Design and Implementation of Object-Oriented Model Libraries using Modelica

Tummescheit, Hubertus

2002

Document Version:
Publisher's PDF, also known as Version of record

Design and Implementation of Object-Oriented Model Libraries using Modelica

Hubertus Tummescheit

Automatic Control

Design and Implementation of Object-Oriented Model Libraries using Modelica

Design and Implementation of Object-Oriented Model Libraries using Modelica

Hubertus Tummescheit

Department of Automatic Control
Lund Institute of Technology
Lund, August 2002

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 LUND
Sweden

ISSN 0280–5316
ISRN LUTFD2/TFRT–1063–SE

Printed in Sweden by Bloms i Lund Tryckeri AB.
Lund 2002

Contents

Preface

The subject of this thesis belongs somewhere into the intersection of the disciplines of Automatic Control, Computer Science and Systems Design Engineering. The thesis explores the design of object-oriented model libraries using the example of a library for thermo-fluid systems. The library is written in the new, object-oriented modeling language Modelica. During the time of the library development I was involved in the design of the Modelica language from an early stage. It was a great pleasure for me to participate in the design of the language and to pioneer its use. The enthusiasm of the members of the Modelica Design group for modeling and simulation problems has been contagious.

The task of writing an interdisciplinary thesis is challenging in terms of finding the right level of detail for readers with widely differing background knowledge. Some readers may wish to skip parts of the thesis that are either not interesting or well known to them. Readers with a special interest in thermo-fluid models should read Chapter 4, people which are curious about the ThermoFluid library should read Chapter 5 and modelers looking for ideas about object-oriented library design in general may find Chapter 6 interesting.

Acknowledgments

Writing a thesis is a project that lives from the exchange of ideas and lively discussions. Many people have helped me in producing this thesis, through discussions and close cooperation in several projects and by being joyful colleagues, good friends and keeping my spirits up.

First of all I have to thank the coordinated efforts of three people who made me come to Lund and stay here: Sven Erik Mattsson who invited me and got me started, Jonas Eborn for making the team work such a nice experience and Karl Johan Åström who convinced me to stay and do my PhD in Lund. Thanks for that, I have a great time here. The collaboration with Jonas has been a source of inspiration during the whole project.

I am glad to express my sincere gratitude to Karl Johan Åström. His constructive criticism on several chapters of the manuscript helped to improve its readability. He has created the necessary conditions for a very stimulating research atmosphere and is a consistent source of enthusiasm. It has been a great pleasure and privilege to learn from you. Many thanks to my other supervisor Anders Rantzer. He has the gift of always asking the most relevant and interesting questions. His incessant demand for new chapters of the thesis and valuable comments on the manuscript helped a lot getting this work done. Many thanks also to other senior

staff at the department to acquire the funding for many graduate students. Many thanks to all who proofread parts of the manuscript and gave feedback: Sven Erik Mattsson, Jakob Munch Jensen, Jonas Eborn, Godela Rossner, Karl Erik Arzén and Johan Åkesson. Special thanks to my mother for correcting my English under time pressure.

I also wish to thank Falko Jens Wagner for the good time we had working together with the ThermoFluid library and Jakob Munch Jensen for many stimulating discussions, good laughs, interesting joint work and cultural expeditions to Copenhagen's Theaters and Jazz Clubs.

Special thanks go to the people at Dynasim AB who provided us with new versions of Dymola as soon as we tested the latest additions to the Modelica language. Particular thanks to Hans Olsson for his enthusiastic greetings when receiving a bug report or feature request on the phone. Special thanks also to all members of the Modelica Association. The discussions about modeling and language design in a group with such a broad background has been a valuable source of insight and ideas. I appreciated the combination of intensive technical discussions and a friendly, relaxed atmosphere around them.

I wish to thank the Masters students that I supervised, Olaf Bauer, Antonio Gómez Pérez and Staffan Haugwitz, for many good ideas and contributions. Olaf has done valuable base work for fluid property functions with his Maple-package for thermodynamic derivatives.

I am grateful to my colleagues at the Department who create a friendly and inspiring atmosphere. I specially would like to thank the following: Eva Schildt, Britt-Marie Mårtensson and Agneta Tuszynski for keeping our spirits up and running the unnoticed background work, Leif Andersson and Anders Blomdell for keeping the computers in good health, Sven Hedlund for his enthusiastic advertisements for running in Skrylle, Magnus Gäfvert for hints about interesting music and Andrey Ghulchak for organizing the "inspirationsfika".

Many thanks to the Sunday dinner gang for many nice evenings, lots of ice-cream, never rejecting a Malt and for almost always doing the dishes before leaving: Andrey, Shi-Lin, Ari, Beatrice, Stephane, Jenny, Stefan, Maru and the many guests who came during their visits in Lund.

This project has jointly been supported by Sydkraft AB under the project name "Modelling and Control for Energy Systems" and by the National Board for Industrial and Technical Development (NUTEK) programme "Complex Systems", Dnr 96-10653. Their financial support is gratefully acknowledged.

1

Introduction

Abstract

This chapter provides background about modeling of physical systems in order to explain the need for better tools and languages for modeling. It is motivated that an important part of engineering know-how is encoded in system models. This knowledge needs to be stored in a formal and reusable way.

1.1 Why Modeling is Important

Mathematical models are compact representations of knowledge. Probably the most important but intangible advantage of modeling is the insight and increased understanding that the process of modeling gives about a system. Knowledge is much easier to communicate in the form of a mathematical model than with a textual description. Engineering knowledge and education is to a large extent based on models in different domains. When this knowledge is made available to the engineering design process, it helps a great deal to increase safety, quality and economy of that system. Modeling is a major part in any engineering development. A model that is executable in a simulation program is much easier and safer to work with than the real system. This has been summarized very well by an executive at one of the largest companies in the process industry:

> Modeling and simulation technologies are keys to achieve *manufacturing excellence* and to *assess risk* in unit operations. As we make our plant more flexible to respond to *business opportunities*, efficient *modeling and simulation* techniques will become *commonly used tools*.
>
> Ralph P. Schlenker, Exxon Chemical

In todays engineering practice, model based analysis, simulation and design are major pillars in the development of advanced technical prod-

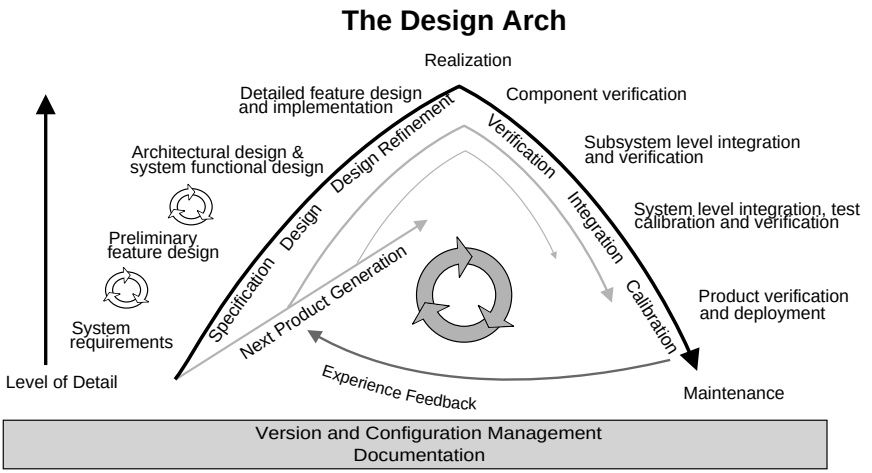


Figure 1.1 Design arch of product development and life cycle. A similar scheme is sometimes referred to as design-V.

ucts. Cost savings achieved by avoiding possibly destructive "smoke tests" of expensive hardware and by doing this test as a model based computer simulation instead, are a strong driving force for model development. Development cycles for new technical products are shortened by making developments in parallel instead of sequential. Emulating a not-yet-existing piece of hardware on a computer, often in a hardware-in-the-loop (HIL) configuration, is a standard technique for achieving concurrent engineering. HIL needs models which are not only accurate representations of reality, but also fulfill stringent performance criteria. The simulation must be executed in real time, otherwise it is not possible to emulate reality to a degree that allows meaningful tests, e. g., of control equipment.

Figure 1.1 illustrates the typical phases of development of a technical product. Almost all phases can to some extent benefit from modeling and simulation. The models which are needed in the phases often have different requirements: the change in the level of detail leads to different models. Modeling language features that help reuse in concurrent engineering and simplify model reduction are important. It is even useful to be able to keep the structure but exchange the underlying model completely. On the right hand side of the design arch, hardware-in-the-loop is a well known means to reduce testing cost. The performance requirements are often difficult to achieve. Reuse of models, both throughout the design process and for the next-generation product, is an important factor to reduce modeling and simulation costs.

1.2 Outline and Contributions

This thesis discusses the development of an object-oriented model library for thermo-fluid systems with focus on the structuring and reuse of models. The development was done in parallel to the development of the underlying modeling language, Modelica™ [Modelica Association, 2002a] which is specifically designed to facilitate model reuse. This parallel development closed the feedback loop between model development and modeling language development in a very fruitful way. New concepts in the language were implemented in the library, experience from the use of the new concepts was used to refine the language definition and make it more powerful and easier to use in the next iteration of the language. This interplay of serious model development, structuring of models for reuse and language design by experts in many engineering domains has helped to shape Modelica[1] into its current form. The process is not finished: mathematical modeling of systems is and will remain to be a challenging activity. The main contributions in the thesis are the following:

- **Model library design.** The desire to develop object-oriented, reusable physical models has been a driving force of this work which started before the idea of Modelica was born. The first library was written in the SMILE language, [Mühlthaler, 2000], jointly developed by GMD FIRST and the Technical University of Berlin. The Smile language was oriented more specifically towards the simulation of power plants and was successfully used in a fluidized bed combined heat and power plant [Buse, 2001] and a combined solar thermal power plant [Tummescheit and Pitz-Paal, 1997]. The scope of the library was broadened to general thermo-fluid systems when it was redesigned in Modelica. Experiences were combined with those gained in the development of the **K2** library developed at Lund University, [Eborn and Nilsson, 1996; Eborn, 1998]. Earlier stages of this work were presented in [Tummescheit and Eborn, 1998; Eborn *et al.*, 1999; Tummescheit *et al.*, 2000; Tummescheit and Eborn, 2002; Tummescheit, 2000a; Tummescheit, 2000b].
- **Modelica language design.** The design of the Modelica language was a joint effort with contributions from many experts in several engineering domains, computer science and numerical mathematics. It was a collaborative development that I had the pleasure to participate in. An important aspect of the Modelica evolution was the tight feedback loop between model language design and use of Modelica in real world problems. My special interests here were high

[1] The ™-sign is omitted from now on to improve readability.

level parameters (also called class parameters) and efforts to make sure that external functions written in C or FORTRAN are easy to integrate. The result of this work is published in Modelica specification [Modelica Association, 2002b]. An early design stage of class parameters in Modelica is presented in [Tummescheit *et al.*, 1997].

- **Models for thermo-fluid systems.** Modeling expertise and the challenge of relevant industrial problems are a necessary background to test a model library for its usefulness. Thermo-fluid systems is my area of experience. Thermo-fluid systems have in the past been a domain where no general purpose modeling tools or languages have been available[2]. Two phase flow models like the moving boundary models presented in Chapter 4 have been of special interest. Publications on two phase flow models are [Bauer and Tummescheit, 2000; Jensen and Tummescheit, 2002].

- **Industrial applications.** An important aspect of the work was the participation in industrial modeling projects, applying the ThermoFluid library to a diverse range of real world modeling problems. Relevant projects were the modeling of combustion for automotive systems at Ford Motor Company [Tummescheit and Tiller, 2000; Tiller *et al.*, 2000], modeling of fuel cell systems at United Technologies Research Lab, modeling of a steam distribution network in a paper plant [Lindstrand, 2002] and modeling of CO_2 -based refrigeration cycles. These industrial projects have given useful input to the library and the Modelica language.

The thesis is organized in the following way. This chapter gives an introduction to the background and fundamental aspects of modeling and simulation of systems. Chapter two presents some modeling techniques. The development of the Modelica language and a description of the key features of Modelica that are a necessary prerequisite to understand the library design discussion follow in chapter three. Chapter four presents an overview over thermo-fluid models used in the implementation of the ThermoFluid library, described in the next chapter. Chapter six summarizes the experiences from object-oriented library design. Recommendations for future work are proposed in chapter seven and conclusions are drawn in chapter eight.

[2]There are many simulation tools for thermo-fluid systems, but all with black box models without possibilities to create new models

1.3 Purpose of Modeling

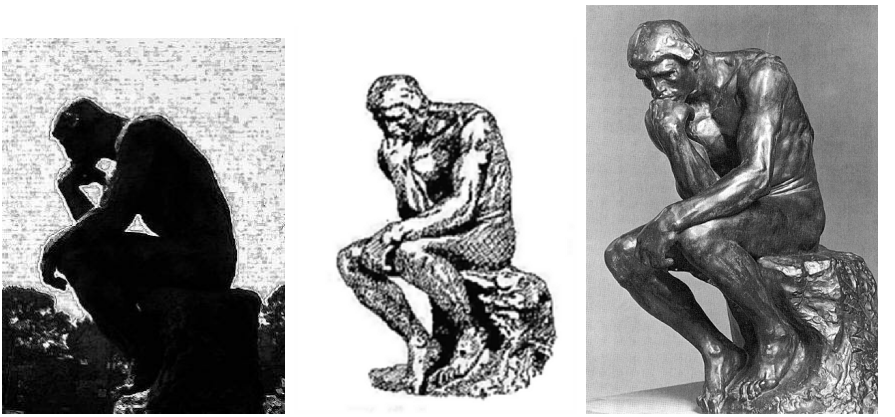
Modeling is a rich activity with a broad scope. The focus in this thesis is on modeling of complex technical systems. Mathematical models of systems are never done as an intellectual exercise to find the best possible mathematical representation of reality. Reality is complex, models do not and should not seek to obtain the same complexity. Models in this thesis are always done with a *purpose*, they are developed to answer specific questions about the system's behavior and often they are restricted to certain boundary conditions or inputs to the model. As Marvin Minsky, [Minsky, 1965] put it:

> A model (M) for a system (S) and an experiment (E) is anything to which E can be applied to answer questions about the system S.

Asking two different questions about the same systems often results in two different, possibly even entirely unrelated mathematical models which are best suited to answer the particular questions. It is important to realize that there is no such thing as a perfect model for a system. This is a widespread belief, based on the idea that with growing sophistication, the model eventually converges to the system. In the best case, the similarity between the behavior of the model and the modeled system increases until no difference between the two behaviors can be observed within the limits of experimental results.

A simple illustration of a set of models with increasing sophistication are physical pictures of an object with increasing level of detail: sketch, drawing, black and white photography, color photography, hologram and sculpture [Preisig, 2001], see Figure 1.2. It is interesting to note in this context that a simpler representation of reality may be more efficient in communicating the characteristic features of the system. A drawing or black and white photography may be better suited to reproduce the three dimensional features of an object than a color photography which undeniably has a larger amount of information about the real object. The term that is usually used to describe model variants of the same system is model *granularity*. Granularity refers to the amount of detail that a model reveals, like magnifying glasses with higher magnification reveal more spatial details of an object. In modeling for control, the magnifying glass could refer to a frequency range as well as a finer spatial subdivision. Different facets of system models may lead to models with a completely different mathematical representation.

In industrial modeling practice this often results in heated disputes between departments about how a system model should be done and what phenomena should be included. The reason is that they want to ask differ-



(a) Sketch (b) Pencil Drawing (c) Black&White Photo

Figure 1.2 Different representations of Rodins sculpture "The Thinker". Inclusion of a holographic picture was rejected due to budget reasons.

ent questions about the same system, when budgets and time schedules only allow the development of one model. This procedure often leads to *over-modeling*: models contain more details than necessary. As a result, the combined model may not be the best possible for any of the interesting questions. This typical problem demonstrates a real need for flexible modeling languages and model libraries, see [Åström, 2002]. The cost of developing a model is high, therefore we want the same model to answer as many questions as possible about the system. High level models, often called meta-models, should provide straightforward ways to switch between different model implementations. Various terms have been used to describe this property of models. Multi-facet [Nilsson, 1993] modeling or multi-paradigm modeling [Mostermann and Vangheluwe, 2000] have been used to denote models which combine several behavioral descriptions of a model into a meta-model. A meta-model representation in a computer tool should present the user with intuitive means to select the model facet that best answers a particular question.

A typical problem in process engineering is when process design engineers meet control engineers and they try to settle for a common model. The design engineers want a model that optimally represents the steady state behavior of the system over the whole operating range. The control engineers want a model that represents the dynamic behavior of the sys-

tem in the vicinity of the crossover frequency of the feedback loop. When controllers with integral action are used, the closed loop gain is infinity at low frequency and therefore the accuracy of the steady state model is not important. The dissimilarity of model purposes is frequently not understood by all members of an engineering team. From personal experience I can say that this problem is a major obstacle for successful teamwork in engineering projects.

Webster's dictionary defines "system" as "a regularly interacting or interdependent group of items forming a unified whole." The key property here is the *interaction* of items. The notion of system thus implies that it is possible to divide the domain of interest into meaningful subunits. This observation is the starting point of all work seeking to build libraries of reusable model parts. It will be a recurring theme in Chapter 6.

1.4 A Turbine System

A micro gas turbine system has recently been modeled in a master's thesis project using the ThermoFluid and other Modelica libraries, [Haugwitz, 2002]. It is a typical example of a multi-domain system model which demonstrates the strengths of modeling based on libraries and the need for flexible models of varying degrees of complexity.

A micro turbine system is a small, compact unit for decentralized generation of electricity and heat, a so called combined heat and power plant. The recent de-regulation of the electricity market has spawned the development of these types of systems which did not exist a few years ago. Customized solutions based on micro gas turbines are actively developed now. The first generation systems where not designed for islanding power production during blackouts of the electrical grid, but customers request this additional feature. System models in several degrees of granularity help to speed up the development process of the more advanced controls needed for islanding power generation.

For the particular case of this system, accurate steady state models were available but four types of dynamic models were needed:

- Simple, low order models for control design.
- Models that are suitable for hardware-in-the-loop tests of controllers for the main, continuous controls.
- Dynamic Models for off-line simulations and tests. These should be as accurate as possible and should be as close as possible to steady state models.

- Simple models of the gas turbine and all auxiliary systems for detailed testing of the discrete sequential controls of start-up and safety procedures.

All models are essentially for the same system, but with the focus on different aspects and with different questions in mind. Clearly, reusability and sharing of implementation code between these models results in a big gain in productivity.

The implementation of the system model makes heavy use of many existing model libraries. Around 95 % of the total model code is from library models, the rest is divided between creation of new models and composition of subsystems from libraries and new models. Without heavy code reuse, the project clearly would have been infeasible for a four month master's thesis project. Time was too short to fulfill all wishes for modeling, but the first three of the above mentioned models could be realized and the last remaining model would make complete reuse of the existing models.

1.5 How the Work Developed

My first attempt of dynamical systems modeling was in a student project with the goal to model the combined heat and power plant of the Technical University Hamburg Harburg with Simulink™ [MathWorks, 2001b]. The attempt ended with the firm conclusion that the directed, signal based modeling formalism of block diagrams, the very basis of Simulink, was completely inadequate for physical systems modeling. That spawned the search for better tools and more appropriate formalisms.

The next attempt was to use the object-oriented language and the simulation environment Smile, [Jochum and Kloas, 1994]. It was a joint development of the Technical University Berlin and GMD FIRST[3]. Smile was under development at the start of the project, a master's thesis with the goal to implement an object-oriented model library for power plant simulation. This attempt was quite successful, but it required a large initial investment of developing basic models for everything. During the literature review of dynamic power plant modeling it became obvious that many models that essentially contained the same or very similar mathematical models where implemented again and again. The problem was that the existing models were too unflexible to cope with even minor changes in the goal of the modeling task. This made it very clear that better methods

[3]Gesellschaft für Mathematik und Datentechnik, Forschungsinstitut für Rechnerarchitektur und Softwaretechnik, Berlin Adlershof.

for code reuse in modeling were urgently needed. Smile offered two clear advantages over earlier FORTRAN based models and Simulink:

- A clean separation between the modeling tool and the solution method for the model equations.
- An object-oriented, declarative, open and documented language to describe the model.

The Smile prototype tool was rather primitive: a language, a compiler, a command line executable and a text editor was all that was available. Still, the object-oriented features and hierarchical model composition made it possible to build complex models quickly. The Smile model library is still in use and has proven to be reusable for very different power plant designs, see [Buse, 2001], where a pressurized fluidized bed steam power plant is modeled using the same base models.

Smile had been designed in a Masters thesis [Biersack, 1994] and had a few essential shortcomings – no structured connectors and a clumsy implementation of equations – due to lack of modeling experience of the Smile developers. When the Modelica initiative was started as an attempt to unify the know-how of the separate groups that had worked on object-oriented modeling languages, each group with the focus on a particular engineering domain, it became obvious that this was a great opportunity to develop a clean, declarative, object-oriented modeling language.

During the first year of the Modelica development, I was involved in the detailed modeling of a solar thermal central receiver power plant integrated with a conventional heat recovery boiler [Tummescheit and Pitz-Paal, 1997] at DLR[4] in Cologne. The experiences from this project, and in this case especially the shortcomings of the currently used tools and the Smile language, were a valuable asset for the Modelica language design. In 1998 I joined the Department of Automatic Control at Lund University as a PhD student.

An interesting facet of the work was the parallel development of the Modelica language and modeling projects based on model libraries. Work on either side of the border between language development and use gave feedback for the work in the other area. A recurring theme was the modeling of physical properties of fluids. Most standard commercial packages for property calculation do not consider the specific requirements for dynamic simulation. This shortcoming made it necessary to implement physical property calculations from scratch all too often. Interaction with serious industrial modeling projects was another important aspect of the thesis work:

[4] Deutsches Zentrum für Luft- und Raumfahrt e. V.

- Modeling of a solar thermal steam power plant, [Tummescheit and Pitz-Paal, 1997].
- Combustion engine modeling at Ford Motor Company [Tiller *et al.*, 2000].
- Fuel cell system modeling in collaboration with United Technologies Research, UTRC.
- Refrigeration cycles, especially evaporators [Jensen and Tummescheit, 2002] in collaboration with DTU[5] and UTRC.
- Modeling of steam networks for a paper plant in collaboration with Solvina AB, [Lindstrand, 2002].

This interaction was important to make sure that language and library design were in accordance with real industrial needs. The fuel cell systems library developed at UTRC is an application that was not included in the intended use of the ThermoFluid library in its first design iteration, but is now the largest application library built on top of ThermoFluid. The object-oriented design has proven flexible enough to add chemical reactions, membrane diffusion and electrochemistry to the existing library and still make optimal use of the existing code base.

[5]Danish Technical University

2

Modeling Techniques

Abstract

An overview of the mathematical basics for the representation of dynamics outlines the scope and needs for a modeling language. Structuring of models in libraries is the other pillar of object oriented modeling. Model calibration and validation is the step that tunes general purpose models to resemble real systems. Modeling tools define the framework for the implementation of the mathematics and structure into reusable building blocks.

2.1 Representation of Dynamics

All models use mathematics as their foundation to express the aspects of reality that are of interest in building a model. A modeling language should thus be well suited to express the mathematical formalisms that are used for modeling. The range of concepts needed to model physical systems and their man-made controls is very broad. A quick inspection of existing modeling tools and languages reveals that their design is typically done in the following way: First choose the appropriate mathematical formalism that is needed to express models for a specific purpose and then the language or tool is designed to handle that case well. Often this decision is hidden in the choice of an engineering domain which then in turn leads to the choice of mathematics. Models for simulation are solved using methods in numerical mathematics. A considerable part of the modeling effort has to be spent on deriving models that have good numerical properties. The choice of the model is often strongly influenced by the reliability or availability of the numerical solution methods. Sometimes particular numerical methods are also integrated in the modeling language. Some of the formalisms can also be represented graphically. This can be of great value to communicate complex model semantics to humans.

Reality is complex and so are the models that are derived in an attempt

to capture the behavior of real systems. For most practical purposes the models have to be simplified substantially before they are useful. Many model reduction techniques exist, heuristic ones as well as methods based on established mathematical methods like singular perturbations, see [Lin and Segel, 1988]. One of the important simplifications in modeling of dynamical systems are *time scale* abstractions. Three of these time scale abstractions are very common:

Slow ⇒ constant: features of the system that change much slower than the current time scale of interest are treated as constants, e. g., ageing effects.

Fast dynamics ⇒ steady state: dynamics which settle on a timescale faster than those of main interest in the model are treated as always being in steady state.

Short time ⇒ impulse Changes in conserved quantities which happen in much shorter times than those of interest are treated as jumps.

As presented here, timescale decomposition is used as a heuristic model simplification procedure by engineers, but it can be formalized using singular perturbation theory, as will be discussed later in this section.

Physics is very accurate with accounting of fundamental extensive quantities like mass, momentum and energy. The accounting balance for these quantities constitutes the core of many physical models. One has to be aware though, that conservation-like laws often include source terms, a contradiction to conservation, e. g., for species mass balances in chemical reactions. The conserved quantity is simply used as an accounting basis for practical reasons. The advantage of fundamental extensive quantities is that they are easier to verify. A drift or error in a fundamental extensive quantity gives an estimation of the numerical error of the solution method.

Modelica was conceived from the beginning to be a domain independent language, but with a focus on system dynamics of physical systems. This leads to a preferred choice of mathematical tools, differential equations of various flavors. Ordinary differential equations (ODE) deal with problems with one independent variable, which always represents time in dynamical systems. Differential algebraic equations (DAE) add algebraic equations to an ODE. Partial differential equations (PDE) treat problems with more than one independent variable, usually space and time. The time scale abstractions and also models of sampled data systems arising from models of computer controlled systems lead to hybrid – discrete time and continuous time – systems. Pure discrete time dynamical systems can be expressed in many formalisms. Some of them, such as finite state machines, Petri nets and Grafcet, have been considered in the design of the

Modelica language.

Ordinary Differential Equations

Ordinary differential equations (ODE) are the workhorse for modeling and simulation of dynamical systems. Nonlinear ODE exhibit an amazingly rich spectrum of behavior considering that their basic structure is relatively simple. They are applied to diverse and countless problems in all natural and social sciences. When the "Method of Lines" discretization is used, PDE are transformed into ODE with a special structure. System dynamics is a branch of applied mathematics that has ODE as its main subject. This branch includes such fashionable subjects as chaos theory and bifurcations. Beyond all fashion and in line with the main subject of this thesis they provide the theoretical background for dynamic modeling of engineered systems. ODEs are very powerful in describing the behavior of such systems in a way that permits both computational exploration and analysis.

Choosing a notation in accordance with common practice in control oriented modeling, using a vector of unknowns $x \in \mathbb{R}^n$ and a vector of exogenous inputs $u \in \mathbb{R}^p$ with known time trajectories, an ODE can be written in state-space form as:

$$\dot{x} = f(x, u) \tag{2.1}$$

and $f : \mathbb{R}^n \mapsto \mathbb{R}^n$, assuming $\dim(\mathrm{u}) = \mathrm{p} \leq \dim(\mathrm{x})$. When used in control oriented models, a measurement equation is added to the differential equation:

$$y = g(x, u) \tag{2.2}$$

where the vector $y \in \mathbb{R}^m$ denotes the measurable outputs from the system with $g : R^n \mapsto \mathbb{R}^m$. Particular solutions to ODEs can only be given when additional information about initial conditions is given. The initial conditions can be either conditions on the states $x(t_0) = x_0$ or conditions on the state derivatives $\dot{x}(t_0) = \dot{x}_0$, the second is mostly the steady-state condition $\dot{x}(t_0) = 0$. When $\dim(\mathrm{x}) = \mathrm{n}$, exactly n initial conditions in either of the two forms have to be given that permit a unique solution to $x(t_0)$.

Because differential equations can be amazingly complex and are often difficult to analyze, it is common practice in many engineering disciplines to linearize them around a stationary point $\dot{x} = 0$. Theory for linear systems is well developed and most powerful control design and analysis methods use linear ordinary differential equations as their starting point. The equation is linearized by taking the partial derivatives of the func-

tions f and g with respect to x and u at a point $u_0, x_0, \dot{x}_0 = 0$.

$$\begin{aligned} &\forall i, j \in 1, 2, \ldots n, \quad k \in 1, 2, \ldots p, \quad l \in 1, 2, \ldots m \\ &A_{ij} = \frac{\partial f_i}{\partial x_j}, \quad B_{ik} = \frac{\partial f_i}{\partial u_k} \quad C_{lj} = \frac{\partial g_l}{\partial x_j}, \quad D_{lk} = \frac{\partial g_l}{\partial u_k} \end{aligned} \tag{2.3}$$

This linearized, time invariant ODE (LTI-model) with coefficient matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ is the standard model for control design. The linearization is valid around u_0, x_0, therefore new variables $\tilde{x} = x - x_0$, $\tilde{u} = u - u_0$ and $\tilde{y} = y - y_0$ are introduced, resulting in

$$\begin{aligned} \dot{\tilde{x}} &= \mathbf{A}\,\tilde{x} + \mathbf{B}\,\tilde{u} \\ \tilde{y} &= \mathbf{C}\,\tilde{x} + \mathbf{D}\,\tilde{u} \end{aligned} \tag{2.4}$$

Often $\mathbf{D} = 0$ when the control signal is not directly coupled with the output. It is also possible to linearize the nonlinear system (2.1–2.2) along a trajectory for given x_0 and u. This is closely related to the way that numerical methods use to find a solution to (2.1–2.2). The requirement is that the derivatives A_{ij} etc. exist and are sufficiently smooth along the trajectory. This results in the linear, time-varying ODE

$$\begin{aligned} \dot{\tilde{x}} &= \mathbf{A}(t)\,\tilde{x} + \mathbf{B}(t)\,\tilde{u} \\ \tilde{y} &= \mathbf{C}(t)\,\tilde{x} + \mathbf{D}(t)\,\tilde{u} \end{aligned} \tag{2.5}$$

This simplification captures the behavior of the non-linear ODE much better than the linearization with constant coefficients along the chosen trajectory. Model reduction techniques based on trajectory linearizations are discussed in [Öhman, 1998].

On the numerical side, a lot of research has been done in the last decades to get high quality numerical approximations to the solutions of ODE and DAE. Quality refers to the question "How much computing time is needed to solve a given problem with an upper bound on the global error of the solution". Exact answers to this question are difficult to obtain, but satisfying error bounds for engineering purposes are the tolerance parameters in most state-of-the-art ODE solvers. The amount of work that a chosen tolerance requires still has to be found by trial and error for each problem.

An important classification regarding the numeric behavior of ODE is the classification as stiff or non-stiff. Naively, a stiff differential equation has modes at drastically different time scales. Experts in numerical mathematics define stiffness using the following operational definition (quoted from [Hairer and Wanner, 1996], original from [Curtis and Hirschfelder, 1952]): *stiff equations are equations where certain implicit methods, in particular BDF*[1]*, perform better, usually tremendously better, than ex-*

[1]Backward Differentiation Formulas

plicit ones. The problem in classification is that many factors play a role, among others the smoothness of the solution, the dimension of the system and the integration interval. The most often quoted factor and undeniably a very important one is the magnitude ratio of the largest and smallest eigenvalues of the Jacobian $\partial f/\partial x$. If the magnitude ratio of the largest to the smallest eigenvalue is a large number, say 1000 or more, the equation is stiff.

The current situation is that the selection of the right solver for a given problem requires a lot of experience and basic knowledge about the system. By engineers it is often regarded as much an art as a science.

Singular Perturbations There is a connection between stiff ordinary differential equations and differential algebraic equations, the subject of the next section.

Consider the following model with two groups of time scales

$$\dot{x} = f(t, x, z, \varepsilon) \tag{2.6a}$$

$$\varepsilon \dot{z} = g(t, x, z, \varepsilon) \tag{2.6b}$$

Here z are the fast states and x are the slow states of a stiff ODE system. The fast states can be eliminated by letting $\varepsilon = 0$ which implies that $g(t, \hat{x}, \hat{z}, \varepsilon) = 0$. Under the assumption that the Jacobian

$$\frac{\partial g(t, x, z, \varepsilon)}{\partial z}$$

is invertible in the neighborhood of the solution to (2.6a), this equation can be solved for $\hat{z}(t, \hat{x}, \varepsilon)$. Replacing z in the first equation with this expression results in the simplified model

$$\dot{\hat{x}} = f(t, \hat{x}, \hat{z}, \varepsilon) = \hat{f}(t, \hat{x}, \varepsilon).$$

The technique is called singular perturbation, see [Lin and Segel, 1988]. If the problem is not solved for $\hat{z}(t, \hat{x}, \varepsilon)$, the problem is equivalent to a DAE of index 1 while the original problem is a stiff ODE system. The DAE can thus be regarded as the limiting case of $\varepsilon \to 0$ of a stiff ODE, which in many cases is the origin of DAE.

Remark: in simple cases the heuristic engineering method of using quasi steady state approximation leads to the same model reduction that singular perturbation theory provides.

Differential Algebraic Equations

While ODE are the form of differential equations that has gained most attention in engineering numerics, there are few engineering systems which

actually can be described by an ODE without algebraic equations for some of the variables. A general, non-linear DAE can be written as

$$F(x, \dot{x}, y, t) = 0 \tag{2.7}$$

where x are the variables that appear differentiated and y, the algebraic variables. In some cases, particularly when the DAE is the result from a singular perturbation, the DAE can be written in semi-explicit form:

$$\dot{x} = f(x, y, t) \tag{2.8a}$$
$$0 = g(x, y, t). \tag{2.8b}$$

From a numerical point of view, most semi-explicit differential algebraic equations (DAE) can be integrated like ODEs, when the initial conditions are known. An essential requirement for the solution of DAE is that the initial values x_0, z_0 are *consistent* with the algebraic equations $0 = g(x_0, y_0, t_0)$. Finding initial conditions may be a serious practical problem. A general assumption to achieve this is that the Jacobian

$$\frac{\partial g(x, y)}{\partial y}$$

is invertible in a neighborhood of the solution of (2.8a). Equation 2.8b then possesses a locally unique solution $y = G(x)$ ("implicit function theorem") which inserted into 2.8a reduces that equation to an ordinary differential system in state space form, see [Hairer and Wanner, 1996]. The equations 2.8a and 2.8b are then said to be of *index 1*. This procedure is the same as the second step in the singular perturbation simplification. Another way to express this is to say that "some DAE are very similar to ODE" [Pantelides, 2000].

The geometrical interpretation of a DAE compared to an ODE with the same number of dynamic states n is as follows. Solution trajectories of the ODE can start on any point in $\mathbb{R}^n$ and all points in $\mathbb{R}^n$ are part of a legal solution trajectory. The solution of a DAE is *constrained* to the manifold in $\mathbb{R}^n$ defined by $0 = g(y, x)$. All legal solution trajectories which are *consistent* with the DAE have to always be on that manifold. If there are m independent constraints between states, e.g., $k = 1 \ldots m$, $0 = g_k(x_j, x_i)$, the dimension of the manifold is $n - m$. If there is exactly one such constraint, the manifold is a surface of dimension $\mathbb{R}^{n-1}$ in $\mathbb{R}^n$.

DAE can be linearized in the same way as ODE. To simplify notation, the DAE is written as

$$F(\dot{z}, z, t)$$

where z is the union of x and y. Linearizing around a trajectory $z_0(t)$ and applying the same change of variables as for ODE, $\tilde{z}(t) = z(t) - z_0(t)$, we then get

$$\mathbf{E}(t) = \frac{dF}{d\dot{z}} \qquad \mathbf{A}(t) = \frac{dF}{dz} \tag{2.9}$$

$$\mathbf{E}(t)\frac{\tilde{z}}{dt} = \mathbf{A}(t)\tilde{z} + b(t) \tag{2.10}$$

If $\mathbf{E}(t)$ is regular for all t this is an ODE, but $\mathbf{E}(t)$ may change rank along the trajectory. The matrix $\lambda\mathbf{E}(t) - \mathbf{A}(t)$ is called a matrix pencil. It is singular if $det(\lambda\mathbf{E}(t) - \mathbf{A}(t))$ is singular for all λ and otherwise regular.

The Notion of Index

The problem of "high index" differential algebraic equations is closely linked with the idea of object-oriented modeling. This connection may not be obvious at first sight. Object orientation is perceived as belonging to the computer science domain while high index DAEs are a mathematical problem. We are going to look more closely at one of the definitions of high index. Two examples illustrate how high index problems naturally arise from the division of systems into subsystems, one of the most important features of object orientation and demonstrate the problem to define consistent initial conditions for such systems.

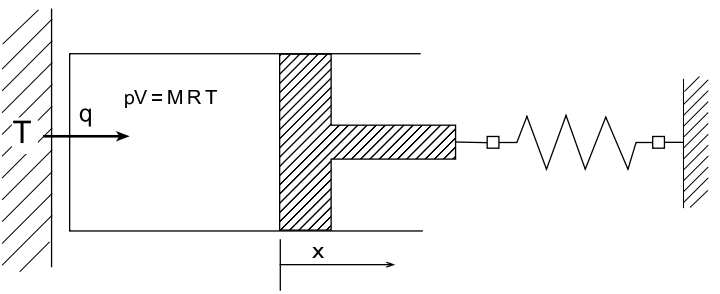


Figure 2.1 Coupling of thermodynamic control volume and piston

EXAMPLE 1—INITIAL CONDITIONS

Consider the simple system of a gas filled control volume in contact with a heat reservoir, Figure 2.1, closed by a piston held by a spring. Assuming that the cylinder is tightly closed, the equations for the control volume

are:

$$\begin{aligned} pV &= M_{gas}RT \quad \rightarrow pV = const\ T \\ M_{gas}c_v\frac{dT}{dt} &= -p\frac{dV}{dt} + q \\ V &= V_0 + Ax \end{aligned}$$

For the control volume by itself, two initial conditions have to be specified, typically pressure and temperature.

The force balance on the piston gives

$$m_{piston}\frac{d^2x}{dt^2} = kx - pA$$

where k is the spring constant and A is the piston area. The piston as a stand-alone model needs two initial conditions, position and speed. When the system is combined, it is no longer possible to specify four independent initial conditions. With given piston position and speed, only the temperature *or* the pressure can be specified.

When the gas volume is used for the initial condition of the control volume, it is obvious that the volume and piston position have to be consistent. While this example is trivial, finding consistent initial conditions for more complex high index DAE is difficult. Simulation tools for areas where high index problems are common provide algorithmic aid for finding consistent initial conditions. □

Definition: Equation 2.7 has *differential index* $di = m$ if m is the minimal number of analytic differentiations

$$F(x, \dot{x}, y, t) = 0, \quad \frac{df(x, \dot{x}, y)}{dt} = 0, \quad \ldots, \quad \frac{d^m f(x, \dot{x}, y)}{dt^m} = 0 \qquad (2.11a)$$

such that equations 2.11 can be transformed by algebraic manipulations into an explicit ordinary differential system $\dot{x} = \Phi(x, u)$ which is called the "underlying ODE".

Two practical problems arise with the numerical solution of DAE:

- calculation of consistent initial conditions and
- reliable numerical solution of the trajectories.

The details of the difficulties of the numerical solution are described in [Hairer and Wanner, 1996]. A few solution methods for DAE are actually capable of handling high index DAE directly. Another possibility is to use

the definition (2.11) as a symbolic procedure to reduce a DAE to index 1. This option offers more and particularly more reliable possibilities for the numerical solution.

If high index DAE would be a rare exception, they would not deserve much attention. They are very common, especially when systems are modeled by dividing them into subsystems, the key property of object oriented model libraries. Because of this, libraries would be useless if the simulation environment could not deal with the index problem. Either the numerical solver has to deal with the index directly – this is limited to cases of index 2 or 3 – or the simulation tool has to do symbolic index reduction. Modelica is designed to allow symbolic index reduction. The Dymola[2] tool that was used in the development of the ThermoFluid library does a good job at detecting and (most often) reduce higher index to index one which Dymola can integrate, but nonetheless it is valuable to know when and why high index DAE will occur. The most typical occurrences are:

Simplification: imposing constraints between states (or quantities related to them) due to a simplifying assumption introduces an index problem. In process engineering these constraints are often in the form of "unmodeled" flows, e.g. mass flows which are not calculated explicitly. Standard cases are

- Incompressibility ⇒ volume constraint
- Phase equilibrium ⇒ constrained sum of volumes.
- Reaction equilibrium ⇒ fixed ratio between concentrations of components.

Coupling: high index due to coupling of models is a special case of simplification because it arises from idealized couplings, e. g., connecting two capacitors in parallel. The simplification is to assume that the resistance between them really is zero. In this case the current between the capacitors is the unmodeled flow. High index due to coupling is the most important reason why object-oriented modeling requires automatic handling of high index problems.

Perfect control: Specifying the trajectory of an output variable as a time function imposes a trajectory constraint on the states and thus also leads to an index problem.

The problem of high index models is usually discussed in detail in all modeling courses for process engineering, see [Pantelides, 2000] and [Preisig, 2001]. The following example demonstrates that high index problems are often introduced through simplifying assumptions. While this is

[2]Dymola is a simulation environment using the Modelica language by the Swedish company Dynasim AB.

