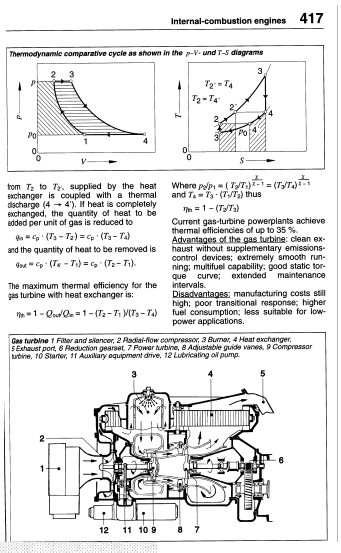
Modeling Knowledge

Where to find?

- Books
- Experts

Main Idea:

Computer based storage of modeling knowledge



Internal-combustion engines 417

Thermodynamic comparative cycle as shown in the p–V- und T–S diagrams

from T_2 to T_3, supplied by the heat exchanger is coupled with a thermal discharge (4 → 4'). If heat is completely exchanged, the quantity of heat to be added per unit of gas is reduced to

$$q_{in} = c_p \cdot (T_3 - T_2) = c_p \cdot (T_3 - T_4)$$

and the quantity of heat to be removed is

$$q_{out} = c_p \cdot (T_4 - T_1) = c_p \cdot (T_2 - T_1).$$

The maximum thermal efficiency for the gas turbine with heat exchanger is:

$$\eta_{th} = 1 - Q_{out}/Q_{in} = 1 - (T_2 - T_1)/(T_3 - T_4)$$

Where $p_2/p_1 = (T_2/T_1)^{\frac{\kappa}{\kappa-1}} = (T_3/T_4)^{\frac{\kappa}{\kappa-1}}$ and $T_4 = T_3 \cdot (T_1/T_2)$ thus

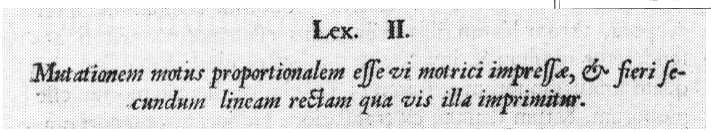
$$\eta_{th} = 1 - (T_2/T_3)$$

Current gas-turbine powerplants achieve thermal efficiencies of up to 35 %.

Advantages of the gas turbine: clean exhaust without supplementary emissions-control devices; extremely smooth running; multifuel capability; good static torque curve; extended maintenance intervals.

Disadvantages: manufacturing costs still high; poor transitional response; higher fuel consumption; less suitable for low-power applications.

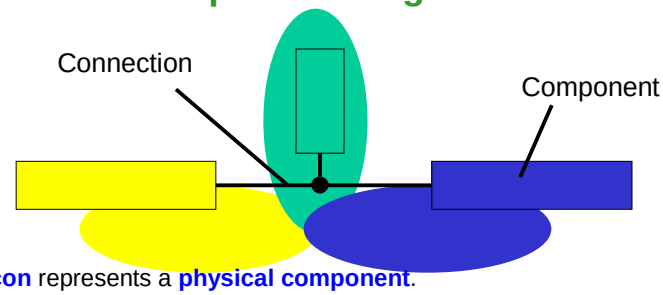
Gas turbine 1 Filter and silencer, 2 Radial-flow compressor, 3 Burner, 4 Heat exchanger, 5 Exhaust port, 6 Reduction gearset, 7 Power turbine, 8 Adjustable guide vanes, 9 Compressor turbine, 10 Starter, 11 Auxiliary equipment drive, 12 Lubricating oil pump.



Lex. II.

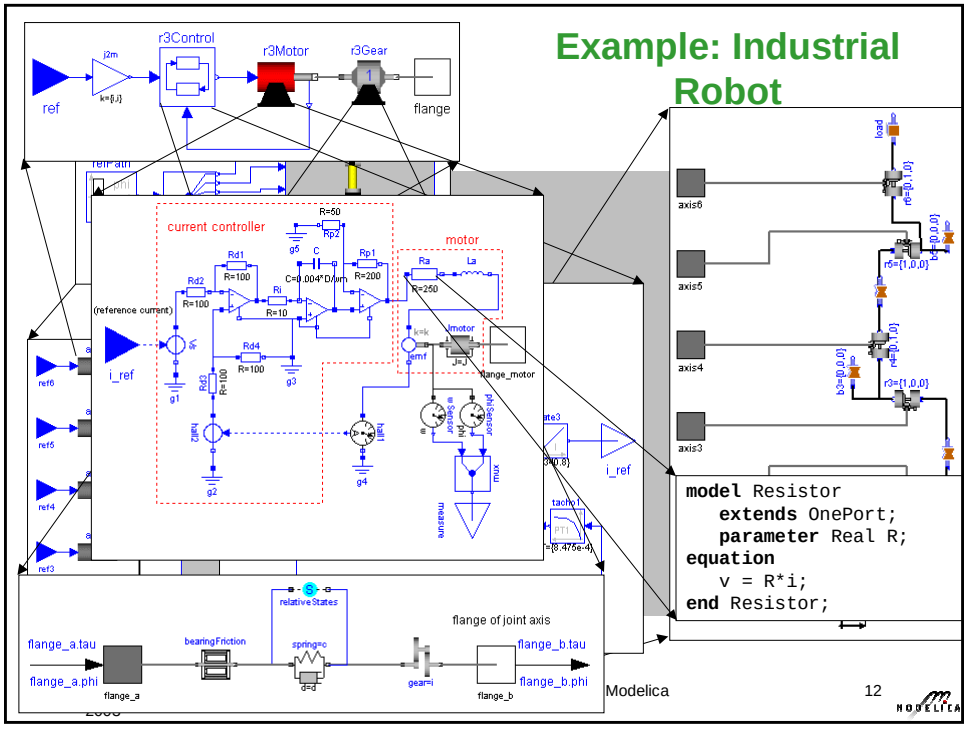
Mutationem motus proportionalem esse vi motrici impressæ, & fieri secundum lineam rectam qua vis illa imprimitur.

