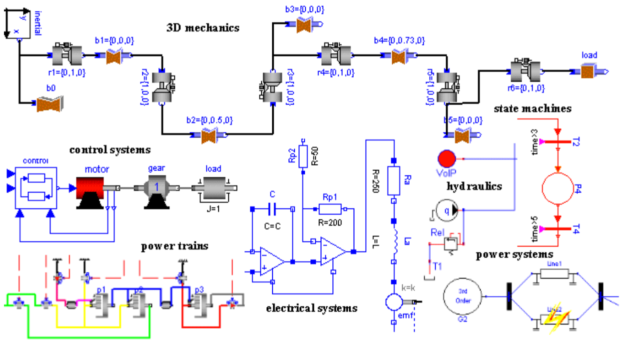
Advanced Modelica Tutorial: Developing Modelica Libraries

Martin Otter, DLR
Hilding Elmqvist, Dynasim

Modelica 2003, November 3-4, Linköping University



Contents

1. Overview

Learn how to use Modelica and Dymola for developing your **own Modelica libraries** to model complex systems.

- New **Modelica 2.1** language constructs
- Advanced **Modelica language constructs**, such as "replaceable" (+ usage in Dymolas graphical user interface).
- Advanced **modeling** issues (e.g., initialization, state selection)
- **New annotations** to allow convenient usage of your libraries (e.g., nicer parameter menus, version handling)
- **Examples** from new libraries (MultiBody, Modelica_Media)
- An **exercise** to try out the learned topics on your notebook
  Note: Install Beta-release of Modelica_Media manually from
  CD: Modelica'2003 Material\readme.txt

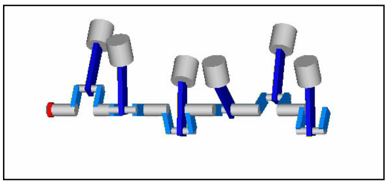
Changes since the last conference, Modelica'2002

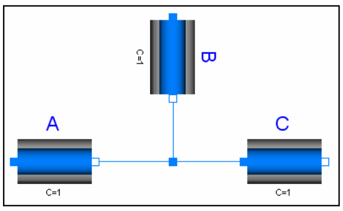
1. Development of **new libraries**

   - 3-dimensional mechanical systems (MultiBody)
   - Thermo-fluid systems (Modelica_Media, _Fluid)

   required:

   - new Modelica language constructs
   - new symbolic transformation algorithms
   - non-standard way of writing equations to achieve "arbitrary connection feature"





2. Modelica, **version 2.1**
(nearly ready, release after the next Modelica design meeting)

- Overdetermined connectors (DAE with more equations as unknowns)
- Arrays and array indices of Enumerations
- Connections into hierarchical connectors (e.g. for buses)
- *break* (while loop); *return* (Modelica function)
- Optional output arguments of Modelica functions.
- *isPresent*(..) (inquire whether actual argument is present).
- *String*(..) (string representation of Boolean, Integer, Real, Enumer.)
- *Integer*(..) (Integer representation of an Enumeration type).
- *semiLinear*(..) (upstream property calculation in flow systems)
- Annotations for version handling and revisions
- Fixing some minor errors in the grammar and semantic specification.
- ...

2. Modelica Libraries

Use as much as possible from available libraries, see
Modelica library page: http://www.Modelica.org/libraries.shtml

Modelica Standard Library

- **SI unit types** (450 types)
- **Control** systems (continuous/sampled)
- **Electric** and **electronic** systems
- **1-dim. translational mechanics**
- **1-dim. rotational mechanics** (clutches, brakes, ...)
- **1-dim. heat transfer**

soon (see conference papers):

- **3-dim. mechanical systems**
- **1-dim. thermo-fluid systems**
(incompressible/compressible, single/multiple substances, one/multiple phases, ...)
- **Fluid media** (1240 gases, IF97 water, refrigerants, ...)

Under development for Modelica Standard Library

- **Digital electrical** systems
- **Advanced Sampled ↔ Continuous**
- **Interpolation** (B-Splines)
- **Linear algebra** (LAPACK)
- **Electrical motors**

Other "public domain" libraries

- **ThermoFluid**
- **Hydraulic** components (HyLibLight)
- **Pneumatic** components (PneuLibLight)
- **Electric power** systems
- **Vehicle dynamics** (Beta)

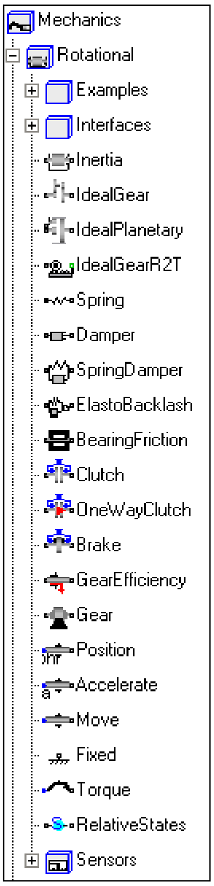
Commercial libraries

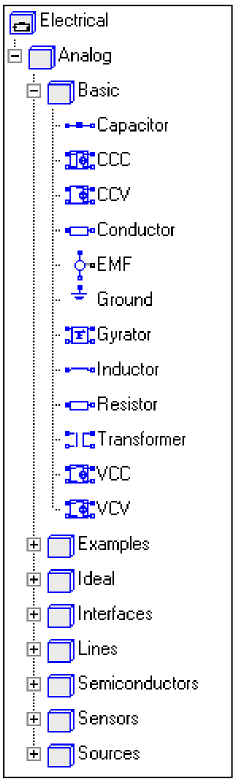
- **Hydraulic** components (HyLib)
- **Pneumatic** components (PneuLib)
- **Vehicle power trains** (PowerTrain)

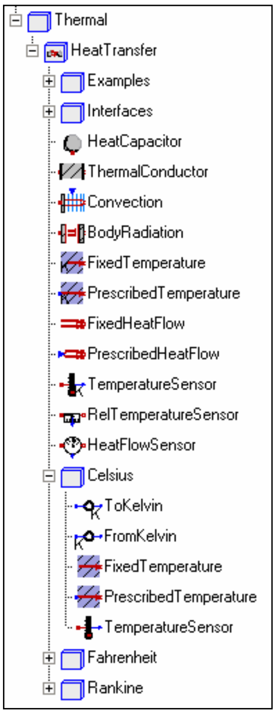
Conversions (commercial)

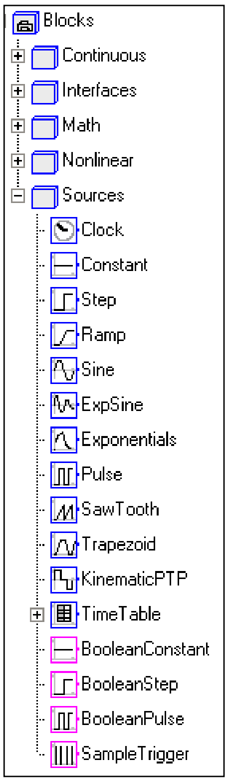
- **Modelica → Simulink** converter (Dymola option)
- **Simulink → Modelica** converter (Simelica)

Modelica Standard Library









MultiBody | **Modelica_Media** | **Sampled** | **Hydraulics (commercial)** | **PowerTrain (commercial)**

3. Packages

Modelica models are structured in hierarchical **libraries** (**package**)

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

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

   ...
end Modelica;
```

The **first** part of a **hierarchical name** is searched from "**lower**" to "**upper**" hierarchies **within** a package:

```
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;
```

identical definitions

A **hierarchical package** can be **stored** in **one** file, e.g.,

file: `Modelica.mo`

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

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

   ...
end Modelica;
```

A **hierarchical** package can be stored on **several files**.
Package hierarchy is **mapped** to corresponding **directory hierarchy**.
Example:

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

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

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

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

```
package Mechanics
end Mechanics;
```

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

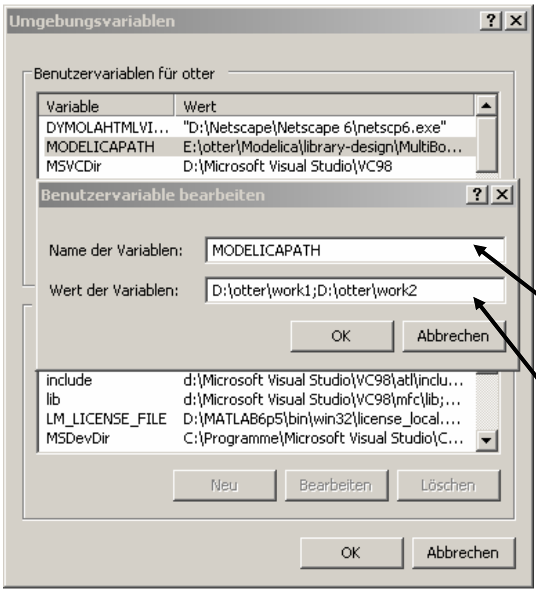
Every package directory must have a file **package.mo** that contains information directly belonging to the package (e.g. annotations of the package)

Even when a package does **not** use **annotations**, a file **package.mo must** be present on which the package name is defined. This file is used to **uniquely** define the **package name**, since, e.g., **directory names** in **Windows** are **not case sensitive** whereas in **Modelica** they are **case sensitive**.

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

```
package Mechanics
end Mechanics;
```

A **hierarchical Modelica name**, such as **A.B.C**, is first searched in the "current directory" (file **A.mo** or **A\package.mo**).
If not found, search in directories defined in "**MODELICAPATH**".
If not found, search in vendor specific directories (e.g. Dymola\Modelica\libraries)



In Windows'NT, '2000, 'XP, environment variables are defined under "System".

