Introduction to Modelica

Michael Wetter and Thierry S. Nouidui
Simulation Research Group

June 23, 2015

BERKELEY LAB Lawrence Berkeley National Laboratory

Purpose and approach

The purpose is to have basic understanding of Modelica and be able to develop simple models.

The slides follow largely, and use many examples from, the online book from Michael Tiller: http://book.xogeny.com

Other references (and Buildings library user guide): http://simulationresearch.lbl.gov/modelica/userGuide/gettingStarted.html

Modelica reference: http://modref.xogeny.com/

Interactive tour: http://tour.xogeny.com

Basic syntax

Basic equations

Consider

$\dot{x} = 1 - x$

Initial conditions

$x_0 = 2$

```
model FirstOrderInitial "First order equation with initial value"
  Real x "State variable";
initial equation
  x = 2 "Used before simulation to compute initial values";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderInitial;
```

$\dot{x}_0 = 0$

```
model FirstOrderSteady
  "First order equation with steady state initial condition"
  Real x "State variable";
initial equation
  der(x) = 0 "Initialize the system in steady state";
equation
  der(x) = 1-x "Drives value of x toward 1.0";
end FirstOrderSteady;
```

Adding units

$$m\, c_p\, \frac{dT}{dt} = h\, A\, (T_{inf} - T)$$

```
model NewtonCoolingWithUnits "Cooling example with physical units"
  parameter Real T_inf(unit="K")=298.15 "Ambient temperature";
  parameter Real T0(unit="K")=363.15 "Initial temperature";
  parameter Real h(unit="W/(m2.K)")=0.7 "Convective cooling coefficient";
  parameter Real A(unit="m2")=1.0 "Surface area";
  parameter Real m(unit="kg")=0.1 "Mass of thermal capacitance";
  parameter Real c_p(unit="J/(K.kg)")=1.2 "Specific heat";
  Real T(unit="K") "Temperature";
initial equation
  T = T0 "Specify initial value for T";
equation
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingWithUnits;
```

To avoid this verbosity, **`Modelica.SIunits`** declares types such as

```
type Temperature=Real(unit="K", min=0);
```

Now, we can write

```
parameter Modelica.SIunits.Temperature T_inf=298.15 "Ambient temperature";
```

`records` are convenient to collect data that belong together

Declare the class Vector

```
record Vector "A vector in 3D space"
  Real x;
  Real y;
  Real z;
end Vector;
```

Declare an instance

```
parameter Vector v(x=1.0, y=2.0, z=0.0);
```

Use it in the code

```
equation
volume = v.x*v.y*v.z;
```

See for example `Buildings.Fluid.Chillers.Data` and the various other `Data` packages in the Buildings library.

Discrete behavior

If-then to model time events

```
model NewtonCoolingDynamic
  "Cooling example with fluctuating ambient conditions"
…
initial equation
  T = T0 "Specify initial value for T";
equation
  if time<=0.5 then
    T_inf = 298.15 "Constant temperature when time<=0.5";
  else
    T_inf = 298.15-20*(time-0.5) "Otherwise, increasing";
  end if;
  m*c_p*der(T) = h*A*(T_inf-T) "Newton's law of cooling";
end NewtonCoolingDynamic;
```

Note: time is a built-in variable.

An alternative formulation is

```
T_inf = 298.15 - (if time<0.5 then 0 else 20*(time-0.5));
```

when construct

```
model BouncingBall "The 'classic' bouncing ball model"
  type Height=Real(unit="m");
  type Velocity=Real(unit="m/s");
  parameter Real e=0.8 "Coefficient of restitution";
  parameter Height h0=1.0 "Initial height";
  Height h;
  Velocity v;
initial equation
  h = h0;
equation
  v = der(h);
  der(v) = -9.81;
  when h<0 then
    reinit(v, -e*pre(v));
  end when;
end BouncingBall;
```

Becomes active *when* the condition becomes true

Reinitializes the state variable

Use the value that *v* had prior to this section

State event handling

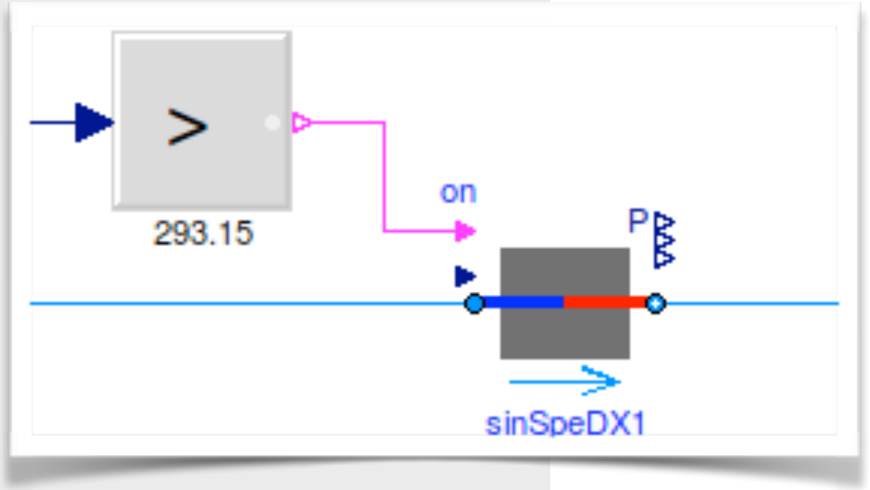
```
model Decay
  Real x;
initial equation
  x = 1;
equation
  // wrong: der(x) = -sqrt(x);
  // wrong: der(x) = if x>=0 then -sqrt(x) else 0;
  der(x) = if noEvent(x>=0) then -sqrt(x) else 0;
end Decay;
```

Why are the other two formulations wrong?

Avoid chattering by using hysteresis

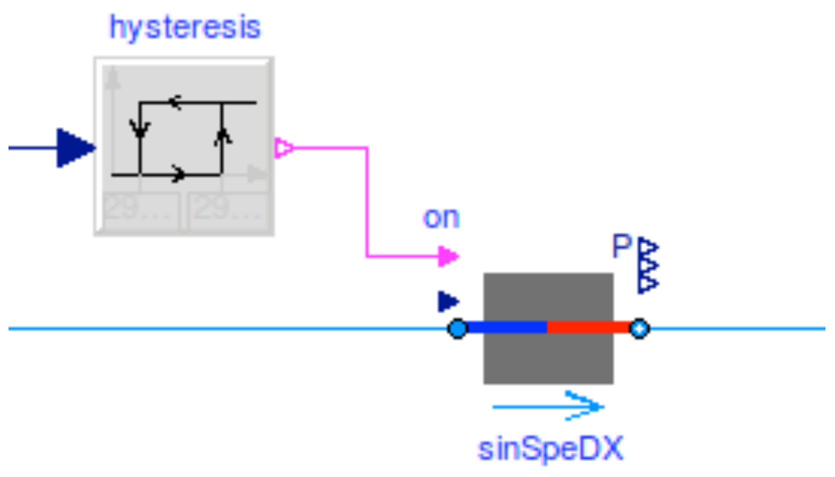
What will go wrong with this code?

```
model ChatteringControl "A control strategy that will ‘chatter'"

  type HeatCapacitance=Real(unit="J/K");
  type Temperature=Real(unit="K");
  type Heat=Real(unit="W");
  type Mass=Real(unit="kg");
  type HeatTransferCoefficient=Real(unit="W/K");

  parameter HeatCapacitance C=1.0;
  parameter HeatTransferCoefficient h=2.0;
  parameter Heat Qcapacity=25.0;
  parameter Temperature Tamb=285;
  parameter Temperature Tbar=295;

  Boolean heat "Indicates whether heater is on";
  Temperature T;
  Heat Q;
initial equation
  T = Tbar+5;
equation
  heat = T<Tbar;
  Q = if heat then Qcapacity else 0;
  C*der(T) = Q-h*(T-Tamb);
end ChatteringControl;
```

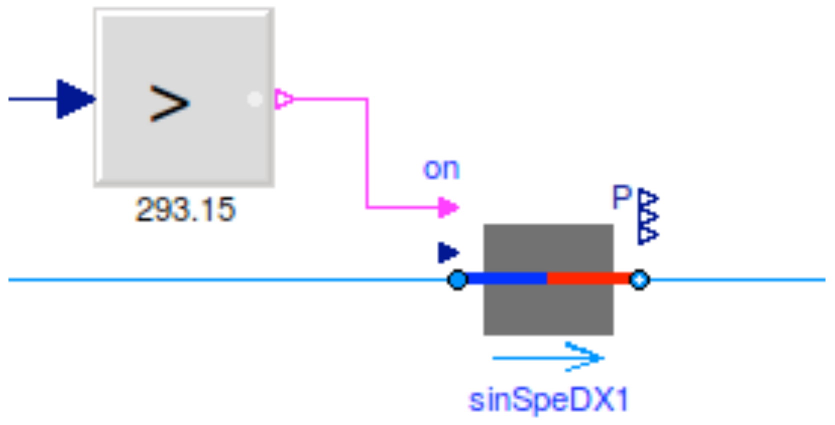


Such a problem was indeed reported by a user

Correct configuration



Avoid this configuration

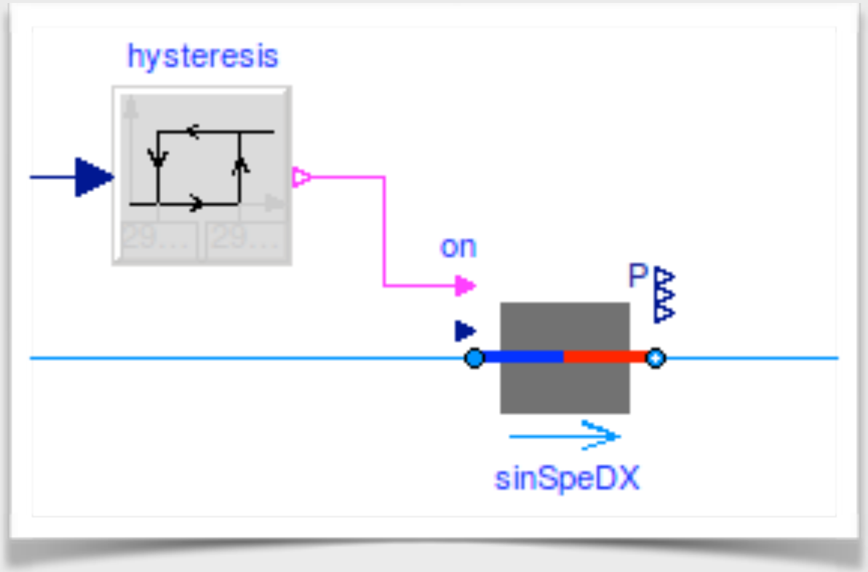


This can work in fixed time step simulators, but it won't in variable time step simulators that handle events.

We need to add a hysteresis when switching the value of `heat`

```
model HysteresisControl "A control strategy that doesn't chatter"
  ...
  Boolean heat(start=false) "Indicates whether heater is on";
  parameter Temperature Tbar=295;
  Temperature T;
  Heat Q;
initial equation
  T = Tbar+5;
  heat = false;
equation
  Q = if heat then Qcapacity else 0;
  C*der(T) = Q-h*(T-Tamb);
  when {T>Tbar+1, T<Tbar-1} then
    heat = T<Tbar;
  end when;
end HysteresisControl;
```

Active when any element is true



Events

This can trigger events

```
Boolean late;
equation
late = time>=5.0 "This will generate an event";
```

It is hard for a code translator to understand that this expression is differentiable

```
x = if (x<0) then 0 else x^3;
```

Use the **smooth()** operator

```
x = smooth(if (x<0) then 0 else x^3, 2);
```

Expression is 2 times continuously differentiable

Events can also be generated by certain functions,
see http://book.xogeny.com/behavior/discrete/events/

`if` expressions

```
if cond1 then
  // Statements used if cond1==true
elseif cond2 then
  // Statements used if cond1==false and cond2==true
// ...
elseif condn then
  // Statements used if all previous conditions are false
  // and condn==true
else
  // Statements used otherwise
end if;
```

Each branch must have the same number of equations because of the single assignment rule.

