Thermodynamic Property and Fluid Modeling with Modern Programming Language Constructs

Martin Otter[1] Hilding Elmqvist[2] Dirk Zimmer[1] Christopher Laughman[3]

[1]DLR - Institute of System Dynamics and Control, Germany
{martin.otter, dirk.zimmer}@dlr.de
[2]Mogram AB, Magle Lilla Kyrkogata 24, 223 51 Lund, Sweden, Hilding.Elmqvist@Mogram.net
[3]Mitsubishi Electric Research Laboratories, Cambridge, MA, USA, laughman@merl.com

Abstract

Modelica is used extensively to model thermo-fluid pipe networks. Experience shows that Modelica models in this domain have limitations due to missing functional expressiveness of the Modelica language. In this paper, a prototype is described that demonstrates how thermodynamic property and thermo-fluid pipe component modeling could be considerably enhanced via modern language constructs. This prototype is based on the Modia modelling and simulation prototype and relies on features of the Julia programming language. It utilizes some key ideas of Modelica.Media, and part of Modelica.Media was semi-automatically translated to Julia.

Keywords: Modelica, Modia, Julia, Modelica.Media, Modelica.Fluid, ModiaMedia, thermodynamic property models, thermo-fluid models

1 Introduction

Thermodynamic property models (abbreviated as Media models in the rest of this article) require a great deal of flexibility with regards to the choice of thermodynamic and dynamic states to achieve robust and fast simulations. These medium models need functions to describe thermodynamic relationships with different inputs and differential equations to describe dynamic behavior. When such medium models using the Modelica language were first introduced, the only mechanism available that satisfied these requirements was that of a replaceable Modelica package (Elmqvist, et al. 2003). Special constructs for functions were also added to enable media modeling. This use of packages was not part of the initial Modelica language design, however, as they were primarily intended for the organization of model components. As a result, compilers typically handle packages completely at compile time. This fact has several significant implications, such as the restriction from changing the medium or the level of detail of the medium model during simulation.

This paper investigates alternative media and fluid modelling architectures available in the modern programming language Julia (Bezanson, et al. 2017). Mechanisms of interest instead of replaceable packages include member functions, function references, and multiple dispatch[1]. The resulting architecture provides more dynamic flexibility and uses common language constructs so that it is easier to understand and maintain.

The design of the fluid library for Modia is based on a new approach by (Zimmer et al. 2018). This approach is currently used in aircraft industry and enables the robust modeling of fluid streams and avoids the creation of large non-linear equation systems that can be a major source of problems for conventional fluid libraries in Modelica.

2 Thermodynamic Property Models

2.1 Users view

A medium model consists of a data structure that holds the data of the medium and a set of functions operating on this data. The fluid properties are computed from a set of variables called the *thermodynamic states* of the medium. For example, the *thermodynamic states* of the ideal-gas moist air model Modelica.Media.Air.MoistAir are temperature T, pressure p, and the mass fraction of water X, because all other quantities can be computed from them.

A fluid component model, such as a volume model, defines independent variables called *model states* that describe the differential equations of a component model as functions of these states. For example, if a medium is used only in a single phase region, often pressure p and temperature T are used as states of the model, whereas pressure p and specific enthalpy h might be used if the medium enters the two-phase region. Other choices, such as pressure p and density d, may also be necessary to address application-specific requirements (Laughman, Qiao 2016). All media models in Modelica.Media therefore have various possibilities for *model states*, including (p,T), (p,h), (p,s), and (d,T), as well as mass fractions X.

[1]Multiple dispatch in Julia means that method selection is based on the types of *all* non-optional function arguments (if possible at compile-time, otherwise at run-time).

In general, fluid properties are computed by (a) transforming *model states* into medium-specific *thermodynamic states,* and (b) executing medium-specific functions having the medium-specific *thermodynamic states* as input arguments. A programming language that supports member functions can usually implement such a scheme in a reasonable way.

Modelica does not support member functions and therefore the definition of media is awkward and limited. Julia does not have member functions, but instead supports multiple dispatch to select the desired function based on the types of all (non-optional) function arguments (rather than base the selection on a single input argument, as in object-oriented programming languages).

The implementation of this media modeling approach, called ModiaMedia, is available on Github[2]. From a user's point of view, a medium is an object (an instance of a Julia immutable struct) that is returned by function getMedium(..), given the medium name as a string. Example:

```
using ModiaMedia
Medium = getMedium("Air")
```

All media models are stored in a dictionary and getMedium(..) inquires the medium from this dictionary and returns the reference to it. In a second step, the *thermodynamic state* of the medium is computed from the desired independent variables of the application, for example,

```
state = setState_pT(Medium, 1e5, 300)
```

Here the thermodynamic state of the Air medium is computed from p = 10^5 Pa and T = 300 K. Given the thermodynamic state, functions are provided to compute other desired medium properties. For example, the density and specific enthalpy of air can be computed by

```
d = density(Medium, state)
h = specificEnthalpy(Medium, state)
```

An alternative implementation of setState_pT(..) could store a reference to Medium in state. This would then simplify the further access calls, for example: d = density(state). This has not been implemented, as there are open questions about this approach.

