Modelica IBPSA Tutorial

Michael Wetter
Berkeley Lab

… and many contributors to the library

October 9, 2023

Agenda

Overview of the library

Structure

Best practices and modeling hints

Hands-on tutorial

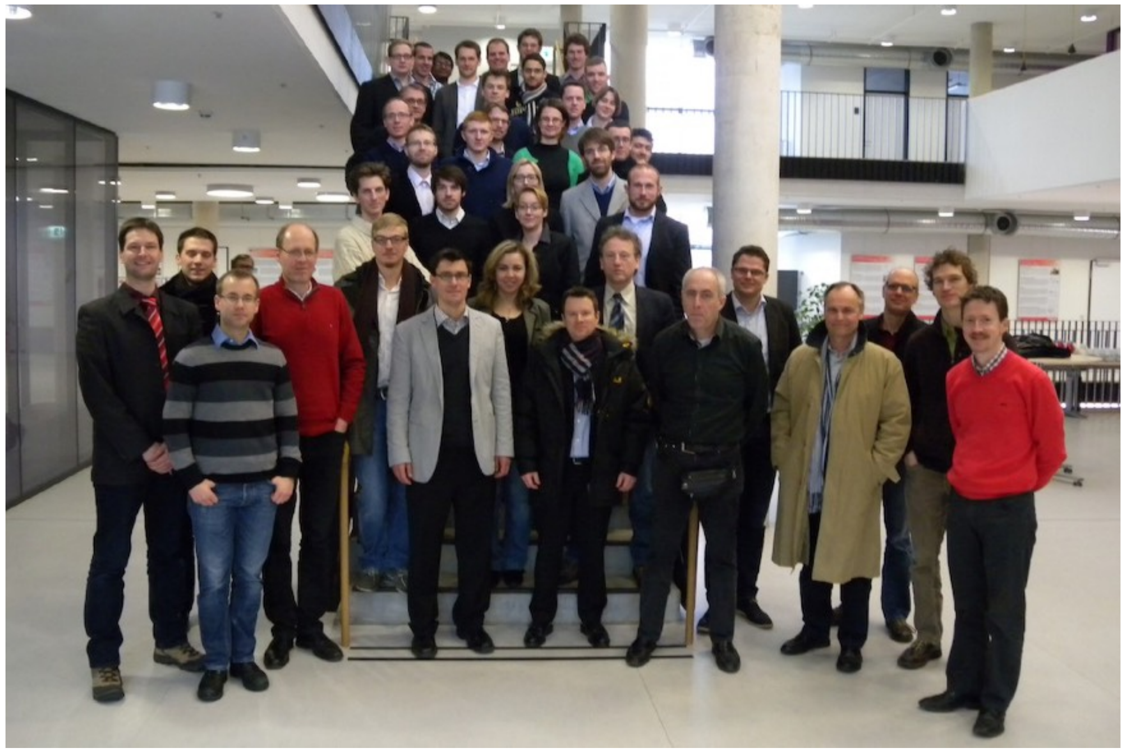
Modelica IBPSA Library Overview

Primary use of Modelica IBPSA Library

- Model repository for building and district energy simulation, to be used as the core of
  - AixLib
  - BuildingSystems
  - Buildings
  - IDEAS

- License
  - All development is open-source under BSD.

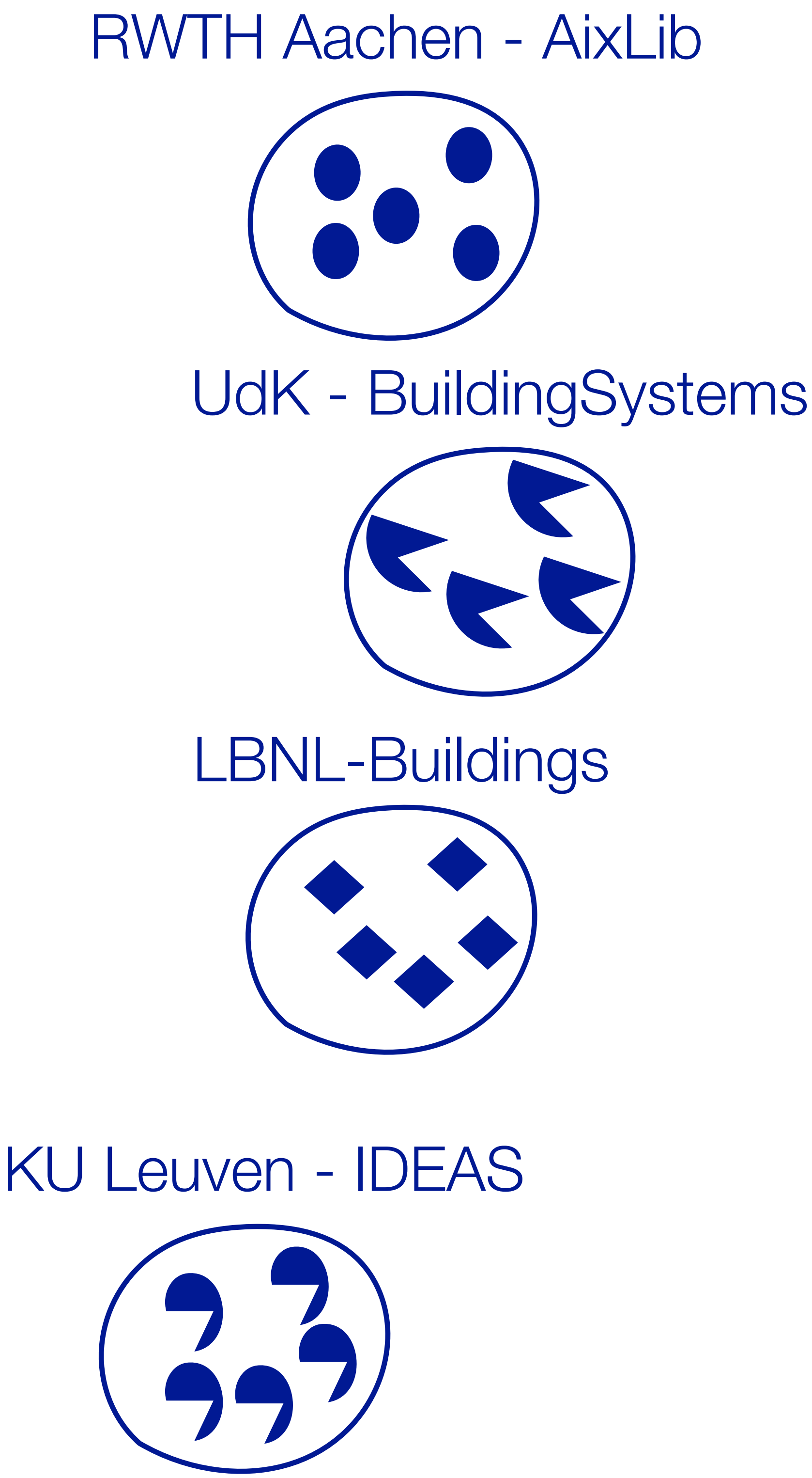
In 2013, a joint effort started to avoid fragmentation, collaborate on development, implement best practices and share everything open-source and free