In Modelica, there must be exactly one equation used to determine the value of each variable.

`if` versus `when`

`if` branches are always evaluated **if** the condition **is** true.

`when` statements become active only for an instant **when** the condition **becomes** true.

Use `when` for example to reinitialize states.

Arrays

Arrays are fixed at compile time. They can be declared as

```
parameter Integer n = 3;
Real x[n];
Real y[size(x,1), 2];
Real z[:] = {2.0*i for i in 1:n};  // {2, 4, 6}
Real fives[:] = fill(5.0, n);      // {5, 5, 5}
```

Many functions take arrays as arguments, see http://book.xogeny.com/behavior/arrays/functions/.

For example, $s = \sum_{i=1}^{3} z_i$

```
s = sum(z);
```

Looping

```
parameter Integer n = 3;
Real x[n];
equation
  for i in 1:n loop
    x[i] = i;
  end for;
```

Functions

Functions have imperative programming assignments

```
within Buildings.Fluid.HeatExchangers.BaseClasses;

function lmtd "Log-mean temperature difference"
  input Modelica.SIunits.Temperature T_a1 "Temperature at port a1";
  input Modelica.SIunits.Temperature T_b1 "Temperature at port b1";
  input Modelica.SIunits.Temperature T_a2 "Temperature at port a2";
  input Modelica.SIunits.Temperature T_b2 "Temperature at port b2";
  output Modelica.SIunits.TemperatureDifference lmtd
    "Log-mean temperature difference";

protected
  Modelica.SIunits.TemperatureDifference dT1
   "Temperature difference side 1";
  Modelica.SIunits.TemperatureDifference dT2
    "Temperature difference side 2";

algorithm
  dT1  := T_a1 - T_b2;
  dT2  := T_b1 - T_a2;
  lmtd := (dT2 - dT1)/Modelica.Math.log(dT2/dT1);

annotation (…);
end lmtd;
```

`algorithm` sections can also be used in a `model` or `block` if needed, but functions must use an `algorithm` section.

Annotations are used to tell a tool properties of the function, such as how often it can be differentiated

```
function enthalpyOfLiquid "Return the specific enthalpy of liquid"
  extends Modelica.Icons.Function;
  input Modelica.SIunits.Temperature T "Temperature";
  output Modelica.SIunits.SpecificEnthalpy h "Specific enthalpy";
algorithm
  h := cp_const*(T-reference_T);
annotation (
  smoothOrder=5,
  Inline=true,
Documentation(info="…", revisions="…"));
end enthalpyOfLiquid;
```

Annotations are used to tell a tool properties of the function, such as how often it can be differentiated

By default, functions are pure, i.e., they have no side effect.

Functions can call C, Fortran 77, and dynamic and static linked libraries.

Functions can have memory.

- See http://book.xogeny.com/behavior/functions/interpolation/
- Used for example by borehole (`Buildings.Fluid.HeatExchangers.Boreholes.BaseClasses.ExtendableArray`) and by `Buildings.Rooms.BaseClasses.CFDExchange`

See http://book.xogeny.com/behavior/functions/func_annos/ for other function annotations

`inverse` allows inverting functions without iteration

```
function Quadratic "A quadratic function"
  input Real a "2nd order coefficient";
  input Real b "1st order coefficient";
  input Real c "constant term";
  input Real x "independent variable";
  output Real y "dependent variable";
algorithm
  y := a*x*x + b*x + c;
  annotation(inverse(x = InverseQuadratic(a,b,c,y)));
end Quadratic;
```

```
function InverseQuadratic
  "The positive root of a quadratic function"
  input Real a;
  input Real b;
  input Real c;
  input Real y;
  output Real x;
algorithm
  x := sqrt(b*b - 4*a*(c - y))/(2*a);
end InverseQuadratic;
```

Can compute without iteration: `5=Quadratic(a=2, b=3, c=1, x=x);`

An example of an `impure` function implemented in C

```
impure function computeHeat "Modelica wrapper for an embedded C controller"
  input Real T;
  input Real Tbar;
  input Real Q;
  output Real heat;
  external "C" annotation (Include="#include \"ComputeHeat.c\"",
    IncludeDirectory="modelica://ModelicaByExample.Functions.ImpureFunctions/source");
end computeHeat;
```

```
#ifndef _COMPUTE_HEAT_C_
#define _COMPUTE_HEAT_C_

#define UNINITIALIZED -1
#define ON 1
#define OFF 0

double
computeHeat(double T, double Tbar, double Q) {
  static int state = UNINITIALIZED;
  if (state==UNINITIALIZED) {
    if (T>Tbar) state = OFF;
    else state = ON;
  }
  if (state==OFF && T<Tbar-2) state = ON;
  if (state==ON && T>Tbar+2) state = OFF;

  if (state==ON) return Q;
  else return 0;
}

#endif
```

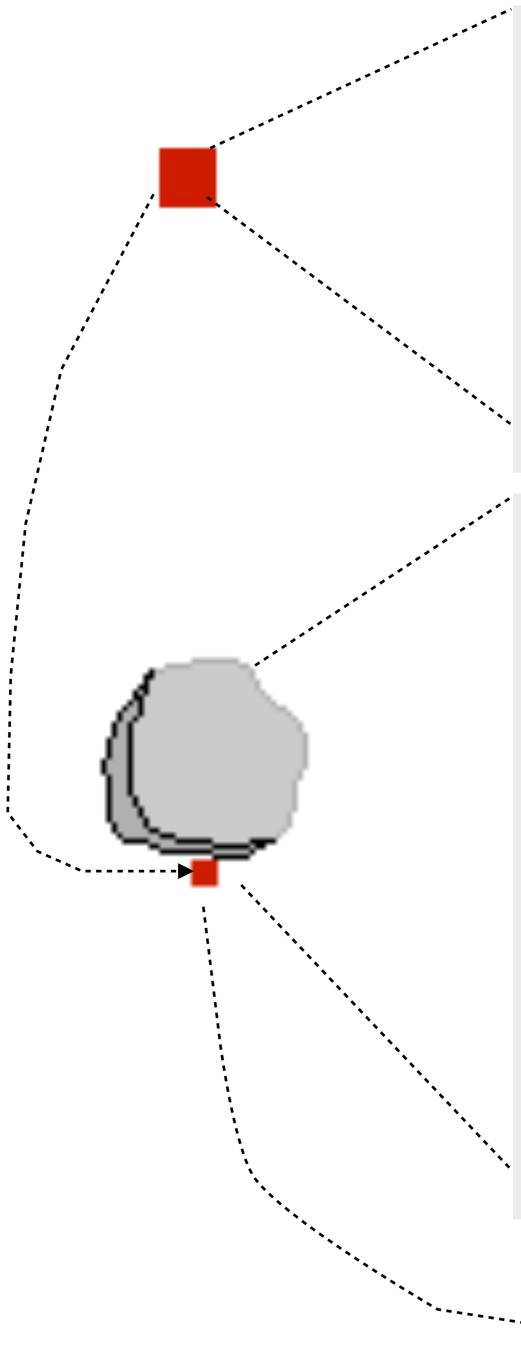
static variable

This function can return a different result if called with the same arguments. A translator must know this.

`impure` functions can only be called from other `impure` functions, from a `when`-equation or a `when`-statement.

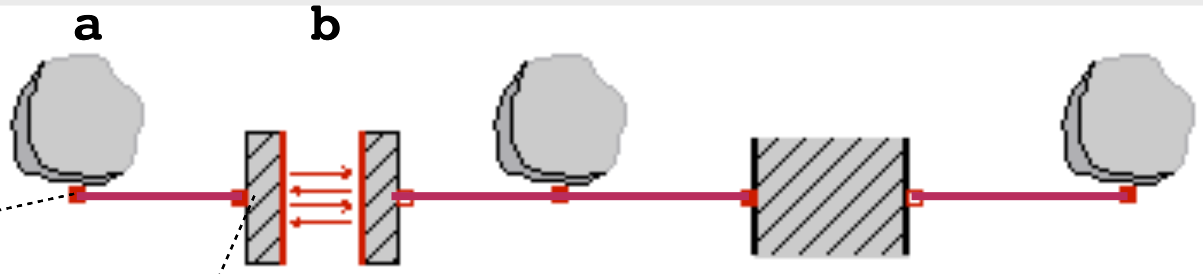
Object-oriented modeling

Object-oriented modeling



```
connector HeatPort_a "Thermal port for 1-dim. heat
transfer"
  Modelica.SIunits.Temperature T "Port temperature";
  flow Modelica.SIunits.HeatFlowRate Q_flow
    "Heat flow rate (positive if flowing from
     outside into the component)";
end HeatPort_a;
```

```
model HeatCapacitor "Lumped thermal element storing heat"

  parameter Modelica.SIunits.HeatCapacity C "Heat
capacity";
  Modelica.SIunits.Temperature T "Temperature of element";
  Interfaces.HeatPort_a port;

equation
  T = port.T;
  C*der(T) = port.Q_flow;
end HeatCapacitor;
```



```
connect(a.port, b.port);
```

```
a.port.T = b.port.T;
0 = a.port.Q_flow + b.port.Q_flow;
```

Packages

In Modelica, everything is part of a package

models are organized in hierarchical packages

```
within Buildings.Fluid.HeatExchangers.BaseClasses;
function lmtd "Log-mean temperature difference"
  input Modelica.SIunits.Temperature T_a1 "Temperature at port a1";
  …
```

Type definitions are part of packages

Packages can contain

- other packages,
- constants,
- functions,
- models,
- blocks,
- types

They cannot contain

- parameters,
- variable declaration,
- equation or algorithm sections

Packages

Basic syntax

```
package OuterPackage "A package that wraps a nested package"
  // Anything contained in OuterPackage
  package NestedPackage "A nested package"
    // Things defined inside NestedPackage
  end NestedPackage;
end OuterPackage;
```

Storing in the file system

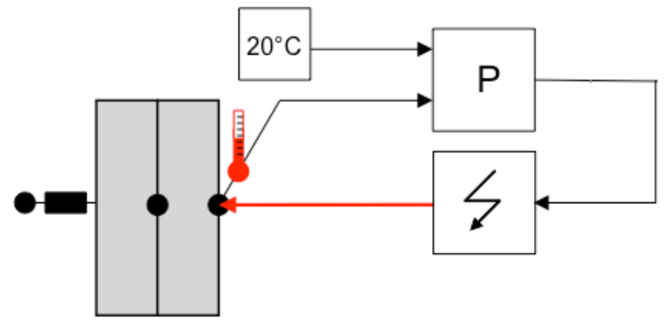
```
/RootPackage                 # Top-level package stored as a directory
  package.mo                 # Indicates this directory is a package
  package.order              # Specifies an ordering for this package
  NestedPackageAsFile.mo     # Definitions stored in one file
  /NestedPackageAsDir        # Nested package stored as a directory
    package.mo               # Indicates this directory is a package
    package.order            # Specifies an ordering for this package
```