In an object-oriented programming language, the syntax would be:

```
d = Medium.density(state)
h = Medium.specificEnthalpy(state)
```

Functions are also provided to plot properties of the medium. In the current version of this project, the most important characteristics of a medium are plotted with the call standardPlot(Medium). This interface is identical for all media. Figure 1 illustrates the (current) standard plot for Air.

In addition to constants and functions, a medium package in Modelica.Media also defines a Modelica *model* called BaseProperties that computes the properties of a medium needed for mass and energy balances. Since ModiaMedia is a standalone package that does not depend on Modia and can be used in other standalone modeling environments, no equivalent Modia model to BaseProperties is defined in ModiaMedia.

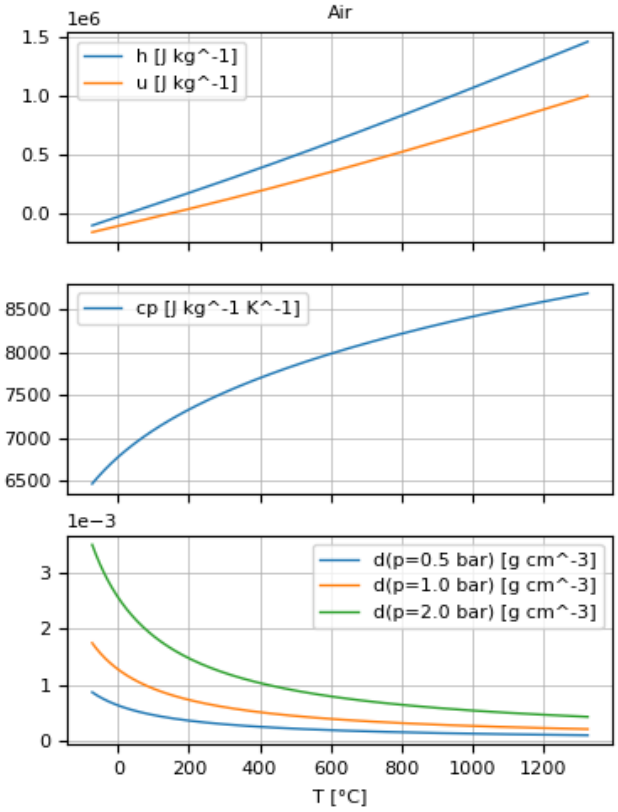

Figure 1. Result of: standardPlot(getMedium("Air"))**.**

2.2 Structure of the ModiaMedia package

The Julia package ModiaMedia has many features in common with Modelica.Media, but is based on a hierarchical type system that allows for greater simplicity and flexibility. The abstract type system of ModiaMedia is a direct mapping of the Modelica.Media class hierarchy:

```
abstract type AbstractMedium end
abstract type PureSubstance <: AbstractMedium end
abstract type MixtureMedium <: AbstractMedium end
```

The above definitions state that PureSubstance and MixtureMedium are subtypes of AbstractMedium. A Medium model is defined as a medium-specific Julia struct that is either a direct or indirect subtype of AbstractMedium and has the following structure:

[2] https://github.com/ModiaSim/ModiaMedia.jl

DOI 10.3384/ecp19157589

```
struct MediumTypeName <: AbstractMedium
  infos::FluidInfos
  fluidConstants::SVector{1,AbstractFluidConstants}
  fluidLimits::FluidLimits
  data::Any         # fluid spec. data
end
```

where

- FluidInfos contains all constants that are similarly defined for a Modelica.Media package (such as mediumName and singleState),
- fluidConstants contains all the data of the fluidConstants vector of records of the equivalent Modelica.Media package,
- fluidLimits defines the validity range of the medium model and
- data contains additional fluid specific data.

Example:

```
struct SingleGasNasa <: PureSubstance
  infos::FluidInfos
  fluidConstants::SVector{1, IdealGasFluidConstants}
  fluidLimits::FluidLimits
  data::SingleGasNasaData

 function SingleGasNasa(...)
   # Constructor function
   ...
 end
end
```

The functions that are available for an AbstractMedium are defined in the following form:

```
density(m::AbstractMedium,
        state::ThermodynamicState) = error("undefined")

specificEnthalpy(m::AbstractMedium,
      state::ThermodynamicState) = error("undefined")
```

where ThermodynamicState is the abstract super type of all thermodynamic states.

A medium model must provide concrete implementations for these functions, e.g.,

```
density(m::SingleGasNasa, state:: State_pT) =
                                  state.p/(m.data.R*state.T)
```

In summary, while ModiaMedia models store analogous data to that which is contained in a Modelica medium package, it is stored in a hierarchical data structure. In comparison, all data is stored in form of constants inside a Modelica package. The benefit of the hierarchical data structure is that this data can be passed as argument to a function allowing a user to easily add functionality for pre- and post-processing. Since a Modelica medium model is actually a package, and Modelica does not support functions that can operate on packages, Modelica medium models can be used for simulation only and it is not possible to easily implement other functions, such as those that plot data of a Modelica medium.

It should be noted that there are several thermodynamic property packages available where medium models are defined with objects and member functions implemented in a programming language such as C++, e.g., FluidProp[3], CoolProp[4], and TILMedia[5]. In comparison with these packages, ModiaMedia is a very early prototype to experiment with a tight integration of a thermodynamic property package with fluid component modeling to achieve fast simulations of transient thermodynamic processes.

