ThermoFluid
A Thermo-Hydraulic Library in Modelica

Documentation

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 ThermoFluid 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. As the library is still under development, this document may not reflect in all details the current ThermoFluid version. The basic principles of the library, the dynamic state models, the discretization scheme and the library structure, are the same as described in this document.

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.

1. Scope of the Library

The general goal of the library is to provide a framework and basic building blocks for modeling thermo-hydraulic systems in Modelica™. For obvious reasons it is impossible to provide a complete library, so one of the main goals is extensibility. For the same reason, much more emphasis will be put on the basic parts of the library, such as medium models and essential control volumes, than on an exhaustive application library. The focus of the library is on models of homogeneous one- and two-phase flows, non-homogeneous and multi-phase flows are not taken into account yet. It is necessary to support bidirectional flow, because flow directions can change during simulation or are not known initially in networks. Flow splitters or junctions must also be modeled correctly for arbitrary flow directions in all branches.

The models in the library are designed for system level simulation, not for detailed simulation of flows which are usually done in CFD packages. The models are thus discretized in one dimension or even lumped parameter approximations.

It has to be emphasized that especially in the area of fluid flow different assumptions about the importance of terms in the general equations can lead to models which are very different mathematically. We offer a choice of assumptions which should nonetheless cover a broad range of applications.

The documentation here is only for the overall structure and ideas of the library. Apart from that, only few partial components are described in order to give an overview. Documentation for specific models is available with the html-documentation for that model or in the case of very generic models in the documentation for the package.

Reading instructions This documentation contains parts directed at the inexperienced user as well as experienced thermo-hydraulic modelers. Users only interested in using the library models should only read Sections 1-2.3 and the examples in Section 9. Advanced users who want to build

their own components and/or use other medium models than the ones available should also read the rest of the documentation.

1.1 Acknowledgments

This package would not have been possible without ideas and help of many people apart from the authors. Many thanks to all members of the Modelica Design group for such a nice modeling language. Thanks to Karl Johan Åström for many fruitful discussions. Thanks to Olaf Bauer for laying the ground for the implementation of the medium models and more.

This package and its predecessor, the **K2** library, was developed at the department of Automatic Control, Lund University in collaboration with the Department of Energy Engineering at the Technical University of Denmark. The development was made with financial support from Sydkraft Research Foundation and NUTEK, which is gratefully acknowledged.

1.2 Boundary conditions

Naming convention for packages and components within the Modelica standard library uses the 'dot' (.) notation. That is, the ThermoFluid library within the Modelica standard library is accessed by `Modelica.ThermoFluid`. All references in this documentation use this as an offset. Therefore, when referencing the subpackage Modelica.ThermoFluid.Components.Water, Modelica.ThermoFlow is omitted and only Components.Water is used.

Where no dot notation is used, i.e. only the package or model name is stated, it is assumed, that the package or model resides in the current scope (local).

2. Basic Design Ideas

The basic design principles of the library are:

1. one unified library both for lumped and distributed parameter models,
2. separation of the medium submodels, which can be selected through class parameters,
3. both bi- and unidirectional flows are supported,
4. assumptions (e. g., gravity influence yes or no) can be selected by the user from the user interface.

The first guideline puts a constraint on the discretization method that must be used in the distributed parameter models: only the "multinodal", "staggered grid" or finite volume method (see figure 1), where all fluxes are calculated on the border of a control volume and the intensive quantities are calculated in the center of a control volume, reduces to a useful model in the lumped parameter case. The classical introduction to the finite volume method for fluid flows is the book by Patankar, [4]. This model is very common for system modeling and for one-dimensional discretizations, but it has the drawback that the approximation of the spatial derivatives is always only first order accurate.

A finite difference model is used for heat conduction in solid structures.

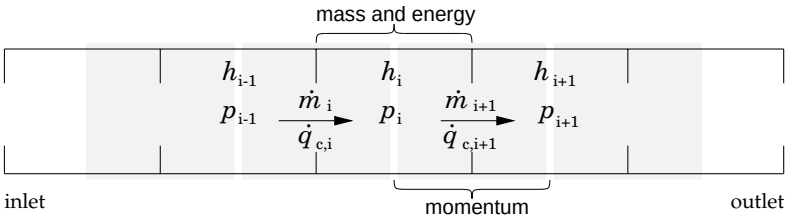

Figure 1 Discretized Flow Grid

2.1 Library Structure

There are many different ways how a general, medium independent library for thermo-hydraulic flows can be structured. The ThermoFluid 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.

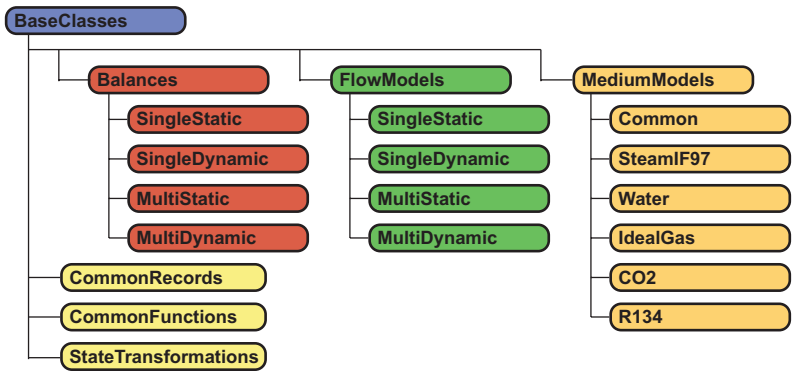

Figure 2 Basic Package Structure of the ThermoFluid Library

2.2 Connectors

All connectors between models and the “Partial Components” are organized in 5 different flavors which can be recognized by the shape and color of the connectors. The 5 types are:

Figure 3 Basic connectors of the ThermoFluid Library

It should be noted that under certain circumstances a dynamic momentum balance may be easier to simulate than a static one, because a linear ODE with one nonlinear term can be easier to solve than a nonlinear algebraic equation for the mass flow.

2.3 Lumped and 1D Discretized Models

There are three different basic types of atomic models in the library, which can be combined into more complex ones which share the same type of connectors. They are the result of different abstractions:

Control Volumes , abbreviated CV here, have a finite volume and are stores for mass and energy. They can be either lumped or discretized in space.

Figure 4 Atomic, Compound and Distributed Models

Lumped Flow Models are the result of a modeling abstraction, where the volume is neglected and usually there is an algebraic equation that relates pressure drop and mass flow, like in valves or pumps. The abbreviation for flow models is FM.

Dynamic Flow Models calculate the storage of momentum in a control volume which usually is staggered by 1/2 a grid length wrt. the grid of the control volumes.

In terms of the connectors or the associated variables and equations, the dynamic and static flow models, abbreviated FM, can be treated as equal. One very important rule for the usage of the models is the consequence of these atomic models:
Control volume models and flow models always have to alternate each other.

In terms of the connector types that we choose this can also be expressed as:
Always connect a filled connector with a hollow one.

This may seem like a restriction, but is useful to avoid mistakes with getting a well posed model. Furthermore this alternating of models that store mass and energy and models that calculate the mass flow between them, has been used in most simulation packages for thermo-hydraulics.

These atomic models are not very well suited for 1D distributed models and they are also not well suited for higher level building blocks like heat exchangers or turbines. But for connecting models from building blocks there is a very simple relation between the atomic models and distributed or compound models, see figure 4. This means that a distributed or a compound model is equivalent to a lumped control volume and a lumped flow model in a row.

One particular detail that should be noted about control volumes is, that **all transported properties are mean values for the whole control volume**, see figure 5. This means that the temperature at the west side connector of a CV is not the temperature of the flow into the CV at that place: it is the mean temperature of the CV. The same holds for density, entropy, enthalpy etc. If these quantities are of interest, they would have to be calculated in a derived class. This is a consequence of building the models for flow in both directions. All information for calculating the mass and energy balances is contained in the two flow variables, `mdot` and `q_conv`.

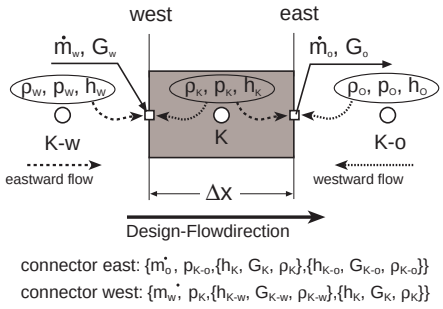

Figure 5 Lumped model of a Control Volume

It is possible to model idealized control volumes with 0 mass and volume. This has been done for ideal flow joints and splitters and it can also be used to effectively connect 2 flow models in a row. These idealized control volume calculate pressure p and specific enthalpy h (and possibly the mass fractions $mass_x$) from algebraic equations.