Attendees of the Annex 60 planning meeting at RWTH Aachen, March 11-13, 2013

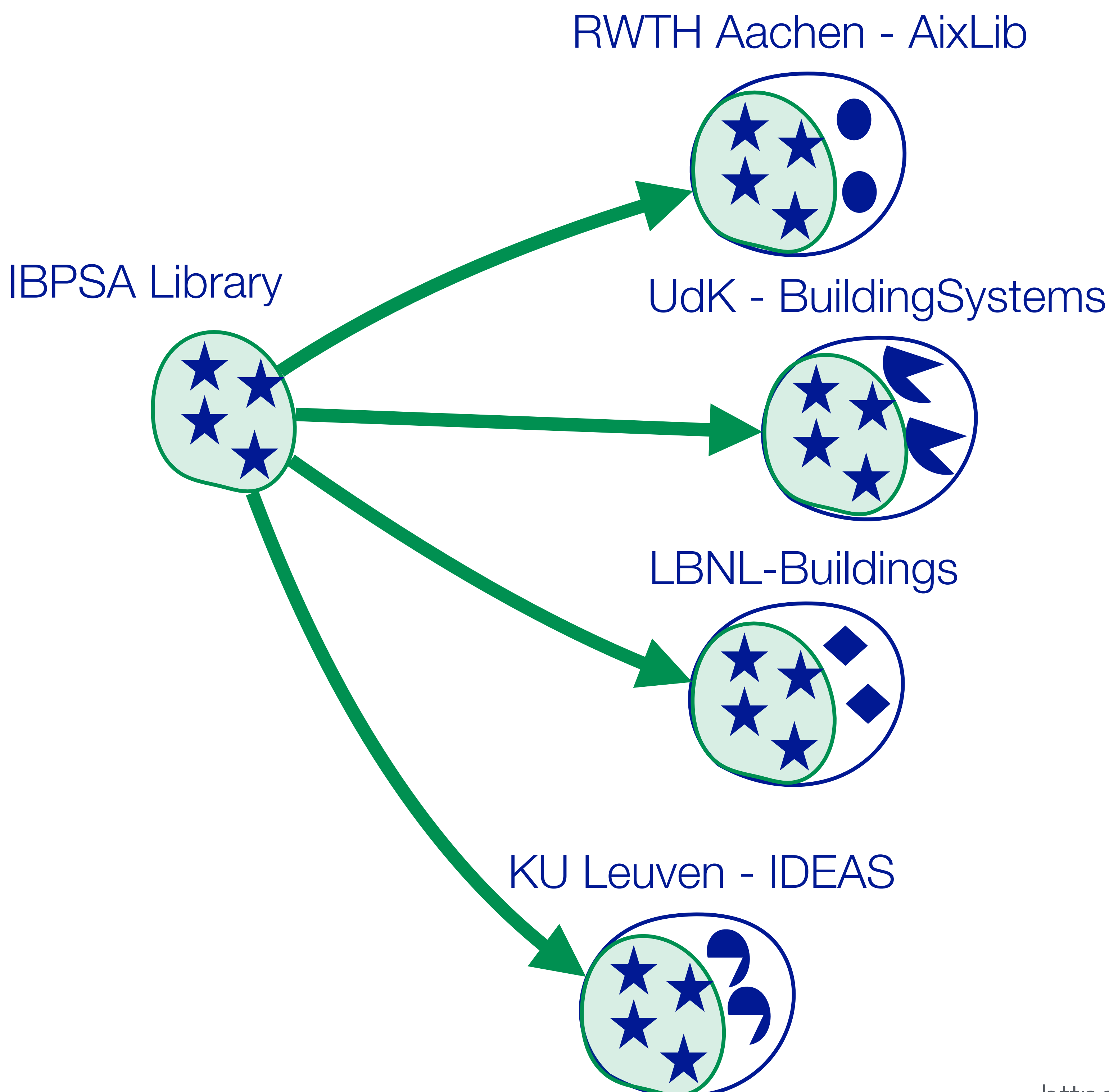
Attendees of the first IBPSA Project 1 Expert Meeting at UdK Berlin, February 27-28, 2018

In 2013, Modelica for buildings was very fragmented. Libraries were incompatible, they replicated each other and best practices were not understood

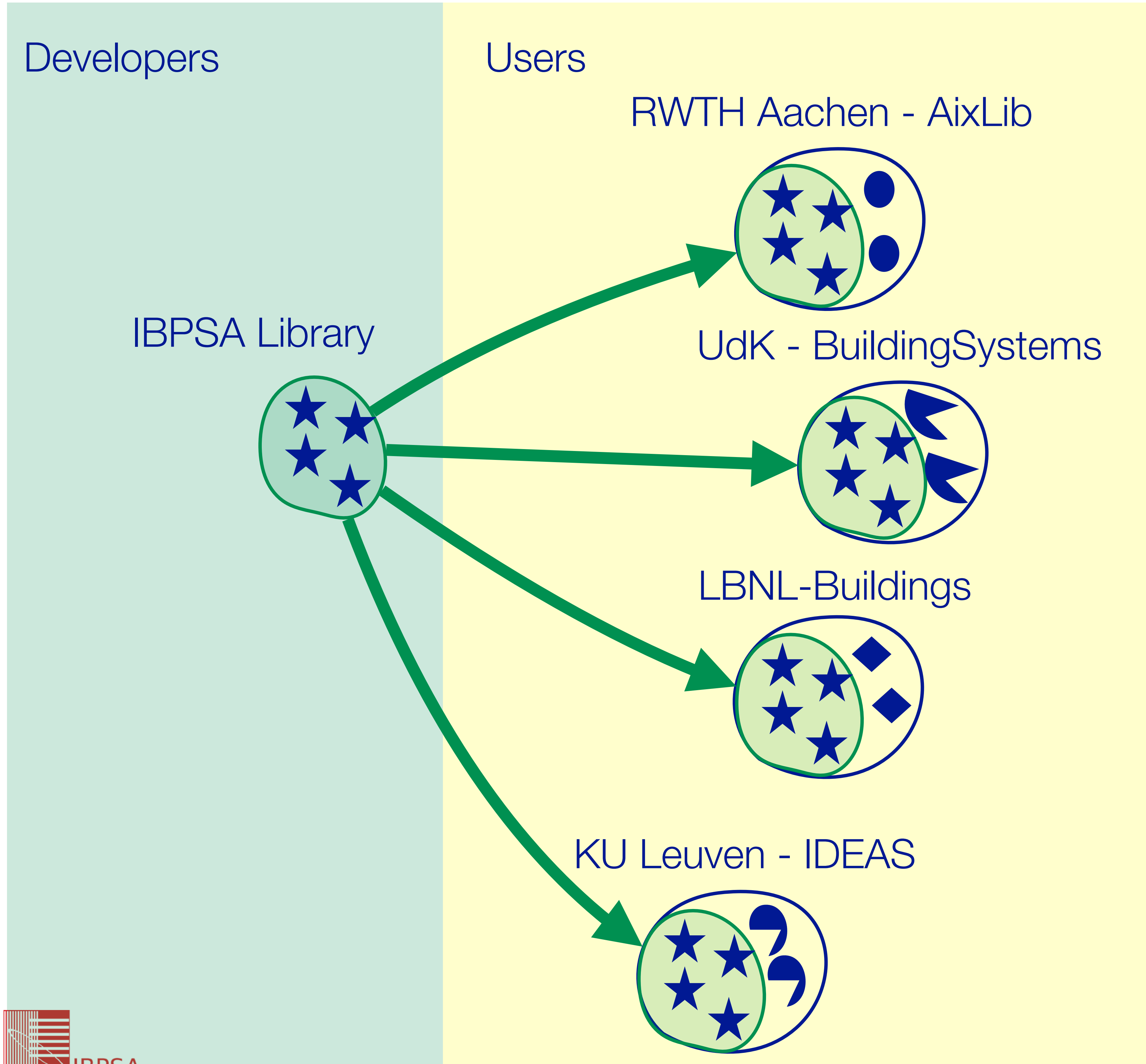


IBPSA

In 2013, a joint effort started to avoid fragmentation, collaborate on development, implement best practices and share everything open-source and free



Users will use derivative Modelica libraries that contain IBPSA, or tools that package these derivative libraries

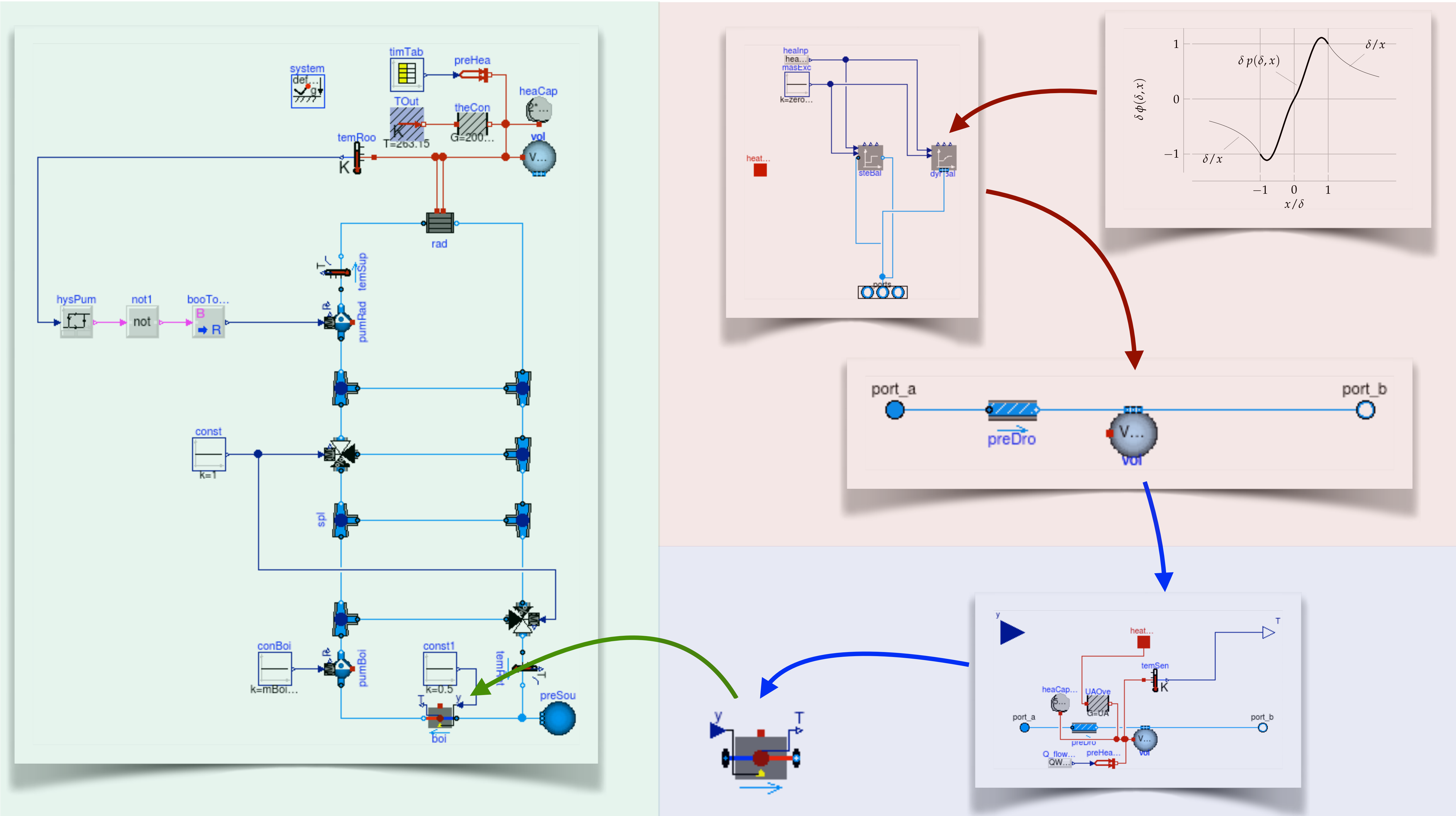


While this tutorial uses the IBPSA library, don't use it for your project work!