In Windows'95, '98, .. in "autoexec.bat"

MODELICAPATH

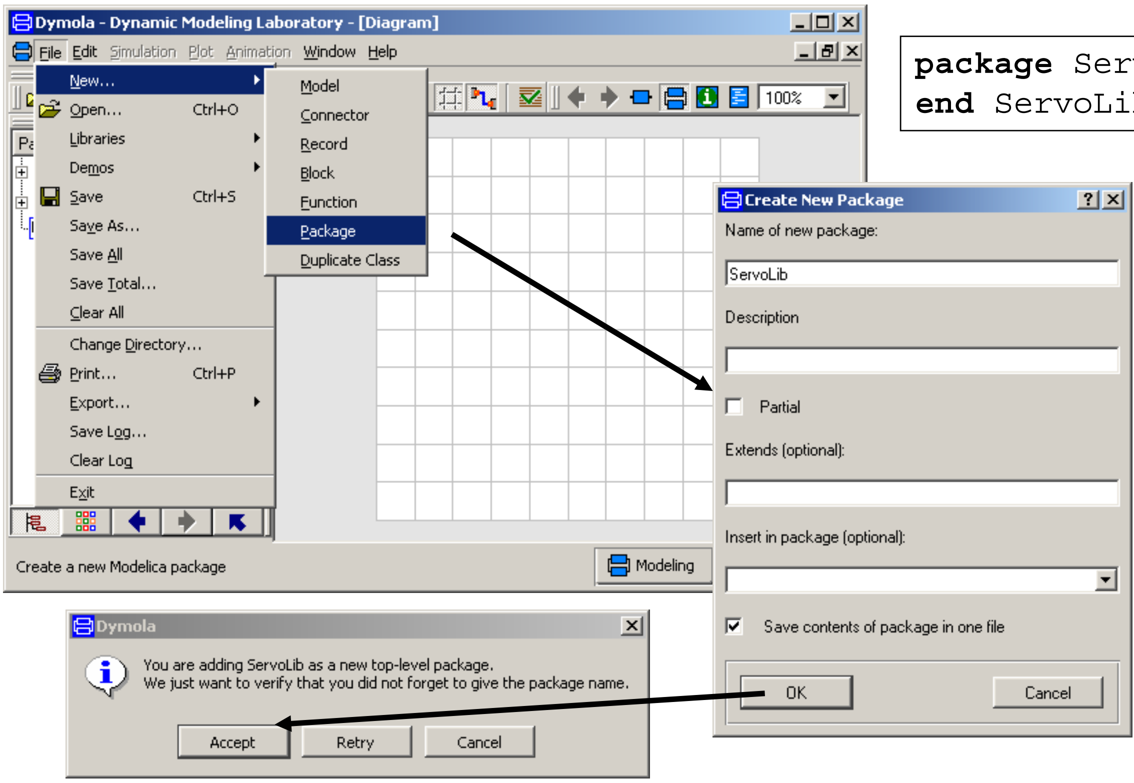
list of directory names, e.g.:
"D:\otter\work1**;** D:\otter\work2"

Define a package.

```
package ServoLib
  model Motor
    ...
  end Motor;

  model Gear
  end Gear;

  model Load
  end Load;
end ServoLib;
```

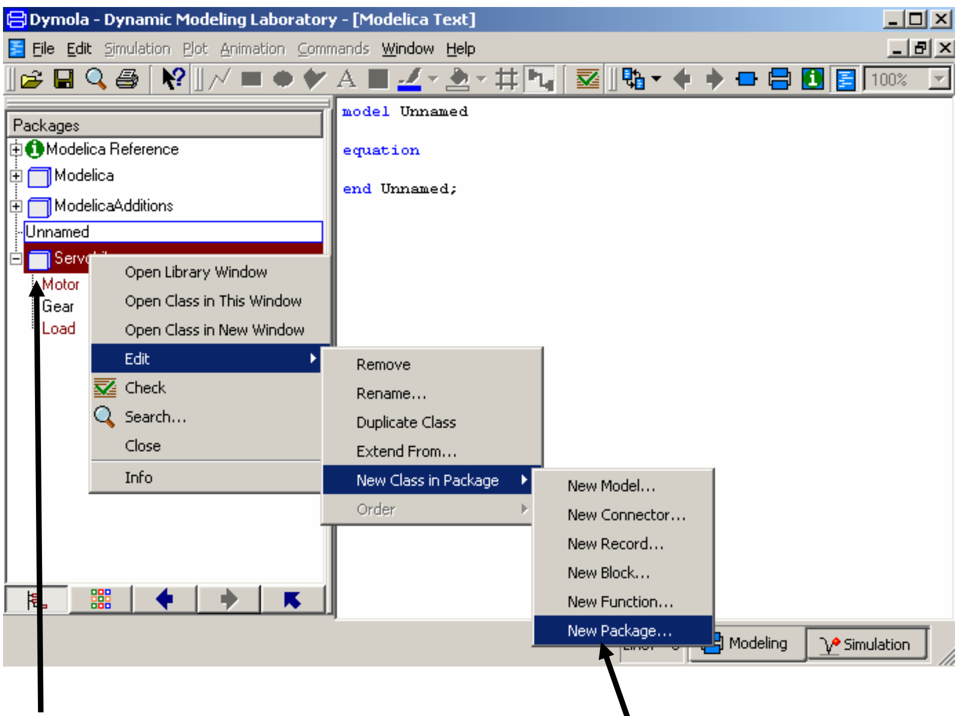
Seems simple, but one needs to **generate**, **copy**, **remove**, **rename**, **resort** etc. models and sublibraries (important for convenience of "daily work").
On the following slides, it is shown how this is performed with Dymola.

Dymola:
Create a new package (i.e., a library)




```
package ServoLib
end ServoLib;
```

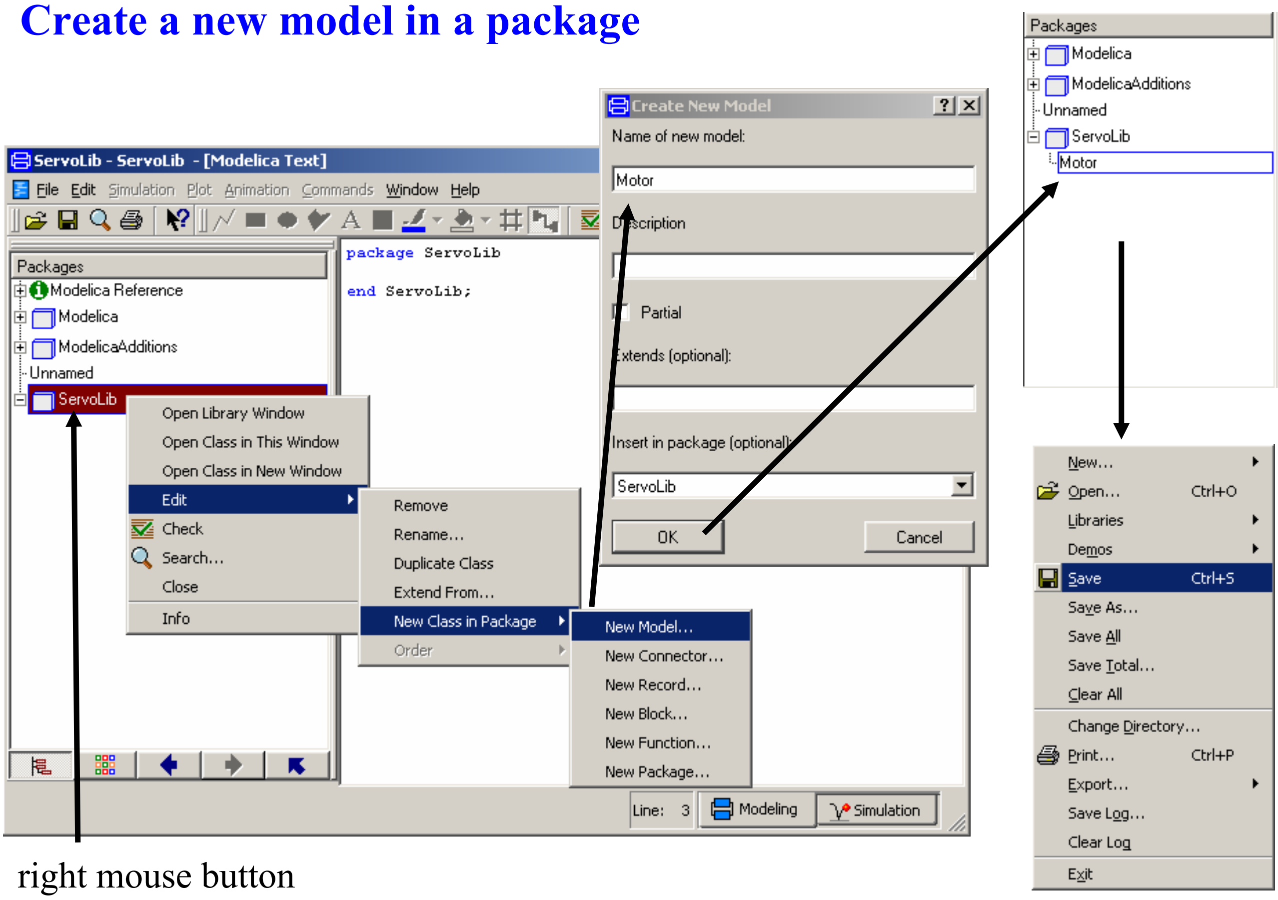

Create a new subpackage



right mouse button

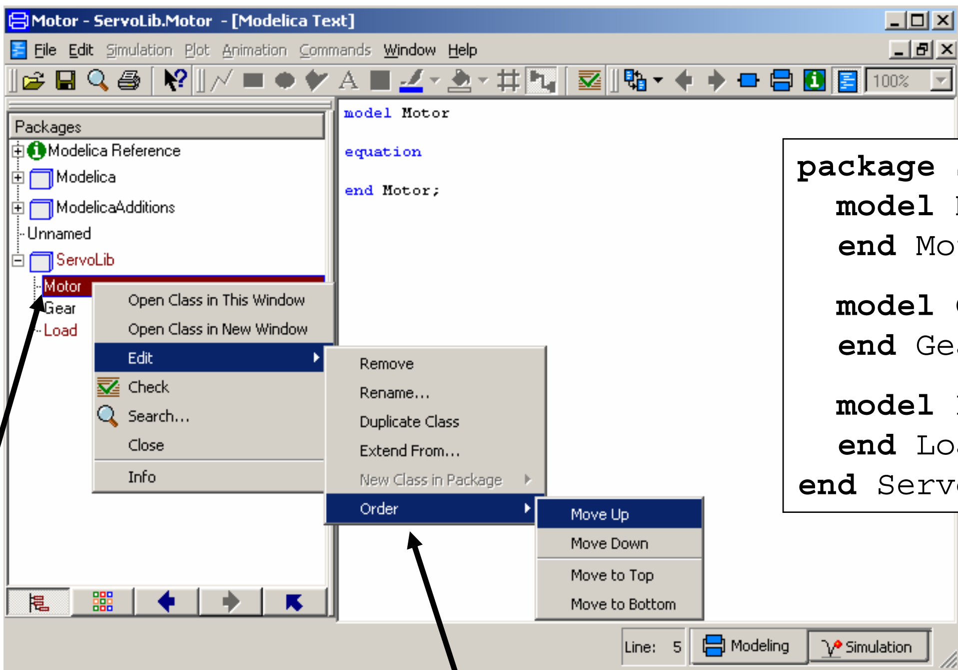
new package in package

Create a new model in a package



right mouse button

Changing a package




```
package ServoLib
  model Motor
  end Motor;

  model Gear
  end Gear;

  model Load
  end Load;
end ServoLib;
```