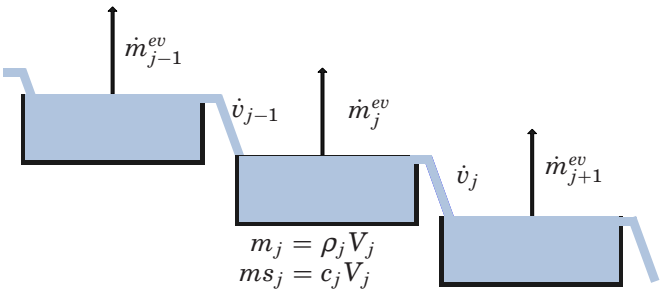


Figure 2.2 Series of salt basins

typical and well-known for mechanical and electrical systems, this example shows a simple process model of open evaporation basins for salt harvesting. The example is a variation of an example from [Weiss and Preisig, 2000]

EXAMPLE 2—SALT BASINS IN SERIES

Consider a series of salt basins as in Figure 2.2. At the top of each basin are inflows of low concentration salt brine. Water evaporates in each basin and the outflow into the next lower basin has a higher salt concentration. It is difficult to model the exact flow into the next basin[3], but an adequate simplification is to assume that the brine volume equals the maximum value of the basins volume. The brine in the j-th basin is assumed to have a density that depends on the salt concentration c_j. The model for each basin $j, j \in 2..N$ can be written as follows:

$$\dot{m}_j = \rho_{j-1}\dot{v}_{j-1} - \dot{m}^{ev}_j - \rho_j \dot{v}_j \qquad \text{total mass balance} \quad (2.12a)$$

$$\dot{ms}_j = c_{j-1}\dot{v}_{j-1} - c_j \dot{v}_j \qquad \text{salt mass balance} \quad (2.12b)$$

$$m_j = \rho_j V_j \qquad \text{mass } m \quad (2.12c)$$

$$ms_j = c_j V_j \qquad \text{salt mass } ms \quad (2.12d)$$

$$xs_j = \frac{m_j}{ms_j} \qquad \text{salt concentration } xs \quad (2.12e)$$

$$\rho_j = g(xs_j) \qquad \text{function to calculate density } \rho \quad (2.12f)$$

The inflow conditions into the first basin, c_0 and $\dot{v}_0$, are known. For a known evaporation mass flow rate $\dot{m}^{ev}_j$ this is a well-posed index two DAE problem. For each basin, there are 6 variables: $m, ms, \dot{v}, xs, \rho$ and c ($V, \dot{m}^{ev}$ are assumed known), as well as 6 equations. But there are

[3]The overflows in open air salt basins have irregular geometries.

no explicit equations for $\dot{v}$, these have to be calculated from the volume constraint and the mass balances. By differenting the volume constraint, the concentration equation and the density definition, it is possible to compute the volumetric flow explicitly:

$$\dot{v}_j = \frac{g'(xs_j)(xs_j - xs_{j-1})\rho_{j-1} + \rho_{j-1}\rho_j}{\rho_{j-1}^2}\dot{v}_{j-1} + \frac{g'(xs_j)xs_j + \rho_{j-1}}{\rho_{j-1}^2}\dot{m}_j$$

Modelica was designed to make algorithmic approaches to these transformations possible. The algorithmic conversion to an index one problem involves *detecting* the constraint equations and differentiating them symbolically. Two algorithms provide the necessary methods: Pantelides' algorithm [Pantelides, 1988] and the method of Dummy-Derivatives [Mattsson and Söderlind, 1993]. A remaining problem is to select which of the differentiated variables is going to be used by the numerical integrator.

Clearly, the index two DAE with the implicit definition of the volume flow is much easier to derive than the equation which is the result of transforming the problem to an index one problem. The algebraic constraint is caused by the constant volume assumption. Equations 2.12c, 2.12e and 2.12f have to be differentiated, then it is possible to calculate $\dot{v}_j$ explicitly and reduce the problem to index one. □

There are three points to note in the index reduction procedure used above:

- In spite of two differential equations there is only one independent state per basin.
- Index reduction involves a *global* analysis of the equations, the volumetric flow $\dot{v}_j$ contains variables from the upstream basin.
- The algorithmic solution and manual index reduction yield the same solution.

With object-oriented modeling of each basin and local index reduction by the modeler this would mean that the concentration xs_j of the upstream basin would have to be in the connectors, which is not obvious from the original equations. Similar problems occur in boiler modeling, see [Åström and Bell, 2000].

In some engineering domains the presence of a high index DAE is regarded as a modeling error, which may actually be true. In other domains (electrical and mechanical) high index problems occur naturally from standard modeling assumptions. These differences lead to drastically different ways of dealing with the index problem:

- Send the modeler back to the step 1 to resolve the problem with pen and paper, reformulating the model into an equivalent index 1 problem.
- Build the capacity of recognizing and resolving high index problems into the modeling tool.

Because Modelica is designed as a multi-domain modeling tool, it has to support automatic index handling. This does not mean that automatic index reduction is the silver bullet that solves all high index problems in the best possible way. There are a few exceptions when manual derivation of an index 1 problem is preferable to the automatic procedure. This is the case when the automatic procedure gives results with numerical disadvantages, at least with the current version of the Modelica language and the index reduction algorithms in the Dymola tool. Examples of these rare cases are presented in Chapter 4. But independent of an automatic handling, model users have to understand the implications of high index problems in order to provide the right initial conditions.

Partial Differential Equations

Partial differential equations (PDE) are the mathematical formulations that permits the highest level of detail for system level models. They provide the tool corresponding to the magnifying glass with the highest magnification power. PDEs form an essential part of the mathematical description of physical systems depending on space and time. They arise in a large number of modeling applications, but are not that common in systems level modeling. The reason for this is that PDE can have widely differing properties and that they are much more difficult to deal with numerically than ODEs. Using PDE for dynamical systems means that we have at least one more independent variable apart from time. Usually these are spatial coordinates, but variables can also be distributed in other dimensions like particle sizes in crystallization processes.

Numerical solutions to PDE for design calculations are used in almost all engineering disciplines and are established as independent research areas, e. g., Computational Fluid Dynamics (CFD) and Finite Element Methods (FEM).

Consider a set of PDEs expressed over the solution domain Ω which is a compact region in $\mathbb{R}^{m+1}$ and its boundary $\delta\Omega \in \mathbb{R}^m$, compare Figure 2.3 for a two-dimensional example. If the independent variables are separated into time t and a vector of spatial variables x, then the dependent variables u can be written as

$$u = u(x, t)$$

where $u \in \mathbb{R}^n, x \in \mathbb{R}^m$ and $t \in \mathbb{R} \geq 0$. In mathematical texts the independent space and time variables are usually treated alike, here Ω is used

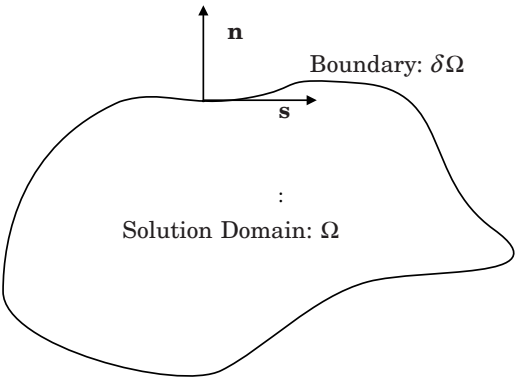


Figure 2.3 Computational domain Ω of a PDE

.

for the union of the spatial dimensions and time. The partial derivatives are often denoted as follows

$$(u_x)_{ij} = \frac{\partial u_i}{\partial x_j}, \qquad (u_{xx})_{ijk} = \frac{\partial^2 u_i}{\partial x_j \partial x_k}, \qquad \text{etc.}$$

For modeling of physical systems, PDEs of second order account for many important applications. For simplicity of exposition we restrict the following discussion (adapted from [Logan, 1994]) to one space dimension x, a single scalar variable u and get

$$G(x, t, u, u_x, u_t, u_{xx}, u_{tt}, u_{xt}) = 0 \tag{2.13}$$

where the independent variable x and t lie in some given domain Ω. By a solution to (2.13) we mean a twice continuously differentiable function $u(x, t)$ defined on Ω which, when substituted into 2.13 reduces 2.13 to an identity on the domain Ω. These solutions are called classical or genuine. When the solutions are extended to include functions which are discontinuous or have discontinuous derivatives, such functions are called *weak solutions*. For one-dimensional PDEs the solution of 2.13 can be represented as a smooth surface in three-dimensional xtu-space lying over the domain Ω in the xt-plane, as shown in Figure 2.4.

Boundary and Initial Conditions In ordinary differential equations solutions depend on arbitrary constants. Initial or boundary conditions fix the arbitrary constant and pick out a unique solution. The situation

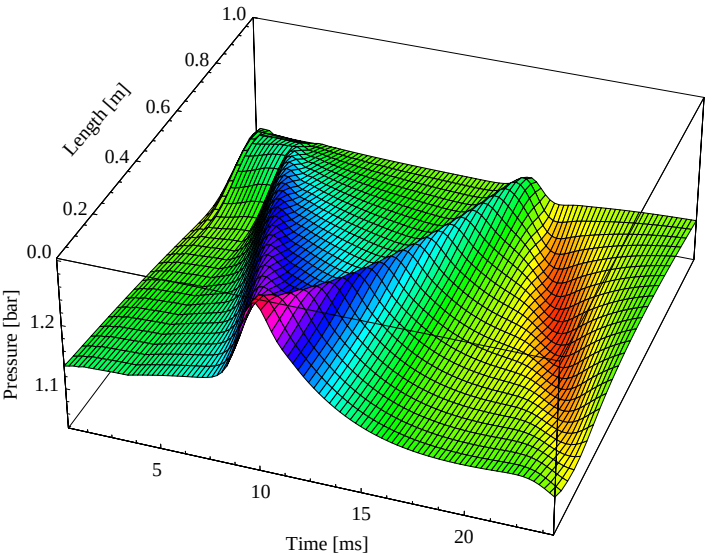


Figure 2.4 Graphical representation of the solution to the pressure wave demo model from the ThermoFluid library. The model simulates a pipe whose back end at $(x = 1)$ was instantaneously opened to a lower pressure a few milliseconds before this "snapshot" of spacetime was taken. It represents the space-time diagram of a pressure wave traveling back and forth in a one-dimensional pipe model. It clearly shows one characteristic of a hyperbolic PDE, the wave moving from the back end of the pipe $(x = 1)$ to the front, reflection at the front $(x = 0)$ and traveling towards the back end again. The slope of the linear ridges between the x- and t axes correspond to the characteristics, $u + c$ and $u - c$ with u gas flow speed and c speed of sound.

for PDE is similar. Imposing initial and boundary conditions selects one from the infinitely many solutions. These auxiliary conditions are usually suggested by the underlying physical problem or by the type of the PDE. Conditions on u or its derivatives at $t = 0$ along segments of the domain boundary are called initial conditions while conditions along any other curves of the xt-plane are called boundary conditions. For a well-posed physical problem, a complete set of initial conditions has to be specified, i. e. $u(x, t = 0)$ has to be known for all of Ω.

A typical modeling error in PDE problems is to specify non-physical boundary conditions which lead to an ill-posed problem. For example, in heat - or electrical problems, a potential (temperature or voltage) has to be specified somewhere on the boundary, otherwise the level of the potential is free to float and any potential level would give a mathematical solution.

The mathematical literature traditionally classifies boundary condi-

tions into three categories:

- Dirichlet boundary conditions: $u = g$ on $\delta\Omega$
- Neumann boundary conditions: $\frac{\partial u}{\partial n} = g$ or $\frac{\partial u}{\partial s} = g$ on $\delta\Omega$
- Robin boundary conditions: $\frac{\partial u}{\partial n} + ku = g$, $k > 0$ on $\delta\Omega$

Here n denotes an outward normal vector and s a tangential vector with respect to the boundary $\delta\Omega$ at a given point, see Figure 2.3. Dirichlet boundary conditions can be applied exactly on $\delta\Omega$ when g is analytic, but practical problems with Neumann and Robin conditions are that errors are easily introduced when representing them numerically.

In some interesting problems the calculation of the boundary location is part of the problem and not known in advance. Moving boundary problems are associated with time dependent processes where behavior changes abruptly at a boundary which changes its location in space. Flame propagation and phase change problems are typical examples of moving boundary problems.

Symbolic approaches to the solution of PDE often involve free functions of a certain type which have to fulfill the boundary conditions. Fourier's approach of solving the heat conduction problem was to construct the solution as an infinite sum of functions all of which fulfill the homogeneous boundary conditions.

Classification For second order PDEs, a special classification has been established both from a mathematical standpoint and based on the physical phenomena that they arise from. From the physical point of view, there are three types of equations: those that govern diffusion processes, those that govern wave propagation and those that govern equilibrium phenomena. Equations of mixed type also occur, but locally for a fixed point $\omega \in \Omega$ they belong to one of the three categories. Considering a single, second order PDE of the form

$$au_{xx} + 2bu_{xt} + cu_{tt} = d(x, t, u, u_x, u_t), \qquad (x, t) \in \Omega \tag{2.14}$$

where a, b and c are continuous functions on Ω and not all of a, b and c vanish simultaneously at some point of Ω. Also the function d on the right hand side is assumed to be continuous. The equation is classified on the basis of its second order derivatives using the discriminant Δ, defined by $\Delta = b^2 - ac$. Then (2.14) is said to be

hyperbolic if $\Delta > 0$, a wave propagation equation,

parabolic if $\Delta = 0$, a diffusion process equation or

elliptic if $\Delta < 0$, an equilibrium or steady-state process.

In the case of elliptic PDE, time derivatives do not occur and only space coordinates are independent variables. They are a rare exception in dynamical systems modeling. It should be pointed out again that the type of the equation may change on Ω because Δ depends on x and t.

Numerical Methods for PDE The numerical solution of PDE is generally a difficult problem. Solutions are always approximations to the true mathematical solution of the equation and all the more to the real physics that they represent. Solutions can be very sensitive to small changes in parameters or boundary conditions. The situation is not hopeless, there are several numerical methods which can solve technically relevant problems reliably. There is active research in the area of solutions to PDE and from the many methods available only the key features of a few of them will be discussed.

A generalized concept that comprises the majority of numerical PDE solution methods is the "Method of Lines". The methods of lines converts time dependent PDEs into ODEs (or DAEs) with respect to time by choosing a fixed spatial discretization and usually also a fixed order approximation to the spatial derivatives of $u(x,t)$. The name "Method of Lines" is based on the fact that many of these methods approximate the solution of the equations on a fixed spatial grid of straight lines. Method of lines as a group name also refers to finite element methods where the spatial discretization is fixed, but the discretization uses triangles of different sizes. Traditional PDE solvers also use a fixed time discretization, but in recent years it has become common to use variable time step ODE solvers for discretized PDE.

The method of lines can provide good numerical approximations to the solution of PDE in a wide variety of applications. The family of methods of lines comprises finite difference, finite volume, finite element and weighted residual methods in which piecewise local or global approximation functions in the space dimension are used to convert hyperbolic and parabolic PDE problems into initial value ODE problems. The key advantage of the "Method of Lines" is twofold:

- It is straightforward to combine subsystem models which are described by DAEs or ODEs with PDEs because they are all converted to the same mathematical form.
- ODE and DAE solvers can be employed for the solution. These solvers have reached a higher degree of sophistication than PDE solvers. Many programs are available which permit fast and accurate solutions to large sets of DAE [Petzold, 1982].

The advantage of ODE/DAE solvers is that they use sophisticated algorithms to adjust the time step size and the order of a method in order to

control the user-specified approximation error [Gustafsson, 1992].

However, the advantage of the method points to the disadvantage: The DAE solver used for integration in time cannot control or even estimate the space discretization error. For coarse discretizations the spatial error can easily be an order of magnitude larger than the (controlled) time discretization error. There are no general guidelines on how to determine the spatial discretization such that a computationally efficient and accurate solution is obtained. Trial and error and engineering intuition are the prevailing methods. Adaptive or moving grid methods are attempts to handle the spatial discretization error better, but their complexity and restriction to special classes of problems has discouraged implementation in general purpose simulation tools [Oh, 1995]. This is one of the problems in using PDE in systems modeling: there are many special methods for special cases, which makes implementation of general purpose tools for all types of PDE prohibitively complex.

Discrete Time and Hybrid Systems

Discrete time dynamic systems arise from the time discretization of continuous systems or from time scale abstractions of such systems. In physical systems modeling they are the predominant way to model control actions of all kind: finite difference equations and discrete time transfer functions for feedback control and particular description formalisms such as Petri nets, [Murata, 1989; David and Alla, 1992], finite state machines, [Kohavi, 1978], Grafcet, [Årzén, 1994; Johnsson and Årzén, 1999], Grafchart [Årzén, 1996; Johnsson, 1999] or State Charts [Harel, 1987] for sequential control, safety and redundancy management control logic. Changes of the state of discrete time systems occur only at discrete points in time.

Most controllers are implemented as digital controllers with algorithms based on discrete time process models. The controllers often contain process models, e. g., in the form of Kalman filters. A discrete time model can be obtained by sampling the continuous time model. Sampling the continuous time ODE-model

$$\dot{x}(t) = \mathbf{A}x(t) + \mathbf{B}u(t), \qquad y(t) = \mathbf{C}x(t)$$

at equidistant time instants with sampling interval h results in the discrete time model

$$x(kh + h) = \Phi x(kh) + \Gamma(kh), \qquad y(kh) = \mathbf{C}x(kh)$$

where $\Phi = e^{AH}$ and $\Gamma = \int_0^h e^{As} B ds$. There are also approximative discretization methods, including Euler's method and Tustin's approxima-

tion. Numerical methods for solving ODEs are mostly based on more sophisticated discrete approximations. Further details on discretization of continuous time models can be found in [Åström and Wittenmark, 1990].

Hybrid systems combine the continuous behavior specified by differential equations with discontinuous changes specified by discrete event switching logic or finite difference equations at sample times. For successful modeling of hybrid phenomena, the semantics of the hybrid modeling language elements have to be well understood. In certain difficult cases even an understanding of the implementation of the computational solution procedure are necessary. At least three modeling situations lead to hybrid phenomena in systems modeling:

- Combination of continuous physical systems with discrete time, computer implemented control logic.
- Discrete approximations to physical phenomena which exhibit steep changes within short time- or length scales, e. g., the time scale abstractions mentioned in Section 1.3 like colliding objects in mechanics.
- The modeled device changes its characteristic behavior in the interesting range of operation so much that different states or even a different number of states are needed for an adequate description of the behavior. An example could be an evaporator that is flooded with liquid at startup.

Modeling on a macroscopic scale often gives rise to empirical relations that are piecewise functions. Typical examples are friction, pressure drop or heat transfer correlations. As long as the relations do not change the causality of the calculation, these are fairly easy to handle in simulation. Hybrid phenomena are still the topic of intensive research activity, even fundamental issues such as the existence and uniqueness of executions of hybrid automata are not understood in all details. Good introductions to ongoing research in hybrid systems is found in [Johansson *et al.*, 1999] and [Zhang *et al.*, 2000].

From the point of view of numerical mathematics, the following classification can be made:

1. ODE or index 1 DAE where a discrete event or discontinuous function does not change the index or the size of the state vector.
   (a) Only the right hand side of the ODE/DAE changes discontinuously, all states are C^0 continuous. The values of the states before and after the event are the same. Computational causality of algebraic variables in an index 1 DAE might change also.

   (b) The states may change discontinuously through an impulsive action. Physically, this arises from a time scale abstraction. The mathematical tool is a Dirac impulse the amplitude of which has to be calculated. The values of the states after the event have to be calculated by conditions or extra equations in the model.

2. ODE or DAE of any index where the index or the number of states changes after an event. Both of these change the dimension and possibly the structure of the dynamical problem.

   (a) States are added to the state vector. From an implementation point of view the problem is similar to 1(b) because the new state needs to be initialized. Algebraic variables may also be added but this is less problematic.

   (b) Constraints on states are introduced during integration and the index of the DAE changes. Causality might change and a physically reasonable treatment of the problem may include Dirac impulses.

In practical terms, the cases 1 are easier to handle than the cases 2, modeling of them needs both support from the modeling language and numerical solvers.

The possibility that a system of equations may become singular during integration due to topological constraints is a related problem. This occurs for the pendulum in Cartesian coordinates [Mattsson *et al.*, 2000] such that a different set of equations has to be used instead. Strictly speaking this is not a hybrid problem – the system's numerical condition gets gradually worse – but the implementation problems are related.

Many hybrid problems can usually be dealt with in an algorithmic fashion, but the challenge for a modeling language is to describe hybrid phenomena in an intuitive and completely declarative way. This is a complex task which is partially taken into account in Modelica although it does not yet offer satisfactory solutions for the more difficult cases above.

The first step in describing hybrid problems in a declarative manner is to write equations in a way that allows the tool to automatically *detect* and *locate* the time of events. This is done with the help of indicator or crossing functions (see [Cellier, 1991]) which change their sign at the discontinuity. In Modelica, these crossing functions are automatically generated from the declarative discontinuity conditions. Users only have to supply their own crossing functions when the discontinuities are hidden, e. g., in external programming code.

A particularly annoying problem for practitioners is the phenomenon of "chattering" which occurs sometimes in case 1a above. It is an unwanted

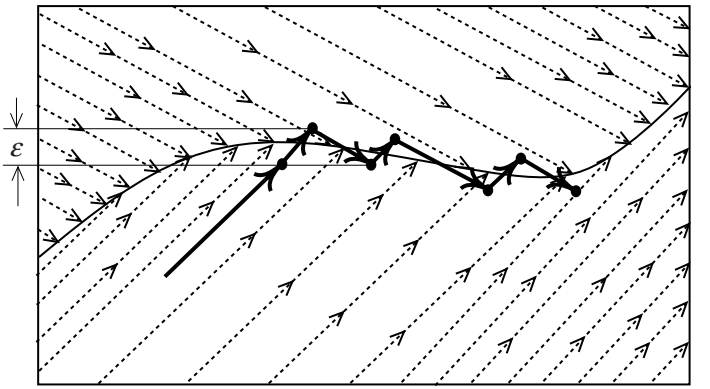


Figure 2.5 Chattering (ε not to scale)

effect which under certain circumstances is triggered by the exact event location. On the other hand, the exact event location is a precondition to high accuracy solution of the system trajectory. A zoom into the solution trajectory in the case of chattering is given in Figure 2.5. Two different right-hand sides of the ODE are symbolized by the two vector fields. In this case both vector fields give a gradient that drives the solution towards the discontinuity. The solver locates the exact time point on the trajectory when the discontinuity triggers and restarts the solver with the new right hand side. Note that omitting the step of locating the discontinuity exactly leads a large error in the solution trajectory because the wrong vector field is used for a large part of the time step. The restart is computationally expensive, multi-step high order methods restart with low order and small step size. In order to account for small numerical errors, the solver will not switch back to the old right hand side within a small ε-environment around the discontinuity. The gradient will drive the trajectory across the discontinuity again and this repeats for many times. The progress in simulation gets prohibitively slow and this is experienced as a stand-still in simulation.

Currently there is no simulation tool which treats the chattering problem in a satisfying way or would even give a clear diagnosis of the occurrence. The pragmatic solution as of today is to change the hybrid model to a smooth, continuous model, compare Figure 2.6. Physically this is often an admissible solution because often the hybrid phenomenon is a result of an abstraction that can equally well be parameterized in a smooth way. The change in Reynolds number between laminar and turbulent flow is such a case as well as boundary layers in temperature or velocity fields.

For certain classes of hybrid systems and for this type of chattering or "sliding mode" behavior, efficient and accurate simulation methods were explored in [Malmborg, 1998] and [Mattsson, 1996] using the concept of Filippov solutions. However, these methods are not general and not implemented in simulation packages yet.

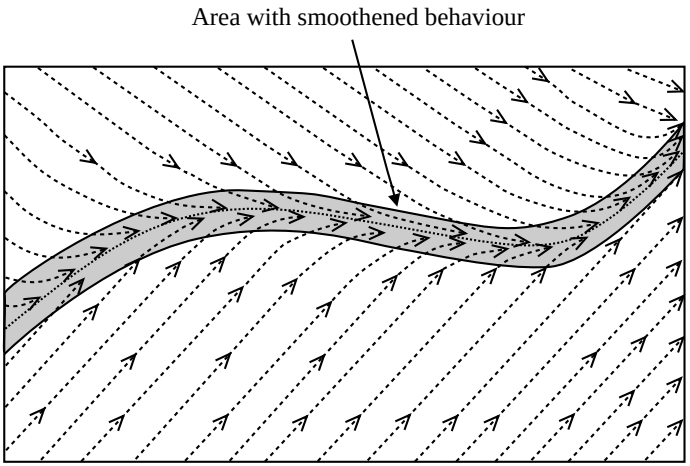


Figure 2.6 Chattering is avoided by smoothing the discontinuity. The Filippov solution can be visualized by letting the gray area shrink to zero width. The sum of the vector fields defines the gradient on the boundary.

From a modeling perspective it is very natural to combine DAE descriptions with hybrid phenomena. This leads to many interesting problems such as DAEs which change their index when a discrete submodel switches states. There are many interesting research problems stimulated by such models: some semantic issues related to declarative descriptions and reliable execution of simulation of the models are still open problems.

2.2 Model Libraries

Model libraries are collections of reusable model parts and subsystems in a modeling and simulation tool. In object-oriented modeling these parts will almost always coincide with physical model parts and subsystems. Model libraries can increase reusability and consequently productivity in systems modeling substantially. This has been one of the main incentives in the development of object-oriented model libraries during the last two

decades. In the following discussion we will restrict the subject to physics based models for engineering systems.

Classification

Model libraries in some form are part of all modern simulation tools. On the first look, they seem to follow the same ideas: model components can be dragged from a model palette or browser onto a diagram which is used to compose a complex system from simpler parts. In terms of their implementation and usefulness for different purposes, there can be tremendous differences. Most domain specific modeling tools do not show the details of the model implementation and a user has to rely on the completeness and correctness of the documentation. These *black box* models, even if they are based on sound physical models, can only serve some simulation purposes. However, other tasks such as model validation is much more difficult or even impossible. Internally, different models may be used for different ranges of parameters or operating conditions. This is for example the case for the SPICE [Massobrio and Antognietti, 1993], but at least the source code for the SPICE models is available even if sparse documentation may make it difficult to comprehend. Trusting black box models can be a hazard. K. J. Åström has described this in the following way:

> The practice of not publishing source code is a factor which strongly contributes to poor quality (of models). A good analogy is to ask the question: what would mathematics look like if theorems were published together with a manual how to use them and the proofs were hidden as company secrets?

The philosophy behind the Modelica language and other equation based modeling languages is orthogonal to black box models. Model users should be able to see all equations that comprise the model. They should also be allowed to modify and extend the model if necessary for them. *Open Code* libraries can be designed with different ideas about their use in mind. In some domains it is possible to provide a complete set of base models, e. g., in the Modelica Multi-Body-Systems (MBS) library. This library is open and extensible. It was designed to be complete for all common MBS. In process engineering and thermodynamic systems it is impossible to provide a library of similar completeness as the MBS library because of a large number of model assumptions and case-specific empirical correlations which lead to a huge number of models. Therefore, the ThermoFluid library has been designed from the start to provide base models and structure to help rapid model development. It is necessarily incomplete, but it offers hooks and standard-case placeholders for user written model extensions.

Three types of libraries can be distinguished:

Black Box model libraries This is the most widespread case for commercial domain specific libraries, e. g., in Adams [Adams, 2002], APROS [Juslin, 1995] and ITISIM [ITI GmbH, 2002].

Open Code model libraries The Modelica Standard Library is a typical open code library. The standard models are selected in a way that covers many common cases and defines *connectors* which are valid for all models in that domain.

User Extensible Libraries The idea of a user extensible base library has been explored with the ThermoFluid library. That the concept works well for a broad range of processes which are only related through their basic physical phenomena will be clear from the examples in Chapter 5. User extensible libraries have to be open code libraries.

The main topic of this thesis is user extensible model libraries. Modelica offers many language constructs that make Modelica particularly interesting for this type of library development.

Structuring, Interfaces and Decomposition

With the simple guideline that model structuring and interfaces correspond to the equivalent features of the real system, structuring of model libraries should be trivial. This is not quite true for several reasons. As soon as hierarchical decomposition is used, the number of levels and the interfaces between subsystems are debatable. Many levels may give a completely logical structure but may be cumbersome to browse and use. Too fine-grained decomposition can result in a large number of models with good code reuse but it is difficult to get an overview. There are many possible solutions how the obvious physical quantities in the interface of a real system e. g., the flange of a pipe, are mapped to a mathematical model. Structuring is more important when the size of the model libraries and the organization using them grows. For very large libraries, database-like search capabilities are necessary, otherwise modelers might prefer to rewrite a model instead of finding an existing one. This has been recognized as a common occurrence in large companies with many, often poorly documented models, see [Jeandel *et al.*, 1996].

Many solutions for structuring of model libraries exist and often the resulting structure reflects personal preferences of the library designer. The library structure is always a compromise. The developer should take the prospective users into consideration. If the users want to assemble systems from subsystems and get simulation results as quickly as possible, complete subsystems representing parts of the system which are as large as possible with few parameters will help them most efficiently.

Users which have to develop models by themselves and have to deal with rapidly changing prototypes under development benefit a lot from a much more fine-grained structure. They will also prefer a multi-layered approach which gives them access to all levels of model composition from basic physical phenomena to subsystems.

A detailed investigation of structuring of object-oriented libraries will be given in Chapter 6.

2.3 Validation and Verification

Models are developed with the purpose of drawing conclusions about the real system, either by simulating them with inputs which resemble the inputs of the real system or by analyzing them. Consequently, the validity of the model is a very important question. The model must accurately describe the important physical phenomena of the real system in order to be of any use. Validation of models is a complicated task. The fundamental problem is that no model can perfectly capture all aspects of the real system, abstraction and simplification are the very nature of modeling. Thus model validation is always relative to the requirements of the model use. It is important to recognize that it is impossible to conclusively validate a model. According to [Popper, 1935], a theory can only be *falsified* i. e. proven wrong by experimental tests. By passing any number of tests, a model is only as yet unfalsified. The art of model validation is thus to design validation test cases for the model that make sure the model holds sufficiently well for the interesting aspects of the real system. The part of model validation that deals with adaptation of parameters to make the model and the real system match is often called model calibration.

There are two important aspects to the validation of mathematical models in a computer implementation:

1. Does the model as it is coded in a modeling language correspond to the mathematical model that it is supposed to represent? This is called *verification* or *internal* validation.
2. Does the model correspond to the real system? This is called *validation*, sometimes with the additional attribute *external*.

For models obtained by system identification, validation of the model structure and parameters are part of the standard procedures in model development. For linear models the repertoire of systematic validation methods is rich and well established. The quantitative bounds that characterize the quality of the model give practitioners a good indication about how much they can trust a model. On the other hand, system identification models are inherently limited by being linear.

When validating models, there are two questions that have to be put:

- Is the chosen model structure adequate?
- Are the parameters used in the model optimal in some sense when the model output is compared to measurements of the real system.

In practical terms the questions can usually not be separated like above, usually parameters are optimized for different model structures and then the structures are compared with their best-fit parameters. It is obvious that the choice of the model structure sets limits to the achievable performance of the model, see [Eborn, 2001]. The most common criterion for selecting a model structure is Akaike's Information Criterion (AIC). It was developed for linear models but can also be applied to nonlinear models.

The practical problem with model validation for first principle physical models is that they are non-linear. The mathematics and stochastics of nonlinear models are much less developed than for linear systems. Parameter optimization for hybrid-DAE models results in a nonlinear mixed-integer optimization problem which may be very hard to solve. Therefore, model calibration for physical models usually involves hand-tuning of parameters based on trial and error or physical insight. Often model and system output are compared for typical input signals like step responses. Because of the current lack of systematic methods, this can be a very time-consuming activity, but for high quality models there is little chance to avoid it. The physical parameters of the real system are often not known accurately and may be impossible to measure directly. The model is calibrated by hand-tuning such parameters, sometimes with the help of optimization methods.

One of the few attempts that have been made to bridge this gap between system identification and physical modeling by developing methods and tools for *gray-box identification* i. e. estimation of parameters in physical models is MoCaVa and its predecessor IDKIT developed at KTH by Bohlin and coworkers, see [Bohlin, 1991; Bohlin, 1998].

A prototype of an integration of the modeling tool OMOLA and the grey-box identification tool IDKIT was used to select between different structures for a non-linear drum boiler model. [Åström and Bell, 2000] had developed a series of simple drum boiler models of second, third and forth order. The third and forth order models were compared in [Sørlie and Eborn, 1997; Sørlie and Eborn, 1998]. The results demonstrate that systematic model calibration can give very good agreements between model and real system and that the integration of a modeling and identification tool accelerates the work flow. Nonetheless this has not become industrial practice, mainly due to the lack of easy-to-use, integrated tools.

It is obvious from the definition of library models and validation that external validation of library models is not possible because they do not represent any particular real system. The *internal* validation on the other hand is very important because users of the library rely on the internal validation when they select models. In practical terms internal validation for library models means that the implemented model corresponds to its documentation[4]. Automated regression tests and cross checking numerical results with analytic ones where these exist for special cases are tools that can be used to obtain a high quality internal validation of library models.

2.4 Physics Based Model Reduction

The main purpose of dynamic models is to capture the dynamic behavior of a system. When the model is to be used for control, the dynamic accuracy of the model is important in the vicinity of the crossover frequency of the feedback loop and less important at other frequencies. When controllers with integral action are used, the closed loop gain is infinity at $\omega = 0$ and therefore the accuracy of the open loop gain at $\omega = 0$ is not important, neither are the properties at frequencies that are higher than the crossover frequency. As a motivating example, an investigation of a model reduction from the electrical domain will be presented as a general, but simple case of dynamic model reduction. Often the question arises whether it is possible to lump energy storage of two closely connected physical submodels into one unit or not. The Modelica language and in particular the Dymola simulation tool allow to design model libraries where the tedious part of such model reduction procedures is handled automatically be the tool when the user chooses model combinations that call for this simplification. The details of how this works in thermo-hydraulic cases will be presented in the following.

Example: An Electrical Circuit

The example consists of the simple electric circuit in Figure 2.7. The source voltage is the input to the system, the voltage over the capacitor C_2 is the output. The system can be transformed into a linear state space system with the voltages of the two capacitances as states. The

[4]Note that it is well possible for a model to pass an external validation with respect to a real system and to fail the internal validation.

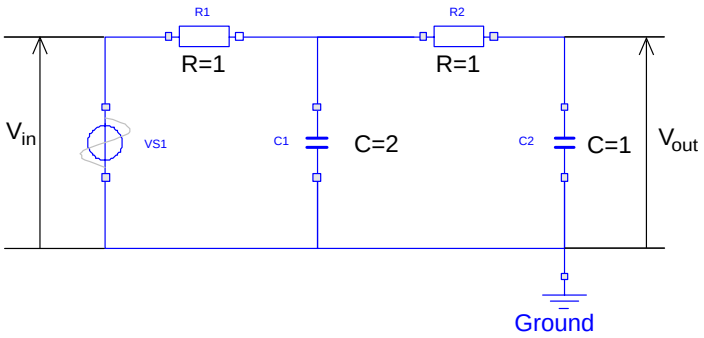


Figure 2.7 Simple electrical circuit with two capacitors.

linear system becomes:

$$\begin{aligned} \dot{x} &= \mathbf{A}x + \mathbf{B}u & x &= \begin{pmatrix} v_{C_1} \\ v_{C_2} \end{pmatrix} \\ y &= \mathbf{C}x \end{aligned}$$

$$\mathbf{A} = \begin{pmatrix} -\frac{R_1+R_2}{C_1C_2R_1} & \frac{1}{C_1R_2} \\ -\frac{1}{C_2R_2} & -\frac{1}{C_2R_2} \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} \frac{1}{C_1R_1} \\ 0 \end{pmatrix}, \qquad \mathbf{C} = (0 \quad 1)$$

with the transfer function

$$\frac{v_{out}}{v_{in}} = \frac{1}{C_1C_2R_1R_2s^2 + s(R_1(C_1+C_2)+C_2R_2)+1}.$$

The system stores energy in the two capacitors. The question of interest is now: under which circumstances is it possible to describe the system with one capacitor of the capacitance C_1+C_2 instead of two? A simple way to do the model reduction is to neglect the resistance and set $R_2 = 0$. The transfer function simplifies to

$$\frac{v_{out}}{v_{in}} = \frac{1}{sR_1(C_1+C_2)+1}.$$

This has the effect to lump the capacitors into one with the sum of the original capacitances. Observe that a second order linear ODE system where we simply set $R_2 = 0$ is singular. In the DAE-framework this is easy to treat. From $R_2 = 0$ we have that $v_{C1} = v_{C2}$, the *constraint equation*, compare Section 2.1, of the index 2 DAE. The index reduction can be handled automatically, resulting in a reduced order ODE system. The singularity

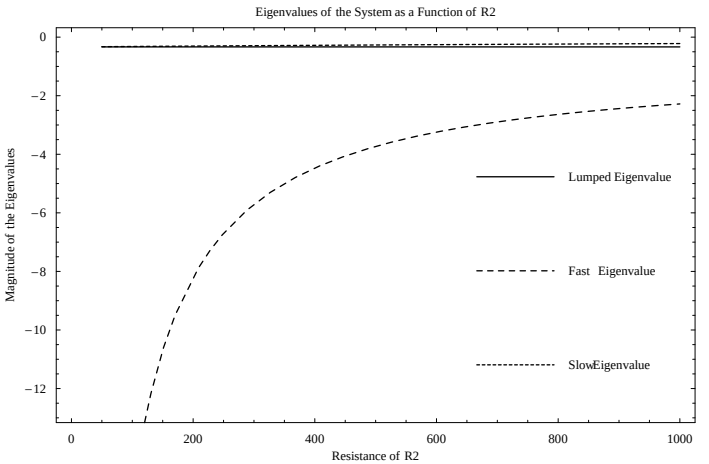


Figure 2.8 Eigenvalues of dynamics matrix **A** for the electrical system in Figure 2.7

problem of the higher order ODE formulation is avoided. This feature of the DAE formulation can also be exploited for model order reduction for a non-zero, but small R_2. For a choice of parameters $R_1 = 1000, C_1 = 0.002$ and $C_2 = 0.001$ we look at the eigenvalues of **A** (Figure 2.8) and the Bode plot (Figure 2.9) of the transfer function as R_2 decreases. As can be seen from Figure 2.8, one eigenvalue is around $1/3$ whereas the other goes to infinity as R_2 gets small. For comparison, the constant eigenvalue resulting from a simplified circuit with $R_2 = 0$ and and a lumped capacitor with $C = C_1 + C_2$ is also shown. From the look at the eigenvalues it seems to be possible to neglect the resistance R_2 for values of $R_2 < 500$. Observe that the eigenvalues are independent of the choice of inputs and outputs. An inspection of the bode plot in Figure 2.9 for the chosen input- and output signals reveals that the justification of the model reduction depends on the frequency content of the input signal. It is clear from the bode plot that the closed loop bandwidth is the limiting factor that determines for which values of R_2 it is possible to use the simplified system. It is clearly not advisable to neglect any of the energy stores C_1 or C_2 which are of the same order of magnitude. After this short excursion into the electrical domain, a similar situation for thermodynamic models is explored.

