ThermoFlow
A Thermo-Hydraulic Library in Modelica

Howto

Jonas Eborn[†], Hubertus Tummescheit[†] & Falko Wagner[‡]

[†]Department of Automatic Control
Lund University, Sweden
{*jonas,hubertus*}*@control.lth.se*

[‡]Department of Energy Engineering
Technical University of Denmark
falko@et.dtu.dk

Contents

Abstract

The Modelica package in its original or in a modified form is provided "as is" and the copyright holder assumes no responsibility for its contents what so ever. Any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright holders, or any party who modify and/or redistribute the package, be liable for any direct, indirect, incidental, special, exemplary, or consequential damages, arising in any way out of the use of this software, even if advised of the possibility of such damage.

The ThermoFlow package is a subpackage of the Modelica package. It is currently under development and the authors of the package do not make any commitments to keep future versions of the library fully compatible to the current version.

0.1 Library Structure

There are many different ways how a general, medium independent library for thermo-hydraulic flows can be structured. The ThermoFlow library provides a lot of building blocks, which cannot be simulated directly and some examples, which are ready to be connected on a flow sheet. These come in three flavors:

Base Classes : Function libraries and basic models which model fundamental properties of a fluid system, e. g., mass- and energy conservation, but which need to be put together in the right way with other fundamental models in order to obtain a useful model for simulation.

Partial Components : Models which are built up from submodels, but still need some minor change like setting the medium type in order to be usable

Components : Complete, ready to use models which are either for a specific medium, e. g., water, or medium independent.

For a newcomer to Modelica and/or the library it is a good idea to start with the examples in components and play with them before diving into building own models.

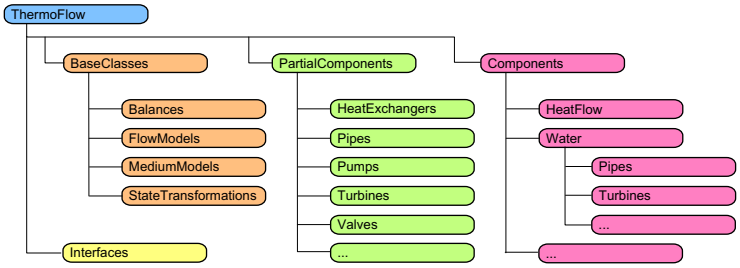

Figure 1: Basic Package Structure of the ThermoFlow Library

0.2 How to Do Your Own ... from Existing Classes

The following section will give an introduction on how to create your own components from the base classes. We will take you through this step by step, covering all important aspects of component design and implementation.

0.2.1 The basics

As mentioned earlier in 0.1, the ThermoFlow library consists of base classes, partial components and components. The base classes are described in detail in the documentation. Now, the development of new components can be done on two levels:

Partial components : Components that contain important knowledge about the general behaviour. But the supplied information (i.e. the equations, variables and parameters) is not enough to make an actual instance of that component for simulation. Additional information has to be supplied to make the component work.

Components : Components that are *finished* and therefore ready for simulation.

There will be given examples of both.

Examples

If you take a look at a model of a pipe, some properties can be expressed in general, whether the pipe contains a liquid (e.g. water) or a gas. So for example the mass and energy balances for the pipe can be expressed in general equations. But for the model of the pipe to work, we need to add some more information about the actual medium that flows through the pipe. Because we need to add this additional information, we call the general model of the pipe for *partial*.

Likewise, the momentum balance for the pipe can be expressed in general (see the documentation), only accounting for a certain size, which is called the pressure loss. But this pressure loss element in the momentum balance does not need to be specified on a general level. But for the model to work, this information has to be added.

When modeling turbines, a general expression for the turbine work W_T is

$$W_T = \dot{m}(h_1 - h_2) \tag{0.2.1}$$

where the enthalpy difference $h_1 - h_2$ can be replaced by a term containing the isentropic enthalpy, $h_{2,is}$ and the isentropic efficiency, η_{is}

$$(h_1 - h_2) = \eta_{is}(h_1 - h_{2,is}) \tag{0.2.2}$$

This model of the turbine work is very general, but in order to apply it to a final component of a turbine to be used for system simulation, $h_{2,is}$ has to be calculated for the distinct medium flowing through the turbine

$$h_{2,is} = f_{medium}(p_2, s_1) \tag{0.2.3}$$

Another aspect of partial models is that they (usually) don't contain geometry information (e.g. parameters). The partial models are expressed using general terms, typically only involving *variables*. For a control volume, the actual volume V is a variable. But for the purpose of modeling a pipe, the volume V is fixed by a parameter V_0. Whereas for the purpose of modeling a water tank, where the actual volume of the water can vary, V is not a parameter but a (dynamic) variable. Hence a partial model should be implemented only using variables, which in the end can be set equal to certain parameters for the final component.