right mouse button

remove, rename, duplicate model,
extend from another model, **change order**

4. Connectors

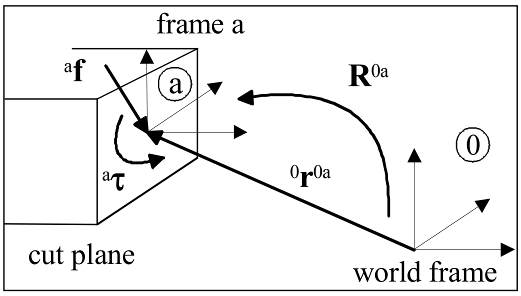
4.1 Elementary Connectors

For your components, use connectors from available libraries, in order that components from these libraries can be utilized (also difficult to design connectors). Most important:

type	potential	flow	location	icon
electrical	electrical potential	current	Modelica.Electrical.Analog.Interfaces.PositivePin	PositivePin NegativePin
translational	position	force	Modelica.Mechanics.Translational.Interfaces.Flange_a	Flange_a Flange_b
rotational	angle	torque	Modelica.Mechanics.Rotational.Interfaces.Flange_a	Flange_a Flange_b
heat transfer	temperature	heat flow rate	Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a	HeatPort_a HeatPort_b
hydraulic	pressure	volume flow rate	HyLibLight.Interfaces.Port_A	Port_A Port_B
pneumatic	pressure	mass flow rate	PneuLibLight.Interfaces.Port_1	Port_1 Port_2

3-dim. mechanics

MultiBody.Interfaces




```
connector Frame
  import SI = Modelica.SIunits;
  SI.Position          r_0[3] "= 0r0a";
  Frames.Orientation R      "= R0a";
  flow SI.Force        f[3]   "= af";
  flow SI.Torque       t[3]   "= aτ";
end Frame;

connector Frame_a = Frame;
connector Frame_b = Frame;
```

Frame_a
Frame_b

1-dim. thermo-fluid flow (one/multiple substances/phases, incomp./compressible)

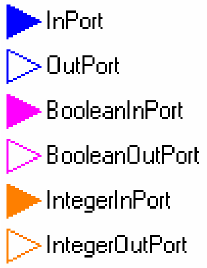
```
connector FluidPort
  replaceable model Medium = PartialMedium;

  Medium.AbsolutePressure      p;
  flow Medium.MassFlowRate     m_dot;

  Medium.SpecificEnthalpy      h;
  flow Medium.EnthalpyFlowRate H_dot;

  Medium.MassFraction          X[Medium.nX];
  flow Medium.MassFlowRate     mX_dot[Medium.nX];
end FluidPort;
```

Modelica_Media.Interfaces

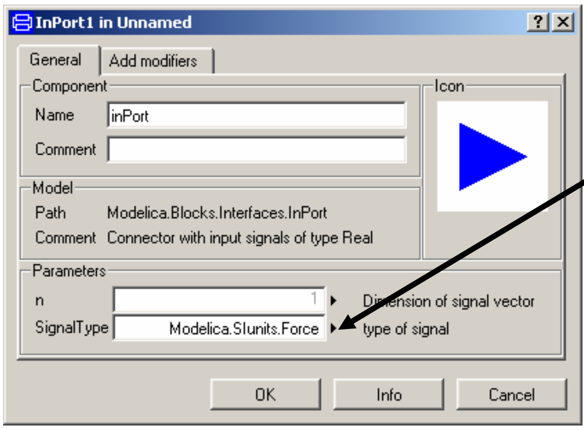
FluidPort_a
FluidPort_b

signal connectors

Modelica.Blocks.Interfaces




```
connector InPort
  parameter Integer n=1;
  replaceable type SignalType = Real;
  input SignalType signal[n];
end InPort;
```



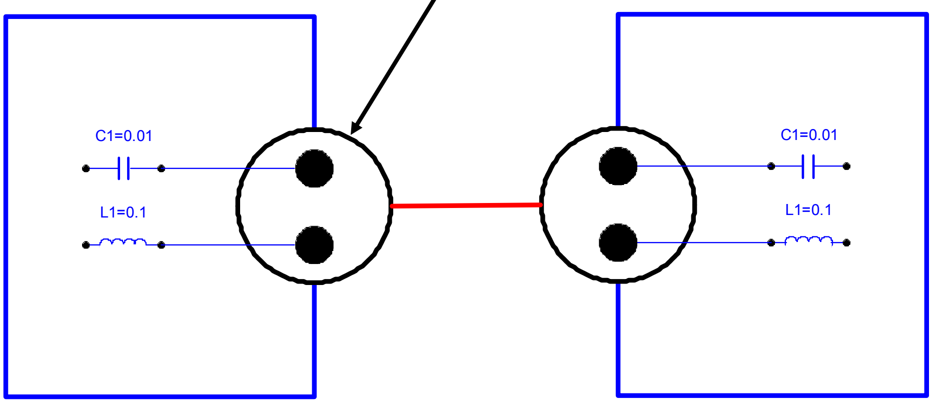
When using InPort/OutPort,
the type may be redefined
(to get units in the plot window)

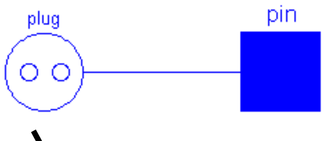
4.2 Hierarchical Connectors

```
connector Pin
  import SI=Modelica.SIunits;
  SI.Voltage v
  flow SI.Current i;
end Pin;

connector Plug
  Pin phase, ground;
end Plug;
```

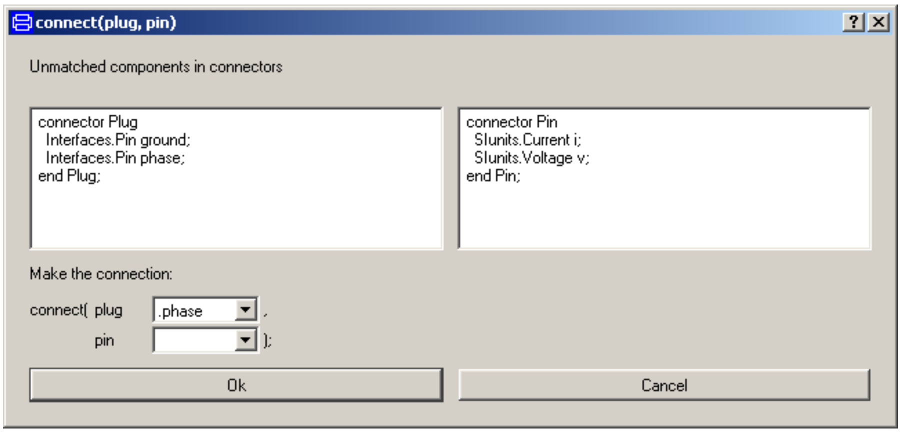
is equivalent to:

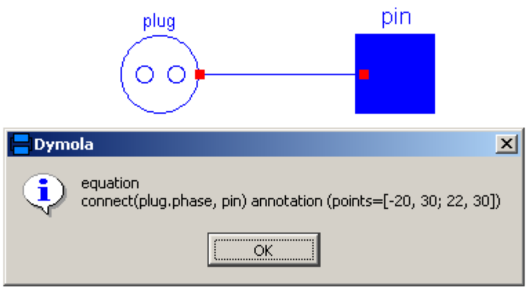
```
connector PlugExpanded
  import SI=Modelica.SIunits;
  SI.Voltage      phase.v
  SI.Voltage      ground.v
  flow SI.Current phase.i;
  flow SI.Current ground.i;
end PlugExpanded;
```





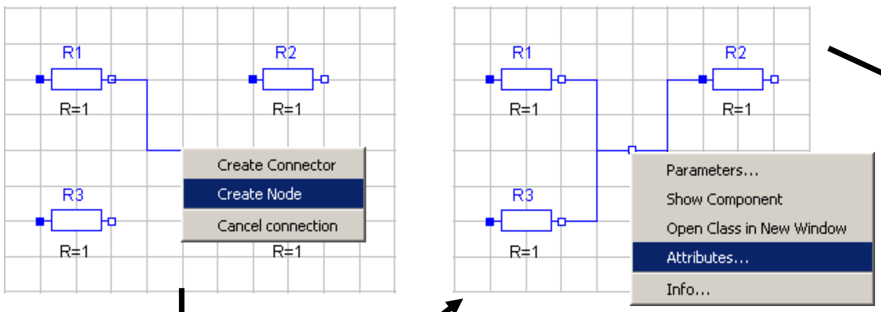
connect **plug** und **pin** connector

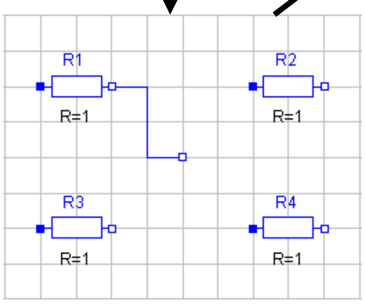




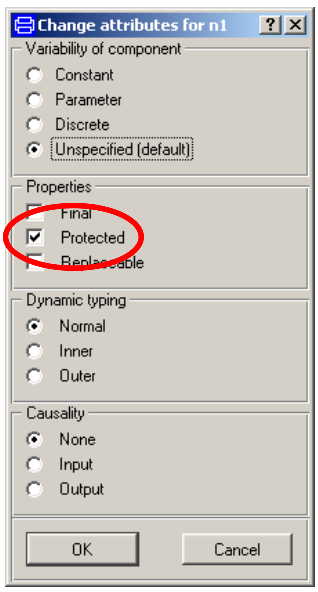
Click on connection line to get information what is connected