Instead, use any or several of the user-facing libraries.

They have all functionality of the IBPSA library, plus much more.

Separation between library developer, component developer and end user



Legend:

Library developer

Component developer

End user

Main modeling assumptions

Media	Can track moisture (X) and contaminants (C).
HVAC equipment	Most equipment based on performance curve, or based on nominal conditions and similarity laws. Refrigerant is not modeled. Most equipment optional steady-state or 1st order transient.
Flow resistances	Based on m_flow_nominal and dp_nominal plus similarity law. Optional flag to linearize or to set dp=0.
Room model	Any number of constructions are possible. Layer-by-layer window model (similar to Window 6). Optional flag to linearize radiation and/or convection.
Electrical systems	DC. AC 1-phase and 3-phase (dq, dq0). Quasi-stationary or dynamic phase angle (but not frequency).

Validation

All components are verified with analytical solutions, comparative model validation, or against guidelines such as from VDI.

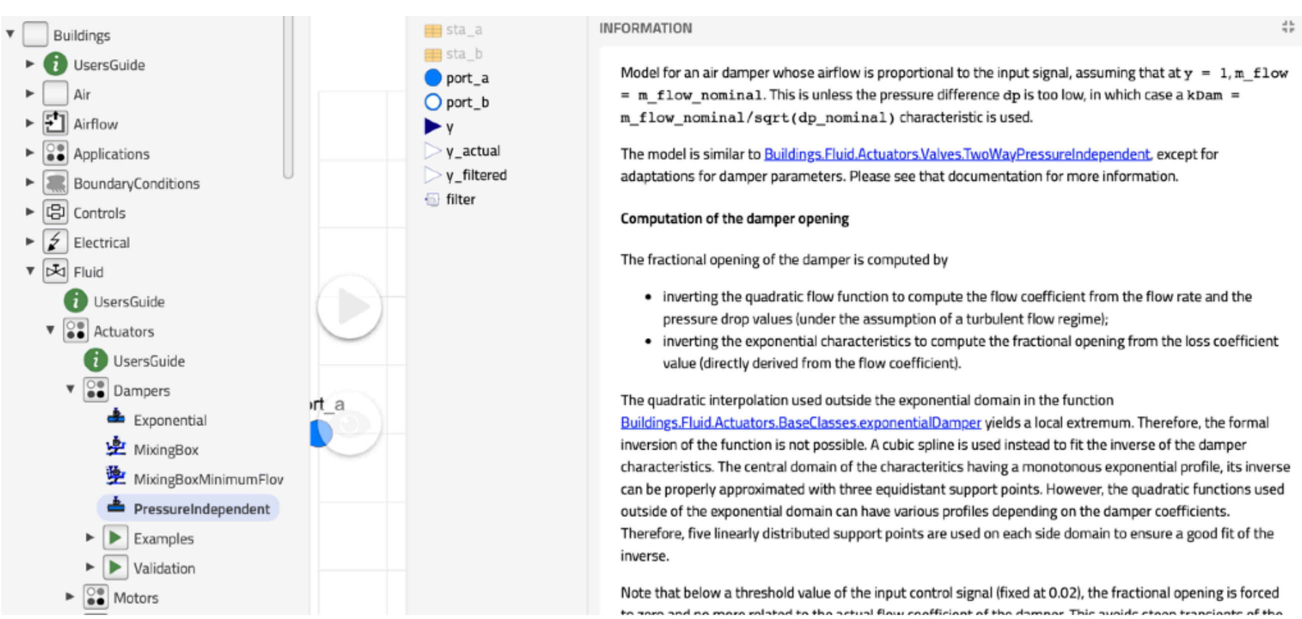
600+ regression tests compare results to reference results as part of development, see https://github.com/ibpsa/modelica-ibpsa/wiki/Unit-Tests

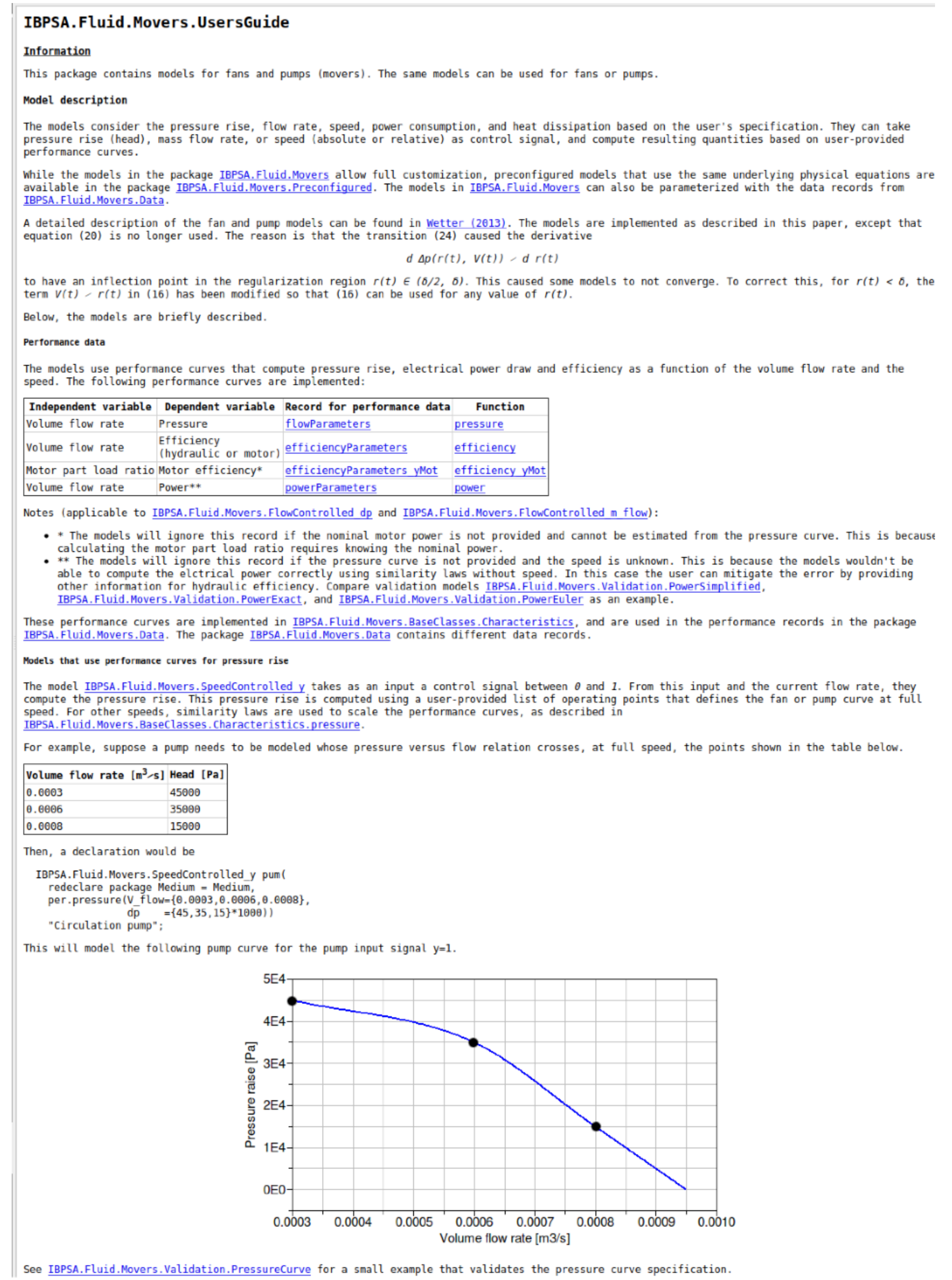
Structure of the library

Documentation and distribution

Documentation

- All models contain an “info” section.
- Various models contain users’ guide.
- Various models in Examples and Validation packages illustrate model use.
- Derivative libraries contain additional documentation.