Referencing resources with a URL

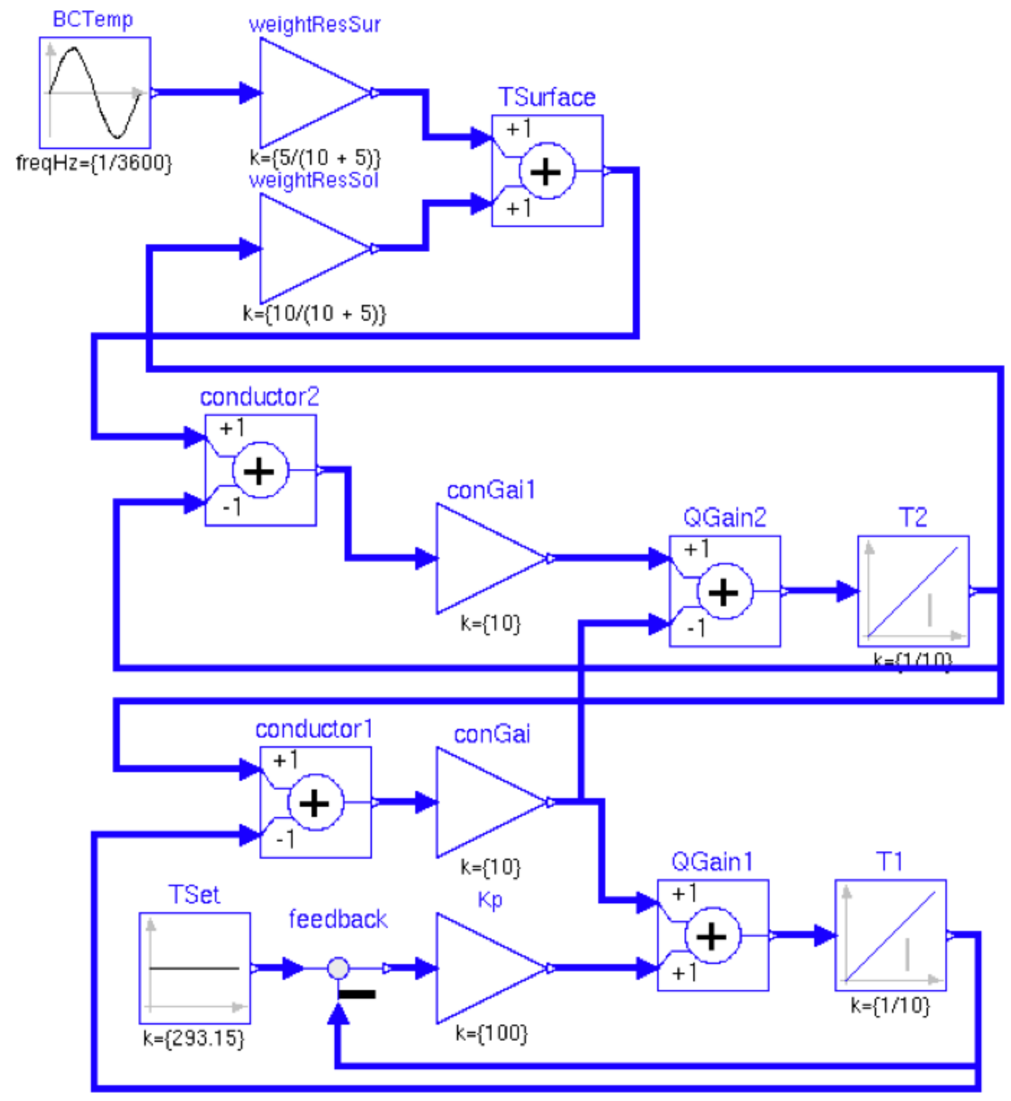
```
modelica://RootPackage/Resources/logo.jpg
```

A convention of the Annex60 and Buildings library is that each package must be in a separate directory, and each class in a separate file. Reason: easier merging and version control.

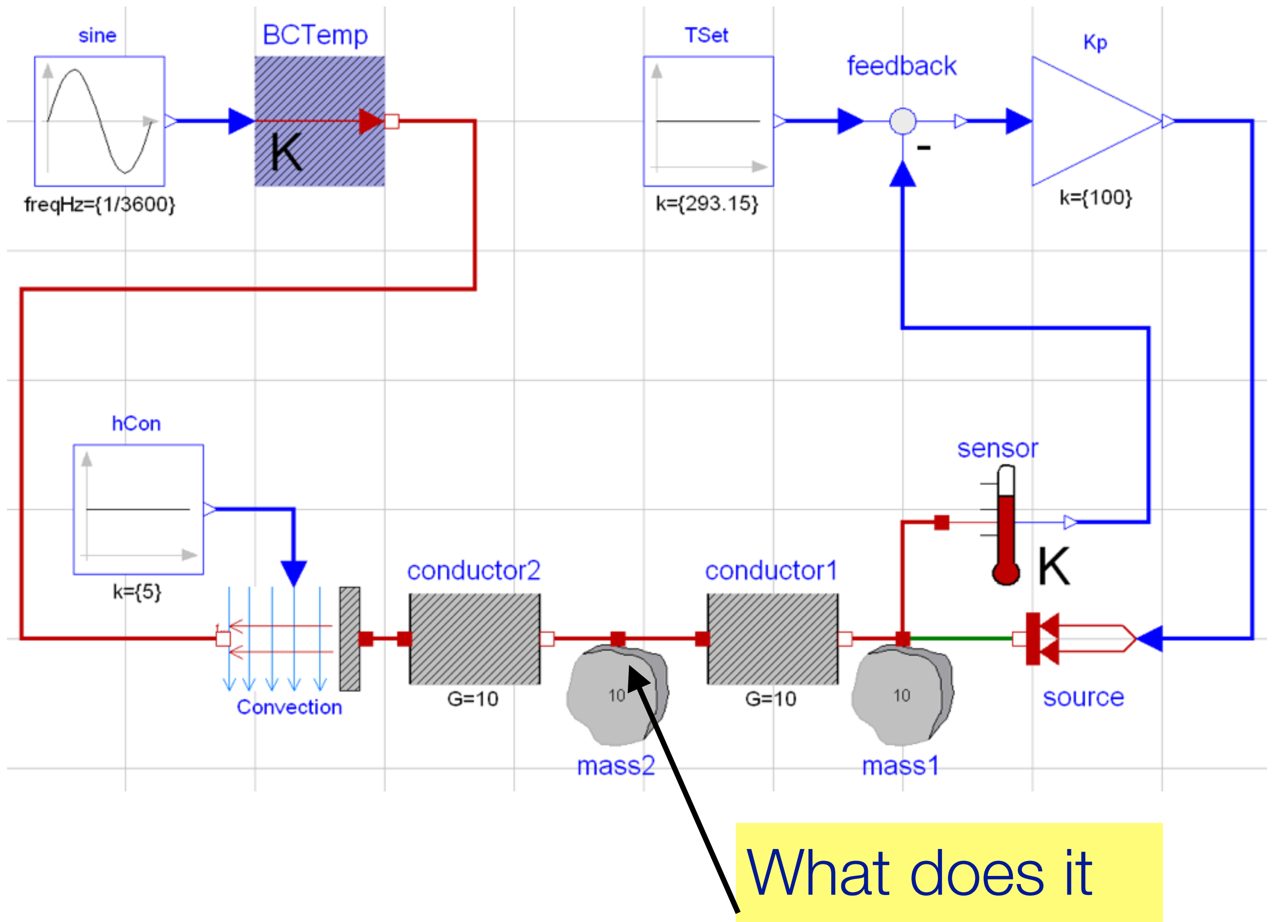
Acausal connectors are used to enable assembling models schematically



Block Diagram Modeling



Acausal Modeling



What does it mean to connect three ports?

Physical connectors and balanced models

Acausal connectors

- input/output determined at compilation time
- can connect none or multiple components to a port
- A connector should contain all information required to uniquely define the boundary condition

Domain	Potential	Flow	Stream
Heat flow	T	Q_flow	
Fluid flow	p	h_outflow X_outflow C_outflow	m_flow
Electrical	V	I	
Translational	x	F	

Automatically summed up at connections to satisfy conservation equation.

Requirement of locally balanced models

- # of equations = # of variables, at each level of model hierarchy.

```
connector Pin "Pin of an electrical component"
  SIunits.Voltage v "Potential at the pin";
  flow SIunits.Current i "Current flowing into the pin";
end Pin;

model Ground "Ground node"
  Modelica.Electrical.Analog.Interfaces.Pin p;
equation
  p.v = 0;
end Ground;
```

Connectors declare the interfaces, or ports, of models.

No equations are allowed.

Connectors for most physical ports exists in the MSL.

```
 connector Thermal
     Modelica.SIunits.Temperature T;
     flow Modelica.SIunits.HeatFlowRate Q_flow;
  end Thermal;
```

```
connector FluidPort
  replaceable package Medium =
Modelica.Media.Interfaces.PartialMedium;

  flow Medium.MassFlowRate m_flow;
  Medium.AbsolutePressure p;
  stream Medium.SpecificEnthalpy h_outflow;
  stream Medium.MassFraction Xi_outflow[Medium.nXi];
  stream Medium.ExtraProperty C_outflow[Medium.nC];
end FluidPort;
```

Components

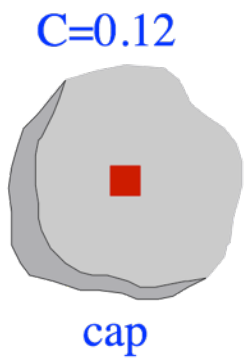
A simple heat storage element

```
within Modelica.Thermal.HeatTransfer;

package Interfaces "Connectors and partial models"

  partial connector HeatPort "Thermal port for 1-dim. heat transfer"
    Modelica.SIunits.Temperature T "Port temperature";
    flow Modelica.SIunits.HeatFlowRate Q_flow
      "Heat flow rate (positive if flowing from outside into the
component)";
  end HeatPort;

  connector HeatPort_a
    "Thermal port for 1-dim. heat transfer (filled rectangular icon)"
    extends HeatPort;

    annotation(...,
      Icon(coordinateSystem(preserveAspectRatio=true,
                            extent={{-100,-100},{100,100}}),
                            graphics={Rectangle(
                              extent={{-100,100},{100,-100}},
                              lineColor={191,0,0},
                              fillColor={191,0,0},
                              fillPattern=FillPattern.Solid)}));
  end HeatPort_a;

end Interfaces;
```

A simple heat storage element

```
within ModelicaByExample.Components.HeatTransfer;
model ThermalCapacitance "A model of thermal capacitance"
  parameter Modelica.SIunits.HeatCapacity C "Thermal capacitance";
  parameter Modelica.SIunits.Temperature T0 "Initial temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port
    annotation (Placement(transformation(extent={{-10,-10},{10,10}})));
initial equation
  port.T = T0;
equation
  C*der(port.T) = port.Q_flow;
end ThermalCapacitance;
```

$$\longleftarrow \quad C\frac{dT}{dt} = \dot{Q}$$



```
within ModelicaByExample.Components.HeatTransfer.Examples;
model Adiabatic "A model without any heat transfer"
  ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
    "Thermal capacitance component"
    annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
end Adiabatic;
```