The current models in the library deal only with purely convective flows. Mixed convection and diffusion can easily be implemented, we are planning to add them in a future version of the library. A mixed convection-diffusion model builds on the assumption that the tepmeratures follow a certain profile along the flowpath. This has important consequences with respect to "sliding mode" problems **??**.

2.4 Thermodynamic Property Models

The availability of fast, accurate medium models is a notorious problem in thermal hydraulic applications as soon as something more elaborate than ideal gas models are needed. In some areas there exist recommended formulations (IAPWS/IF97 for water [6]) or de-facto standards (NIST-REFPROP routines for refrigerants, [3]) that have to be taken into account. External function call interfaces in Modelica make it possible to use these standards directly. Available routines and most medium property models in the literature (see, e. g., [5]) are designed with stationary calculations in mind, therefore they have to be extended to include some needed extra derivatives for dynamic calculations.

Currently we have implemented high-accuracy medium models for the whole fluid region for the media in Tabelle 2.4.

Water/Steam	IAPWS/IF97
CO_2	Spann/Wagner 1996
R134a	NIST Refprop

Table 1 High accuracy Medium Property Routines

Some more are going to follow soon. It is relatively easy to add your own medium model to the existing ones, see the ThermoFluid HowTo.

Vapor-liquid equilibrium calculations (VLE) for cubic and other medium models have to be performed iteratively and numerically, either by making use of Maxwells criterium or calculating that Gibbs free enthalpy is equal

for both phases. The numerical calculations are too inefficient to be performed at each time step during dynamic simulation. In order to calculate medium properties inside the two-phase region, it is sufficient to know the properties on the phase boundaries and interpolate with the vapor mass fraction x. An efficient implementation of medium properties for pure components requires that VLE are calculated before the simulation and that VLE data is approximated either with a suitable function or with smooth spline interpolation. For the above listed media, high accuracy approximations are either available in the standard formulation (e. g., for water and CO_2) or provided by us.

There is an interdependence between the choice of the medium model and the selection of state variables, see also section 3.1. Many details of the medium model depend on the choice of the state equations. Most medium models are available for all of the choices of state variables in the library.

Summarizing this means that the medium properties that are provided with this library:

1. are adapted for use with dynamic simulations and
2. use non-iterative, auxiliary equations for the calculation of VLE.

There are quite a number of properties available for ideal gases and mixtures of ideal gases. The up-to date list is contained in the general library documentation for the package `IdealGas.mo`.

Water and Steam Among the implemented medium property routines, the ones for water and steam take a special role. They are much more complex than the other ones, they are the only ones which are truly standardized by an international committee and they have been designed with high computational efficiency in mind. The standard is implemented directly from the equations given in [6], using Mathematica™ for all symbolic computations. The only exceptions that are not (yet) implemented are the non-equilibrium properties inside the two phase region for subcooled steam or overheated water. For details about the internals of the steam properties, e. g., the division into regions, please refer to the standards publication in [6]. There are a few additions for the sake of computational efficiency compared to the standard:

- the borders between the regions are provided as explicit functions of the state variables of the dynamic model.
- In order to avoid iterative VLE calculations, we have provided high-accuracy approximations to the specific enthalpy, the specific entropy and the density on the phase boundary of region 3, that is the near-critical region for pressures above ≈ 16.5 MPa.

Properties based on a Helmholtz fundamental equation Many of the high quality medium property routines are based on a dimensionless function of the Helmholtz free energy, $f(\rho, T)$, a function of density ρ and temperature T. Like other fundamental equations (e. g., the Gibbs free enthalpy fundamental equation used in the IAPWS/IF97), they completely define the thermodynamic state and all other properties can be derived by thermodynamic relationships of this function and its derivatives wrt. ρ and T. The Modelica functions in the package `MediumModels` are organized in such a way that once you have a Helmholtz function for any Medium, you

can very easily calculate all other variables of interest for that medium by feeding a record with the Helmholtz function and its derivatives into given library functions. For details, look into the documentation for the subpackage `MediumModels.Common`.

Future Extensions The model of using a Helmholtz function as a basis for further calculations can easily be extended to work with cubic equations of state or higher order approximations of the $p(v,T)$ thermodynamic surface. For that purpose, the Helmholtz function has to be split up into an ideal gas part and a residual part. We plan to implement this in a future version of the library.

2.5 Separation of Thermodynamic and Hydraulic Models

One of the important decisions when modeling thermo-hydraulic flow is if the momentum balance should be modeled dynamically or stationary. This choice depends on the purpose of the simulation study at hand and is mainly a question of the time scales of interest: If the momentum balance is included, pressure or massflow disturbances travel through the system with the speed $u \pm w$ where u is the speed of the fluid and w is the speed of sound. These speeds are also the eigenvalues of the characteristic equation or, short, characteristics. This makes it possible to simulate fast transients like emergency shutdowns, but for modeling much slower thermal processes it can be an unnecessary computational burden. On the other hand: after modeling a complex system it may turn out that one wants to exchange model that contains only the slow modes against one with a dynamic momentum balance.

We leave this decision open to the user of the library and provide models for both stationary pressure drop relations and a dynamic momentum balance. The use of multiple inheritance makes it possible to separate the hydraulic description of the momentum balance (subpackage `FlowModels` from the thermal models, which are in the subpackages `Balances` (the part independent of the dynamic state variables) and subpackage `StateTransforms`, which is independent of the connectors and the hydraulic model.

By separating these different issues:

- slow or fast dynamics,
- type of the medium,
- choice of the dynamic states and
- empirical relations for heat transfer or friction pressure drop

we provide a very flexible possibility to build up a suitable thermodynamic model.

There is a price to pay for this flexibility: The modeler has to understand and be able to use some of Modelicas advanced language features and must not hesitate to use *type parameters* and *multiple inheritance*.

2.6 Design for Reusability: Using Modelicas Advanced Language Features

In designing the ThermoFluid Library we have made extensive use of Modelicas modern programming language features. Everybody that wants to use the library to build new models from the low-level building blocks needs

to understand these features and know, which kind of submodels are necessary to build a new model. The `Components` subpackage contains lots of complete models as examples.

Generic Models through the use of class parameters In modeling it is very often useful to have a *generic model* as a placeholder for a group of models that may be used in certain context. The most prominent use of generic models, or type parameters in computer science terms (see [1] for an inspiration of Modelicas type system) in this library is the use of type parameters for all thermodynamic property related calculations.

Modelica provides two possibilities for the use of class parameters:

- a single *component* can be declared as **replaceable**
- a *class* can be declared as **replaceable**. All components with that class as its type will change their type if the type of the class is changed. It is also possible to inherit from the type parameter and via this construct it is possible to exchange one of the parent classes against one which is "type compatible" to the original class.

These components and classes are changed from their default types to a new one with the **redeclare** statement.

The restriction for both types of redeclarations are, that the new model has to be "type compatible" to the original restricting model (class). The exact definition of "type compatible" should be looked up in the Modelica reference manual [2], but a rough definition is as follows: imagine the hierarchical structure of a model expanded into all its basic components. A "type compatible" class contains at least the same basic components, but may contain any number of additional basic components. Take the following simple example:

```
record ThermoBaseVars
  parameter Integer n "number of discrete volumes"
  SIunits.Density d[n];
  SIunits.Temperature T[d];
  SIunits.SpecificEnthalpy h[n];
  SIunits.Pressure p[n];
end ThermoBaseVars;
```

A type compatible class does not need to inherit from this class, but it has to contain the variables n, d[n], T[n], h[n] and p[n] which have to be subtypes of Real variables and with the same definitions for units as in the Modelica base Library for SI units.

This definition makes it possible to define flexible but safe model libraries. The easiest way to use this feature is of course that all classes that are supposed to serve as an alternative in a **redeclare** statement inherit from the same base class.

Multiple Inheritance Multiple inheritance is another of Modelicas useful language features, but it should be used with caution. We have used it often to really separate base models for physical phenomena which can then be assembled from that two parts, e. g., the hydraulic models in subpackage `FlowModels` from the thermal models in subpackage `Balances`. The separation of concerns and later reassembly works in the following way for distributed models:

1. Hydraulic and thermal classes inherit from the same interface classes, which contain e. g., two connectors in the most common case.
2. The balance classes implement the mass and energy balance, using variables in the connectors.
3. The flow model classes implement the momentum balance equation (dynamic or static), also using variables in the connectors.
4. A control volume class inherits from one balance equation class and a flow model class and reassembles the equations from both classes into one model. The same interfaces are inherited twice and the semantic rule in this case is to just discard the second occurrence of the inherited class. All other definitions from both classes are merged.
5. There is one exception to this in Dymola: graphical annotations are not merged but always taken from the first class in the order of declaration.

The minor disadvantage of this approach is, that information about the complete set of equations that is used in one class is spread out over many different files. We strongly suggest to read this documentation and to make extensive use of the hyperlinked documentation for reading the model code.