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

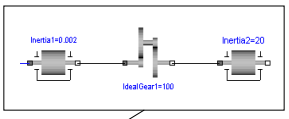
Example: Industrial Robot




```
model Resistor
   extends OnePort;
   parameter Real R;
equation
   v = R*i;
end Resistor;
```


Modeling and Simulation with Modelica tools

Graphical Editor



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

Component-Library

Modelica model (Text file: test1.mo)

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

C-function

```
void dsblock(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
  - http://www.Modelica.org/library/library.html

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 Grozman**	BrisData AB, Stockholm, Sweden
• **Johan Gunnarsson**	MathCore, Linköping, Sweden
• **Mats Jirstrand**	MathCore, Linköping, Sweden
• **Kaj Juslin**	VTT, Finland
• **Clemens Klein-Robbenhaar**	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 Sahlin**	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
• **Hansjürg Wiesmann**	ABB Corporate Research Ltd, Baden, Switzerland

Status of Modelica

http://www.Modelica.org

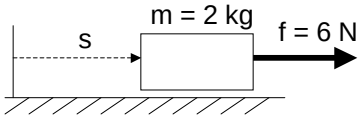
- Design started September 1996
- Modelica Version 1.0 - September 1997
- Modelica Version 1.1 - December 1998
- Modelica Version 1.2 - June 1999
- Modelica Version 1.3 - December 1999
- Modelica Version 1.4 - December 2000
- Modelica Version 2.0 - March 2002
- **Modelica Version 2.1 - October 2003**
- **3rd International Modelica Conference** 3./4. November 2003, Linköping
- > 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:



$$m \cdot \ddot{s} = f$$

graphical information

parameters (changeable before start of simulation)

new model

floating point number

name + default-value

comment

(display in dialogue)

```
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;
```

```
"Moving Mass";
"Mass of block";
"Force";
"Position of block";
"Velocity of block";
```

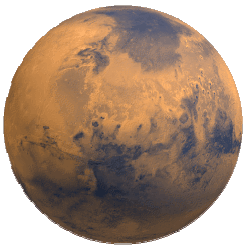
differentiation with regards to time

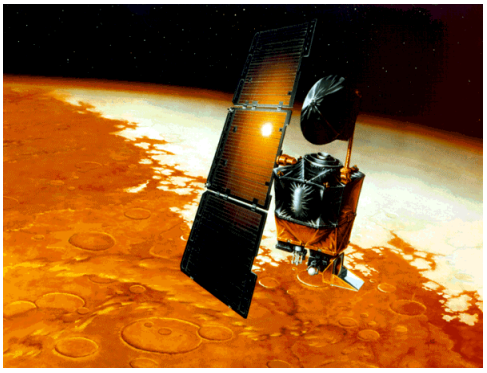
mathematical equation, (non–causal)

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



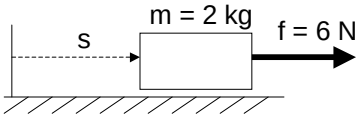


Picture from NASA/JPL/Caltech.

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:



$$m \cdot \ddot{s} = f$$

attribute of real data type

```
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;
```

unit of variables can be checked in equations

SI-Base Units

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

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:

Type	*Unit*	*in SI-base units*
Moment	Nm	kgm^2/s^2
Energy	J	kgm^2/s^2

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 =	unit =	displayUnit =
"Torque"	"N.m"	
"Energy"	"J"	
"Angle"	"rad"	"deg"

Syntax of unit-expressions, Examples:

```
kg.m2/s2, kg.m.m/(s.s), rad/s, 1/s, s-1
```

More Attributes of Real Variables:

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

Example:

```
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:

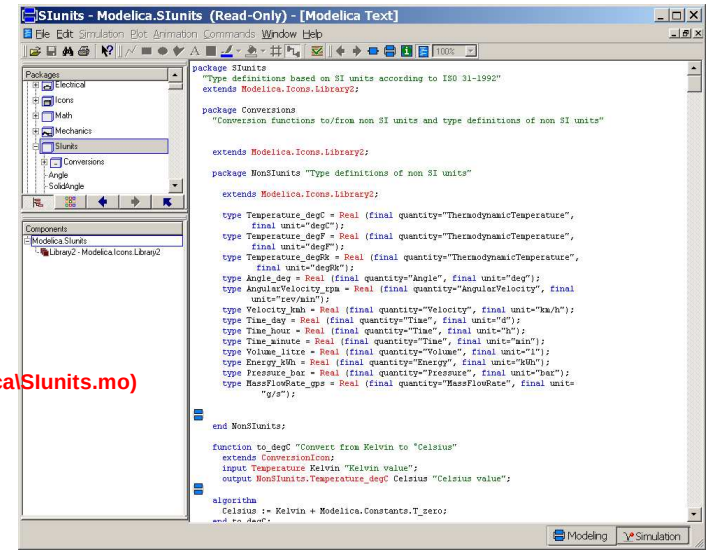
```
type Angle    = Real(quantity    = "Angle", unit = "rad",
                     displayUnit = "deg");
type Torque   = Real(quantity = "Torque", unit = "N.m");
type Mass     = Real(quantity = "Mass", unit = "kg", min=0);
type Velocity = Real(quantity = "Velocity", unit = "m/s");
```

Use of variable types:

```
parameter Mass m = 2;
Velocity v(start=3);
```

The Modelica Library Modelica.SIunits

includes all **450** ISO-standard units in form of pre-defined variable types!