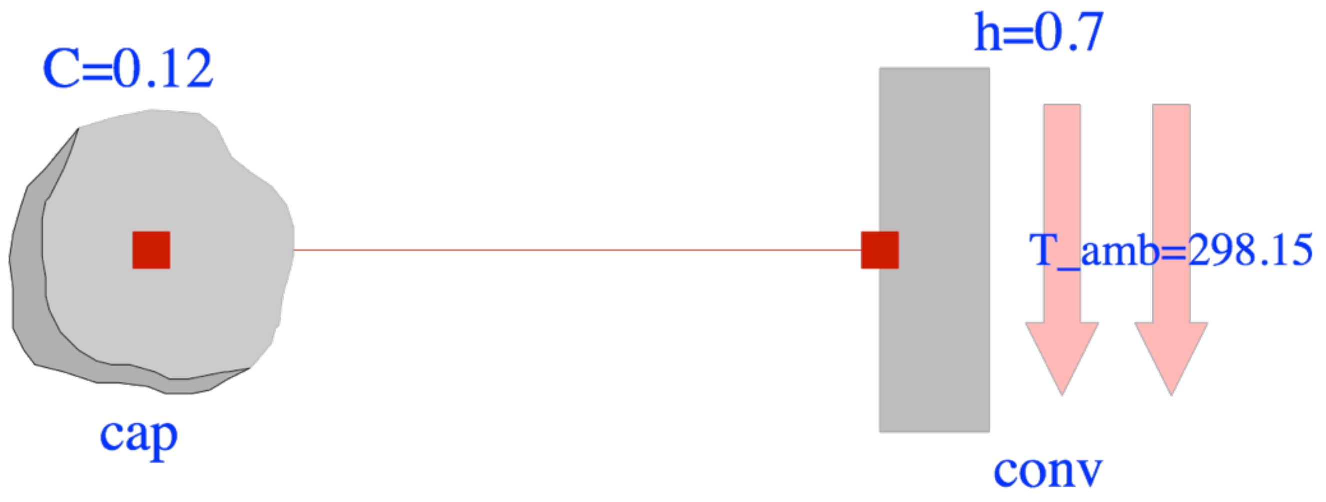
Add convection to ambient

```
within ModelicaByExample.Components.HeatTransfer;
model ConvectionToAmbient "An overly specialized model of convection"
  parameter Modelica.SIunits.CoefficientOfHeatTransfer h;
  parameter Modelica.SIunits.Area A;
  parameter Modelica.SIunits.Temperature T_amb "Ambient temperature";
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a
    annotation (Placement(transformation(extent={{-110,-10},{-90,10}})));
equation
  port_a.Q_flow = h*A*(port_a.T-T_amb) "Heat transfer equation";
end ConvectionToAmbient;
```

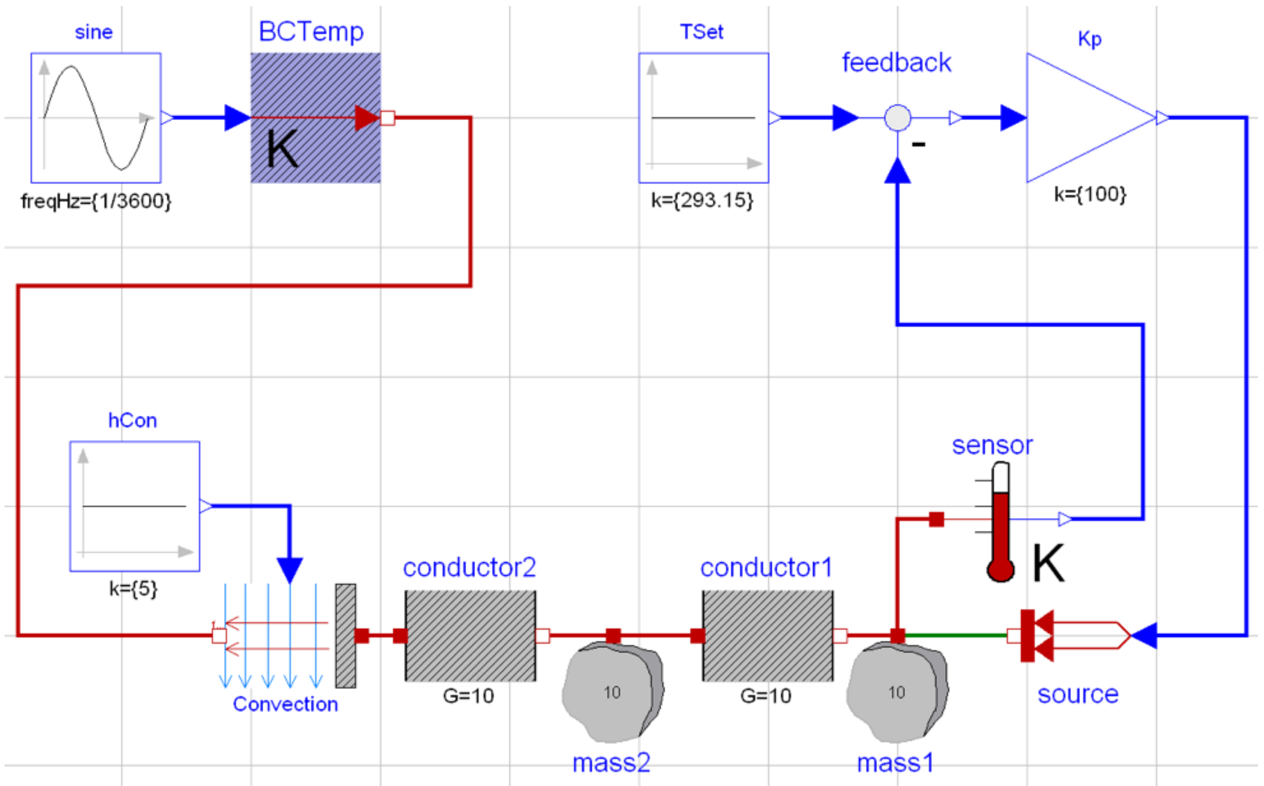
Add convection to ambient

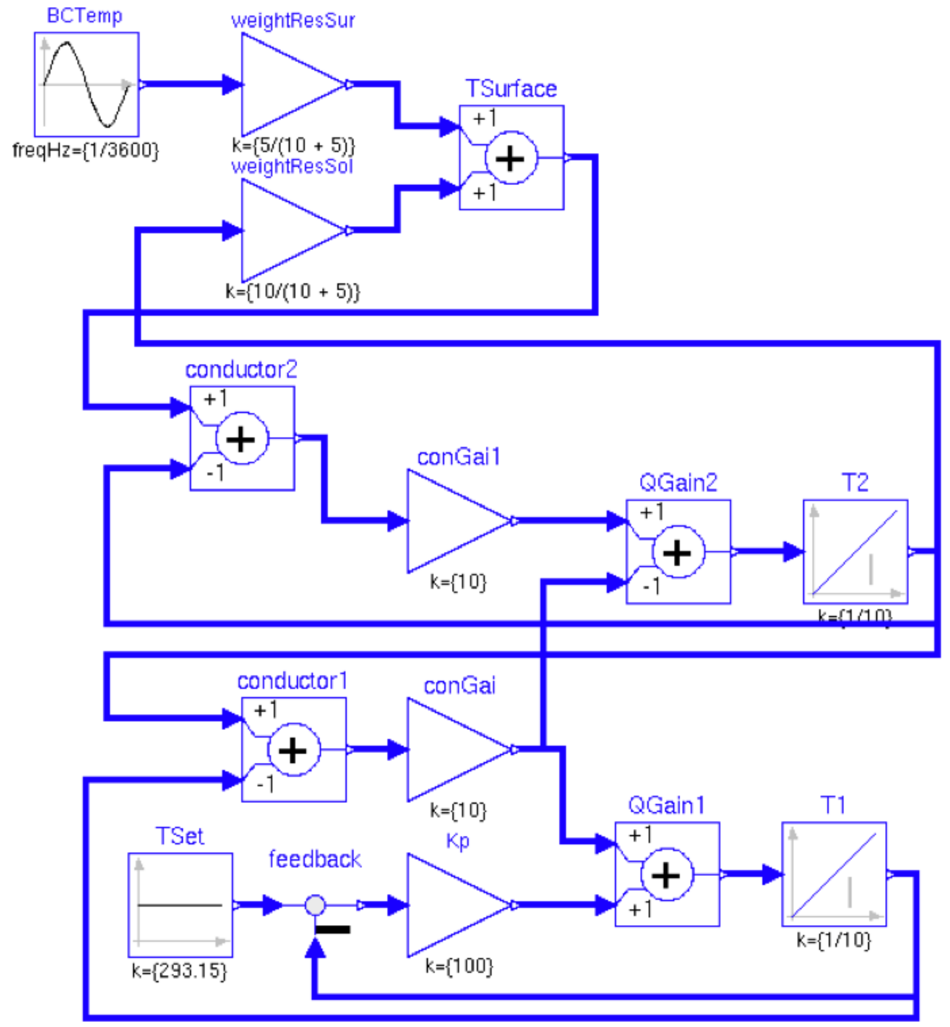
```
within ModelicaByExample.Components.HeatTransfer.Examples;
model CoolingToAmbient "A model using convection to an ambient condition"

  ThermalCapacitance cap(C=0.12, T0(displayUnit="K") = 363.15)
    "Thermal capacitance component"
    annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));

  ConvectionToAmbient conv(h=0.7, A=1.0, T_amb=298.15)
    "Convection to an ambient temprature"
    annotation (Placement(transformation(extent={{20,-10},{40,10}})));

equation
  connect(cap.port, conv.port_a)
   annotation (
  Line(points={{-20,0},{20,0}},
       color={191,0,0},
      smooth=Smooth.None));
end CoolingToAmbient;
```

```
cap.port.T = conv.port_a.T;
cap.port.Q_flow + conv.port_a.Q_flow = 0;
```



Composability





Connectors

- Designed for *physical* compatibility, not causal compatibility
- No *a-priori* knowledge is needed to connect components

Multi-physics

- Components can have a heat port and a fluid port (and a control input signal, …)

