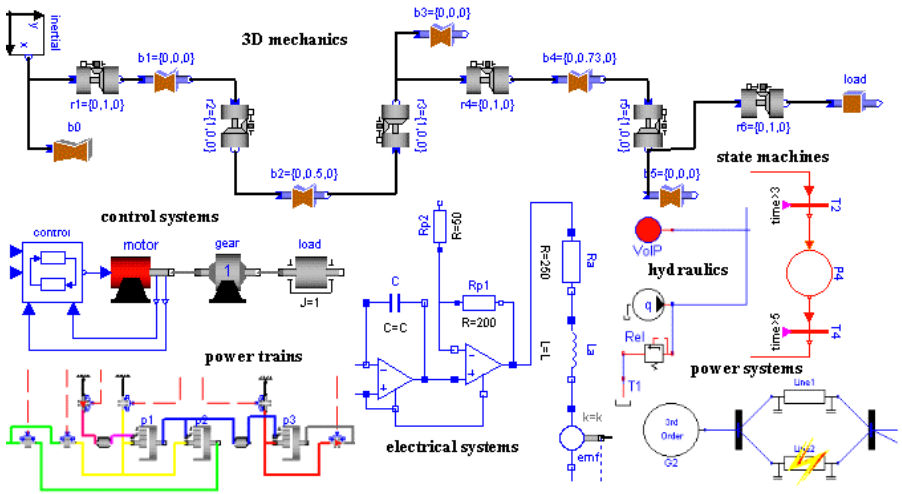
Advanced Modelica Tutorial

Hilding Elmqvist, Dynasim
Hans Olsson, Dynasim
Martin Otter, DLR

Modelica 2002, March 18-19, DLR Oberpfaffenhofen



Contents

- Initialization
- Enumerations
- Control of state selection
- Improved Component Arrays
- Reduction operators
- Replaceable components, Replaceable packages
- Record constructor and datasheet libraries
- Extended function call + scripting
- External Utility functions; ModelicaError etc.
- External Objects

Initialization

Flexible specification of initial conditions as well as the correct solution of difficult, non-standard initialization problems occurring in industrial applications, for example:

- Stationary initialization around a constant reference velocity of an aircraft.

- Stationary initialization around periodic solutions,
  needed in power systems or in detailed engine models.

- Stationary initialization of continuous systems controlled by sampled data systems (the states of the discrete controllers are computed in such a way that the overall system is in a steady state when simulation starts).

- Initialization of discontinuous or variable structure systems, e.g., systems containing friction or backlash.

DAE formulation

$$F(t, x, \frac{dx}{dt}, w) = 0$$

After index reduction

$$G(t, x_1, \frac{dx_1}{dt}, x_2, der\_x_2, w_1, w_2, der\_w_2) = 0$$

ODE formulation (locally)

$$(\frac{dx_1}{dt}, x_2, der\_x_2, w_1, w_2, der\_w_2) = f(t, x_1)$$

Dynamic state (x_1) selection

Initial conditions

At initial time, solve index-one DAE equations
for all variables except t_0

$$G(t_0, x_1(t_0), \frac{dx_1}{dt}(t_0), x_2(t_0), der\_x_2(t_0), w_1(t_0), w_2(t_0), der\_w_2(t_0)) = 0$$

$\dim(x_1)$ more equations needed

The start and fixed attributes

- `Real v(start=expr, fixed=true);`
  // $v(t_0) = expr(t_0)$;

- Steady-state initialization:
  `Real der_v(start=0, fixed=true) = der(v);`
  // $dv/dt(t_0) = 0$;

- General initialization equation (residue):
  `Real ic(start=0, fixed=true) = expr1- expr2;`
  // $expr_1(t_0) = expr_2(t_0)$;

NEW: initial equation

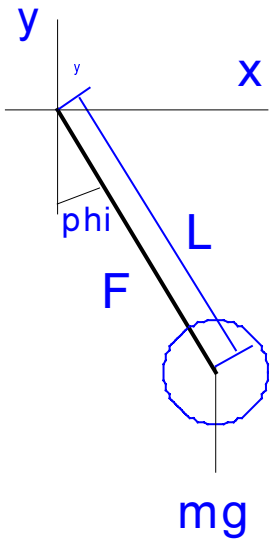
- **initial equation**
  ```
  v = expr;
  ```
  // $v(t_0) = expr(t_0)$;