2.3 Conversion of Modelica to Modia/Julia

The Modelica Standard Library has a rich set of media models, containing data, functions for thermodynamic properties calculation, table lookup and interpolations, and basic media model equations. Each medium is represented as a Modelica package. To utilize the extensive knowledge and effort encoded in this library, a translator[6] performing source-to-source transformation from Modelica to Modia/Julia has been written in Julia. It has a recursive descent handwritten LL(2) parser. Each grammar production of Modelica (Modelica Association 2017, Appendix B) is represented by a Julia function. Example:

Modelica grammar production:

```
extends-clause :
    extends type-specifier
    [ class-modification ]
    [ annotation ]
```

Julia function:

```
function extends_clause(env)
  expect("extends")
  type_specifier(env)
  if nextItem == "("
    class_modification(env)
  end
  if nextItem == "annotation"
    annotation(env)
  end
end
```

A scanner updates global variables nextItem and nextType. The function expects checks nextItem and if found, scans the next item. The First and Follow sets used in LL parsers have been determined manually and are used to select productions/functions and to end repetition. The variable env is used to transfer which output file is used, indentation level, etc.

[3] http://www.asimptote.nl/software/fluidprop
[4] http://www.coolprop.org/
[5] https://www.tlk-thermo.com/index.php/en/software-products/tilmedia-suite
[6] https://github.com/ModiaSim/ModiaFromModelica

Top level packages and classes of Modelica are translated to Julia modules, while subpackages cannot be converted to modules because cyclic dependencies between Julia modules, such as exist between Modelica subpackages, are not allowed. These subpackages are therefore removed and flattened by introducing long names such as: Modelica_Blocks_Interfaces_SISO for the content.

Models, connectors, blocks are converted to Modia @model macros, while records are converted to mutable structs. Julia supports unit handling also in functions using the package Unitful (Keller, et al 2018).

Since expressions in Modelica are quite similar to those in Julia, there is no need to introduce an abstract syntax tree; as a result, the corresponding Julia text can be generated directly by the parsing function. Examples of slightly different syntax:

```
{1,2,3} → [1,2,3]
[1,2,3; 4,5,6] → [1 2 3; 4 5 6]
```

While regular expression substitution could be considered for expressions, if-then-else expressions pose problems due to the need to introduce end in Julia:

```
if a then b else c ⟶ if a; b else c end
```

(One could have used the syntax a ? b : c instead.)

Variable declarations in models have a different structure in Modelica and Modia:

```
Real x[10] = ones(10) ⟶
                    x = Float(size=(10,), start=ones(10))
```

This means that information needs to be moved from one parsing function to another. This is accomplished by temporarily building small ASTs using tuples.

In comparison, syntax for declarations in functions is quite different, e.g.,:

```
Real x[10] = ones(10) ⟶ x::Array{Float64,1} = ones(10)
```

2.4 Conversion of Modelica.Media to Modia/Julia

Since the Medium definitions in Modelica and in Modia are quite different, it is not yet possible to fully automatically transform a Modelica medium package in an equivalent ModiaMedia model. Instead, the Julia code generated by the translator is currently semi-manually transformed into the desired form with the help of an editor that supports regular expressions.

For example, the SingleGasNasa coefficients that have been transformed from Modelica to Julia are defined as:

```
const Ag = IdealGases.Common.DataRecord(
  name="Ag",
  MM=0.1078682,
  ...)
```

With an editor (defining the changes with regular expressions), all 1200 definitions have been automatically transformed to the assignment of dictionary entries:

```
singleGasesData["Ag"] =
    IdealGases_Common_DataRecord(
      name="Ag",
      MM=0.1078682,
      ...)
```

This dictionary is then serialized and stored so that these medium data can be quickly loaded by the user, rather than be regenerated on every use.

A Modelica medium function of the form:

```
redeclare function extends density
algorithm
  d := state.p/(data.R*state.T);
end density;
```

can be changed to the equivalent ModiaMedia model function:

```
density(m::SingleGasNasa, state::state_pT) =
                                  state.p/(m.data.R*state.T)
```

via the following rules:

1. Add the medium instance m as the first argument to the function.
2. Add the appropriate type information for the input and return variables.
3. Prepend m to all variable data, e.g., "data.Hf" is changed to "m.data.Hf".

We plan to transform the complete Modelica.Media package to ModiaMedia. The base Modelica package has already been transformed to Julia and the somewhat labor-intensive semi-manual final adaptation is currently on the way.

3 Fluid Component Models

In general, our objective is to model and simulate thermo-fluid pipe networks, such as heat exchangers, air conditioning systems, distillation columns, or power plants. Traditional simulation programs in this field tightly couple the equations of the fluid components to the equations of the medium that is flowing in the components. Modelica.Fluid was designed to increase the flexibility of these models by separating the model of the fluid component from the medium model, enabling the use of a pipe model for media that have different thermodynamic states. The Modia fluid prototype continues to pursue the simplification and generalization of the Modelica.Fluid approach.

There are different ways to formulate fluid network models, depending on the application and the properties of the fluid that need to be taken into account. To experiment with simpler and more robust network models, the new method from (Zimmer et al. 2018) is used as basis for the fluid component models

DOI 10.3384/ecp19157589

and has been implemented in a small experimental library named ModiaFluid for unidirectional, 1D thermo-fluid pipe flow that is suited for incompressible media or for compressible media if Ma ≤ 0.3. The Modelica.Fluid library has a similar application area but supports bidirectional fluid flow.