IBPSA.Fluid.Movers.UsersGuide

Information

This package contains models for fans and pumps (movers). The same models can be used for fans or pumps.

Model description

The models consider the pressure rise, flow rate, speed, power consumption, and heat dissipation based on the user's specification. They can take pressure rise (head), mass flow rate, or speed (absolute or relative) as control signal, and compute resulting quantities based on user-provided performance curves.

While the models in the package IBPSA.Fluid.Movers allow full customization, preconfigured models that use the same underlying physical equations are available in the package IBPSA.Fluid.Movers.Preconfigured. The models in IBPSA.Fluid.Movers can also be parameterized with the data records from IBPSA.Fluid.Movers.Data.

A detailed description of the fan and pump models can be found in Wetter (2013). The models are implemented as described in this paper, except that equation (20) is no longer used. The reason is that the transition (24) caused the derivative

$$d\,\Delta p(r(t),\ V(t)) / d\,r(t)$$

to have an inflection point in the regularization region $r(t) \in (\delta/2,\ \delta)$. This caused some models to not converge. To correct this, for $r(t) < \delta$, the term $V(t) / r(t)$ in (16) has been modified so that (16) can be used for any value of $r(t)$.

Below, the models are briefly described.

Performance data

The models use performance curves that compute pressure rise, electrical power draw and efficiency as a function of the volume flow rate and the speed. The following performance curves are implemented:

Independent variable	Dependent variable	Record for performance data	Function
Volume flow rate	Pressure	flowParameters	pressure
Volume flow rate	Efficiency (hydraulic or motor)	efficiencyParameters	efficiency
Motor part load ratio	Motor efficiency*	efficiencyParameters_yMot	efficiency_yMot
Volume flow rate	Power**	powerParameters	power

Notes (applicable to IBPSA.Fluid.Movers.FlowControlled_dp and IBPSA.Fluid.Movers.FlowControlled_m_flow):

- * The models will ignore this record if the nominal motor power is not provided and cannot be estimated from the pressure curve. This is because calculating the motor part load ratio requires knowing the nominal power.
- ** The models will ignore this record if the pressure curve is not provided and the speed is unknown. This is because the models wouldn't be able to compute the elctrical power correctly using similarity laws without speed. In this case the user can mitigate the error by providing other information for hydraulic efficiency. Compare validation models IBPSA.Fluid.Movers.Validation.PowerSimplified, IBPSA.Fluid.Movers.Validation.PowerExact, and IBPSA.Fluid.Movers.Validation.PowerEuler as an example.

These performance curves are implemented in IBPSA.Fluid.Movers.BaseClasses.Characteristics, and are used in the performance records in the package IBPSA.Fluid.Movers.Data. The package IBPSA.Fluid.Movers.Data contains different data records.

Models that use performance curves for pressure rise

The model IBPSA.Fluid.Movers.SpeedControlled_y takes as an input a control signal between *0* and *1*. From this input and the current flow rate, they compute the pressure rise. This pressure rise is computed using a user-provided list of operating points that defines the fan or pump curve at full speed. For other speeds, similarity laws are used to scale the performance curves, as described in IBPSA.Fluid.Movers.BaseClasses.Characteristics.pressure.

For example, suppose a pump needs to be modeled whose pressure versus flow relation crosses, at full speed, the points shown in the table below.

Volume flow rate [m^3/s]	Head [Pa]
0.0003	45000
0.0006	35000
0.0008	15000

Then, a declaration would be

```
IBPSA.Fluid.Movers.SpeedControlled_y pum(
  redeclare package Medium = Medium,
  per.pressure(V_flow={0.0003,0.0006,0.0008},
               dp    ={45,35,15}*1000))
  "Circulation pump";
```

This will model the following pump curve for the pump input signal y=1.

5E4
4E4
3E4
2E4
1E4
0E0
Pressure raise [Pa]
0.0003 0.0004 0.0005 0.0006 0.0007 0.0008 0.0009 0.0010
Volume flow rate [m3/s]

See IBPSA.Fluid.Movers.Validation.PressureCurve for a small example that validates the pressure curve specification.

IBPSA

Organization of the main packages

Packages are typically structured as shown on the right.

To add a new class, look first at `Interfaces` and `BaseClasses`.

You probably will never implement a thermofluid flow component without extending a base class from `IBPSA.Fluid.Interfaces`

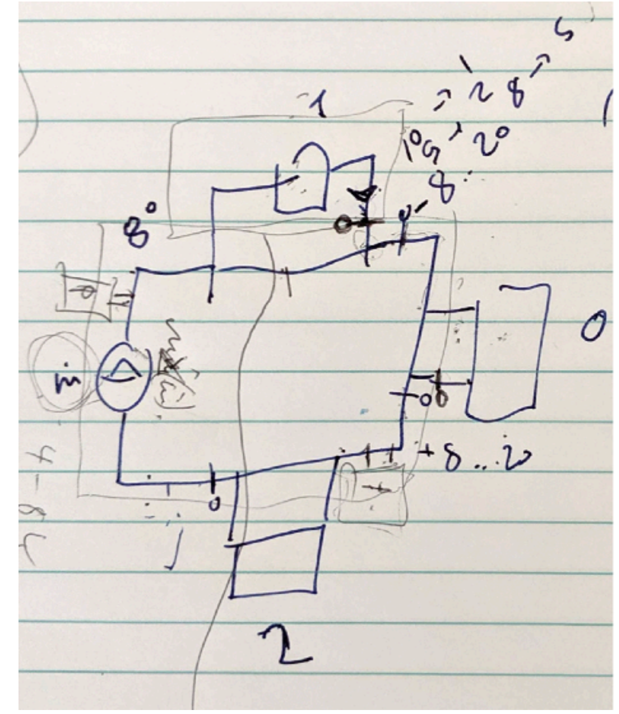
```
IBPSA
  Airflow
    Multizone
  BoundaryConditions
    SolarGeometry
    SolarIrradiation
    SkyTemperature
    WeatherData
  Controls
    Continuous
    Discrete
    SetPoints
  Electrical
    {AC, DC}
  Fluid
    Actuators
    Boilers
    Chillers
    FixedResistances
    HeatExchangers
  Examples
    …
  ThermalZones
    ISO13790
    ReducedOrder
  Utilities
    Diagnostics
    Math
    Psychrometrics

  Resources
    Bin
    C-Sources
    Data
    Documentation
    Images
    Library
    ReferenceResults
    Scripts
    src
    weatherdata
```

Best practice
and
modeling hints

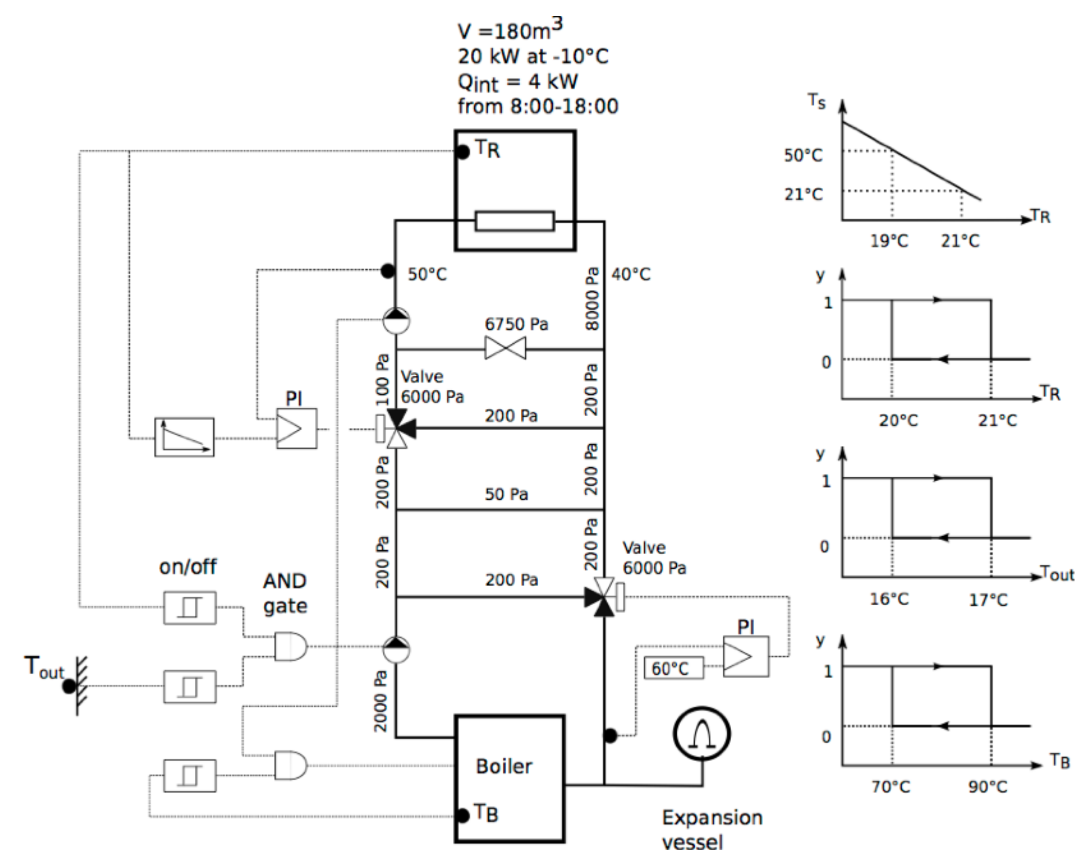
Building large system models

1. **Understand the problem**:
    1. What question do you want to answer?
    2. Know what you want to model.
        1. Draw system schematics.
        2. Identify control input.
        3. Draw the control loops.
        4. Determine the control sequences.
2. **Compartmentalize**:
   Split the system into subcomponents that can be tested in isolation.
3. **Implement**:
   Now, <u>and only now</u>, start implementing in software.
    1. Document and build test cases as you go along.