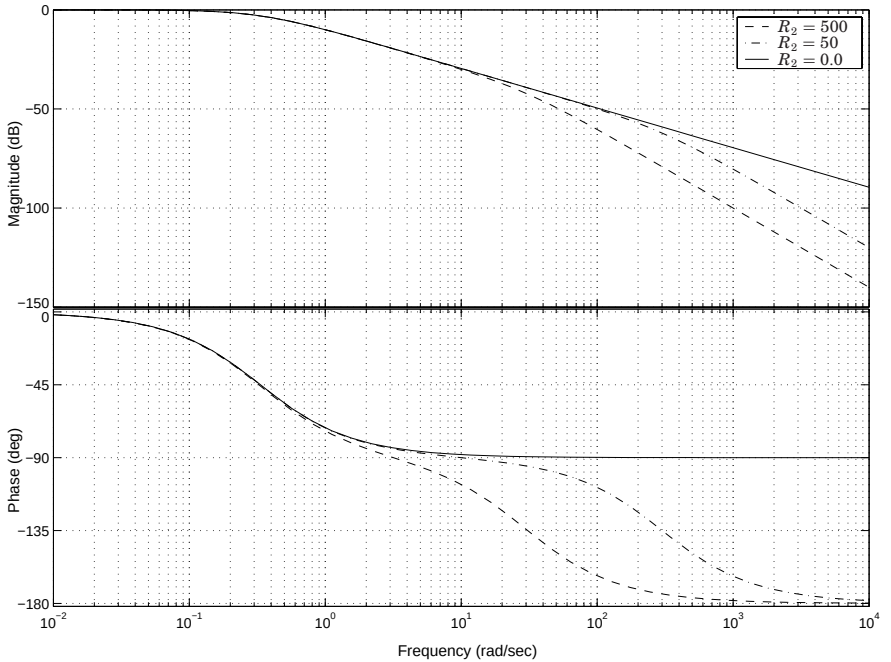


Figure 2.9 Bode Plots of the electrical system in Figure 2.7 for different values of R_2

Heat Exchangers

The type of model reduction presented in the previous example seems natural in that case. When the example is transferred to heat exchangers instead, the equivalent model reduction is not commonly used. Exceptions are special purpose low order models like the drum boiler model by [Åström and Bell, 2000]. This model reduction refers to an interesting case from Section 4.10 of combining a solid structure with a fluid when both the boundary layer heat resistance and the metal resistance are neglected. Instead of having separate energy balances for the wall and the fluid, these are combined into one energy store, equivalent to the lumping of the capacitors in the last example. Equating the temperatures $T_{fluid1} = T_m$ (see Figure 4.6) gives the wanted result. Again, an index two DAE system is the result where $T_{fluid1} = T_m$ is the constraint equation which is used to calculate the heat flow when differentiated. This means, similarly as in the electrical case, that the heat flow resistance is set to zero. Assuming for now that the fluid uses pressure p and specific enthalpy h as states

and that the thermodynamic equation of state is explicit in these states, the temperature difference can be expressed as

$$\frac{dT}{dt} = \left.\frac{\partial T}{\partial p}\right|_h \frac{dp}{dt} + \left.\frac{\partial T}{\partial h}\right|_p \frac{dh}{dt} \tag{2.15}$$

in the one phase region and as

$$\frac{dT_{sat}}{dt} = \frac{dT_{sat}}{dp}\frac{dp}{dt} \tag{2.16}$$

in the two phase region, where the temperature does not depend on the enthalpy. Usually the temperature is calculated via a function and not an equation. In current Modelica implementations, functions can not be differentiated unless a derivative function is written by the user and the annotation for function derivatives (see Section 3.4) is used to convey the derivative information to the simulator. If this is implemented in the model library, the model reduction is automatic whenever a user connects a fluid control volume and a resistance-free wall model without an explicit heat transfer law between them. The validity of doing this model reduction depends on the parameters of the actual physical system: in a condenser with very high heat transfer coefficients this assumption is usually valid for the condensing flow. The same holds for most walls in contact with boiling or condensing two phase flow. Focusing on the energy storage point of view this model reduction is equivalent to lumping two fluid control volumes into one volume. The lumping of small control volumes into a larger one is done routinely without questioning that simplification technique. With regard to the energy storage the example of lumping the solid and fluid energies is identical to the case of the capacitors.

Another case where this model reduction is particularly useful is modeling of thermal stresses during load changes in power plants. In that case the situation is slightly different: The metal mass has to be discretized in radial direction in order to model thermal stress. The innermost section of the metal in contact with the fluid is then modeled to have the same temperature as the fluid. The speed of possible load changes is limited by the thermal stress in the thick-walled metal mass. The thermal stress is proportional to the temperature difference between the innermost and outermost metal fiber.

In the case of the drum boiler derived in [Åström and Bell, 2000] the model reduction is not made by the simulation tool but by the model developer: this is an error-prone procedure which should be avoided.

While the automatic model reduction presented here is possible and desirable, it does not fully fit together with other aspects of library organization and is not the most efficient solution possible. In current Modelica

it is neither very convenient nor very efficient to provide derivative functions like the one for the temperature in all cases. Problems arise when one wants to organize the medium properties in records because derivatives have to be provided for all other record members, most of which are unneeded. When evaluating the derivative, the function for calculating temperature from pressure and enthalpy has to be re-evaluated. This is inefficient for steam tables where those functions are computationally expensive. Concerning the automatic generation of derivatives there is room for improvement both in the Modelica language and the Dymola tool.

2.5 Modeling Tools

Modeling tools provides means to bridge the gap between a conceptual model of a physical system and computer code that makes it much easier to communicate the conceptual model to other people. Modeling is an activity which requires knowledge of several domains: the application domain, numerical mathematics and computer science. Modeling tools can be designed to support model development, identification, selection, execution, visualization, calibration and validation. Some notable cases are noted below:

- Symbolic tools help in the model derivation and development, e. g., Mathematica [Wolfram, 1990] and Maple [Char *et al.*, 1992]
- Model databases and modeling tools built around model libraries help in model composition and selection of pre-defined models, e. g., Omola [Nilsson and Eborn, 1994], Dymola [Brück *et al.*, 2002] and MathModelica[5] [Fritzson *et al.*, 2002].
- Tools for model execution is an area that is covered by all simulation tools. All tools also offer some form of visualization of results. Matlab [MathWorks, 2001a] and Simulink [MathWorks, 2001b] are probably the most widely spread general purpose tools of this class.
- Calibration and validation tools. Tools for systematic comparison of results from physical model simulation with measurements and optimization of parameters are much less common. A promising approach is demonstrated with MoCaVa [Bohlin, 1998].
  gEST [PSEnterprise, 2002] offers estimation features for physical models and is to some extent coupled with the gPROMS [Barton and Pantelides, 1994] modeling environment.

[5] MathModelica is another simulation environment using the Modelica language. MathModelica is developed by the Swedish Company MathCore AB

- Expert systems and meta-modeling tools like [Gensym, 1992] or DoME [Honeywell, 2002] are designed to build modeling tools. They do not offer support for creation of all of the above classes of tools.

And, last but not least, many people would still name pen and paper as their most important modeling tools. Pen and paper aside, even with a clear conceptual model of a system, the modeling process contains many detailed steps that can be automated. Todays accelerated development process forces engineers to take as much advantage of that as possible. A seamless change from manual to automatic or computer-aided modeling is still in an early development stage. The practical problem is still that no single tool or suite of tools currently offers an integrated approach that supports all of the phases in model development and model evolution along the design arch, see Figure 1.1 equally well. MathModelica [Fritzson *et al.*, 2002] is an attempt to offer a broad range of capabilities and in particular extensibility.

One central task that current modeling tools are only beginning to take care of is the *complexity management* that is necessary to successfully cope with the growing complexity of engineering systems. Complexity is difficult to define, it may refer to completely different meanings in different contexts

- Large amounts of data that need to be collected and converted to the right format before it ends up as usable parameters of a model and result in *data handling complexity*.
- Team work of many people with heterogeneous backgrounds working in the same modeling project gives *organizational complexity*.
- *Mathematical complexity* with respect to analysis has nothing to do with the size of the system. Three coupled equations may exhibit very complex behavior which is extremely difficult to analyze.
- Large equation systems cause *computational complexity*. In spite of the continuous growth of computer resources, the growth rate of interesting problems keeps up the same pace.

Todays traditional modeling tools are not designed to handle all of these aspects of complexity at the same time. The lack of tool support for these tasks makes modeling an unnecessarily expensive endeavor. CAD (Computer Aided Design) tools are much more widespread in industry and almost all product data is available in some form in CAD-data. Currently there are no tools that could extract this data and bring them into a form that can be used by system modeling and simulation tools. This means that it is usually done by error-prone hand-transformation. The problem with this approach is the existence of too many different standards for

data exchange (at least one per engineering domain), most of which are either outdated by the rapid technological development, not supported any longer or not yet finalized. Hopefully there will be a consolidation in data formats in the future, until then partial solutions for specific models will have to do. Dymola can use some standard CAD files for visualization, but higher level data extraction like calculating volumes or areas which are parameters in the model is not supported. Similarly, many simulation tools allow a limited form of data exchange with other computer based design tools.

Ad-hoc approaches borrowed from related domains are often used to fill in missing tool features. Evolution of models over time and version management can be done with standard version control systems for software design like CVS (Concurrent Versioning System)[6]. CVS has been used successfully in the development of the ThermoFluid library and it is used in companies dealing with large model libraries, but yet there is no integration with any modeling tool.

Advanced mathematical analysis of models is only done in research and advanced development departments, but there the same model has to be coded again in different tools for each purpose. Even if part of the transformation is automatized, this process is cumbersome and modeling work needs to be repeated.

Compared to mainstream computer usage, systems modeling is an uncommon activity with a relatively small user base. Compared to other engineering computer tools like CAD, there is still a long way to go until modeling tools achieve the same level of integration into the design process as other computerized design automation tools.

[6]CVS is a freely available version control system with graphical clients for most computing platforms. It can be downloaded from http://www.cvshome.org.

3

The Modelica Language

Abstract

Modelica is a publicly available, object-oriented language for modeling of large, complex, and heterogeneous physical systems. It is maintained and improved by the Modelica Association. Models in Modelica are described by differential, algebraic and discrete equations which are mapped to a mathematical description form called hybrid DAE. This chapter illustrates the most important features of Modelica. Basic principles and those key properties of Modelica which are important for library design will be described.

3.1 Introduction

Languages should make it easy and natural to express ideas. Computer languages and mathematics have to express ideas unambiguously. Ideas in special disciplines are much easier to describe in a language that encodes these ideas as efficiently as possible. This is one of the reasons for the existence of so many different computer languages. Object-oriented modeling has many features which are similar to object-oriented programming, but also other features which are substantially different. A glossary of the special terms of object orientation with an emphasis on the differences between object-oriented modeling and programming is given in Appendix A. The first occurrence of a term defined in the glossary is marked with a triangle and typeset in ▷*slanted font*. Keywords of the Modelica language are typeset in **bold** on a gray background, other Modelica code uses the normal font on gray background.

Mathematical modeling of systems definitely is a discipline that gains by having a language that is adapted to its particular needs. It is a discipline with challenging and diverse problems which makes it hard to define a language that is both easy to use and powerful enough to cope with complex problems. In spite of the long experience with mod-

eling, object-oriented modeling techniques are still under development. This necessarily leads to an iterative design of languages and libraries. The design of Modelica took advantage of the experiences from a number of predecessor languages: Dymola [Elmqvist, 1978], Omola [Andersson, 1994], Smile [Jochum and Kloas, 1994], Object-Math [Viklund and Fritzson, 1995], NMF [Sahlin *et al.*, 1996], U.L.M. [Jeandel *et al.*, 1996] and SIDOPS+ [Breunese and Broenink, 1997]. The concurrent development of the Modelica language and Modelica libraries has fostered many improvements in the language. Designing a special purpose programming language with a variety of partly contradictory requirements for users which are usually not programming experts is not an easy task. The current version of Modelica, 2.0, [Modelica Association, 2002b] has come a long way in improving ▷*declarative* mathematical modeling from, e. g., FORTRAN77. In spite of its age and the fact that it was designed as a general purpose language, FORTRAN is probably still the computer language with the largest available code base for physical models.

The Modelica development, like all standardization efforts in areas with rapid technological progress, has to live with one major dilemma. Standards that companies are willing to invest in need to be *stable*, and should be backed by an accepted international standardization body. If there is evolution, the cost of adapting to the evolution has to be foreseeable. On the other hand, if the language does not follow the progress in its field, it will be obsolete before it has been fully established. This phenomenon has been observed often in computer science related standards. There are techniques which can help avoid this trap, a clear plan of future milestones is a good start.

It is a declared goal for the Modelica initiative to make model writing *simpler* for modelers. *Simpler* is not easy to define in an unambiguous way and many would prefer *cheaper* or *easier to maintain* etc., but it is easy to agree on the principle that a model language should be as close as possible to the natural form in which modeling knowledge is available: equations and diagrams in text books. Unfortunately, computer languages in general – and Modelica is no exception – have a long way to go to reach the flexibility of natural languages. The difference originates mainly in the implicit context knowledge that helps humans to disambiguate the meaning of natural languages. A special purpose declarative language like Modelica uses context knowledge in mathematics, numerical methods and special algorithms to allow tools to translate a declarative model into efficient, computer executable code. Even with a well designed modeling language, there are enough traps and pitfalls in simulation that can cause trouble through complete failure or very slow simulations. Modeling of physical systems covers such a broad range of applications and knowledge that it seems a hopeless endeavor to code that knowledge into the semantic rules

of a modeling language at the current state of the art. The alternative to implicit knowledge in the modeling language or simulation environment is to build the knowledge more explicitly into a model library. These two ways of representing the modeling knowledge are complementary and both are needed.

The following sections give a short overview over the main features of the Modelica language. They assume some familiarity with object-oriented concepts. Readers which have not seen Modelica before are recommended to consult an introductory text like the Modelica tutorial [Modelica Association, 2000b] or [Tiller, 2001].

3.2 Key Features of Modelica

Mathematical modeling of systems imposes two main requirements on modeling languages. The language has to be able to express the mathematics required to define system behavior and needs powerful structuring properties to cope with complex interconnected systems. Object-oriented modeling languages combine these two properties in a unique way. Object orientation is a well established principle of programming languages and most modern programming languages make use of its features. Equations as language elements only occur in dedicated modeling languages and in the languages of computer algebra systems, e. g., Mathematica [Wolfram, 1990]. The main property that makes equations attractive for modeling is that they represent the system behavior in a declarative way. A declarative representation of system behavior does not determine *how* something is calculated, instead it defines *what* it is. Declarative descriptions are therefore more flexible, but the compiler or tool that works with that knowledge representation needs the capability to derive a procedural description that can be executed on a computer.

Modelica's key features can be classified to belong to either of the categories *behavior definition* or *model structuring*:

- Language features to describe the mathematical behavior of systems.

  Equations are needed to express the mathematics of models in a declarative way. It is important to realize the difference between ▷*equations* and assignments. Assignments fix the computational causality, equations do not. Vectors and matrices can be part of the expressions in equations, with the usual mathematical restrictions for compatible dimensions and sizes. Modelica can handle ordinary differential and difference equations and differential algebraic systems. All types of equations can,

however, not be expressed in Modelica. Integral and partial differential equations are not yet part of the language.

Algorithms are not a declarative way to represent behavior, but sometimes this "escape mechanism" is the most efficient way to encode behavior. Discrete logic control algorithms are conveniently expressed as Modelica ▷*algorithms* and they can be identical to the one used in the real control hardware. In some cases a hand-written algorithm may be more efficient than the equations transformed by a tool compiler.

Functions are a language construct to ▷*encapsulate* an algorithm so that the algorithm can be reused anywhere. The input-output relation of a function is fixed and given in the function declaration. When functions are used inside equation systems, they can nonetheless be used to calculate an input from a given output.

External functions are only a minor feature in the language design, but the consequence of being able to reuse well tested legacy code and interface to a multitude of scientific software libraries is an important facet to ease the transition to equation based modeling.

- Language features for managing complexity, structuring models and promoting flexibility in a safe way.

  Models are the Modelica equivalent of ▷*classes* in object-oriented programming languages. They are the main structural units in system modeling. A model is the blueprint of a model ▷*instance* or ▷*component*.

  Strong typing is a successful technique to reduce risk for errors in programming languages. Strong typing is a means to define data types as precisely as possible. In physical modeling, this can for example be achieved by declaring a variable to be a temperature in Kelvin instead of a plain real variable.

  Hierarchical structuring by composing complex models from simpler parts is used in all systems-oriented modeling tools, not only in object-oriented ones. It is a fundamental requirement for organizing structured models. Interaction between elements on the same hierarchical level is achieved by ▷*connectors*. Connect statements are transformed into equations.

  Inheritance is a key feature for code reuse in object-oriented languages. Behavior shared between similar models is coded in a ▷*base class*. A specialized ▷*child class* ▷*inherits* this part of its definition from its base class.

Class parameters are a way of parameterizing a model by declaring some of its components as exchangeable against similar components. The replacement components have to fulfill compatibility conditions which are defined by the declaration of the original component.

Formal definitions are essential elements of computer languages. The formal definition of the Modelica language, [Modelica Association, 2002b], is publicly available on the Internet. It defines the syntax and semantics of Modelica models, but does not provide a reference implementation for simulation. Modelica is only a language: in order to run a simulation, the Modelica model has to be translated into executable code by a compiler and linked to a numerical integrator capable of handling differential algebraic equations with events. A Modelica compiler flattens the hierarchical structure of a Modelica model using equations from components and connect statements into a flattened equation system. Dymola from Dynasim AB is the Modelica compiler and simulation environment that was used to develop ThermoFluid. Another Modelica environment is MathModelica from MathCore AB.

The Modelica Association continues to improve the language and the design meetings are open to interested participants. They are announced on the web site http://www.Modelica.org.

3.3 Modelica Basics

Predefined Data Types

Modelica has five basic, ▷*built-in* data types. The predefined ▷*types* contain internal attributes to characterize them more precisely. The attributes use the names RealType, IntegerType, BooleanType, StringType and EnumType to refer to corresponding machine representations.

Real Real variables are the dominant data type in physical systems modeling. In Modelica, real variables have a number of attributes for distinguishing them.

- The RealType attribute *value* holds the value of the real variable. It is accessed without ▷*dot-notation*.
- The StringType *quantity* attribute specifies the physical quantity of the real variable: `Real L(quantity="Length")`
- The StringType *unit* attribute permits to specify the unit of real variables: `Real T(quantity="Temperature" unit="K")` . Us-

ing the additional attribute *displayUnit*, a different unit can be used for plotting.

- The RealType attributes *min* and *max* allow to set limits to the range of a variable.
- With the RealType *nominal* attribute, a modeler can define an order of magnitude of the variable for scaling purposes.
- The attributes BooleanType *fixed* and RealType *start* are used to specify initial values or initial guesses to variables.
- The BooleanType attribute *enable* has default true. It is defined for all classes. It is used to determine whether outputs of functions are computed or not.
- The parameter attribute *stateSelect* is of the predefined enumeration type StateSelect. It is used to guide the state selection mechanism in the index reduction, see Section 2.1.

Integer Integer variables have the same attributes as Reals except for *nominal*. The *value*, *start*, *min* and *max* attributes are of machine representation IntegerType.

Boolean Boolean has the attributes *quantity* (StringType), and *value*, *fixed* and *start* (BooleanType).

String String variables have the *value*, *start* and *quantity* attributes, all of them are StringType.

Enumeration Enumerations have been added to Modelica in version 2.0. They represent an ordered collection of named items. For every new enumeration type defined by `type E = enumeration(e1,e2, .. en)`, a conceptual simple type is defined containing an attribute **constant** `EnumType e1=...` for each of e1 ... en. The remaining parts of the enumeration type definition resemble the definition for Integer.

An array variable can be declared by appending dimensions after the class name or after the component name. It is also possible to declare an array type, e. g., for a transformation matrix.

```
Real[3] position, velocity; // array dimension follows class, Pascal−style
Real acceleration[3]; // array dimension follows variable, C−style
type Transformation = Real[3,3]; // this declares a 3 × 3 matrix type
Transformation myTransform; // this declares a 3 × 3 matrix
Real[3,2,10] table; // a three−dimensional array.
```

The extensive use of the additional attributes is a programming style element that contributes considerably to better readability and safety of model code. In Dymola's Modelica implementation, unit attributes are

checked for connectors, making it impossible to connect e. g., a pressure to a temperature. The *nominal* attribute is used for scaling and the *min* and *max* attributes are also checked against violation.

Structure of a Simple Model

The mathematical behavior of models is programmed using the basic types. Modelica programs are built from models, the Modelica synonym for classes in object-oriented programming. From a class definition, a Modelica compiler can create any number of objects called instances. The model is used as a blueprint by the compiler to create an ▷*object* which contains instances of the elements defined in the model. These elements can be built-in types or models, defining a hierarchical tree structure. The leaves of that tree structure are the five built-in Modelica types listed above. The data structure of Modelica models is similar to that of many object-oriented programming languages. The following listing demonstrates a simple model using only built-in data types.

```
model LimiterAndSwitch "an example of Modelica's data types"
  Real signal(start=0.0) "a variable named signal of built-in type Real";
  Real limited "a limited signal";
protected // the following declarations are protected
  Integer count(start=0) "an Integer variable counting switches to true";
  parameter Real ulimit=1.0 "upper limit: a parameter";
  parameter Real llimit=-1.0 "lower limit: another parameter";
  parameter String message="this model is trivial" "an example string";
public // The next declaration is public
  Boolean switch(start=true) "a Boolean variable initialized to true";
equation // this is the start of an equation section
  // the der-operator means time derivative: der(x)= dx/dt with t =time
  1/1.2*der(signal) = Modelica.Math.cos(time); //using a library function cos
  limited = if signal > ulimit then ulimit else if signal < llimit then llimit
    else signal;
  switch = if signal > ulimit then false else true;
algorithm // this is the start of an algorithm section
  if edge(switch) then
    count := count + 1; /* this is an assignment statement */
  end if;
end LimiterAndSwitch;
```

Listing 3.1 A model demonstrating Modelica's basic data types, equations and algorithms

The listing contains a number of Modelica elements which will be explained below.

There are three types of comments in the code: C-style comments until the end of the line as in `//comment 1`, enclosed comments as in

/*comment 2*/ and string-comments "comment 3" which are used in graphical user interfaces.

A Modelica model consists of two sections: a declaration section defining all data fields and an implementation section defining the behavior. The declarations can be either ▷*public* (the default) or ▷*protected*. Protected variables can only be used inside a model and in ▷*derived classes*, public elements can be accessed from an outer hierarchical level using dot-notation. The keywords **public** and **protected** are section headings. This means that they are valid for all following declarations until a new section heading.

The implementation section can be composed of either **equation** or **algorithm** sections. In Modelica 2.0 there are also **initial equation** and **initial algorithm** sections for defining the behavior at the start of a simulation. Equations are a declarative definition of the model behavior. Modelica equations can consist of arbitrary Modelica expressions on both sides of an = sign. The "flattened" equations from a Modelica model form a "hybrid DAE", i. e. a differential algebraic equation allowing some variables to have jumps or discontinuous derivatives at some places, as in the equation above for the variable limited. The semantics of hybrid models are treated in more detail later in this section.

Another way of distinguishing real variables is with respect to their variablity during the transient analysis of a model:

- Variables with the **discrete**-▷*prefix* can only change at discrete points in time, at so called *events*.
- The prefix **parameter** has two meanings. These are:
  - ▷*Parameters* are constant during continuous time integration. It is possible to compute the values of parameters during initialization or in optimization problems.
  - Parameters are picked up by graphical user interfaces, users influence the behavior of models mainly via parameters.
- A **constant** variable is similar to a parameter with the difference that constants can not be changed after they have been declared. They can be used to represent mathematical constants, for example **constant** Real PI=4*arctan(1);

The variability of Integer and Boolean variables is always discrete.

Other ▷*prefixes* in Modelica are **input** and **output**. They are used in situations where the computational causality is fixed, e. g., for function declarations and in models with known input-output behavior called blocks. The **flow**-prefix will be discussed later in this section.

Variability and other prefixes are very useful for applying semantic checks to the model. Errors caused by violating the semantic restrictions imposed by prefixes can be diagnosed with understandable error messages.

Continuous Time Models

As a prototype example of a continuous time system, let us take a look at a state space system:

```
block StateSpace "a simple state space system"
  parameter Real A[:,:] = {{0,−1},{1, −2}}; // A is initialized
  parameter Real B[size(A, 1), :]; // B is declared in C−style
  parameter Real[:, size(A, 2)] C; // C is declared in Pascal−style
  parameter Real D[size(C, 1), size(B, 2)];
  input      Real u[size(B , 2)];
  output     Real y[size(C , 1)];
protected
  Real x[size(A , 2)];
equation
  der(x) = A∗x + B∗u; // note that in these equations
  y      = C∗x + D∗u; // A∗x means matrix times vector etc.
end StateSpace;
```

This example demonstrates a number of continuous time modeling features of Modelica:

- Matrices are declared by using square braces after the variable name or after the type name. Dimensions are separated with commas. Arrays can have any number of dimensions.
- Curly braces are delimiters for array initialization. Array elements and dimensions are separated by commas.
- A colon in the size declaration means that the size is unspecified.
- The size-operator returns the size of an array variable. The argument `n` to `size(A,n)` denotes the dimension whose size is returned.
- The `*`, `+` and `-`-signs are overloaded for matrix operations. Matrix operations are only defined when the matrix sizes are compatible.
- The expression `der(X)` means derivative with respect to time of the variable X. If `der()` is applied to an array, it is applied to all elements of the array.

An important feature of equations is that they do not preclude the computational causality of the calculation. An equation `u = R*i` can be used to compute either `i` or `u` or `R`.

Apart from equations, Modelica offers functions for defining model behavior. Functions use assignments in algorithms instead of equations.

```
function LimitAbove "keeps input smoothly below an upper limit"
  input Real x "input";
  input Real limit "upper limit";
  input Real slimit "start limiting function value asymptotically";
  output Real xlim "limited output";
algorithm
  assert(startlimit < limit, "startlimit has to be smaller than final limit");
  if x < startlimit then
    xlim := x;
  else
    xlim := limit-(limit-slimit)*exp((1.0/(limit-slimit))*(stlimit-x));
  end if;
end LimitAbove;
```

The function illustrates some additional features of Modelica:

- All inputs and outputs to a function have to be declared with the respective prefixes.
- Algorithms are used to specify what a function computes. Inside algorithms, assignment statements, `:=`, have to be used.
- Modelica has `assert`-statements to check for illegal conditions. If the condition in an `assert`-statement evaluates to false, the message contained in the string-argument is printed.
- Modelica has the same control-flow statements as most programming languages: for-loops, do-loops and conditional execution. For details, see [Modelica Association, 2000a] and [Modelica Association, 2000b].

The function can be used as follows:

```
limited = LimitAbove(time,10,8);
```

The variable limited will rise linearly until it reaches the value 8 and then asymptotically approach 10.

Hybrid Models

Some hybrid modeling constructs have been used in the previous examples without explanation. Currently there are four ways to express hybrid phenomena in Modelica equations:

- Conditional expressions or if-expressions,
- conditional equations or if-clauses,

- conditional evaluation or when-clauses and
- discrete time operators

A natural way to define impulses in physical modeling would be to define Dirac impulses as a genuine language construct. This is currently discussed in the Modelica Association and tested with prototype implementations.

The simplest way to define discontinuous behavior is by using conditional expressions:

```
limited = if signal > ulimit then ulimit else
          if signal < llimit then llimit
          else signal;
```

Conditional expressions are a behavioral declaration of hybrid phenomena. They are equivalent to the following mathematical definition:

$$limited = \begin{cases} \text{ulimit} & \text{if signal} > \text{ulimit} \\ \text{signal} & \text{if llimit} \leq \text{signal} \leq \text{ulimit} \\ \text{llimit} & \text{if signal} < \text{llimit} \end{cases}$$

Conditional equations are another way to express hybrid behavior. All branches in the if-clause have to contain the same number of equations. For example, piece-wise linear systems can be expressed as:

```
model PieceWise "a piece wise linear system"
  parameter Integer n=2, m=2, p=2 "sizes of state, input and output vector";
  Real x[n], u[m], y[p] "state, input and output vector";
  ... // matrix-declarations for A1, B1, C1, A2, ,B2, C2 omitted
equation
  if x[1] > 0 then
    der(x) = A1*x + B1*u;
    y      = C1*x;
  else
    der(x) = A2*x + B2*u;
    y      = C2*x;
  end if;
end PieceWise;
```

Modelica also defines conditional equations which are only evaluated when a condition becomes true, i. e. when-clauses are not valid at all times as other equations. The `reinit`-operator can only be applied to dynamic states and is used to model impulsive changes. The `pre()`-operator can only be applied to discrete-time expression. The expression `pre(y)` returns the "left limit" $y(t^{pre})$ of variable $y(t)$ at a time instant t. Real

variables assigned inside when-clauses are discrete-time expressions, the same holds for variables of type Integer and Boolean. The bouncing ball example demonstrates when-clauses and two operators for discrete behavior:

```
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g = 9.81 "gravitational constant";
               Real h, v "height and velocity";
equation
  der(h) = v;
  der(v) = -g;
  when h <= 0 then
    reinit(v,-e*pre(v)); // when the ball touches the ground
  end when;              // the velocity is re-initialized
end BouncingBall
```

The when-clause is evaluated once each time the expression `h<=0` becomes true.

Hierarchical Structure

Modelica models can be much more complex than the simple examples above. Composition of complex, structured models is achieved via a component oriented architecture and a restricted type of class called **connector**. In Modelica, connection points between subsystems are abstracted to have no extension. This allows to unify Kirchhoff's current and voltage law,

$$\sum_i I = 0 \qquad V_i = V_j$$

to a generalized network approach. The same rules apply to potential type variables like voltage and flow-type variables like current in all engineering domains. In mechanics, forces sum to zero and positions are equal, the mass- and energy flows in fluid flow sum to zero and pressures are equal at connection points. Variables of flow type are marked with the **flow**-prefix.

Model composition and reuse will be illustrated with simple mechanical models similar to the ones in the Translational library, a part of the Modelica Standard Library. In Modelica, libraries are called ▷*packages*. Packages can get access to the definitions of other packages using the **import** statement. In the example, we will use the Modelica.SIunits package which provides predefined types for physical modeling. The first model that should be defined for every physical library is the connector.

```
package OneDExample "1-dimensional translational mechanical components"
```

```
import Modelica.SIunits; // makes predefined types known
  connector Flange "1D translational flange"
    SIunits.Position s "absolute position of flange";
    flow SIunits.Force f "cut force directed into flange";
  end Flange;
    ...
end OneDExample;
```

The next step for designing library models is to find some practical abstractions for reusable base models. Two models characterizing translational mechanical components are the following `Rigid` and `Compliant` models.

```
partial model Rigid "Rigid connection of two translational 1D flanges"
  SIunits.Position s "center of component (s=flange_a.s+L/2=flange_b.s-L/2)";
  parameter SIunits.Length L=0 "length of component between flanges";
  Flange flange_a, Flange_b
equation
  flange_a.s = s - L/2;
  flange_b.s = s + L/2;
end Rigid;
partial model Compliant "Compliant connection of 2 translational 1D flanges"
  Flange flange_a, flange_b;
  SIunits.Distance s_rel "relative distance (= flange_b.s - flange_a.s)";
  flow SIunits.Force f "force, positive in direction of flange axis R";
equation
  s_rel = flange_b.s - flange_a.s;
  flange_b.f = f;
  flange_a.f = -f;
end Compliant;
```

Note the keyword **`partial`**. It indicates that these models are not complete. Additional behavior has to be added before a well-defined model is obtained. The definitions of these base classes is reused in derived classes, for example a model for a sliding mass. Inheritance of all features of the base class `Rigid` is achieved using the **`extends`** keyword.

```
model SlidingMass "Sliding mass with inertia"
  extends Rigid;
  parameter SIunits.Mass m=1 "mass of the sliding mass";
  SIunits.Velocity v "absolute velocity of component";
  SIunits.Acceleration a "absolute acceleration of component";
equation
  v = der(s);
  a = der(v);
  m*a = flange_a.f + flange_b.f; // known since Newton's times
end SlidingMass;
```

Similarly, models for a spring and a damper are defined. In order to get a system that has dynamics, a periodic external force is added. The schematic for the complete system is shown in Figure 3.1. The signal block and the force in Figure 3.1 use a connector type for signals. The force model acts as an interface between a signal-based system part and the physical part using the mechanical flange for connections. The Modelica

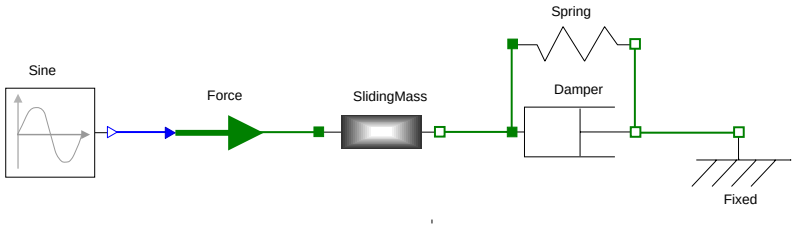


Figure 3.1 Schematic of a simple translational mechanical system.

code for the system is very simple, consisting of the component declarations and the connect-statements. It also shows how to refer to library models and uses ▷*modifications* to models to assign values to parameters and set initial conditions.

```
model simpleSystem "a simple mechanical system"
  SlidingMass slidingMass(
    L=1, m=1.23,     // modification to the original model parameters
    s(start=-0.5),   // the inital position is now -0.5
    v(start=0.0));   // the inital velocity is set to 0.0
  Spring spring(s_rel0=1, c=10000);
  Damper damper{d=10.0};
  Fixed fixed(s0=1.0);
  Force force;
  Modelica.Blocks.Sources.Sine sine(freqHz={15.9155}); // a library model
equation
  connect(sine.outPort,force.inPort);
  connect(force.flange_b,slidingMass.flange_a);
  connect(slidingMass.flange_b,spring.flange_a);
  connect(spring.flange_a,damper.flange_a);
  connect(spring.flange_b,damper.flange_b);
  connect(spring.flange_b,fixed.flange_b);
end simpleSystem;
```

The **connect**-statements are transformed into equations, taking the **flow**-prefix into account. The zero-sum rule for flow variables has to take all flows at a given connection point into account. In the spring-damper case

the generated equations from the connection between the sliding mass, the spring and the damper become:

$$\begin{aligned} &\text{slidingMass.flange\_a.f} + \text{spring.flange\_a.f} + \text{damper.flange\_a.f} = 0 \\ &\text{slidingMass.flange\_a.s} = \text{spring.flange\_a.s} \\ &\text{slidingMass.flange\_a.s} = \text{damper.flange\_a.s} \end{aligned}$$

The forces at the connection sum to zero, the positions are equal.

Modelica Class Parameters

A key feature that makes Modelica a flexible modeling language while maintaining safety from modeling errors are class parameters. Class parameters make it possible to parameterize models in a high-level fashion, exchanging submodels or even whole subsystems for a type-compatible replacement. Type compatibility ensures as far as possible that the replacement model is adequate and works correctly. The Modelica type system is based on concepts for subtyping in object-oriented general purpose programming languages introduced in [Abadi and Cardelli, 1996]. In that reference, a calculus for object-oriented programming languages is developed and used to prove theoretical properties of computer languages. Some of the proofs concern the soundness of high level programming concepts like class parameters. Due to the differences in the semantic definitions of object-oriented computer programs and Modelica models, no final conclusions can be drawn from the soundness of class parameters in object-oriented programming (OOP) and in object-oriented modeling (OOM).

- The semantics of a Modelica model are defined by a mixed system of differential-algebraic and discrete equations, a “hybrid DAE” as defined in [Modelica Association, 2000a].
- The semantics of a computer program is defined by communicating objects executing methods operating on data defined inside the objects.

In spite of the fact that the formal methods presented in [Abadi and Cardelli, 1996] apply only partly to a modeling language, the adaption of strong typing using a proven type system offers clear advantages.

Simplifying, one can claim that the *data structures* represented by an object-oriented model and an object-oriented program are very similar. Actions, operations and the time evolution of their states are, however, different. The type safety offered in OOM by adapting a type system designed for OOP thus only considers the data structure of the model and not its equations. This becomes clear when looking at the definition

of ▷*type compatibility* in [Modelica Association, 2000a], quoted in Appendix D. In simple words one can say that the replacement model must contain at least the same data elements as the original model, but it may also contain additional elements. This has to hold for the complete tree of a hierarchical data structure. There are no requirements regarding the equations.

EXAMPLE 1—STIRRED TANK

Consider the following example: A modeler uses a stirred tank model from a commercial model library in a system model of a chemical plant, but after the first trial runs he realizes that the library model omits physical phenomena which are important in the particular case. He declares the tank component to be ▷*replaceable* and builds his own model instead. The effects of the different models have to be investigated with a simulation of the whole system. According to Modelica semantics, the model can only be redeclared to be of his newly developed class if the class of the new tank is a *subtype* of the existing one. Because equations can not be removed from a model once they are inherited, it is very likely that the new model can not be constructed by inheriting from the existing one. But the modeler can build a type-compatible model (hopefully with heavy reuse of library models) to obtain a *type-compatible* replacement tank model. Usage of ▷*redeclaration* makes it straightforward to compare the results of the two models in the system context.

The replaceable tank can be defined and used as follows:

```
model StirredTank
  parameter Real A = 1.0 "a parameter";
  ... // rest of the model omitted
end StirredTank;
replaceable StirredTank stirredTank(A=2.0) extends SimpleTank;
```

replaceable indicates that the submodel can be exchanged, StirredTank is the default class of the component stirredTank and the statement **extends** SimpleTank defines the *constraining* class. All legal replacements to the component stirredTank have to be type-compatible to the class SimpleTank. The redeclaration in a system model can look like this:

```
model BigSystem
  ...
  ReactionSubSystem subSys(redeclare AdvancedTank stirredTank);
end BigSystem;
```

The redeclaration changes the original class of component stirredTank to AdvancedTank. This is done in a modification to a subsystem which is a component of a larger system. Note that the modification (A=2.0) of the parameter A from the original declaration is retained. □

The concept of type-compatible replaceable models has been used extensively in the ThermoFluid library. In most cases the data structures of a set of type-compatible models are identical, but the equations are different. This covers the very common case of constitutive equations. The ThermoFluid library has been a test-bed for the development of high level parameters in Modelica.

3.4 Annotations and Pragmas

The Modelica language has a particular construction to store information which is not associated with the model semantics of a hybrid differential algebraic equation. The keyword **annotation** is used for describing the graphical representation and other additional information about the model. Some of the ▷*annotations* can be regarded as ▷*pragmas*: hints to Modelica tools which can be used to improve the numerical efficiency.

Graphical Annotations

An important feature for the user-friendliness of object-oriented modeling is a representation for graphical model composition. It is highly desirable that the on-screen representation of a system resembles the customary engineering schematics as closely as possible. In a standardized language, the graphical appearance of models should be transportable between different tools. Modelica has chosen a particular way of using annotations for this purpose. The structure of annotations is defined in the grammar and the basic parts of graphical annotations are standardized. Tools may define additional annotations for fancier graphics, e. g., animations, which are silently ignored by tools which do not understand them.

The one-dimensional mechanical system model in Figure 3.1 looks on screen as in that figure. The annotations to display the graphics have been omitted from the example code. For hierarchical models, the graphical representation has to store the following information:

- An iconic picture of the model to be used as a placeholder for the model as part in a larger system.
- The connectors of the model have to be visible in that representation. They are used to graphically connect the model to other models.
- The model can be composed from several other models. The diagram of the graphical composition has to be stored as well.

The usual way to get an overview over a system would be to take a look at the composition or diagram-layer. Deeper levels of the hierarchy are browsed by opening the lower level models.

The two Modelica tools which are currently available, Dymola and MathModelica, use different graphical model editors. Models can be exchanged between the tools without problems and the graphical information is retained.

Parameterization Annotations

The high level parameters described in Section 3.3 make models very flexible. This flexibility needs to be supported by a suitable graphical equivalent that makes it available to model users graphically. For a user that does not know the internals of a library, it may be a substantial effort to find all models which are type-compatible to a given replaceable model. Not all users want to remember and use the Modelica syntax. A library developer knows all compatible models in the library and can use the following annotation to make all choices known to a tool:

```
model FlexibleSystem
replaceable Spring spring(c=10e5) extends Compliant
  annotation(choices(choice(redeclare NLSpring spring(c=12e5,c2=42)
                                    "a nonlinear spring"),
                     choice(redeclare SpringDamper spring(c=10e5,d=10)
                                    "spring damper combination")));
end FlexibleSystem;
```

This **annotation** assumes the existence of two model classes NLSpring and SpringDamper which have to be type-compatible to the Compliant model from Section 3.3. A graphical user interface can pick up these choices and provide intuitive means for switching between the models.

Derivative Annotations

Modelica has been designed to permit the symbolic manipulation of the model equations. Functions, in particular external functions, are more difficult to handle symbolically than equations. Derivatives of Modelica functions are useful for several purposes, e. g., to calculate symbolic Jacobians in non-linear equation systems and for the index reduction mechanism. An annotation is used to make Modelica tools aware of the existence of analytic derivatives for a Modelica function. Here is a trivial example:

```
function rad2deg "conversion function between radians and degree"
  input Real rad;
  output Real deg;
algorithm
  deg := 180.0*rad/Modelica.Constants.PI;
  annotation (derivative=rad2deg_der); /* the derivative function is called
    rad2deg_der. The name must be found in the same scope as rad2deg. */
end rad2deg;
```

The only way of getting the derivative of the function without knowing it analytically is to take numerical derivatives. Numerical derivatives are less accurate than symbolic ones and it can be difficult to guess the right perturbation to the input. In that sense the derivative annotations are like compiler pragmas, they give hints for better performance and accuracy.

This chapter has given a brief overview over the most important Modelica features. A far more complete overview with many examples is the book by Tiller (2001). The most recent version of the Modelica Specification, currently version 2.0 [Modelica Association, 2002b], is the authoritative document concerning the language definition. It is available for download on the Internet at http://www.modelica.org.

4

Physical Models for Thermo-Hydraulics

Abstract

Modelica is a language designed for physical modeling from first principles. In this chapter the fundamentals of thermodynamic and hydraulic models are developed from the laws of conservation of the extensive quantities mass, momentum and energy. Thermodynamics offers many different possibilities of expressing these conservation laws in terms of intensive or extensive variables. Properties and reasons for the choice of these secondary variables are mainly governed by numerical considerations and the availability of thermodynamic property relations. The availability of computationally efficient physical property functions with an appropriate complexity for the problem at hand is in most practical cases a factor that has the strongest influence on the choice of the model. Models for valves, turbines, pumps and solid structures are briefly outlined. Finally, a special class of models called moving boundary models is developed at the end of the chapter.