0.2.2 Add your own Pipe

Let us take a look at a general pipe. A typical description of a pipe would be to set up a control volume, with balance equations for mass and energy transport and off course a flow model describing the pressure loss - mass flow relationship. As mentioned earlier, medium properties have to be added. Unless the pipe is considered adiabatic, a heat connector for the heat flux in or out of the pipe has to be added also.

The building blocks that we need are therefore:

1. Balance equations for mass and energy
2. Connectors at the ends
3. Heat connector for heat loss
4. Medium properties
5. General flow model (momentum balance)
6. Pressure loss model

Items 1 to 5 are provided in the standard ThermoFlow base classes. So is a simple pressure loss model, but in this example we will show you how to build your own.

A general way to express the pressure loss in a pipe is formulated by [2] and states that

$$\frac{\Delta p}{\rho} = h_l \tag{0.2.4}$$

where h_l denotes the head loss. In terms of the general momentum balance equations, the variable $Ploss$ has to be specified. The momentum balance equation is formulated as a force balance (see section ???), why $Ploss$ shall have the unit [N] and a translation of $Ploss$ to h_l is required:

$$Ploss = h_l A \rho \tag{0.2.5}$$

A general term for the head loss can be expressed by

$$h_l = f \frac{L}{D} \frac{\overline{V}^2}{2} \tag{0.2.6}$$

and in this expression, the friction factor f has to be determined.

Combining 0.2.5 and 0.2.6 we obtain an equation for the pressure loss term $Ploss$

$$Ploss = f \frac{L\, mdot^2}{2D\rho A} \tag{0.2.7}$$

This partial model has been implemented in the class FrictionFactorD (D for distributed pipes) and can be used to implement all kinds of friction models by extending from it at specifying the desired relation for f, in this case Blasius, equation (0.2.8):

In the following we will derive an expression for the pressure loss after Blasius.

For turbulent flow in smooth pipes, with not too high Reynolds number ($Re < 10^5$), the relationship by Blasius gives us the friction factor f

$$f = \frac{0.3164}{Re^{0.25}} \tag{0.2.8}$$

whereas the Reynolds number Re is given by

$$Re = \frac{\rho \overline{V} D}{\mu} \tag{0.2.9}$$

where

$$\overline{V} = \frac{mdot}{\rho A} \tag{0.2.10}$$

This gives us the necessary expression for f, which we then, together with the base class FrictionFactorD, use to implement a pressure loss model after Blasius:

```
model BlasiusPressureLossD
  extends FrictionFactorD;
  Real Re[n];
  Real mu[n];
equation
  for i in 1:n loop
    Re[i] = noEvent(abs(mdot[i + 1]*Dhyd/(A*mu[i])));
    f_friction[i] = 0.3164/(Re[i]^0.25);    // Blasius (Fox&McDonald, p.364)
  end for;
end BlasiusPressureLossD;
```

We are now ready to implement our own pipe model using the pressure loss model just implemented.

The pipe model implemented in Components.Water.PipesAndVolumes is used for this purpose. This model is called PipeDS, denoting, that it is a distributed pipe (D), has single flow connectors (S) and static momentum balance (S). Basically it is a control volume, combining balance equations for mass and energy and a static momentum balance equation, as well as single flow connectors. Additionally the model contains a simple quadratic pressure loss model and a medium model for water.

With this basis, we can now implement our pipe model with the Blasius pressure loss model developed earlier. The recipe is to make an extend from the base pipe, change the pressure loss model and the geometry information. In Modelica language this looks like the following:

```
model PipeDSBlasius "Distributed pipe model, Blasius pressure loss model"
  extends PipeDS(
    redeclare PartialComponents.Pipes.BlasiusChar char,
    redeclare model PressureLoss
```

```
    extends PartialComponents.Pipes.BlasiusPressureLossD
      (mu=ones(n)*char.mu);
  end PressureLoss);
end PipeDSBlasius;
```

This is how to make your own pipe model.

0.2.3 Add your own Reservoirs

Two essential components are reservoirs and sources. The class hierarchy for reservoirs and sources is shown in figure 2. For detailed information about the partial components, see [3].