(File: ...\Modelica\SIunits.mo)

Use of the Modelica.SIunits Library

Variant 1 (**full name**):

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

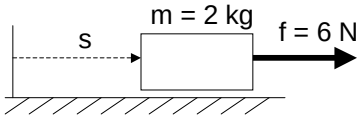
Variant 2 (**short name**):

```
package SI = Modelica.SIunits;  // Alias-Name

parameter SI.Mass m = 2;
SI.Velocity v(start=3);
```

comment

Simple Modelica-Model (Flat)

Version 3:



$$m \cdot \ddot{s} = f$$

component library

```
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;
```

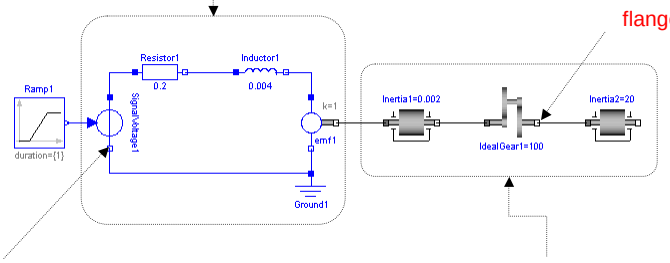
This is the prefered style of modeling!

Hierarchical Modelica Model (Model of a simple drive train)

Graphical Editor

DC-motor

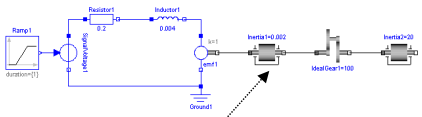
flange



electric wire

motor-inertia + ideal gear box + load-inertia

Part of drive train model (without graphics-information)



new model-class

model-class

model-instance (component)

modifier

```
model SimpleDrive
   Modelica.Mechanics.Rotational.Inertia Inertia1(J=0.002);
   Modelica.Mechanics.Rotational.IdealGear IdealGear1(ratio=100)
      ...
   Modelica.Electrical.Analog.Basic.Resistor Resistor1(R=0.2)
      ...
equation
   connect(Inertia1.flange_b, IdealGear1.flange_a);
   connect(Resistor1.n, Inductor1.p);
      ...
end SimpleDrive;
```

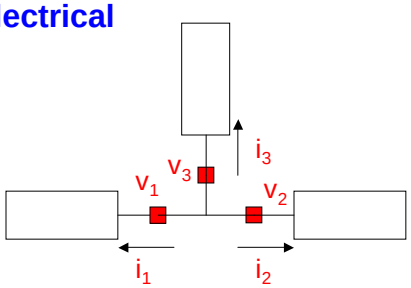
connector "flange_a" of IdealGear1

connection

connector "n" of Resistor1

Variables within interfaces (connector variables)

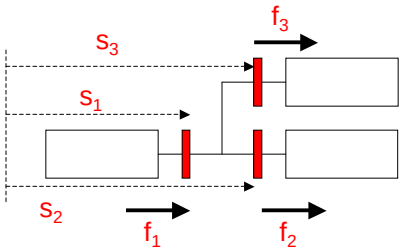
electrical



$$v_1 = v_2 = v_3$$

$$i_1 + i_2 + i_3 = 0$$

1D-mechanical



$$s_1 = s_2 = s_3$$

$$f_1 + f_2 + f_3 = 0$$

```
connect(R1.p, R2.p);
```

```
connect(m1.flange_a, m2.flange_a);
```

Two kind of variables in connectors

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

connector-class

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

new connector-class

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

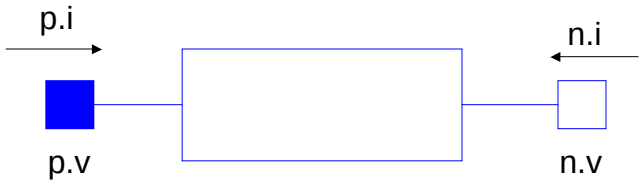
flow-variable

Interface variables within the Modelica standard library

Type	Potential variable		Flow variable	
electric	V	potential	i	current
translatorial	s	distance	f	force
rotatorial	φ	angle	τ	torque
hydraulic	p	pressure	$\dot{V}$	flow rate
thermal	T	temperature	$\dot{Q}$	heat flow
chemical	μ	chem. potential	$\dot{N}$	Current of particles

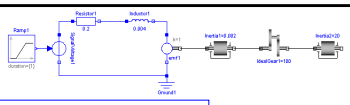
```
connector XXX
        Real PotentialVariable
   flow Real FlowVariable
end XXX;
```

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   Inertia1  (J=0.002);
   ..Rotational.IdealGear IdealGear1(ratio=100)
   ..Basic.Resistor       Resistor1 (R=0.2)
      ...
equation
   connect(Inertia1.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**)

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

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

   ...
end Modelica;
```

Modelica.Mechanics.Rotational.Inertia

The package concept in Modelica

Name-lookup within a package

Name lookup stops at "encapsulated"

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

         model Inertia
            Interfaces.Flange_a flange_a;
            Modelica.Mechanics.Rotational.Interfaces.Flange_a a;
         end Inertia;
         ...
      end Rotational;
   end Mechanics;
   ...
end Modelica;
```

equivalent definitions

The package concept in Modelica

Storage of a **hierarchical package** within one file

File: `Modelica.mo`

```
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

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

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

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

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

Modelica is case-sensitive !

```
package Mechanics
end Mechanics;
```

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

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;

   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;
```

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

incomplete model (cannot be instantiated)

inheritance

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;
      C*der(u) = i;
   end Capacitor;
end Electrical;
```

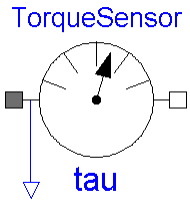
Advantage of **extends**:

Common properties are defined only once!