Using „**nodes**“ to simplify connections





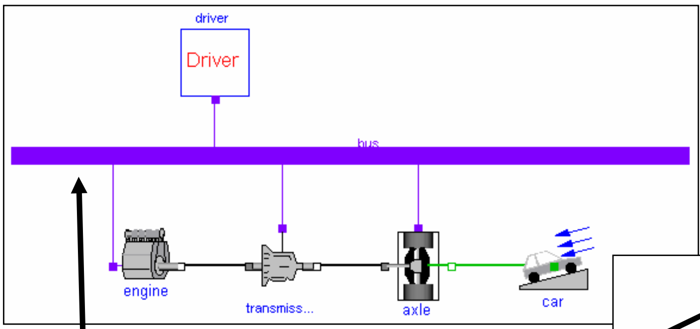

```
model testNodes
  import A = Modelica.Electrical.Analog;
  A.Basic.Resistor R1 a;
  A.Basic.Resistor R2 a;
  A.Basic.Resistor R3 a;
  A.Basic.Resistor R4 a;
  a;
protected
  A.Interfaces.NegativePin n1 a;
equation
  connect(R1.n, n1) a;
  connect(n1, R4.p) a;
  connect(n1, R2.p) a;
  connect(R3.n, n1) a;
end testNodes;
```



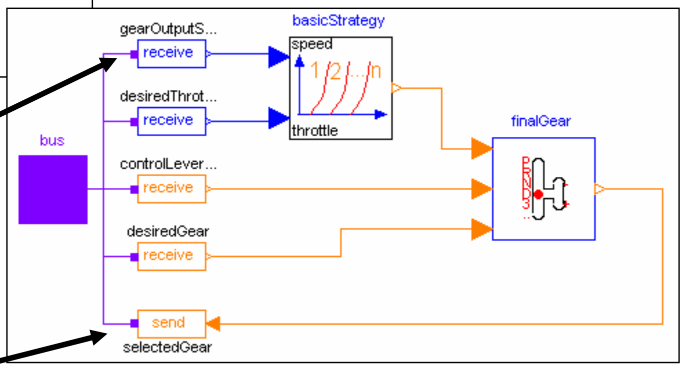
A new connector instance is generated

4.3 Bus Connectors

Signal connections in technical systems become easily quite complicated.
In reality, often field buses are used. This can be mimicked with Modelica.



Example:
PowerTrain library



"Usual" connector, that is
drawn in form of a "bus"

"receive" block to get a variable
"send" block to set a variable

Bus, version 1:

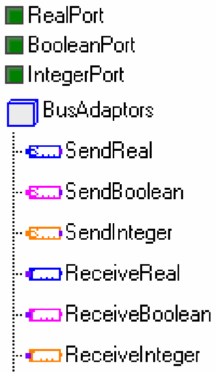
```
connector Bus
  import SI = Modelica.SIunits;
  SI.Velocity        vehicleSpeed;
  SI.AngularVelocity engineSpeed;
  Real               desiredThrottle;
  Integer            desiredGear;
  Boolean            ignition;
   ...
end Bus;
```

Drawback:
It is not possible to connect to single variables, such as vehicleSpeed.
Every model needs to know the complete bus.

It is only possible to connect to connectors.
Therefore, we need single variable connectors.

Bus, version 2:

Modelica.Blocks.Interfaces.



Connector for a single variable

```
connector RealPort
  replaceable type SignalType = Real;
  extends SignalType;
end RealPort;
```

default: `connector RealPort = Real;`

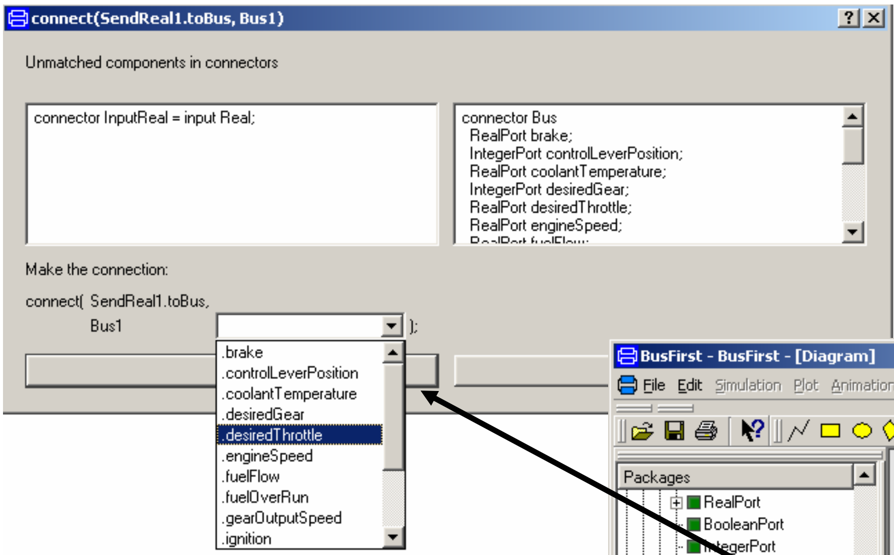
```
connector Bus
  extends Modelica.Blocks.Interfaces.*;
  RealPort    vehicleSpeed;
  RealPort    engineSpeed;
  RealPort    desiredThrottle;
  IntegerPort desiredGear;
  BooleanPort ignition;
   ...
end Bus;
```

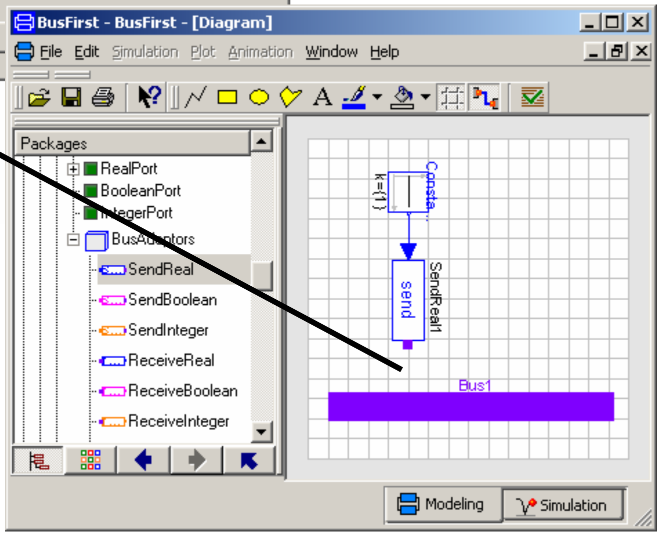
\+ Possible to connect directly to, e.g., vehicleSpeed

\- Variables have no units

Bus, version 3 (recommended):

```
connector Bus
  import SI = Modelica.SIunits;
  extends Modelica.Blocks.Interfaces.*;
  RealPort    vehicleSpeed(redeclare type SignalType
                                  = SI.Velocity);
  RealPort    engineSpeed(redeclare type SignalType
                                  = SI.AngularVelocity);
  RealPort    desiredThrottle;
  IntegerPort desiredGear;
  BooleanPort ignition;
   ...
end Bus;
```



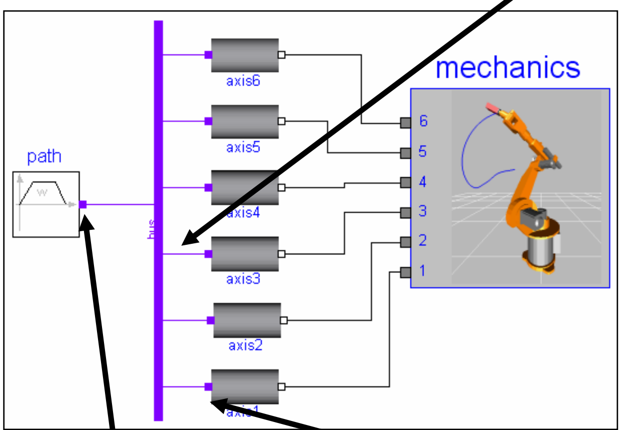


hierarchical connection
directly to "desiredThrottle"

Hierarchical buses

Example: Robot with 6 axes.

```
connect(axis3.bus, bus.axis[3]);
```




```
connector Bus
  parameter Integer naxis=6;
  AxisBus axis[naxis];
end Bus
```

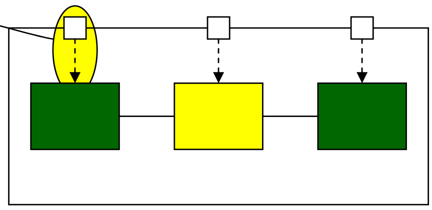
Bus for one axis:

```
connector AxisBus
  BooleanPort motion_ref;
  RealPort    angle_ref(..);
  RealPort    speed_ref(..);
  BooleanPort motorOn;
  ...
end AxisBus;
```

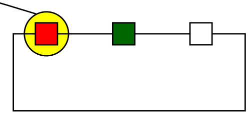
5. Replaceable components

- Redeclare **component** model
- Individually change model
- Keep connections and parameters
- Checking for consistency

```
model C
  replaceable GreenModel  comp1(p1=5);
  replaceable YellowModel comp2;
  replaceable GreenModel  comp3;
  connect(…);
end C;
```