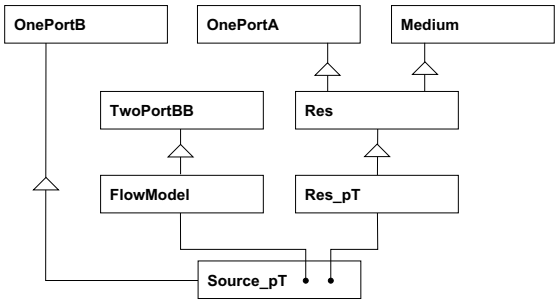

Figure 2: Class hierarchy of PartialComponents.Reservoirs

This section will shortly explain how to add a new reservoir and source of a certain medium.

Given the appropriate medium model, the following steps have to be taken to add reservoirs and sources of that medium:

- Create new reservoir
- Add default initial conditions
- Create new source using the reservoir

In the following examples it is assumed that there exists a medium model called `MediumModels.IdealGas.OxygenRichFlueGasMix`.

Reservoir

The first step is to create a reservoir. For this purposes the partial models in PartialComponents.Reservoirs can be used.

For a mixture of ideal gasses the right starting point would be the class `Res_pTX`. Since OxygenRichFlueGasMix consists of 4 different ideal gasses, the parameter nspecies has to be set to 4. The parameter `mass_x0` defines the constitution of the gas mix based on mass fractions.

In order to implement a reservoir with the flue gas mix, the model is extended (inherited) from `Res_pTX`, the mass fractions are set, and the default medium is replaced by the flue gas mix.

```
model FluegasResS_pTX "simple flue gas source or sink"
  parameter Integer nspecies=4;
  parameter Real mass_x0[nspecies]=0.01,0.03,0.75,0.21;
  extends PartialComponents.Reservoirs.Res_pTX(
    nspecies=nspecies,
    mass_x0=mass_x0,
    redeclare model Medium
      extends MediumModels.IdealGas.OxygenRichFlueGasMix(
            n=1,nspecies=nspecies);
    end Medium);
end FluegasResS_pTX;
```

This finishes the reservoir.

Source

Now, with the reservoir from above, the according source can be implemented.

A source is basically a reservoir with an additional flow model. This has been implemented in the PartialComponents.Reservoirs.Source classes. So in order to create a source with the reservoir from above, the new model is extended from the partial model `Source_pTX`, with appropriate modifications of `nspecies` and `mass_x0`, and the redeclaration of the reseroire `res_pTX` with the reservoir `FluegasResS_pTX` from above.

```
model FluegasSourceS_pTX "simple flue gas source"
  parameter Integer nspecies=4;
  parameter Real mass_x0[nspecies]=0.01,0.03,0.75,0.21;
  extends PartialComponents.Reservoirs.Source_pTX(
    nspecies=nspecies,
    mass_x0=mass_x0,
    redeclare FluegasResS_pTX res_pTX(nspecies=nspecies));
end FluegasSourceS_pTX;
```

This finishes the implementation of the source.

0.2.4 Add your own Turbomachine

In this section we will learn how to make a model of a water pump.

Modeling a pump (or in fact any other flow machine) is somewhat different from modeling a pipe. First of all, it gives no meaning to make a distributed model of a pump. Not in the way a distributed pipe works. Therefore the lumped base classes have been developed.

As described in the documentation, control volumes and flow models have to be placed in an alternating sequence, always. So in order to keep the same structure as in the distributed case (e.g. the pipe model), we need a lumped control volume and a lumped flow model in order to build our pump.

For the lumped control volume, we can use the implemented LumpedVolume in Components.Water.PipesAndVolumes. It has the necessary connectors and medium model.

And to build the flow model for the pump we need

1. single flow connectors

2. some medium properties
3. expression for the change in enthalpy, so the enthalpy of the flow out of the pump can be calculated
4. relationship between pressure difference and mass flow rate

Elements 1 through 3 are formulated in a partial model, called BasePumpSS. The change in enthalpy requires the calculation of $h_{2,is}$, which has to be done also, since it depends on the medium model.

What remains to be added is the relationship between the pressure difference and the mass flow rate (or more typically in fact, the volume flow rate).

As a very simple pump characteristic, we chose the formulation

$$dp(V, n) = R_1 nn + 2R_2 nV - R_3 abs(V)V; \tag{0.2.11}$$

Here, dp is the normalized pressure gradient, V is the normalized volume flow rate and n is the normalized rpm.