3. Thermodynamic Model

The thermodynamic model is split up into two distinct parts, each with its specific tasks: the subpackage `Balances` contains models with all connectors of a control volume and sums up the contributions to the mass and energy balance equations. The subpackage `StateTransformations` contains equations that transform these contributions to the primary equations in mass and energy into equations into those dynamic states, which are better from a numerical point of view.

One note about the choice of introducing *the balance classes*. A pure thermodynamic control volume is essentially a 0-dimensional model and needs only its volume as geometric information. One connector would be enough for any number of connectors to that CV and the balance classes would be superfluous. This situation completely changes as soon as conservation of momentum is added to the model. The momentum is a vector quantity and even its 1-dimensional approximation in flow channels needs a length and a diameter as additional geometric information. The balance classes are necessary to take the number and direction of the flow connections into account.

3.1 State Variable Transformations

Differential equations for ρ and u are obtained from balance equations for mass and internal energy. The resulting primary equations can be transformed into secondary forms providing less stiffness in the equations and thus improve performance in dynamic simulations.

3.2 Primary Equations

A differentiation of $M = \rho V$ and $U = uM$ for a constant volume yields

$$V\frac{d\rho}{dt} = \frac{dM}{dt} \tag{3.1}$$

$$M\frac{du}{dt} = \frac{dU}{dt} - u\frac{dM}{dt} \tag{3.2}$$

Equations for the rate of change of mass and internal energy for a control volume with n flow connections to other control volumes and l heat transfer areas to solid bodies can be written as (positive flow into the CV):

$$\begin{pmatrix} \sum_i^n \dot{m} \\ \sum_i^n \dot{q}_{conv,i} + \sum_j^l \dot{q}_{transfer,j} \end{pmatrix} = \frac{d}{dt}\begin{pmatrix} M \\ U \end{pmatrix} = V\begin{pmatrix} 1 & 0 \\ u & \rho \end{pmatrix}\frac{d}{dt}\begin{pmatrix} \rho \\ u \end{pmatrix} \tag{3.3}$$

The convective heat flow is always calculated in a FM from the enthalpies in the connectors as follows (flow quantities are taken as positive **into** a component):

$$b.\dot{m} = -a.\dot{m} \tag{3.4}$$

$$a.q_{conv} = \text{if } a.\dot{m} > 0 \text{ then } a.\dot{m} \times a.h \text{ else } a.\dot{m} \times b.h \tag{3.5}$$

$$b.q_{conv} = -a.q_{conv} \tag{3.6}$$

With this definition, all control volume models are independent of the flow direction.

For flows with heat transfer the above form of the first law is not in all situations best suited for numerical simulation. Two main reasons exist, why it may be advantageous to choose other states than the total mass (or density) and the inner energy as states:

1. The coupling with the hydraulic equations may render the system very stiff.
2. The computationally often very expensive medium property models may be faster with other states as input variables.

3.3 Pressure and Enthalpy as States

If pressure and enthalpy are chosen as states, the first law and the mass balance can be rewritten into these states as follows:

$$\frac{d}{dt}\begin{pmatrix} \rho \\ u \end{pmatrix} = \underbrace{\begin{pmatrix} \left.\frac{\partial\rho}{\partial p}\right|_h & \left.\frac{\partial\rho}{\partial h}\right|_p \\ \left.\frac{\partial u}{\partial p}\right|_h & \left.\frac{\partial u}{\partial h}\right|_p \end{pmatrix}}_{\text{Jacobian MatrixJ}} \frac{d}{dt}\begin{pmatrix} p \\ h \end{pmatrix} \tag{3.7}$$

To obtain differential equations for pressure and enthalpy eq. (3.7) must be solved for the derivative of (p, h)

$$\frac{d}{dt}\begin{pmatrix} p \\ h \end{pmatrix} = \mathsf{J}^{-1}\frac{d}{dt}\begin{pmatrix} \rho \\ u \end{pmatrix} \tag{3.8}$$

The inverse of the Jacobian is computed as follows

$$\mathsf{J}^{-1} = \frac{1}{\det \mathsf{J}}\begin{pmatrix} \left.\frac{\partial u}{\partial h}\right|_p & -\left.\frac{\partial \rho}{\partial h}\right|_p \\ -\left.\frac{\partial u}{\partial p}\right|_h & \left.\frac{\partial \rho}{\partial p}\right|_h \end{pmatrix} \tag{3.9}$$

with the determinant

$$\det \mathsf{J} = \left.\frac{\partial \rho}{\partial p}\right|_h \left.\frac{\partial u}{\partial h}\right|_p - \left.\frac{\partial \rho}{\partial h}\right|_p \left.\frac{\partial u}{\partial p}\right|_h \tag{3.10}$$

The partial derivatives of u can be reduced to the ones of ρ. From $u = h - p/\rho$ we obtain

$$\left.\frac{\partial u}{\partial h}\right|_p = 1 + \frac{p}{\rho^2}\left.\frac{\partial \rho}{\partial h}\right|_p \qquad \left.\frac{\partial u}{\partial p}\right|_h = -\frac{1}{\rho} + \frac{p}{\rho^2}\left.\frac{\partial \rho}{\partial p}\right|_h \tag{3.11}$$

Therefore

$$\det \mathsf{J} = \left.\frac{\partial \rho}{\partial p}\right|_h + \frac{1}{\rho}\left.\frac{\partial \rho}{\partial h}\right|_p = \frac{1}{a^2} \tag{3.12}$$

where a is the velocity of sound. The inverse of the Jacobian becomes

$$\mathsf{J}^{-1} = a^2\begin{pmatrix} 1 + \frac{p}{\rho^2}\left.\frac{\partial \rho}{\partial h}\right|_p & -\left.\frac{\partial \rho}{\partial h}\right|_p \\ \frac{1}{\rho} - \frac{p}{\rho^2}\left.\frac{\partial \rho}{\partial p}\right|_h & \left.\frac{\partial \rho}{\partial p}\right|_h \end{pmatrix} \tag{3.13}$$

and the new system of differential equations, multiplied with the mass $M = \rho V$ is

$$V\frac{\rho}{a^2}\frac{dp}{dt} = \left(\rho + \frac{p}{\rho}\left.\frac{\partial \rho}{\partial h}\right|_p\right) V\frac{d\rho}{dt} - \left.\frac{\partial \rho}{\partial h}\right|_p M\frac{du}{dt} \tag{3.14}$$

$$V\frac{\rho}{a^2}\frac{dh}{dt} = \left(1 - \frac{p}{\rho}\left.\frac{\partial \rho}{\partial p}\right|_h\right) V\frac{d\rho}{dt} + \left.\frac{\partial \rho}{\partial p}\right|_h M\frac{du}{dt} \tag{3.15}$$

3.4 Density and Temperature as States

For certain applications, the previously mentioned stiffness is not a problem, e. g., when

- the region of operation is not in the liquid region or if it is, then at pressures above 80 % of the critical pressure or
- if there are only few liquid control volumes in the system and the stiffness that the liquid control volumes cause is handled by other means. This holds for refrigeration cycles.

Under these circumstances, density and temperature have the advantage that they are the base variables of almost all high accuracy medium property models, see 2.4 and therefore the calculation of the properties is non-iterative and fast. Another advantage is, that this model can very easily be extended to mixtures of media, but this has not been done in the current version of the library with the exception of mixtures of ideal gases.

$$\frac{d}{dt}\begin{pmatrix}\rho\\u\end{pmatrix} = \underbrace{\begin{pmatrix}1 & 0\\ \left.\frac{\partial u}{\partial \rho}\right|_T & \left.\frac{\partial u}{\partial T}\right|_\rho\end{pmatrix}}_{\text{Jacobian Matrix}} \frac{d}{dt}\begin{pmatrix}\rho\\T\end{pmatrix} \tag{3.16}$$

For ideal gases this simplifies because, $\left.\frac{\partial u}{\partial \rho}\right|_T = 0$ and in general it is common to write $\left.\frac{\partial u}{\partial T}\right|_\rho = c_v$, the heat capacity at constant volume.

3.5 Pressure and Temperature as States

Because of the dependence between pressure and temperature in the two phase region, these two variables can, under the assumption of thermodynamic equilibrium, not be used as states for the simulation. But outside the two phase region and for ideal gases, they have the advantage that these two variables are often readily available from measurements.

$$\frac{d}{dt}\begin{pmatrix}\rho\\u\end{pmatrix} = \underbrace{\begin{pmatrix}\left.\frac{\partial \rho}{\partial p}\right|_T & \left.\frac{\partial \rho}{\partial T}\right|_p\\ \left.\frac{\partial u}{\partial p}\right|_T & \left.\frac{\partial u}{\partial T}\right|_p\end{pmatrix}}_{\text{Jacobian Matrix}} \frac{d}{dt}\begin{pmatrix}p\\T\end{pmatrix} \tag{3.17}$$

For ideal gases, $\left.\frac{\partial u}{\partial p}\right|_T = 0$ and $\left.\frac{\partial u}{\partial T}\right|_p = c_v$

Pressure and temperature as states could be extended with e. g., the vapor mass fraction x inside the two phase region, but this has not been implemented in the library.