       Errors are easy to detect in small models, but hard in large models. If you add unit tests, you make sure what has been tested remains intact as the model evolves.
    2. Assemble the subcomponents to build the full model.
    3. Don't copy-paste models, you or your collaborator will regret it…
       Use version control, model instances, `extend`, `replaceable`, …



Iterate using hand-sketch of hydraulics and controls.



Draw diagram, including control loops — even a hand-drawing safes time and increases quality.

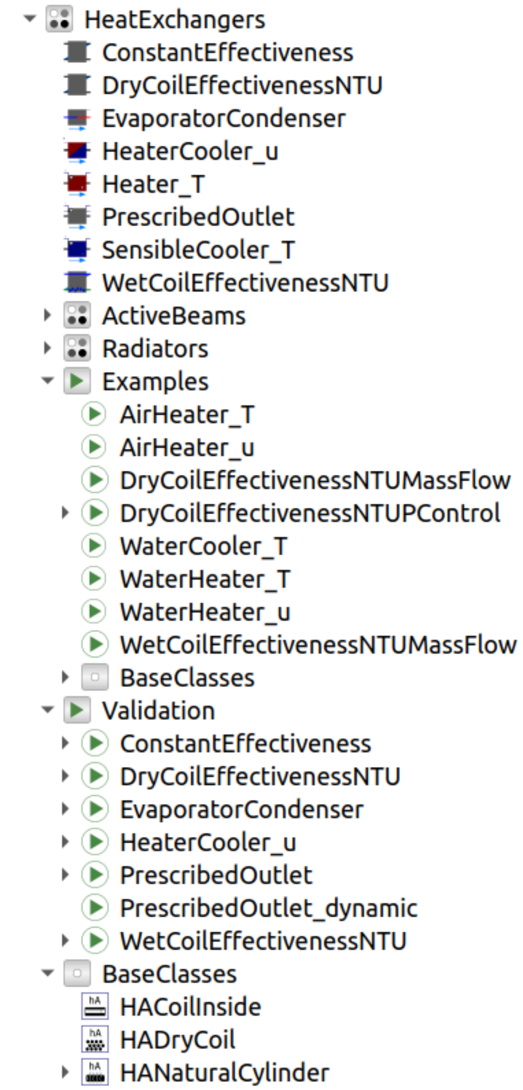
IBPSA

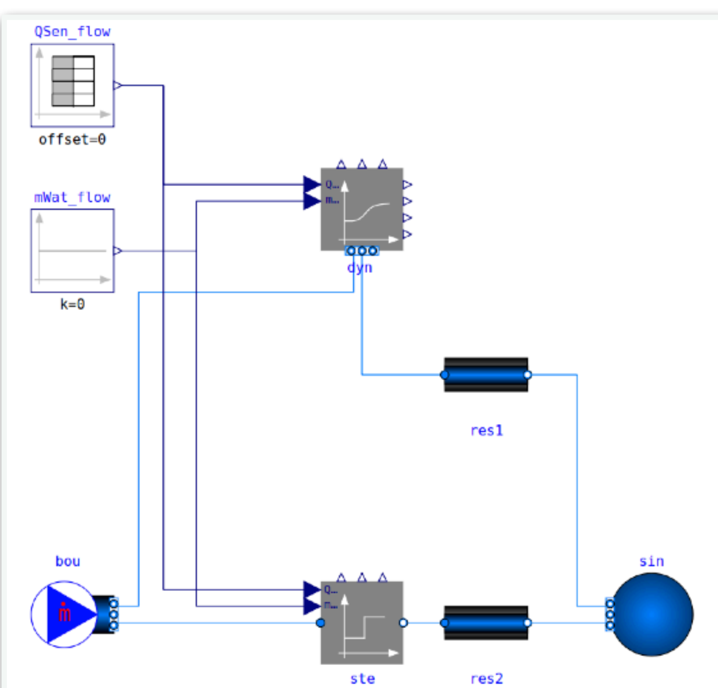
Building large system models

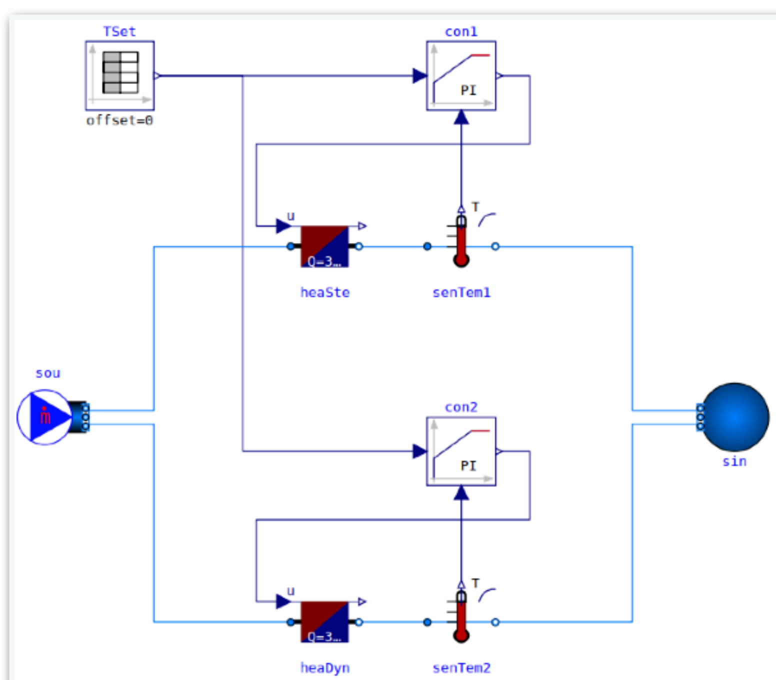
How do you build and debug a large system model?

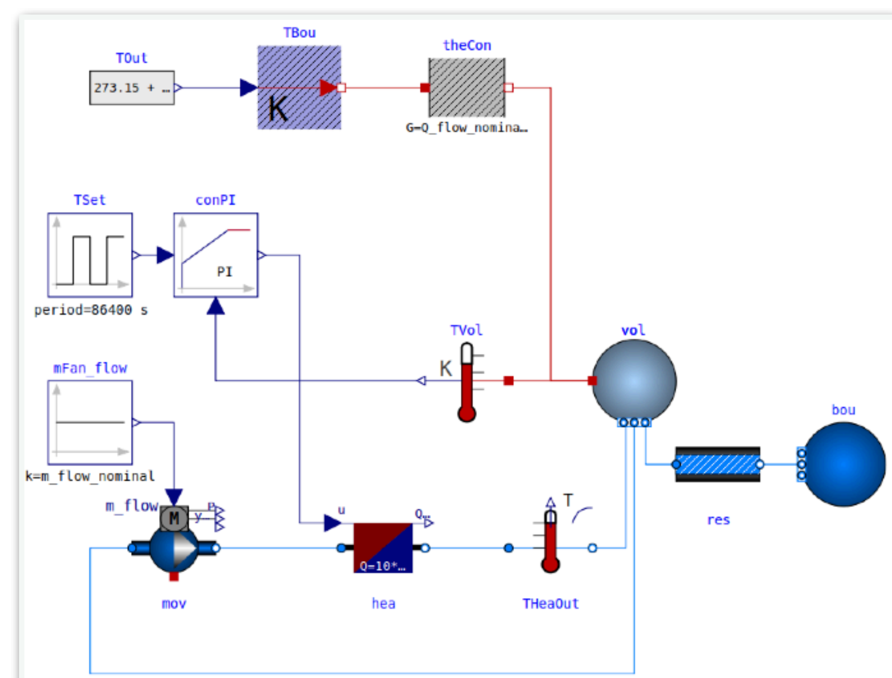
1. Split the model into small models — or better, architect the large model from the beginning to be based on smaller models
2. Test the smaller models for well known conditions.
3. Add smaller models to unit tests.

For example, see IBPSA.Fluid.HeatExchangers, in which each model contains a simple unit test, and components contain their own tests.