This model is implemented in a partial model:

```
partial model PumpCharacteristic
  extends CommonRecords.FlowVariablesSingleStatic;
  SIunits.Pressure dp;
  parameter SIunits.Pressure dp0;
  SIunits.VolumeFlowRate Vdot;
  parameter SIunits.VolumeFlowRate Vdot0;
  parameter Integer rpm=1;
  parameter Real R1=1.0;
  parameter Real R2=0.3;
  parameter Real R3=2*R2;
equation
  dp = dp0*(R1*rpm*rpm + 2*R2*n*(Vdot/Vdot0) -
      R3*abs(Vdot/Vdot0)*(Vdot/Vdot0));
  Vdot = mdot/d;
end PumpCharacteristic;
```

Please note, that this model only runs at the nominal rpm, why the normalized rpm is set to 1. The isentropic enthalpy at the outlet, $h_{2,is}$, is calculated from a function in the Water subpackage

$$h_{2,is} = h_{isentropic}(p_2, s_1) \tag{0.2.12}$$

where p_2 is the pressure at the outlet and s_1 is the entropy at the inlet.

We do now have all the ingredients, and can add them to implement a simple water pump.

```
model SimplePump
  PumpGeometry geo;
  extends BasePumpSS(eta_is=geo.eta_is);
  extends PumpCharacteristic;
equation
  h2is = MediumModels.Water.water_hisentropic(b.p, a.s, 1);
  a_upstream = (p2 > p1);
end SimplePump;
```

One additional equation that is stated in the SimplePump model above is the variable a_upstream, which is not used anywhere in this model, but is part of the general flow interface and therefore has to be assigned a value.

Now we have to combine the lumped control volume and the pump flow model to build a final components, that fits into the library structure. For that purpose we can use an empty shell component, providing two single flow connectors, that can contain the two components mentioned above.

```
model PumpModelSS
  extends Interfaces.SingleStatic.TwoPortAB;
  SimplePump pump;
  LumpedVolume vol;
equation
  connect(a, vol.a);
  connect(vol.b, pump.a);
  connect(pump.b, b);
end PumpModelSS;
```

This was one way to implement a pump model! Typically one would be interested in the work needed to run the pump. An equation for that (and off course proper variable declaration for `W_pump` could be added to the PumpModelSS:

```
  SIunits.Power W_pump;
equation
  W_Pump = pump.dh * pump.mdot;
```

A further development of this model could be to add a rotational joint, connecting the power and rpm to a generator. But we leave this up to the user!

0.2.5 Add your own Heat Exchanger

Now we will show how to build a heat exchanger from existing components.

A heat exchanger consists basically of two flow sections and a heat conducting wall between them. The two flow sections can easily be modeled as two pipes.

We have already seen how to implement a pipe, with medium and pressure loss models. For the heat exchanger we will not use the final pipe, since for the final pipe model, the geometry is already fixed. But for the sake of user-friendliness , we want to provide the geometry information on the next level, i.e. the heat exchanger geometry information. So we will base our heat exchanger on the base pipe model Components.Water.PipesAndVolumes.BasePipeDSS. This pipe model contains a heat flow connector, which we will use to implement the heat flow between the two pipes.

What we need next is a wall model, that calculates the heat flow based on the temperatures in the heat flow connectors of the two pipes.

One dimensional, steady state heat transfer is described after [1]:

$$Q_{12} = \frac{A\lambda(T_1 - T_2)}{d} \tag{0.2.13}$$

where the index 12 means that the heat flows from 1 to 2, A is the area of the wall and d is the thickness.

Using the base class BaseWallD, giving two distributed heat connectors from Interfaces.HeatTransfer

```
model BaseWallD
  parameter Integer n(min=1)=1;
  extends Interfaces.HeatTransfer.TwoPortD(n=n);
end BaseWallD;
```

we can derive a model for a simple, distributed wall model, giving one dimensional, steady state heat transfer:

```
model WallD "Discretized wall model with 1-dim steady state heat transfer"
  parameter Integer n(min=1) = 1;
  WallGeometry geo;
  extends BaseWallD(n=n);
equation
  geo.d*qa.q = (geo.A/n)*geo.k*(qa.T - qb.T);
  qb.q = -qa.q;
end WallD;
```

Please note, that this wall is a massless wall without state variables.