- Steady-state initialization:
  initial equation
  der(v) = expr;
  // $dv/dt(t_0) = expr(t_0)$;

- General initialization equation (residue):
  initial equation
  ```
  expr1 = expr2;
  ```
  // $expr_1(t_0) = expr_2(t_0)$;
  pre(x) = x; // Steady-state for discrete variable

Pendulum example

- ```
  model Pendulum
    parameter Real g = 9.81;
    parameter Real m = 1;
    parameter Real L = 1;
    Real phi, w;
  equation
    der(phi) = w;
    m*der(w) = -(m*g/L)*sin(phi);
  end Pendulum;
  ```



- Warning: The initial conditions for variables of type Real are not fully specified. There are 2 scalar conditions missing. Assuming default start values for:
  phi(start = 0)
  w(start = 0)
  ```

Pendulum initialization (1)

- Initialize angle and angular velocity

```
model PendulumInitialization1
  extends Pendulum(
    phi(start=1, fixed=true),
    w  (start=0, fixed=true)
  );
end PendulumInitialization1;
```

- Equivalent:

```
model PendulumInitialization2
  extends Pendulum;
initial equation
    phi=1;
    w=0;
end PendulumInitialization2;
```

Pendulum initialization (2)

- Alternative - cartesian initial values:

```
model PendulumInitialization3
  extends Pendulum;
  Real x(start=L); // approximate value
  Real y;  // (start = 0, fixed = true);
equation
  x =  L*sin(phi);
  y = -L*cos(phi);
initial equation
  y=0;
  der(y)=0;
end PendulumInitialization3;
```

differentiated

- Nonlinear system of equation `0 = -L*cos(phi)` solved for phi
- Linear equation for w: **der**`(phi) = `**der**`(y)/(L*sin(phi))`

Pendulum initialization (3)

- Alternative - cartesian initial values:

```
model PendulumInitialization4
  extends Pendulum;
  Real x, y;
equation
  x = L*sin(phi);  y = -L*cos(phi);
initial equation
  der(x)=1;
  der(y)=2;
end PendulumInitialization4;
```

- Nonlinear system of equations

```
0 = der(x)-L*der(phi)*cos(phi);     0 = der(y)-L*der(phi)*sin(phi)
```

solved for phi and **der**(phi)

Pendulum initialization (4)

- Alternative - cartesian initial values:

```
model PendulumInitialization5
  extends Pendulum(L(fixed=false));
  Real x, y;
equation
  x = L*sin(phi);  y = -L*cos(phi);
initial equation
  x=1;  y=2;  w=0;
end PendulumInitialization5;
```

- Nonlinear system of equations

```
x = L*sin(phi);  y = -L*cos(phi);
```

solved for phi and L

Initialization of discrete models

- For discrete variables declarations

```
Real    x(start = 10,    fixed = true);
Boolean b(start = false, fixed = true);
Integer i(start = 1,     fixed = true);
```

imply the additional initialization equations

```
pre(x) = 10;
pre(b) = false;
pre(i) = 1;
```

- Needed for discrete systems or difference equations (when **pre**(v) appear)
- The value of variables assigned in when clauses before the first sampling

Initialization of difference equations

- Steady state initialization:

```
Real x;
equation
when {initial(), sample(1,10)} then
   x = a*pre(x) + b*u;
end when;
initial equation
   pre(x) = x;
```

Difference equation also valid initially

- Assume $u(t_0)$ given
- Solution:

```
pre(x):=b*u(t0)/(1-a);
x := pre(x);
```

Continuous and discrete initialization