4.1 Introduction

The conservation equations can be written in two forms: the differential and integral forms of the general conservation equations. The details of the working form of these equations differ considerably between engineering domains, in spite of being based on the same principles. The conservation equations can either be written in *intensive* variables or *extensive* variables. Extensive properties depend on the amount of matter, like mass or energy, intensive properties are defined by the ratio of extensive properties that do not depend on the amount of mass, e. g., the density ρ. The different purposes of the models in e. g., fluid dynamics and

process engineering lead to different simplifying assumptions and different numerical methods such that the final models have little in common on first sight. The problem with the project of trying to design a physical base library which is applicable to a range of engineering domains is to find a unifying framework which "keeps everybody happy". One possibility to do this is to be able to postpone simplifying assumptions in a way permitting the user to select them when the model is used. The modeling software has to be able to generate efficient code from the more general models. The ThermoFluid library currently covers a substantial subset of the necessary physical phenomena of thermo-fluid systems. The goal was to focus on those model assumptions that are important for the design and analysis of system dynamics. Even then there is a necessity for more or less detailed models. Three fundamental distinctive features of models need attention:

Temporal resolution The range of timescales covered by the continuous dynamics of a single model has to be restricted to a reasonable range for the problem of interest.

Spatial resolution Lumped or zero-dimensional models are common for system modeling, but certain applications make it necessary to increase the spatial resolution to 1, 2 or even 3 dimensions. A restriction to low spatial resolution is called for to reduce the computational cost and it usually helps the understanding of the system to reduce complexity.

Physical resolution In systems models, not all of the physics in a given process need to be modeled for all purposes. If a pressure controller is to be designed, it may be unnecessary complex to consider reactions or changes in composition. *Replaceable Objects* in Modelica make it possible to disregard that part of the dynamics which is not of interest in a certain context.

These features together with the constant desire to make modeling more efficient through reuse leads to a very important quality of a library model: the model should be *polymorphic*[1] with respect to the required spatial and temporal resolutions of its problem domain.

The distinctive feature of the "finite volume method" that is used later on for solving these equations is the formal integration of the fluid flow equations over the finite volumes of the solution domain. The derivation of the equations is done as follows: the governing partial differential equations are developed in the differential form which is then integrated over a finite control volume with fixed or varying size, depending on the purpose of the model.

[1] see definition in Appendix A

All fluid properties are functions of time and space, but to avoid too cumbersome a notation, the explicit dependence on time and space coordinates is omitted and instead of $\rho(x,t)$, the shorthand notation ρ is used.

4.2 Fluid Transport Equations

The conservation of the extensive quantities mass M, energy E and momentum $\boldsymbol{I}$ in a problem domain with given volume V is the mathematical basis for modeling of single phase fluids. For mixtures of single phase fluids, the mass M is replaced by the vector of component masses $\mathbf{M}$. In non-equilibrium multi phase fluids, which will not be considered here, the conservation laws have to be expressed for each phase separately with exchange terms between the phases. In order to justify the assumption of homogeneous fluid properties, the thermodynamic state of a fluid particle will be described by the infinitesimal quantities of volume dV, mass dM, momentum $d\boldsymbol{I}$ and energy dE. It is often more appropriate to use specific variables, representing a quantity $d\Psi$ per unit mass

$$\psi = \frac{d\Psi}{dM}$$

The specific values of volume, momentum and energy are therefore

$$v = \frac{dV}{dM} \qquad \boldsymbol{w} = \frac{d\boldsymbol{I}}{dM} \qquad e = \frac{dE}{dM}$$

in which $\boldsymbol{w}$ is the velocity of the fluid. The reciprocal value of the specific volume is the density

$$\rho = \frac{1}{v} = \frac{dM}{dV}$$

The energy dE may be split up into internal and kinetic energy

$$dE = dU + dE_{kin}$$

which gives two more specific values

$$u = \frac{dU}{dM} \qquad e_{kin} = \frac{|\boldsymbol{w}|^2}{2}$$

where u quantifies the mechanical energy of the molecules in a motionless fluid. The overall specific energy becomes

$$e = u + e_{kin}$$

The ThermoFluid library is designed from the beginning with system simulation in mind and therefore restricted to a one dimensional discretization of the flow field or lumped parameter models which are a reasonable approximation for most technical systems with internal flow. Initially an arbitrary shape of the control volume will be assumed, but because of the assumption of one dimensional flow the most general form of a discretized one-dimensional flow channel will be used in the next step, deriving the discretized variables. The shape, shown in Figure 4.1, serves as an approximation to a slice of infinitesimal length of any flow channel which can be approximated by a one-dimensional flow field.

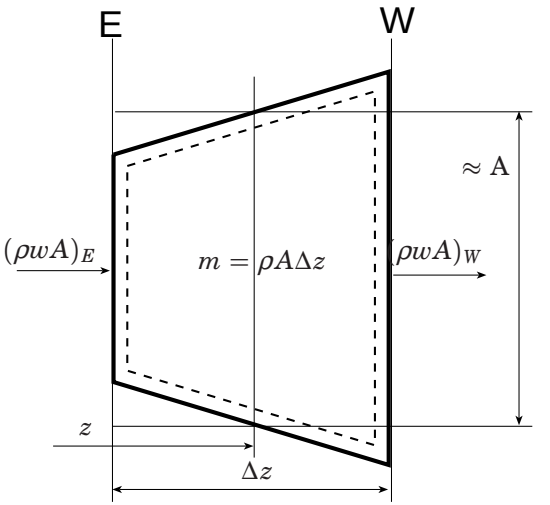


Figure 4.1 Geometry of a one-dimensional flow channel

One important point in the procedure of deriving the momentum and energy balance is that these two have to be consistent: the work done on the fluid by a force that is taken into account in the momentum balance has to be included in the energy balance. If this is not done, transformations of the equations to other working forms, e. g., to an entropy balance, will contain non-physical entropy production terms. If no such transformation is done it can be justified to neglect certain terms. It is a common approach to neglect the work of the gravitational field in the energy balance. Here, a general dilemma in modeling becomes obvious: neglecting a term will always reduce the range of validity of the model somehow, but often the order of magnitude of such a term compared to the important ones is so small, that the size of the neglected term is much smaller than

the remaining uncertainties in the model. Fortunately the symbolic processing in Modelica allows decisions to remain undecided until the model is used in a concrete application. A boolean variable for inclusion of a term will remove the term if it is not needed and thus cause no burden during simulation.

4.3 Balance Equations

Integration over the infinitesimal values $d\Psi$ of a control volume yields the overall, arbitrary quantity Ψ

$$\Psi = \int d\Psi = \int_M \psi dM = \int_V \rho\psi dV$$

The rate of change of a quantity is written as

$$\frac{d\Psi}{dt} = \frac{d}{dt}\int_V \rho\psi \, dV$$

Application of the Leibnitz rule [Hetsroni, 1982] yields the general transport theorem:

$$\frac{d\Psi}{dt} = \int_V \frac{\partial(\rho\psi)}{\partial t} \, dV + \int_A \rho\psi \boldsymbol{w}_A \boldsymbol{n} \, dA \tag{4.1}$$

The first term on the right hand side represents the rate of change of a quantity Ψ for a control volume keeping its shape; the derivative operator can thus be put under the integral. The second term accounts for the rate of change of Ψ due to a displacement of the volume's surface A. Therein, $\boldsymbol{w}_A$ denotes the local velocity of surface displacement, while $\boldsymbol{n}$ is the unit normal vector of the surface (outward direction is positive). The scalar product $\boldsymbol{w}_A\boldsymbol{n}$ yields the component of $\boldsymbol{w}_A$ normal to the surface, see Figure 4.2.

The influence of the second term becomes obvious when inserting $\psi = v$, which gives the rate of change of volume

$$\frac{dV}{dt} = \int_A \boldsymbol{w}_A \boldsymbol{n} \, dA$$

The Leibnitz rule serves to switch between different approaches of balancing: In an Eulerian approach, the control volume is considered as fixed, $\boldsymbol{w}_A = 0$, and the second term in (4.1) disappears. In a Lagrangian approach, the surface velocity equals the velocity of the fluid particles on

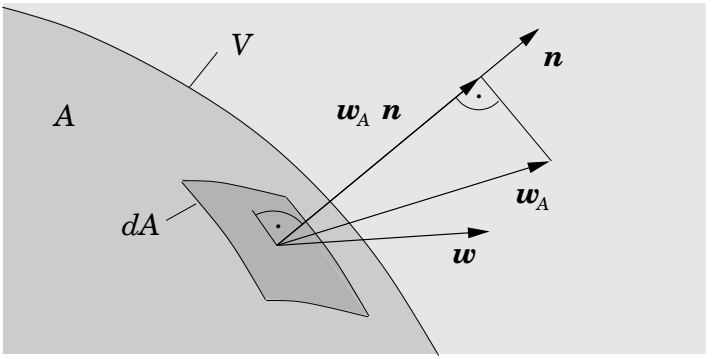


Figure 4.2 Velocities on a surface element. Nomenclature: $\mathbf{w_A}$: velocity of control volume boundary, **w**: fluid velocity, **n**: normal vector, dA: infinitesimal element of boundary surface.

the surface, $w_A = w$; in that case no particle enters or leaves the control volume, which therefore contains permanently the same particles. Besides these two approaches, w_A may be defined in any way that makes sense for the definition of a control volume.

The Leibnitz rule will now be applied to the quantities mass, momentum and energy, leading to the basic balance equations.

Mass Balance

For $\psi = 1$ the general transport theorem rule (4.1) yields the mass balance

$$\frac{dM}{dt} = \int_V \frac{\partial \rho}{\partial t}\, dV + \int_A \rho w_A n\, dA \tag{4.2}$$

In a Lagrangian approach, $w_A = w$, the control volume contains a constant mass, thus

$$\int_V \frac{\partial \rho}{\partial t}\, dV + \int_A \rho w n\, dA = 0$$

which essentially describes the conservation of mass and is also known as the continuity equation. Solving this equation for the first term and inserting it into (4.2) gives

$$\frac{dM}{dt} = \int_A \rho(w_A - w) n\, dA \tag{4.3}$$

Since no mass is created or destroyed, the term on the right hand side represents the flow of mass through the surface of the control volume, i.e.

the mass flow rate

$$\dot{m} := \int_A \rho(\boldsymbol{w}_A - \boldsymbol{w})\boldsymbol{n}\, dA$$

which, in this definition, is positive for a flow of mass into the control volume. For simple geometries with n fixed boundaries and flow perpendicular to the surface this evaluates simply to the sum of the mass flows with the above sign rule:

$$\frac{dM}{dt} = \sum_i^n \dot{m}_i \tag{4.4}$$

For multi-component fluids, the same derivation can be repeated for the vector of component masses. The result for non-reacting systems is the equivalent, but using vectors of masses and mass flows instead of scalars. When chemical reactions occur, the species masses are not conserved any more, only the total mass. The mass conservation equation can still be written in a similar form, but now includes a reaction source term:

$$\frac{d\mathbf{M_x}}{dt} = \sum_i^n \dot{\mathbf{m}}_i + \mathbf{rM} \tag{4.5}$$

M_x is the vector of component masses, **m** the vector of component mass flows and **rM** vector of mass source terms. In process engineering it is very common to use mole based units for mass and energy conservation instead of mass based ones because this form links more naturally to the stoichiometry of the reactions, see Section 4.9.

Often the mass balance is written in terms of the density. Taking into account the geometry of a flow channel as in Figure 4.1 which is symmetrical around its axis where the two areas at E and W are perpendicular to the flow direction, the integrals can be evaluated after changing the order of integration and differentiation. Using:

$$\frac{\partial}{\partial t}\int_V \rho\, dV = \frac{\partial}{\partial t}(\rho A \Delta z) \text{ and}$$

$$\int_A \rho \boldsymbol{w}_A \boldsymbol{n}\, dA = (\rho w A)_E - (\rho w A)_W = \frac{\partial}{\partial z}(\rho w A)\Delta z,$$

and dividing by Δz the result is:

$$\frac{\partial}{\partial t}(\rho A) = \frac{\partial}{\partial z}(\rho w A) \tag{4.6}$$

The vector quantities used in the general case are now replaced by scalars because of the restriction to one-dimensional flow.

Momentum Balance

The general transport theorem, (4.1), evaluated for $\psi = \boldsymbol{w}$, yields the momentum balance

$$\frac{d\boldsymbol{I}}{dt} = \int_V \frac{\partial(\rho \boldsymbol{w})}{\partial t}\, dV + \int_A \rho \boldsymbol{w}(\boldsymbol{w}_A \boldsymbol{n})\, dA \tag{4.7}$$

According to Newton's second law, the momentum of body with constant mass ($\boldsymbol{w}_A = \boldsymbol{w}$) increases due to the sum of the applied forces

$$\int_V \frac{\partial(\rho \boldsymbol{w})}{\partial t}\, dV + \int_A \rho \boldsymbol{w}(\boldsymbol{w}\boldsymbol{n})\, dA = \sum \boldsymbol{F} \tag{4.8}$$

It is common to distinguish between body forces and surface forces. If gravity is the only body force taken into account, the force on a mass dM becomes $d\boldsymbol{F}_g = \boldsymbol{g} dM$, where $\boldsymbol{g}$ is the constant vector of the acceleration due to gravity. Integration results in the overall gravity force

$$\boldsymbol{F}_g = \int_V \rho \boldsymbol{g}\, dV = M\boldsymbol{g} \tag{4.9}$$

The surface forces are usually split up into the pressure force and the friction force. The pressure force acts opposite to the unit normal vector, $d\boldsymbol{F}_p = -p\boldsymbol{n}dA$, causing an overall force on the surface of an amount

$$\boldsymbol{F}_p = -\int_A p\boldsymbol{n}\, dA \tag{4.10}$$

The friction force of an infinitesimal small element, caused by viscous and turbulent forces, is expressed by use of a stress tensor

$$\mathsf{T} = \begin{pmatrix} \tau_{11} & \tau_{21} & \tau_{31} \\ \tau_{12} & \tau_{22} & \tau_{32} \\ \tau_{13} & \tau_{23} & \tau_{33} \end{pmatrix}$$

where $1, 2, 3$ are the Cartesian coordinates and τ_{ji} denotes the shear stress in direction of i on a surface $j = \text{const}$. Multiplication with the unit normal vector $\boldsymbol{n}$ yields the stress force vector on a surface element, $d\boldsymbol{F}_F = \mathsf{T}\boldsymbol{n}dA$. Integration gives

$$\boldsymbol{F}_F = \int_A \mathsf{T}\boldsymbol{n}\, dA = \int_A \sum_{i,j} \tau_{ji}(\boldsymbol{n}\boldsymbol{e}_j)\boldsymbol{e}_i\, dA \tag{4.11}$$

where e_i is the unit vector in direction of i. Pressure and friction forces are also present inside the volume, but cancel themselves out and thus have no influence on the momentum of the control volume as a whole. Solving (4.8) for the first term and inserting it into (4.7) yields

$$\frac{d\boldsymbol{I}}{dt} = \int_A \rho \boldsymbol{w}(\boldsymbol{w}_A - \boldsymbol{w})\boldsymbol{n}\, dA + \boldsymbol{F}_g + \boldsymbol{F}_p + \boldsymbol{F}_F \tag{4.12}$$

The first term on the right hand side accounts for the convective transport

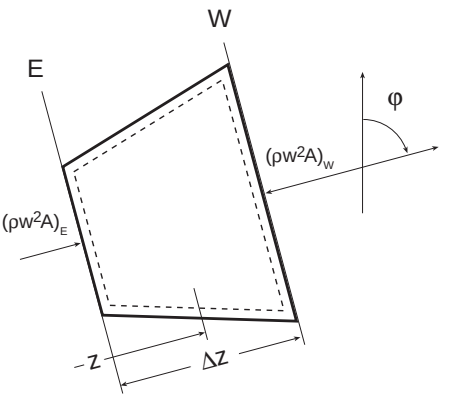


Figure 4.3 One-dimensional flow channel and pressure forces

of momentum. Taking a closer look at the integrals again for the simple, one-dimensional flow channel with constant volume in Figure 4.3, the following simplifications can be made:

$$\begin{aligned}
\frac{\partial}{\partial t}\int_V \rho \boldsymbol{w}\, dV &= \frac{\partial}{\partial t}(\rho w A \Delta z),\\
\int_A \rho \boldsymbol{w} w_A \boldsymbol{n}\, dA &= (\rho w^2 A)_E - (\rho w^2 A)_W = \frac{\partial}{\partial z}(\rho w^2 A)\Delta z,\\
\int_V \rho \boldsymbol{g}\, dV &= \rho g cos\phi A \Delta z = F_g\\
\int_A p\, dA &= \left\{ \lim_{\Delta z \to 0} \frac{p_2 A_2 - p_1 A_1}{\Delta z} - \frac{A_2 - A_1}{\Delta z}\bar{p} \right\} \Delta z\\
&= \frac{\partial}{\partial z}(pA) - p\frac{\partial A}{\partial z} = A\frac{\partial p}{\partial z} \text{ and}\\
\int_A \mathsf{T}\boldsymbol{n}\, dA &= F_F.
\end{aligned}$$

The friction force is calculated via empirical pressure drop laws because an evaluation of the integral in (4.11) is only possible for very simple flow fields not commonly found in reality. The derivation of the pressure integral for the flow channel of length Δz makes use of the assumption of a linear change in pressure on the area between the faces E and W.

Using the continuity equation, $\dot{m} = \rho w A$ and introducing the variable I for the momentum, the equation can be brought into the following discretized form:

$$I = \int_V \rho w \, dV = \int_{\Delta z} \int_A \rho w \, dA dz = \dot{m} \Delta z$$

$$\Delta z \frac{d\dot{m}}{dt} = \frac{dI}{dt} = \dot{I}_1 - \dot{I}_2 + p_1 A_1 - p_2 A_2 - F_F - F_g \tag{4.13}$$

This form of the equation still leaves an important implementation detail unspecified: how should the momentum fluxes $\dot{I}_1$ and $\dot{I}_2$be evaluated? The approximation for the flux terms has considerable influence on the numerical stability properties of the momentum balance which, if the friction term F_F is small, has eigenvalues close to the imaginary axis. Different methods of calculating these terms are described in textbooks dealing with numerical fluid dynamics, see [Versteeg and Malalasekera, 1995]. Implementation of the momentum balance model are treated in Section 5.6.

Energy Balance

Applying the Leibnitz rule (4.1) with $\psi = e$ yields the rate of change of energy

$$\frac{dE}{dt} = \int_V \frac{\partial(\rho e)}{\partial t} \, dV + \int_A \rho e \boldsymbol{w}_A \boldsymbol{n} \, dA \tag{4.14}$$

The energy of a closed system ($\boldsymbol{w}_A = \boldsymbol{w}$) is, according to the first law of thermodynamics, increased only by an addition of heat and work. If $\dot{Q}$ denotes the heat flux and P denotes the power, this gives

$$\int_V \frac{\partial(\rho e)}{\partial t} \, dV + \int_A \rho e \boldsymbol{w} \boldsymbol{n} \, dA = \sum P + \sum \dot{Q} \tag{4.15}$$

The power P is the integral of the local power dP, which is the local work per unit time $dP = dW/dt$. The work results from the movement of a particle along a line $d\boldsymbol{z}$, caused by a force $d\boldsymbol{F}$ acting on the particle. The work is the component of the force in direction of the movement times the length, i.e. the scalar product $dW = d\boldsymbol{F} d\boldsymbol{z}$. The quotient of $d\boldsymbol{z}$ and time dt is the flow velocity vector $\boldsymbol{w}$, thus

$$P = \int dP = \int \frac{dW}{dt} = \int \frac{d\boldsymbol{F} d\boldsymbol{z}}{dt} = \int \boldsymbol{w} d\boldsymbol{F}$$

Inserting the forces introduced in the previous section yields

$$P_g = \int_V \rho \boldsymbol{g} \boldsymbol{w} \, dV$$
$$P_p = -\int_A p \boldsymbol{w} \boldsymbol{n} \, dA$$
$$P_\tau = \int_A \mathsf{T} \boldsymbol{w} \boldsymbol{n} \, dA$$

When solving (4.15) for the first term and inserting it into (4.14) one obtains

$$\frac{dE}{dt} = \int_A \rho e (\boldsymbol{w}_A - \boldsymbol{w}) \boldsymbol{n} \, dA + P_g + P_p + P_\tau + \dot{Q} \tag{4.16}$$

where the first term on the right hand side quantifies the convective transport of energy.

The evaluation of the terms in the energy balance for the control volume in Figure 4.1 will follow two paths in order to show a complete energy balance and one where the kinetic energy terms are neglected and only the internal energy is considered. The models that are implemented in the ThermoFluid library were designed for processes where the kinetic energy is usually such a small fraction of the total energy that this simplification is justified. As one example, the role of the kinetic energy in a steam power plant is examined. Typical values for the specific enthalpy in a steam power plant are between 1000 and 3500 kJ/kg. Maximum speeds are around 30 m/s with a total upper limit of 50 m/s. This results in a ratio of specific energies of $(0.5\, w^2)/h \approx 0.5 \times 10^{-5} - 0.5 \times 10^{-4}$. Even inside turbines with much higher flow speeds, the error is only a few percent. The relative importance does not justify the inclusion of the kinetic energy terms in the cases considered.

The purpose of certain flow equipment like diffusers and nozzles is the recovery or generation of kinetic energy. Models for this equipment are currently not part of the library. They can easily be added when the kinetic energy is included in the energy balance.

The change of the total energy in the control volume is now expressed in terms of inner and kinetic energy:

$$\frac{dE}{dt} = \frac{dU}{dt} + \frac{dE_{kin}}{dt} = \frac{\partial}{\partial t} \int_V \rho \left(u + \frac{w^2}{2} \right) dV$$

The integral over the area for the convective terms setting $\boldsymbol{w}_A = 0$ (Eu-

lerian approach with fixed control volume) evaluates to:

$$\frac{\partial}{\partial t}\int_V \rho\left(u+\frac{w^2}{2}\right)\,dV = \frac{\partial}{\partial t}\left[\rho\left(u+\frac{w^2}{2}\right)A\Delta z\right]$$
$$\int_A \rho\left(u+\frac{w^2}{2}\right)\boldsymbol{w}\boldsymbol{n}\,dA = \frac{\partial}{\partial z}\left[\rho w\left(u+\frac{w^2}{2}\right)A\right]\Delta z$$

Taking into account that the velocity is zero at the channel wall:

$$\int_A \boldsymbol{w}\boldsymbol{n}p\,dA = (w_2p_2A_2 - w_1p_1A_1) = \frac{\partial}{\partial z}(wpA)\Delta z$$
$$\int_V \rho\boldsymbol{g}\boldsymbol{w}\cos\phi\,dV = \rho g w\cos\phi A\Delta z$$
$$P_\tau = \int_{A_E,A_W} \mathsf{T}\boldsymbol{w}\boldsymbol{n}\,dA \approx 0$$

The forces which apply at the wall perform no work because the pipe wall is fixed. That means that P_τ is the work of the normal shear stresses at A_E and A_W which are negligible. P_τ is not the power due to friction which does not appear in an overall energy balance. Friction does not affect the overall energy, but causes transformation of kinetic energy to internal energy within the fluid.

For the control volume in Figure 4.1 and a single phase fluid it is of no importance to include the pressure work for a variable size volume. For a control volume with variable volume like a cylinder in a compressor or motor, another power is part of the equation:

$$P_{dV} = p\frac{dV}{dt} \tag{4.17}$$

Because of symbolic processing in equation based languages, there is no disadvantage to include this term in the general base classes: it will only be used when $dV/dt \neq 0$. The same is true for dissipative work W_{diss}, which has not been included in the derivation of the flow channel but may be included in lumped volumes like a stirred tank reactor.

Using the above integrals, replacing $u + p/\rho$ with the enthalpy h and dividing by the length Δz, the final energy balance is obtained:

$$\frac{\partial}{\partial t}\left[\rho\left(u+\frac{w^2}{2}\right)A\Delta z\right] + \frac{\partial}{\partial z}\left[\rho w\left(h+\frac{w^2}{2}\right)A\right]\Delta z = \frac{\dot{Q}}{\Delta z} - \rho g w\cos\phi A\Delta z \tag{4.18}$$

Resolving this equation into one equation for kinetic energy and one for inner energy is difficult. In order to do so with a physical model, a velocity profile has to be assumed for the flow channel. With a given, constant velocity profile for laminar flow or a turbulence model, see e. g., [Versteeg and Malalasekera, 1995], it would be possible to calculate the dissipation rate of kinetic energy and evaluate the integrals in the following equations:

$$\frac{dE_{kin}}{dt} = -\int_{A_{1,2}} \rho \frac{\boldsymbol{w}^2}{2} \boldsymbol{w}\boldsymbol{n}\, dA + P_g - \int_V w_i \frac{\partial p}{\partial z_i}\, dV + \int_V w_i \frac{\partial \tau_{ji}}{\partial z_j}\, dV \quad (4.19)$$

$$\frac{dU}{dt} = -\int_{A_{1,2}} \rho u \boldsymbol{w}\boldsymbol{n}\, dA + \dot{Q} - \int_V p \frac{\partial w_i}{\partial z_i}\, dV + \int_V \tau_{ji} \frac{\partial w_i}{\partial z_j}\, dV \quad (4.20)$$

The last two terms in (4.19) quantify the rate of change of kinetic energy due to surface forces. The related terms in (4.20) cause a deformation of the fluid particles. The deformation work cannot be stored as potential energy, but is irreversibly transformed into heat, and thus increases the internal energy [Guyon *et al.*, 1997]. (4.20) is equivalent to

$$\frac{dU}{dt} = -\int_{A_{1,2}} \rho u \boldsymbol{w}\boldsymbol{n}\, dA + \dot{Q} + P_p + \int_V w_i \frac{\partial p}{\partial z_i}\, dV + \int_V \tau_{ji} \frac{\partial w_i}{\partial z_j}\, dV$$

The last term therein is positive – otherwise internal energy would be transformed into kinetic energy, which violates the second law of thermodynamics. The pressure integral is negative, since a negative pressure gradient is required to overcome the friction. For simplicity and following similar order-of-magnitude arguments as for the kinetic energy it will be assumed that both integrals sum up to approximately zero.

$$\int_V w_i \frac{\partial p}{\partial z_i}\, dV + \int_V \tau_{ji} \frac{\partial w_i}{\partial z_j}\, dV \approx 0$$

For a simple, lumped control volume with n connections to the surroundings, including pressure-volume and dissipative work, heat transfer and heat contributions from reactions, the inner energy balance becomes:

$$\frac{dU}{dt} = \sum_{i=1}^{n} H_i + p\frac{dV}{dt} + W_{diss} + \dot{Q}_{HT} + \dot{Q}_{reac} \quad (4.21)$$

where $H_i = \dot{m}\ h_{i,upstream}$ is always calculated with the specific enthalpy from the control volume upstream of the connection. If the enthalpy of formation from reactions has to be included explicitly or not depends on the definition of the specific enthalpy of the components. This issue is discussed in more detail in Section 4.9.

4.4 The General Transport Equation

The approach used in fluid dynamics a slightly different from the one presented in the previous section: the basis for the equations is the transport of an intensive property in a flow field, diffusion is added to the physical phenomena that are part of the basic description and the similarities between the intensive forms of the conservation equations for mass, energy and momentum are used to derive a *general transport equation* for a variable ϕ. It is obvious from the source term that this accounting equation is not based on a fundamental conservation law. ϕ can be used for any scalar quantity which is transported with the flow, e. g., temperature or pollutant concentration. The general transport equation is described by four terms:

$$\frac{\partial \rho\phi}{\partial t} + \nabla \cdot (\rho\phi \boldsymbol{w}) = \nabla \cdot (\Gamma \nabla^2 \phi) + S_\phi \tag{4.22}$$

In words this can be expressed as:

Rate of increase of ϕ in fluid element	+	Net rate of flow of ϕ out of fluid element	=	Rate of increase of ϕ due to diffusion	+	Rate of increase of ϕ due to sources

The first three terms defined above are well defined, but the source term usually serves as a collection of cross-coupling terms to other balance equations and is also often used to correct for numerical artefacts introduced through the discretization of the divergence terms. The above representation holds independently of the dimension of discretization and applies equally to an infinitesimal control volume and when each term is integrated over a finite size control volume.

$$\int_{CV} \frac{\partial \rho\phi}{\partial t} + \int_{CV} \nabla \cdot (\rho\phi \boldsymbol{w}) = \int_{CV} \nabla \cdot (\Gamma \nabla^2 \phi) + \int_{CV} S_\phi$$

For the standard equations in process engineering, $\phi = 1$ gives the mass balance, $\phi = \mathbf{w}$ results in the momentum balance and $\phi = h$ gives the energy balance.

From a thermodynamic point of view another characterization can be be made for the terms in (4.22). The terms on the left hand side describe the *reversible* thermodynamics and the terms on the right hand side contain terms of *irreversible* thermodynamic phenomena.

4.5 Thermodynamic Equations of State

In order to close the system of equations mathematically, two types of material laws are needed: a correlation between velocity gradients and shear

stresses, most often the assumption of a Newtonian fluid, and a thermodynamic equation of state (EOS). The latter of these relations originates from the assumption that the transients of the system under consideration are much slower than the time that a fluid needs to reach thermodynamic equilibrium. The assumption of thermodynamic equilibrium is very common in thermodynamic models. It holds very well for single phase systems, but for two phase systems with fast dynamics, thermodynamic non-equilibrium is sometimes taken into account. The deviation of the phases from equilibrium is the physical driving force for the phase change mass transfer.

The existence of an EOS can be derived directly from the First and Second Law of thermodynamics, written in differential form in (4.23–4.24), when the reversible limit is taken for system changes between equilibrium states in the Second Law[2]. When the First and Second Law for simple systems consisting of n distinct chemical species are integrated over an infinitesimal time interval dt:

$$dU = dQ - dW + \sum_{i=1}^{n} h_i dM_i \tag{4.23}$$

$$dS_{gen} = dS - \frac{dQ}{T_0} - \sum_{i=1}^{n} s_i dM_i \geq 0 \tag{4.24}$$

and reversible changes are assumed ($dS_{gen} = 0$ and $dW = dW_{rev} = pdV$), the equations can be combined into the following form:

$$dU = TdS - pdV + \sum_{i=1}^{n} (h - Ts) dM_i \tag{4.25}$$

If a molar description and basis is used for all variables, the dependence of the energy on three different potentials is even more obvious:

$$dU = TdS - pdV + \sum_{i=1}^{n} \mu_i dN_i \tag{4.26}$$

where μ_i is the chemical potential of species i. This form of the combined law proclaims the existence of a function of $(n+2)$ variables

$$U = \mathcal{U}(S, V, N_1, N_2, ..., N_n) \qquad \text{where} \tag{4.27}$$

[2]A rigorous derivation is given in [Bejan, 1997], chapter 4. The definition of thermodynamic equilibrium is closely connected to the derivation of an equation of state and the assumption of *reversibility*.

$$T = \left.\frac{\partial U}{\partial S}\right|_{V,,N_1,N_2,\ldots,N_n}$$
$$-p = \left.\frac{\partial U}{\partial V}\right|_{S,N_1,N_2,\ldots,N_n}$$
$$\mu_i = \left.\frac{\partial U}{\partial N_i}\right|_{S,V,N_1,\ldots,N_{i-1},N_{i+1},\ldots,N_n}$$

From this fundamental relation in energy representation, a variety of other forms can be derived. Equations are called fundamental because they contain the complete thermodynamic property information of the fluid. Using this derivation of an equation of state, it is now possible to give a simple explanation of thermodynamic equilibrium: In a system in thermodynamic equilibrium, the values of the potential variables p, T and μ_i are homogeneous throughout the system. Thermodynamic equilibrium is thus equivalent to the "perfect mixing" assumption.

In order to get a reasonable parameterization, fundamental relations are calculated for pure chemical species and *mixing rules* are applied to construct multi-component relations. These material laws give an algebraic relation between the thermodynamic variables pressure p, temperature T, density ρ, specific energy u, enthalpy h, entropy s, free energy f and free enthalpy g. As can be seen from (4.27), T and p can be calculated by symbolically differentiating the EOS. The enthalpy is then obtained from $h = u + pv$, other properties follow from further differentiation or non-linear combinations of known variables.

Three further fundamental equations can be derived by applying the Legendre transformation [Bejan, 1997] to the above combined First and Second Law of thermodynamics. Loss of information through the coordinate change is avoided when applying the Legendre transformation, details about how to apply the transformation to thermodynamic functions are given in Appendix B. The further fundamental equations are:

- the Helmholtz free energy $F = \mathcal{F}(T, V, N_i)$,
- the Gibbs free enthalpy $G = \mathcal{G}(T, p, N_i)$,
- the enthalpy fundamental equation $H = \mathcal{H}(S, p, N_i)$.

Most often these fundamental equations are derived for single species properties and intensive variables as $u(s, v), f(T, v), g(T, p)$ and $h(s, p)$. For numerical reasons they are also normalized, mostly with the critical values of pressure, temperature or density as normalization factors. For high-accuracy EOS, the Helmholtz energy function is the most popular form closely followed by the Gibbs free energy $g(T, p)$ because temperature T, density $\rho = 1/v$ and pressure p are the most easily measured

thermodynamic variables. The basic $u(s,v)$ or $h(s,p)$ which use the non-measurable entropy as one of their base variables are more of theoretical interest. Equations of state are derived by adapting coefficients in high-order two-dimensional polynomials, sometimes augmented with nonlinear terms, to measurement data, see [Span, 2000].

A practical approach with a long history which is very popular in process engineering is to use pressure – volume – temperature correlations, mostly cubic equations of state of the general form:

$$p = \frac{RT}{V-b} - \frac{\Theta V - \eta}{(V-b)V^2 + \delta V + \varepsilon} \tag{4.28}$$

where R is the gas constant, T is the temperature and V is the volume, $\Theta, b > V, \eta, \delta$ and ε may be constants including 0 or they may vary with T and/or composition. They are also based on easily measurable quantities but the cubic EOS have the disadvantage that they are not fundamental equations. In order to calculate all fluid properties, the heat capacity in the ideal gas state is needed in addition. It can be shown that the combination of a $p(V,T)$ surface and a function for the heat capacity $c_p = c_p(T)$ in the ideal gas state are equivalent to a fundamental equation. The attractive feature of cubic EOS is that they allow relatively good property estimates with few parameters, namely the critical values of temperature T_c, pressure p_c and specific volume v_c. Numerical features of cubic EOS in combination with dynamic simulation are discussed in Section 4.6.

The simplest form of an EOS is the Ideal Gas Law

$$pV = N\ R\ T$$

with the general gas constant R. In the simplest form it is complemented with a constant c_p and sometimes called Perfect Gas Law. For ideal gases c_p and h are functions of temperature only. The following relations are used to calculate the caloric properties:

$$c_p(T) = f(T) \tag{4.29}$$

$$h(T) = h_0 + \int_{T_0}^{T} c_p(T)dT \tag{4.30}$$

$$s(T,p) = s_0 + \int_{T_0}^{T} c_p(T)\frac{dT}{T} - R\ \ln\frac{p}{p_0} \tag{4.31}$$

Note that the pressure dependent part of the specific entropy is identical for all ideal gases, only the temperature dependent part depends on the gas.

Implemented forms of these material laws in the ThermoFluid library are the Ideal Gas Law, the steam tables for water and high accuracy property functions for some refrigerants. The steam tables use both Gibbs functions $g(T, p)$ and Helmholtz functions $f(T, d)$ in different regions of the input coordinates T and p. The development of these equations of state is a complex task, using non-linear optimization techniques to fit the thermodynamic surface to available measurements, see [Span, 2000]. The generation of such property models on numerically efficient form is often the most troublesome area in simulation of thermodynamic flow problems, especially when high accuracy models are required. This problem is also well known in process industry, where the quality of fluid property routines is a limiting factor for simulation accuracy in spite of a large number of commercially available medium property databases. The optimal situation for modelers and control engineers would be to have a spectrum of options ranging from simple, but computationally efficient to complex, high accuracy EOS models with a free choice of primary variables.

For steam power plants the situation has improved recently with the standardization of the IAPWS/IF97[3]. The IF97 splits the region of validity of the steam tables into 5 regions, three of them use a Gibbs equation, one uses a Helmholtz equation and the two-phase region is defined by a saturation pressure curve and the adjoining fundamental equations. Even for the steam tables, an alternative with lower computational cost for online computations for control tasks is not available, but highly desirable.

Another, related problem with the available models is that they are not designed to be used in dynamic simulation. For dynamic computations, efficiency is of higher priority than for steady state calculation or generation of tables. The main issues are:

- Computational efficiency is improved a lot if iterative routines are replaced by explicit computations. One type of iterations is avoided when the dynamic states of the thermodynamic model are identical to the input variables of the equation of state. Usually, there is no choice on the side of the equation of state (the only exception are the IF97 steam tables), but it is possible to apply a non-linear transformation to the mass and energy balance, see Section 4.6, to use the transformed version with any pair of intensive thermodynamic variables. The transformations require knowledge of certain thermodynamic derivatives used in the Jacobian matrix. These derivatives can easily be computed analytically from the equation of state, but are not included in standard implementations.

[3]The standard of the "Industrial Formulation of the properties of Water and Steam" is described in detail in [Wagner and Kruse, 1998].

- Calculation of phase equilibria is mostly done iteratively, but according to Gibbs' phase rule the dimensionality of the equilibrium subspace is one less than that of the full thermodynamic surface. The phase equilibrium is defined implicitly via the equality of the values of the Gibbs' free enthalpy for both phases, e.g., $g(T, p, X)_{liquid} = g(T, p, X)_{vapor}$ for a multi-component fluid. Often it is possible to calculate a very accurate approximation to the equilibrium surface, which can be used from a library of stored definitions or generated on the fly before the start of the dynamic simulation.
- Most functional models of the thermodynamic surface are by design singular at the critical point. For robust dynamic simulations, a workaround which does not have non-physical side-effects on the simulation result has to be worked out. This and similar implementation issues will be the topic of Chapter 6.

Implementation issues of the medium property calculations are discussed in Section 5.6.

This discussion about efficiency of the medium property calculation, combined with the derivation of the basic balance equation in Section 4.3 leads to an interesting question: which choice of state variables will give the most efficient and reliable simulation results?

4.6 Choice of Dynamic State Variables

A clean and straightforward way of modeling thermodynamic and process systems is to always use the conserved extensive quantities as dynamic states: component (or total) mass, inner energy and momentum. Unfortunately there are several choices of fundamental extensive quantities which can be chosen as states, even from this puristic perspective, see [Weiss and Preisig, 2000] and [Westerweele and Preisig, 2001]. The situation gets worse if simulation efficiency, integration of existing code and other software limitations are taken into account in addition. An ideal situation would be to write models always as conservation laws for fundamental extensive quantities and have the simulation tool take the decision which numerical formulation is most appropriate. The optimal choice of states this might even change during one simulation run. For modeling of transient two-phase flows, there is a long tradition of transforming the equations to state variables which are numerically efficient, see [Kolev, 1986]. In the following, a couple of efficiency motivated model reformulations are presented. Currently, Modelica tools can not perform this type of of model reformulation automatically[4].

[4]They can theoretically be handled by the current version of Dymola (4.2), but rewriting

Single Component Fluids

In order to be as general as possible, the derivations will be done for a control volume of variable size, first for single component fluids and, equivalent in the single phase case, fluid mixtures with fixed mass composition. The change of volume dV is usually caused by moving systems parts like an engine piston.

$$\frac{d}{dt}\begin{pmatrix} M \\ U \\ V \end{pmatrix} = \begin{pmatrix} \sum_i^n \dot{m} \\ \sum_i^n \dot{q}_{conv,i} + \sum_j^l \dot{q}_{transfer,j} - p\frac{dV}{dt} \\ dV \end{pmatrix} = \begin{pmatrix} \frac{dM}{dt} \\ \frac{dU}{dt} \\ \frac{dV}{dt} \end{pmatrix} \tag{4.32}$$

In a first step, this equation is written in the intensive variables density ρ and specific inner energy u and the volume.

$$\frac{d}{dt}\begin{pmatrix} \rho \\ u \\ V \end{pmatrix} = \begin{pmatrix} \frac{1}{V} & 0 & -\frac{\rho}{V} \\ -\frac{u}{M} & \frac{1}{M} & \frac{u}{V} \\ 0 & 0 & 1 \end{pmatrix} \frac{d}{dt}\begin{pmatrix} M \\ U \\ V \end{pmatrix}$$

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

$$\frac{d}{dt}\begin{pmatrix} \rho \\ u \\ V \end{pmatrix} = \underbrace{\begin{pmatrix} \left.\frac{\partial \rho}{\partial p}\right|_h & \left.\frac{\partial \rho}{\partial h}\right|_p & 0 \\ \left.\frac{\partial u}{\partial p}\right|_h & \left.\frac{\partial u}{\partial h}\right|_p & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{Jacobian Matrix J}} \frac{d}{dt}\begin{pmatrix} p \\ h \\ V \end{pmatrix} \tag{4.33}$$

To obtain differential equations for pressure and enthalpy (4.33) must be solved for the derivative of (p, h)

$$\frac{d}{dt}\begin{pmatrix} p \\ h \\ V \end{pmatrix} = \mathrm{J}^{-1} \frac{d}{dt}\begin{pmatrix} \rho \\ u \\ V \end{pmatrix}$$

existing EOS-implementations in such a way that an automatic choice of states can be done by Dymola is a lot more work than the manual coordinate transformation presented in this section. Another possibility would be to extend Modelica's annotations for functions with more details about derivatives into specific directions.