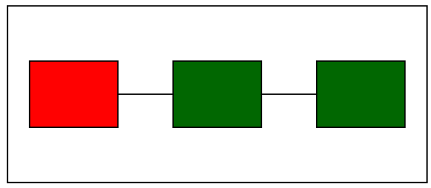


```
model C2 =
  C(redeclare RedModel  comp1,
    redeclare GreenModel comp2);
```



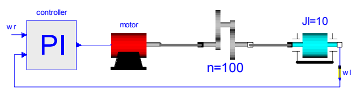
Equivalent to

```
model C
  RedModel   comp1(p1=5);
  GreenModel comp2;
  GreenModel comp3;
  connect(…);
end C;
```



Example - redeclare component model

```
model MotorDrive
  replaceable PI controller;
  Motor       motor;
  Gearbox     gearbox(n=100);
  Shaft       Jl(J=10);
  Tachometer wl;
equation
  connect(controller.out, motor.inp);
  connect(motor.flange  , gearbox.a);
  connect(gearbox.b     , Jl.a);
  connect(Jl.b          , wl.a);
  connect(wl.w          , controller.inp);
end MotorDrive;
```

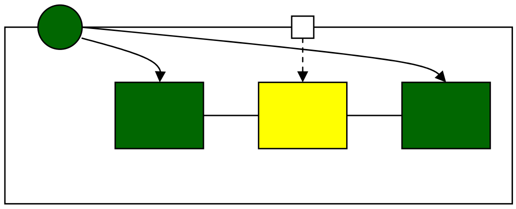



```
model MotorDrive2 = MotorDrive
  (redeclare AutoTuningPI controller);
```

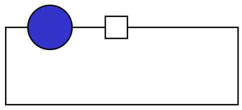
Replaceable models

- Redeclare **model**
- replace the model of many components

```
model C
  replaceable model ColouredClass = GreenClass;
  ColouredClass comp1(p1=5);
  replaceable YellowClass comp2;
  ColouredClass comp3;
  connect(…);
end C;
```

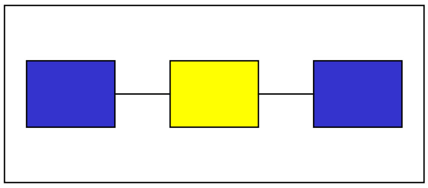


```
class C2 =
  C(redeclare model ColouredClass = BlueClass);
```

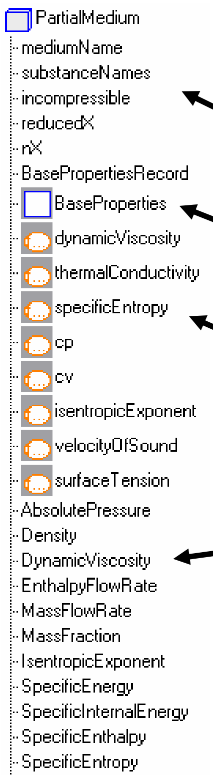


Equivalent to

```
model C
  BlueClass   comp1(p1=5);
  YellowClass comp2;
  BlueClass   comp3;
  connect(…);
end C;
```



Replaceable packages
Example: medium model



A medium "model" consists of several definitions.
Collect everything in one replaceable package

constants (e.g. mediumName)

models (e.g. basic medium properties)

functions (e.g. optional medium properties)

types
e.g. medium specific definitions:

```
type Temperature = SI.Temperature(min=293)
```

Replaceable packages continued

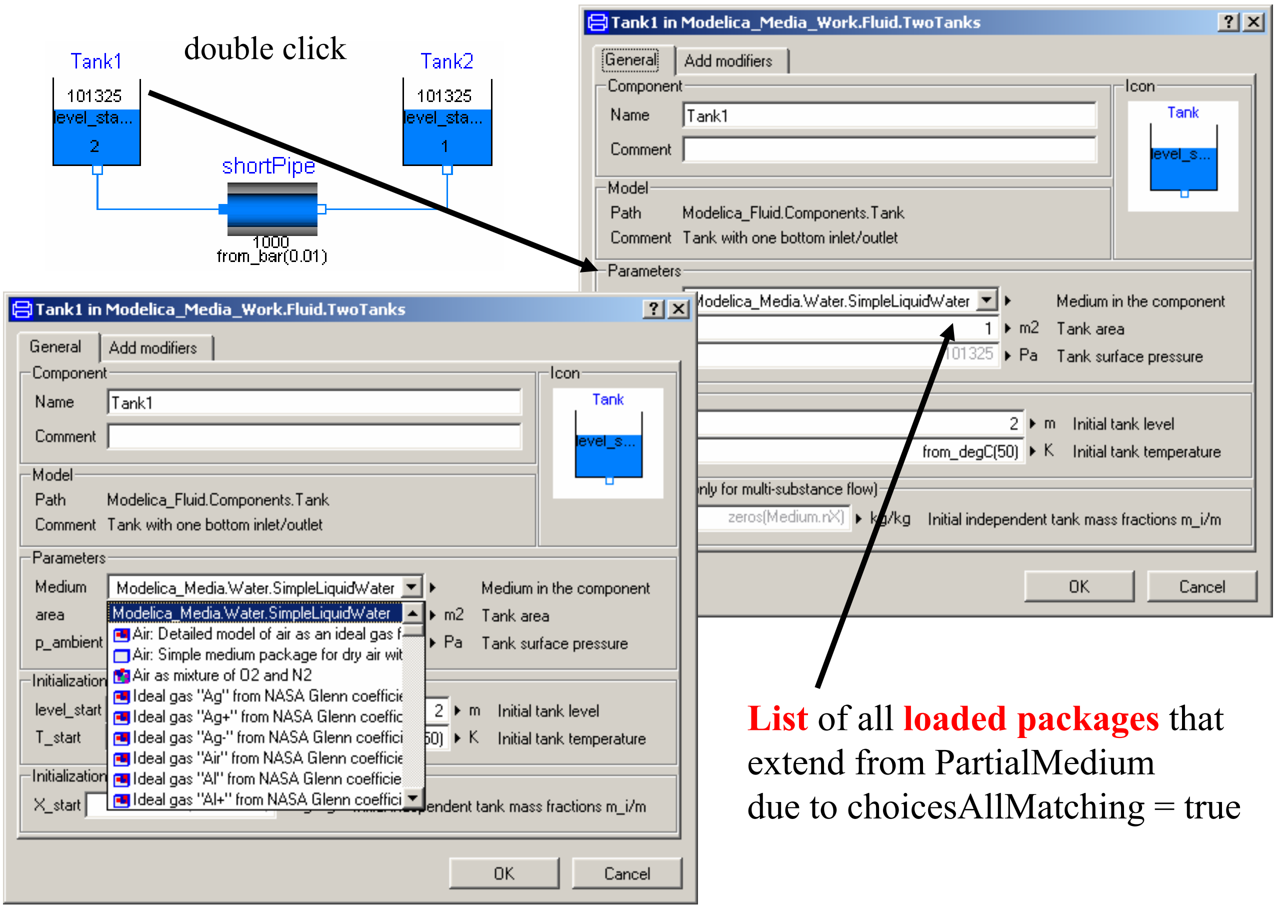
```
package Components
  replaceable package PackageMedium = PartialMedium
                                   annotation(choicesAllMatching=true);

  model Pipe
    replaceable package Medium = PackageMedium extends PartialMedium;
    Medium.BaseProperties medium;    // basic medium model

    ...
  equation
    U = m*medium.u;
    ...
  end Pipe;
  ...
end Components;
```

Default medium

package can be replaced

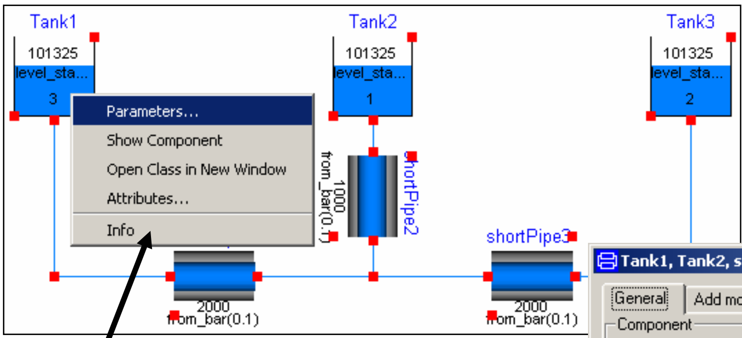
A selection box contains all loaded classes that extend from PartialMedium

Constraining class:
All used packages for Medium, must have the public components of PartialMedium. Within model Tank, only definitions from PartialMedium can be used!!!

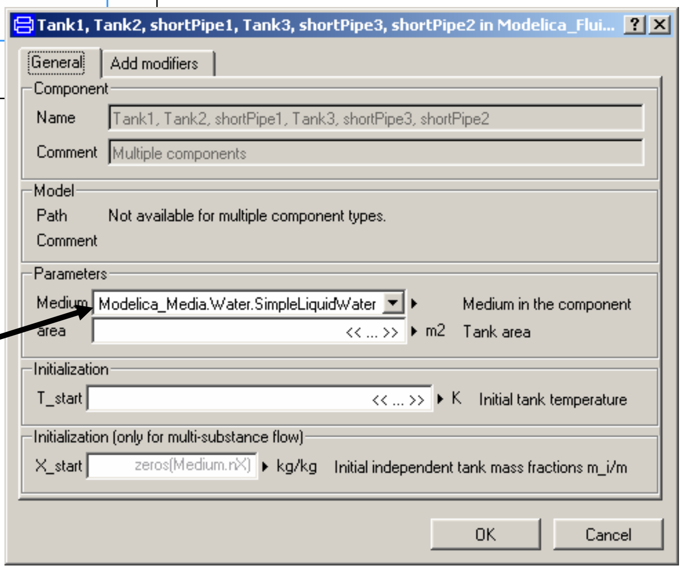


List of all **loaded packages** that extend from PartialMedium due to choicesAllMatching = true

1. Multiple select: Select all components where the medium should be set



2. **Right mouse button**:
   choose "Parameters"