```
Real             x;
discrete  Real xd;
discrete  Real u;
parameter Real k=10, T=1;
parameter Real Ts = 0.01 "Sample time";
input     Real xref  "reference input";
equation
// Plant model
der(x) = -x + u;
// Discrete PID controller
when {initial(), sample(0, Ts)} then
  xd = pre(xd) + Ts/T*(x - xref);
   u = k*(xd + x - xref);
end when;
initial equation
  der(x)  = 0;
  pre(xd) = xd;
```

Continuous and discrete initialization (2)

- The initialization problem becomes

```
der(x) := 0
// Linear system of equations in the
// unknowns: xd, pre(xd), u, x
pre(xd) = xd
xd      = pre(xd) + Ts/T*(x - xref)
u       = k*(xd + xref - x)
der(x)  = -x + u;
```

- Solving the system of equations gives

```
der(x)  := 0
x       := xref
u       := xref
xd      := xref/k
pre(xd) := xd
```

Enumerations

- New user defined base data types
- Ordered set of values with unique identifiers

- `type TextStyle = enumeration(Bold, Italic, UnderLine);`
- `TextStyle t1 = TextStyle.Bold;`
- `TextStyle t2 = t1;`

- Operations: <, <=, >, >=, ==, <>

- Future - arrays over enumerations:
- `Boolean textStyles[TextStyle] = {false, false, true};`
- `textStyles[TextStyle.Bold] := true;`

State Variable Selection

- DAE:
  $\mathbf{0} = \mathbf{f}(d\mathbf{x}/dt, \mathbf{x}, \mathbf{y}, t)$

- After index reduction conceptually transformed to ODE (locally):
  $d\mathbf{x}_s/dt = \mathbf{f}_1(\mathbf{x}_s, t)$
  $\mathbf{x}_n = \mathbf{f}_2(\mathbf{x}_s, t)$
  $\mathbf{y} = \mathbf{f}_3(\mathbf{x}_s, t)$

- $\mathbf{x}_s$: states, $\mathbf{x}_n$: non-states

- Selection of states can almost always be done automatically

- Selection might need to be done dynamically along the trajectory

Manual State Selection

Reasons for manual state selection:

- Accuracy - relative coordinates
- Efficiency by avoiding inverting functions - e.g. thermodynamic properties
- Selecting a less nonlinear representation - e.g. Park transformation
- Avoiding dynamic state selection - mechanical systems
- Sensors - variables not states

State Selection Control

- `type StateSelect = enumeration(`
  `never, avoid, default, prefer, always);`

- stateSelect - new attribute of Real variables
- `Real w(stateSelect = StateSelect.prefer);`

  - *never*: Do not use as a state at all.
  - *avoid:* Avoid it as state in favour of those having the *default value*
  - *default*: If the variable does not appear differentiated in the model this means *never.*
  - *prefer*: Prefer it as state over those having the *default value.*
  - *always*: Do use it as a state.

Component arrays

- Basic idea of additions:
  - Make Modelica.Blocks user-friendly
    - Vectorization handled by translator not by user-models
  - Make blocks in Modelica.Blocks easy to write
    - Text-book models
    - More components
  - Support component arrays
  - Clearly defined language

Component arrays [old Modelica.Blocks.Sources.Sine]

```
block Sine "Generate sine signals"
  parameter Real amplitude[:]={1} "Amplitudes of sine waves";
  parameter SIunits.Frequency freqHz[:]={1} "Frequencies of sine waves";
  parameter SIunits.Angle phase[:]={0} "Phases of sine waves";
  parameter Real offset[:]={0} "Offsets of output signals";
  parameter SIunits.Time startTime[:]={0}  "Output = offset for time < startTime";
  extends Interfaces.MO(final nout=max([size(amplitude, 1); size(freqHz, 1);
      size(phase, 1); size(offset, 1); size(startTime, 1)]));
protected
  constant Real pi=Modelica.Constants.PI;
  parameter Real p_amplitude[nout]=(if size(amplitude, 1) == 1 then ones(nout)*
amplitude[1] else amplitude);
  parameter Real p_freqHz[nout]=(if size(freqHz, 1) == 1 then ones(nout)*freqHz[1] else
freqHz);
  parameter Real p_phase[nout]=(if size(phase, 1) == 1 then ones(nout)*phase[1] else
phase);
  parameter Real p_offset[nout]=(if size(offset, 1) == 1 then ones(nout)*offset [1] else
offset);
  parameter SIunits.Time p_startTime[nout]=(if size(startTime, 1) == 1 then
      ones(nout)*startTime[1] else startTime);
equation
  for i in 1:nout loop
    outPort.signal[i] = p_offset[i] + (if time < p_startTime[i] then 0 else
      p_amplitude[i]*Modelica.Math.sin(2*pi*p_freqHz[i]*(time - p_startTime[i])
       + p_phase[i]));
  end for;
end Sine;
```

Component arrays [new Modelica.Blocks.Sources.Sine]