3.1 Users view

A simple pipe network is shown in Figure 2.

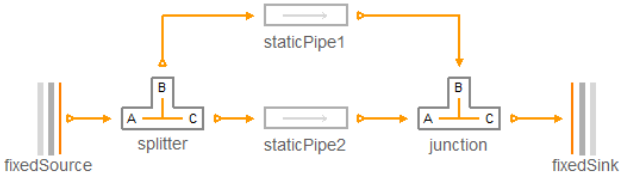

Figure 2. Simple pipe network with splitter and junction**.**

With the ModiaFluid library, this network is defined as:

```
const air = getMedium("Air")
@model PipeSystem begin
  fixedSource = FixedSource(Medium=air,
                            p0=1.0e5, h0=10000,)
  fixedSink   = FixedSink(p0=0.8e5)
  staticPipe1 = StaticPipe(k=3e5, l=1.0, A=0.01)
  staticPipe2 = StaticPipe(l=1.0, A=0.01)
  junction    = Junction()
  splitter    = Splitter()
  @equations begin
    connect(staticPipe2.outPlug, junction.inPlugA)
    connect(staticPipe1.outPlug, junction.inPlugB)
    connect(junction.outPlugC, fixedSink.inPlug)
    connect(splitter.outPlugB, staticPipe1.inPlug)
    connect(splitter.outPlugC, staticPipe2.inPlug)
    connect(fixedSource.outPlug, splitter.inPlugA)
  end
end
```

Currently, Modia has no graphical user interface, and the system must be manually defined with the textual definition above. The definition on this level looks close to a corresponding Modelica model. The essential difference is that the medium model is defined only at one component (because the medium is propagated through the connection structure), whereas in Modelica it must be defined for every component.

3.2 Fluid connectors

The ModiaFluid library currently supports only unidirectional fluid flow. This assumption is already built into the connectors that are defined corresponding to (Zimmer et al. 2018):

```
MediumVariable() = Variable(size=())

@model InPlug(:connector) begin
  Medium = MediumVariable()
  m_flow = Float(flow = true)
  r  = Float()
  p = Float(input=true)
  h = Float(input=true)
end

@model OutPlug(:connector) begin
  Medium = MediumVariable()
  m_flow = Float(flow = true)
  r  = Float()
  p = Float(output=true)
  h = Float(output=true)
end
```

The variables have the following meaning:

- Medium is a reference to the medium data structure of section 2 and defines the medium that is flowing through the connector. This reference is propagated through the connection structure by means of alias elimination and is treated as one Modia variable in the symbolic engine. Note, a Modia variable can be any Julia data structure.
- m_flow is the mass flow rate into the connector,
- r is the pressure that is used to accelerate the fluid (see section 3.4),
- p = staticPressure - r, and
- h is the specific enthalpy.

A connector is modelled as a Modia @model macro with the Symbol :connector as macro parameter. Note that p and h are declared as either input or output.

Formally, an InPlug connector can only be connected with an OutPlug connector and not with another InPlug connector, so there are restrictions how components can be connected together.

3.3 Medium propagation

In the connectors of section 3.2, Medium is a Modia variable, where the type of the variable is not yet defined but will be deduced by type inference. At one or at several components, this variable is redefined to an instance that is a subtype of AbstractMedium. Example:

```
@model FixedSource begin
  Medium = MediumVariable()
  outPlug  = OutPlug()
  state    = Variable()
  ...
@equations begin
  outPlug.Medium = Medium
  state = setState_ph(Medium, outPlug.p, outPlug.h)
  d     = density(Medium, state)
 ...
end

const air = getMedium("SimpleAir")

@model PipeSystem begin
```

```
  source=FixedSource(Medium=air)
  ...
end
```

In model FixedSource, the Medium variable must be redefined when the component is used. This is performed by first generating a medium model with

```
air = getMedium("SimpleAir")
```

and then using air as modifier to the FixedSource instance:

```
source=FixedSource(Medium=air)
```

In the FixedSource component, an equation outPlug.Medium = Medium is present. Furthermore, the Medium variable might be used to compute medium properties such as the density d.

Modia treats Julia structs (such as variable Medium) specially: Struct variables can be used as modifier or as variable in an equation "var1 = var2". When propagating a reference in this way, an overdetermined system of equations can occur (when connections form a loop, or when the same medium is defined at several components). This issue is automatically resolved by extended alias elimination.

Since the media in Modelica.Media are packages and Modelica cannot use packages in equations and also does not have special language elements to propagate packages in connections, a medium has to be defined in every component where it is used. Therefore, if a pipe system has 20 components, then the medium needs to be defined 20 times.

In ModiaFluid, a medium can be defined in one component model and is then propagated through all components and connections where the fluid is flowing. The current implementation of medium propagation has however the temporary limitation that a medium must also be defined at all volumes, for details see section 3.5. In principal, this mechanism would allow changes to the medium during simulation as long as the same BaseProperties models (see section 3.5) are used.