```
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;
```

Constraints on variables and parameters

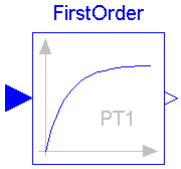
Restriction of **outputs**
i.e. no connections to other outputs.
(Same yields for **input**)




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

model TorqueSensor
   SIunits.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;
```

May not be changed any more

Efficient and reliable modeling



$$y = \frac{k}{T \cdot s + 1} \cdot u \longrightarrow T \cdot \dot{y} + y = k \cdot u$$

= model, for which all public variables are input, output, parameter or constant.

```
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;
```

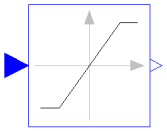
equation in declaration part

variable, which cannot be accessed from outside

Comparison of equations and algorithms

Variant 1:

Limiter



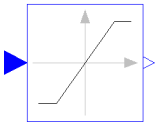
```
block Limiter
   parameter Real uMax= 1  "maximum value";
   parameter Real uMin=-1  "minimum value";
   Blocks.Interfaces.InPort  inPort (final n=1);
   Blocks.Interfaces.OutPort outPort(final n=1);
protected
   Real u = inPort.signal[1];
   Real y = outPort.signal[1];
equation
   y = if u > uMax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

if - expression

Comparison of equations and algorithms

Variant 2:

Limiter



```
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;
```

procedural part
(no equations)

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

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

assignment operator

Functions in Modelica

Variant 3:

```
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);
   x2 = Limiter(u=x0);
end test;
```

Function, as in C or Fortran

default-value

Function calls

(means: uMax=1, uMin=-1, u=x0)
(means: uMax=2, uMin=-2, u=x0)

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:

```
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])];
```


Array access operator

":" 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]}
```

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;
```

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

```
   // 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;
```

Example: Transfer function

$$y = \frac{b_1 s^m + b_2 s^{m-1} + \ldots + b_m s + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + \ldots + a_n s + a_{n+1}} \cdot u$$

Transformation in **controller canonical form** (for n=m=5):

$$\dot{\mathbf{x}} = \begin{bmatrix} -\frac{a_2}{a_1} & -\frac{a_3}{a_1} & -\frac{a_4}{a_1} & -\frac{a_5}{a_1} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \mathbf{x} + \begin{bmatrix} \frac{1}{a_1} \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot u$$

$$y = \begin{bmatrix} b_2 - a_2 \frac{b_1}{a_1} & b_3 - a_3 \frac{b_1}{a_1} & b_4 - a_4 \frac{b_1}{a_1} & b_5 - a_5 \frac{b_1}{a_1} \end{bmatrix} \cdot \mathbf{x} + \frac{b_1}{a_1} \cdot u$$

Example: Transfer function

```
partial block SISO "Single Input/Single Output block"
  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}
  parameter Real a[:] ={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[na] = vector( [zeros(na-nb);b] );
equation
  der(x[2:n]) = x[1:n-1];
  a[1]*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;
```

$$\dot{\mathbf{x}} = \begin{bmatrix} -\frac{a_2}{a_1} & -\frac{a_3}{a_1} & -\frac{a_4}{a_1} & -\frac{a_5}{a_1} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \mathbf{x} + \begin{bmatrix} \frac{1}{a_1} \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot u$$

$$y = \begin{bmatrix} b_2 - a_2 \frac{b_1}{a_1} & b_3 - a_3 \frac{b_1}{a_1} & b_4 - a_4 \frac{b_1}{a_1} & b_5 - a_5 \frac{b_1}{a_1} \end{bmatrix} \cdot \mathbf{x} + \frac{b_1}{a_1} \cdot u$$

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

Modelica	*Description*
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.

Pre-defined *Array*-Functions

Modelica	*Description*
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];

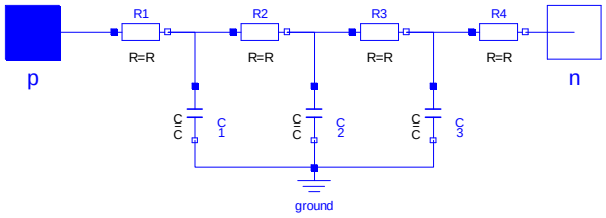
Arrays of components

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

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

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

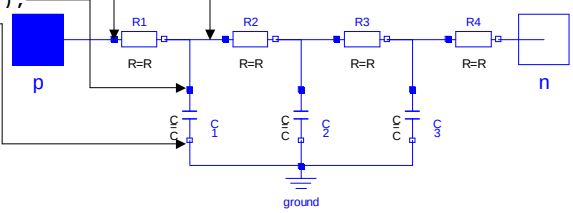
This can be utilized to discretize simple partial differential equations in a modular way.

Example: electrical line with losses



Arrays of components

```
model ULine "Lossy RC Line"
  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
```



Modeling of physical fields

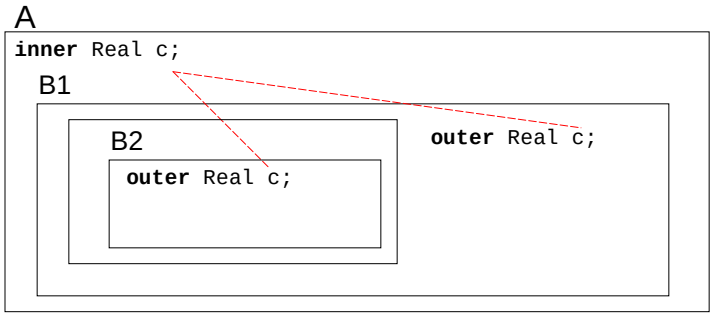
Modeling of physical fields, such as

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

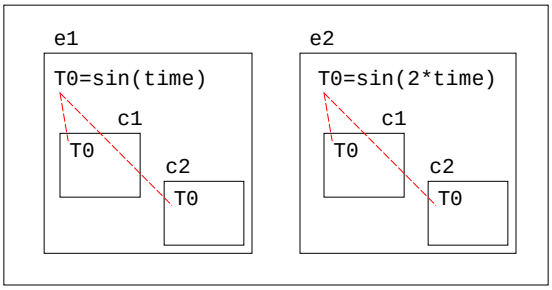
can be done in Modelica with the
inner/outer language element
(= more selective than global variables)

The inner / outer concept

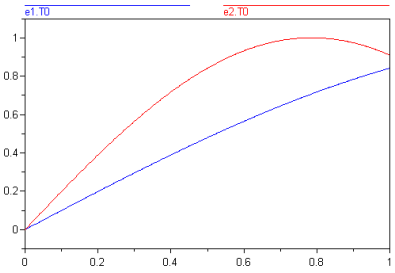
A component c with the **outer** prefix in an object B refers to a component with the same name and having the inner prefix in an object A, provided B is contained in the hierarchy of A



Example:

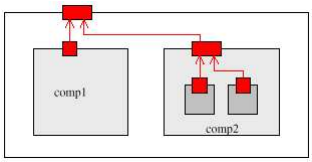