```
block Sine "Generate sine signal"
  parameter Real              amplitude=1;
  parameter SIunits.Frequency freqHz=1;
  parameter SIunits.Angle     phase=0;
  parameter Real              offset=0;
  parameter SIunits.Time      startTime=0
     "Output = offset for time < startTime";
  extends Interfaces.SO;
equation
  y = offset + (if time < startTime then 0 else
        amplitude*Modelica.Math.sin(
          2* Modelica.Constants.pi *
          freqHz*(time-startTime)+ phase));
end Sine;
```

Component arrays [connectors]

```
connector InPortNew = input Real annotation(...);
```

Advantages:

- Drag-and-drop of real-valued connector
- Remove size-parameter
- Avoid `.signal[1]`
- Already existing u, y in Interfaces

Component arrays [use of library]

```
Sine base[3];
Sine source[3](each frequency=50,
               phase = {0,2,-2});

block CrossProduct
  extends MI2MO(final n=3);
equation
  y = cross(u1, u2);
end CrossProduct;
CrossProduct a;
equation
connect(source.y, a.u1);
connect(base.y,   a.u2);
```

Iterators

- Reduction Operators: sum, product, min, max
  - `sum(u[i]*v[i]^2 for i in 1:size(u,1))`
- Array construction
  - `{u[i]^2 for i in 1:size(u,1)}`
- Deduction of Ranges
  - `for i loop A[i] = B[i]^2; end for;`
  - `sum(u[i]*v[i]^2 for i)`
- Using **end** as array index
  - `v[end-1]=u[end];`

Replaceable / redeclare

Very minor additions in Modelica 2

Contents

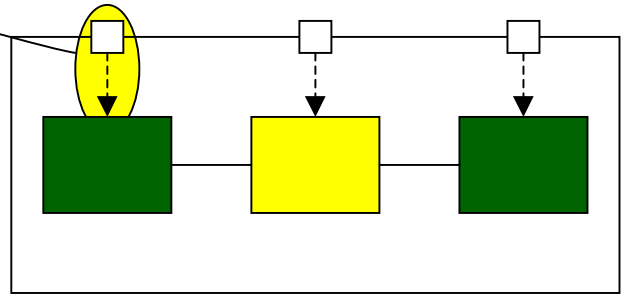
- Replaceable component
- Replaceable model
- Replaceable package
  - Medium models

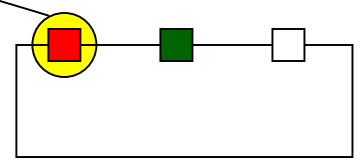
Replaceable component

- 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;

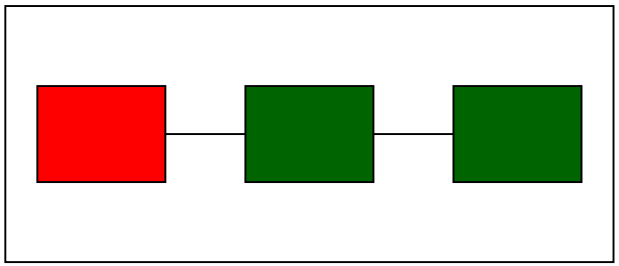
model C2 =
  C(redeclare RedModel   comp1,
    redeclare GreenModel comp2);
```





Equivalent to

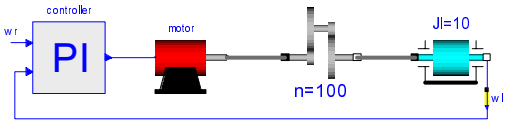
```
model C
  RedModel   comp1(p1=5);
  GreenModel comp2;
  GreenModel comp3;
  connect(…);
end C;
```