The inverse of the Jacobian is computed as

$$\mathsf{J}^{-1} = \frac{1}{\det \mathsf{J}} \begin{pmatrix} \frac{\partial u}{\partial h}\Big|_p & -\frac{\partial \rho}{\partial h}\Big|_p & 0 \\ -\frac{\partial u}{\partial p}\Big|_h & \frac{\partial \rho}{\partial p}\Big|_h & 0 \\ 0 & 0 & \frac{\partial \rho}{\partial p}\Big|_h \frac{\partial u}{\partial h}\Big|_p - \frac{\partial \rho}{\partial h}\Big|_p \frac{\partial u}{\partial p}\Big|_h \end{pmatrix}$$

with the determinant

$$\det \mathsf{J} = \frac{\partial \rho}{\partial p}\Big|_h \frac{\partial u}{\partial h}\Big|_p - \frac{\partial \rho}{\partial h}\Big|_p \frac{\partial u}{\partial p}\Big|_h$$

It is possible to reduce the partial derivatives of u to the ones of ρ by using $u = h - p/\rho$, but this rewrite does not improve the clarity of the model. It may be useful for implementation though, if only the ρ-derivatives are available[5].

Equivalent derivations can be done for pressure and temperature as states and for density and temperature. The advantage of the latter two forms is that there are many medium property models which are explicit in these variables. This avoids a non-linear system of equation in a central part of the model and is therefore very efficient. In order to simplify notation, the volume is assumed constant in these derivations.

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

For ideal gases this simplifies further because, $\partial u/\partial \rho|_T = 0$ and further it is common to write $\partial u/\partial T|_\rho = c_v$, the heat capacity at constant volume. Solving for density and temperature yields:

$$\frac{d}{dt}\begin{pmatrix} \rho \\ T \end{pmatrix} = \mathsf{J}^{-1} \frac{d}{dt}\begin{pmatrix} \rho \\ u \end{pmatrix},$$

with the inverse of the Jacobian:

$$\mathsf{J}^{-1} = \frac{1}{c_v}\begin{pmatrix} c_v & 0 \\ -\frac{\partial u}{\partial \rho}\Big|_T & 1 \end{pmatrix}$$

[5]Note: this way of writing the transformations to select suitable state variables assumes that some other part of the model is able to calculate the partial derivatives in the matrix efficiently. In ThermoFluid, the medium models compute the needed derivatives.

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 dynamic states. But outside the two phase region they have the advantage that these two variables are often readily available from measurements.

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

For ideal gases the above simplifies again as $\frac{\partial u}{\partial p}\Big|_T = 0$ and $\frac{\partial u}{\partial T}\Big|_p = c_v$. Solving for pressure and temperature yields:

$$\frac{d}{dt}\begin{pmatrix}p\\T\end{pmatrix} = \mathsf{J}^{-1}\frac{d}{dt}\begin{pmatrix}\rho\\u\end{pmatrix}. \tag{4.34}$$

Writing the same physical model of a fluid in a control volume in three different ways may seem nothing more than an academic exercise, but if we look at the numerical implications of the combination of a thermodynamic EOS and one of the above dynamic equations into a DAE, there are two reasons for using different models in different situations. The goal of the reformulation is to arrive at a model in which the dynamic states are

- explicit inputs to the EOS (in the cases of $\{\rho, T, V\}$ and $\{p, T, V\}$ as states) or
- make use of the shape of the EOS to choose numerically favorable state coordinates in the case of $\{p, h, V\}$.

The shape of the EOS can be such that a small error in one of the states (e. g., with an implicit equation for the EOS which is solved numerically only to a certain accuracy) results in a large error of other variables calculated via the EOS. If liquids have to be modeled as compressible, the density (equivalently, the total mass) have the property that a small numerical error in them is amplified via a gain given by $\partial p/\partial\rho|_h$ to the corresponding pressure given by the EOS. The pressure in turn influences strongly the mass flows into the control volume and the change in mass in the next time step. The result is a fluctuation in the pressures and mass flows that looks like a noise signal for moderate tolerance choices of the integration routine. Thus, density is a bad choice of state variable in the liquid region. A good visualization of this property is given by the plots

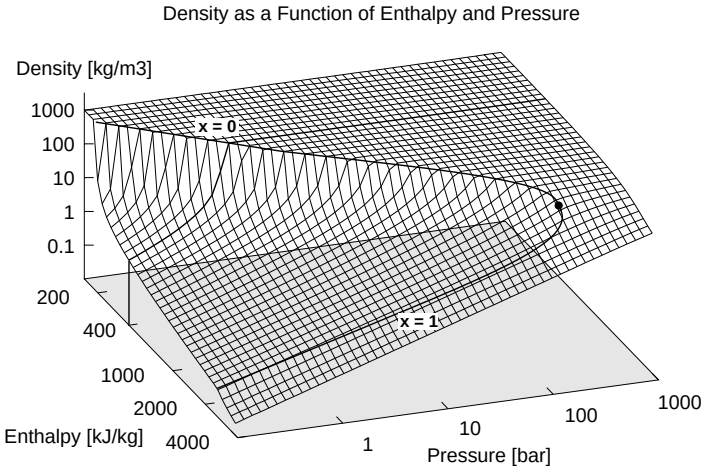


Figure 4.4 EOS for water: density as a function of pressure and specific enthalpy. Used with permission from [Mühlthaler, 2000].

of the EOS for water in Figure 4.4 and 4.5. Note the logarithmic scales for all variables, which are necessary to catch the technically interesting region.

Multi-Component Fluid Mixtures

For multi-component flows, the number of possible choices for the dynamic states gets larger and the availability of numerically robust and simple definitions for the EOS gets worse. For the EOS, the basis are the EOS of the single components which are combined using empirical mixing rules. A detailed presentation of mixing rules is found in [Poling *et al.*, 2001]. Most of the equations of state which are in use in process engineering are cubic[6] equations of state of the general form

$$p = \frac{RT}{V - b} - \frac{a}{V^2 + ubV + wb^2} \tag{4.35}$$

where a, b, u and w are component specific constants, R is the gas constant, T is the temperature and V is the volume. This is a simplified version of (4.28) assuming that $\eta = b$. It can easily be seen that this equation can not be made explicit in variables which can be used as dynamic states,

[6]These equations are called cubic because they can be transformed into a cubic polynomial in the *compressibility*, see [Poling *et al.*, 2001].

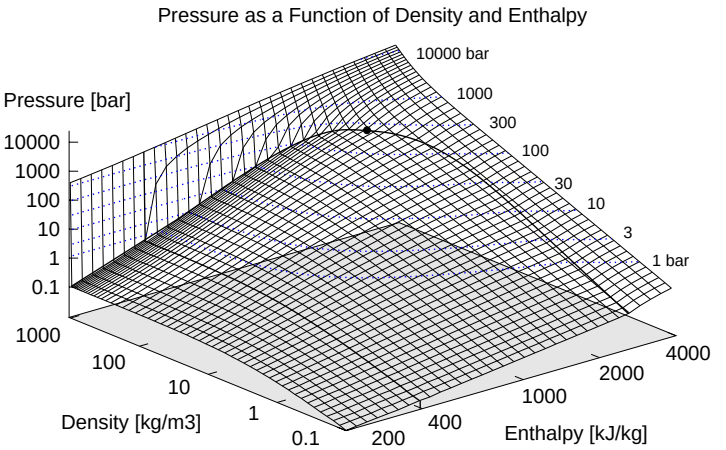


Figure 4.5 EOS for water: pressure as a function of density and specific enthalpy. The strong variations of pressure in the liquid region caused by small density variations are obvious from this plot of the density-pressure-enthalpy surface. Used with permission from [Mühlthaler, 2000].

e. g., p and T. It can be rewritten to a cubic equation in the compressibility Z (see [Poling *et al.*, 2001]), but this equation has three solutions in some areas and only one of them is physically meaningful. A non-linear system of equations is therefore not avoidable, but selecting T instead of U as one of the dynamic states reduces the system of equations to dimension one, solving for p. A common choice of property computations for static calculations is to treat p as an input in the calling structure, using the compressibility form of the cubic EOS. This gives a non-linear system of equations if the volume V is a known input (constant or, in piston engines, a state). The derivative $\partial p/\partial V|_T$ can be calculated analytically to improve efficiency when solving for p with a Newton iteration.

When dealing with mixtures of components, both mass- and mole based models can be used, they are fully equivalent. Rewriting the inner energy and mole amounts into temperature and moles as dynamic states is done as follows (block-matrix notation, boldface for vectors and matrices):

$$\frac{d}{dt}\begin{pmatrix}\mathbf{N}_n\\ U\\ V\end{pmatrix} = \begin{pmatrix}\mathbf{I}_{n,n} & \mathbf{0}_{n,1} & \mathbf{0}_{n,1}\\ \left.\frac{\mathbf{dU}}{\mathbf{dN_i}}\right|_{T,V\,1,n} & \left.\frac{dU}{dT}\right|_{\mathbf{N},V} & \left.\frac{dU}{dV}\right|_{\mathbf{N},T}\\ 0 & 0 & 1\end{pmatrix}\frac{d}{dt}\begin{pmatrix}\mathbf{N}\\ T\\ V\end{pmatrix} \tag{4.36}$$

The inverse of the Jacobian is used again to make this model explicit in

the mole vector, the temperature and the volume:

$$\mathsf{J}^{-1} = \frac{1}{\frac{dU}{dT}\big|_{\mathbf{N},V}} \begin{pmatrix} \mathbf{I}_{n,n} & \mathbf{0}_{n,1} & \mathbf{0}_{n,1} \\ -\frac{\mathbf{dU}}{\mathbf{dN_i}}\Big|_{T,V\,1,n} & -\frac{dU}{dT}\big|_{\mathbf{N},V} & -\frac{dU}{dV}\big|_{\mathbf{N},T} \\ 0 & 0 & 1 \end{pmatrix}$$

The structure of the Jacobian inverse reveals that only the equation for the inner energy is transformed into one for the temperature. The equations for the moles remain unchanged from (4.36).

The same transformation can be applied equivalently for component masses. By exchanging moles N_i with component masses M_i and derivatives with respect to N_i with derivatives with respect to M_i, (4.36) is the same for a component mass based model.

The generality and multi-purpose formulation of the models makes it difficult to recognize the underlying PDE in the above equations. A variable size volume is rarely assumed in PDE models and only used in lumped control volume models, but this makes it possible to use the same Modelica classes for both cases with different parameters and boundary conditions. Transformation into different forms of states makes it difficult to see the connection between these equations and the balance equations in Section 4.3. Furthermore, PDE formulations are usually written in intensive variables for an infinitesimal control volume. A comparison to the PDE-formulation in (4.4) reveals the following differences:

- Diffusion is neglected because it does not apply to lumped parameter models. Inside distributed models it is a trivial extension to add diffusion to the mass- or energy flow terms.
- The derivations take extensive quantities as the fundamental description and only change to intensive ones if necessary for improving efficiency.
- The transport equations for energy and mass (or component masses) are regarded as vector equations allowing a change of coordinates involving both of them. This is not done in computational fluid dynamics.
- Source terms and in/outflow terms are lumped together for the change of coordinates.

The different forms of the dynamic state equations for mass and energy are implemented in `BaseClasses.StateTransforms`, see Section 5.6.

Some other modeling packages, like gPROMS for process modeling, recommend differing guidelines for writing lumped and distributed parameter models. The gPROMS developers recommend to write lumped

parameter models in extensive variables and distributed parameter models in intensive variables[7]. This corresponds to the prevailing presentation of lumped and distributed parameter models in the literature. On the other hand, this precludes the savings in coding and maintenance effort which is a major motivation behind object-oriented library development. Differing guidelines for these two cases are incompatible with the ThermoFluid principle of unifying lumped and distributed parameter models.

4.7 Turbines and Valves

Models for turbines and valves share many properties with respect to the modeling of the fluid passing through them. In models for turbines and valves it is common to neglect mass- and energy storage, they are *flow models*, see Section 5.6 They cause a pressure drop across them in the flow direction and choking occurs for high mass flows in turbines and valves. Turbines convert flow enthalpy into mechanical energy of the shaft while valves change the thermodynamic state of the fluid along paths that lead to lower pressure.

Turbine Stages

The mass flow through a group of turbine stages is calculated from the well-known Stodola steam-cone equation:

$$\mu_T = C_1 \cdot \sqrt{1 - \Pi_T^2} \tag{4.37}$$

where the variables are

μ_T : the reduced mass flow, $\mu_T = \dfrac{\dot{m}\sqrt{T_{in}}}{p_{in}}$

Π_T : the pressure ratio between turbine inlet and outlet, $\Pi_T = \dfrac{p_{out}}{p_{in}}$

C_1 : a constant dependent on the nominal conditions

(4.37) computes the mass flow for a turbine with an infinite number of stages. In practice this holds well for power plant steam turbines near normal operating conditions. The practical meaning is that the mass flow is much smaller than the mass flow at choking conditions. When turbines are modeled in more detail or gas turbines with few stages are modeled, this simplification causes large errors. The disadvantages of the

[7]Per volume quantities are the most usual choice for chemical engineering problems.

Stodola equation can be avoided with an improved relation developed by *Linnecken*, see [Cordes, 1963]. The mass flow is calculated as

$$\mu_T = C_2 \cdot \sqrt{(1 - \Pi_k)^2 - (\Pi_T - \Pi_k)^2} \tag{4.38}$$

with a constant C_2 analogous to (4.37). In the case of an infinite number of stages, $\Pi_k = 0$, this equation reduces again to the Stodola steam-cone.

The critical pressure ratio can according to [Cordes, 1963] be approximated to

$$\Pi_{crit} = \left[\frac{n-1}{2 \cdot \overline{\Lambda}} + 1\right]^{\frac{n}{1-n}} \tag{4.39}$$

$$n = \frac{\kappa}{\kappa - \eta_p \cdot (\kappa - 1)}. \tag{4.40}$$

The gradient parameter $\overline{\Lambda}$ depends on the number of stages. Reference values for $\overline{\Lambda}$ are listed in Table 4.1 The polytropic exponent n can be calculated from the relation between isentropic and polytropic efficiencies,

$$\eta_s = \frac{1 - \Pi^{\eta_p \cdot \frac{\kappa-1}{\kappa}}}{1 - \Pi^{\frac{\kappa-1}{\kappa}}}.$$

For pressure ratios different from nominal conditions, the isentropic efficiency falls below the nominal isentropic efficiency $\eta_{s,0}$ and can be approximated by

$$\eta_s = \left[1 - k_\alpha \cdot \left(\frac{\Pi_0}{\Pi} - 1\right)^2\right] \cdot \eta_{s,0}.$$

The characteristic constant k_α depends on the number of stages and Π_0. [Cordes, 1963] reports a range of $k_\alpha \approx 0.1 \ldots 1.5$.

Valves and Orifices

Gas flow in valves and orifices is usually modeled along two idealized thermodynamic paths:

- Isentropic flow is the limiting case of a reversible flow.
- Isenthalpic flow can be characterized as giving the maximum increase in entropy for a given pressure drop. In isenthalpic flow, pressure energy is irreversibly dissipated to inner energy.

Most thermodynamic texts emphasize the isentropic case, even though this is only a good approximation for nozzles operated near the design

number of stages	gradient parameter $\overline{\Lambda}$
1	0.5 ... 1.0
2	0.3 ... 0.5
3	0.25 ... 0.3
4 and more	0.25

Table 4.1 Reference values for $\overline{\Lambda}$ from [Linnecken, 1957]. The higher values are for impulse turbines, the lower ones for reaction turbines.

point. A control valve that is half open is much closer to the isenthalpic case.

A nozzle designed for subsonic flow limits the flow speed at the neck of the nozzle to the speed of sound. When the pressure ratio gets larger than the Laval pressure ratio, speed and mass flow stay constant at their maximum independent of the pressure ratio. The Laval pressure p_L is a function of the isentropic exponent κ:

$$\frac{p_L}{p_0} = \left(\frac{\kappa+1}{2}\right)^{\frac{\kappa}{1-\kappa}}, \tag{4.41a}$$

The flow function Ψ for isentropic flow through a nozzle is a function of the pressure ratio and defined as:

$$\Psi = \sqrt{\frac{\kappa}{\kappa-1}}\sqrt{\left(\frac{p}{p_0}\right)^{\frac{2}{\kappa}} - \left(\frac{p}{p_0}\right)^{\frac{\kappa+1}{\kappa}}} \qquad \text{if} \quad \frac{p}{p_0} \geq \frac{p_L}{p_0} \tag{4.41b}$$

$$\Psi = \Psi_{max} = \sqrt{\frac{\kappa}{\kappa+1}}\left(\frac{2}{\kappa+1}\right)^{\frac{1}{\kappa-1}} \qquad \text{if} \quad \frac{p}{p_0} < \frac{p_L}{p_0} \tag{4.41c}$$

Index 0 stands for the entry properties assumed to be at zero flow speed. When the nozzle equation is used in a control valve with a variable cross section area $A_V(y)$, the mass flow is a function of the valve position y, the inlet properties and the flow function Ψ

$$\dot{m}_V = A_V(y)\,\Psi\,\sqrt{2p_0\rho_0} \tag{4.42}$$

Numerical difficulties arise when the nozzle equation is used at flow speeds close to zero due to the singular derivative of the root function. In ThermoFluid this is avoided by using an interpolating polynomial instead of the root function in a user-definable region around zero flow.

The calculation of the enthalpy at the outlet follows from the assumption of an isentropic path to be

$$h_s = f(s_{inflow}, p_{outflow}) \tag{4.43}$$

The computation of h_s is the same for turbines and isentropic valves. Even for ideal gases the exact solution of the isentropic enthalpy requires solving a non-linear equation. For ideal gases, κ is only weakly dependent on temperature. It is feasible to use an average κ to get a good approximation to the isentropic enthalpy change:

$$h_{out,s} = h_{in} + \frac{\kappa}{\kappa - 1}\left(\frac{p_{in}}{d_{in}}\right)\left(\left(\frac{p_{out}}{p_{in}}\right)^{\frac{\kappa-1}{\kappa}} - 1.0\right).$$

An exact solution to the isentropic enthalpy can be found by iteratively solving the EOS used for property calculations using (4.43).

Isenthalpic flow can be modeled in a simpler way. A simple pressure loss model that is easy to adapt to measurements and works even for large variations in density is:

$$\frac{\dot{m}}{\dot{m}_0} = \sqrt{\frac{\Delta p}{\Delta p_0} \cdot \frac{\rho}{\rho_0}}$$

Again, the root function is replaced by a polynomial with finite derivative in a small region around zero flow speed. The density dependence can be omitted for nearly incompressible fluids or small density variations.

Base classes for turbines and valves are implemented in the package `PartialComponents`, except for calculations of the isentropic enthalpy change which are found in sub-packages of `MediumModels`, see Chapter 5.

4.8 Pumps and Compressors

The characteristic behavior of pumps and compressors in their stable operating range is captured by graphical maps or data tables. The map consists of two steady-state relationships describing the normalized pressure ratio to mass flow map and the efficiency for all stable compressor or pump operating points. There are two ways to represent this type of maps:

- Use data-intensive, piece-wise approximations to the data, e. g., bicubic spline interpolation or similar methods. When the data is available, this is the most accurate way to compute the maps.

- Non-dimensionalized relations for turbo machinery using much fewer parameters from nominal operating points can be scaled to the actual machine.

The second method has the advantage that it is often easier to extend to the unstable operating range of the machine where usually no measurements can be obtained.

One simple example of an expression of a normalized characteristic curve for turbo-pumps from [Pfleiderer and Petermann, 1991] which is also easy to adapt to a specific pump using manufacturer data is

$$\Delta p_n = R_1 n_n + 2R_2 n_n V_n - R_3 |V_n| V_n \tag{4.44}$$

where, p_n, n_n and V_n are normalized variables for pressure p, revolution rate of the pump n and volume flow rate V. The design point is $(p_n, n_n, V_n) = (1, 1, 1)$ and represents the pump in normal operation. This relationship works correctly in the case of back flow through the pump, in so called surge operation. For system level simulation it may be more important to capture the behavior in the unstable operating region qualitatively correct than to capture the stable behavior as accurately as possible.

More advanced dimensionless models of pumps which can also scale pump geometries of similar pump types to different sizes have not been implemented in ThermoFluid, they can be found in most books on pumps like [Pfleiderer and Petermann, 1991].

The main modeling challenge for compressors, similar to that of pumps, consists in a numerically robust and accurate representation of the so called "compressor map", which is given graphically or as tabulated data. Smooth interpolation of these tables with two-dimensional splines would be a suitable way to implement this form of data, but currently no such implementation is available in Modelica.

Several systems containing compressors have been modeled using the ThermoFluid library, but the models were specifically adapted to a specific compressor map using function approximations. The models are therefore not well suited for a general model library.

Similar to pumps it is possible to compute off-design behavior of multi-stage axial compressors using non-dimensionalized models. Methods using nominal per-stage values of pressure ratio, mass flow, isentropic efficiency and revolution rate to obtain corresponding values for the overall machine at all working conditions are called "stage composition methods"[8]. They are described in [Gašparović and Stapersma, 1973] and [Herzke, 1983] where the later author even includes rotating stall and surge operation.

[8] translated from German: "Stufenaufbauverfahren".

Base classes for pumps are implemented in PartialComponents.Pumps with some ready-to-use components for water in Components.Water.

4.9 Chemical Reactions

Modeling of chemical reactions can be done in two ways, either by using kinetic expressions for the reaction rates or by assuming that the reactions are much faster than all other dynamics so that chemical equilibrium can be assumed. This is similar to thermodynamic phase equilibrium or a quasi steady-state assumption for the momentum dynamics.

Currently only kinetic reactions are implemented in the ThermoFluid library, mainly because it is easier to obtain numerical robustness with dynamic reaction models when some concentrations are close to zero.

We assume that a number of nr reactions take place simultaneously in a lumped or discretized control volume. The rate r_j of reaction j denotes the number of moles of reaction taking place per unit time and unit volume. Several reactions can contribute to either produce or consume component i, so that the actual rate at which a component is produced or consumed gets

$$rN_i = V \sum_{j=1}^{nr} \nu_{ij} r_j \tag{4.45}$$

where ν_{ij} is the stoichiometric coefficient of component i in reaction j, with positive values for products and negative for reactants.

The mass balance given in (4.5) is repeated here after changing to a molar basis:

$$\frac{d\mathbf{N}}{dt} = \sum_{k}^{n} \dot{\mathbf{n}}_k + \mathbf{rN} \tag{4.46}$$

where the elements of the molar production term **rN** are computed according to (4.45).

Energy conservation in reacting systems has to make sure that the "heat of reaction", or, more precisely, enthalpy of formation is taken into account correctly. There is often a slight misconception about these terms because some software implementations do not make it very clear what type of enthalpy they return from function calls.

The convention in chemical systems modeling is to set the reference specific enthalpy for pure chemical elements in their naturally occurring form at standard conditions (25°C and 1 bar pressure) to zero. This means that di-atomic gases like O_2 have zero enthalpy at standard conditions, elements which do not form such molecules like Fe have zero enthalpy for single atoms. When elements in their natural states react with each

other, the energy of the newly formed chemical bonds differs from the original energy at the same standard conditions. This enthalpy difference is called *enthalpy of formation* or *heat of reaction*. The specific enthalpy that was used in the parts of this thesis dealing with non-reacting flows contains an arbitrary integration constant. The value of that constant for any species does not matter in non-reacting systems, but it can be set to a value that simplifies modeling of reactions.

If the arbitrary constant in the latent enthalpy is taken to be the enthalpy of formation of that chemical component, changes in mass due to reactions will contribute exactly the correct enthalpy of formation that is generated or consumed by the reaction. For reacting flows, the energy balance needs not to be changed because no separate contribution for the enthalpy of formation has to be included.

In conclusion, the energy conservation in chemical reaction systems is taken care of automatically if the enthalpy of formation is included in the latent specific enthalpy. This is the case for the ideal gas property models in the ThermoFluid library. When external property calculations are used with the library, this should be assured in the same way.

Expressions for computing the reaction rate r_j are in general functions of the thermodynamic state, mostly temperature, pressure and composition:

$$r_j = r_j(T, p, x), \qquad j = 1 \ldots nr$$

Usually these functions are empirical or semi-empirical and they rely on substantial experimentation. The uncertainties involved with reaction rates are often considerable, which is one of the main reasons to use the assumption of chemical equilibrium. Chemical equilibrium models are not yet included in the ThermoFluid library.

Implementation of chemical reactions is treated in Section 5.6, the base class for kinetic reactions is included in package BaseClasses.Reactions and an example of a very fast reaction is described in Section 5.9.

4.10 Solid Structures

Models for pipe walls and other solid structures which are in thermal contact with the fluids are simple to treat. Two phenomena are of interest when modeling solids: their capacity to store heat and their resistance to heat transfer. In system level models it is common to have simple models where one of these effects is neglected or the capacity or the resistance of a wall are lumped with the capacity or resistance of the fluid in contact. For example,when measurements are used to estimate the resistance of the separating wall in a heat exchanger, only the total resistance of the

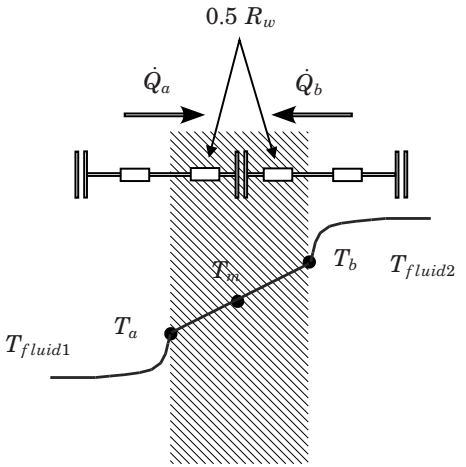


Figure 4.6 Heat resistances and capacitances in a heat exchanger wall.

two fluid boundary layers and the wall can be estimated. Neglecting the lumping of fluid - and wall properties, the following model types for walls are common:

- A wall with negligible mass, without any capacity or resistance just to separate two fluids
- A wall without capacity but including one lumped resistance for the metal and the boundary layers.
- A lumped wall model without resistance with heat capacity.
- A lumped wall model including resistance. For this type of model it makes sense to split the heat transfer resistance of the wall into two equal parts. The temperature at the wall surface is different from the average temperature of the wall.
- Discretized wall models with a sequence of heat capacities and resistances. The discretization is perpendicular to the dominant heat flow direction.

All these models can be discretized in the direction of flow, but they are usually modeled neglecting heat flow in that direction so that the only interaction between the neighboring discretized wall parts is via the fluid. The choice of model is governed by the ratio of the heat capacity parameters of the wall and the fluid in contact. The most common model for

system simulation is a wall model with a lumped heat capacity but without resistance. The heat resistance in the metal is often small compared to the heat resistances in the fluid boundary layers. A simple lumped model which even permits a temperature dependent heat capacity can be written as

$$m\frac{d(C_pT)}{dt} = \sum \dot{Q}. \tag{4.47}$$

The different possibilities are illustrated in Figure 4.6.

- In the case of no wall resistance, the heat flows $\dot{Q}$ are calculated in fluid boundary layer models,
- In the case with resistance only in the metal wall, the heat flow is calculated from the temperature difference $\dot{Q}_a = 0.5R_w(T_m - T_a)$ where the variables are explained in Figure 4.6 and R_w is the total resistance of the wall.
- When the resistances of the boundary layer are neglected, $T_{fluid1} = T_a$ and $T_{fluid2} = T_b$.
- If both resistances are taken into account, the model is described by a system of equations for the heat flows between wall and fluid. The system is linear when the heat transfer coefficient α of the fluid boundary layer is constant.

The common lumped wall models without resistance will be explored further in the next section. A particularly interesting special case is obtained when wall models are combined with a control volume model with neglected heat resistances in boundary layer and metal, see the model reduction example in Section 2.4.

4.11 Moving Boundary Models

The models described in the previous sections are general purpose models for fluid flow using lumped or distributed parameters. The generality of the models makes them well suited for model libraries, but in many situations these models are far from optimal for dynamic purposes. Distributed parameter models are in general not well suited for control design, where low order models are preferable. Steady state oriented models of thermo-fluid systems put large emphasis into getting pressure drop and heat transfer equations correct: the accuracy of those often does not matter very much for control purposes, except when feed forward control is used. A controller with integral action will bring the system to the desired operating point even if the model on which the controller is based is not very

accurate at steady state. Physical phenomena at much higher frequencies than the bandwidth should be neglected altogether for the sake of simplicity. Model accuracy is needed most for the eigenvalues of the linearized system in the vicinity of the desired closed loop bandwidth. The important lesson from modeling of fluid systems for control is that this feature can often be achieved with models which are lower order and simpler than distributed parameter models. Even worse, the standard distributed parameter models are in some cases particularly unsuited for describing the model in the frequency range of the control. Plant design engineers know that distributed parameter models are good at predicting steady state performance but fail to recognize their shortcomings for control design.

A particular example of low order models which are superior to distributed models for control design are a class of models which are usually classified as *moving boundary models* of two phase flows. The main idea of these models is that they make use of the observation that in two phase flows, the physical behavior differs a lot between the liquid single phase, two phase and gas single phase regions. A modeling idea using this observation uses control volumes with variable sizes. These models capture the behavior in the volume and the boundaries of the regions.

Moving boundary models as a low order alternative to distributed parameter models have been used by other authors before, e. g., a model for incomplete vaporization [Beck and Wedekind, 1981], several variants of two region dry expansion evaporator models in [He and Liu, 1998], [He *et al.*, 1997], [He *et al.*, 1994] and [He *et al.*, 1995] and a three region model by Bittanti et al. [Bittanti *et al.*, 2001].

Moving Boundary Equations

The distribution of liquid and gas in a typical evaporator looks approximately like in Figure 4.7. Some technically relevant variants of this general form are:

Once through boilers. The flow patterns are illustrated in Figure 4.7 with subcooled inflow and superheated outflow, but the flow is usually vertical and upward.

Risers in drum boilers. They always have vertical up-flow in the risers and incomplete evaporation with subcooled inflow and two phase flow at the outlet.

Dry expansion evaporators are typical for household and commercial refrigeration systems. The inflow is usually in the two-phase region and the control variable of interest is the superheat, the temperature difference between the outflow temperature and the saturation temperature.

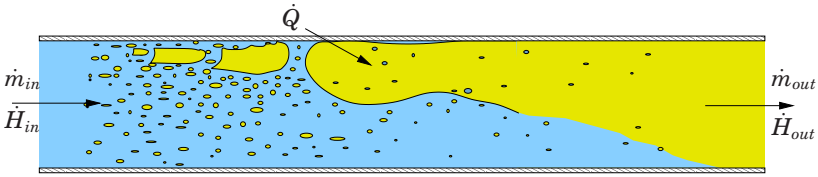


Figure 4.7 Horizontal flow of gas and liquid in an evaporator.

The physical phenomena found in two phase flows are complex and the special literature about it provides many very detailed models with fine-grained classification of flow patterns. For low order models it is natural to choose three regions: subcooled, two phase and superheated. The three model types that correspond to the three applications listed above are:

Three-zone models consist of a subcooled, a two phase and a superheated region.

Two-zone flooded evaporator models have a subcooled and a boiling two phase region.

Two-zone dry expansion evaporators start with two phase flow at the inlet, followed by a superheated region.

The detailed derivation of these models follow the same pattern: the mass- and energy balance equations are integrated over a control volume of variable size.

For the fluid mass balance we start from (4.6) and for the fluid energy balance with (4.15). The energy balance for the metal is also based on (4.15) with the additional assumption of a constant heat capacity. The momentum balances are replaced by a static relation of pressure drop at the outflow because their time constants are outside the bandwidth of interest for control. The result is a set of differential algebraic equations for the boundaries of the regions and for the averages of the variables that characterize the storage of mass and energy in the central volume. For the convenience of reading the next section, notation for the moving boundary model equations is provided in Table 4.2.

To integrate the equations over the central volume it is necessary to make assumptions about the distributions of mass and energy inside that volume. Reasonable distributions are obtained by using essential parameters from a detailed investigation of inhomogeneous, distributed models. The distribution of mass depends on the void fraction, presented in Section 4.12.

Roman and Greek Letters			
A	area	h	enthalpy
C_w	heat cap. of wall	$\dot{m}$	mass flow
t	time	q	heat flux
D	diameter	L	length
v	velocity	z	length coordinate
S	slip ratio	x	mass fraction
α	heat transfer coeff.	ρ	density
γ	void fraction	ω	pump speed
μ	density ratio	Φ	dissipation function
η	liquid fraction	Ψ	vapor generation rate
Subscripts			
1	subcooled	i	inner
2	two-phase	in	inlet
3	superheated	l	saturated liquid
12	interface 1-2	o	outer
23	interface 2-3	out	outlet
amb	ambient	r	refrigerant
g	saturated gas	w	wall
$w1$	subcooled wall	$w2$	two phase wall
$w3$	superheated wall	f	fluid
Double subscripts are used for some flows. q'_{w11} means: heat flux from subcooled wall to region 1 fluid			
Superscripts			
$'$	flux per length	$*$	normalized variable

Table 4.2 Notation for moving boundary models.

By neglecting work terms like viscous stresses, axial conductance and assuming a single heat transfer interaction the energy balance (4.15) can

be simplified to

$$\frac{\partial(A\rho h - Ap)}{\partial t} + \frac{\partial \dot{m} h}{\partial z} = q'_{wf} \tag{4.48}$$

A simplified energy balance for the wall is obtained by setting all convection terms in (4.48) equal to zero, assuming two heat transfer terms and neglecting the axial conductance, hence

$$C_w \rho_w A_w \frac{\partial T_w}{\partial t} = -q'_{wf} + q'_{ambw} \tag{4.49}$$

The heat flows per pipe length, q', can usually be calculated using a constant heat transfer coefficient. For the subcooled wall section this gets:

$$q'_{w11} = \alpha_i \pi D_i (T_w - T_f) \tag{4.50}$$

$$q'_{ambw1} = \alpha_o \pi D_o (T_{amb} - T_w) \tag{4.51}$$

Equations (4.6), (4.48) and (4.49) are the balance equations, which will be integrated over the three regions to give the general three region lumped model for a two-phase heat exchanger, see Figure 4.8. The detailed derivation for the three-zone model is presented below. The derivation of the other models is similar.

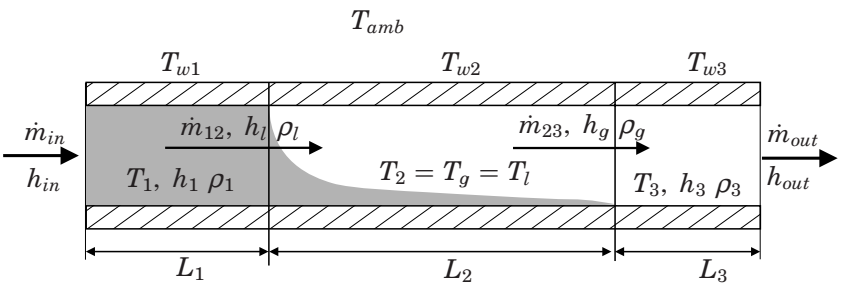


Figure 4.8 Schematic of the three region moving boundary model.

Mass Balance for the Subcooled Region

Consider the three region model outlined in Figure 4.8. This model is representative for a once-through boiler. Integration of the mass balance (4.6) over the subcooled region gives

$$\int_0^{L_1} \frac{\partial(A\rho)}{\partial t} dz + \int_0^{L_1} \frac{\partial \dot{m}}{\partial z} dz = 0 \tag{4.52}$$

Integrating the second term and differentiating the resulting equation gives for a constant area pipe:

$$A\frac{d}{dt}\int_0^{L_1}\rho dz - A\rho(L_1)\frac{dL_1}{dt} + \dot{m}_{12} - \dot{m}_{in} = 0.$$

The density at the interface $\rho(L_1)$ is equal to the saturated liquid density ρ_l. Pressure and mean enthalpy are chosen as the states in the subcooled region. The mean enthalpy is defined as

$$\bar{h}_1 = \frac{1}{2}(h_{in} + h_l)$$

where h_{in} is known from the boundary conditions and h_l is a function of the pressure. The mean density and temperature in the subcooled region is approximated by

$$\bar{\rho}_1 = \frac{1}{L_1}\int_0^{L_1}\rho dz \approx \rho(p,\bar{h}_1) \qquad \bar{T}_1 \approx T(p,\bar{h}_1).$$

Using the above expressions, the mass balance for the subcooled region can be written as

$$A\Big[(\bar{\rho}_1 - \rho_l)\frac{dL_1}{dt} + L_1\frac{d\bar{\rho}_1}{dt}\Big] = \dot{m}_{in} - \dot{m}_{12}. \tag{4.53}$$

The term $d\bar{\rho}_1/dt$ is calculated using the chain rule:

$$\begin{aligned}\frac{d\bar{\rho}_1}{dt} =& \frac{\partial\bar{\rho}_1}{\partial p}\Big|_h\frac{dp}{dt} + \frac{\partial\bar{\rho}_1}{\partial\bar{h}_1}\Big|_p\frac{d\bar{h}_1}{dt}\\ =& \Big(\frac{\partial\bar{\rho}_1}{\partial p}\Big|_h + \frac{1}{2}\frac{\partial\bar{\rho}_1}{\partial\bar{h}_1}\Big|_p\frac{dh_l}{dp}\Big)\frac{dp}{dt} + \frac{1}{2}\frac{\partial\bar{\rho}_1}{\partial\bar{h}_1}\Big|_p\frac{dh_{in}}{dt}\end{aligned}$$

The term dh_{in}/dt is determined from the boundary conditions to the evaporator model. Inserting this expression into the mass balance (4.53), gives the final version of the mass balance for the subcooled region

$$\begin{aligned}A\Big[(\bar{\rho}_1 - \rho_l)\frac{dL_1}{dt} + L_1\Big(\frac{\partial\bar{\rho}_1}{\partial p}\Big|_h + \frac{1}{2}\frac{\partial\bar{\rho}_1}{\partial\bar{h}_1}\Big|_p\frac{dh_l}{dp}\Big)\frac{dp}{dt}\\ + \frac{1}{2}L_1\frac{\partial\bar{\rho}_1}{\partial\bar{h}_1}\Big|_p\frac{dh_{in}}{dt}\Big] = \dot{m}_{in} - \dot{m}_{12}.\end{aligned} \tag{4.54}$$

Energy Balance for the Subcooled Region

Integration of the energy balance (4.48) over the subcooled region gives

$$\int_0^{L_1} \frac{\partial(A\rho h - Ap)}{\partial t} dz + \int_0^{L_1} \frac{\partial \dot{m} h}{\partial z} dz = \int_0^{L_1} q'_{w1l} dz. \tag{4.55}$$

For a constant area pipe and heat flow per length q', integration over the length and subsequent differentiation result in:

$$\begin{aligned} A\frac{d}{dt}\int_0^{L_1} \rho h dz - A\rho(L_1)h(L_1)\frac{dL_1}{dt} - AL_1\frac{dp}{dt} \\ = \dot{m}_{in}h_{in} - \dot{m}_{12}h_l + L_1 q'_{w11}. \end{aligned} \tag{4.56}$$

Using

$$\bar{\rho}_1 \bar{h}_1 \approx \overline{\rho_1 h}_1 = \int_0^{L_1} \rho h dz$$

and expanding equation (4.56) gives the expression for the energy balance of the subcooled region:

$$\begin{aligned} &\frac{1}{2}A\Bigg[\Big(\bar{\rho}_1(h_{in}+h_l) - 2\rho_l h_l\Big)\frac{dL_1}{dt} + \Big(\bar{\rho}_1 L_1 + \frac{\partial \bar{\rho}_1}{\partial h}\Big|_p\Big)\frac{dh_{in}}{dt} \\ &+ L_1\Big\{\bar{\rho}_1\frac{dh_l}{dp} + (h_{in}+h_l)\cdot\Big(\frac{\partial \bar{\rho}_1}{\partial p}\Big|_h + \frac{1}{2}\frac{\partial \bar{\rho}_1}{\partial h}\Big|_p \frac{dh_l}{dp} - 2\Big)\Big\}\frac{dp}{dt}\Bigg] \\ &= \dot{m}_{in}h_{in} - \dot{m}_{12}h_l + L_1 q'_{w11} \end{aligned} \tag{4.57}$$

Superheated and Two Phase Zones

The derivation of the mass- and energy balances for the superheated and two phase regions follow the same pattern. The integrals are expanded, mean values are introduced which are defined by the integrals and derivatives of the density are expanded into derivatives of pressure and specific enthalpy. The details of the derivation are presented in Appendix C. The final results for the superheated region becomes:

$$\begin{aligned} A\Big[L_3\Big(\frac{1}{2}\frac{\partial \bar{\rho}_3}{\partial \bar{h}_3}\Big|_p \frac{dh_g}{dp} + \frac{\partial \bar{\rho}_3}{\partial p}\Big|_h\Big)\frac{dp}{dt} + (\rho_g - \bar{\rho}_3)\frac{dL_1}{dt} + (\rho_g - \bar{\rho}_3)\frac{dL_2}{dt} \\ + \frac{1}{2}L_3\frac{\partial \bar{\rho}_3}{\partial \bar{h}_3}\Big|_p \frac{dh_{out}}{dt}\Big] = \dot{m}_{23} - \dot{m}_{out}. \end{aligned} \tag{4.58}$$

The energy balance for the superheated region reads

$$\begin{aligned} A\Big[&\Big(\rho_g h_g - \frac{1}{2}\bar{\rho}_3(h_g + h_{out})\Big)\Big(\frac{dL_1}{dt} + \frac{dL_2}{dt}\Big) \\ &+ L_3\Big[\frac{1}{2}(h_g + h_{out})\Big(\frac{1}{2}\frac{\partial \bar{\rho}_3}{\partial \bar{h}_3}\Big|_p \frac{dh_g}{dp} + \frac{\partial \bar{\rho}_3}{\partial p}\Big|_h\Big) \\ &+ \frac{1}{2}\bar{\rho}_3\frac{dh_g}{dp} - 1\Big]\frac{dp}{dt} + \Big(\frac{1}{2}\bar{\rho}_3 L_3 + \frac{1}{4}\frac{\partial \bar{\rho}_3}{\partial \bar{h}_3}\Big|_p (h_g + h_{out})L_3\Big)\frac{dh_{out}}{dt}\Big] \\ &= \dot{m}_{23}h_g - \dot{m}_{out}h_{out} + L_3 q'_{w33} \end{aligned} \tag{4.59}$$

The mean properties of the superheated region are calculated in the same way as in the subcooled region. Thus

$$\bar{h}_3 = 0.5(h_g + h_{out}), \bar{\rho}_3 \approx \rho(p, \bar{h}_3) \text{ and } \bar{T}_{r3} \approx T(p, \bar{h}_3).$$

The heat fluxes between the pipe wall and the two phase zone respectively superheated zone are calculated as in (4.50) with the appropriate substitutions in the variables:

$$q'_{w33} = \pi D_i \alpha_{i3}(T_{w3} - \bar{T}_{r3}) \tag{4.60}$$

$$q'_{w22} = \pi D_i \alpha_{i2}(T_{w2} - T_{r2}) \tag{4.61}$$

The flow in the two-phase region is assumed to be at equilibrium conditions with a mean density of $\bar{\rho} = \bar{\gamma}\rho_g + (1-\bar{\gamma})\rho_l$, where the void fraction is defined as $\gamma = A_{vap}/A$. The average void fraction is defined as

$$\bar{\gamma} = \frac{1}{L_2}\int_{L_1}^{L_1+L_2} \gamma dz.$$

The central assumption for the following derivation is that $\bar{\gamma}$ changes much slower in time than the other variables such that it can be treated as a constant when differentiated with respect to time. This precludes using the model for fast pressure transients. A detailed model of the calculation of the void fraction is derived in section 4.12. The mass balance for the two-phase region becomes

$$\begin{aligned} A\Big\{&(\rho_l - \rho_g)\frac{L_1}{dt} + (1-\bar{\gamma})(\rho_l - \rho_g)\frac{dL_2}{dt} \\ &+ L_2\Big(\bar{\gamma}\frac{d\rho_g}{dp} + (1-\bar{\gamma})\frac{d\rho_l}{dp}\Big)\frac{dp}{dt}\Big\} = \dot{m}_{12} - \dot{m}_{23}. \end{aligned} \tag{4.62}$$

and the energy balance for the two-phase region is given by

$$A\Bigg\{L_2\Big[\bar{\gamma}\frac{d(\rho_g h_g)}{dp}+(1-\bar{\gamma})\frac{d(\rho_l h_l)}{dp}-1\Big]\frac{dp}{dt}+\Big[\bar{\gamma}\rho_g h_g+(1-\bar{\gamma})\rho_l h_l\Big]\frac{dL_1}{dt}$$
$$+\Big[(1-\bar{\gamma})(\rho_l h_l-\rho_g h_g)\Big]\frac{dL_2}{dt}\Bigg\}=\dot{m}_{12}h_l-\dot{m}_{23}h_g+L_2 q'_{w2p} \tag{4.63}$$

The derivative of the properties at the phase boundaries are written in a short notation and can be rewritten as e. g., $d(\rho_g h_g)/dp = h_g(d\rho_g/dp) + \rho_g(dh_g/dp)$. Both $d(\rho_g h_g)/dp$ and $d(\rho_l h_l)/dp$ are functions of only pressure because they are on the phase boundary.