```
model Component
  outer Real T0;
  Real T;
equation
  T = T0;
end Component;

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
```



Example: Heat exchange

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

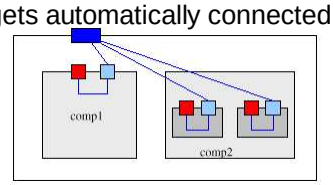



```
connector HeatCut
   SIunits.Temp_K        T;
   flow SIunits.HeatFlux q;
end HeatCut;

model Component
  HeatCut heat;
end Component;

model TwoComponents
  Component Comp[2];
  HeatCut heat;
equation
  connect(Comp[1].heat, heat);
  connect(Comp[2].heat, heat);
end TwoComponents;

model CircuitBoard
  HeatCut environment;
  Component     comp1;
  TwoComponents comp2;
equation
  connect(comp1.heat,environment);
  connect(comp2.heat,environment);
end CircuitBoard;
```


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

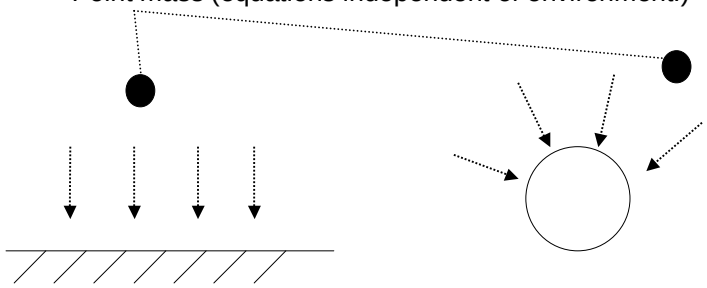



```
connector HeatCut
   SIunits.Temp_K        T;
   flow SIunits.HeatFlux q;
end HeatCut;

model Component
  outer HeatCut environment;
  HeatCut heat;
equation
  connect(heat, environment);
end Component;

model TwoComponents
  Component Comp[2];
end TwoComponents;

model CircuitBoard
  inner HeatCut environment;
  Component     comp1;
  TwoComponents comp2;
end CircuitBoard;
```

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!)

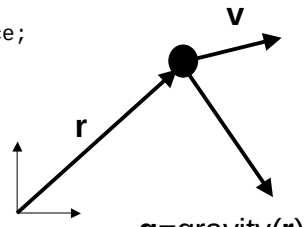


parallel field

central field

Model of point mass in gravitational field

```
model Particle
   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
     der(r) = v;
   m*der(v) = m*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;
```

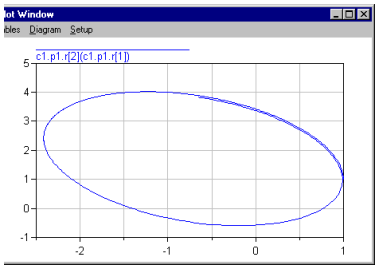


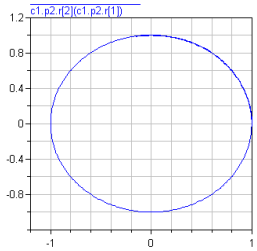
g=gravity(**r**)

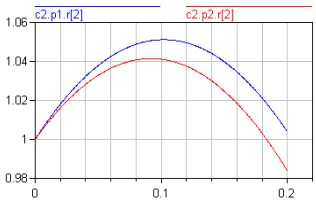
```
function pointGravity
   extends gravityInterface;
   parameter Real k=1;
protected
   Real n[3]
algorithm
   n := -r/sqrt(r*r);
   g := k/(r*r) * n;
end pointGravity;
```

Use of point masses in different gravitational fields

```
model Composite1
   inner function gravity =
                  pointGravity(k=1);
   Particle p1, p2(r(start={1,0,0}));
end Composite1;


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.

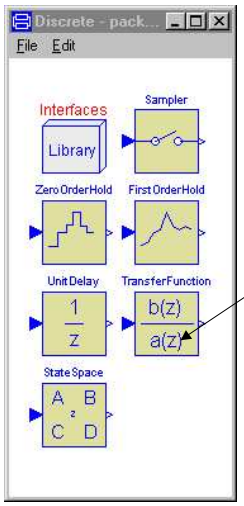
Examples for **„simple Discontinuities“**:

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



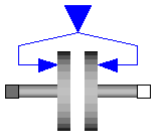
i.e. **Sample systems**:
ModelicaAdditions.Blocks.Discrete

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.

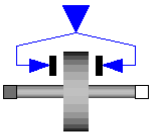
Pre-defined discontinuous components

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



Clutch

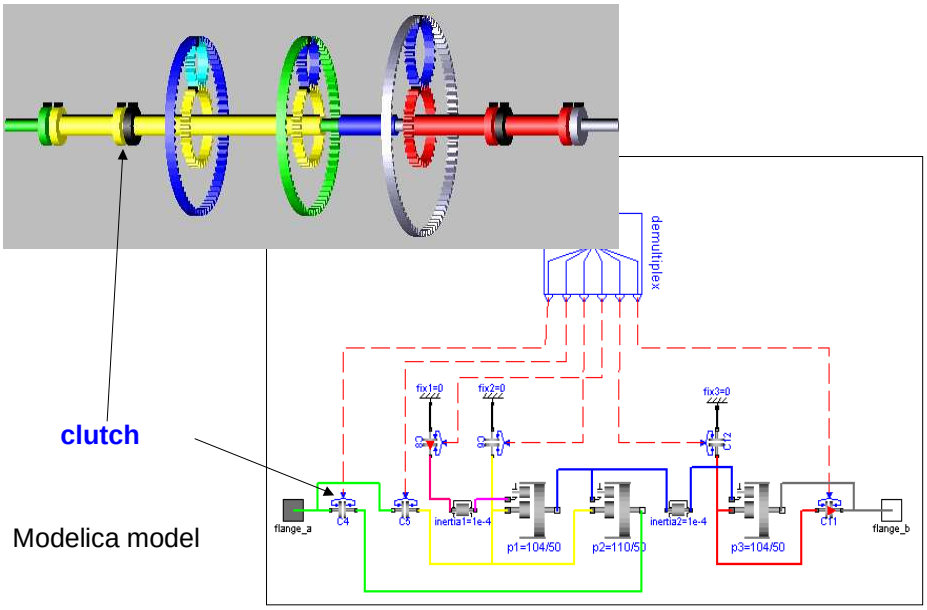
The input signal defines the pressing force of the clutch and the break