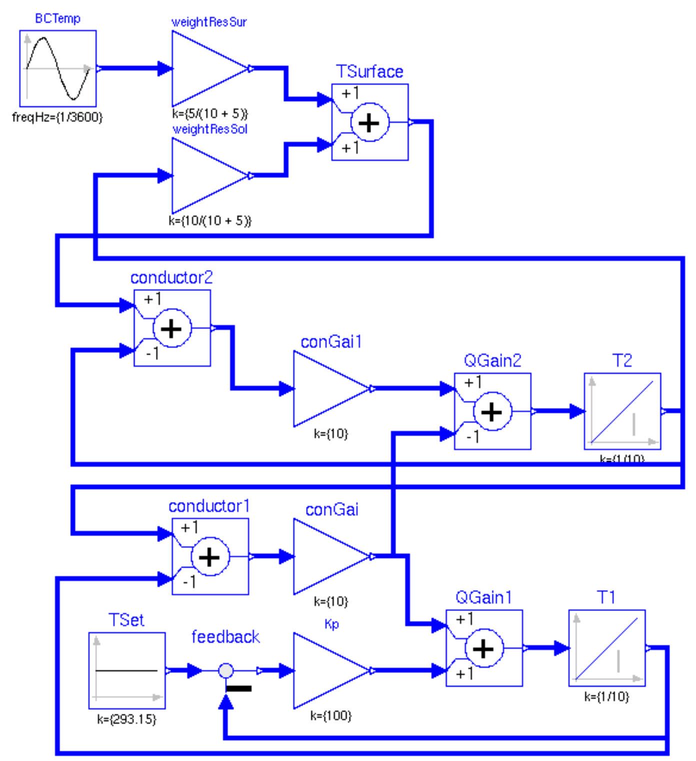
Multi-domain

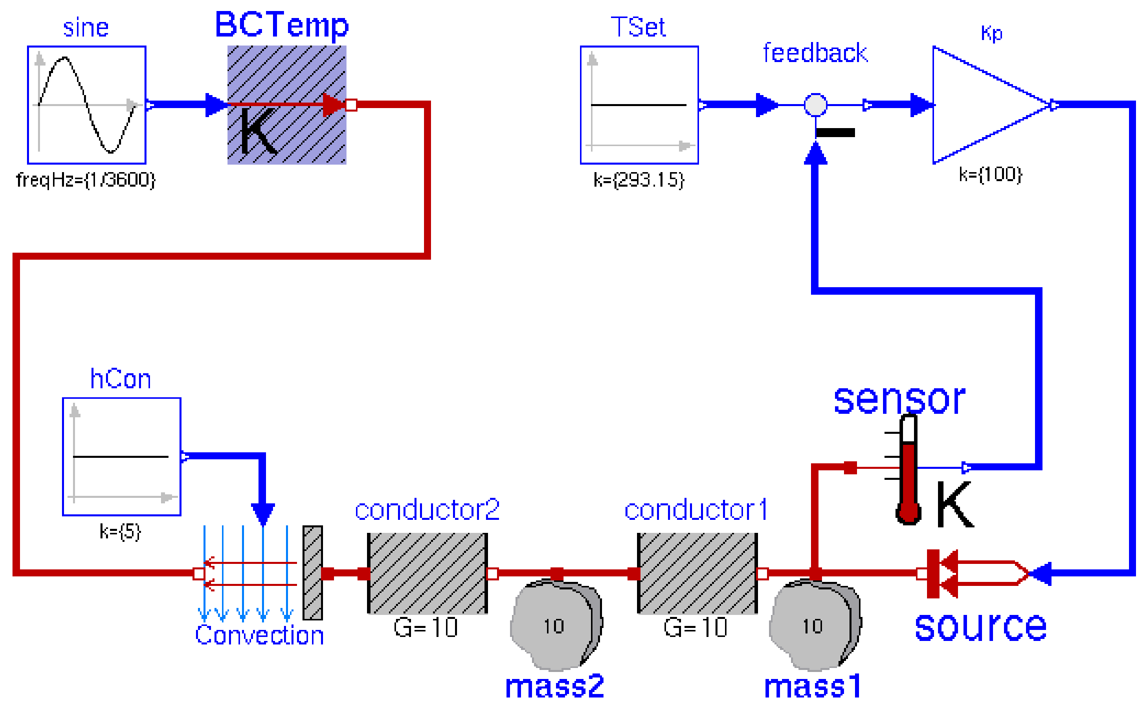
- Can combine schematic diagrams, block diagrams (and state machines, …)

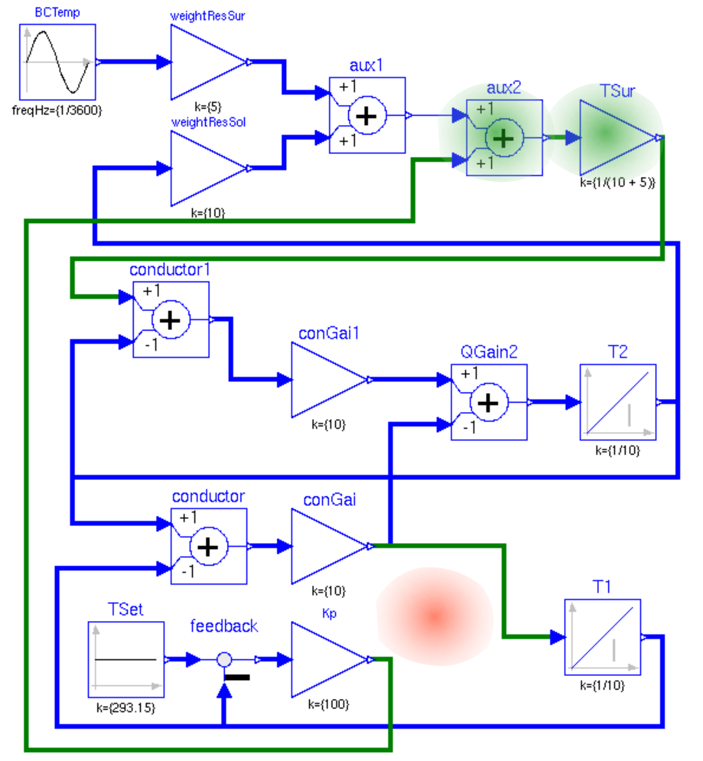
Reusability

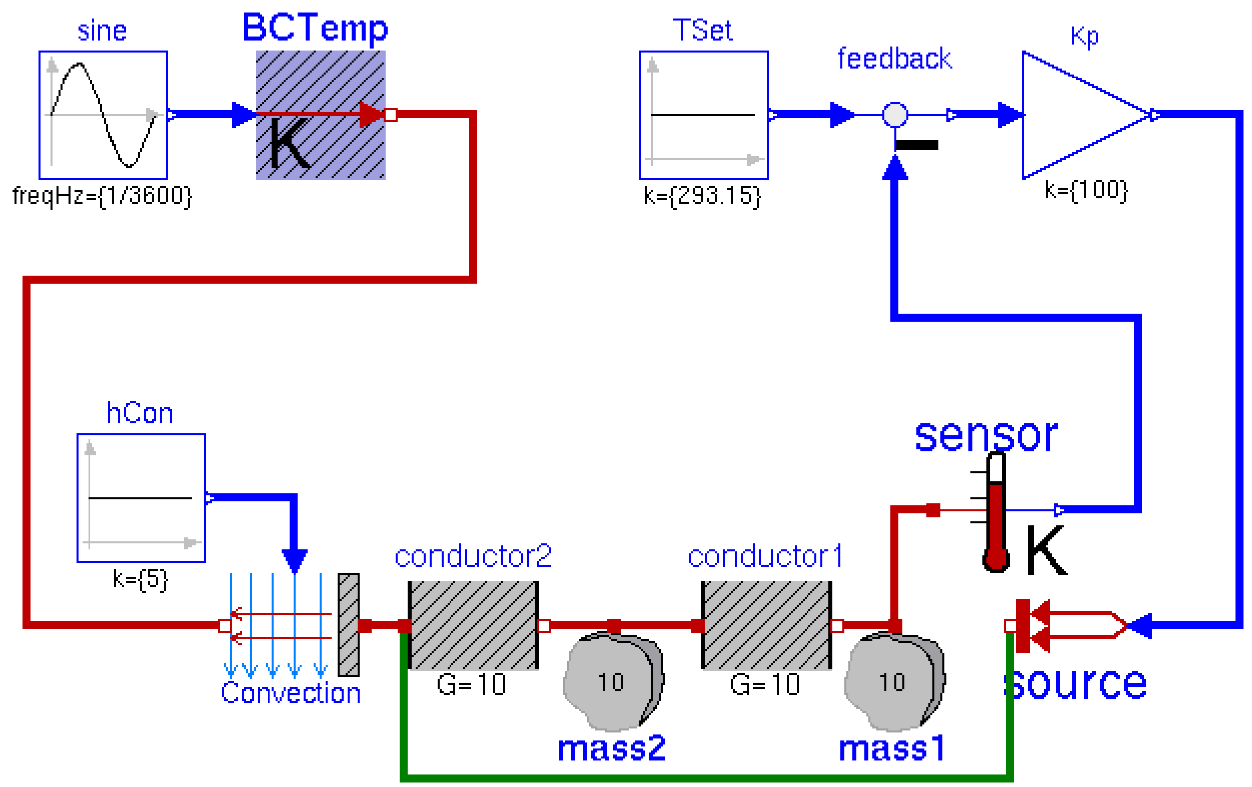
- Change the system architecture by deleting and dragging components from a library (in block diagrams, profound changes would be required)

Acausal components leads to much higher readability and reusability









Arrays of components

```
  HTC.HeatCapacitor capacitance[n](
      each final C=C/n,
      each T(start=T0, fixed=true));
  HTC.ThermalConductor wall[n](each final G=G_wall/n);
  HTC.ThermalConductor rod_conduction[n-1](each final G=G_rod);
```

Array length must be known at translation time.

See Buildings.HeatTransfer.Conduction which extensively uses arrays.

Propagation of parameters and the medium package

Users should only have to assign parameters and the medium at the top-level model.

```
within Buildings.Fluid.HeatExchangers;
model HeaterCooler_T
  "Ideal heater or cooler with a prescribed outlet temperature"
  extends Interfaces.PartialTwoPortInterface;
  extends Interfaces.TwoPortFlowResistanceParameters(
    final computeFlowResistance=
            (abs(dp_nominal) >  Modelica.Constants.eps));

 parameter Modelica.SIunits.Pressure dp_nominal
   "Pressure difference at nominal mass flow rate";
…

  Buildings.Fluid.FixedResistances.FixedResistanceDpM preDro(
    redeclare final package Medium = Medium,
    final m_flow_nominal=m_flow_nominal,
    …) "Pressure drop model";
```

The `final` keyword prevents users from changing the assignment.

Default values for parameters should only be used when those defaults are reasonable for the vast majority of cases.

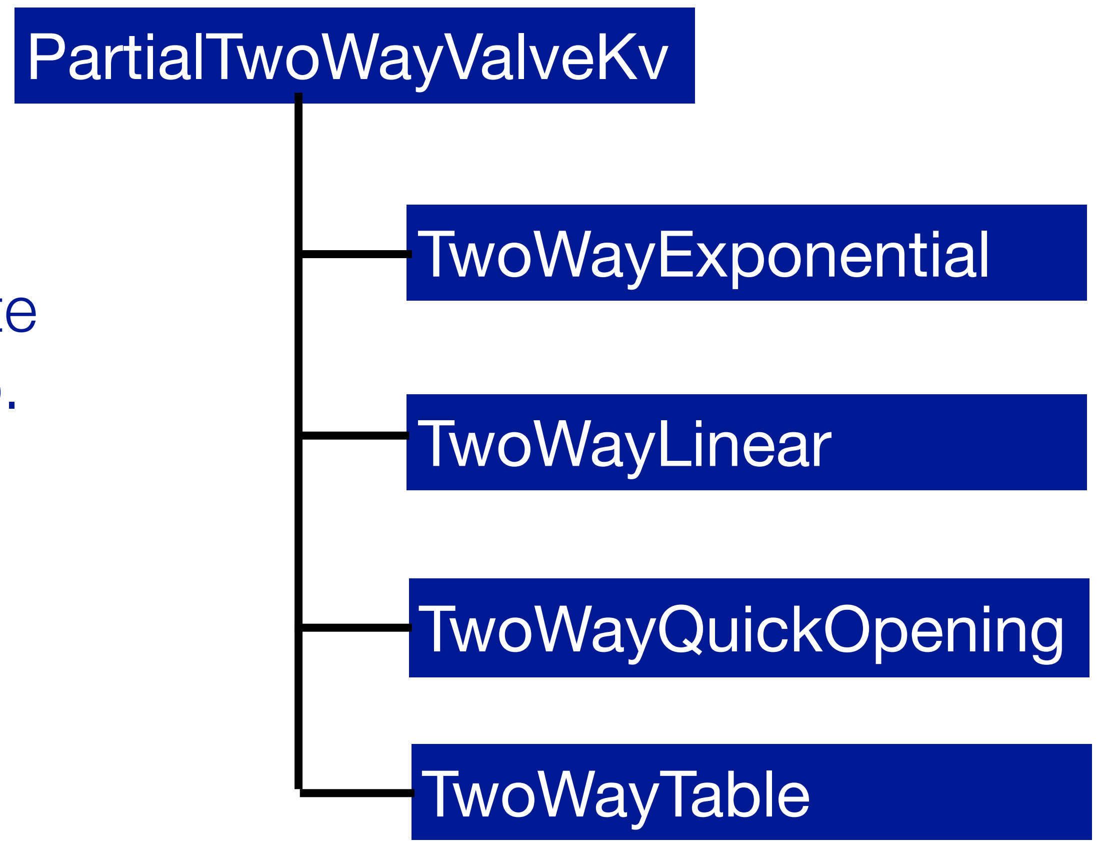
Architecture

Object-oriented component development

These valves differ only in the opening function

$$\phi(y) = \frac{k(y)}{K_v}$$

where y is the control input and k is the flow rate divided by the square root of the pressure drop.