4. Hydraulic Models

Currently two types of hydraulic models are implemented in the library:

- A stationary pressure drop model
- A dynamic momentum balance for flows in pipes with constant cross-sectional area.

The hydraulic models are slightly different for lumped parameter models and distributed parameter models, see 4 for the difference in the base models. Then it should be noted that it makes only sense to model the pressure wave propagation through a system when it is mainly modeled with distributed parameter models. But in that case there will be lumped

parameter models that are part of the system e. g., simple valves, and they have to handle the momentum transport.

Keep in mind that it is possible to add different types of hydraulic models to the existing structure, e. g., a mometum balance for for a variable cross-sectional area along the flow channel.

$$\text{momentum} \qquad I = \int_V \rho w \, dV = \int_{\Delta z} \int_A \rho w \, dA dz = \dot{m} \Delta z \tag{4.1}$$

$$\text{mass flow rate} \qquad \Delta z \frac{d\dot{m}}{dt} = \frac{dI}{dt} = \dot{I}_1 - \dot{I}_2 + (p_1 - p_2)A - F_{wall} - F_{grav} \tag{4.2}$$

$$\text{wall friction force} \qquad F_{wall} = \xi \frac{\Delta z}{2D} \dot{m} |w| \tag{4.3}$$

$$\text{gravity forces} \qquad F_{grav} = g \; L \; \sin(\alpha \frac{\pi}{180}) \tag{4.4}$$

5. Classes

According to the library structure (Figure 2), this section will systematically describe the individual classes of the ThermoFluid library. After a description of the base classes, some of the partial components are presented, followed by a range of end-user components.

5.1 Interfaces

According to the Modelica standard, the Interfaces-class is situated at the top-level in the ThermoFluid library. Due to the diversity of the library, 4 different flow connectors are implemented (see also 2.2). Also, because the finite volume method is used with a staggered grid formulation, there needs to be a difference between two adjacent connectors. Thereby a design flow direction is introduced. This is described earlier in section 2.2. As a result of this, there are two of each connector, giving room for different graphical symbols (see figure 3). The rule of thumb is that the flow connector B (i. e., `FlowB`) is next to a flow model, whereas connector A (i. e., `FlowA`) is not related to a flow model within the component.

Apart from the graphics and the presence of a flow model in the respective component model, there is no difference in the two connectors A and B. Therefore, the actual connector is only implemented once in the connector `BaseFlow`. This is then inherited to the connectors `FlowA` and `FlowB`, which add the necessary graphical representation according to figure 3.

SingleStatic Starting with the most simple connector, covering single component medium with a static flow model, the flow connector for SingleStatic contains the following variables:

```
connector BaseFlow
  SIunits.Pressure p;
  SIunits.SpecificEnthalpy h;
  flow SIunits.MassFlowRate mdot;
  flow SIunits.Power q_conv;
```

```
  SIunits.Density d;
  SIunits.Temp_K T;
  SIunits.SpecificEntropy s;
  SIunits.RatioOfSpecificHeatCapacities kappa;
end BaseFlow;
```

SingleDynamic For a single component medium with dynamic flow model, two additional variables are added to the flow connector from SingleStatic. These are `G_norm` for the momentum flux, and `dG` for the change in the momentum flux over the component on the downstream side of the connector.

```
connector BaseFlow
  ...
  flow SIunits.MomentumFlux G_norm;
  SIunits.MomentumFlux dG;
end BaseFlow;
```

MultiStatic For multi component media, the connector has a vector, containing the mass fractions of the composition. Further, the mass flow rate `mdot` (from the single component case) is exchanged with a mass flow rate vector, `mdot_x`. The parameter nspecies defines the number of components in the medium.

```
connector BaseFlow
  parameter Integer nspecies(min=1);
  SIunits.MassFraction mass_x[nspecies];
  SIunits.Pressure p;
  SIunits.SpecificEnthalpy h;
  flow SIunits.MassFlowRate mdot_x[nspecies];
  flow SIunits.Power q_conv;
  SIunits.Density d;
  SIunits.Temp_K T;
  SIunits.RatioOfSpecificHeatCapacities kappa;
  SIunits.SpecificEntropy s;
end BaseFlow;
```

MultiDynamic The connector for multi component media with dynamic flow model also contains two variables for the momentum flux and the change in momentum flux over the component on the downstream side of the connector.

```
connector BaseFlow ... flow SIunits.MomentumFlux G_norm;
SIunits.MomentumFlux dG; end BaseFlow;
```

Shell components Following the tradition of implemented Modelica component libraries, the ThermoFluid library also offers a range of so-called shell components. These components merely form an empty shell with one, two or more connectors of the different types described earlier.

For each case of the earlier mentioned connector types, there are implemented the following shell components:

```
partial model OnePortA "One port for sinks, no flow model"
  // Typical one port for sinks
  replaceable FlowA a;
end OnePortA;
```

```
partial model TwoPortAA "Two port for lumped volumes"
  replaceable FlowA a;
  replaceable FlowA b;
end TwoPortAA;

partial model TwoPortAB "Two port for distributed volumes, with flow model"
  replaceable FlowA a;
  replaceable FlowB b;
end TwoPortAB;

partial model TwoPortBB "Two port for lumped flow models"
  replaceable FlowB a;
  replaceable FlowB b;
end TwoPortBB;
```

Aditionally, a series of three-ports are also implemented. Also here, the design flow direction has to be taken into account. Therefore there are 3 different three-ports implemented:

```
partial model ThreePortAAB "Three port for junction, w one flow model"
  replaceable FlowA a;
  replaceable FlowA b;
  replaceable FlowB c;
end ThreePortAAB;

partial model ThreePortABB "Three port for split, w two flow models"
  replaceable FlowA a;
  replaceable FlowB b;
  replaceable FlowB c;
end ThreePortABB;

partial model ThreePortAAA "Three port for volume, no flow models"
  replaceable FlowA a;
  replaceable FlowA b;
  replaceable FlowA c;
end ThreePortAAA;
```

HeatTransfer For dealing with heat transfer problems, e.g. heat exchangers, walls, heat sources, etc., a special interface class for heat flow has been implemented. The heat flow connector contains the following variables:

```
connector HeatFlowD "Heat flow connector, distributed"
  parameter Integer n(min=1) = 1;
  flow SIunits.Power q[n] "Heat flux";
  SIunits.Temp_K T[n] "Temperature";
end HeatFlowD;
```

This connector is discretized in order to build distributed component models, according to figure compartments 4. The parameter n specifies the number of discrete volumes in the component. This parameter has to match the discretization parameter of the distributed component that uses this connector. This structural parameter should be set once in the top level model that sets the discretization and from there propagated to all submodels using this discretization via *modifications*.

```
partial model OnePortD
  parameter Integer n(min=1) = 1;
  HeatFlowD qa(n=n) ;// ann.4
end OnePortD;

partial model TwoPortD
  parameter Integer n(min=1) = 1;
  HeatFlowD qa(n=n);
  HeatFlowD qb(n=n);
end TwoPortD;
```

6. Base classes

This section will describe the base classes in the ThermoFluid library. The base classes represent the modeling principles that are introduced in earlier chapters, 2.4, 3 & 4. Further, the base classes also include the state transformations mentioned in 3.1.

- Balances
- FlowModels
- MediumModels
- StateTransformations
- CommonFunctions
- CommonRecords

`CommmonFunctions` contains functions used indifferent parts of the library. `CommonRecords` on the other hand contains the variable and parameter sets that are used throughout the library. This section will give an overview of these classes and how they depend on each other.

6.1 CommonRecords

CommonRecords contains record and variable definitions that are general for the library base classes.