3. The **intersection** of the parameters
   of selected components is shown:

4. **Changing** this **parameter**,
   changes parameters in all selections


```
model ThreeTanks
  replaceable package Medium = Media.Water.SimpleLiquidWater
                      extends PartialMedium
                      annotation (choicesAllMatching=true);

  Components.Tank tank1(
          redeclare package Medium = Medium, ...);
  ...
end ThreeTanks;
```

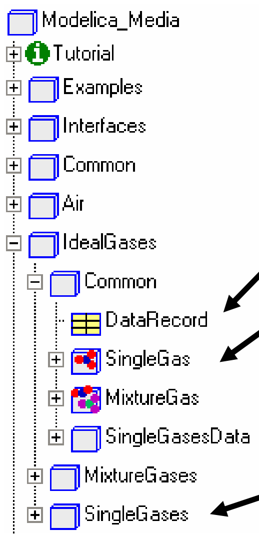
Modification

Changing Medium at this place, will exchange the Medium for all components

6. Datasheet Libraries

Goal: Use records and packages to store **product data** for the **same model**

Examples: Library of ideal gas models (Modelica_Media.IdealGases)
Library of motor models.



record to hold data for a gas medium

medium package using the data record

data for 1241 gases

1241 medium packages of the gases
(these packages can be selected in the
pull down menu of "Medium" parameter)

Variant 1: Using constant record instances (= allowed in packages)

```
record DataRecord "based on NASA Glenn coefficients"
  String               name "Name of ideal gas";
  SI.MolarMass         MM   "Molar mass";
  SI.SpecificEnthalpy  Hf   "Enthalpy of formation at 298.15K";
  SI.SpecificEnthalpy  H0   "H0(298.15K) - H0(0K)";
   ...
end DataRecord;

partial package SingleGas
  constant DataRecord data;
  ... // use data in models and functions
end SingleGas;
```

No value for **constant** given. This is allowed,
provided a value is given in a class that uses SingleGas

Define **constant record instances** for the 1241 gases

```
package SingleGasData
  constant DataRecord Air(
    name = "Air",
    MM   = 0.0289651159,
    Hf   = -4333.833858403446,
    H0   =  298609.6803431054, ...);
  constant DataRecord AL    ( ... );
  constant DataRecord ALBr ( ... );
   ...
end SingleGasData;
```

constant record instance (cannot be changed anymore)

Define media packages for the 1241 gases

```
package SingleGases
  package Air = SingleGas(data = SingleGasData.Air);
  package AL  = SingleGas(data = SingleGasData.AL);
  ...
end SingleGases;
```


Advantage:

Several different components (packages, models, functions, ...) can access the constant data records (by name).

Disadvantage:

It is not possible to modify the data records since they are declared as constant.

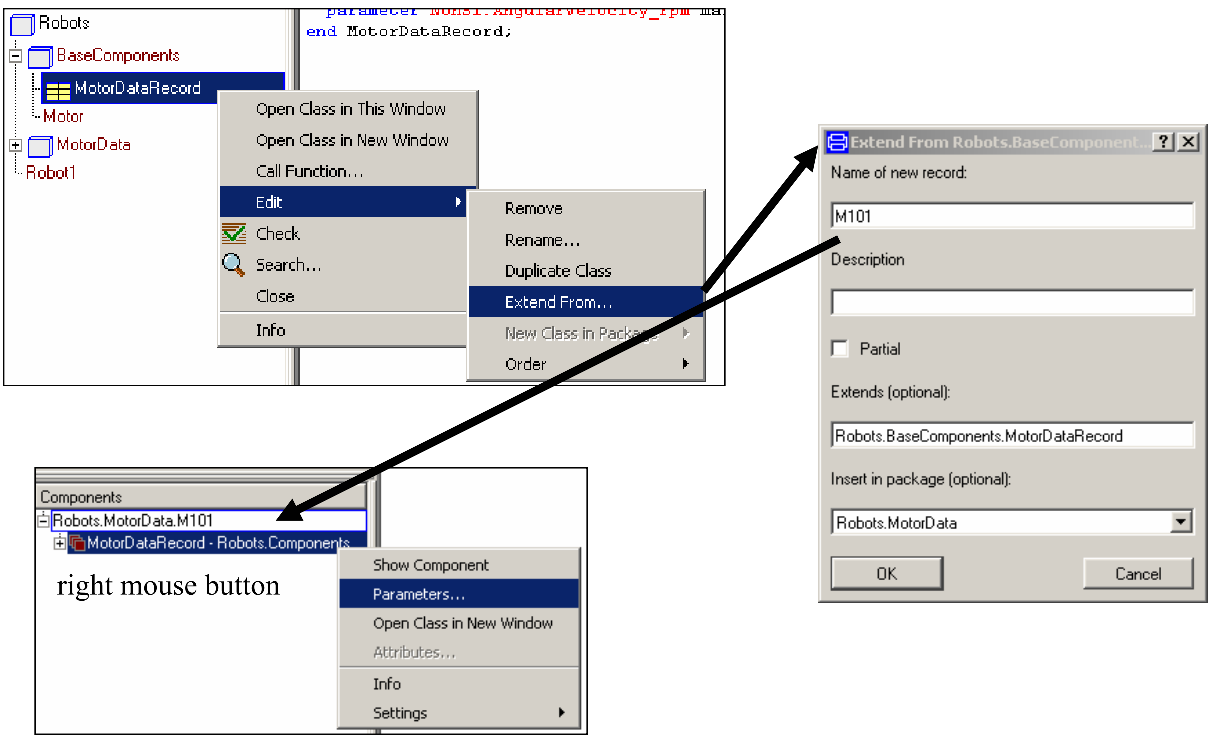
Note:
It is not allowed to have parameter instances in packages.

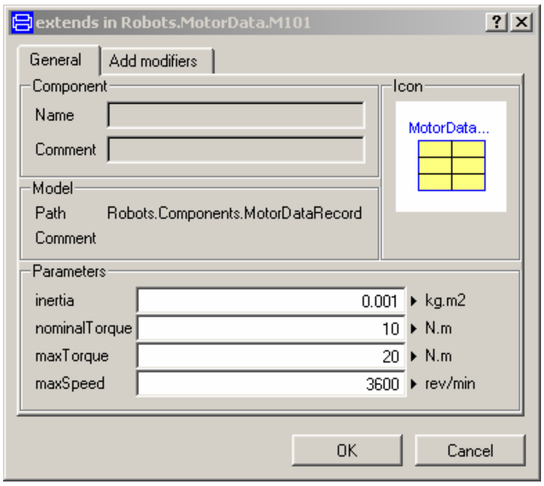
Variant 2: Using parameter record classes

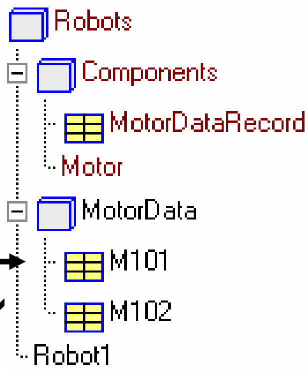
```
package Components
  record MotorDataRecord "Data defining a motor"
    import SI     = Modelica.SIunits;
    import NonSI = Modelica.SIunits.Conversions.NonSIunits;
    parameter SI.Inertia                  inertia;
    parameter SI.Torque                   nominalTorque;
    parameter SI.Torque                   maxTorque;
    parameter NonSI.AngularVelocity_rpm maxSpeed;
     ...
  end MotorDataRecord;

  model Motor "Motor model"
    MotorDataRecord data;
    ...
  equation
    ...
  end Motor
end Components;
```

Record with parameters

Build up a library of motor data, by extending from MotorDataRecord and providing values for the parameters








```
record M101
  extends Components.MotorDataRecord(
    inertia       = 0.001,
    nominalTorque = 10,
    maxTorque     = 20,
    maxSpeed      = 3600);
end M101;
```


Using the motor model and the data of particular motors:

```
model Robot1
  Components.Motor motor1(data=MotorData.M101());
  Components.Motor motor2(data=MotorData.M102(maxTorque=21));
  ...
end Robot1;
```

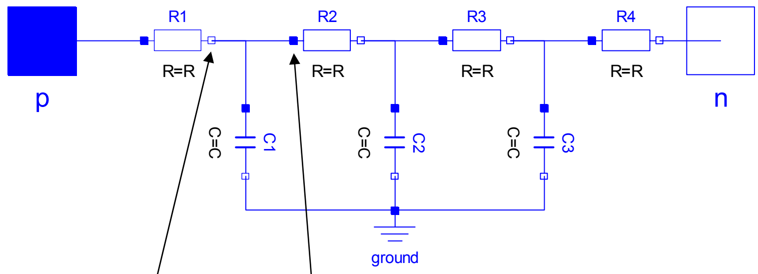
record constructor:

- A function that has the same name as the record
- All record variables are input arguments to this function
- The output argument is an instance of the record
- Allows to modify the original record data when using it

7. Component Arrays

Arrays cannot only be constructed from **Real** variables, but from **every model class**. Example:

electrical line with losses



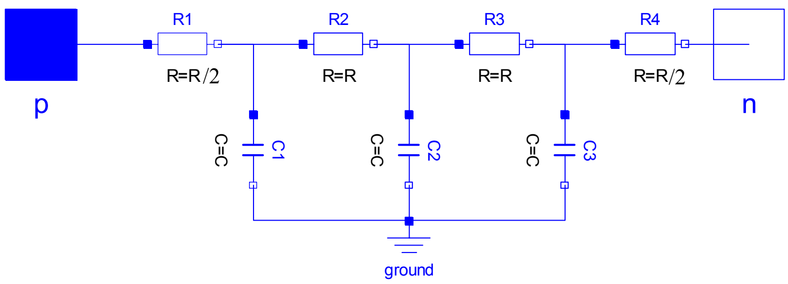

```
   import Modelica.Electrical.Analog.Basic;
   parameter Integer n=4;
   Basic.Resistor Rvec[n]       // 4 Resistances
equation
   for i in 1:n-1 loop
      connect(Rvec[i].n, Rvec[i+1].p);   // connect resistors
   end for;
```


Modification of parameters of component arrays:

```
Basic.Resistor Rvec1[n](each R = 100);      // same value each
Basic.Resistor Rvec2[3](R = {10, 20, 10}); // different values
Basic.Resistor Rvec3[n](R = vector( [10; fill(20,n-2); 10] ));
```