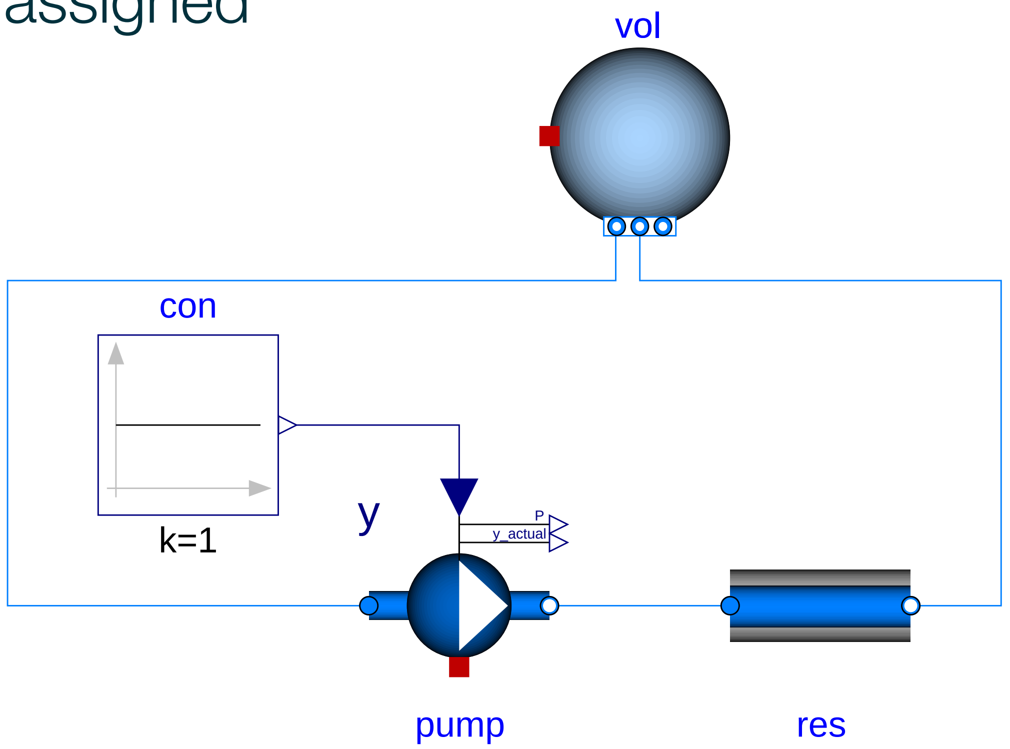




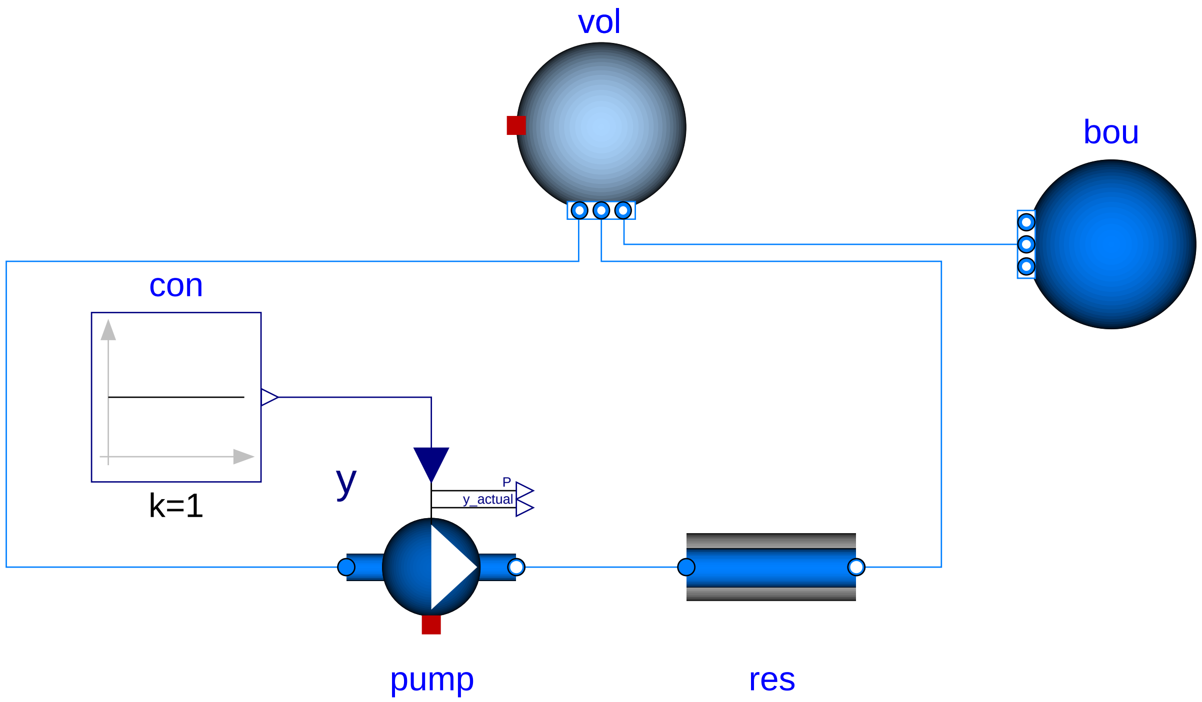
IBPSA

All system models must have a reference pressure

Underdetermined model as no pressure state is assigned



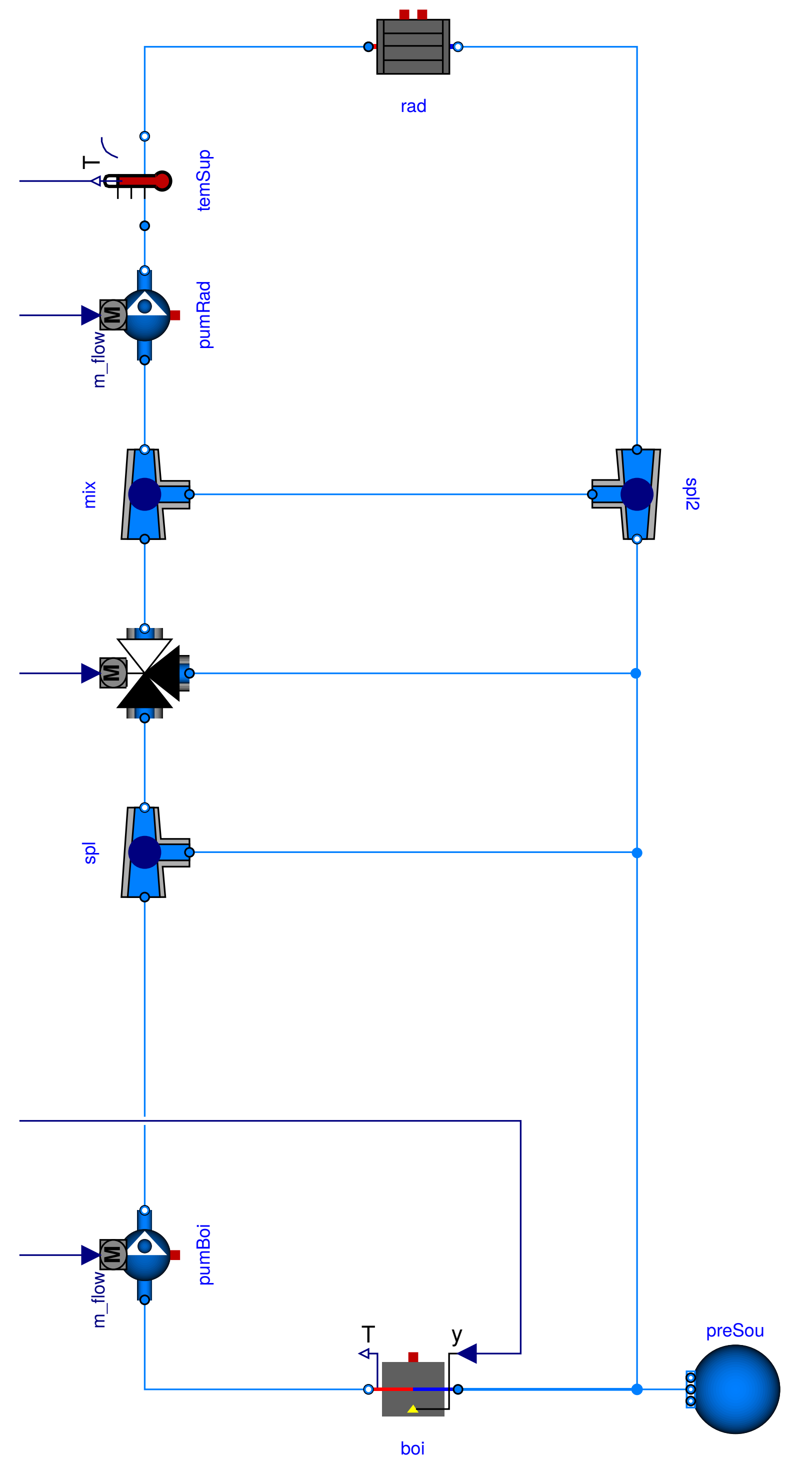
Model that provides a reference presssure through the instance **bou**.