ThermoBaseVars Starting with the records that define the variable set, which is necessary for using the base classes, the basis record is called `ThermoBaseVars`.

```
record ThermoBaseVars
  parameter Integer n(min=1) = 1 "discretization number";
  SIunits.Pressure p[n] "Pressure";
  SIunits.Temperature T[n] "temperature";
  SIunits.Density d[n] "density";
  SIunits.SpecificEnthalpy h[n] "enthalpy";
  SIunits.SpecificEntropy s[n] "entropy";
  SIunits.RatioOfSpecificHeatCapacities kappa[n] "ratio of cp/cv";
  SIunits.Mass M[n](start=ones(n), fixed=false) "Total mass";
  SIunits.Energy U[n](start=ones(n), fixed=false) "Inner energy";
  SIunits.MassFlowRate dM[n] "Change in total mass";
  SIunits.Power dU[n] "Change in inner energy";
  SIunits.Volume V[n] "Volume";
end ThermoBaseVars;
```

For mixtures, i. e., multi component media, additional base variable are collected in `MixtureVariables`

```
record MixtureVariables "additional variables for homogeneous mixtures of fluids"
  parameter Integer n(min=1) = 1 "discretization number";
  parameter Integer nspecies(min=2) "number of species";
  SIunits.MassFraction[n,nspecies] mass_x "mass fraction";
  SIunits.MoleFraction[n,nspecies] mole_y "mole fraction";
    SIunits.Mass[n,nspecies] M_x(start=ones(n,nspecies), fixed=false) "compo-
nent M";
  SIunits.Mass[n] M "total mass";
  SIunits.Volume[n] V "volume";
  SIunits.Density[n] d "density";
end MixtureVariables;
```

ThermoProperties For practical reasons when calculating medium properties, it was necessary to create a record that contains the relevant medium property variables. This record includes the relevant derivatives of the state variables. The actual calculation of the medium properties is handled by a function. The return variable of this function is declared by this record. Here, the record for properties using the ph state model is shown:

```
record ThermoProperties_ph
  "Thermodynamic property data for p and h as states"
  SIunits.Temp_K T "temperature";
  SIunits.Density d "density";
  SIunits.SpecificEnergy u "inner energy";
  SIunits.SpecificEntropy s "entropy";
  SIunits.SpecificHeatCapacity cp "heat capacity at constant pressure";
  SIunits.SpecificHeatCapacity cv "heat capacity at constant volume";
  SIunits.SpecificHeatCapacity R "gas constant";
  SIunits.RatioOfSpecificHeatCapacities kappa "ratio of cp/cv";
  SIunits.Velocity a "speed of sound";
  SIunits.DerDensityByEnthalpy ddhp "deriv. of d by h at constant p";
  SIunits.DerDensityByPressure ddph "deriv. of d by p at constant h";
end ThermoProperties_ph;
```

Available thermo property records are (according to section 3):

- `ThermoProperties_ph`
- `ThermoProperties_pT`
- `ThermoProperties_dT`
- `ThermoProperties_pTX`
- `ThermoProperties_dTX`

The latter two are used for multi component mixtures, e. g., mixtures of gases. The rule of thumb here is that the record for ph as a state model does not contain the variables p and h, the record for pT as state model does not contain the variables p and T, etc.

StateVariables The two variable sets (ThermoProperties and ThermoBaseVars) are combined in main base class StateVariables, defining the variable-structure that should be used by the component models.

For ph state models, `StateVariables_ph` is shown here:

Figure 6 Classes for connecting variables

In the balances class the variables `pdown`, `ddown` and `dGdown` are assigned:

```
ddown = b.d;
pdown = b.p;
dGdown = b.dG;
```

The according connector variables (`b.d`, `b.p` and `b.dG`) are assigned in the following (downstream) component:

```
p[1] = a.p;
d[1] = a.d;
G_norm[1] - G_norm[2] = a.dG;
```

Notice that the flow models are formulated using central differences. Therefore, `a.dG` represents only half the central difference for the flow model used in the last compartment of the previous component.

```
dG[n] = 0.5*(G_norm[n] - G_norm[n+1] + dGdown);
dz*der(mdot[n+1]) = dG[n] + (p[n] - pdown)*A - ...
```

Here, also information about the pressure of the downstream component is used. The last connecting variable, `ddown`, is used to calculate the momentum flux, depending on the flow direction:

```
G_norm[n+1] = if mdot[n+1] > 0 then
                mdot[n+1]*mdot[n+1]/d[n]/A
              else
                -mdot[n+1]*mdot[n+1]/ddown/A;
```

PressureLoss For making the pressure loss a replaceable model, PressureLossLumped and PressureLossDistributed are some basic records. They are usedPloss to build actual pressure loss models later. Here, only the variable is declared. In the actual model, `Ploss` should depend on `mdot`.

```
partial model PressureLossLumped
  SIunits.Pressure Ploss[1];
end PressureLossLumped;

partial model PressureLossDistributed
  parameter Integer n(min=1) = 1;
  SIunits.Pressure Ploss[n];
end PressureLossDistributed;
```

6.2 CommonFunctions

Available functions in this class are:

- `rad2deg (input Real rad, output Real deg)`
- `deg2rad (input Real deg, output Real rad)`
- `ThermoRoot (input Real x,dx, output Real y)`
- `CubicSplineEval (input Real x,coefs[4], output Real y)`
- `CubicSplineDerEval (input Real x,coefs[4], output Real yder)`
- `FindInterval (input Real x,breaks[:], output Integer i)`

Whereas the first two functions are kind of self-explaining, the third needs some explanation.

ThermoRoot The `ThermoRoot` function is a square root function that enables operation close to zero. When doing numerical calculations, it can sometimes happen that the solver calls the square root function with a negative input. For physical systems, this gives no meaning, and the solver would typically break down. The `ThermoRoot` function has a smooth transition through (0,0) and can be called with a negative argument, adding stability to the numerical solver at this point. The size of the region around (0,0) is given by the second input argument, dx. For further information, the reader is referred to the function itself in `CommonRecords.mo`.

CubicSplineEval

CubicSplineDerEval

FindInterval

6.3 Balances

In Balances the mass and energy balances are set up according to section 3.2. The Balances subdirectory contains 4 different packages, each corresponding to one of the interface classes described in section 2.2 and later in section 5.1. For each of these situations, there are two general classes according to section 2.3. Thus, for the most common component with two connectors (hence the name two-port), there are a total of 8 classes describing the balance equations.

Actually, the balance equations for mass and energy are rather indifferent to the existence of either the static or dynamic momentum balance. But since the balance classes also take care of connecting the internal variables to the connectors, there is a slight difference anyway.

Mass and energy balances typically follow each other. Therefore, it was not considered necessary to model these in individual classes. So the following classes contain both mass and energy balances.

Single First, the necessary variable sets are set up for the lumped case. For practical as well as structural reasons, these variables sets are modeled in individual `records`.

```
record SingleLumped "Balance variables"
  extends CommonRecords.ThermoBaseVars(final n=1);
  SIunits.Power W_t[1] "Technical work term";
  SIunits.Power Q_s[1] "Heat source term";
end SingleLumped;
```

and for the distributed case

```
record SingleDistributed "Balance variables, distributed"
  extends CommonRecords.ThermoBaseVars;
  SIunits.Power W_t[n] "Technical work term";
  SIunits.Power Q_s[n] "Heat source term";
  SIunits.MassFlowRate mdot[n+1];
  SIunits.Power edot[n+1];
end SingleDistributed;
```

These variable sets are then inherited by the two-ports and used to form the mass and energy balances. Here the lumped formulation

```
partial model TwoPortLumped "Balance model for lumped models"
  extends Interfaces.TwoPortAA;
  extends SingleLumped;
equation
  ...
  dM[1] = a.mdot + b.mdot;
  dU[1] = a.q_conv + b.q_conv - p[1]*der(V[1]) + Q_s[1] + W_t[1];
end TwoPortLumped;
```

and the distributed formulation

```
partial model TwoPortDistributed  "Balances for distributed models"
  extends Interfaces.TwoPortAB;
  extends SingleDistributed;
  extends CommonRecords.ConnectingVariablesSingleStatic;
equation
  ...
  mdot[1] = a.mdot;
  mdot[n + 1] = -b.mdot;
  edot[1] = a.q_conv;
  edot[n+1] = -b.q_conv;
  for i in 2:n loop
    edot[i] = if mdot[i] > 0 then mdot[i]*h[i-1] else mdot[i]*h[i];
  end for;
  edot[n+1] = if mdot[n+1] > 0 then mdot[n+1]*h[n] else mdot[n+1]*b.h;
  dM = mdot[1:n] - mdot[2:n+1];
  for i in 1:n loop
    dU[i] = edot[i] - edot[i+1] - p[i]*der(V[i]) + Q_s[i] + W_t[i];
  end for;
end TwoPortDistributed;
```

As it can be seen from `TwoPortDistributed` it is possible to write the mass balances for the individual compartments in vector form. Due to the more complex nature of the energy balance, this has to be written in a FOR-loop.

Also, the lumped case is inherited from `Interfaces.TwoPortAA`, meaning that in the lumped formulation this component does not contain any flow model. Later, the respective flow model is declared with `Interfaces.TwoPortBB`. Thus the rule, that only unequal connectors can be connected (i. e., A and B), is preserved. The distributed model is inherited from `Interfaces.TwoPortAB`. As it can be seen from the naming convention, this model is prepared to be mated with the respective flow model to form a control volume, which contains a flow model.

These classes are actually general for both the static and the dynamic flow cases, except for the equations that connect the variables of the component to the variables of the connectors. For the dynamic flow model case additional equations for connecting `G_norm` and `dG` are needed. Here, `G_norm` is the momentum flux, normal to the control surface, and `dG` is the change in momentum flux over the adjacent compartment.