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);
```

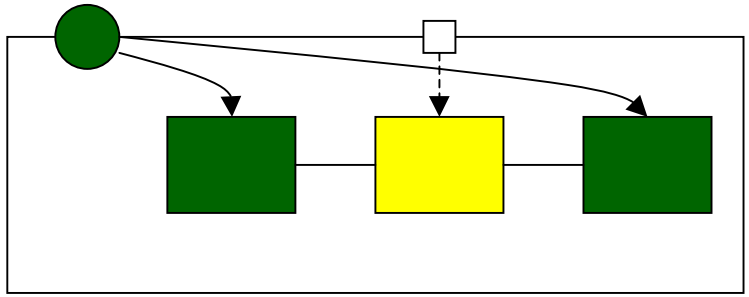


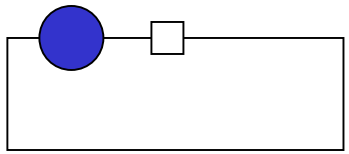
Replaceable model

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

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

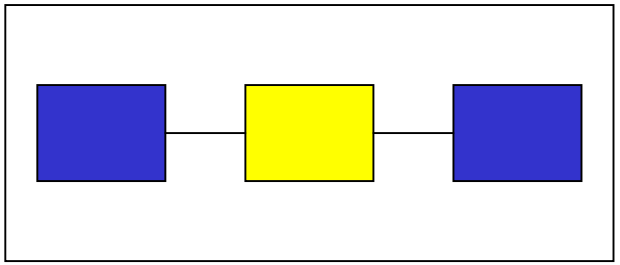
class C2 =
  C(redeclare model ColouredClass = BlueClass);
```





Equivalent to

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



Modifying packages

```
package FixedSampler=Modelica.Blocks.Discrete(
   redeclare model Sampler=...);
```

- Useful when:
  - All models depend on the same base-model
- Useful for:
  - Different sampler
  - Different fidelity
  - Difference in a *single* model

Replaceable packages

[courtesy of Mike Tiller, Ford]

- Consistent set of
  - models
  - connectors
  - utility functions
  - constants

- When replaceable model does not suffice:
  - Medium model
  - Heat-transfer property

What is a Medium Model?

- Material properties
  - Enthalpy, Energy
  - Viscosity
- Chemical representations (# of species)
- Chemistry (e.g. kinetics, combustion)
- Functions
  - AFR, Equivalence ratio, Ambient conditions

Key Points

- Reusability
  - Develop a single component model
- Variable Fidelity
  - Substitute different physics

- Conflicting requirements?

Design of an Engine

- Manifold (manifold volume)
- Throttle (throttle diameter)
- Engine block (inertia)
  - Engine Cylinders (quantity)
    - Engine Valves (diameters)
    - Combustion Chamber (bore, stroke, …)
  - Crankshaft (crank, connecting rod)
  - Engine Mount (stiffness)

Medium Components

- Manifold (manifold volume)
- Throttle (throttle diameter)
- Engine block (inertia)
  - Engine Cylinders
    - Engine Valves (diameters)
    - Combustion Chamber (bore, stroke, …)
  - Crankshaft (crank, connecting rod)
  - Engine Mount (stiffness)

Changing Medium Models

- “Flip of a switch”.
- Should not have to replace existing component models (not changing physical design)
- Should not be a burden on the model developers to maintain different fundamental components.

What is a Medium Model?