IBPSA

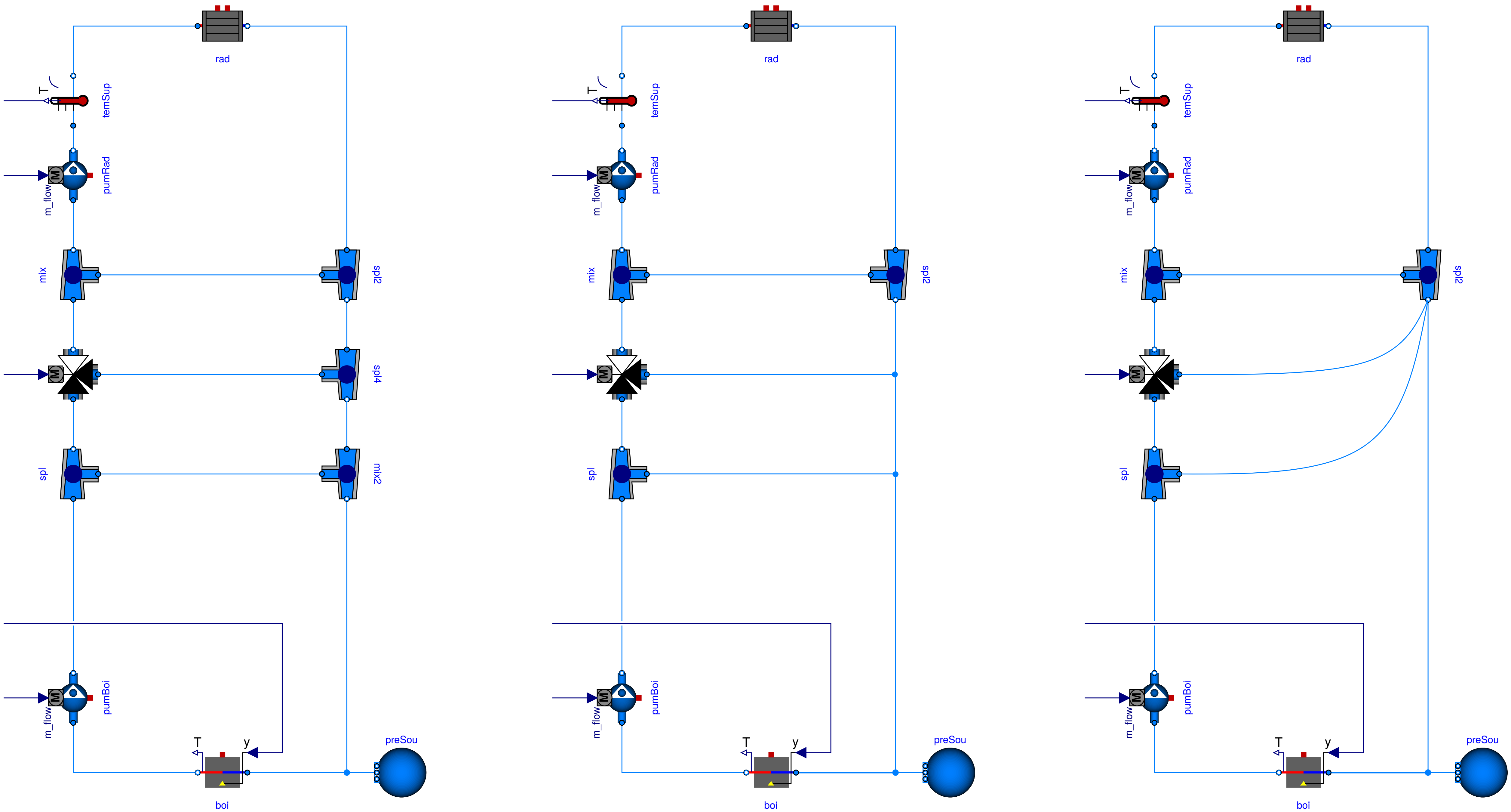
Modeling of fluid junctions

What is wrong with this model?



IBPSA

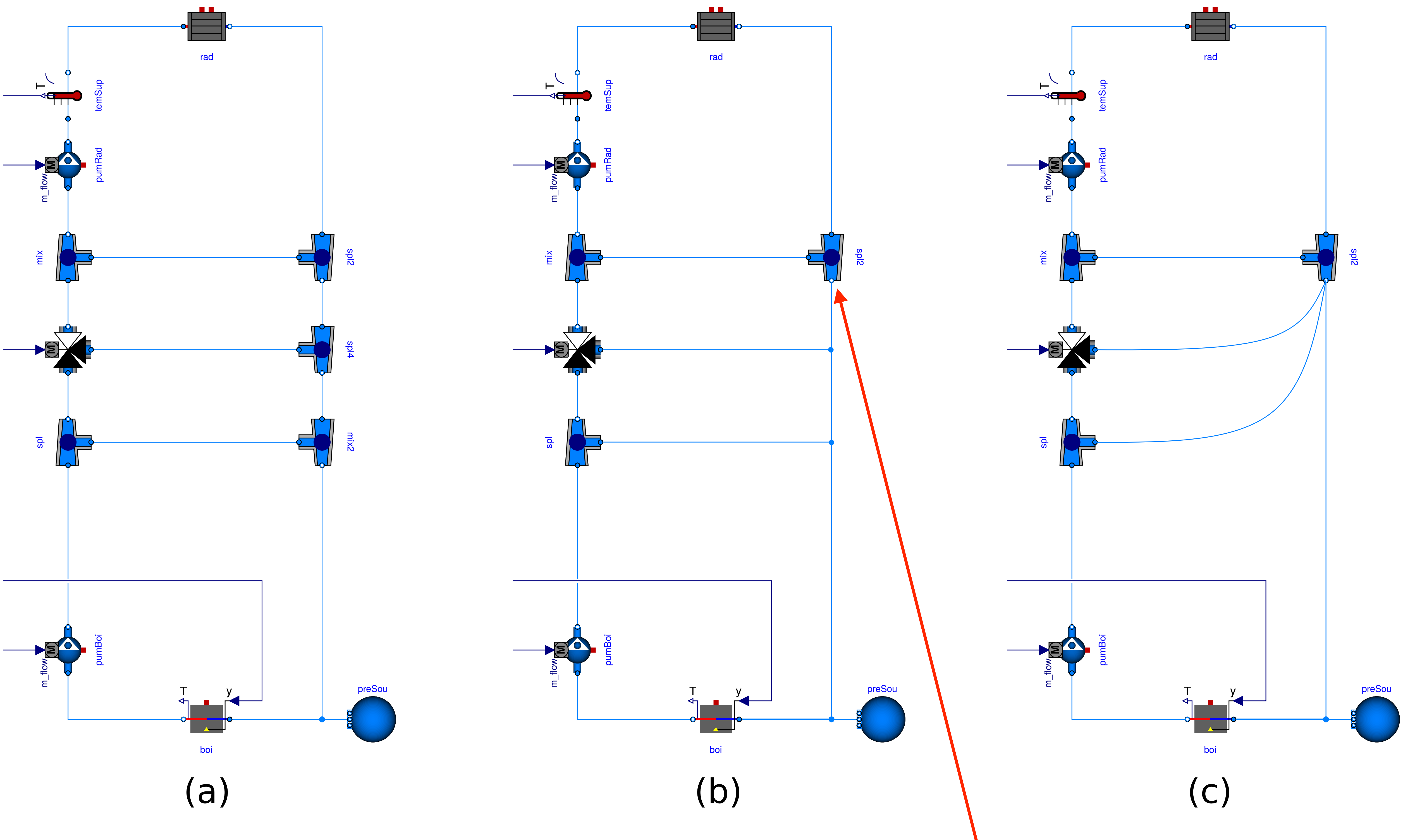
Modeling of fluid junctions



(a) (b) (c)