For the lumped case:

```
a.dG = a.G_norm + b.G_norm;
b.dG = a.G_norm + b.G_norm;
```

is added, and for the distributed case:

```
  G_norm[1] = a.G_norm;
  G_norm[n+1] = -b.G_norm;
  G_norm[1] - G_norm[2] = a.dG;
```

Multi For multi component media, the energy and mass balances look slightly different. The variable sets are based on the variable sets from single component flows. Here, vectors and matrices for mass fractions and flow rates are added.

```
record MultiLumped "Balance variables, multi component"
  extends SingleStatic.SingleLumped;
  parameter Integer nspecies(min=2) = 10;
  SIunits.Mass M_x[1, nspecies];
  SIunits.MassFraction mass_x[1, nspecies];
end MultiLumped;

partial model MultiDistributed "Balance variables, distributed, multi component"
  extends SingleStatic.SingleDistributed;
  parameter Integer nspecies(min=2) = 10;
  SIunits.Mass M_x[n, nspecies];
  SIunits.MassFraction mass_x[n, nspecies];
  SIunits.MassFlowRate mdot_x[n + 1, nspecies];
end MultiDistributed;
```

`nspecies` denotes the number of species, i. e., components, of the medium, `mass_x` is a vector containing the mass fractions of the individual components and `M_x` are the masses of the individual components inside a compartment. For distributed component there is an additional vector, `mdot_x`, containing the mass flow rates of the individual components of the medium.

These variable sets are then used to form the mass and energy balances

```
partial model TwoPortLumped
  extends Interfaces.TwoPortAA;
  extends MultiLumped;
  Real dMx[1, nspecies];
equation
  ...
  mass_x[1, :] = b.mass_x;
  mass_x[1, :] = a.mass_x;
  dU[1] = a.q_conv + b.q_conv - p[1]*der(V[1]) + Q_s[1] + W_t[1];
  dM_x[1,:] = a.mdot_x + b.mdot_x;
  dM[1] = sum(dM_x[1,:]);
  mass_x[1,:]*d[1]*V[1] = M_x[1,:];
end TwoPortLumped;
```

and for the distributed case

```
partial model TwoPortDistributed
  extends Interfaces.TwoPortAB;
  extends MultiDistributed;
  extends CommonRecords.ConnectingVariablesMultiStatic;
equation
  ...
  mdot[1] = sum(a.mdot_x);
  edot[1] = a.q_conv;
  edot[n+1] = -b.q_conv;
  mdot_x[1,:] = a.mdot_x;
```

```
  for i in 2:n loop
    edot[i] = if mdot[i] > 0 then mdot[i]*h[i-1] else mdot[i]*h[i];
   mdot_x[i,:] = if mdot[i] > 0 then mdot[i]*mass_x[i-1,:] else mdot[i]*mass_x[i,:];
  end for;
  edot[n+1] = if mdot[n+1] > 0 then mdot[n+1]*h[n] else mdot[n+1]*b.h;
 mdot_x[n+1,:] = if mdot[n+1] > 0 then mdot[n+1]*mass_x[n,:] else mdot[n+1]*b.mass_x;
  mdot_x[n+1,:] = -b.mdot_x;
  mass_x[1,:] = a.mass_x;
  mass_x[n,:] = b.mass_x;
  dM_x = mdot_x[1:n,:] - mdot_x[2:n + 1,:];
  for i in 1:n loop
    dM[i] = sum(M_x[i,:]);
    dU[i] = edot[i] - edot[i+1] - p[i]*der(V[i]) + Q_s[i] + W_t[i];
    M_x[i,:] = mass_x[i, :]*d[i]*V[i];
  end for;
end TwoPortDistributed;
```

Again these classes are general for both the static and the dynamic flow cases. For the dynamic flow model case additional equations for connecting `G_norm` and `dG` are needed.

For the lumped case:

```
a.dG = a.G_norm + b.G_norm;
b.dG = a.G_norm + b.G_norm;
```

and for the distributed case:

```
G_norm[1] = a.G_norm;
G_norm[n+1] = -b.G_norm;
G_norm[1] - G_norm[2] = a.dG;
```

These equations will be further explained in the following subsection, describing the flow models.

6.4 FlowModels

Whereas there was little difference between the balance models for static and dynamic flow models, there is of course a significant difference when it comes to the flow models themselves. Therefore, in FlowModels there are models covering each case, i. e., single/multi component media, static/dynamic flow models and lumped/distributed. This gives a total of $2^3 = 8$ different classes.

These classes are described systematically in the following subsections.

SingleStatic For single component media with static flow conditions, the momentum balance reduces to a simple pressure loss equations. The pressure loss over a compartment is represented by the term `Ploss`. For structural reasons, the pressure loss term is encapsulated in a partial model and inherited replaceable. This makes it possible to redeclare the pressure loss later. The basic pressure loss model, which in fact is only a record containing the term Ploss, is declared in CommonRecords, see section 6.1.

For the lumped flow model, `q_conv` is calculated at the connector in the flow model and used in the energy balance equation, see section 6.3. The momentum balances is reduced to a pressure loss relationship, using `Ploss` and a term for the pressure loss due to gravity.

```
model TwoPortLumped "Lumped Flow model, including connectors"
  extends Interfaces.TwoPortBB;
  extends CommonRecords.BaseGeometryVars;
  SIunits.MassFlowRate[1] mdot;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossLumped;
  end PressureLoss;
  extends PressureLoss;
equation
  a.q_conv = if mdot[1] > 0 then a.h*mdot[1] else b.h*mdot[1];
  a.q_conv = -b.q_conv;
  dz = L;
  mdot[1] = a.mdot;
  mdot[1] = -b.mdot;
  0 = (a.p-b.p)*A - Ploss[1]*dz*Dhyd*Pi -
      A*d[1]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortLumped;
```

As we can see for the distributed case, `q_conv` is not calculated here, but in Balances (section 6.3).

```
model TwoPortDistributed "Distributed flow model, to be used in CVs"
  parameter Integer n(min=1) = 5;
  extends CommonRecords.ConnectingVariablesSingleStatic;
  extends CommonRecords.BaseGeometryVars;
  extends Balances.SingleDistributed;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossDistributed;
  end PressureLoss;
  extends PressureLoss;
equation
  dz = L/n;
  for i in 1:n - 1 loop
    0 = (p[i]-p[i+1])*A - Ploss[i]*dz*Dhyd*Pi -
        A*d[i]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
  end for;
  0 = (p[n]-pdown)*A - Ploss[n]*dz*Dhyd*Pi -
      A*d[n]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortDistributed;
```

A FOR-loop is used to write the pressure loss relationship for all compartments in the distributed model. Except for the last compartmen, where `pdown` is used. This variable comes from ConnectingVariables that are used together with the connectors to pass information between adjacent models.

As mentioned earlier, the term `Ploss` should depend on the mass flow rate `mdot`, giving an implicit equation for `mdot`.

SingleDynamic Unlike the mass and energy balances, which were similar for static and dynamic cases, the flow model (or momentum balances) classes are different, obviously.

```
model TwoPortLumped "Lumped Flow model, including connectors"
  extends Interfaces.TwoPortBB;
  extends CommonRecords.BaseGeometryVars;
  SIunits.MassFlowRate[1] mdot;
  SIunits.MomentumFlux[1] G_norm;
```

```
  SIunits.MomentumFlux[1] dG;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossLumped;
  end PressureLoss;
  extends PressureLoss;
equation
  a.q_conv = if mdot[1] > 0 then a.h*mdot[1] else b.h*mdot[1];
  a.q_conv = -b.q_conv;
  dz = L;
  mdot[1] = a.mdot;
  mdot[1] = -b.mdot;
  G_norm[1] = if mdot[1] > 0 then
      mdot[1]*mdot[1]/a.d/A else -mdot[1]*mdot[1]/b.d/A;
  dG[1] = a.dG + b.dG;
  dz*der(mdot[1]) = dG[1] + (a.p-b.p)*A - Ploss[1]*dz*Dhyd*Pi
      - A*d[1]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortLumped;
```

The momentum balances is implemented according to equation 4.2. `dG` is the change in momentum flux over two adjacent compartment, aka a central difference. For the lumped model, `dG` sums up the contributions from the adjacent volumes on either side, where the actual differences are calculated. `Ploss` denotes the pressure loss due to wall friction. The last term represents the pressure loss due to gravitational forces.