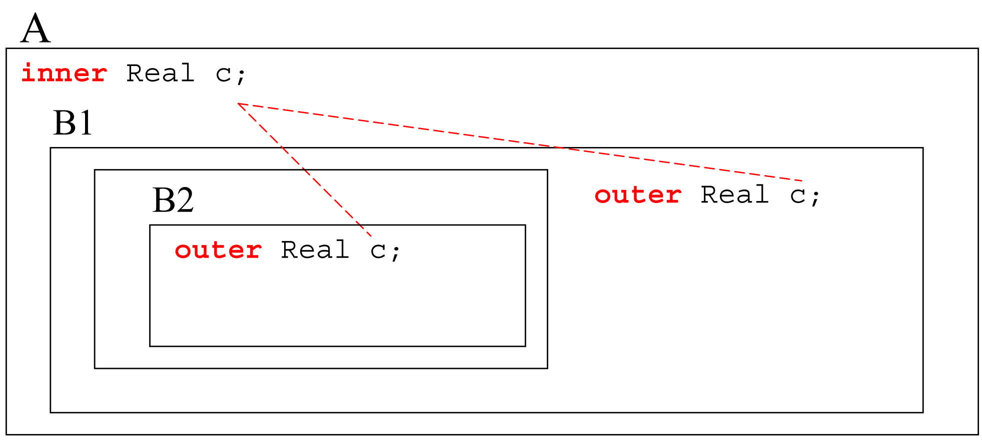
`fill(20,n-2)`	vector with n-2 elements
`[10; fill(20,n-2); 10] ));`	column matrix with n rows
`vector(...)`	vector with n elements

```
model ULine "Lossy RC Line”
  import A=Modelica.Electrical.Analog;
  A.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
  parameter Real Re = r*L/(N + 1);
  A.Basic.Resistor  R[N + 1](R = vector([Re/2; fill(Re,N-1); Re/2]) );
  A.Basic.Capacitor C[N](each C = c*L/N);
  A.Basic.Ground    g;
equation
  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(p, R[1].p);
  connect(R[N + 1].n, n);
end ULine
```



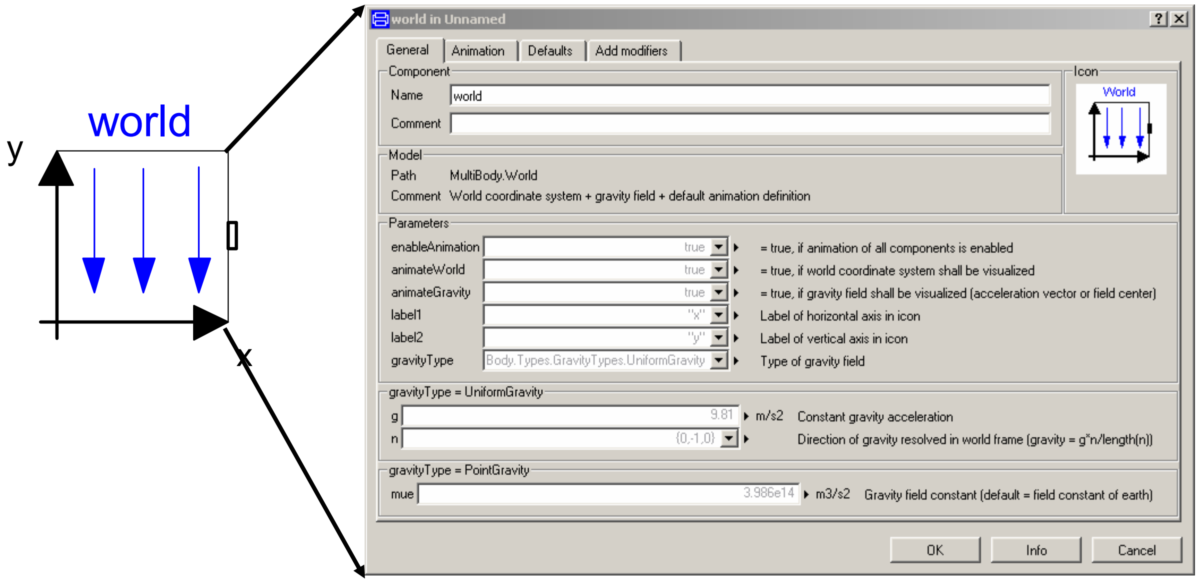
8. Global Variables

A component c with an **outer** prefix in an object B1 or B2 is a **reference** on a component with the same name and the **inner** prefix in an object A. This is only possible if objects B1 and/or B2 are within A.



Previously, some drawbacks when using inner/outer

Example: MultiBody.World



Contains
definition of **gravity field** (inquired by all bodies via inner/outer) and
animation defaults (inquired by all models via inner/outer)

New annotations to allow convenient usage

```
model World
  annotation (
   defaultComponentName  = "world",
   defaultAttributes     = "inner",
   missingInnerMessage   = "
No 'world' component is defined. A default world
component with the default gravity field will be used
(g=9.81 in negative y-axis). If this is not desired,
drag MultiBody.World into the top level of your model.
");
  ...
end World;
```

When dragging model "World" into the diagram layer, a **declaration** is **generated**:

```
inner World world;
```

When the user does **not** drag the **World** object, an "**inner**" object is **missing**:
A default "inner World world" is automatically generated for the
simulation and the "missingInnerMessage" is printed as warning.

9. Initialization

DAE (Differential Algebraic Equation system) derived from Modelica model

(1) $\boxed{\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{y}, t)} \longrightarrow$

State space form

$$\boxed{\begin{aligned}\dot{\mathbf{x}} &= \mathbf{f}_1(\mathbf{x}, t)\\ \mathbf{y} &= \mathbf{f}_2(\mathbf{x}, t)\end{aligned}} \quad (2)$$

x: variables appearing differentiated
y: algebraic variables

For solving (1) or (2) **initial conditions** have to be provided.

Simple: Provide $\mathbf{x}(\mathrm{t}_0)$. Compute $\dot{\mathbf{x}}(t_0)$, $\mathbf{y}(t_0)$
by solving the non-linear system of equations (1)

General: Provide dim(**x**) additional equations **g**(...) and solve

$$\boxed{\begin{aligned}\mathbf{0} &= \mathbf{f}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), t_0)\\ \mathbf{0} &= \mathbf{g}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), t_0)\end{aligned}} \quad \text{for} \quad \dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0)$$

Example: **steady-state** initialization

$$\boxed{\begin{aligned}\mathbf{0} &= \mathbf{f}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), t_0)\\ \mathbf{0} &= \dot{\mathbf{x}}(t_0)\end{aligned}}$$

This means that for the initialization the additional equations

$$\mathbf{0} = \dot{\mathbf{x}}(t_0)$$

have to be added.

Note, that not all components can be initialized in stead-state, e.g., signal sources or integrators such as in a PI controller.

In principle **two different codes** need to be generated:

- **one code** for **initialization**
  (equations **f**(...) and **g**(...); all variables are unknown)

- **one code** for **simulation**
  (equations **f**(...); variables **x**(t) are known, $\dot{\mathbf{x}}(t)$, $\mathbf{y}(t)$ are computed)

Advantage:
In both cases the **symbolic** engine is used to determine a robust and efficient solution of the algebraic equations with good diagnostics in problematic cases (e.g., if structurally redundant initial conditions are given).

Initial conditions **g**(..) can be defined in Modelica in two ways.

1. By **additional attributes**

start	initial value of variable at t_0 (default = 0)
fixed	= **true** : v(start=v0, fixed=true) results in the additional initial equation: "v = v0" (**default** for **parameter**) = **false**: "start" is a **guess value** that may be changed during initialization (= default for non-paramter variables)

```
SI.Velocity v(start = 1, fixed = true)
```

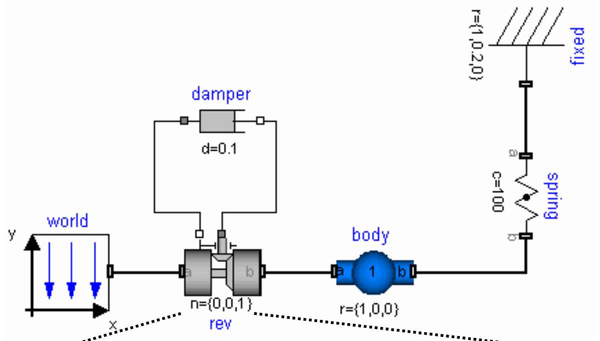
parameter expression required

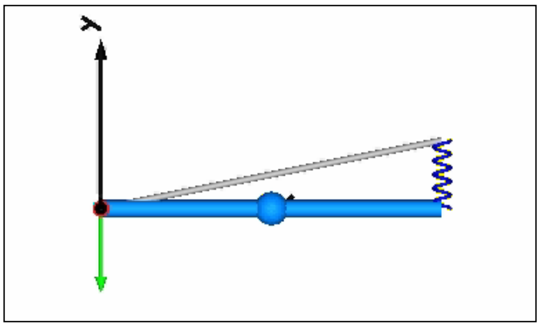
2. By equations in the "**initial equation**" or "initial algorithm" section. The equations/assignments in these sections are only used during initialization

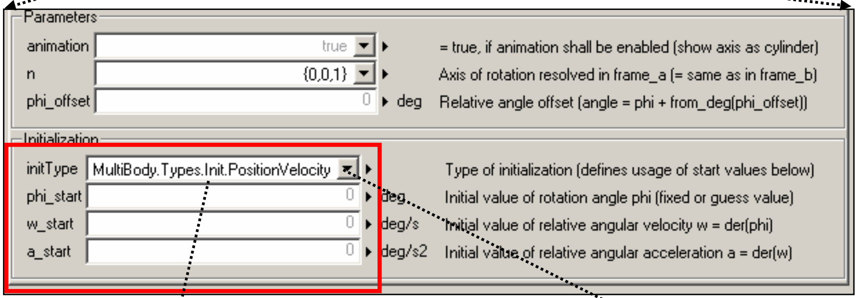
```
initial equation
  v = 1;
equation
  ...
```