Store all commonality in a common base class:

```
within Buildings.Fluid.Actuators.BaseClasses;
partial model PartialTwoWayValveKv
  "Partial model for a two way valve using a Kv characteristic"
  extends Buildings.Fluid.Actuators.BaseClasses.PartialTwoWayValve;

equation
 k = phi*Kv_SI;
 m_flow=BaseClasses.FlowModels.basicFlowFunction_dp(dp=dp, k=k,
                                m_flow_turbulent=m_flow_turbulent);
…
```

Object-oriented component development

Provide implementations that assign

$$\phi(y) = \frac{k(y)}{K_v}$$

and that introduce the valve-specific parameters.

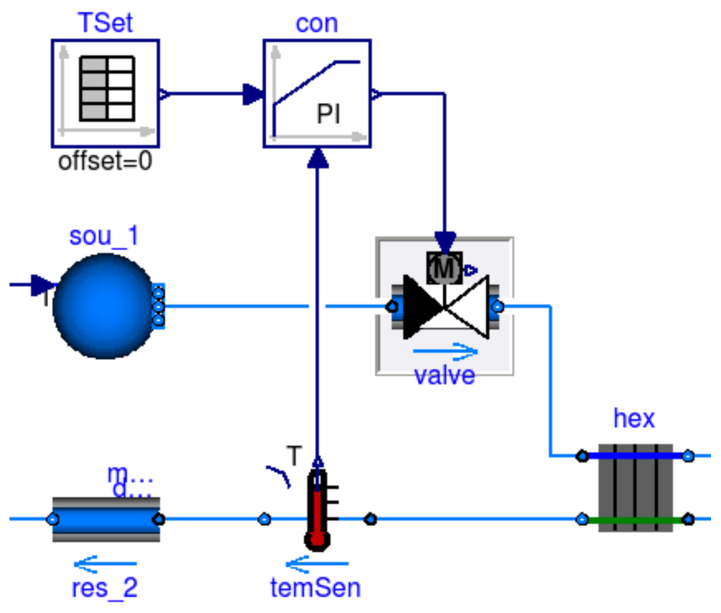
```
model TwoWayEqualPercentage
  "Two way valve with equal-percentage flow characteristics"
  extends BaseClasses.PartialTwoWayValveKv(
    phi=BaseClasses.equalPercentage(y_actual, R, l, delta0));

  parameter Real R=50 "Rangeability, R=50...100 typically";
…
end TwoWayEqualPercentage;
```

```
within Buildings.Fluid.Actuators.Valves;
model TwoWayLinear
  "Two way valve with linear flow characteristics"
  extends BaseClasses.PartialTwoWayValveKv(
    phi=l + y_actual*(1 - l));
…
end TwoWayLinear;
```

Architecture-driven modeling

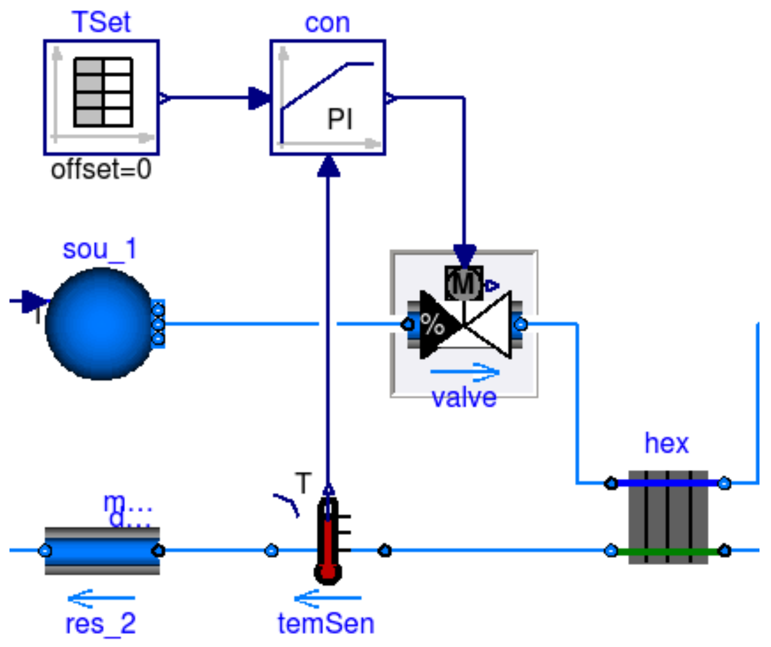
Make valve replaceable, and constrain it as desired:

```
replaceable TwoWayLinear valve
 constrainedby BaseClasses.PartialTwoWayValveKv(
  redeclare package Medium = Medium,
  from_dp=true,
  CvData=Buildings.Fluid.Types.CvTypes.Kv,
  Kv=0.65,
  m_flow_nominal=0.04)
  "Replaceable valve model”;
```



Make a new model that uses a different valve

```
within Buildings.Fluid.HeatExchangers.Examples;
model EqualPercentageValve
  "Model with equal percentage valve"
  extends DryCoilCounterFlowPControl(
  redeclare Actuators.Valves.TwoWayEqualPercentage
    valve);
end EqualPercentageValve;
```



See DataCenterDiscreteTimeControl for a model that uses this construct to change the controls, or http://book.xogeny.com/components/architectures/thermal_control/ for simple application.

Thermofluid flow modeling

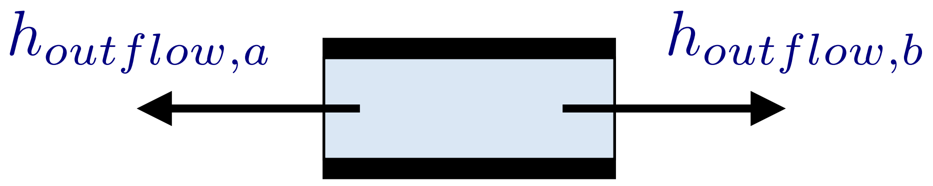
Stream of fluids

For electrical systems, we have a potential (voltage) that drives a flow (current).

For fluids, we have a potential (pressure) that drives a flow (mass flow rate) which carries properties (temperatures, mass concentration).

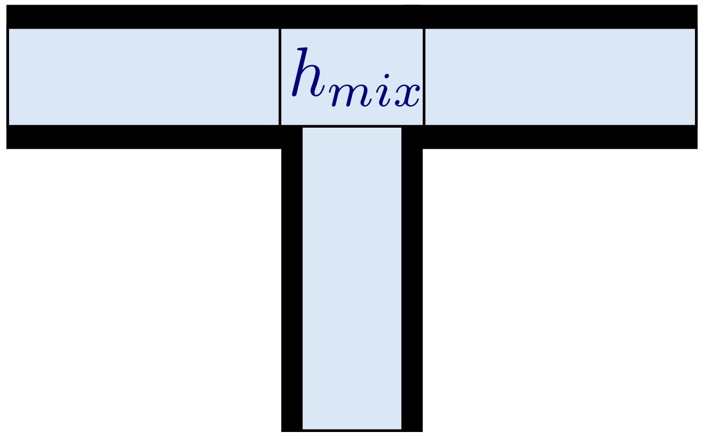
Conservation of mass

$$0 = \sum_{i=1}^{n} \dot{m}_i$$



Conservation of energy

$$0 = \sum_{i=1}^{n} \dot{m}_i \begin{cases} h_{mix}, & \text{if } \dot{m}_i > 0 \\ h_{outflow,i}, & \text{otherwise.} \end{cases}$$



If mass flow rates are computed based on (nonlinear) pressure drops, this cannot be solved reliably as the residual equation depends on a boolean variable.

For reliable solution of fluid flow network, stream variables have been introduced

Connector variables have the `flow` and `stream` attribute:

```
connector FluidPort
  replaceable package Medium =
Modelica.Media.Interfaces.PartialMedium;

  flow Medium.MassFlowRate m_flow;
  Medium.AbsolutePressure p;
  stream Medium.SpecificEnthalpy h_outflow;
  stream Medium.MassFraction Xi_outflow[Medium.nXi];
  stream Medium.ExtraProperty C_outflow[Medium.nC];
end FluidPort;
```

Thermofluid components have one balance equation for each outflowing stream:

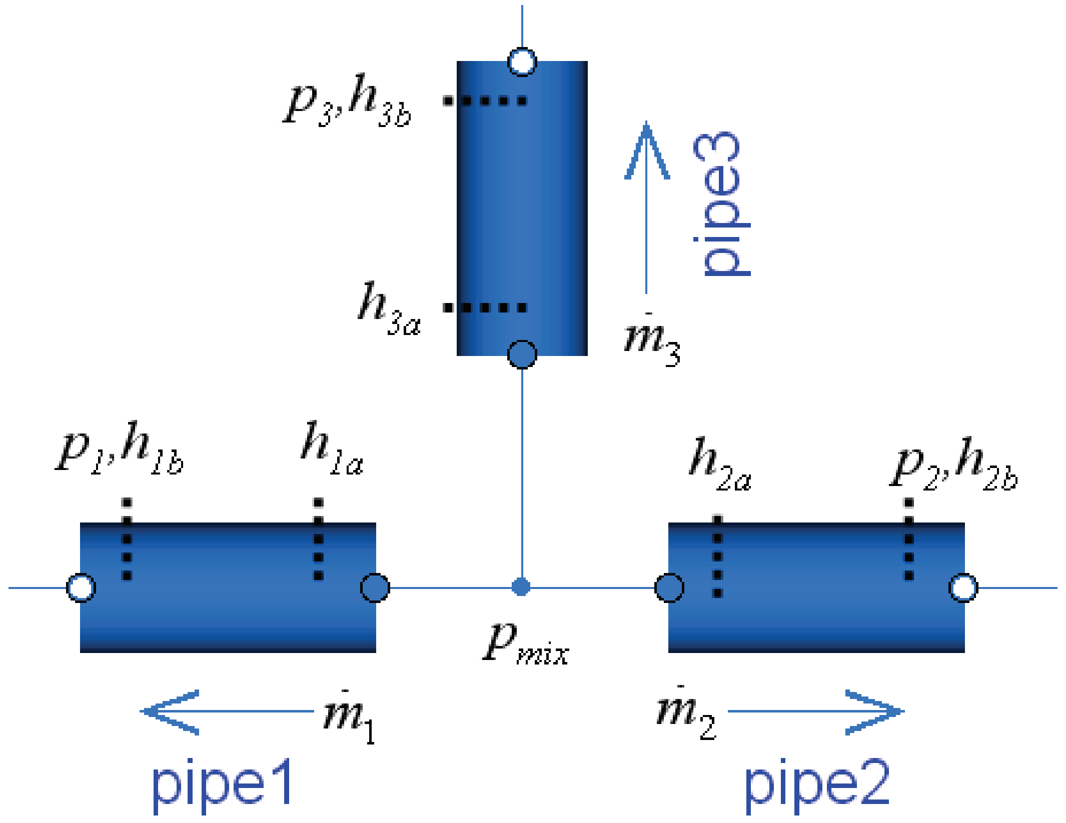
```
port_a.m_flow * (inStream(port_a.h_outflow) - port_b.h_outflow) = -Q_flow;
port_a.m_flow * (inStream(port_b.h_outflow) - port_a.h_outflow) = +Q_flow;
```

R. Franke, F. Casella, M. Otter, M. Sielemann, H. Elmqvist, S. E. Mattsson, and H. Olsson.
Stream connectors – an extension of modelica for device-oriented modeling of convective transport phenomena.
In F. Casella, editor, Proc. of the 7-th International Modelica Conference, Como, Italy, Sept. 2009.