Energy Balance for the Wall Regions

Integration of the wall energy equation (4.49) from α to β gives

$$\int_\alpha^\beta C_w\rho_w A_w\frac{\partial T_w}{\partial t}dz=\int_\alpha^\beta \alpha_i\pi D_i(T_r-T_w)dz + \int_\alpha^\beta \alpha_o\pi D_i(T_{amb}-T_w)dz \tag{4.64}$$

Integrating, assuming constant wall properties and rearranging gives the general energy balance for a wall region:

$$C_w\rho_w A_w\left[(\beta-\alpha)\frac{dT_w}{dt}+\big(T_w(\alpha)-T_w\big)\frac{d\alpha}{dt}+\big(T_w-T_w(\beta)\big)\frac{d\beta}{dt}\right] = (\beta-\alpha)q'_{wf}-(\beta-\alpha)q'_{ambw}. \tag{4.65}$$

For the wall region adjacent to the subcooled region $\alpha = 0$ and $\beta = L_1$, which gives

$$C_w\rho_w A_w\left[L_1\frac{dT_{w1}}{dt}+\big(T_{w1}-T_w(L_1)\big)\frac{dL_1}{dt}\right] = L_1 q'_{w1l}-L_1 q'_{ambw1}. \tag{4.66}$$

The wall temperature in the model is discontinuous at L_1 giving

$$T_w(L_1)=T_{w2}\ \text{for}\ \frac{dL_1}{dt}>0$$
$$T_w(L_1)=T_{w1}\ \text{for}\ \frac{dL_1}{dt}\le 0 \tag{4.67}$$

Similar expressions are derived for the walls adjacent to the two-phase and the superheated regions.

Two Zone Models

When the evaporation is incomplete or the fluid inflow is in two-phase conditions, only two zones are needed. The models are similar to the three-zone model above and many of the equations are valid unchanged. The subcooled and superheated zone and their adjacent metal models are identical for the three-zone model and for the two-zone model which has the corresponding zone. The details of the two phase model are different for the three types of models. For the two variants of the two-zone model, the schematic of the three zone model in Figure 4.8 can be imagined to be cut in the two phase zone. The left part of the cut figure corresponds to a flooded evaporator configuration and the right part to a dry expansion evaporator.

The detailed derivation for all the equations of the five different zone types (one subcooled zone, one superheated zone, three different types of two phase zones) needed in the above three models is a tedious exercise that follows the same ideas in all cases: integrate the balance equations over a variable size control volume and write all occurring derivatives in terms of derivatives of pressure and specific enthalpy, because this is the most suitable pair of inputs to medium property calculations. The models are currently integrated into the ThermoFluid library. They can be combined with all medium property descriptions capable of handling two phase flows, currently water, CO_2 and refrigerant R134a. The models are designed as building blocks for low order two phase flow system models. Which model is best suited to a given situation and whether model order reduction as presented in Section 2.4 is appropriate has to be decided separately.

4.12 Void Distribution

Introduction

Fluid flow in evaporators, which typically have a constant cross section area along the flow path, has to be accelerated by a factor equal to the density ratio between liquid and gas. This follows from the continuity equation

$$\dot{m} = \rho w A.$$

Heterogeneous flow means the assumption of equal flow speeds of gas and liquid, heterogeneous flow denotes models with different speeds for the phases.

Neglecting effects from the surface tension and forces due to mass transfer between the phases, the pressure drops along the flow path and is homogeneous in both phases at any given cross section. The pressure drop is the main driving force of the flow. This means that the gas and liquid phases are subject to the same acceleration force between two given points on the flow path. In well-mixed flow patterns, e. g., bubbly flow at the beginning of the boiling zone, the acceleration of the two phases is approximately the same. When the flow is separated like in annular flow, which is the case for most of the length of the evaporator, the much lighter gas phase is accelerated faster than the liquid phase. A graphical representation of this flow features is given in figures 4.9 and 4.10. One consequence of the difference in flow speeds which does have significant influence on the slow part of the evaporator dynamics is the resulting distribution between gas and liquid. This distribution is described by the void fraction along the pipe, $\gamma(z)$, which for the case of lumped parameter models is integrated from the beginning to the end of the evaporation zone. If instead of the complex changes in flow patterns in a real pipe, averaged uniform flow properties are used, the resulting gas-liquid distribution looks like in Figure 4.13, where the heterogeneous flow solution is compared to the most often used assumption of homogeneous flow.

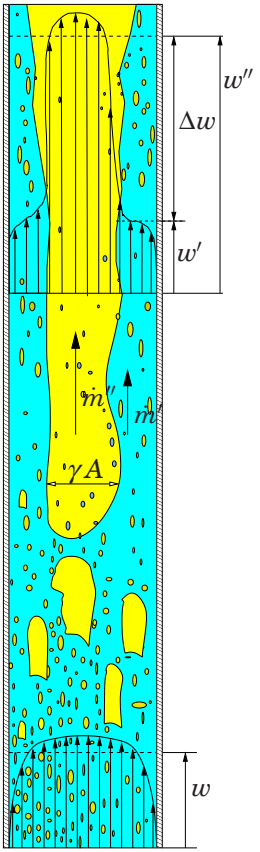


Figure 4.9 Flow pattern in vertical two-phase flow.

Steady State Profile

The void fraction derived in this section is used to obtain a good estimate of the fluid mass in the evaporator. For a given pressure, the total mass depends on the void fraction of the pipe. The void fraction is calculated as the integral of the void fraction profile over the evaporator length. The normalized void fraction profile depends on the velocity ratio between the phases and the pressure. For the derivation of a $\gamma(z)$-profile, a couple of assumptions are necessary:

1. All assumptions for the derivation of the moving boundary model

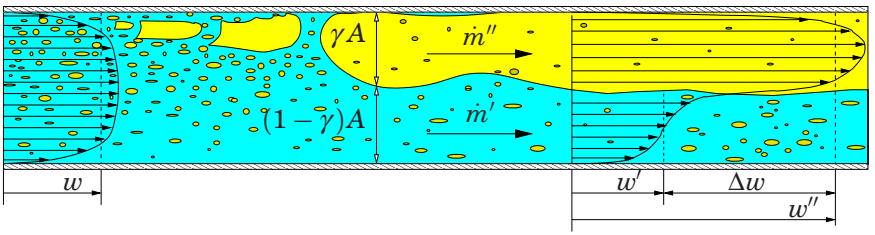


Figure 4.10 Flow pattern in horizontal two-phase flow. The gas moves faster, due to gravity it is accumulated at the top.

also apply to the derivation of the void fraction profile, in particular that the pressure is assumed constant along the pipe.

2. The profile can be evaluated under steady state conditions. For the purpose of slow, start-up transients as well as for linearization purposes this does not pose any restrictions. This means in particular that the pressure is in steady state.

3. The steam generation rate Ψ' is uniform over the evaporator length.

4. The slip velocity ratio $S = u_g/u_l$ between the gas and the liquid velocities is evaluated under steady state conditions. Two cases are treated: (1) S is constant along the evaporator length, giving a symbolic solution of the profile. (2) $S = S(\gamma, \mu)$. Many slip correlations can be transformed to this functional dependence. This leads to a numerical solution of the void fraction profile. A slip velocity ratio different from one distinguishes this model from other moving boundary models in the literature.

The key assumption here is that the profile retains its shape during transients. This excludes sharp gradients in the inflow velocity and large amplitude pressure disturbances. The importance of the void fraction in two phase process dynamics has often been emphasized, compare e. g., the drum boiler model in [Åström and Bell, 2000]. A similar derivation to the one presented here but assuming a slip velocity ratio of one has been published in [Bittanti *et al.*, 2001].

Under the above assumptions, the following coupled ODE boundary value problem holds:

$$\begin{aligned} \rho_l \frac{\partial(A_l u_l)}{\partial z} &= -\Psi' \qquad \text{and} \\ \rho_g \frac{\partial(A_g u_g)}{\partial z} &= \Psi'. \end{aligned} \tag{4.68}$$

Ψ' is the net generation of saturated steam per unit length in $[kg/(ms)]$, A_l and A_g are the cross sectional areas taken up by liquid and vapor respectively and the densities are independent of the length coordinate because we assumed no pressure loss and steady state conditions for the pressure. This equation is normalized by setting $A = A_l + A_g = 1$ and letting the length of the evaporation zone run from zero to one. The cross section area $A_g(z)$ is now equivalent to the liquid volume fraction $\gamma(z)$. Then, replacing u_l with u and u_g with Su and dividing by ρ_l, the following normalized equations are obtained:

$$\begin{aligned} \frac{\partial((1-\gamma)u)}{\partial z} &= -\Psi^* \\ \mu S(z)\frac{\partial(\gamma\, u)}{\partial z} &= \Psi^* \end{aligned} \tag{4.69}$$

where

$$\Psi^* = \frac{\Psi'}{\rho_l A}, \quad \text{and} \quad \mu = \frac{\rho_g}{\rho_l}.$$

The boundary conditions at the length coordinates $z = 0$ and $z = 1$ are

$$\gamma(0) = 0, \qquad \gamma(1) = 1. \tag{4.70}$$

Two slip correlations are going to be investigated more closely. A simple one allowing the symbolic solution of 4.69 and 4.71 and a more complex and realistic one. For complex slip correlations, the void profiles and their integrals have to be calculated numerically.

When the slip S is assumed constant along the pipe, equations (4.69), and the boundary conditions can be solved symbolically to give the following function for $\gamma(z)$:

$$\gamma(z) = \frac{z}{S\mu + z(1 - S\mu)}. \tag{4.71}$$

The influence of the slip ratio S on the void fraction in the evaporation zone, γ can be estimated from the plot in Figure 4.11. A constant slip ratio based on the minimization of total kinetic energy is the one from Zivi, first published in [Zivi, 1964] but quoted from [Whalley, 1987].

Looking at the flow patterns in Figure 4.9 and from physical intuition it is clear that a constant slip ratio is not realistic at the onset of boiling. Initially, the flow speed of the phases will be the same and along the pipe the gas velocity and slip will increase. Only few of the numerous slip correlations are derived to hold for all possible flow patterns. A simple correlation which fulfills this criterion is the one from Levy, [Levy and

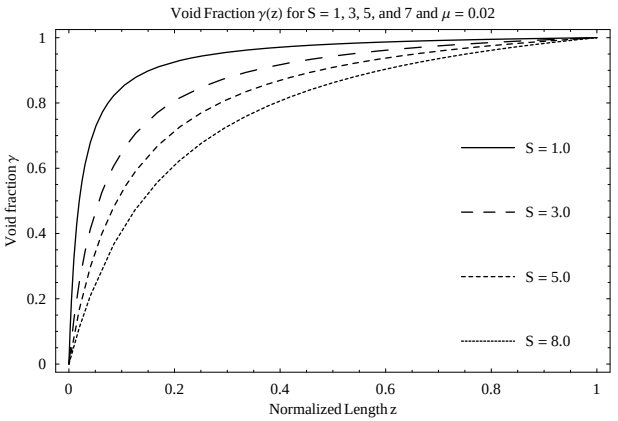


Figure 4.11 Void Fraction $\gamma(z)$ along the normalized evaporation zone for $\mu = 0.02$.

Abdollahian, 1982] as reported in [Wang, 1991]. In its original form it relates void fraction, steam mass flow rate and density ratio, but can be rewritten as a slip correlation:

$$S(\gamma, \mu) = \frac{1 - \gamma + \sqrt{1 + 2\gamma(\mu^{-1} - 1)}}{2 - \gamma\mu}. \tag{4.72}$$

This slip correlation reduces to $S = 1$ at the beginning of of the boiling region and increases monotonically to an upper limit which is a function of μ. Using the slip correlation (4.72) does not allow a symbolic solution of the profile equations, but it is straightforward to find a numerical solution for a fixed μ. The influence of either choosing a constant slip ratio or a variable slip ratio as the one in (4.72) on the void fraction profile is not large. Figure 4.12 gives an indication of the typical influence of the different slip ratios. The plot is at the maximum difference of the integrated void fraction using the correlation of Zivi (independent of γ) and the correlation of Levy.

One important conclusion that can be drawn from the more realistic Levy slip correlation is that the cases of evaporators with incomplete evaporation – typically between 5 % and 20 % of the total mass flow evaporate – and dry-expansion evaporators with two-phase inflow have very different slip ratios for the same pressures. For incomplete evaporation and low outlet steam qualities the slip ratio is so close to one that the slip influence can safely be neglected.

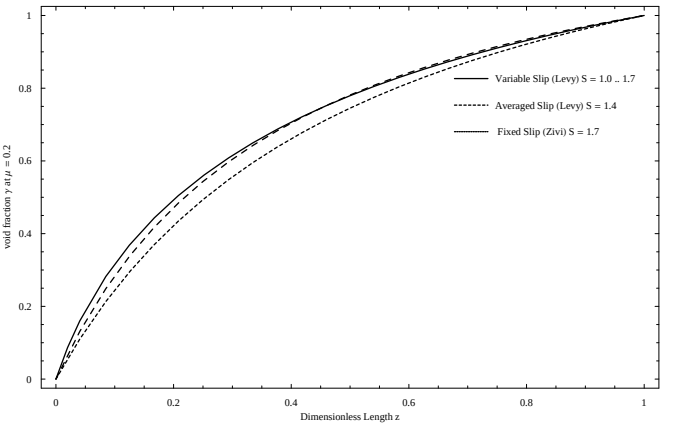


Figure 4.12 Comparison of void fraction profiles for fixed and variable slip correlations.

Average Void Fraction

The average void fraction $\bar{\gamma}$ is computed as the integral over the normalized profile along the pipe. In the case of constant slip S, $\gamma(z)$ can be integrated symbolically to give:

$$\bar{\gamma} = \int_0^1 \gamma(z)dz = \frac{1 + S\mu\,(ln(S\mu) - 1)}{(S\mu - 1)^2}. \tag{4.73}$$

This $\bar{\gamma}(p, S)$ can only be used together with the dynamic model from the previous section when its time derivative, $d\bar{\gamma}/dt$, can be neglected. This holds for slow pressure transients. Slow means here that transients from the momentum balances for gas and liquid, which are the origin of the velocity slip, relax on a faster timescale than the transient of interest.

The density ratio $\mu(p)$ is a simple function of the pressure, but for the slip ratio S many empirical correlations are available. For a closed symbolic solution, a slip ratio which is independent of the local void or steam mass fraction has to be chosen. A simple and appealing correlation is the one from Zivi (1964), cited from [Whalley, 1987]. It minimizes the total kinetic energy flow along the pipe:

$$S = \frac{u_g}{u_l} = \left(\frac{\rho_l}{\rho_g}\right)^{1/3} = \mu^{1/3} \tag{4.74}$$

Using this slip correlation, the average void fraction in the pipe becomes a function of the density ratio $\mu(p)$ which is a function of pressure. Inserting

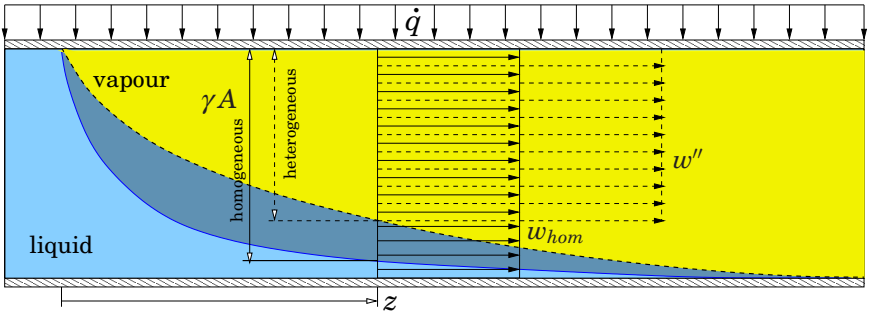


Figure 4.13 Approximate gas-liquid distribution in a pipe

$S = \mu^{(1/3)}$ into (4.73),

$$\bar{\gamma}(p) = \int_0^1 \gamma(p,z)dz = \frac{\left(\frac{1}{\mu}\right)^{2/3}\left(\left(\frac{1}{\mu}\right)^{2/3} - 1 - \frac{2}{3}\,ln\left(\frac{1}{\mu}\right)\right)}{\left(\left(\frac{1}{\mu}\right)^{2/3} - 1\right)^2}. \qquad (4.75)$$

With Levy's slip correlation it is not possible to solve the boundary value problem given by (4.69) symbolically. But it is possible to obtain a numerical representation of the average void fraction $\bar{\gamma}$ with the following steps:

- Create a sufficiently fine grid of physically realistic values for the density ratio μ, e. g., from 0.005 to 1.0.
- For all fixed μ, solve (4.69) with the slip correlation (4.72) numerically, obtaining a numerical profile in the form of pairs of numbers $(z, \gamma(z))$ with z in the interval $(0, 1)$ for each μ.
- Integrate numerically over all profiles to get

$$\bar{\gamma}(\mu) = \int_{z=0}^{z=1} \gamma(z,\mu)dz.$$

- Approximate the pairs of numbers $(\mu, \bar{\gamma}(\mu))$ obtained by integration with an analytic function. Rational function approximations work well in this particular case.

This procedure has been followed using the combined symbolic and numerical tool Mathematica [Wolfram, 1990]. Using one of the built-in optimization tools for rational function approximation, the following function

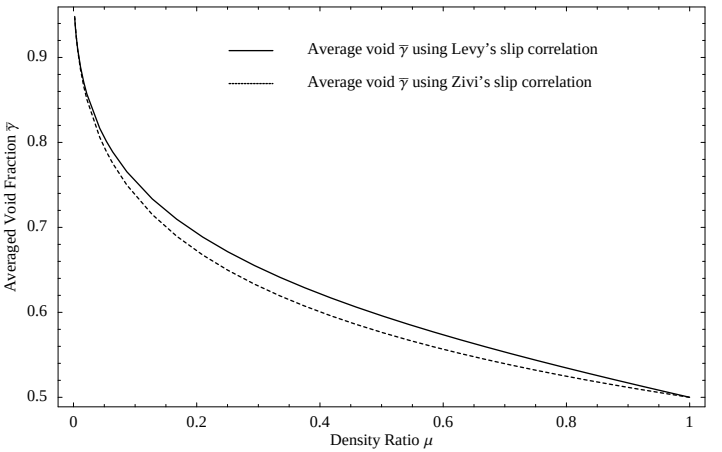


Figure 4.14 Difference of the average void fractions obtained from Levy's and Zivi's slip correlation.

can be obtained for $\bar{\gamma}(\mu)$[9]:

$$\frac{0.98582 + 567.884\,\mu + 20924.2\,\mu^2 + 53923.6\,\mu^3}{1 + 612.285\,\mu + 26266.8\,\mu^2 + 98848.6\,\mu^3 + 17480.6\,\mu^4 + 7596.18\,\mu^5}$$

The deviation of this approximation from any of the densely grided numerical function values is less than 0.01 %. The deviation from the simpler average void $\bar{\gamma}(\mu)$ using the simpler slip correlation from Zivi is less than 3.5 % For lower density ratios the difference is negligible, see Figure 4.14.

[9]Only the six most significant digits are shown

5

The ThermoFluid Library

Abstract

This chapter describes the current state of the ThermoFluid library, formerly called ThermoFlow library. The library is still under active development. Where appropriate, possible improvements over the current status will be pointed out. Finally, some industrial and academic examples that use the ThermoFluid library are briefly presented.

5.1 Introduction

There are Modelica libraries for mechanical systems, electrical systems, block diagrams and basic mathematical functions. A base library for modeling and simulation of thermo-fluid systems has been a missing extension to expand the range of applications for Modelica. A thermo-hydraulic base library should cover the basic physics of flows of fluids and heat transfer. It also needs to cover models for properties of fluids like water, air, important technical gases and refrigerants. The original ThermoFlow library was not designed to handle chemical reactions, but the object-oriented design made it possible to add reactions without changing the existing models. The library has been successfully applied in several application areas, e. g., power generation plants, fuel cell systems, steam distribution networks and refrigeration systems. The applications have been made by both academic and industrial groups.

s The general goal of the library is to provide a framework and basic building blocks for modeling thermo-hydraulic and process systems in Modelica. For obvious reasons it is impossible to provide model components for every conceivable application for this class of systems. The primary goal of the library is to provide a base library with common model parts without limiting the freedom of the user to extend and adapt the library for a particular application. For the same reason, more emphasis

has been put on the basic parts of the library, such as physical property models and control volumes, than on components for all possible applications. The focus of the library is on models of compressible, homogeneous one- and two-phase flows[1]. Incompressible flows are simpler to deal with and are also not part of ThermoFluid, a library for incompressible flow is described in [Fabricius and Badreddin, 2002]. The models in the library are designed for system level simulation, not for detailed simulation of components, which is usually done in CFD packages. The models are thus discretized in one dimension or even lumped parameter approximations.

Initially, designing models for reuse takes considerably more time than use-once models, but the pay-back comes quickly, often after the second related modeling task. It has to be said that designing models for reuse adds complexity to the modeling process and for model development it is often a better approach to first develop and test the model and when it works, is tested and documented it should be refactored into parts which are useful for reuse in a library. This was how the ThermoFluid library developed: the models have been known and used for many years. Based on the experience from long time model use the task was then to structure equations, variables and components in such a way that the resulting building blocks are handy for as broad a range of modeling tasks as possible.

To make the library general and extensible, the design must accommodate different choices of fluid properties, single- or multi component fluids, one- or two phase flows and constitutive equations. 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. The library offers only a limited selection of assumptions, which nonetheless should cover a broad range of applications. The most fundamental of these assumptions is a singular perturbation applied to the full conservation equations, as described in Section 2.1. The momentum balance is reduced to an algebraic equation because it typically evolves on a faster timescale than the other equations. This assumption is far-reaching for the library design, because the flow of momentum in the full, dynamic momentum balance requires momentum flow as a variable in the connectors.

For the ThermoFluid library it was decided to neglect all kinetic energy terms, as discussed in Section 4.3. The decision is based on the order of magnitude of the kinetic terms in the typical applications that ThermoFluid was designed for. This has a number of consequences for the type of applications which can be tackled using the library: the models will obey the basic assumptions taken for the base models and if these are not

[1]Non-homogeneous two phase flows and multi-phase flows are not covered.

library	files	lines of code	packages	models
Modelica	33	22165	29	258
Electrical	19	6145	9	78
MultiBody	19	7451	8	96
ThermoFluid	89	31251	98	1084

Table 5.1 Statistics for various Modelica libraries. Note that the Electrical library is a sub-library of the Modelica Standard Library.

justified, new models have to be derived. In this context it is very important to realize one of the features of object oriented modeling: coupling models which were derived based on different assumptions will result in the violation of one of the conservation laws. In some cases the order of magnitude of the terms that violate the conservation equations is smaller than typical numerical errors, in others they may be of significant size.

This chapter focuses on the object-oriented structure of the library and examples that show how the library can be used. The underlying physical models are documented in Chapters 4, B and C. This reflects also the work process when creating model libraries. A modeler must be familiar with the underlying physical phenomena and the key components of the library. Only with a clear "mind map" of the models it is possible to identify elements of the library that are useful in many different contexts.

Some ideas for previous versions of the thermo-hydraulic base library have previously been presented in [Tummescheit and Eborn, 1998; Eborn *et al.*, 1999; Tummescheit, 2000a; Tummescheit, 2000b]. Object-oriented component based modeling in various modeling languages has been discussed in [Wagner, 2000; Mühlthaler, 2000; Nilsson, 1993].

There are several issues which make modeling of thermo-fluid systems different to modeling in other engineering domains. The subproblem of fluid properties is complex by itself: 50 % of the code in ThermoFluid deals with fluid properties, see the statistics in Table 5.1. Many variants of model assumptions are possible and commonly used. The MultiBody library is more complete for the modeling of multi body systems than the ThermoFluid library is for thermo-fluid systems but it is only ≈ 25 % of the size, taking lines of code as the measure. This is mainly due to two facts: complex property functions and many model variants.

There is considerable public interest in the library. On average there are about 70 downloads per month since the library has been published on the Internet. The library has been used in a number of applications

by users from industry and academia. Feedback from the users has influenced the development of the library.

5.2 Basic Ideas

In the design of a model library, a few central decisions have to be made early in the work. Compromises are inevitable which means that there are solutions which differ in some details but cover approximately the same applications.

- Flexibility vs. Complexity. A library exclusively built for drag-and-drop flow sheet modeling is simple to use. A more flexible library, allowing for user choices and modifications, is inevitably more complex but can cover much wider applications.
- Small, specialized vs. broad scope. The trade-off is similar to the above. Large libraries require more learning time than small compact ones.
- Numerical efficiency vs. simplicity of the models. Symbolic methods for model transformations can not always find the numerically best form of the equations. When the transformation to a numerically favorable form is done by hand, the generality of the model is often reduced.

The ThermoFluid library was initially intended to be flexible with a broad scope. This leads to too large complexity for occasional users or for small projects. Several applications indicated that a hierarchical structure with more specialized application libraries built on top of ThermoFluid is useful for larger projects. For a particular application, it would be better to have a small, high level application library that is built on top of ThermoFluid.

The basic design principles of the ThermoFluid library are:

- One unified library both for lumped and one-dimensional distributed parameter models,
- Models should be valid for the complete operating range including start-up and shut-down sequences whenever possible. This requires support for bidirectional flow in all models.
- High level parameterization of all constitutive equations in flow and process models. This comprises fluid property submodels, pressure loss and heat transfer correlations and also reaction rates in reaction submodels.
- initialization procedures are provided for all standard cases, but users can easily build models with customized initialization procedures by calculating parameters at initial time.

- Common alternatives for assumptions (e. g., influence of gravity) can be selected from the user interface.

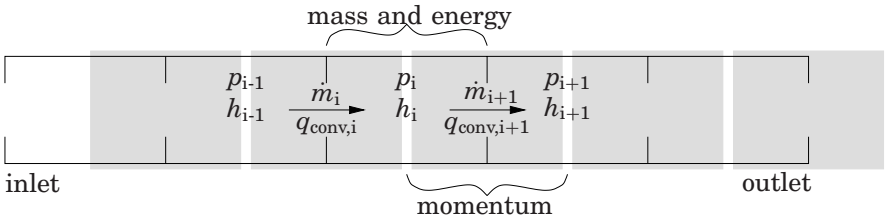


Figure 5.1 Staggered grid discretization. The control volumes for mass/energy and momentum are translated relative to each other. The variables p_i, h_i denote pressure, specific enthalpy in volume i, and $\dot{m}_i, q_{conv,i}$ mass and energy flow between volumes i-1 and i.

The first guideline constrains the discretization method used in the distributed parameter models. Only the so-called "staggered grid" method, illustrated in Figure 5.1, gives a useful model in the lumped parameter case. In this method, [Harlow and Welch, 1965], all fluxes are calculated on the border of a control volume and the intensive quantities are calculated in the center of a control volume. The method is a special case of the finite volume method, [Patankar, 1980], which is commonly used for one-dimensional discretizations. In the standard implementation that is used in the ThermoFluid library there is one disadvantage: the spatial derivatives are only accurate to first order. This is often sufficient for system simulation. Due to the modular character of the library the discretization can easily be refined if this high precision is required.

It is interesting to note that a lumped approximation of the finite volume method is used in almost all commercial simulation packages that deal with fluid flow or process simulation. The only difference in process simulators is that usually the upstream component is used to calculate the flow into the next process. This is equivalent to combining one control volume type model with one flow model.

The ability to handle reversing flows requires extra information in the connectors. This would not be necessary for infinitesimally small control volumes but is caused by two properties of convective flows:

- Convection-dominated processes are inherently asymmetrical. The properties at any given point are dominated by the properties upstream of that point. This is a fundamental difference to diffusion processes which are symmetrical.

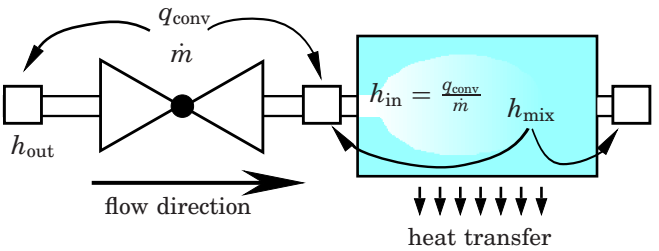


Figure 5.2 Illustration of the properties at connectors for bidirectional flow. In the presence of heat transfer, the average properties which appear in both connectors of a volume can differ a lot from the properties of the inflow stream. This is a property of the ideal mixing assumption of lumped control volumes. The inflow enthalpy $h_{in} = h_{out}$ is never calculated because the energy balance uses q_{conv}. The convected heat flow q_{conv} is calculated as $q_{conv} = (h_{out} + \Delta h)\dot{m})$. For isentropic throttles, $\Delta h = 0$. The properties in the connectors depend on the *location*, not on the flow direction. This is in contrast to usual conventions in flow-sheeting simulators, but is an unavoidable consequence of models capable of handling reversible flows. The model code is given in Section 5.4.

- Spatial discretization of a PDE needs to take the asymmetry into account. This can be done either by an asymmetric discretization scheme in the case of Finite Difference Methods or by integration over a finite size control volume in the case of Finite Volume Methods.

In the Finite Volume Method, transported properties such as enthalpy and composition have to be included twice in connectors in some way. Depending on the flow direction, either the upstream or the downstream properties have to be used in the balance equations.This has been achieved by including convective heat flow and component mass flows in the connectors. The information needed for the balance equations is thus contained in variables that depend on the flow direction, i. e. mass flow and convective heat flow. The transport properties in the connector are always taken from the closest control volume, see Figure 5.2.

The choice of state variables has a significant influence on numerical efficiency. For single component flows different pairs of state variables are possible, e. g., $\{p, h\}, \{p, T\}$ or $\{\rho, T\}$. The state variables can be chosen to fit the purpose of the model. The basic entity of the library, the control volume, is constructed by multiple inheritance from four parts.

- The balance submodels define all interfaces and contributions to the mass and energy balance. The heat- and mass transfer object in the balance submodel makes it possible to add an arbitrary number

of heat transfer areas or mass transfer phenomena to the model without changing the basic equations, see in detail in Section 5.6.

- The partial thermal model contains dynamic state equations derived from conservation laws of mass and energy.
- The partial hydraulic model contains the mass flow equation that is defined as either a static or a dynamic momentum balance.
- The last part of a control volume model are the initial condition specifications.

This composition is detailed in Figure 5.5 and Section 5.4. The property calculations form an exchangeable submodel in the partial thermal model. The details of the physical models are presented in Section 5.6.

5.3 Control Volumes and Flow Models

Lumped approximations of thermo-fluid systems use two types of abstractions which work well for a large class of real process equipment. *Large volume* equipment is modeled by storage of mass and energy in a control volume. Equipment with *small volumes* but high power densities like pumps, turbines, valves, short pipes and orifices have a negligible mass and energy storage but often large changes in pressure and sometimes kinetic energy. Large volume equipment is of *control volume* type, small volume equipment is of *flow model* type. This separates the basic conservation equations into two model types: the dynamic mass- and energy balances are modeled in control volume models and the quasi steady state or dynamic momentum balance is modeled in flow models. These abstractions have traditionally been used in all black-box simulation packages for thermo-fluid systems. Traditionally the models have to be used in an alternating sequence for several good reasons:

- Combining two volume models without a flow model in-between leads to an index two DAE problem, compare Section 2.1. Automatic index reduction procedures can handle this, but this would require differentiable property calculation routines which are usually not available. They are also very cumbersome to implement. In contrast to mechanical systems where the introduction of a DAE-index through couplings is natural, direct coupling of two control volume type models is a sign of poorly chosen subsystem boundaries. One volume model with the sum of the volumes is a simpler and better solution.

- Combination of two flow models can lead to a rather unpleasant non-linear system of equations. This is not a big problem but it reduces robustness unnecessarily. If flow resistance parameters are going to be identified from plant data, flow models in series lead to over-parameterized models and should be avoided for that reason.

For these reasons the ThermoFluid library follows the same convention of alternating flow and volume models even if models in Modelica could be written so that they can be combined in an arbitrary order. Disregarding this rule leads by design to an error. If necessary, two flow models can be coupled directly by including a zero volume control volume between the flow models.

Distributed models using the finite volume method have the property of alternating flow models and control volumes by definition, this follows from the derivation of the method, see [Patankar, 1980]. For a static momentum balance the situation is completely identical to alternating control volume and flow models, but for a dynamic momentum balance there is a slight conceptual difference. The momentum balance is also formulated for a control volume, but the control volume is staggered by half a grid distance with respect to the control volumes for the mass and energy balance. Half of the fluid mass of the upstream and half of the downstream control volume of the flow channel are used in the momentum balance. A visual representation of the staggered grid discretization is shown in Figure 5.1.

Static flow models are prevailingly used in system simulation where the thermal behavior is the main concern. The dynamic momentum balance is useful for pressure wave propagation studies in a system which is mainly modeled with distributed models. It is possible to add other types of flow models to the existing structure, e.g., a momentum balance for variable cross-sectional area along the flow channel.

The different kinds of lumped and distributed models in the ThermoFluid library are shown in Figure 5.3. In order to guide users to follow the rule of alternating models, the two connectors are visually different. Always connecting different connectors guarantees that the simulation problem is well specified and that unnecessary non-linear equation systems are avoided.

When systems with very large control volumes are modeled, e.g., simple models of steam distribution systems, it is advantageous to replace very small volumes with a zero-volume model in order to avoid numerical stiffness. From the computational viewpoint these act like real volumes. Zero-volume models compute the state variables of ordinary control vol-

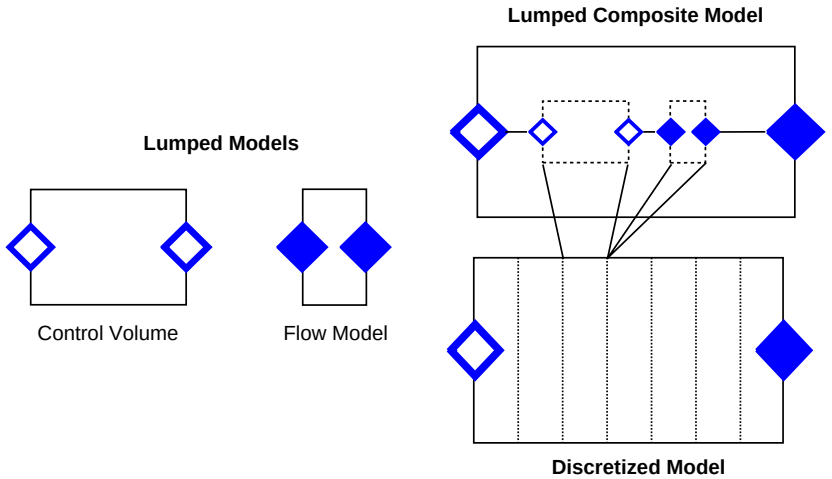


Figure 5.3 Lumped, composite and discretized model objects. Lumped composite models are composed from an alternating sequence of control volume and flow models in a container model. The equations in a discretized model are interleaved control volume and flow model equations. Identical boundary conditions are needed for lumped composite and distributed models: flows at the outlined and potentials at the filled connectors.

umes algebraically from the conditions

$$\frac{dM}{dt} = 0, \qquad \frac{dU}{dt} = 0.$$

5.4 Object-Orientation in ThermoFluid

The basic concepts of object-oriented modeling languages have been treated in Chapter 3. In this section we will give examples of how these concepts are applied in practice in the ThermoFluid library.

The main idea of the ThermoFluid library is to provide an extensible basis for a robust thermo-hydraulic component library. The library is divided into five main parts:

Interfaces define the types of connectors used in the library. The flow connectors are of two different types; either with a static or dynamic flow description. For easy accessibility and inter-operability between different Modelica base libraries, the Interfaces package is always directly beneath the top level package.

Icons define the graphical appearance and the positioning of connectors for the classes in the Components package.

BaseClasses are the central part of models, the basic physical equations for a control volume: balance equations, state transformations and medium models. This is by far the largest package in ThermoFluid. All base classes are abstract and only the functions are usable "as is", without further programming.

PartialComponents contain common code for component models, they allows code sharing and simplifies maintenance. Partial components are composed from base classes. While base classes are building blocks, partial components are almost complete models where few building blocks are missing.

Components are the user part of the library, models that can be used to build a system for simulation by graphical composition. Some components are composed of simpler components, they form a hierarchy of models and subsystems by themselves.

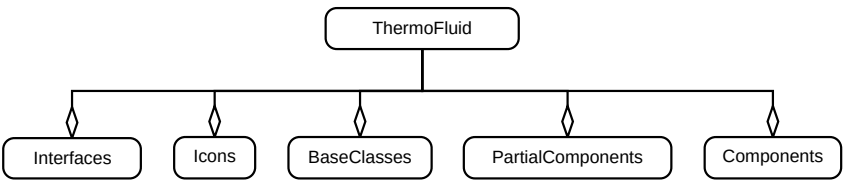


Figure 5.4 Top level package structure of the ThermoFluid library. The graphical symbols are explained in Appendix A.