We now have all the ingredients for a heat exchanger model. The procedure is to take a shell component with 4 single flow connectors, include two pipes and a wall model and connect them properly. Add geometry information and compile gently!

```
model SimpleHeatExD "Heatexchanger with paralell flow, discretized model"
  extends Interfaces.HeatExchangers.HeatXSD;
  parameter Integer n(min=1) = 1;
  HeatExGeometry heatexgeo;
  Components.Water.PipesAndVolumes.PipeGeometry pipegeo1(mdotdp0=5.0);
  Components.Water.PipesAndVolumes.PipeGeometry pipegeo2(mdotdp0=5.0);
  replaceable class TWall
    extends Components.HeatFlow.Walls.WallD;
    end TWall;
  TWall wall˙D;
  Components.Water.PipesAndVolumes.BasePipeDSD pipe1;
  Components.Water.PipesAndVolumes.BasePipeDSD pipe2;
equation
  connect(pipe1.a, h1);
  connect(pipe1.b, h2);
  connect(pipe2.a, c1);
  connect(pipe2.b, c2);
  connect(pipe1.q, wall˙D.qa);
  connect(pipe2.q, wall˙D.qb);
end SimpleHeatExD;
```

This finishes the heat exchanger model. Please note, that in this example the geometry information is not propagated to the wall and pipe models. This is only done to save space. Furthermore the implementation of the pipe model earlier demonstrates this nicely.

The model of the heat exchanger SimpleHeatExD can easily be done in Dymola, by drag and drop from the earlier mentioned models of wall and pipes. Here the connections can also be done in the graphical user interface, resulting in the same equations as above.

0.2.6 Add a new Medium Property Model

The property models in the library can be divided into 3 different classes:

- Pure ideal gas models for a single substance,
- Mixtures of any number of these ideal gases,
- Full property models for liquid, gas and two phase regions.

Adding ideal gas models or mixtures of those is quite straightforward and can be done similar to the existing classes in subpackage *MediumModels/IdealGas*. All data is contained in a record, see the ones in IdealGasData, which was automatically generated from a database of medium models. A base model for any Ideal gas should contain three equations for the enthalpy h, the specific heat capacity c_p and the entropy s, analogous to the class *PureIdealGas* in package *IdealGas*, A complete model for ideal gas is then created by using multiple inheritance from the new class and the class *IdealGasSingle*, exactly as in the example for *IdealGasDryAir*.

Models for ideal gases can be used for pressure and temperature as states or for density and temperature as states. It is possible to use enthalpy and pressure as states, but not recommended.

The physical property model for a new medium model for a mixture of ideal gases inherits most of the equations from a model from the library that defines all thermodynamic properties (except transport properties) in terms of vectors of the component properties. By default, all library models assume a one-dimensional discretization of all flows and are vectorized. Some of the variables are therefore 2-dimensional arrays with space as one coordinate and number of the component as the other. In order to define a new ideal gas mixture, one needs to

1. define the number of components,
2. introduce an instance of the pure component property model for each component of the mixture,
3. connect the temperature in the property data structure to the temperatures in the pure component models and
4. combine the pure species base properties into vectors/arrays as in the example below for the moist air model.

One has to make sure to use the same order of pure species for all vectors. All other derived properties are calculated in the base class IdealGasMixProps[1]. As an example, here is the code how to define a new model for moist air from the three components IdealGasN2, IdealGasO2 and IdealGasH2O:

```
model MoistAirMix "moist air gas mixture  H20, N2, O2"
  parameter Integer nspecies=3;
  extends IdealGas.IdealGasMixProps(nspecies=nspecies);
protected
  IdealGas.IdealGasN2 n2(n=n);
  IdealGas.IdealGasO2 o2(n=n);
```

[1] See the documentation of the ThermFlow library about the details.

```
  IdealGas.IdealGasH2O h2o(n=n);
equation
  for i in 1:n loop
    pro[i].T = n2.T[i];
    pro[i].T = o2.T[i];
    pro[i].T = h2o.T[i];
    compcp[i, :] = n2.cp[i],o2.cp[i],h2o.cp[i];
    comph[i, :] = n2.h[i],o2.h[i],h2o.h[i];
    comps[i, :] = n2.s[i],o2.s[i],h2o.s[i];
  end for;
  invcompMM = 1/n2.MM,1/o2.MM,1/h2o.MM;
  compMM = n2.MM,o2.MM,h2o.MM;
  compR = n2.R,o2.R,h2o.R;
end MoistAirMix;
```

Bibliography

[1] Donald R. Pitts and Leighton E. Sissom. *Theory and Problems of Heat Transfer*. Schaum's Outlines. Mc Graw Hill, 1977.

[2] Alan T. McDonald Robert W. Fox. *Introduction to Fluid Mechanics*. Wiley, third edition, 1985.

[3] Hubertus Tummescheit, Jonas Eborn, and Falko Jens Wagner. *ThermoFlow - Documenation*, 2000.