Brake

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)

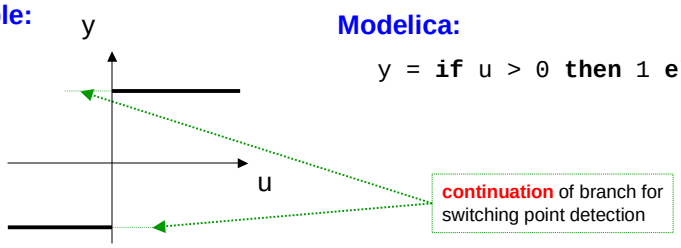


clutch

Modelica model

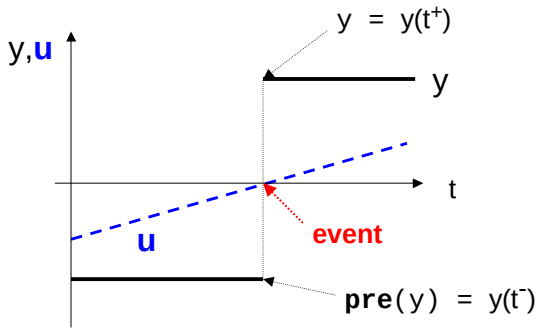
Modeling of events

Example:



Modelica:

```
y = if u > 0 then 1 else -1;
```



Modeling of events

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

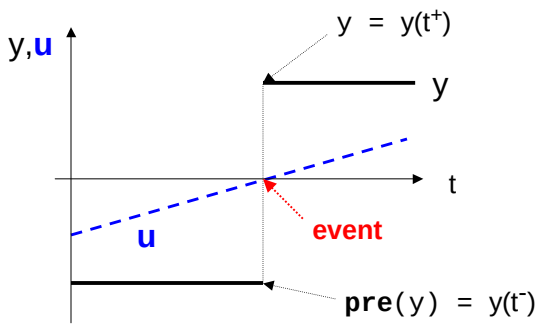
This feature can be switched off with the **noEvent**() Operator:

```
y = if noEvent(u >= 0)  then u^2 else u^3;
y = if noEvent(u > eps) then 1/u else 1/eps;
```

At a discontinuous point yields:

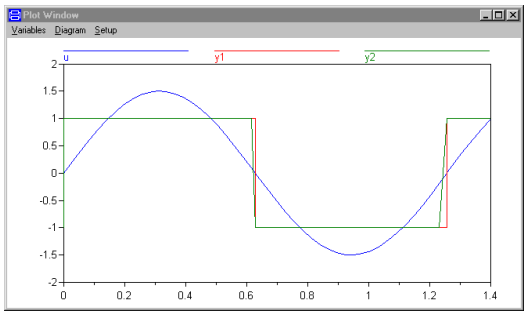
y is the **right limit**

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

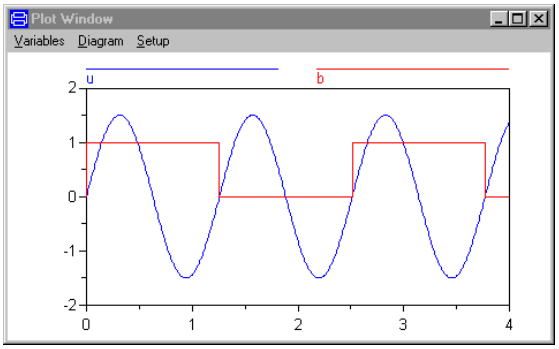
Additional equations can be declared at an **event** using the **when**-statement. These equations are **de-activated** during the **continuous integration.**

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

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

Example: when-Operator

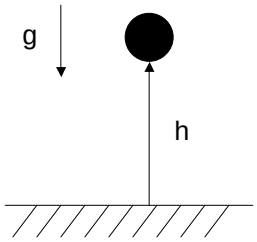
```
model whendemo
   parameter Real A=1.5, w=4;
   Real    u;
   Boolean b;
equation
   u = A*Modelica.Math.sin(w*time)
   when u > 0 then
      b = not pre(b);
   end when;
end whendemo;
```

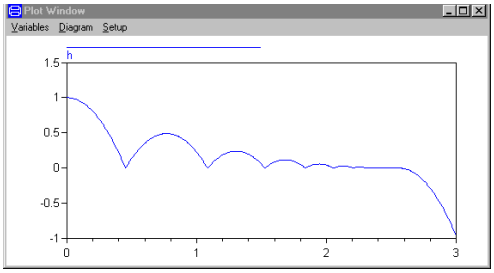


Hybrid Operators in Modelica

Modelica	*Description*
initial()	Returns true at the simulation start (where time is equal to time.start).
terminal()	Returns true at the end of a succesful simulation
noEvent(expr)	Real elementary relations within expr are taken literally i.e., no state or time event is triggered.
sample(start,interval)	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.
pre(y)	Returns the "left limit" $y(t_{pre})$ of variable y(t) at a time instant t.
edge(b)	Is expanded into "(b **and not pre**(b))" for Boolean variable b.
change(v)	Is expanded into "(v<>**pre**(v))".
reinit(x, expr)	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.