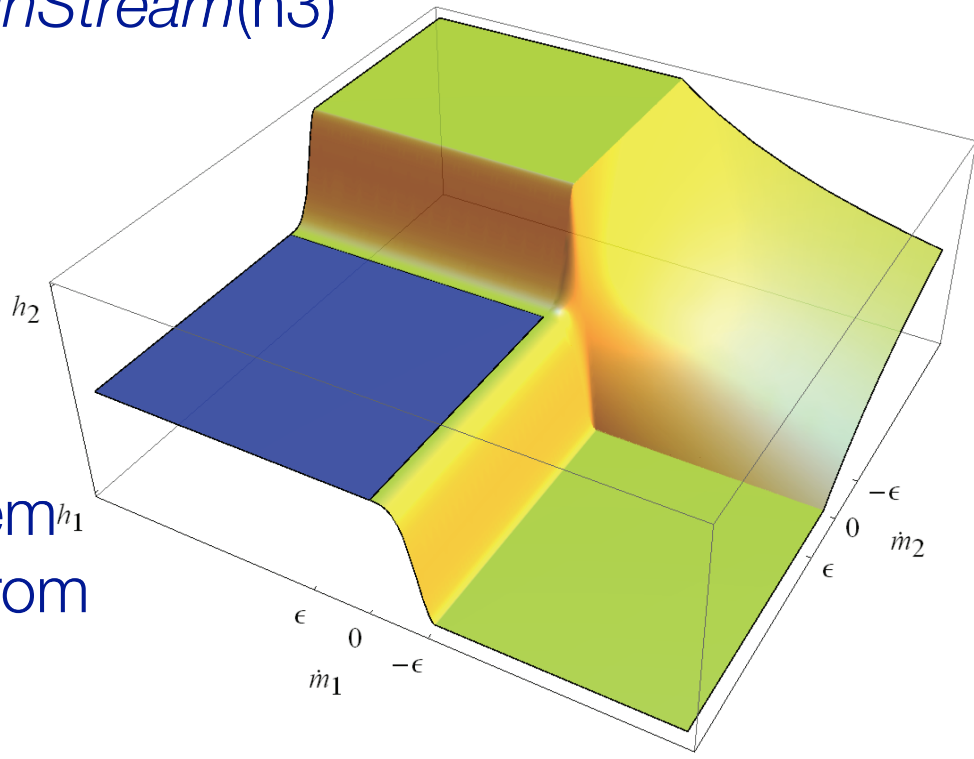
Fluid junction

Require *inStream* operator to be continuously differentiable

$$
\begin{aligned}
0 &= \dot m_1 + \dot m_2 + \dot m_3 \\
\dot m_1 &= f_1\left(p_{mix} - p_1, \rho_{1ab}, \rho_{1ba}, \eta_{1ab}, \eta_{1ba}\right) \\
\dot m_2 &= f_2\left(p_{mix} - p_2, \rho_{2ab}, \rho_{2ba}, \eta_{2ab}, \eta_{2ba}\right) \\
\dot m_3 &= f_3\left(p_{mix} - p_3, \rho_{3ab}, \rho_{3ba}, \eta_{2ab}, \eta_{2ba}\right) \\
\rho_{1ab} &= \rho(p_{mix}, T_{1a\_inflow}, \mathbf{inStream}(Xi_{1a})) \\
T_{1a\_inflow} &= T\left(p_{mix}, \mathbf{inStream}(h_{1a}), \mathbf{inStream}(Xi_{1a})\right) \\
\mathbf{inStream}(h_{1a}) &= \frac{h_{2a}\cdot max(-\dot m_2, \varepsilon) + h_{3a}\cdot max(-\dot m_3, \varepsilon)}{max(-\dot m_2, \varepsilon) + max(-\dot m_3, \varepsilon)}
\end{aligned}
$$



Suggested regularization of *inStream*(h3)



Iterate on p_{mix} and two mass flow rates.

The number of iteration variables for the above mixing problem was reduced from 22 to 3, and residual functions changed from discontinuous to continuous.

R. Franke, F. Casella, M. Otter, M. Sielemann, H. Elmqvist, S. E. Mattsson, and H. Olsson. Stream connectors – an extension of modelica for device-oriented modeling of convective transport phenomena. Proc. of the 7-th International Modelica Conference, Como, Italy, Sept. 2009.

Numerics

Initial value ordinary differential equations

Consider initial value problem

$$\begin{aligned} \frac{dx(t)}{dt} &= f(x(t)), \qquad t \in [0,\, 1] \\ x(0) &= x_0 \end{aligned}$$

A unique solution exists if $f(\cdot)$ and $\partial f(\cdot)/\partial t$ are Lipschitz continuous.

If your model does not satisfy these properties, what can you do if a solver does not converge, or gives unexpected results? Is it a problem of the model or the solver?

A model that we often see implemented but can cause Newton to fail

Consider the mass flow relation $\dot{V} = sign(\Delta p)\, k\, \sqrt{|\Delta p|}$

Suppose this expression is part of an algebraic loop that is solved for the pressure *p* using a Newton algorithm.

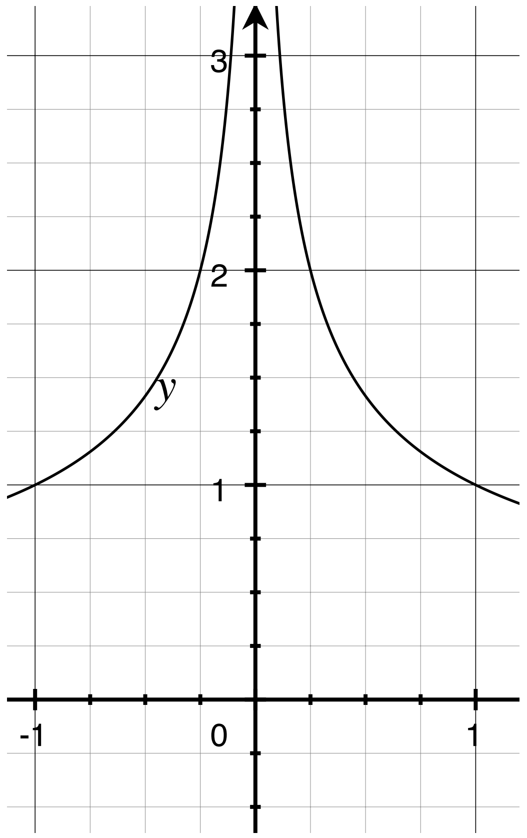
Let $g(p) = 0$ be the residual function.

Newton will iterate using

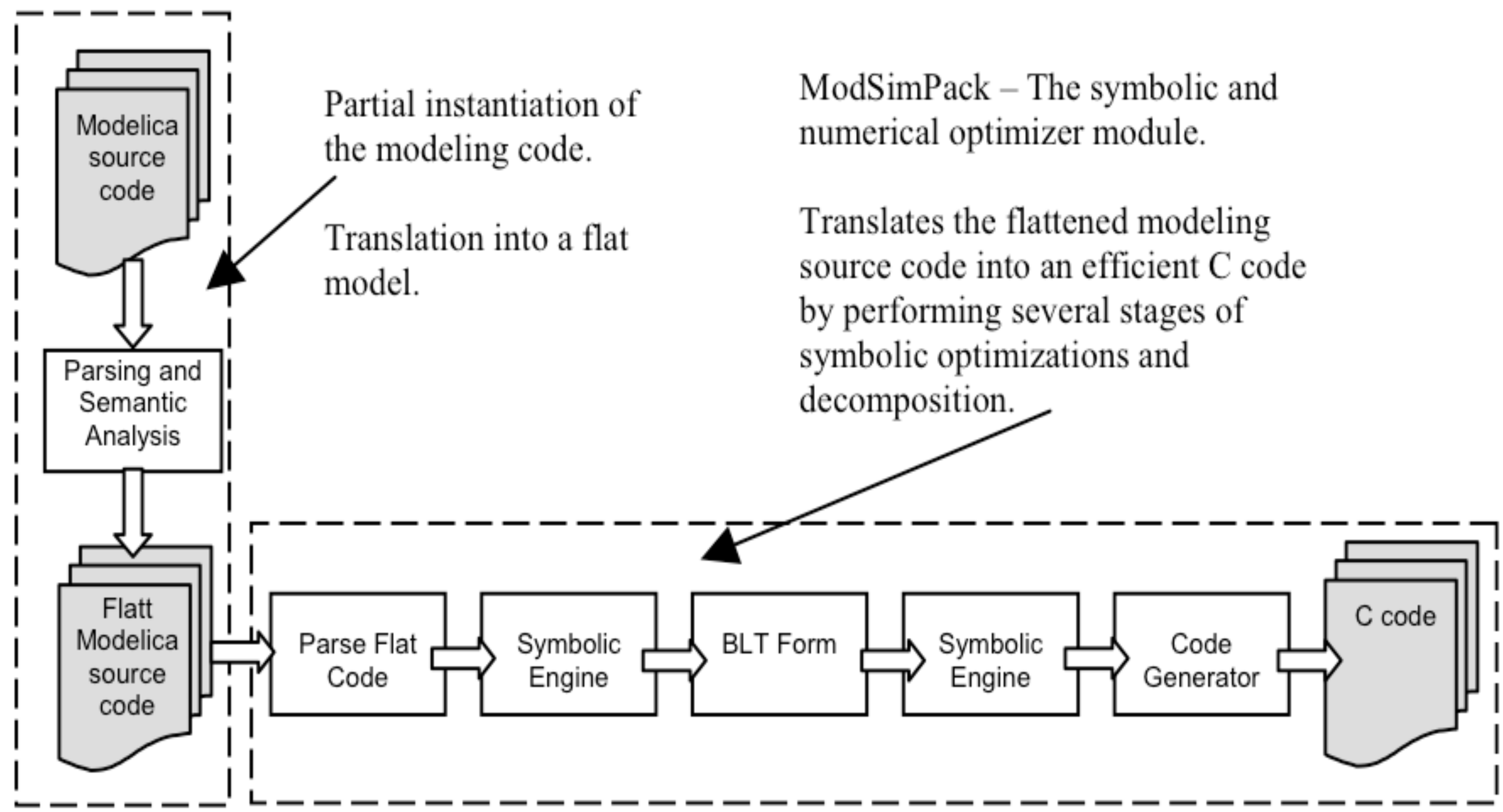
$$p_{k+1} = p_k - \frac{g(p_k)}{\partial g(p_k)/\partial p}$$

until a convergence criteria on p_k is met.

The denominator tends to infinity, and hence the Newton step becomes arbitrarily small.