- Consistent set of
  - Models
  - Functions
  - Constants
  - Connectors
- Usage is on demand
  - Well organized
  - Computationally efficient

Modelica Implementation

- Modelica has great features to support this approach:
  - Packages
  - Partial definitions
  - Replaceable+Redeclare

Using Packages

- Consistent set of types.
- Partial package definition includes:
  - Property model
  - Number of species (constant)
  - Connectors
  - Utility functions

Package definition

```
within Ford.Thermodynamics;
partial package MediumModel
  constant Integer nspecies;
  model PropertyModel … end PropertyModel;
  connector MediumFlow … end MediumFlow;
  function AirFuelRatio … end AirFuelRatio;
  …
end MediumModel;
```

Medium Component

```
within Ford.Thermodynamics;
model MediumComponent
  replaceable package MediumModel
  =Ford.Thermodynamics.MediumModel;
end MediumComponent;

model Pump
  extends MediumComponent;
  MediumModel.MediumFlow flow_a;
  MediumModel.MediumFlow flow_b;
equation
  for i in MediumModel.nspecies loop … end for;
end Pump;
```

Diagrams

```
model subsystem
  extends MediumComponent;
  Pump p1(redeclare package
  MediumModel=MediumModel);
  Valve v1(redeclare package
  MediumModel=MediumModel);
equation
  connect(p1.flow_a,v1.flow_b);
  …
end subsystem;
```

Top Level

```
model Engine=Ford.Engines.V6(redeclare package
      MediumModel=Ford.Media.PerfectGas);
```

Positional *and* Named Arguments in Functions

```
function RealToString
  input  Real number;
  input  Real precision = 6;
  input  Real minLength = 0;
   ...
end RealToString;
```

Modelica 1.4:

```
RealToString(number = 2.0);
RealToString(2.0, 6, 0);
```

Added in Modelica 2.0:

```
RealToString(2.0);
RealToString(2.0, 6);
RealToString(2.0, precision=6);
RealToString(2.0, minLength=0);
```

Benefits of mixed arguments

- Possible to extend a function with new optional arguments
  - *Must have default values for extra argument*
  - Existing use not influenced
  - As adding extra parameters to model
  - Replaceable functions
- Same function for experts and novice users
  - Easier to find them
  - Various expert options with reasonable defaults
  - Easier maintenance

Optional outputs [Example]

```
function eigen
  parameter Integer n = size(A,1);
  input  Real    A[:,n];
  input  Boolean getREV = false;
  input  Boolean getLEV = false;
  output Real eigenValues[n,2];
  output Real REV[n,n](enable=getREV);
  output Real LEV[n,n](enable=getLEV);
 ...
end eigen;

  ev = eigen(A);
  b  = isStable(eigen(A));
  (ev, REV) = eigen(A, getREV=true);
  (ev, REV, LEV) = eigen(A, getREV=true,
                            getLEV=true);
```

Optional outputs

- Enable-attribute of ouputs
  - Default is enable=true
  - Given directly by inputs

- Explicitly use enable-attribute since:
  - Type safe
  - Understandable results
  - Experience from Matlab

Scripting/Interactive use of functions

- Modelica syntax:
- Can call:
  - Modelica functions
  - external functions
- Use for:
  - Post-processing
  - Understanding Modelica functions
- Will benefit from large function library (under development)

```
A = sum(B[:,k,:] for k);
ev = eigen(A);
(ev, REV, LEV) = eigen(A,getREV=true, getLEV=true);
eigen(A, getREV=true, getLEV=true);
```

Record constructor

```
record Complex "Complex number"
  Real re "real part";
  Real im "imaginary part";
end Complex;

function Complex "Automatically generated"
  input  Real re "real part";
  input  Real im "imaginary part";
  output Complex out(re=re,im=im);
end Complex;

function add "Add Comp. numbers"
   input  Complex u, v;
   output Complex w(re=u.re + v.re, im= u.im +
v.im);
  end add;

  Complex c1;
  Complex c2 = add(c1,Complex(sin(time), cos(time));
```

Data sheet libraries [using record constructor]

```
record MotorData
  parameter Real inertia;
  parameter Real nominalTorque;
  parameter Real maxTorque;
  parameter Real maxSpeed;
     ...
end MotorData;

model Motor
   MotorData data;
   // connector definitions
equation
   ...
end Motor;
```

When using a motor, specific values of the motor data could be given in the usual way:

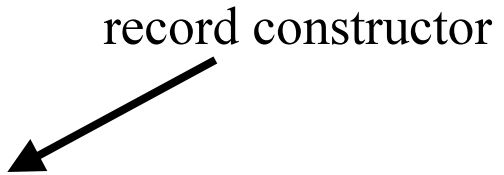
```
model Robot1
  Motor m1(data(inertia        = 0.001,
                nominalTorque = 10,
                maxTorque     = 20,
                maxSpeed      = 3600));
  Motor m2(data(...));
    ...
end Robot1;
```

When using the same motor type several times, it is better to define the motor data just ones, i.e., build up a **data sheet library** by **modifications** of the default values of the basic MotorData record

```
package Motors
  record M103 = MotorData(
    inertia        = 0.001,
    nominalTorque = 10,
    maxTorque      = 20,
    maxSpeed       = 3600);

  record M103a = M103(maxTorque=12);
  ...
end Motors;

model Robot2
  Motor m1(data = Motors.M103());
  Motor m2(data = Motors.M103a(
                        inertia=0.0012));
    ...
end Robot2;
```



External Utility Functions

- `char* ModelicaAllocateString(size_t len)`
  - Return strings from external routines
  - Variant for error handling
- `void ModelicaError(const char* string)`
  - " Assert" in external routines
  - Variant for format control of message
- `void ModelicaMessage(const char* string)`
  - Information messages from external routines
  - Variant for format control of message

External Utility Functions [Example]

Modelica:

```
function blanks "Creates string with n blanks"
   input  Integer n(min=0);
   output String blankString;
   external "C"
end blanks;
```

C:

```
#include "ModelicaUtilities.h"
const char* blanks(int n) {
  /* Create string with n blanks */
  char *c = ModelicaAllocateString(n);
  int i;
  for(i=0; i<n; ++i) c[i]=' ';
  c[n]='\0';
  return c;
}
```

External Objects

```
class MyTable
  extends ExternalObject;

  function constructor
    input  String  fileName;
    input  String  tableName;
    output MyTable table;
    external "C";
  end constructor;

  function destructor
    input  MyTable table;
    external "C";
  end destructor;
end MyTable;
```

External Objects [Example: Declaration continued]

```
function interpolateMyTable
  input  MyTable table;
  input  Real  u;
  output Real  y;
  external "C";
end interpolateMyTable;
```

External Object [Example: Use]

```
model test
  MyTable table1("testTables.txt", "table1");
  input  Real u1;
  output Real y1;
equation
  y1 = interpolateMyTable(table1, u1);
end test;
```

External Object [Example: C-implementation]

```
typedef struct {
  double* array;
  int     nrow;
  int     ncol;
  int     type; /* interpolation type */
  int     lastIndex; /* for search */
} MyTable;

void* initMyTable(const char* fileName,
                  const char* tableName) {
  MyTable* table=malloc(sizeof(MyTable));
  if (!table) ModelicaError("Not enough memory");
  // read table from file and store data in *table
  return (void*) table;
}
```

External Object [Example: C-implementation]

```
void closeMyTable(void* object) {
  MyTable* table = (MyTable*) object;
  free(table->array);
  free(table);
}

double interpolateMyTable(void* object,double u) {
  MyTable* table = (MyTable*) object;
  double y;
  // Interpolate using "*table" data
  return y;
}
```

Use of External Objects

- Special data-structures
  - User-defined table data structures
  - Access to external databases for properties
  - Handling of special matrices, e.g. sparse

- Hardware interfaces, since the constructor and destructor are called exactly once, even in case of errors

Try to avoid using External Objects (if straightforward)!

- How:
  - Convert property data into Modelica
    - Data-sheet libraries
  - Use one external Routine for reading data into Modelica table
  - Table-interpolation in Modelica (state-less with binary search)
    - Might be less efficient

- Why:
  - Better support for unexpected re-use
  - Better support for code modification
  - Interactive use
  - Not a black box

What is implemented of Modelica 2.0 in Dymola 4.2a?

- Initialization
- State-select-attribute
- Translator for component arrays (each, connect, etc)
- Replaceable component/model/function/package
- Scripts
- Array indexing with **end**

- Almost implemented:
  - Utility functions
  - Iterators
  - Mixed positional and named arguments