These parts are illustrated in Figure 5.4. The ThermoFluid library makes systematic use of object-oriented structuring concepts in order to achieve reuse on the level of basic physical phenomena. This can be compared with traditional flow-sheet simulation packages that provide reuse on the level of engineering components like pumps and heat exchangers. Partial components and base classes offer two additional levels of model reuse on a more fine-grained level. The structuring mechanisms that are used in ThermoFluid are the same as in object-oriented programming:

Generalization is a strategy to handle complexity by classification of models and code sharing. Modelica offers

- inheritance and
- parameterization of generic class

as important concepts for code reuse.

Decomposition is a divide-and-conquer strategy to deal with complexity. In Modelica, decomposition can be achieved in two ways, either using

- multiple inheritance or
- component aggregation.

The language tools are not completely orthogonal to the concepts, multiple inheritance can also be classified as partial generalization and class parameterization is also applicable to components. In the following subsections we will give examples of how these features are applied in the ThermoFluid library. A critical discussion of the experiences with object-oriented modeling in Modelica and an attempt to deduce general guidelines is found in Chapter 6.

Generalization

Inheritance A simple example of generalization is to identify common sets of variables and interfaces. Also some aspects of behavior are common for a large class of quite different models. A general feature for flow equipment is the bi-directional convective heat transport, which can be expressed as

```
partial model FlowModelBase
  extends FlowVariables;
  Boolean a_upstream "flow direction: true if positive at port a";
  extends TwoPort;
equation
  // for static flow, a_upstream depends on p_2 − p_1
  // for dynamic flow, a_upstream depends on sign(ṁ)
  // the equation for a_upstream is deferred to a derived class.
  a.q_conv = if a_upstream then mdot*a.h else mdot*b.h;
end FlowModelBase;
```

Notice that the specification of the flow direction, `a_upstream`, and the mass flow, `mdot`, is postponed until later. For quasi steady state flow, the value of the Boolean variable `a_upstream` should not be based on the sign of the mass flow because `a_upstream` and `mdot` would end up in a mixed Boolean-Real equation system. Since the calculation of the mass flow depends on the type of flow equipment used, this additional information has to be provided in a derived class. For example, a quasi steady state flow resistance with a linear expression for pressure losses can be derived as:

```
model LinearOrifice "linear pressure loss − mass flow correlation"
  extends FlowModelBase;
equation
  a_upstream = a.p > b.p; //steady state flow
  mdot = mdot0/dp0*(a.p−b.p);
end LinearOrifice;
```

The mass flow depends on the parameters `mdot0, dp0` and the pressure difference over the valve.

The selection of equations that are included in a base class has to be done very carefully because equations can not be replaced or altered. The only way to change equations is when the equation is to encapsulate it in a replaceable component and replace the component.

Class Parameterization Class parameterization is a form of generalization due to the restrictions on replacement classes, compare Chapter 3. *Type compatibility* enforces the constraining class to be a generalization of all classes that can be supplied as a replacement class.

As an example for class parameterization in ThermoFluid we take the `FlowModel` from Section 5.6. Any flow model needs a loss model for the frictional pressure drop. In order to make the class `FlowModel` as general as possible, only a generic flow model is specified during base class implementation.

```
partial model FlowModel "flow model with generic pressure loss"
  replaceable model
    PressureLoss = GenericPressureLossModel;
  extends PressureLoss;
  ...
end FlowModel;
```

The `GenericPressureLossModel` does not have to contain any equations, but for practical reasons (and as a base class for inheritance in specialized pressure loss models) it contains the variables that are common to all pressure loss models.

In specialized classes, the generic pressure loss model is then replaced by a more adequate model, which contains the equation(s) for computing the actual pressure loss.

```
model BlasiusPipe "Pipe using Blasius' pressure loss correlation"
  extends TwoPort;
  extends Balances;
  extends FlowModel(redeclare PressureLoss = BlasiusPlossModel);
end BlasiusPipe;
```

Strictly speaking it is not necessary to use a *replaceable base class* for the pressure loss as above, it would be equally possible to use a *replaceable component*. In Modelica, this looks as follows:

```
partial model AlternateFlowModel "replaceable component flow model"
  replaceable BlasiusPloss ploss extends GenericPressureLossModel;
end AlternateFlowModel;
```

In the alternative form of `AlternateFlowModel` a component `ploss` of type `BlasiusPloss` is declared. Possible replacements are constrained to be subtypes of `GenericPressureLossModel`. The functionality is the similar as in `FlowModel`, with two differences:

- The `AlternateFlowModel` has a *default class* `BlasiusPLoss` and replacement objects are constrained to be type compatible to `GenericPressureLossModel`, the *constraining class*. The `GenericPressureLossModel` is a **partial model**, therefore classes using `FlowModel` can not be instantiated without redeclaring the pressure loss model. `AlternateFlowModel` can be used directly because of an instantiable default class.
- Variables inside the ploss-component in `AlternateFlowModel` have to be accessed by dot-notation, e. g., as in:
  `ploss.dp = ploss.k*ploss.mdot^2`
  This makes equations more difficult to read.

The last point, enhanced readability of equations, is one reason to use replaceable base classes similar to the `FlowModel` example above. This example also illustrates that the language means for decomposition and generalization have some overlap.

Decomposition

Decomposition can be achieved by declaring instances of models in a compound model or through multiple inheritance. Both types of decomposition are used in the central elements of ThermoFluid, control volumes and flow models. Composition of compound models from submodels is also called *aggregation*. In ThermoFluid this is mainly used as assembly of physical phenomena into a complete model. In order to demonstrate decomposition with a larger example, the models for assembling a distributed pipe are presented.

Model Structure of a pipe A control volume formulation of a pipe uses all code structuring means described above. It can be decomposed into the following individual parts:

Balance equations take care of all mass- and energy interaction of the volume.

Thermal state equations use the mass and energy balances to define dynamic state equations and include a replaceable property model.

A flow model formulates the momentum balance and includes a friction pressure loss correlation.

Initialization code that defines typical initial conditions.

An heat transfer law. Heat transfer is optional because a simple pipe is often modeled as adiabatic.

A geometry record. The pipe models in system level models are often abstractions from more complex flow geometries. The geometry has to be parameterized in such a way that the simplified model parameters can be approximated.

Decomposition makes use of multiple inheritance and composition from parts. For a pipe model this is illustrated in Figure 5.5. The base class for all types of pipes is a vectorized sequence of control volume and flow models which inherits from four base classes. Components are symbolized by smaller blocks inside the puzzle bits. The darker top level class inherits from one partial component and adds a geometry parameterization and a heat transfer law as components. The heat transfer law connects to the `HeatAndMass` object via a `HeatFlow` connector.

These parts are modeled individually using the following classes:

```
partial model TwoPort "base class for models with two flow interfaces"
  parameter Integer nspecies(min=1) "number of chemical components";
  Interfaces.FlowA a(nspecies=nspecies) "design−inflow";
  Interfaces.FlowB b(nspecies=nspecies) "design−outflow";
end Balances;
```

```
partial model TwoPortLumped "mass− and energy balances"
  extends TwoPort;
  extends ThermoBaseVars(n=1) "variable definitions";
  replaceable HeatObject heat(n=1) "heat transfer object";
  equation
    // contributions to mass− and energy balances
    dM_x[1, :] = a.mdot_x + b.mdot_x;
    dU[1] = a.q_conv + b.q_conv − p[1]∗der(V[1]) + heat.Q_s[1];
    dM[1] = sum(dM_x[1,:]);
  ... // further code omitted
end Balances;
```

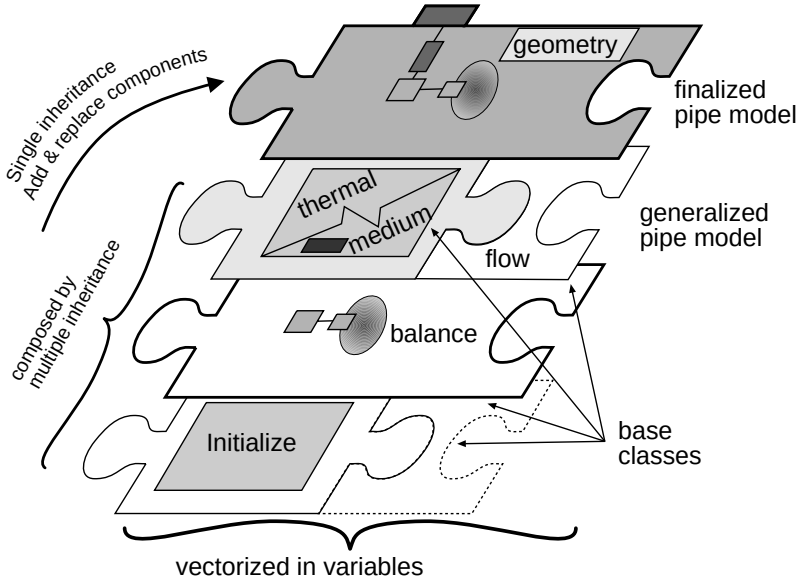


Figure 5.5 Composition of a generalized pipe model in ThermoFluid. The three bottom levels are composed via multiple inheritance in a control volume class. This is specialized in a next step to a concrete pipe model, specifying the type of fluid, geometry and heat transfer mechanism. Initialization can be seen as a *mixin class*, defining a part of the behavior more precisely. Initialization of the flow model is only needed for a dynamic momentum balance.

```
partial model ThermalModel_xy "state equations for example states x and y"
  replaceable model Medium = CommonRecords.StateVariables_xy;
equation
  der(x) = A*dM + B*dU + C*der(V); //real coefficients depend on states
  der(y) = D*dM + E*dU + F*der(V);
  ... // further code omitted
end ThermalModel_xy;
```

The state variables x and y are for illustration, the pairs which are actually implemented are presented in Section 4.5, where also the coefficients A–F of the state transformation are given.

```
partial model Initialization_xy "initialization"
  parameter Integer n(min=1) "number of cv's";
  SIunits.Pressure[n] p(start=init.p0, fixed=false) "pressure";
initial equation
  der(p) = zeros(n);
```

```
  ... // further code omitted
end Initialization_xy;
```

The variables that are chosen as states are often known at initial time and they tend to give well-behaved equation systems. It is convenient to relate the initialization to the states, but this is not a requirement.

```
partial model FlowModel
  extends TwoPort;
  dz*der(mdot) = dG + (a.p − b.p)*A − dp*A;
  ... // the momentum balance
end FlowModel;
```

and aggregation of these base classes leads to a general description of a control volume, e. g., a pipe

```
partial model Pipe_xy "base class of pipe models with states x,y"
  extends Balances; // heat transfer connector etc.
  extends ThermalModel_xy; // the dynamic states
  extends Initialization_xy; // covers typical initial conditions
  extends FlowModel; // contains a replaceable pressure loss model
end Pipe;
```

By the Modelica keyword **extends** the new model Pipe_xy *inherits* the attributes of all base classes. Common parts of the base classes are only inherited once. The parts which are inherited multiple times have to be identical, which is important due to the possibility of modifications of base classes along the inheritance paths. This may lead to the unpractical consequence that identical modifications have to be applied to several of the inherited classes in order to render them identical.

Some examples have shown how object-oriented constructs of the Modelica language are applied in ThermoFluid. The constructs are used throughout the library structure to facilitate code sharing and make the library more flexible. In Chapter 6 it is investigated in more detail how to apply Modelica's language features to the design of model libraries.

5.5 Interfaces

Interfaces between submodels are fundamental in the design of a library. The choice of the interfaces is a crucial step in the design of a reusable model library. Some of the requirements for interfaces originate from the underlying mathematics, others follow from software engineering considerations. For domains which deal with only one type of energy transport there are widely accepted solutions of interface variables. Usually, this

is a potential or an "across" variable and one flow or "through" variable whose product gives the power flow at the interface. This solution is the simplest possible interface which fulfills the requirement that energy is conserved in the component. Domains which deal with more than one type of energy or flows based on interacting potentials and with mass transport have more choices, but the following guidelines can be useful:

- It must be possible to express correct boundary conditions for the model. This requirement is clear for PDE models but it is equivalent for ODEs derived from PDEs describing conservation laws.
- Interfaces should in general be as small as possible. They should not carry more information than necessary. Sometimes this may conflict with minimizing numerical effort when variables have to be calculated in two components instead of one.
- interface variables should have good physical interpretations that are familiar to the user. They can be measured variables and are often related to boundary conditions.

The guideline to use the across variable with potential character and the corresponding through variable breaks down for heat conduction when the last guideline is used. The correct solution according to the principle of conjugate effort and flow variables, see [Cellier, 1991], for heat conduction in solids is to use temperature and entropy flow, but engineers are not familiar with entropy flow and it is not measurable.

For thermo-fluid systems the design of connectors is not as straightforward as is e. g., for electrical systems. One connector has to carry several flows and transmit several potentials. A further complication is that the true physical potentials that are the driving force for a phenomenon, e. g., the chemical potential, are often replaced by other, approximative quantities for practical reasons. It is often convenient to use extra variables in connectors to avoid redundant property calculations. A boundary layer for mass- or heat transfer can be implemented as a separate submodel and certain fluid properties are needed within the boundary layer submodel, but for both coding and computational efficiency it is a disadvantage to have two identical or overlapping property calculation functions in the boundary layer and the control volume representing the free stream.

In summary, for thermo-fluid models there are several possible definitions of connectors. The choice in ThermoFluid has proven practical and in case that more variables need to be communicated between submodels, this can be handled via definition equations in modifications, see Chapter 3.

The connector for quasi-steady state single- or multi component flows contains the variables:

```
connector BaseFlow "connector for quasi steady−state flow"
  parameter Integer                          nspecies(min=1);
  parameter String                           MediumType = "unspecified";
  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.Temperature                        T;
  SIunits.SpecificEntropy                    s;
  SIunits.RatioOfSpecificHeatCapacities      kappa;
end BaseFlow;
```

The quantities are number of components, a string for the fluid type and the further types are self-explaining and taken from the Modelica.SIunits library. The variable set in the `Static.BaseFlow` connector is not minimal but it has been chosen in the above way for efficiency reasons. Fluid property calculations are expensive and via including density, entropy and ratio of specific heats in the connector, fluid property calculations can be omitted in simple flow models.

In the case of a dynamic momentum balance, the standard flow connector is extended with

```
connector BaseFlow "connector for dynamic flow"
  ... // up to here identical with the static connector
  flow SIunits.MomentumFlux  G_norm;
  SIunits.MomentumFlux       dG;
end BaseFlow;
```

The momentum flux G_{norm} is originally a three dimensional vector which has been reduced to a scalar due to the one dimensional character of the library. The variable dG is an approximation of the spatial derivative of the momentum flux which allows to connect submodels with dynamic momentum flow without loss of accuracy at the connectors.

The definition of the `HeatFlow` connector is

```
connector HeatFlow "discretized heat flow connector"
  parameter Integer          n;
  SIunits.Temperature[n]     T;
  flow SIunits.Power[n]      q;
end HeatFlow;
```

where n stands for the spatial discretization, $T[n]$ for the temperature and $q[n]$ for the heat flow.

In the ThermoFluid library it is possible to model chemical reactions in a reaction subcomponent. The reaction subcomponent uses Modelica flow semantics for the molar net flows of reactants and products. The reaction components use the following connector definition:

```
connector ChemFlow "connector for reaction submodels"
  parameter Integer                          n, nspecies;
  parameter String                           MediumType="unspecified";
  SIunits.Temperature[n]                     T "temperature";
  SIunits.Pressure[n]                        p "pressure";
  SIunits.Concentration[n,nspecies]          conc "concentration";
  flow SIunits.Power[n]                      q "heat of reaction";
  flow SIunits.MolarFlowRate[n,nspecies] rZ "product + reactant flows";
end ChemFlow
```

The variables in the ChemFlow connector are discretization number n, number of components *nspecies*, the fluid type, temperature, pressure, concentrations of the components, component specific enthalpies, molar flow rates and heat generation from the heat of reaction. In the standard ThermoFluid implementation, the heat of formation of the reaction is taken care of by including it in the specific enthalpy so that the heat flow $q = 0$. Under some circumstances, e. g., when reactions take place in a porous catalytic layer, it may be better to treat the heat of reaction separately. For those cases, q is included in the ChemFlow connector.

A connector for mass diffusion for semi-permeable membranes is defined similarly, here the partial pressures of the components are the potential variables causing the diffusion mass flow.

```
connector DiffusionFlow "connector for membrane diffusion"
  parameter Integer                          n, nspecies;
  parameter String                           MediumType="unspecified";
  SIunits.Temperature[n]                     T "temperature";
  SIunits.SpecificEnthalpy[n]                dHMx "change of enthalpy";
  SIunits.Pressure[n,nspecies]               pp "partial pressures";
  flow SIunits.MassFlowRate[n,nspecies] rM "diffusion mass flow";
  flow SIunits.Power[n]                      q "heat flow through membrane";
end DiffusionFlow
```

Apart from the basic connectors it is useful to provide shell models for typical configurations of connectors, e. g., two BaseFlow connectors and a HeatFlow connector for a control volume with heat transfer. All standard connectors configurations composed from the basic connectors above for control volumes, pipes, junctions, heat exchangers and reservoirs are predefined in the Interfaces package.

There is a particular shortcoming in the Dymola user interface that influences the class structure of ThermoFluid. When multiple inheritance

is used, only the graphics contained in the first inherited class are used in the user interface. This means that care has to be taken to make sure that all connectors and the model icon are defined in the first class in declaration order.

5.6 Base Models

The sub-library BaseModels is by far the largest and most important part of the ThermoFluid library. Base models are models of physical phenomena which form a clear conceptual unit but can not be simulated by themselves. A base model is the smallest unit of reuse in the ThermoFluid library. The implementation parts of different base models have no overlap, but they often use identical or largely overlapping sets of variable declarations. This is a technique to apply multiple inheritance in the composition of models in a straightforward way.

Model developers that want to build new application libraries on top of ThermoFluid have to understand the content and structure of the base models. This section together with the description of how to assemble control volume models in Section 5.7 should give a model developer all information required to build special purpose models while reusing existing code.

The simpler parts of BaseModels are:

- CommonRecords collects sets of variables which are used by more than one base class. Base classes with wisely selected sets of common variables makes it easy to use *type compatibility* for replaceable classes,
- The package CommonFunctions provides utility functions for thermodynamic models.
- Package InitialConditions defines standard cases for the specification of initial conditions.
- ThermoFluidInitLimits sets reasonable defaults of nominal, minimum and maximum values for all variable types used in the ThermoFluid library. These may have to be overridden in derived classes.

These models are largely self-explanatory and don't need to be described in detail.

Balance Equations

The submodels for the balance equations take all contributions to the mass- energy and momentum balances for a control volume into account.

This is a straightforward exercise, but the design has to be done carefully to be flexible for later additions to the model. The basic responsibility of the balance submodel is to add the contributions from the BaseFlow connectors to the balance equations. It would be possible to vectorize all Flow and HeatFlow connectors and make one balance submodel for almost all connector configurations, but this does not work well for graphical component usage. Each connector which is used for graphical model composition in a user interface needs a distinct position in the component layout. For the case of the BaseFlow connectors, all common configurations are therefore provided in the Balance sub-library. The important cases for graphical layout are TwoPort and ThreePort. To use the same solution for all possible combinations of heat transfer, reactions and membrane diffusion connectors would lead to a huge number of classes that could never cover all cases. The basic flow classes contain instead a component which allows addition of transfer phenomena and interfaces later, as an add-on.

The basic balance equation of a control volume with two connectors is implemented as

```
dM_x = a.mdot_x + b.mdot_x + rM;
dU   = a.q_conv + b.q_conv + Q_s;
```

This means that rM and Q_s, corresponding to the source terms in (4.5) and (4.20) should be unspecified in the general base class, and specified at a later stage when the balance class is reused in the model of a specific component. When there are no reactions or heat interaction with the volume there is no need for any source terms. In this case the model should provide a default value of zero production.

In Modelica it is not possible to change an equation after it has been introduced into the model or "overwrite" it as it is e. g., in the SMILE language. One way to prepare models for adding user defined heat- or mass transfer phenomena later is to use the Modelica connect semantics to add contributions from heat transfer, reaction or diffusion processes that are added to the default behavior of the control volume. Equations can not be changed or expanded after they are defined. A special construction is required to overcome a seemingly contradictory situation: a production term should be zero by default, but without writing this explicitly into the model. This contradiction can be solved with unconnected flow connectors: a **flow** variable has a zero default value in an unconnected connector, see Chapter 3. Therefore it is possible to use an object as a gateway to the balance equations, the HeatAndMass -object. Heat and mass transfer laws can be added as components to derived classes or the connectors in the HeatAndMass -object can be connected to additional outside connectors

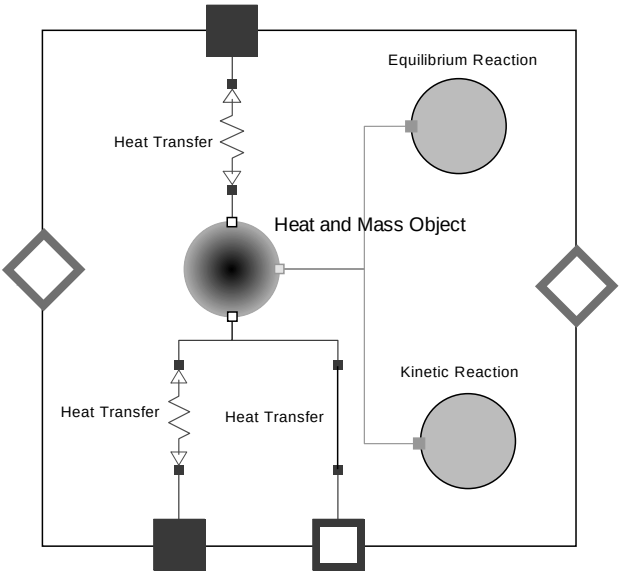


Figure 5.6 Schematic of the HeatAndMassObject with heat interaction and reaction objects (no diffusion and work connectors present).

directly.

In the reaction objects, the mass flows have to be calculated with the signs such that *reactants* flow into the reaction object and *products* flow back into the control volume. Because mass flows are flow-type variables, the sum of these flows at the connector is 0. Therefore, products are computed with a negative sign in the reaction object, resulting in a positive flow into the control volume.

Medium Property Routines

For simulation of thermo-hydraulic systems, it is necessary to have accurate models for the thermodynamic properties of the fluid that is flowing in the system. For the purpose of dynamic system simulation, the following criteria have to be met:

- Accuracy
- Speed
- Robustness

In some areas there exist recommended formulations (IAPWS/IF97 for water, [Wagner and Kruse, 1998]) or de facto standards (NIST-REFPROP routines for refrigerants, [McLinden *et al.*, 1998]) that define the state of

the art and are expected by users. For process engineering, interfaces to property database systems are indispensable for getting access to properties of commonly used substances. External function call interfaces in Modelica make it possible to use external property databases via simple wrapper functions. Available routines and most medium property models in the literature (see, e.g., [Reid *et al.*, 1987]) are designed with stationary calculations in mind, therefore they have to be extended to include some needed extra derivatives for dynamic calculations.

Proper adaptation of property calculations to dynamic simulation can help to speed up simulations a lot. Whenever possible the medium properties should be non-iterative, which is the case when they are explicit functions of the dynamic states. This is easy to achieve for the steam tables, where the industrial standard formulation, IAPWS-IF97, has explicit routines for a variety of input variables (pressure and temperature, enthalpy or entropy). The complete industrial steam tables with many special adaptations for fast dynamic simulation are implemented in the ThermoFluid library. Inverse functions and high accuracy approximations to the phase boundary are exclusively available for water and steam properties.

Two Phase Properties Huge amounts of computation time can be saved by pre-computing the phase boundaries off-line and use an auxiliary equation for it. Vapor-liquid equilibrium calculations (VLE) for cubic and other medium models have to be performed iteratively and numerically, either by using Maxwell's criterium or by calculating the condition that Gibbs' free energy is equal for both phases. Performing such calculation at each step of the simulation leads to very large simulation time. In order to calculate medium properties inside the two-phase region, it is for non-transient states 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 media listed above, high accuracy approximations are either available in the standard formulation (e.g., partially for water and CO_2) or provided in the base library.

The phase boundaries require special attention: the derivatives of most properties are discontinuous across the phase transition, as illustrated in Figure 5.7. A change of the phase of fluids in a control volume has therefore to be implemented as a discrete variable which restarts the integration routine. This is a requirement for robustness and efficiency in most normal cases, but it can lead to unexpected "sliding mode" behavior. Continuously differentiable trajectories of the enthalpies in a control volume

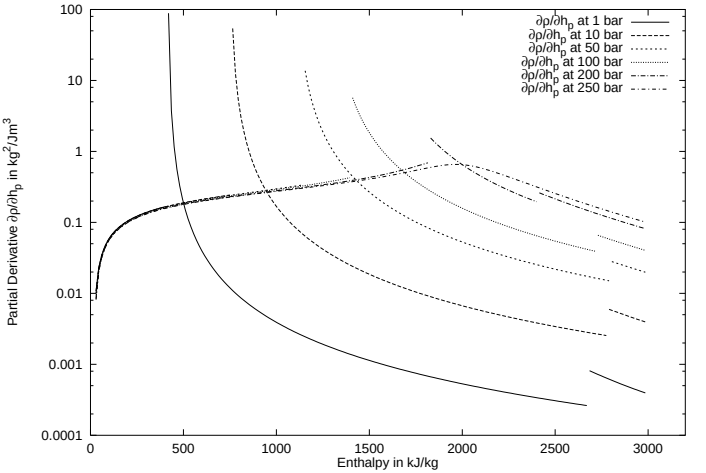


Figure 5.7 Partial derivative $\left.\frac{\partial \rho}{\partial h}\right|_p$ of water.

may lead to density changes with a discontinuous derivative. The density influences several terms in the momentum balance such that the mass flow changes with a phase change. The result is that phase changes may lead to *chattering* of the phase, as discussed in Section 2.1. The chattering is an artefact from the spatial discretization of the momentum balance. There are two ways to avoid this problem:

- Modify the momentum balance so that no spatial derivatives are taken over the phase boundary. This requires that the phase boundary is identified and that the grid is adjusted accordingly. This works well as has been demonstrated in [Heusser, 1996], but the model becomes quite complex.
- Modify the property calculations to be continuously differentiable across the phase boundary. This can be used for all fluid types, but results in non-linearities with very steep gradients.

Either of these solutions would be valuable additions to improve the robustness of ThermoFluid for two phase flows.

Ideal Gases The data and basic formulas for the implementation of ideal gas properties have been taken from [Gordon and McBride, 1994]. The properties are specifically designed for high temperature applications like combustion. All gas properties can be computed from polynomial-like

functions with some rational and logarithmic terms for the specific enthalpy h, specific entropy s and specific heat capacity c_p. The temperature range of the polynomials is split into two intervals, one for the temperature range 200 K – 1000 K and the next from 1000 K to 6000 K. The functions have the following form:

$$h(T) = RT\left(-\frac{a_1}{T^2} + a_2\frac{\log(T)}{T} + \sum_{i=3}^{7} a_i\frac{T^{i-3}}{i-2} + \frac{b_1}{T}\right)$$

$$s(T) = R\left(-\frac{a_1}{2T^2} - \frac{a_2}{T} + a_3\log(T) + \sum_{i=4}^{7} a_i\frac{T^{i-3}}{i-3} + b_2\right)$$

$$c_p(T) = R\left(\sum_{i=1}^{7} a_i T^{i-3}\right)$$

The coefficients a_i and b_i are given in [Gordon and McBride, 1994] for each temperature interval. The functions are continously differentiable at the matching point of 1000 K[2]. Reference enthalpy at ISO standard conditions of 25 °C and enthalpy of formation are also included in the data. The specific enthalpy and entropy are obtained by integrating the specific heat capacity, using the formulas given in (4.29). The enthalpy of formation is included with the latent specific enthalpy because this simplifies modeling of equilibrium reactions considerably, see [Pantelides, 2000]. The ideal gas properties are much simpler than the multi-parameter equations of state. Therefore they are implemented as equations, not as functions. This has considerable advantages if symbolic manipulation of the formulas has to be applied.

Implementation Notes Almost all high accuracy pure fluid property calculations are based on dimensionless forms of the fundamental Helmholtz or Gibbs energy. The reason for this is that the primary variables of these formulations are most easily accessible through measurements, see details in [Span, 2000]. The necessary computational steps for transforming the fundamental equation into the needed properties follow a fixed scheme. The property calculations in ThermoFluid follow this scheme which makes it straightforward to add new properties with little effort. The steps are:

- Compute the fundamental equation[3] (FEQ) together with its first

[2]The original report contains data for many more substances which have not been implemented. The parameterization for some fluids in the data is different from the format presented here.

[3]The functional forms of the fundamental equations are rather complicated. They are not repeated here, but references to their original publication are given below.

two derivatives into both directions, including the mixed derivative, and return all values in a record.

- From the fundamental equation and the normalization constants, all needed properties can be calculated using transformation rules using functional determinants, as described in Appendix B.

Both steps are implemented as functions for all properties needed in ThermoFluid models. When the fundamental equations have the same parameter structure, as is the case for many refrigerants, the same FEQ serves for many fluids.

From a user point of view it would be nice to hide implementation details like e. g., the type of the FEQ with the help of another abstraction layer. For water and steam a high level interface is implemented which hides the messy details of the IAPWS/IF97 and simply provides properties for standard modeling cases.

Currently high-accuracy medium models are implemented for the whole fluid region for water [Wagner and Kruse, 1998], carbon dioxide [Span and Wagner, 1996] and R134a [Tillner-Roth and Baehr, 1994]. More refrigerant properties will soon be available. Modelica language support for computations using structured data and nested function calls passing large amounts of data has improved since the first implementation of the medium properties. In Modelica 2.0, some parts of the property calculations could be done more elegantly and more generally.

To summarize, the medium properties that are provided with this library:

- are adapted for use with dynamic simulations.
- use non-iterative, auxiliary equations for the calculation of VLE.
- are highly accurate for water, CO_2 and $R134a$.
- include ideal gas properties for a wide variety of gases.

Flow Models

Flow models are base classes for all equipment that is modeled with negligible hold-up of mass. Typical examples are valves, orifices, compressors, turbine stages and pumps. Static and dynamic versions of the momentum balance for distributed models described in Section 4.3 are also implemented in the FlowModel sub-library.

The FlowModel base classes contain the general parts for these models, but not those equations that are the central characteristic of a flow component. One of the fundamental features of the ThermoFluid library is that it allows bi-directional flow. This may seem odd for components like pumps and turbines, but a pump may be switched off and have a reverse pressure gradient at the start-up of a flow network. In that case the

pump acts like an orifice with a very high pressure resistance. Reverse mass flow in pumps and compressors during surge operation is important in the analysis of their dynamic behavior. A common base-class can be used for all of the above components that checks the flow direction and makes sure that the upstream properties are used. The modeler that uses the base classes does not have to ensure that mass- and energy balances in the neighboring control volumes are fulfilled. This is achieved by applying two rules:

- The *upstream* value of any property transported with the the flow e. g., density, is used inside the component.
- The energy flow is calculated from the upstream specific enthalpy and the change of specific enthalpy in the component.

A practical problem related to bi-directional flow is the question whether or not a flow reversal has to be handled as a discrete event. This depends on model parameters and the speed of transients and can therefore not be fixed once and for all in the library models. Fast gradients in certain flow networks work better with events because a flow reversal in one network branch can cause sharp gradients or even discontinuous derivatives in mass- and energy balances of the adjacent control volumes. The possibility of discontinuous derivatives depends on the type of the spatial discretization scheme used. Currently, upwind- and centered finite differences are implemented for discretized models. Slow flow reversals in discretized models are smooth in the derivatives, so that in practice simulation is faster when events are omitted. In the limiting case of the PDE, the derivatives are smooth.

Under certain circumstances in network branches with zero flow, events can lead to chattering, as illustrated in Section 2.1. The library offers both options via a Boolean variable `generateEventForReversal` which is set to **false** by default.

```
partial model FlowModel
  ... // declarations omitted
equation
  if generateEventForReversal then
    dir = if mdot > 0 then 1 else -1; // event is caused
  else
    dir = noEvent(if mdot > 0 then 1 else -1); // no event occurs
  end if;
  T = noEvent(if dir > 0 then a.T else b.T); // no further events
  d = noEvent(if dir > 0 then a.d else b.d); // generated here
  ... // more equations;
end FlowModel;
```

Listing 5.1 Flow Model Code.

For simple models of turbines and pumps there are also flow models which allow only one flow direction. They use an **assert** statement to make sure that derived models do not calculate a negative mass flow which violates an assumption of the base class. If a pump component based on such a base class is switched off during simulation time, the modeler has to make sure that the variable mdot is positive at all times.

State Variable Transformations

The balance equations collect changes in mass and energy through convection or other phenomena in the algebraic variables dM (change of mass) and dU (change of inner energy), but they do not introduce the state equations which are used by the integrator. The choice of state variables is a performance question, as discussed in detail in Section 4.6 and is coupled to the availability of medium property routines. The ThermoFluid library offers a selection of models with different dynamic states which should provide a good choice for most practical applications. The derivations of the different forms of dynamic state variables from the canonical form of the mass- and energy balance is presented in section 4.5. The interdependence of the property calculations and the dynamic states is reflected in the inheritance structure connecting the StateTransformations and MediumModels sub-libraries: one class from each package and for each group of state variables inherits from the same base class in the package CommonRecords. The pairs of classes deal with the same variables, but the calculation is split into two submodels, see Figure 5.8. The StateTransformations.Model_xy submodel computes the states x and y and the MediumModels.Fluid.Model_xy model computes all other needed fluid properties. Here, x and y are placeholders for possible state variables like pressure p and temperature T.

It should be pointed out that the coupling of the dynamic states and the property routines as presented above is not necessary, but it results in models which are computationally efficient and easy to initialize.

Chemical Reactions

Chemical reactions are implemented as submodels which are connected to the HeatAndMass-object. This design makes it possible to treat complex reactions modularly and to combine several reactions in a control volume graphically. It is also possible to add more complex or less important reactions later in the modeling process. The reaction object and interfaces are designed to handle both kinetic and equilibrium reactions even though equilibrium reactions have not yet been included in the library.

The empirical three-parameter Arrhenius equation (5.1) is the most popular form of an empirical rate equation. Parameters for this mechanism are tabulated for many reactions in standard chemical engineering

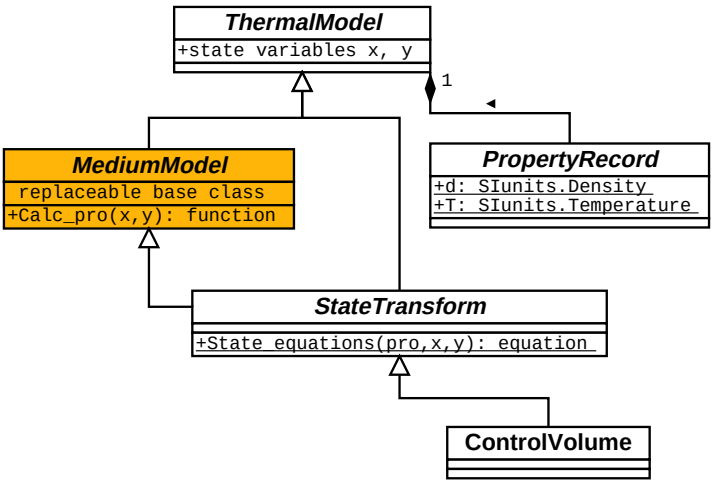


Figure 5.8 Inheritance structure of the Medium and StateTransform models.

handbooks.

$$k(T) = AT^b e^{-E_A/R_u T} \tag{5.1}$$

The parameters A, b and E_A are vectorized with the number of reactions nr as lengths. Note in the code in Listing 29 that the BaseObject does not make any assumption about the reaction rate reacRate, only the general relation between reaction rate, stoichiometry and overall component generation rate is included. The special case of the Arrhenius reaction is implemented in a derived class.

```
package Reactions
import Modelica.SIunits;
partial model BaseObject "base class for all dynamic reactions"
  parameter Integer                          n, nc, nr;
  SIunits.MolarReactionRate[n,nr]            reacRate;
  parameter SIunits.SpecificEnthalpy[nc]     compHf;
  parameter StoichiometricNumber[nr,nc]      stoich=zeros(nr,nc);
equation
  for i in 1:n loop                  // per component reaction rates
    r.rZ[i ,:] = transpose(stoich)*reacRate[i ,:]; // rZ_i = sum_{j=1}^{nr} v_ij r_j
  end for;
end BaseObject;
partial model Basic "Simple Arrhenius reaction"
  extends BaseObject;
  parameter CommonRecords.Rate[nr]           A0;
  parameter Real[nr]                         b;
```

```
  parameter SIunits.MolarInternalEnergy[nr] Ea;
  Concentration[n,nc]                       conc;
equation
  for i in 1:n loop
    for j in 1:nr loop
      rateK[i,j] = A0[j]*r.T[i]^b[j]*exp(-Ea[j]/R/r.T[i]); // A_{0,j} T_i^b e^{-Ea_j/(R T_i)}
      concC[i,j] = product(conc[i,rIndex[j,:]]);
      reacRate[i,:] = rateK[i,j]*concC[i,j];
    end for;
  end for;
end Basic;
end Reactions;
```

Listing 5.2 The Reactions Package.

The example of a hydrogen–oxygen combustion in Section 5.9 demonstrates how the base class is used to model a specific reaction mechanism.

Constitutive Equations

The constitutive equations are empirical relations like heat flow and pressure drop correlations and characteristics of machinery. They are typically formulated as characteristic equations for individual components, often algebraic equations but they can also be differential equations. These constitutive equations should be **replaceable** or left unspecified in a base class, in order to have a general model of a component that can be used in different situations. Exchanging the model for the characteristics or adding the missing equation in a derived model completes the model. From the point of view of high reusability of library components, constitutive equations are difficult to organize for several reasons:

- There is a large number of empirical equations and often two equations for the same phenomenon are based on different sets of measured input variables. Implementation of the practical part of the published equations is a huge effort.
- Constitutive equations are rather small units for reusable objects. Heat transfer equations e. g., compute the Nusselt number Nu from geometric and thermodynamic variables. In Modelica, **replaceable function** is a construct that is well suited for small, reusable units of behavior, but functions have disadvantages as discussed in Chapter 3. A **replaceable** component can be used instead. Both solutions require more implementation overhead than seems appropriate at first sight.
- Some equations are published as graphical maps or data tables for interpolation. A flexible solution has to be open for completely different implementations for one equation type.

It is clear that a model library can not contain all desirable equations. Not many equations are actually implemented in ThermoFluid, but the most important pressure drop equations are available. Pressure drop and heat transfer equations are implemented as **replaceable** components. For them it would be an enormous improvement in the ease-of-use if Modelica tools would implement the **choices** annotation, described in Chapter 3. The code for pressure drops and heat transfer laws for a pipe model from the ThermoFluid-library is illustrated in the following listing.

```
model PipeDS "Distributed base pipe model"
  extends ControlVolumes.Volume2PortDS_ph(
    ... // some modifications omitted
    redeclare model PressureLoss = PressureDrop.PressureLossD
                    (dp0=char.dp0, mdot0=char.mdot0));
  replaceable TransferLaws.Basic HeatRes
    (n=n,Aheatloss=geo.Aheat) extends TransferLaws.Ideal
   ... // further code omitted
equation
    connect(q,heat.q);
    connect(q,HeatRes.qb);
    connect(qa,HeatRes.qa);
end PipeDS;
```

Listing 5.3 Pressure Loss and Heat Transfer Models in a Pipe.

The pipe model inherits from a general distributed control volume. The PressureLoss model that has been declared in the base class is exchanged with a more suitable one via the **redeclare** command. A **replaceable** heat transfer law is added to the model. The replaceable heat transfer law is declared and connected to HeatAndMass - object via a heat flow connector q. The code shows also how to propagate parameters into subcomponents.

5.7 Partial Components

Partial components are models which are almost complete for use in simulation, but some details of the behavior need to be specified by the user in order to make the model complete and well-defined. Partially defined models are a logical consequence of using inheritance and collecting common behavioral parts of a group of similar models in a superclass. The Modelica language and object oriented design supply a rich repertoire of possibilities to define an almost complete model. Common means to leave parts of the model unspecified are:

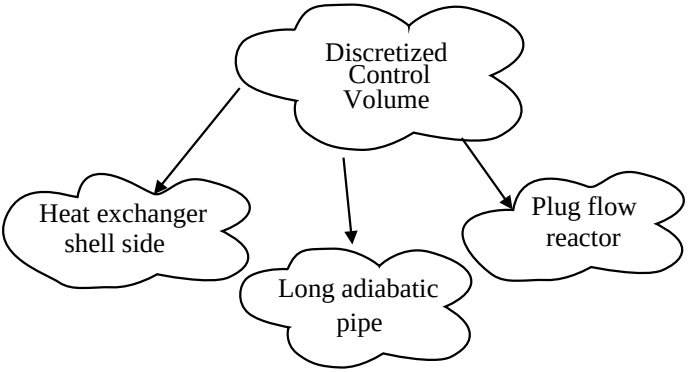


Figure 5.9 Selection of partial models as base classes.