3.4 Momentum balance

Modelica.Fluid utilizes the steady-state *or* the dynamic momentum balance depending on the chosen option. In ModiaFluid the approach from (Zimmer et al. 2018) is used to achieve more efficient and more robust simulations. Hereby, the unsteady Bernoulli equation is the starting point (Zimmer et al. 2018, Schade et al. 2013 eq. 4.4-3, Brennen 2006 section Bnda[7])[8].

[7] http://brennen.caltech.edu/FLUIDBOOK/basicfluiddynamics/Unsteadyonedimensionalflow/Unsteadybernoulli/unsteadybernoulli.pdf

[8] The unsteady Bernoulli equation is derived by integrating the Euler equations for incompressible fluid flow along a stream line. The Euler equations in turn are the differential form of the momentum balance that is used in Modelica.Fluid.

The approach is sketched with the simple example shown in Figure 3, where three pressure drop components (for example pipes, orifices, bends) are connected between two volumes and fluid flows from volume1 to volume2.

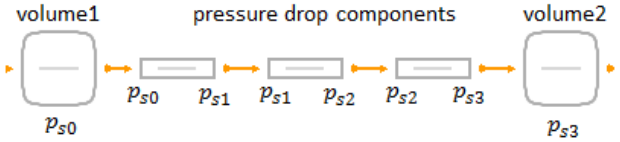

Figure 3. Three pressure drop components connected between two volumes ($\boldsymbol{p_{si}}$ is the static pressure at the indicated location)**.**

For simplicity of the derivation the specific kinetic energy is neglected. Assume that all pressure drop components have the same area A and component i has length Δs_i, that m_{flow} is the mass flow rate, p_{si} is the static pressure at the indicated location and $\Delta p_{i-1,i}$ is the pressure drop correlation of component i. The unsteady Bernoulli equation can then be formulated as:

$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_1}{A} + p_{s1} - p_{s0} = \Delta p_{01}(m_{flow}, p_{s0}, h_0)$$
$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_2}{A} + p_{s2} - p_{s1} = \Delta p_{12}(m_{flow}, p_{s1}, h_1)$$
$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_3}{A} + p_{s3} - p_{s2} = \Delta p_{23}(m_{flow}, p_{s2}, h_2)$$

The specific enthalpies h_i are separately computed, e.g., for isenthalpic pressure drop components the h_i are the upstream specific enthalpies ($h_2 := h_1 := h_0$).

Note, in the Modelica.Fluid library the momentum balance is used in the form[9]

$$\frac{dI_s}{dt} + (p_{s1} - p_{s0})\cdot A = \Delta p_{01}\cdot A$$
$$I_s = m_{flow}\cdot\Delta s_1$$

As can be seen, this is exactly the unsteady Bernoulli equation multiplied with A.[10] So, the starting point of the derivation below are exactly the same equations as used in the Modelica.Fluid library.

The static pressures are now split into two parts:

$$p_{si} := p_i + r_i$$

where r_i is the pressure that is used to accelerate the fluid and p_i is the remaining part of the pressure. In steady state operation, $r_i := 0$, $p_{si} := p_i$. Introducing

[9] Modelica.Fluid.Interfaces.PartialDistributedFlow

[10] The unsteady Bernoulli equation has, however, the advantage that in its general form it holds along a streamline, so also for bends and orifices. The momentum balance along a streamline includes the (unknown) reaction forces on the component and therefore it can only be used in equations for a straight pipe, where the reaction forces in direction of the pipe flow are zero.

DOI 10.3384/ecp19157589

these terms in the unsteady Bernoulli equations and utilizing the abbreviation $\Delta r_{i-1,i} = r_i - r_{i-1}$ results in:

$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_1}{A} + \Delta r_{01} + p_1 - p_0 = \Delta p_{01}$$
$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_2}{A} + \Delta r_{12} + p_2 - p_1 = \Delta p_{12}$$
$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_3}{A} + \Delta r_{23} + p_3 - p_2 = \Delta p_{23}$$

Since the r_i are defined to solely accelerate the fluid, the equations can be split into two parts:

$$p_1 - p_0 = \Delta p_{01}(m_{flow}, p_{s0}, h_0)$$
$$p_2 - p_1 = \Delta p_{12}(m_{flow}, p_{s1}, h_1)$$
$$p_3 - p_2 = \Delta p_{23}(m_{flow}, p_{s2}, h_2)$$
$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_1}{A} = -\Delta r_{01}$$
$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_2}{A} = -\Delta r_{12}$$
$$\frac{dm_{flow}}{dt}\cdot\frac{\Delta s_3}{A} = -\Delta r_{23}$$

Furthermore, we have the boundary conditions at the volumes:

$$p_{s0} = p_0 \quad (r_0 = 0)$$
$$p_{s3} = p_3 + r_3$$

No approximations have been introduced so far (the original equations have been just reformulated). Now, the *approximation* is made, that the dependency of the pressure drop equations on the *inertial pressure r is neglected*:

$$\Delta p_{i-1,i} = \Delta p_{i-1,i}(m_{flow}, \boxed{\boldsymbol{p_{i-1}}}, h_{i-1})$$

Note that the pressure drop equations are typically determined only for steady-state operations, and that the relationships/equations that take the acceleration of a fluid into account are often not known. In particular, all the pressure drop correlations used in Modelica.Fluid hold only for steady-state operations.