As it can be seen in the code, the calculations of `G_norm` actually depend on the flow directions. This ensures that the right densities are used (i. e., `a.d` or `b.d`).

```
model TwoPortDistributed "Distributed flow model, to be used in CVs"
  parameter Integer n(min=1) = 5;
  extends CommonRecords.ConnectingVariablesSingleDynamic;
  extends CommonRecords.BaseGeometryVars;
  extends Balances.SingleDistributed;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossDistributed;
  end PressureLoss;
  extends PressureLoss;
equation
  dz = L/n;
  for i in 2:n loop
    G_norm[i] = if mdot[i] > 0 then
        mdot[i]*mdot[i]/d[i-1]/A else -mdot[i]*mdot[i]/d[i]/A;
  end for;
  G_norm[n+1] = if mdot[n+1] > 0 then
        mdot[n+1]*mdot[n+1]/d[n]/A else -mdot[n+1]*mdot[n+1]/ddown/A;
  for i in 1:n - 1 loop
    dG[i] = 0.5*(G_norm[i] - G_norm[i+2]);
    dz*der(mdot[i+1]) = dG[i]+(p[i]-p[i+1])*A - Ploss[i]*dz*Dhyd*Pi
        - A*d[i]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
  end for;
  dG[n] = 0.5*(G_norm[n] - G_norm[n+1] + dGdown);
  dz*der(mdot[n+1]) = dG[n] + (p[n]-pdown)*A - Ploss[n]*dz*Dhyd*Pi -
          A*d[n]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortDistributed;
```

It can be noticed that the momentum flux difference `dG` is calculated

in the flow model using central differences. For obvious reasons though, the central difference cannot be used in general for all compartments. The last `dG` needs an additional contribution from the adjacent model through `dGdown`, like

```
dG[n] = 0.5*(G_norm[n] - G_norm[n+1] + dGdown);
```

`dGdown` is calculated in the balance classes (see section 6.3) and transferred via the `dG` variable in the connectors (see section 5.1).

MultiStatic The difference between flow models for multi and single component media are not significant. However, some of the extra base variables (according to section 6.1 need to be calculated.

```
model TwoPortLumped "Lumped Flow model, including connectors"
  extends Interfaces.TwoPortBB;
  extends CommonRecords.BaseGeometryVars;
  SIunits.MassFlowRate[1] mdot;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossLumped;
  end PressureLoss;
  extends PressureLoss;
equation
  a.q_conv = if mdot[1] > 0 then a.h*mdot[1] else b.h*mdot[1];
  a.q_conv = -b.q_conv;
  a.mdot_x = if mdot[1] > 0 then a.mass_x*mdot[1] else b.mass_x*mdot[1];
  a.mdot_x = -b.mdot_x;
  dz = L;
  mdot[1] = a.mdot;
  mdot[1] = -b.mdot;
  0 = (a.p - b.p)*A - Ploss[1]*dz*Dhyd*Pi
      - A*d[1]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortLumped;
```

In the lumped flow model, the mass flow rates of the individual components of the media (`mdot_x`) need to be calculated. The distributed static flow model is identical with the distributed static flow model for single component flows.

```
model TwoPortDistributed "Distributed flow model, to be used in CVs"
  parameter Integer n(min=1) = 5;
  extends CommonRecords.ConnectingVariablesMultiStatic;
  extends CommonRecords.BaseGeometryVars;
  extends Balances.MultiDistributed;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossDistributed;
  end PressureLoss;
  extends PressureLoss;
equation
  dz = L/n;
  for i in 1:n - 1 loop
    0 = (p[i] - p[i+1])*A - Ploss[i]*dz*Dhyd*Pi
        - A*d[i]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
  end for;
  0 = (p[n] - pdown)*A - Ploss[n]*dz*Dhyd*Pi
      - A*d[n]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortDistributed;
```

MultiDynamic Also for multi component flows with dynamic flow model the only difference to single component flows lies in the calculation of the individual mass flow rates in the lumped case.

```
model TwoPortLumped "Lumped Flow model, including connectors"
  extends Interfaces.TwoPortBB;
  extends CommonRecords.BaseGeometryVars;
  SIunits.MassFlowRate[1] mdot;
  SIunits.MomentumFlux[1] G_norm;
  SIunits.MomentumFlux[1] dG;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossLumped;
  end PressureLoss;
  extends PressureLoss;
equation
  a.q_conv = if mdot[1] > 0 then a.h*mdot[1] else b.h*mdot[1];
  a.q_conv = -b.q_conv;
  a.mdot_x = if mdot[1] > 0 then a.mass_x*mdot[1] else b.mass_x*mdot[1];
  a.mdot_x = -b.mdot_x;
  dz = L;
  mdot[1] = a.mdot;
  mdot[1] = -b.mdot;
  G_norm[1] = if mdot[1] > 0 then
     mdot[1]*mdot[1]/a.d/A else -mdot[1]*mdot[1]/b.d/A;
  dG[1] = a.dG + b.dG;
  dz*der(mdot[1]) = dG[1] + (a.p - b.p)*A - Ploss[1]*dz*Dhyd*Pi
      - A*d[1]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortLumped;
```

For the distributed case there is no difference.

```
model TwoPortDistributed "Distributed flow model, to be used in CVs"
  parameter Integer n(min=1) = 5;
  extends CommonRecords.ConnectingVariablesMultiDynamic;
  extends CommonRecords.BaseGeometryVars;
  extends Balances.MultiDistributed;
  replaceable model PressureLoss
    extends CommonRecords.PressureLossDistributed;
  end PressureLoss;
  extends PressureLoss;
equation
  dz = L/n;
  for i in 2:n loop
    G_norm[i] = if mdot[i] > 0 then
        mdot[i]*mdot[i]/d[i-1]/A else -mdot[i]*mdot[i]/d[i]/A;
  end for;
  G_norm[n+1] = if mdot[n+1] > 0 then
      mdot[n+1]*mdot[n+1]/d[n]/A else -mdot[n+1]*mdot[n+1]/ddown/A;
  for i in 1:n - 1 loop
    dG[i] = 0.5*(G_norm[i] - G_norm[i+2]);
    dz*der(mdot[i+1]) = dG[i] + (p[i] - p[i+1])*A - Ploss[i]*dz*Dhyd*Pi
        - A*d[i]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
  end for;
  dG[n] = 0.5*(G_norm[n] - G_norm[n+1] + dGdown);
  dz*der(mdot[n+1]) = dG[n] + (p[n] - pdown)*A - Ploss[n]*dz*Dhyd*Pi
        - A*d[n]*Constants.g_n*dz*Math.sin(alpha*Pi/180.0);
end TwoPortDistributed;
```

6.5 MediumModels

The medium models are constructed in such a way, that a function call connects the remaining state variables from `ThermoBaseVars` to the property record `ThermoProperty_xx`, where xx denotes the actual state model.

The class `WaterSteamPhase_ph` calculates the current phase of the medium. This is necessary for the function call of `water_ph` in order for the function to detect the phase boundaries correctly.

```
model WaterSteamPhase_ph
  extends CommonRecords.StateVariables_ph;
  parameter Integer mode=0;
  Integer phase[n];
protected
  Boolean Supercritical[n];
equation
  for i in 1:n loop
    Supercritical[i] = noEvent(p[i] > SteamIF97.data.PCRIT);
    phase[i] = if ((h[i] < SteamIF97.hlofp(p[i]))
                or (h[i] > SteamIF97.hvofp(p[i]))
                or Supercritical[i])
               then
                 1
               else
                 2;
  end for;
end WaterSteamPhase_ph;
```

The following example shows the construction for water and steam, using p and h as state variables.

```
model WaterSteamMedium_ph
  parameter Integer n=1;
  extends WaterSteamPhase_ph;
equation
  for i in 1:n loop
    pro[i] = water_ph(p[i], h[i], phase[i]);
  end for;
end WaterSteamMedium_ph;
```

This finishes the connection of variables that were started in section 6.1. The dependencies are also shown in figure 7.

6.6 StateTransforms

According to section 3.1, the primary differential equations (i. e., mass and energy balance) are transformed in the class StateTransforms into differential equations in the state variable pair of the individual case.