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;
```

Symbolic Processing
for Efficient Simulation

Model instantiation gives **implicit** DAE
(Differential Algebraic Equation system)

$$F(t, \frac{d\mathbf{x}}{dt}, \mathbf{x}, \mathbf{w}, \mathbf{p}, \mathbf{u}, \mathbf{y}) = \mathbf{0}$$

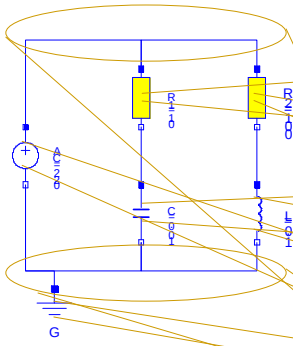
What are known variables depend on problem formulation

- 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



DAE:

R1:
```
R1.v = AC.Vp - R1.Vn
R1.R*R1.i = R1.v
```
R2:
```
R2.v = AC.Vp - L.Vp
R2.R*L.i = R2.v
```
C:
```
C.v = R1.Vn - G.Vp
C.C*der(C.v) = R1.i
```
L:
```
L.v = L.Vp - G.Vp
L.L*der(L.i) = L.Vp - G.Vp
```
AC:
```
AC.v = AC.Vp - G.Vp
AC.Vp - G.Vp =
  AC.VA*sin(2*PI*AC.freq*time)
```
G:
```
G.Vp = 0
```
Circuit:
```
G.i = AC.i + R1.i + L.i
AC.i + R1.i + L.i = 0
```

$$F(t, \frac{d\mathbf{x}}{dt}, \mathbf{x}, \mathbf{w}, \mathbf{p}, \mathbf{u}, \mathbf{y}) = \mathbf{0}$$

Sorting of Equations

Original

```
R1.v = AC.Vp - R1.Vn
R1.R*R1.i = R1.v
R2.v = AC.Vp - L.Vp
R2.R*L.i = R2.v
C.v = R1.Vn - G.Vp
C.C*der(C.v) = R1.i
L.v = L.Vp - G.Vp
L.L*der(L.i) = L.Vp - G.Vp
AC.v = AC.Vp - G.Vp
AC.Vp - G.Vp =
  AC.VA*sin(2*PI*AC.freq*time)
G.Vp = 0
G.i = AC.i + R1.i + L.i
AC.i + R1.i + L.i = 0
```

Sorted

```
G.Vp = 0
AC.Vp - G.Vp =
  AC.VA*sin(2*PI*AC.freq*time)
C.v = R1.Vn - G.Vp
R1.v = AC.Vp - R1.Vn
R1.R*R1.i = R1.v
AC.i + R1.i + L.i = 0
AC.v = AC.Vp - G.Vp
C.C*der(C.v) = R1.i
G.i = AC.i + R1.i + L.i
R2.R*L.i = R2.v
R2.v = AC.Vp - L.Vp
L.v = L.Vp - G.Vp
L.L*der(L.i) = L.Vp - G.Vp
```

F(t, der(x), x, w, p, u, y) = 0; x = [C.v, L.i]

$$\frac{d\mathbf{x}}{dt} = \mathrm{f}(t, \mathbf{x}, \mathbf{p}, \mathbf{u})$$

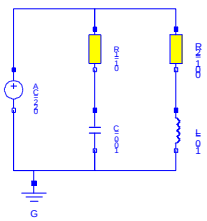
Solving Equations

```
G.Vp = 0
AC.Vp - G.Vp =
  AC.VA*sin(2*PI*AC.freq*time)
C.v = R1.Vn - G.Vp
R1.v = AC.Vp - R1.Vn
R1.R*R1.i = R1.v
AC.i + R1.i + L.i = 0
AC.v = AC.Vp - G.Vp
C.C*der(C.v) = R1.i
G.i = AC.i + R1.i + L.i
R2.R*L.i = R2.v
R2.v = AC.Vp - L.Vp
L.v = L.Vp - G.Vp
L.L*der(L.i) = L.Vp - G.Vp
```

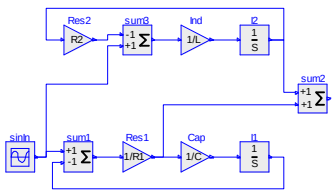
```
G.Vp = 0
AC.Vp = AC.VA*
  sin(2*PI*AC.freq*time) + G.Vp
R1.Vn = G.Vp + C.v
R1.v = AC.Vp - R1.Vn
R1.i = R1.v / R1.R
AC.i = - (R1.i + L.i)
AC.v = AC.Vp - G.Vp
der(C.v) = R1.i/C.C
G.i = AC.i + R1.i + L.i
R2.v = R2.R * L.i
L.Vp = AC.Vp - R2.v
L.v = Vp - G.Vp
der(L.i) = (L.Vp - G.Vp)/L.L
```

$$\frac{d\mathbf{x}}{dt} = \mathrm{f}(t, \mathbf{x}, \mathbf{p}, \mathbf{u})$$

Summary - Simple Circuit - ODE



Data flow:

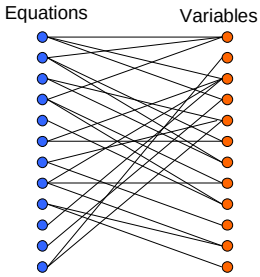


ODE:

```
G:          G.Vp = 0
AC:         AC.Vp = AC.VA*
              sin(2*PI*AC.freq*time) + G.Vp
C:          R1.Vn = G.Vp + C.v
R1:         R1.v = AC.Vp - R1.Vn
            R1.i = R1.v / R1.R
Circuit:    AC.i = - (R1.i + L.i)
AC:         AC.v = AC.Vp - G.Vp
C:          der(C.v) = R1.i/C.C
Circuit:    G.i = AC.i + R1.i + L.i
R2:         R2.v = R2.R * L.i
            L.Vp = AC.Vp - R2.v
L:          L.v = Vp - G.Vp
            der(L.i) = (L.Vp - G.Vp)/L.L
```

$$\frac{d\mathbf{x}}{dt} = \mathrm{f}(t, \mathbf{x}, \mathbf{p}, \mathbf{u})$$

Structural Processing



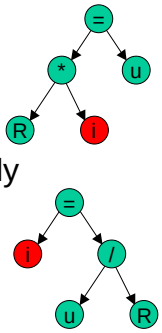
- Conversion to **explicit** ODE form

$$\frac{d\mathbf{x}}{dt} = \mathrm{f}(t, \mathbf{x}, \mathbf{p}, \mathbf{u})$$
$$\mathbf{y} = \mathbf{g}(t, \mathbf{x}, \mathbf{p}, \mathbf{u})$$

- 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:

$$(a+bx)-(c+dx) \longrightarrow a-c+(b-d)x$$

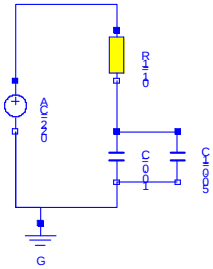
Example of manipulations

- 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

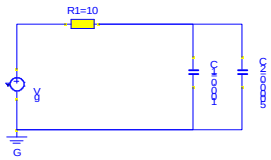


- Constraints on differentiated variables
- Dependent initial conditions
- Reduced degree-of-freedom

- 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 Parallell