The big advantage of this slight approximation is that the equations are now decoupled, as described in the following. First, the pressures p_i can be computed in a forward sequence because the static pressures and the specific enthalpies at the volumes are known, as well as the mass flow rate m_{flow} (since its derivative appears in the unsteady Bernoulli equation, m_{flow} is a state):

$$p_1 := p_0 + \Delta p_{01}(m_{flow}, p_0, h_0)$$
$$p_2 := p_1 + \Delta p_{12}(m_{flow}, p_1, h_1)$$
$$p_3 := p_2 + \Delta p_{23}(m_{flow}, p_2, h_2)$$

The remaining equations then form a *linear* system of equations and the coefficients of the system matrix are constants:

$$\begin{bmatrix} \frac{\Delta s_1}{A} & 1 & 0 & 0 \\ \frac{\Delta s_2}{A} & 1 & -1 & 0 \\ \frac{\Delta s_3}{A} & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{dm_{flow}}{dt} \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} p_{s0} \\ 0 \\ 0 \\ p_{s3} - p_3 \end{bmatrix}$$

The result is that the interconnection of pressure drop components results only in a *small linear equation system* with a constant coefficient matrix. Since this matrix is constant, it is sufficient to perform an LU decomposition once and then reuse it during the simulation. If this approximation is not performed as in the Modelica.Fluid library, *nonlinear algebraic equation systems* often appear that can be either hard to solve (especially, when starting at $m_{flow} = 0$) or in which these equation systems have no unique solutions.

For these reasons, there is a "rule of thumb" in the Pipe-network community to always have a volume between two pressure drop components. In Modelica.Fluid there is, for example, the option modelStructure in the DynamicPipe model to define whether the pipe ends with a volume on either of the ports or not, in order to avoid such nonlinear equation systems. Obviously, such options are only for fluid specialists. When using the new method, such advanced options are no longer needed.

In (Zimmer et al. 2018) more complicated cases with branching and joining piping networks are additionally discussed. Here, the dynamic momentum balance is also taken into account, whereas in Modelica.Fluid only junction models with steady-state momentum balances are provided.

In the ModiaFluid library the new method is used in all components, including junctions. For example, a pipe model is defined in the following way, directly utilizing the equations explained above:

```
@model ShortPipe begin
  inPlug  = InPlug()
  outPlug = OutPlug()
  ...
@equations begin
  # Medium propagation
  outPlug.Medium = inPlug.Medium

  # mass flow balance
  m_flow = inPlug.m_flow
  inPlug.m_flow + outPlug.m_flow = 0

  # Propagation of specific quantities
  outPlug.p = inPlug.p + dp
  outPlug.r = inPlug.r + dr
  outPlug.h = inPlug.h + dh
  dp = -m_flow*abs(m_flow)*k  # pressure drop
  dr = -der(m_flow)/A*l       # inertial pressure
```

```
    dh = 0.0                    # isenthalpic process
    end
end
```

In the Modelica.Fluid package there are many options that can be set either at the component level or globally. For example, the user can choose either a steady-state or dynamic momentum balance, the presence of a volume on either port of a pipe, or the definition of pressure drop components as functions of the pressure difference or as functions of the mass flow rate.

In ModiaFluid the complexity of the code and of the options is drastically reduced by providing only the dynamic momentum balance, by describing pressure drop components as function of mass flow rate, and by having only one discretization scheme for a pipe. The simulation is also potentially more robust than when defined with Modelica.Fluid, because no nonlinear algebraic equations occur, even if pressure drop components are connected together without a volume in between.

3.5 Mass and energy balance

The definition of mass and energy balances are essentially analogous to the approach used in Modelica.Fluid. With the specific internal energy u, density d, volume V, mass m, internal energy U, the sum of the mass flow rates into the volume m_{flow}, the sum of the enthalpy flow rates in to the volume H_{flow}, the contribution due to the unsteady movement of the fluid H_r, and the pressure p and specific enthalpy h as the independent variables of the utilized medium model, the balance equations can be formulated as:

$$
\begin{aligned}
d &= d_{medium}(p,h)\\
u &= u_{medium}(p,h)\\
m &= d \cdot V\\
U &= m \cdot u\\
\frac{dm}{dt} &= m_{flow}\\
\frac{dU}{dt} &= H_{flow} + H_r\\
H_r &= \sum_{i \in inflowing} m_{flow,i}\, r_i/d
\end{aligned}
$$

Since each inlet i of a volume forms a boundary for the pressure, a pressure difference may occur between the volume and the inlet pressure p. This difference is accounted by $r_i = p_s - p_i$. Since the volume work $V_{flow,i} r_i = m_{flow,i}\, r_i/d$ of this pressure gradient is accounting for the acceleration (or deceleration) of the inflowing fluid, the enthalpy of the inflowing fluids needs to be corrected by the term H_r. This is not necessary for the outlets since for the outlets, the term r is zero by definition.