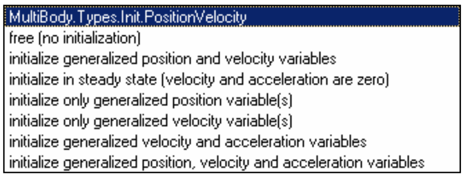
any equation (no restrictions)

Example: MultiBody library



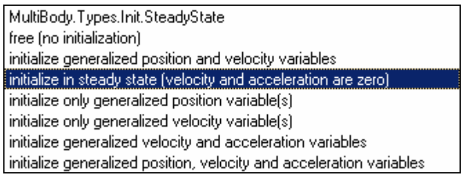






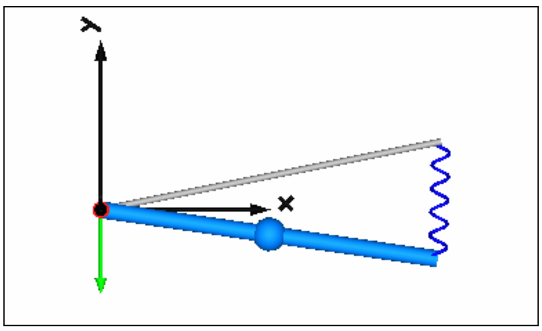
start from given start angle phi_start = 0
and start speed w_start = 0

initialize in **steady-state**



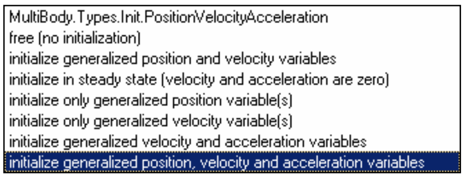
w = 0, der(w) = 0,
phi_start is used as guess value (for non-linear equation)

```
MultiBody.Joints.ActuatedRevolute rev(n={0,0,1},
                 initType = MultiBody.Types.Init.SteadyState)
```

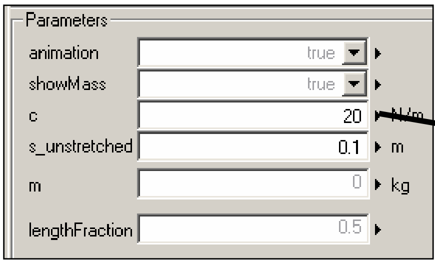


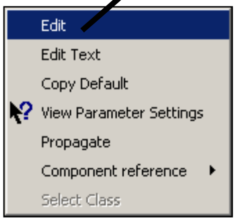
determine **spring constant**, such that
phi = 0, w = 0, der(w) = 0 at initial time

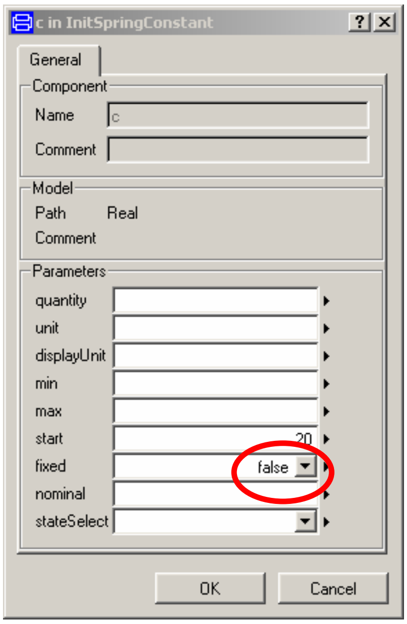
joint

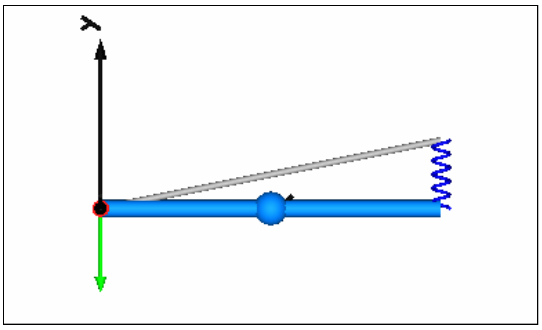


spring









spring(c(start=20, fixed=false))

c = 49.05
(plot window/Advanced/Time = 0)

Implementation:

Every joint, that has potential states, is implemented in the following way:

1. Define choices for initialization

Should be performed with **enumerations** (Modelica 2.0).
Emulate enumerations with a package of constants if not yet supported by tool (e.g. not in Dymola):

```
package Init
  constant Integer Free                          = 1;
  constant Integer PositionVelocity              = 2;
  constant Integer SteadyState                   = 3;
  constant Integer Position                      = 4;
  constant Integer Velocity                      = 5;
  constant Integer VelocityAcceleration          = 6;
  constant Integer PositionVelocityAcceleration = 7;

  type Temp
    extends Integer;
    annotation (choices(
      choice=MultiBody.Types.Init.Free "free (no initialization)",
      choice=MultiBody.Types.Init.PositionVelocity
        "initialize generalized position and velocity variables",
      ...));
  end Temp;
end Init;
```

scroll down menu

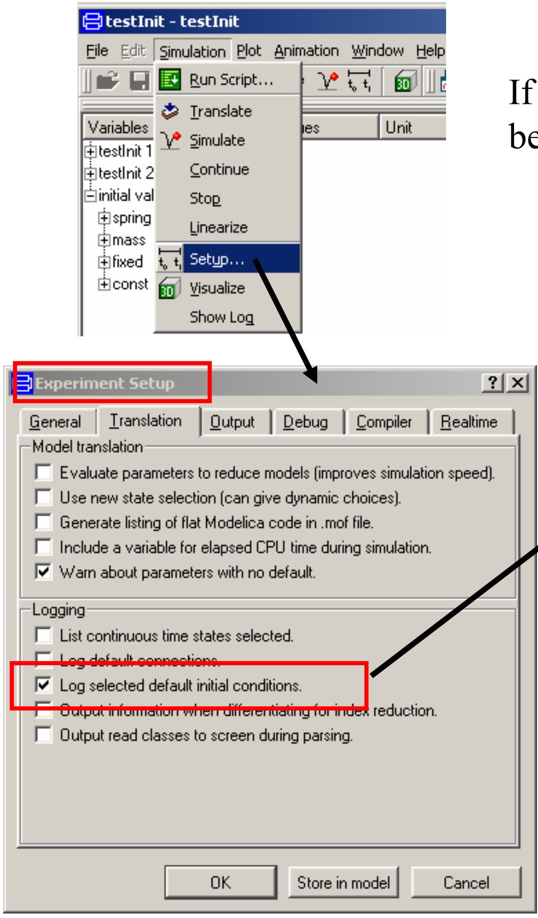
2. Declare variables to define initialization

```
parameter Types.Init.Temp initType = Types.Init.Free;
parameter SI.Angle                  phi_start = 0;
parameter SI.AngularVelocity        w_start   = 0;
parameter SI.AngularAcceleration a_start   = 0;

SI.Angle                  phi(start = phi_start);
SI.AngularVelocity        w;
SI.AngularAcceleration a;
```

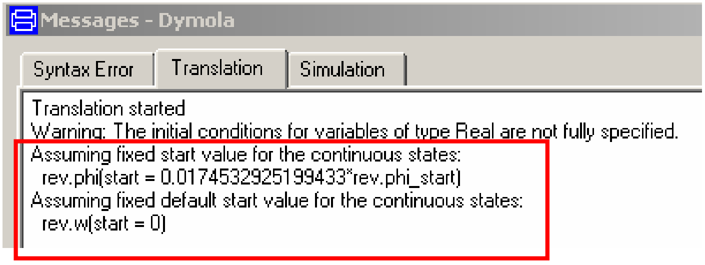
use **phi_start always** as **guess value**
(if non-linear equations occur during initialization, the configuration of the multi-body system is defined with the guess values of the generalized joint coordinates).

3. Definite initialization equations

```
initial equation
  if initType == Types.Init.PositionVelocity then
    phi = phi_start;
    w   = w_start;
  elseif initType == Types.Init.SteadyState then
    w = 0;
    a = 0;
  elseif initType == Types.Init.Position then
    phi = phi_start;
  elseif initType == Types.Init.Velocity then
    w = w_start;
  elseif initType == Types.Init.VelocityAcceleration then
    w = w_start;
    a = a_start;
  elseif initType == Types.Init.PositionVelocityAcceleration then
    phi = phi_start;
    w   = w_start;
    a   = a_start;
  end if;
equation
  w = der(phi);
  a = der(w);
   ...
```



If it is unknown how many initial conditions shall be set, **log** the selected default **initial conditions**:



If **no initial values** are given, Dymola uses as initial values appropriate states with their start values. The number of „default start values" defines how many initial conditions have still to be **fixed**.

10. Version Handling

Annotations introduced in Modelica 2.1:
A top-level package or model can specify the **version** of top-level classes it **uses**, its **own version** number, and if possible how to **convert** from previous versions.

```
package MultiBody
  annotation(version="0.98",
             conversion(noneFromVersion = "0.95 Beta 1",
                        from(version="0.96",
                             script="convertTo_0.98.mos")));
     ...
end MultiBody;

model Robot
  annotation(version="1.0", uses(MultiBody(version="0.98")));
     ...
end Robot;

model B
  annotation(uses(MultiBody(version="0.95 Beta 1")));
     ...
end B;
```

Conversion script is tool dependent (not standardized)

For example in Dymola:

```
convertClear();
convertClass     ("oldClass", "newClass");
convertElement   ("oldClass", "oldElement", "newElement");
convertModifiers("oldClass", oldParameterBindings,
                             newParameterBindings);
```

Examples
(for details, see Dymola\Documentation\Dymola5Manual.pdf, page 261):

```
convertClass("Modelica.Rotational1D",
             "Modelica.Rotational")

convertModifiers("MultiBody.Joints.Cylindrical",
   {"startValuesFixed=false"},
   {"initType=if %startValuesFixed% then
                  MultiBody.Types.Init.PositionVelocity
             else MultiBody.Types.Init.Free"});
```