For p and h as state variables, the state transformations become

```
partial model ThermalModel_ph
  replaceable model Medium
    extends CommonRecords.StateVariables_ph;
  end Medium;
  extends Medium;
protected
  Real km[n];
```

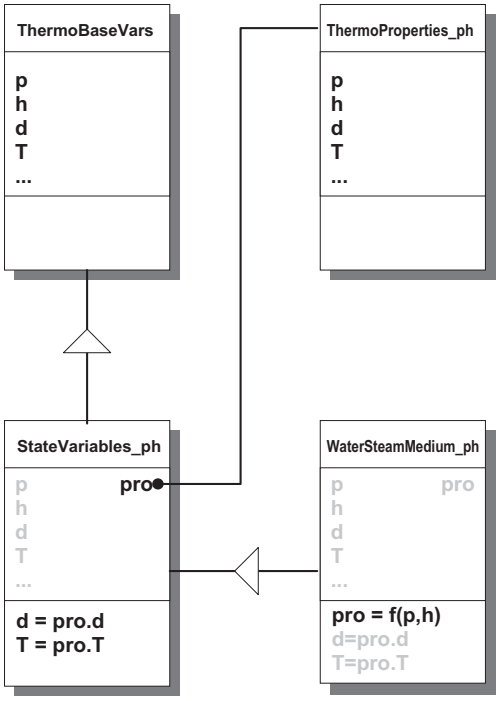

Figure 7 Class hierarchy of variables in the base classes

```
  Real kp[3, n];
  Real kh[3, n];
equation
  for i in 1:n loop
    km[i] = V[i]*(pro[i].ddph*d[i] + pro[i].ddhp);
    kp[1, i] = d[i] + h[i]*pro[i].ddhp;
    kp[2, i] = -pro[i].ddhp;
    kp[3, i] = -d[i]*d[i] - pro[i].ddhp*p[i];
    kh[1, i] = 1 - h[i]*pro[i].ddph;
    kh[2, i] = pro[i].ddph;
    kh[3, i] = pro[i].ddph*p[i] - d[i];
    km[i]*der(p[i]) = kp[1, i]*dM[i] + kp[2, i]*dU[i] + kp[3, i]*der(V[i]);
    km[i]*der(h[i]) = kh[1, i]*dM[i] + kh[2, i]*dU[i] + kh[3, i]*der(V[i]);
    M[i] = d[i]*V[i];
    U[i] = pro[i].u*M[i];
  end for;
end ThermalModel_ph;
```

Here, some of the derivatives calculated by the medium property routines and delivered by the ThermoProperties record are used to transform the differential equations.

Available state transformations are:

- `ThermalModel_ph`
- `ThermalModel_pT`
- `ThermalModel_dT`
- `ThermalModel_pTX`
- `ThermalModel_dTX`

7. Partial components

This section will describe some of the partial components that are implemented in the ThermoFluid library. These classes can of course be expanded. Given the flexibility of the library, arbitrarily many partial components can be implemented. At this point, the library only contains some of the most basic components. The purpose of this documentation is only to demonstrate the structure of the library. Consequently, this documentation only contains a few partial components.

- ControlVolumes
- ThreePorts
- Pumps
- Pipes
- Sources
- Compressors
- Walls

7.1 ControlVolumes

The best way of demonstrating how the base classes can be used to build component models, is by showing how to build a model for a control volume. The control volume should thus have a mass and energy balance, a flow model and the state transformations.

```
model Volume2PortDD_ph "Two port volume, distributed w flow model"
  extends Balances.SingleDynamic.TwoPortDistributed;
  extends FlowModels.SingleDynamic.TwoPortDistributed;
  extends StateTransforms.ThermalModel_ph;
  Heat.HeatFlowD q(n=n);
equation
  Q_s = q.q;
  T = q.T;
end Volume2PortDD_ph;
```

Here the control volume is showed with an additional heat flow connector. The variables of the heat flow connector, q and T, are connected to the variables Q_s and T. If the control volume were adiabatic, the heat flow connector could be omitted, and the heat flow into the control volume could simply be set to zero

```
Q_s = zeros(n);
```

A lumped control volume consists by definition (section 2.3) only of mass and energy balances. The flow model should then be modeled independently. A small, adiabatic, lumped control volume with static momentum balance would then look like

```
model Volume2PortSA_ph "Two port volume, lumped adiabatic"
  extends Balances.SingleStatic.TwoPortLumped;
  extends StateTransforms.ThermalModel_ph;
  CommonRecords.BaseGeometryPars geo;
equation
  V[1] = geo.V;
  Q_s[1] = 0.0;
end Volume2PortSA_ph;
```

What makes these models partial is the absence of information about the media contained in the control volume. Further, the presented control volumes do not contain a valid pressure loss model (i. e., the pressure loss due to wall friction is zero). These information should be added in end-user components.

7.2 ThreePorts

ThreePorts are the base classes for models of flow splitters and flow junctions. All of them correctly handle all possible combinations of flow directions in a flow splitter or junction in one model. ThreePorts come in two flavors, each with advantages and disadvatages:

- they consist of a normal control volume, but with 3 connectors
- they are an idealized control volume of 0 volume with no mass.

The normal control volume needs a medium model for the usual set of medium properties and contains thus more variables as a standard CV. Therefore it is as well necessary to build specialiaztions for each medium.

For the idealized CV's (named *IdealThreePorts*) there are two possibilities: either a real medium model is called with the algebraic pseudo-states (only p and h will work here) or the medium properties which are needed in the adjacent FM are fixed via parameters. This is done in Components.ThreePorts and is a very reasonable assumption in many cases.

The mass, component mass and energy balances in a 0-volume three port are automatically assembled by the Modelica translator from the balance classes for ThreePorts when the correct constraints are given. The result is an equation which e. g., calcules the specific enthalpy in the node as

$$h_{mix} = \frac{\sum \dot{m}_{in} h_{in}}{\sum \dot{m}_{in}} = \frac{\sum q_{conv}}{\sum \dot{m}_{in}}$$

In the generated code this equation is difficult to recognize because of the many IF-structures for all flow directions.

7.3 Pumps

7.4 Pipes

7.5 Sources

The class hierarchy of the sources is rather complex compared to many of the other classes.

A reservoir is a component with one port (`OnePortA`) and a full medium model. Depending on the application of the reservoir, the parameters determining the state of the medium can be p and h, p and T or d and T.

Basically the source consists of a reservoir and a flow model. The flow model is a component with two connectors (`TwoPortBB`). Finally, the reservoir is connected to the flow model, and the flow model is connected to the source output.

```
model Source_pT
  parameter SIunits.Pressure p0=1.2e5;
  parameter SIunits.SpecificEnthalpy T0=300.0;
```

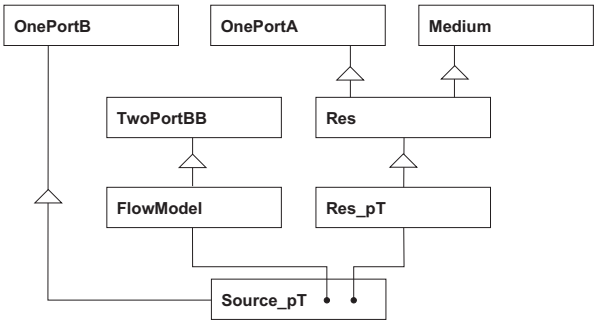

Figure 8 Class hierarchy of reservoirs

```
  parameter SIunits.Pressure dp0=5.0e4;
  parameter SIunits.MassFlowRate mdot0=5.0;
  parameter SIunits.Area A=0.1;
  replaceable PartialComponents.Valves.LinearLoss lloss(
    A=A,
    dp0=dp0,
    mdot0=mdot0);
  replaceable Res_pT res_pT(p0=p0, T0=T0);
equation
  connect(lloss.b, b);
  connect(res_pT.a, lloss.a);
end Source_pT;
```

It can be seen from the code above that the reservoir and the flow models are replaceable in the source. So in order to make a source for a particular medium, the reservoir has to be replaced with a reservoir of that particular medium, and the flow model can be replaced if necessary. Of course there should be given necessary attention to the replaceable connectors as well. Depending on the type of application, the default connectors `SingleStatic` have to be replaced by the appropriate connectors `SingleDynamic`, `MultiStatic` or `MultiDynamic` as well as in the reservoir and the source as in the flow model. For instructions on how to make new reservoirs and sources, the interested reader is referred to [**?**]

7.6 Compressors

7.7 Walls

8. Components

To demonstrate the applicability of the library, a broad range of ready-to-use components are likewise provided. To document them all would be too much. Instead, only some of the most basic components are listed here.

The Components library is categorized after the medium to which it applies. The following sub-libraries are implemented so far:

- Water
- Air

- Heat

Further documentation of the implemented components will not be done here. The reader is referred to the HTML documentation of the library.

9. Examples

This chapter will be explaining the examples in the package *Examples* of the ThermoFluid Library. We will describe here how they were put together from their parts and what the underlying assumptions are. The examples are kept simple, but give a flavor of what can be done with the library.

10. References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, New York, Berlin, 1996.

[2] The Modelica Association. Modelica language specification, version 1.3. www.Modelica.org/Documents, 1999.

[3] Mark O. McLinden, Sanford A. Klein, Eric W. Lemmon, and Adele P. Peskin. *NIST Thermodynamic and Transport Properties of Refrigerants and Refrigerant Mixtures—REFPROP*. U. S. Department of Commerce, version 6.0 edition, January 1998.

[4] Suhas V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Hemisphere Publishing Corporation, 1980.

[5] Robert C. Reid, John M. Prausnitz, and Bruce E. Poling. *The Properties of Gases and Liquids*. Mc Graw Hill, Boston, Massachusetts, 1987.

[6] Wolfgang Wagner and Alfred Kruse. *Properties of water and steam*. Springer, Berlin, 1998.