If the @model equations would be defined in this way, then m and U would be selected as states and the independent medium variables, for example (p,h) or (p,T) would be in general determined by solving nonlinear equation systems. The approach is to rewrite the equations. In Modelica.Fluid this is performed by providing the attribute StateSelect.prefer on the desired states. The goal in Modia is to arrive at a simpler language as Modelica and therefore an attribute StateSelect is not supported. Instead, the derivatives are manually expanded until the state derivatives appear. For example, (p,h) shall be used as states. The mass- and energy balance can then be reformulated to (= manual index reduction):

$$
\begin{aligned}
d &= d_{medium}(p,h)\\
u &= u_{medium}(p,h)\\
\text{der\_d} &= \frac{\partial d_{medium}}{\partial p}\dot{p} + \frac{\partial d_{medium}}{\partial h}\dot{h}\\
\text{der\_u} &= \frac{\partial u_{medium}}{\partial p}\dot{p} + \frac{\partial u_{medium}}{\partial h}\dot{h}\\
m &= d \cdot V\\
U &= m \cdot u\\
\text{der\_d} \cdot V + d \cdot \dot{V} &= m_{flow}\\
m_{flow} \cdot u + m \cdot \text{der\_u} &= H_{flow} + H_r
\end{aligned}
$$

By this reformulation only derivatives for the independent medium variables (and of the volume V) appear and therefore only these variables can be potential states. Note, the equations above are linear in the derivatives and no nonlinear equation system appears anymore.

The first four equations are marked in blue to indicate that these equations are medium specific. *Depending on the medium type*, these four equations need to be provided. This is accomplished by providing a specific BaseProperties Modia model. Example for the SimpleIdealGas medium that has (p,T) as states, where d=d(p,T) and u=u(T):

```
@model SimpleIdealGas_BaseProperties begin
  p_start = 1e5
  T_start = 300.0
  Medium = MediumVariable()
  p = Float(start=p_start)
  h = Float()
  T = Float(start=T_start)
  d = Float()
  u = Float()
  der_d = Float()
  der_u = Float()
  state = MediumState()
@equations begin
  d = d_pT(Medium,p,T)
  u = u_T(Medium,T)
  h = h_T(Medium,T)
  der_d = d_pT_der_2(Medium,p,T)*der(p) +
          d_pT_der_3(Medium,p,T)*der(T)
  der_u = u_T_der_2(Medium,T)*der(T)
  state = setState_pT(Medium,p,T)
  end
```


DOI 10.3384/ecp19157589

```
end

BaseProperties(Medium:: SimpleIdealGasMedium;
    p_start=1e5, T_start=300.0) =
    SimpleIdealGas_BaseProperties(Medium=Medium,
                        p_start=p_start, T_start=T_start)
```

Under the assumption that p,T are states and that all used functions are available in ModiaMedia, all the left hand side variables can be computed from p,T and their time derivatives. In the @equations section the Modia convention is used that fc_der_i(..) is the partial derivative of function fc with respect to its i-th argument. Note, the medium specific functions must reflect the true dependency of the function from the independent variables in order that the symbolic transformation does not introduce singularities in the generated code. For example, although p,T are the thermodynamic states of this medium, the inner energy u is only a function of T and not of p,T.

Above, for every medium type a medium specific function BaseProperties(Medium; ...) is defined that selects the *medium specific* BaseProperties @model, instantiates it and returns this instance. Alternatively, all medium-specific BaseProperties @models could be stored in a dictionary, and function BaseProperties could just return an instance of the corresponding @model using the Medium type as a dictionary key. Note, media types that have the same functional dependency on d,u,h, can use the same BaseProperties model. With these pre-requisites a general volume model can be defined as:

```
@model ClosedVolume begin
  Medium = MediumVariable()
  inPlug   = InPlug()
  outPlug  = OutPlug()
  p0 = Medium.infos.p_default
  T0 = Medium.infos.T_default
  medium = BaseProperties(Medium; p_start=p0,
                                  T_start=T0)
  ...
@equations begin
  outPlug.Medium = Medium
  outPlug.Medium = inPlug.Medium
  m = medium.d*V
  U = m*medium.u
  m_flow = inPlug.m_flow + outPlug.m_flow
  H_flow = inPlug.m_flow*inPlug.h +
           outPlug.m_flow*outPlug.h
  H_r = inPlug.m_flow*inPlug.r / medium.d

  # Mass and energy balance
  medium.der_d*V + medium.d*der(V)   = m_flow
  m_flow*medium.u + m*medium.der_u = H_flow+H_r

  # Propagation of specific quantities
  inPlug.p + inPlug.r = medium.p
  outPlug.p = medium.p
  outPlug.r = 0.0
  outPlug.h = medium.h
  end
end
```

A key part of this @model is the declaration of the BaseProperties @model:

```
medium = BaseProperties(Medium; p_start=p0,
                                T_start=T0)
```

This declaration provides an instance of a medium-specific BaseProperties @model depending on the type of variable Medium. The problem here is that Medium should be propagated through connections and the instantiation of BaseProperties can only be performed when the type of the propagated Medium is known (so instantiation and extended alias elimination must be performed incrementally). Modia does not yet support such a scheme and therefore the current implementation of ModiaFluid requires to define the Medium at volumes to select the BaseProperties @model based on the Medium type.

3.6 Further Issues

ModiaFluid should optionally support bi-directional fluid flow in the future. Additionally, there are other issues:

Caching for media calculations

More complicated media, such as two-phase media or mixture media, may require the solution of nonlinear equation systems whenever medium variables, such as specific internal energy, have to be computed. In Modelica.Fluid typically the nonlinear solver either starts always from the same start values of the iteration variables, or with some very simplified models first start values for the iteration variables are computed. The current version of ModiaFluid only supports the same approach.