Translation process (figure from OpenModelica)



Modelica source code
Parsing and Semantic Analysis
Flatt Modelica source code
Partial instantiation of the modeling code.
Translation into a flat model.
ModSimPack – The symbolic and numerical optimizer module.
Translates the flattened modeling source code into an efficient C code by performing several stages of symbolic optimizations and decomposition.
Parse Flat Code
Symbolic Engine
BLT Form
Symbolic Engine
Code Generator
C code

Figure from Bunus and Fritzson (2004)

Simplifying the equations through block lower triangularization and tearing:
(i) Block lower triangularization

$$1: \quad Q = T_1 - T_2$$

$$2: \quad 0 = T_1$$

$$3: \quad f(t) = T_1 + T_2$$

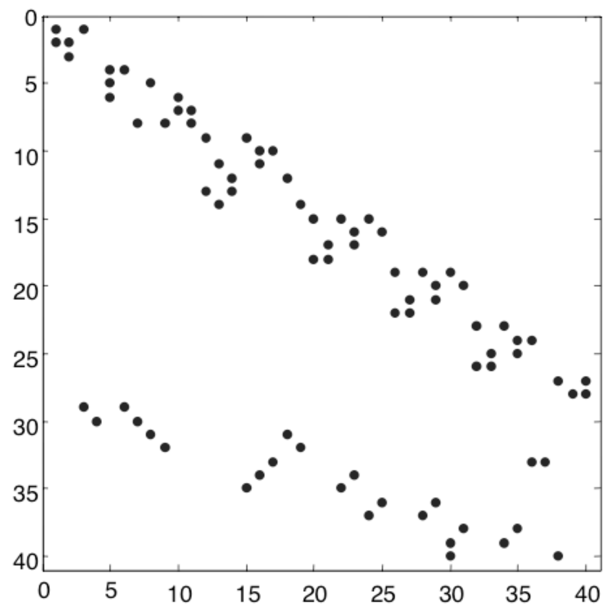
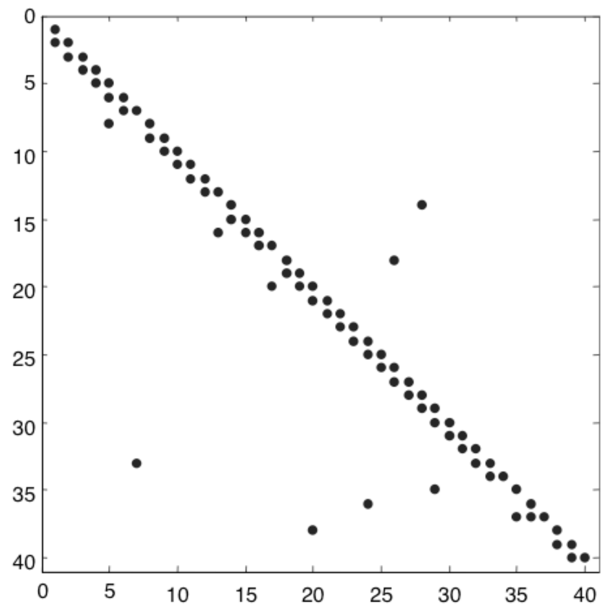
Incidence matrix

	T1	T2	Q
1	■	■	■
2	■		
3	■	■	

After BLT transformation

	T1	T2	Q
2	■		
3	■	■	
1	■	■	■

Application to an electric circuit model

Bunus and Fritzson (2004)

(ii) Tearing

Suppose we have an equation

$$0 = f(x),\ \ x \in \Re^n,\ \ n > 1,$$

that can be written in the form

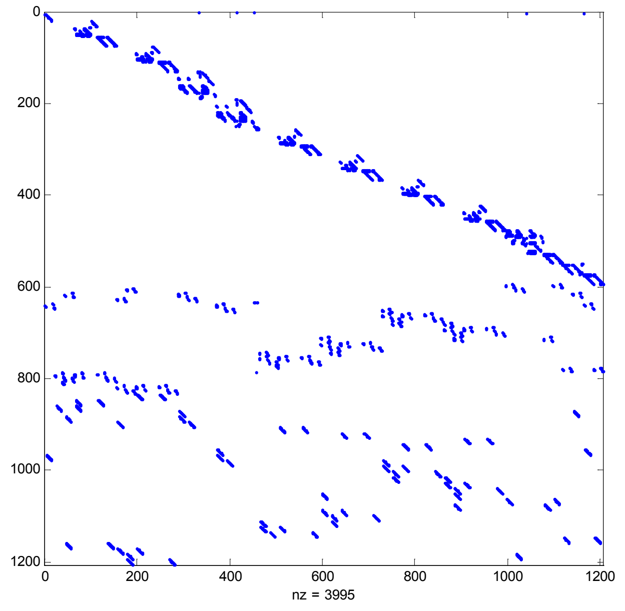
$$(1)\quad L\,x^1 = \hat{f}^1\left(x^2\right),$$

$$(2)\qquad\quad 0 = \hat{f}^2\left(x^1, x^2\right),$$

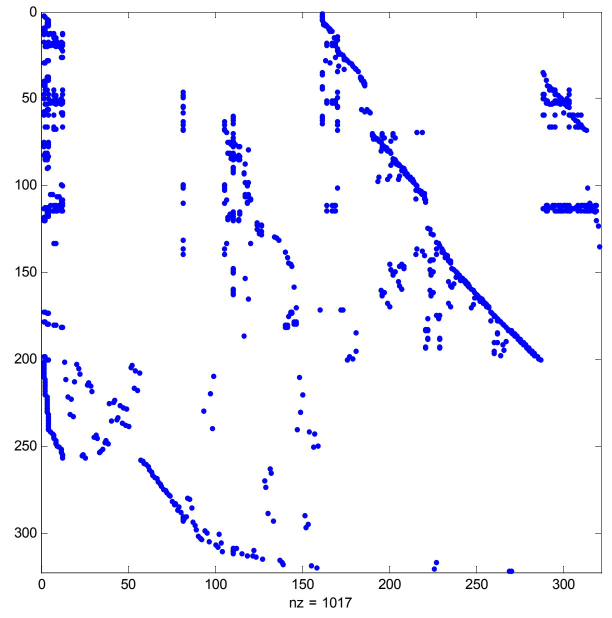
where L is a lower triangular matrix with constant non-zero diagonals.
How do you solve this efficiently?

Pick a guess value for x^2, solve (1) for x^1, and compute a new value for x^2 from (2).
Iterate until x^2 converges to a solution.

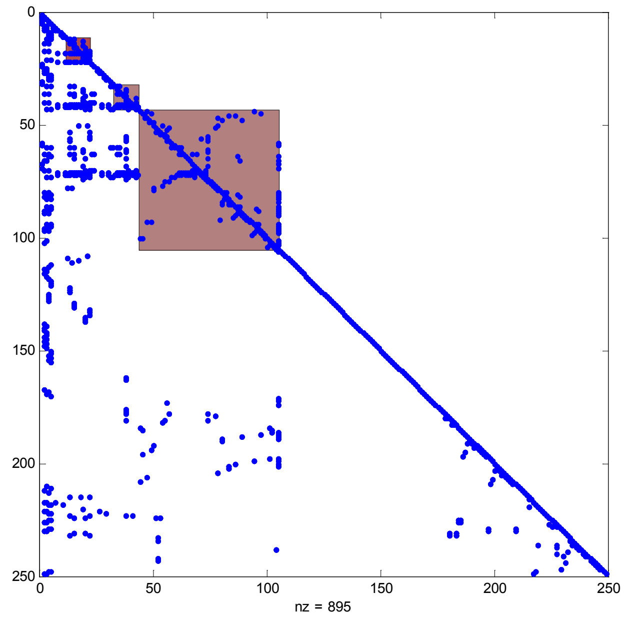
Symbolic manipulations significantly reduce problem size



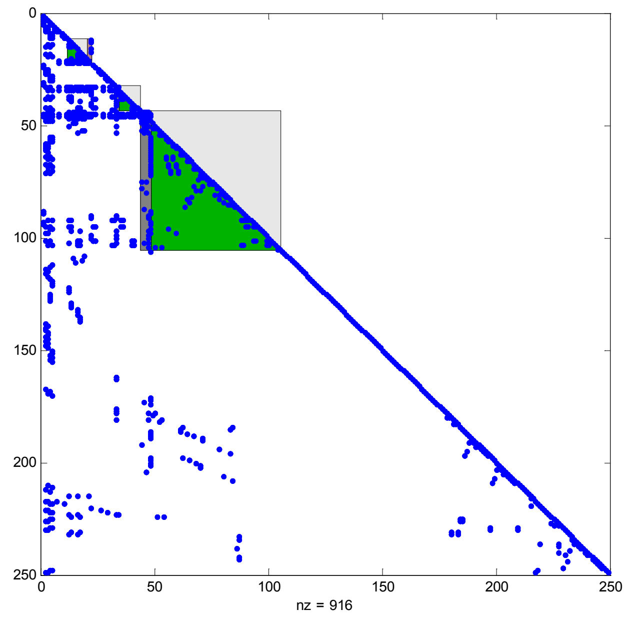
Incidence matrix of the original problem (1200x1200)



Incidence matrix after elimination of alias variables (330x330)



After simplifications and BLT (250x250)



Tearing reduces nonlinear 12 to 2, linear 11 to 2 and linear 57 to 5.

Note: Many numerical algorithms are $O(n^3)$

Figures from Dymola 2016 user manual for mechanical model with kinematic loop. For description of method, see Cellier and Kofman, Continuous System Simulation, Springer, 2006.

Exercise

Exercise: Heat conduction with feedback control loop

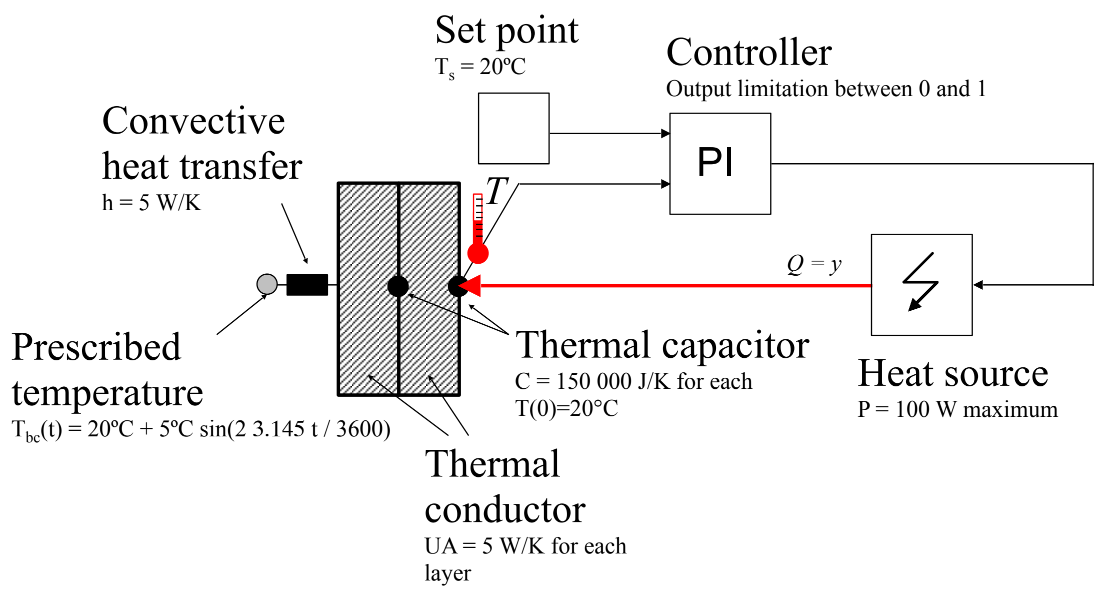


Figure 1: Schematics of the heat conduction problem.

Consider the system shown in Figure 1 that consists of a heat conductor in which one temperature is controlled by injecting heat. This model corresponds to a 1m^2 area of a brick wall that is 0.2 m thick and has the same boundary condition on both sides. Heat is injected in the middle of the construction. Because of symmetry, only half of the construction needs to be modeled.

Open Loop Response

The first assignment is to create a model of the open loop system and simulate the open loop response in a Modelica environment.[1]

[1] Hint: This model can be assembled graphically in OpenModelica using models from the Modelica Standard Library.

Add Feedback Control

The control objective is the keep the temperature in the core of the heat conductor above 20°C. Use a PI controller with output limitation between 0 and 1 and anti-windup from the Modelica Standard Library. You will need to simulate the model for more than one day. Explain why one day is not sufficient even though the disturbance has a periodicity of one day.

Questions