```
DynamicsSection
Vg_u_v = A_0*sin(6.28318530717959*f_0*Time)+v0_0;
Vg_p_v = Vg_u_v;
R1_n_v = C2_v;
R1_v = Vg_p_v-R1_n_v;
Vg_n_i = R1_v/R1_R;
```

C1_der_v = Vg_n_I / (C2_C+C1_C);

```
C1_i = C1_C*C1_der_v;
C2_i = Vg_n_i-C1_i;
R1_n_der_v = C1_der_v;
C2_der_v = C1_der_v;

AcceptedSection
G_p_i = C1_i-Vg_n_i+C2_i;
C1_v = R1_n_v;
```

```
InitialSection
PI_0 = 3.14159265358979;
Vg_n_v = 0;
G_p_v = 0;
C1_n_v = 0;
C2_n_v = 0;
```


Simplifications of equations

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

- Manipulations:
  - substitute constants and fixed parameters
  - partial evaluation of expressions:
    0 * expr = 0, expr/expr = 1, etc

- Reduction in number of arithmetic operations:
  typically a factor of 10

Singular systems - high index DAE

DAE: $$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t)$$

with **singular** Jacobi-Matrix $$\begin{vmatrix} \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} & \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \end{vmatrix} = \mathbf{0}$$

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) **Differantiate** the equation subsets and add the resulting equations to the DAE.

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

$$0 = f_1(\dot{x}, x, y_2)$$
$$0 = f_2(y_1)$$
$$0 = f_3(y_1)$$

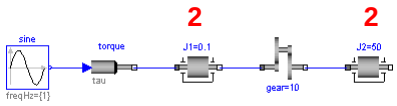
3 equations for the 3 unknown variables x, y_1, y_2.
If the **last two functions are identical**, there is an **infinite number** of **solutions**, else there is a contradiction for calculating y and there is **no solution.**
Such a DAE is called **structurally inconsistent**.
(This property is recognized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

Test (singular systems)

Analyze the following systems:

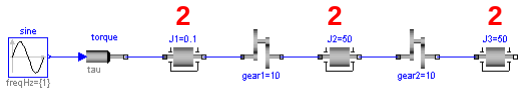
- Write down the **number** of **local states** for each component
- Which **constraint conditions** exist (Write down equations)?
- How many **states** exist in the **total system**?

Number of states



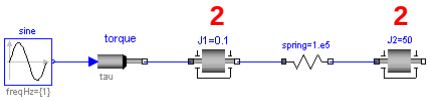

```
J1.phi = i*J2.phi
J1.w   = i*J2.w
```

2

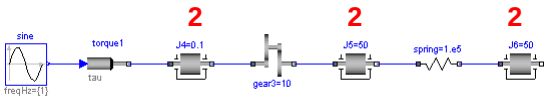



```
J1.phi = i1*J2.phi
J2.phi = i2*J3.phi
J1.w   = i1*J2.w
J2.w   = i2*J3.w
```

2



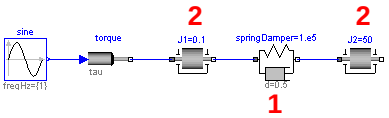
4



4

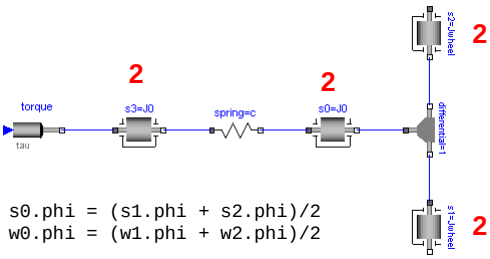
```
J1.phi = i*J2.phi
J1.w   = i*J2.w
```


Number of states



4

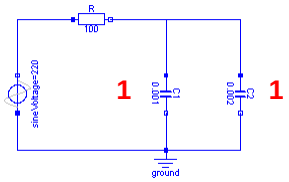
```
springDamper.phi_rel = J2.phi - J1.phi
```



6

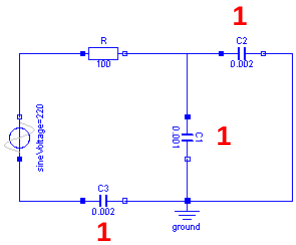
```
s0.phi = (s1.phi + s2.phi)/2
w0.phi = (w1.phi + w2.phi)/2
```

Number of states



C1.v = C2.v

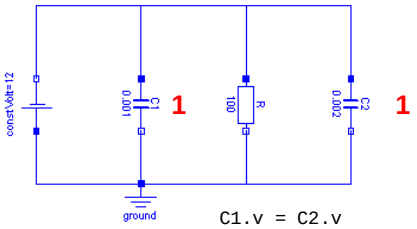
1



C1.v = C2.v

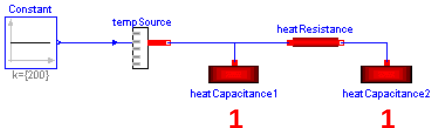
2

Number of states




```
C1.v = C2.v
C1.v = constVolt.v
```

0




```
tempSource.T = heatCapacitance1.T
```

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