In principal it would be possible to make such medium calculations more efficient and more robust by caching the medium states from the previous model evaluation and use them as start values at the next time instant:

```
setState_pT!(state, Medium, p, T)
```

Julia allows to update input arguments and therefore to keep a memory between function calls. The setState_xxx(..) functions would thus be slightly rewritten to update the current state and hereby utilize a cache in the state. In order that Modia knows which variable is computed by such a call (for size inference and equation sorting), the argument that is updated by the call must be "somehow" marked.

Nonlinear equations at junctions

As described in detail in (Franke et al. 2009, section 4.2), junctions may give rise to nonlinear algebraic equation systems where the iteration variables are discontinuous when the mass flow rate changes sign and therefore the solution is hard. In ModiaFluid this cannot occur, because only unidirectional flow is supported where the upstream direction does not change during simulation.

Unnecessary nonlinear equations at 1:1 connections

As described in detail in (Franke et al. 2009, section 4.3), unnecessary nonlinear equation systems occur at every 1:1 connection of fluid components if the *thermodynamic states* are not (p,h) and h is a *nonlinear* function of the *thermodynamic states*. This effect currently appears in ModiaFluid. In Modelica.Fluid this issue is resolved by an inverse function annotation and an involved symbolic manipulation of the equations. Current efforts in ModiaFluid include the pursuit of a simpler solution to this problem.

4 Automatic Differentiation of Media Functions

Partial derivatives of functions are needed since relationships between thermodynamic variables are modelled using functions and these relations needs to be differentiated due to index reduction in the mass and energy balance or for obtaining the Jacobian for iterative solvers. Modelica.Media has many manually provided derivatives of functions. The described approach in ModiaMedia allows automatic differentiation of functions to be easily utilized.

There are several Julia packages for automatic differentiation, see http://www.juliadiff.org/. The partial derivative of a function specificEnthalpy_water(T) is obtained as follows by using the ForwardDiff package:

```
specificEnthalpy_water_der_1(T) =
  ForwardDiff.derivative(specificEnthalpy_water, T)
```

5 Conclusion

Despite past successes of thermo-fluid modeling using the Modelica language, there have been long-standing discussions on how to improve thermodynamic property modeling for dynamic systems by making it more convenient, easier to comprehend, and more powerful. Many of the modern programming constructs of the Julia language, such as multiple dispatch, lend themselves to new approaches to address these existing challenges. The ModiaMedia and ModiaFluid architecture described in this paper represents one experimental effort to leverage these recent developments: Thermodynamic property modeling becomes an order of magnitude simpler, both for implementation and for usage. Furthermore, it becomes then possible to propagate medium models through connection structures and use them in pre- and post-processing. By adopting the new fluid approach from (Zimmer et al. 2018), recent progress within the Modelica community can be directly transferred to Modia. We expect to continue developing and enlarging this thermo-fluid modeling framework to further explore the opportunities afforded by these new computing paradigms and tools.

Acknowledgements

The authors want to thank Jarrett Revels, MIT for valuable advice on Julia and automatic differentiation using Julia.

References

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. (2017): Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59: 65–98. doi: 10.1137/141000671

Brennen, C.E. (2006): An Internet Book on Fluid Dynamics. http://brennen.caltech.edu/FLUIDBOOK/FLUIDBOOK.htm

Elmqvist, H., Tummescheit, H. and Otter, M. (2003): Object-Oriented Modeling of Thermo-Fluid Systems, *Proceedings of the 3rd International Modelica Conference*, Linköping, Sweden, November 3-4, pp. 269-286. https://www.modelica.org/events/Conference2003/papers/h40_Elmqvist_fluid.pdf

Franke R., Casella F., Otter M., Sielemann M., Elmqvist H., Mattsson S.E., Olsson H. (2009): Stream Connectors – An Extension of Modelica for Device-Oriented Modeling of Convective Transport Phenomena. *Proceedings of the 7th International Modelica Conference*, Como, Italy, Sept. 20-22, pp. 108-121. http://www.ep.liu.se/ecp/043/012/ecp09430078.pdf

Keller, A., et al: https://github.com/ajkeller34/Unitful.jl, downloaded 19 Nov 2018.

Laughman, C.R., Qiao, H. (2016): On the Influence of State Selection on Mass Conservation in Dynamic Vapor Compression Cycle Models. *Mathematical and Computer Modeling of Dynamical Systems*, Vol. 23, No. 3, pp. 262-283, December 2016, DOI: 10.1080/13873954.2017.1298625

Modelica Association (2017): *Modelica, A Unified Object-Oriented Language for Systems Modeling. Language Specification, Version 3.4,* April 10, 2017. https://www.modelica.org/documents/ModelicaSpec34.pdf

Schade H., Kunz E., Kameier F. and Paschereit C.O. (2013) *Strömungslehre*. 4. Auflage, de Gruyter.

Zimmer D., Bender B., Pollok A. (2018): Robust Modeling of Directed Thermofluid Flows in Complex Networks. *Proceedings of the 2nd Japanese Modelica Conference*, pp. 39-48, Tokyo, May 17-19. https://elib.dlr.de/120701/