IBPSA

Modeling of fluid junctions



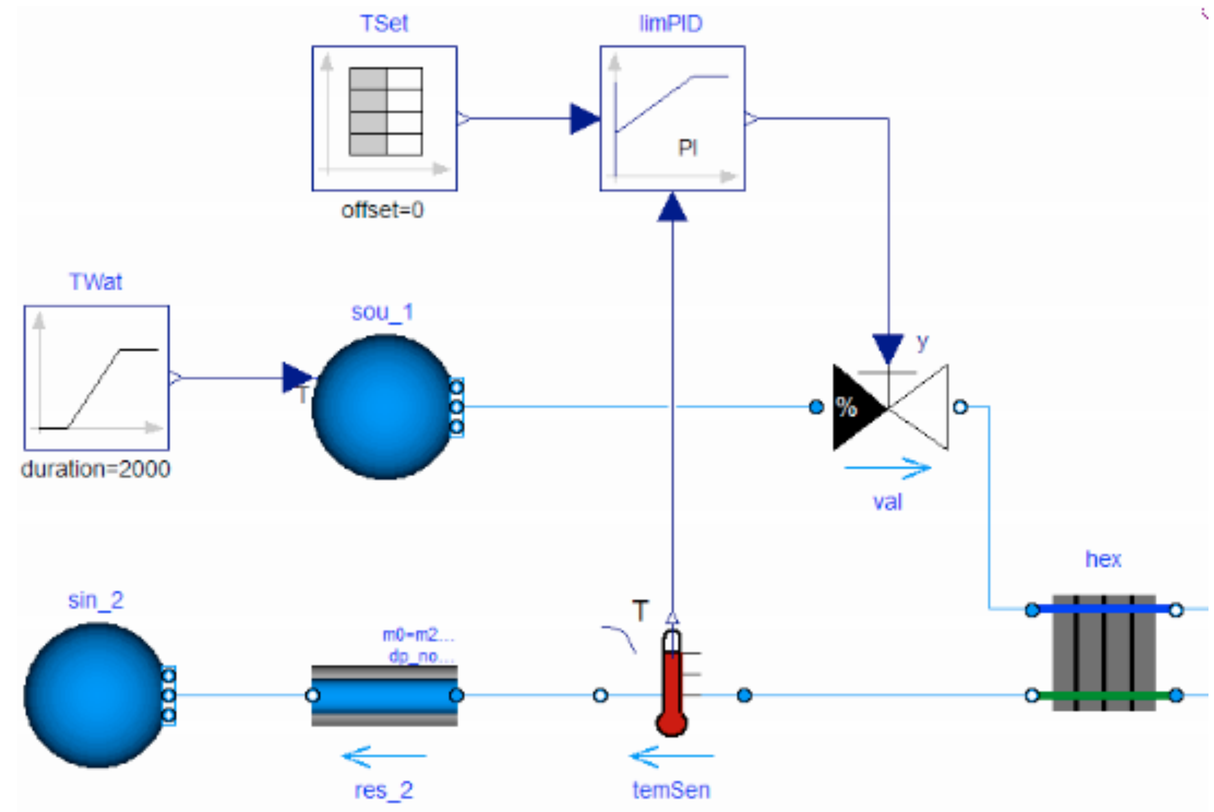
$$h = \frac{\sum_i \max(0, \dot{m}_i)\, h_i}{\sum_i \max(0, \dot{m}_i)}$$

IBPSA

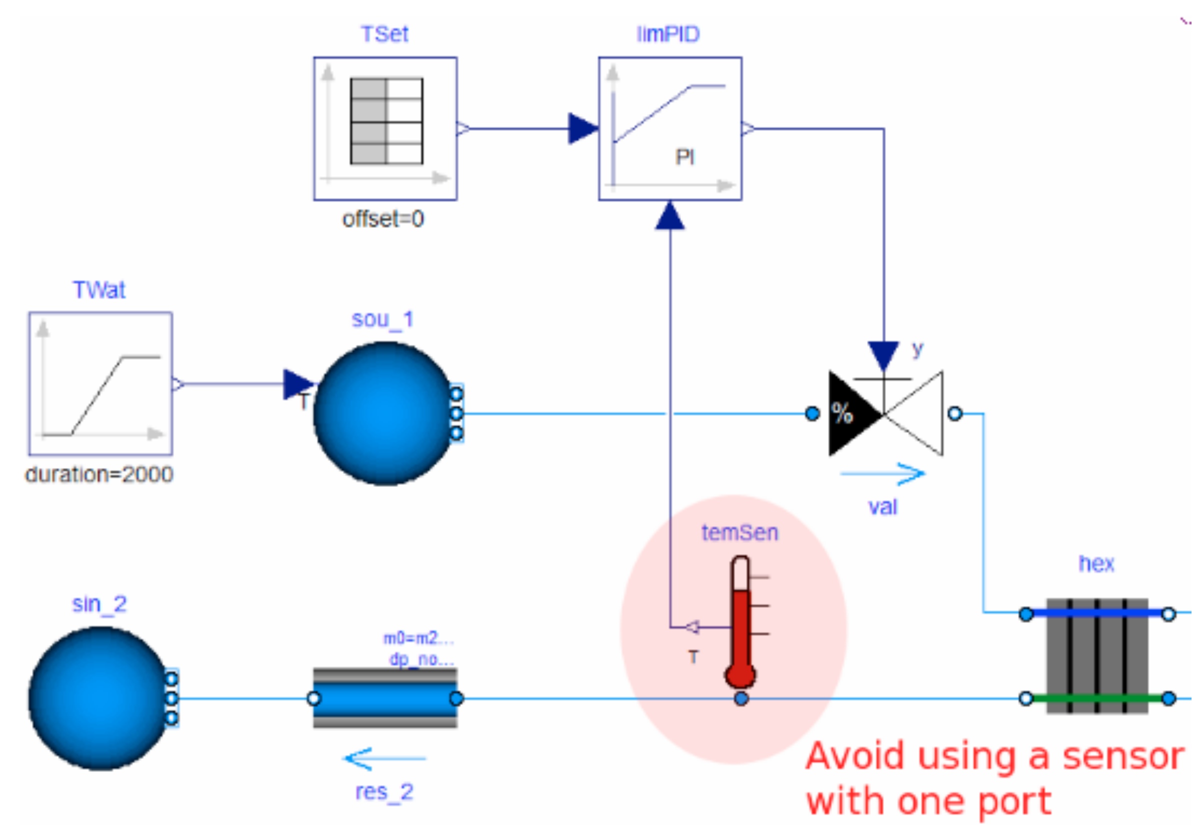
Avoid oscillations of sensor signal

Correct use because

$$\tau \frac{dT}{dt} = \frac{|\dot{m}|}{\dot{m}_0} (\theta - T)$$



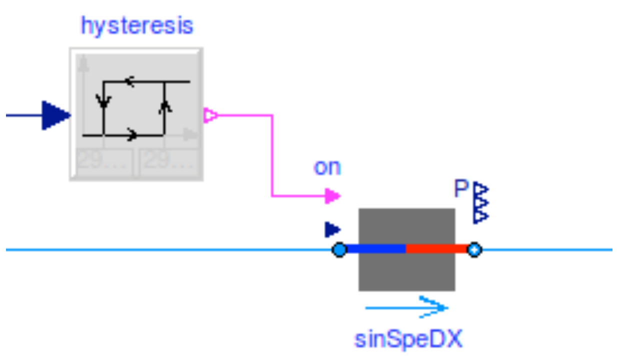
Incorrect, as sensor output oscillates if mass flow rate changes sign.
This happens for example if the mass flow rate is near zero and approximated by a solver.



See also Fluid.Sensors.UsersGuide

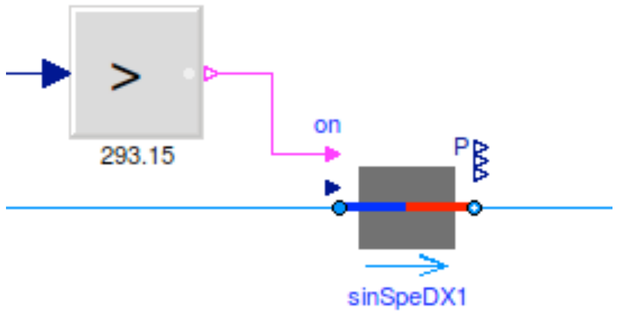
Always guard against oscillations and noise (numerical noise or measurement noise)

Correct configuration



If the control input oscillates around zero, then this model can stall

Avoid this configuration



What happens if this model is simulated with an adaptive time step?

```
model Test
  Real x(start=0.1);
equation
  der(x) = if x > 0 then -1 else 1;
end Test;
```

IBPSA

Don’t **R**epeat **Y**ourself: Propagate common parameters

Don't assign the same values to multiple parameters:

```
Pump pum(m_flow_nominal=0.1) "Pump";
TemperatureSensor sen(m_flow_nominal=0.1) "Sensor";
```

Instead, propagate parameters and assign the value once:

```
Modelica.SIunits.MassFlowRate m_flow_nominal = 0.1
  "Nominal mass flow rate";
Pump pum(final m_flow_nominal=m_flow_nominal) "Pump";
TemperatureSensor sen(final m_flow_nominal=m_flow_nominal) "Sensor";
```

Assignments can include computations, such as

```
Modelica.SIunits.HeatFlowRate QHea_nominal = 3000
  "Nominal heating power";
Modelica.SIunits.TemperatureDifference dT = 10
  "Nominal temperature difference";
Modelica.SIunits.MassFlowRate m_flow_nominal = QHea_nominal/dT/4200
  "Nominal mass flow rate";
...
```

Don’t **R**epeat **Y**ourself:
Always define the media at the top-level

Top-level system-model

```
replaceable package Medium = IBPSA.Media.Air
  "Medium model";
```

Propagate medium to instance of model

```
TemperatureSensor sen(
  redeclare final package Medium = Medium,
  final m_flow_nominal=m_flow_nominal) "Sensor";
```

Note: For arrays of parameters, use the `each` keyword, as in

```
TemperatureSensor sen[2](
  each final m_flow_nominal=m_flow_nominal)
"Sensor";
```

Questions?