- A model without default values for structural parameters. *Structural* parameters have an influence on the number of equations or the structure of the equation system, they have to be assigned a value before the model is compiled. The assignment of values to structural parameters is a requirement if the equation system is manipulated symbolically before compilation.
- Incomplete models simply do not implement all equations that are needed for a complete model. This is the most common way of creating partial models. For the convenience of the model developers that have to fill in the gaps, the model should contain as much of the final model as possible.
- A model with replaceable components or class parameters where the *default class* of the component or class parameter is a partial model. Before the class can be used, the default has to be redeclared to a model which is complete.
- Combinations of the above are also partial models.

Mathematical modeling of systems offers an incredibly broad spectrum of possibilities for modeling the same system in different ways. Nonetheless, there are some basic properties of the system that all dynamic models should include. Using a control volume as an example it is obvious that most models of technical equipment that contain a fluid have to fulfill a mass and an energy balance. On the other hand, there is an rich diversity of physical properties and heat transfer equations that can be used together with that model. Consequently, a partial control volume that does

either not define the variable parts or declares replaceable placeholders, is a model that has a high likelihood of being reused. The main idea is thus to define partial models at branching points where as much common behavior as possible is included in the model, but parts which would exclude a group of other models to use the same code should not be included see Figure 5.9. There are many ways to define such classes, but with modeling experience in an application domain it is not difficult to select common features for partial models.

The packages for partial models in `PartialComponents` are:

- Package **ControlVolumes** includes lumped and distributed control volume models. Missing or replaceable parts are a geometry parameterization, a physical property model, pressure loss and heat transfer models.
- **ThreePorts** are lumped control volumes with three `BaseFlow` connectors which are mainly used for flow junctions or splitters. The unspecified parts are the same as for control volumes.
- Package **Reservoirs** implements thermodynamic reservoirs. They do not include property calculations. Including a property model in reservoirs is strictly speaking not necessary, but it simplifies the parameterization for model users.
- Package **Turbines** provides turbine stage models that usually need further assembly, see Section 5.8, and do not implement equations for the efficiency.
- Package **Pumps** includes base classes for simple pumps and interfaces for controlling them. Missing details are pump characteristics and fluid property calculations.
- **Compressors** have some base classes for compressors. The compressor package needs more work for a practical implementation of base classes for compressors.
- Package **Valves** contains base classes for isentropic and isenthalpic throttles. Medium specific isentropic enthalpies are unspecified.
- Package **Sensors** implements common code for sensor models, excluding the characteristic behavior of specific sensor types e. g., the typical first order dynamics of thermocouples.

The emphasis of the ThermoFluid library has been on the dynamic behavior of fluids. Most of the work has been spent on all types of control volumes, while the models for equipment have been kept simple. The partial components for flow equipment, e. g., pumps, compressors and turbines,

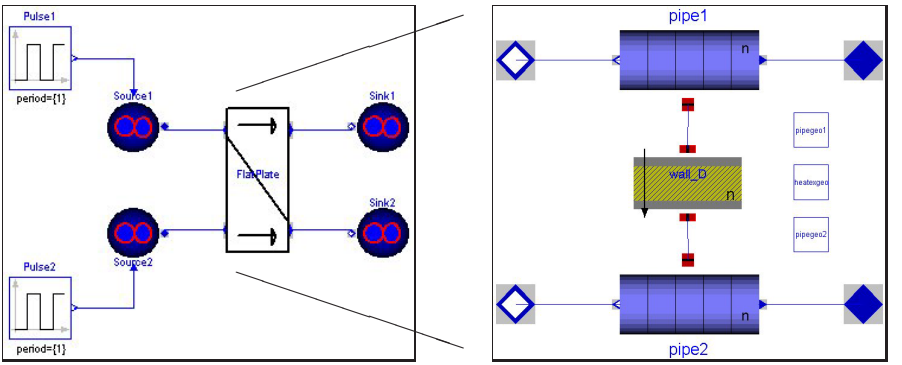


Figure 5.10 Example system using a heat exchanger consisting of two pipes, a wall and 4 connectors. In order to simulate it, all flow connectors are connected to reservoirs as boundary conditions.

cover only a fraction of the models that would be useful for a comprehensive process engineering or power plant modeling library. ThermoFluid contains standard cases for simple models which cover the basic needs for system models.

5.8 Component Models

The aim of the development of the ThermoFluid-library was not a polished component library for a small application field, but to create a basic reusable structure for a broad range of applications. The existing libraries in Components have been implemented for demonstration purposes and to demonstrate how code reuse based on partial models can be used in practice. This section presents some of these components and modeling guidelines for their assembly.

Heat exchangers

Heat exchangers are modeled using base components from the library. Real heat exchangers can have complex geometries and flow patterns, but for system simulation they can be approximated well with one pipe each for the cold and hot fluid and a wall in-between. All of these parts can be modeled with a variety of details, but the overall structure is always similar to Figure 5.10.

Heat exchangers can either be lumped or distributed. In the distributed case the heat transfer model uses a simple temperature difference be-

tween the individual elements of the distributed pipes. The lumped case is either based on this simple model or uses the logarithmic mean temperature.

A different approach to heat exchanger modeling that elegantly and efficiently uses advanced object-oriented modeling constructs has been presented in [Mattsson, 1997]. The heat exchanger is cut into discrete slices perpendicular to the axis of both flow channels (which are assumed to be in parallel). An array of such slices is modeled as an array of components in Modelica. An instance of the heat exchanger can be initialized with the number of slices as a parameter which gives the discretization of the heat exchanger. The authors of ThermoFluid feel however that the approach of building up a heat exchanger from the physical components hot side, cold side and wall gives more flexibility and better reuse in a library which uses these components also in other models.

Reservoirs

Reservoirs are used to add boundary conditions to other components or systems of components. Two types of boundary conditions have to be specified: either the *flows* of mass, energy and momentum or the *potentials* given by the thermodynamic state. The reservoirs provide boundary conditions in accordance with these two types:

- either the thermodynamic state, using any of the combinations of variables listed in Section 4.6. These models are called *reservoirs*.

- A model computing the flows of mass, energy and momentum at the boundary. These models are called *sources*.

The latter boundary condition can be realized using a reservoir defining the thermodynamic state connected to a simple orifice model. The advantage of that way of specifying flow boundary conditions is that the steady state pressure has to be in the interval defined by the boundary reservoirs. The second type of reservoirs connects to the HeatFlow -connectors and again, either the potential or the flow can be specified. In order to give a uniform interface to boundary conditions, all variables defining the boundary conditions can be set via signals, too. For variables like pressure where discontinuous boundary conditions are non-physical and lead to numerical problems, the input signal is interpreted as the time derivative of that condition. The sample system in Figure 5.10 contains 2 controlled sources and 2 sinks. Thermodynamic reservoirs are also called "infinite reservoirs", because the thermodynamic conditions do not vary over time as mass and energy leave or enter the reservoir.

Flow Splitters- and Junctions

Correct modeling of flow splitters and junctions (from now on referred to as T-nodes) for reversible flows in all connecting branches is not difficult conceptually, but there are three different choices of doing it with models from the ThermoFluid library which have implications with respect to numerical robustness and performance. The cases of quasi steady state and dynamic momentum balance differ significantly because a correct modeling of the momentum balance in T-nodes requires a two-dimensional geometry. For quasi steady state flow, the three options are:

A Do not use a dedicated model at all. Instead, simply connect two flow models to the inflow of a volume in the case of a junction and two to the outflow in the case of a splitter. This is the preferred procedure for most cases in system modeling. It works for splitting or joining an arbitrary number of streams, not only two.

B Use a normal control volume with a small, compressible volume and three flow connectors, called `Volume3Port`.

C Use an idealized control volume with 0 volume and without dynamic states, called `Ideal3Port`.

Case A is preferable in most cases due to its simplicity. It has only one slight disadvantage which is the consequence of the ideal mixing assumptions in control volumes and only gives noticeable errors for short periods after flow reversals. For example if a hot liquid is mixed with a cold liquid and the mixture is discharged into a much colder, perfectly mixed control volume, then the temperature at the mixing point after a short flow reversal is the temperature of the cold control volume. This temperature can be quite different from the mixing temperature. The same potential problem occurs in case C.

Case B is the physically most detailed model and is well suited for mixing of flows. It has one numerical disadvantage: because all other volumes in a system are usually much larger, the system gets very stiff. For modeling for control it has the additional disadvantage that it adds dynamic states to the overall system. Case B should therefore only be used for detailed modeling of mixing processes.

Case III is the physically most reasonable model when the difference between in the volume in the T-node and other volumes is large. Numerically it may be more difficult to solve in some cases because the pressure in the T-node ends up in a non-linear equation system. The mixing enthalpy is calculated as:

$$h_{mix} = \forall\,(\dot{m}_i > 0)\,\frac{\sum_i h_i \dot{m}_i}{\sum_i \dot{m}_i} \tag{5.2}$$

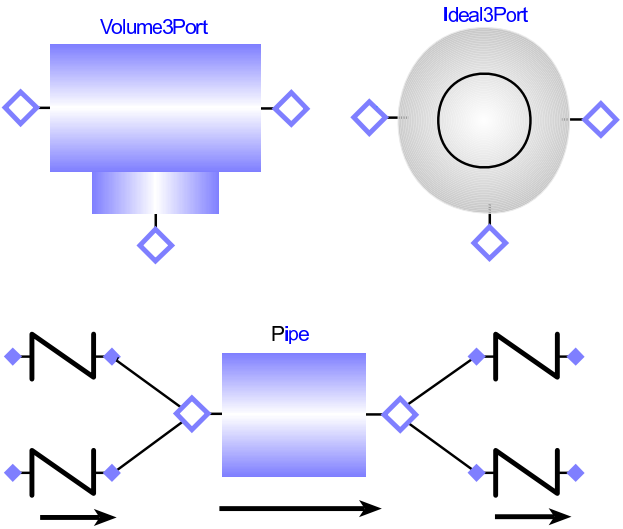


Figure 5.11 Modeling options for T-nodes. Taking advantage of the flow semantics and omitting explicit models is often the best solution. All three cases handle bi-directional flows.

This expression could easily be written directly in Modelica, but in the ThermoFluid library it is assembled symbolically from the equations for reversible flows in the adjacent flow models. It works for all configurations of inflow and outflow in any of the connected branches, but it fails numerically if all flows are zero. This can be handled by a simulation tool in a heuristic manner because the denominator goes to zero at the same time by checking for this condition and keeping the value from the previous time step.

T-nodes for dynamic flow are more difficult to model correctly. The problem is to have data for friction losses for all possible flow configurations. In the ThermoFluid library there are simple models which neglect the losses in the T-node and add or split the momentum in the same ratio as the mass flows. This is an adequate model for smoothly shaped T-nodes near their design flow conditions.

Turbines and Compressors

Multi-stage turbines and compressors are built up from turbine and compressor stage models, as described in Section 4.7 and Section 4.8, and control volumes between them. As an example, consider the schematic in Figure 5.12 showing the high pressure stage and first intermediate pressure stage of a typical steam turbine. There are bleed mass flows to

the pre-heaters between the stages. These flow splits are one reason that makes it numerically advantageous to model the rather small volumes between the stages as real control volumes. The method of modeling turbines as a series of stage models and control volumes has a long tradition, see [Traupel, 1977] and [Gašparović and Stapersma, 1973]. It fits naturally into the scheme of flow models and volumes used in ThermoFluid. There is a numerical difficulty associated with this scheme. Taking (4.6) for a constant volume, neglecting the enthalpy derivative and normalizing the variables with their nominal values we get

$$\tau_{TS}\frac{p^*}{dt} = \dot{m}^*_{in} - \dot{m}^*_{out}, \qquad \tau_{TS} = \left.\frac{\partial \rho}{\partial p}\right|_h V$$

Due to the very small volumes between turbine stages, the time constant τ_{TS} is much smaller than any other time constant in the system and thus renders the system very stiff. There are two possible remedies:

- set the volumes to zero which eliminates the states and introduces algebraic variables instead or
- make the volumes larger than they are in order to increase the time constants, but keep them small enough so that the mass storage effects in the turbine are still one or two orders of magnitude faster than the time constants of interest.

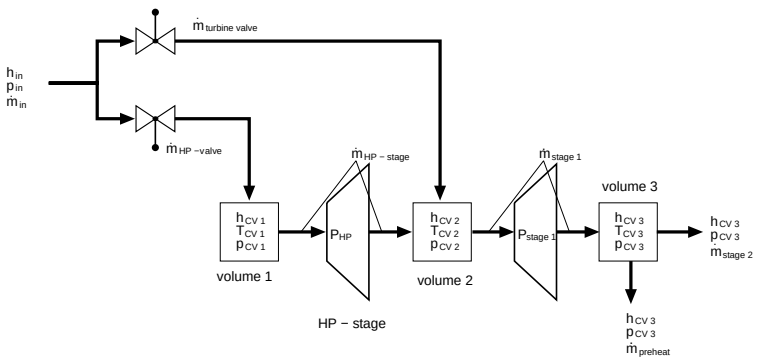


Figure 5.12 High- and medium pressure parts of a steam turbo group.

The first remedy might seem more logical at first sight because the volumes are a lot smaller than other volumes in the system, but due to the tap-off flows to the pre-heaters and the non-linear turbine mass flow equations a large and difficult to solve non-linear system of equations

appears. Large steam turbines may have up to 10 tap-off locations. If all of them are modeled as massless control volumes, all pressure levels and mass flows from the condenser via the pre-heaters to the high pressure end of the boiler are connected in one large non-linear equation system. Therefore, the method of enlarging the actual volumes is preferable and has been used successfully, see [Thumm, 1989]. Thumm investigated the control response to the fastest possible step changes of load demand for large steam power plants and even in that case the turbine time constants could be enlarged by up to two orders of magnitude without influencing the result notably.

The same paradigm of alternating stages and volumes can be used for large axial compressors. The problematic numerics are actually only associated to the pressures and mass flows. For low order models it is therefore possible to only model the pressure in the volumes and simplify the energy balance to be in steady-state, $h_{out} = h_{in}$.

5.9 Examples

This section shows some more complete examples of models that use ThermoFluid from the examples package. They are the result of small modeling projects were written to demonstrate and document the use of base classes. Not all examples are described in detail, but the complete model code and documentation is available for download at
http://www.control.lth.se/~hubertus/ThermoFluid.

Combustion of Hydrogen

The `HeatAndMass`-object as a means to include heat and mass transfer processes and reactions has been described in Section 5.6. An example of the use is included in the examples package of the ThermoFluid library. Using the model is straightforward, but to appreciate how it works, understanding of the Modelica **flow**-semantics and the workings of the reaction object presented in Section 5.6 is required.

The **partial model** `Basic`, see Section 5.6 implements the standard form of the Arrhenius reaction in a general way for a parameterized number of reactions. The reaction rates are calculated from the Arrhenius equation in (5.1) using the concentrations and rate parameters. To use the reaction component, like the kinetic reaction in Figure 5.6, the user only needs to specify the parameters. The stoichiometry matrix is constructed as shown in Table 5.2. The parameters are vectorized with the number of reactions `nr` as lengths. A reaction vessel is built by extending from a control volumes model which includes a replaceable reaction

object that defaults to the Arrhenius reaction. All parameters are passed in a modification. The matrix rIndex is of size $nr \times norder$, number of reactions times maximum order of any reaction in the model. Each row in rIndex stands for one reaction, the entries give the indices of the reactant components in the property array. This is clearly an example where the use of an enumeration data type would render the code more readable.

```
model ReactionVessel
  extends PartialComponents.ControlVolumes.Volume2PortRS_pTM(
    nspecies=7,
    geo(V=vol),
    redeclare model Medium = GasMix,
    reaction(n=1,
             nspecies=7,
             nr=8,
             A0={1.2e11,1.8e7,15,460,100,1.5e4,6.4e5,1e5},
             b={-1,0,2,1.6,1.6,1,-1,-1},
             Ea=1000*{69,0,32,78,14,72,0,0},
             rIndex={{1,5,8},{4,6,8},{2,4,8},{3,5,8},{2,6,8},
               {3,4,8},{7,5,5},{7,4,4}},
             stoich={{-1,0,0,1,-1,1,0},{1,0,0,-1,1,-1,0},
                     {0,-1,0,-1,1,1,0},
                     {0,1,-1,0,-1,1,0},{0,-1,1,0,1,-1,0},
                     {0,0,-1,-1,0,2,0},
                     {0,1,0,0,-2,0,0},{1,0,0,-2,0,0,0}}));
end ReactionVessel;
```

Listing 5.4 Specifying a reaction model via modifications to a base class

To construct models of other types of reactions the reaction BaseObject can be reused. The customized reaction model needs to add expressions for the reaction rates, either by adding equations or by calling a rate function. In this way packages of reactions can be built and reactions can graphically be added to standard reactor models.

We consider the combustion of hydrogen and oxygen into water. In a simple setting, see Figure 5.13, the system consists of a reservoir supplying the reactants, a reactor volume and a sink for the product flow. A heat source is added to provide the heat necessary to ignite the mixture.

The complete set of sub-reactions for this process involves a large number (> 40) of very fast reactions, see [Turns, 1993]. Here we only consider the 8 main reactions, involving the components { O_2, H_2, H_2O, O, H, OH, Ar }. Argon is included as an inert gas. The included reactions are listed in Table 5.2. The corresponding stoichiometry matrix and reaction rate parameters have been coded into a Basic reaction object inside the GasCV reaction vessel.

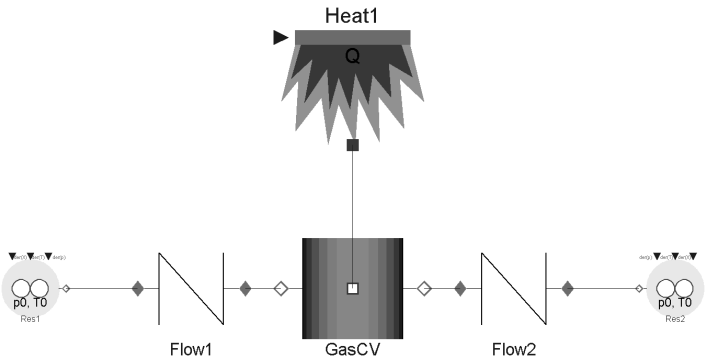


Figure 5.13 Schematic of example system with H_2–O_2 reaction.

The result plots show clearly that the reactions are extremely fast once they start. They saturate when all H_2 is burned and the flow through the volume reaches steady state. After the initial ignition, a steady inflow of premixed gases leads to a steady combustion with plenty of surplus oxygen. The speed of the reactions makes the system very stiff. The whole simulation shown in Figure 5.14 spans only a few milliseconds.

An Attemperator

The attemperator example demonstrates how to use the ThermoFluid components for flue gas and water and steam. Two heat exchangers are built up which are then combined with an injector and a controller to form the attemperator unit. The attemperator realizes the temperature con-

	$\{\ O_2$	H_2	H_2O	O	H	OH	Ar $\}$
$H + O_2 \rightarrow OH + O$	$\lceil -1$	0	0	1	-1	1	$0\rceil$
$OH + O \rightarrow H + O_2$	1	0	0	-1	1	-1	0
$O + H_2 \rightarrow OH + H$	0	-1	0	-1	1	1	0
$H_2O + H \rightarrow H_2 + OH$	0	1	-1	0	-1	1	0
$H_2 + OH \rightarrow H_2O + H$	0	-1	1	0	1	-1	0
$H_2O + O \rightarrow 2\ OH$	0	0	-1	-1	0	2	0
$2\ H + Ar \rightarrow H_2 + Ar$	0	1	0	0	-2	0	0
$2\ O + Ar \rightarrow O_2 + Ar$	$\lfloor 1$	0	0	-2	0	0	$0\rfloor$

Table 5.2 Reactions included in the H_2 O_2 reaction system and the corresponding stoichiometric matrix.

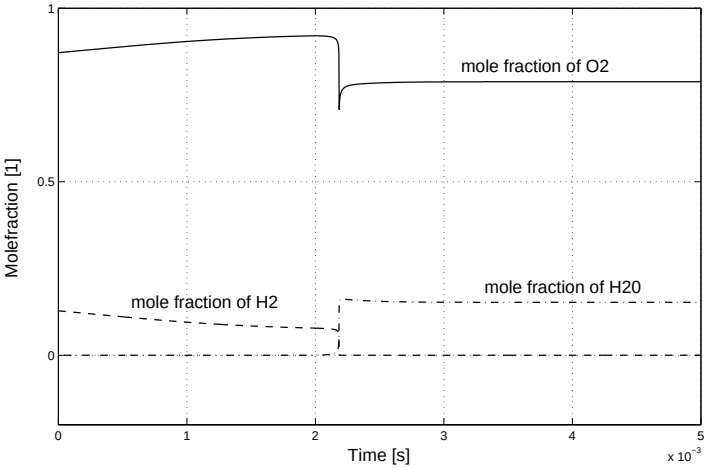


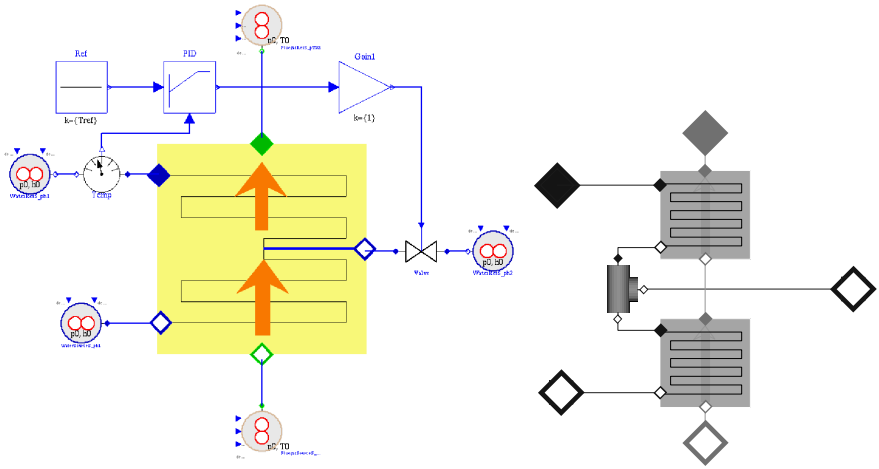
Figure 5.14 Molar fractions of the principal reactants and products

trol in steam power plants. It uses a ThreePort model for the mixing of subcooled water and superheated steam. It uses propagation of record parameter for propagating parameters from the top level into the components. This example uses only available components from the ThermoFluid library.

5.10 ThermoFluid Applications

The ThermoFluid library has reached a state that made it attractive to use in large scale modeling applications in the summer of 2000. Since then it has been available for download and to the knowledge of the authors has been used in the following modeling efforts:

- A steam distribution network in a paper plant, including several boilers and turbines [Lindstrand, 2002].
- Refrigeration plants, especially with CO_2 as the refrigerant [Pfafferot and Schmitz, 2002].
- Modeling of a steam boiler for dynamic online optimization of load changes [Franke, 2002].
- Fuel cell systems including the fuel processing steps using natural gas as a primary fuel.



(a) Top Level and Control

(b) Internal Structure

Figure 5.15 Schematic of a Steam Power Plant Attemperator with a PID-Controller.

- Combined heat and power micro turbine systems.
- Distribution of Cl_2 and H_2 in a large chemical plant.
- Pasteurization by steam injection in food processing plants.

In many cases the use of the library was exactly as intended: the base classes were used extensively to create a customized model library for a particular application. Some of the projects got along with the ready-to-use models in the library, other ones had to add a substantial modeling effort in order to achieve their goal, but all of them have in common that all dynamic parts of the model are built by inheriting from the library classes.

Most of the users of the ThermoFluid library were able to build application oriented libraries on top of ThermoFluid in spite of the initially scarce state of the documentation. The efforts built on top of ThermoFluid had very different goals and perspectives. They range from feasibility studies in the form of a masters thesis project to intensive industrial projects where several man-years of effort were spent on customized libraries.

Feedback from these projects has improved and is continuing to improve the usability of the library. A more widespread acceptance of object

oriented modeling using ThermoFluid from the current level would require three actions:

- Professional level documentation and training for new users.
- Commercial support and consulting services.
- Tools and specialized graphical user interfaces which are better accommodated to non-experts than the current Modelica tools.

Micro Turbine System

A micro turbine system for combined heat and electrical power generation is an industrial project that recently has been completed at the Swedish company Turbec AB. The combined heat and power system consists of the following main parts:

- Gas turbine engine and recuperator
- Electrical generator
- Electrical System
- Exhaust gas heat exchanger
- System for control and supervision

The basic compressor and turbine models were developed in a masters thesis project [GómezPérez, 2001]. These models were later refined and adapted to the turbine model used at Turbec AB, [Haugwitz, 2002], who also modeled the control system and used ThermoFluid to model all auxiliary system parts and heat exchangers. The exhaust gas heat exchanger is a gas-water counter-current flow type. It is assembled from library models for flue gas and water and did not need any user-written code.

One of the main uses of the model is to test control schemes. The model schematic in Figure 5.16 shows turbine, combustion chamber, recuperator, exhaust gas exchanger and turbine control system. As usual for turbines and compressors, a large part of the modeling effort has to be spent on the steady-state characteristics of the compressor and, to a smaller degree, the turbine.

In an islanding power configuration, where the micro turbine systems supplies a small electrical network in a stand-alone manner, the main goal of the control is to follow load changes as quickly as possible, without compromising the turbines lifetime by allowing turbine outlet temperatures to become too high. The electrical generator connected to the turbine generates high frequency three-phase AC. The AC is rectified and then converted to the standard electrical frequency of 50 Hz. This configuration makes it possible to use minor variations in the turbine angular

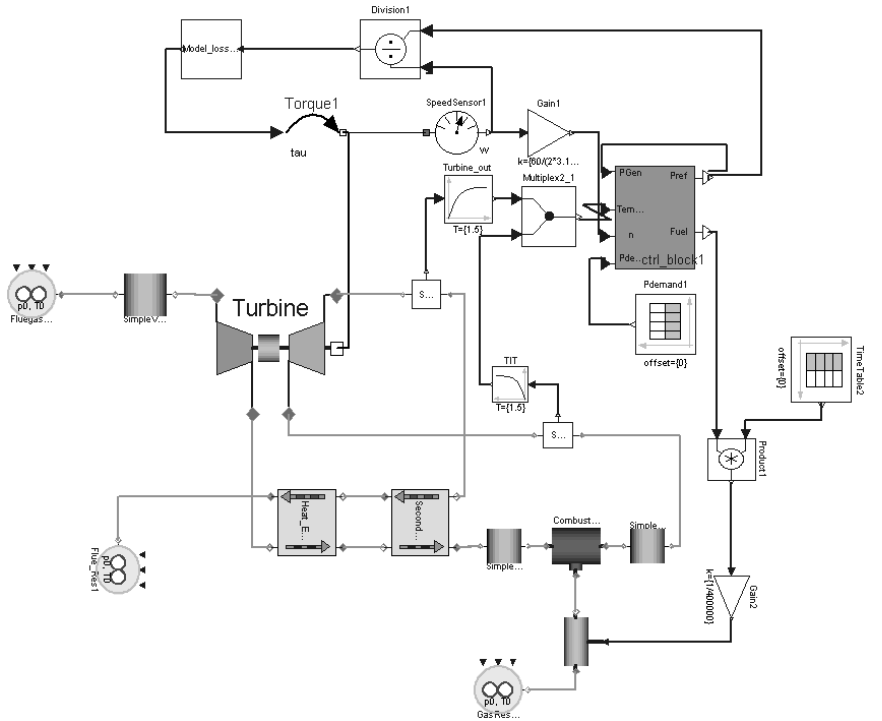


Figure 5.16 Schematic of a micro gas turbine system with recuperator and control system

velocity ω to store energy. This is an important feature, because slower transients result in lower value of the turbine outlet temperatures. Figure 5.17 shows the results of a simulation with a serious of pulses to the demanded electrical power. The turbine outlet temperature is far from the critical limits and the variations in turbine angular velocity are well within acceptable limits.

CO_2 Refrigeration Cycles

A system simulation of future on-board cooling systems for airliners is currently developed in an ongoing research project of European Aeronautic Defense and Space Company (EADS) Airbus, Hamburg (Germany) and the Department of Technical Thermodynamics of the Technical University Hamburg Harburg (TUHH). The aim of the project is a proof of concept of integrated cooling systems using the rediscovered refrigerant CO_2. Carbon dioxide was used as a refrigerant until the 1930s, but

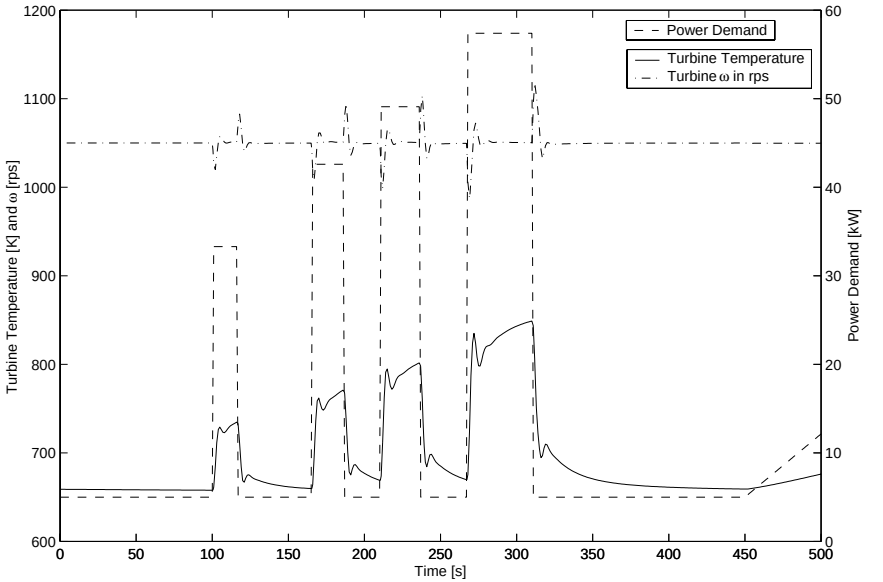


Figure 5.17 Simulation results from applying demand power pulses to the micro turbine in an island power operation mode.

was then replaced by synthetical refrigerants that offered lower absolute pressures which in turn permitted simpler techniques and higher efficiencies in conventional vapor compression systems. Due to the high global warming and ozone depletion potential of synthetical refrigerants, a substitution of synthetical HCFCs is the aim of current research efforts, see [Pfafferot and Schmitz, 2002; Pfafferott and Schmitz, 2001].

The temperature and pressure at the critical point of CO_2 are $304.13K$ and $7.377MPa$. The refrigerant cycle has to be operated transcritically when the heat rejection takes place at an ambient temperature which is near or higher than the critical temperature. In aerospace applications, the heat rejection temperature is low in flight, resulting in a condensating mode, and high on the ground where the cycle operates transcritically.

The ThermoFluid library contains high accuracy fluid property calculations for CO_2 published in [Span and Wagner, 1996] which are indispensable for detailed numerical studies of refrigeration processes. Unfortunately, the critical point singularity of such fluid property models make it impossible to use such published standards "as is" for simulations that pass the critical point, which is the case for transcritical CO_2 cycles. Two alterations to the original formulation help to increase speed and robust-

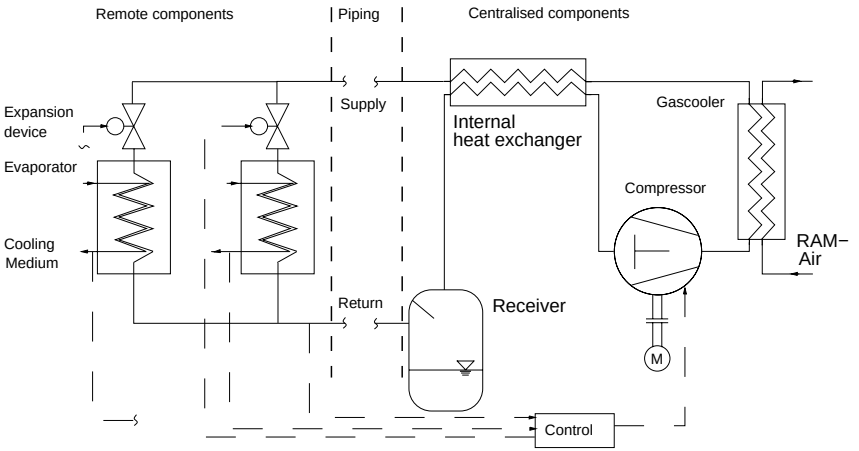


Figure 5.18 Schematic of a CO_2 refrigeration cycle with distributed cooling loads. This configuration is used for on-board cooling systems of airplanes. Diagram used with permission from [Pfafferot and Schmitz, 2002].

ness of such calculations:

- Use of spline approximation to all properties on the phase boundary. These can be made very accurate depending on the number of interpolation points on the phase boundary. The speed-up obtained by using splines is substantial because it avoids phase equilibrium calculations.
- Move the critical point used in the numerical calculations to a slightly higher temperature (fractions of a Kelvin). This and the use of splines inside the 2-phase region avoids the singularity. The change in accuracy is negligible for system simulation and the calculations are robust because numerical failures at the critical point are avoided.

Pfafferott and co-workers developed a CO_2-flow application library on top of ThermoFluid. They use base classes from ThermoFluid for all dynamic model parts. Heat transfer and pressure drop correlations for the specific types of heat exchanger geometries used in CO_2-cycles had to be developed. The models for evaporator, gas cooler, valves and pipes were assembled from partial models in the ThermoFluid library. Using the ThermoFluid library allowed the developers to concentrate on the application specific model parts.

Figure 5.19 illustrates that system start-ups can be simulated without problems. A few seconds after switching on the system, the thermody-

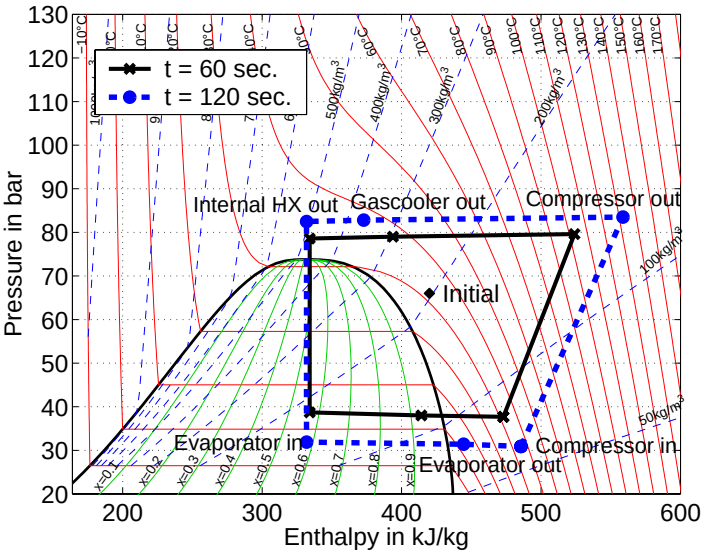


Figure 5.19 Steady state paths of two operating points of main components of a CO_2 refrigeration cycle after start-up. Diagram used with permission from [Pfafferot and Schmitz, 2002].

namic state of CO_2 crosses the critical point without numerical problems.

Steam Network Modeling

The Swedish company Solvina has used the ThermoFluid-library to develop a simulation model of the steam distribution network of the paper plant at Iggesund. The existing steam network was improved by adding two turbines, new control valves and pressure sensors. A new control system was also installed. The complete network with several boilers, turbines and a large steam network having four pressure levels was modeled from existing components in the ThermoFluid-library. The model was used for two purposes:

- To build an operator training simulator, with LabView as a graphical user interface that mimics the real control system.
- To find suitable controller parameters for all important control loops in the steam net.

Both of these tasks contributed to an error-free start-up of the plant and an equally unproblematic operation since then. None of the simulated control loops needed tuning during start-up, see [Lindstrand, 2002].

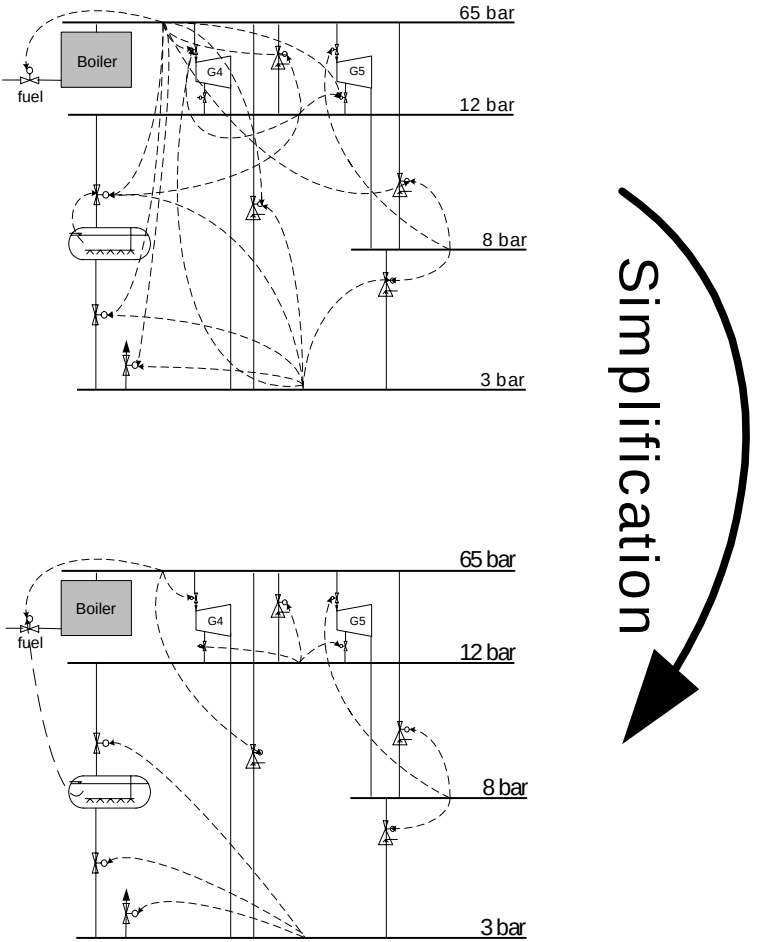


Figure 5.20 The steam net control system was extensively simplified with improved dynamic performance using the simulator. Figure courtesy of Solvina AB.

Analysis of the simulated control system revealed that the structure of the existing control system could be simplified significantly, as shown in Figure 5.20.

Fuel Cell Systems

Fuel Cell engines for power generation and automotive applications is a very active area of research and advanced development. Even though fuel cell technology has been used for many years, e. g., for combined power and

water supply for space applications, fuel cells have not yet reached a development status which allows them to be used in every-day applications. The ThermoFluid-library has been used at United Technologies Research Center, Hartford, CT, to develop dynamic models of fuel cells including the complex fuel reforming unit. The reformer transforms hydrocarbons like natural gas or even gasoline to hydrogen and other byproducts. The hydrogen is used in the fuel cell to generate electric power.

The ThermoFluid library was not designed with fuel cells as a prospective application, but as a base library it offered models for most of the phenomena which are relevant for fuel cell modeling. The combination of the Modelica language, ThermoFluid-library and Dymola simulation environment was evaluated against a range of other possible modeling tools and techniques and was found to best fulfill all requirements of a flexible base to develop models for a range of different fuel cell products currently under development. The models are used both by members of the research department who develop the models of subsystems and libraries and by engineers at UTC Fuel Cells for development work. The fuel cell system models are the most complex systems built so far based on the basis of ThermoFluid. The models are also used for hardware-in-the-loop investigations of control system performance.

5.11 Comparison with Domain Specific Tools

Some examples where the ThermoFluid library has been used for dynamic modeling of systems were presented in Section 5.10. In all examples there exist domain specific modeling tools which provide functionality similar to the combination of Modelica and Dymola. It is interesting to compare these cases and identify advantages and disadvantages specific to certain tools.

The discussion will be focused on issues related to model representation and the support of modeling techniques. One of the important aspects of Modelica is the fact that it is a standardized language with readable, declarative model definitions. Many uses of models in research and advanced development require that the model definitions are readable and can be adapted as needed. Simulators with black-box models do not have that property.

One advantage of domain specific tools is that large model libraries are included which may be sufficient for many applications. Another advantage of domain specific tools is that they integrate the data and work flow over different stages of a domain specific plant design. Static design optimization, plant topology definition, dynamic simulation and possibly dynamic optimization should use the same model data and work together

seamlessly. Because most of these tools emerged independently and were integrated later on, the work flow and data consistency across different tools is much less integrated than software vendors pretend.

The comparison is meant to point out the main features of similar tools and their differences. A thorough comparison of all the differences would require more space than given.

gPROMS and ABACUSS II

The acronym gPROMS stands for general PRocess Modeling System. This program consists of an equation based modeling language and environments for simulation and optimization. It was originally developed at the Department of Chemical Engineering at the Imperial College of Science, Technology and Medicine in London. Later, it was commercialized in the spin-off company PSE Ltd, Process Systems Engineering. ABACUSS II is derived from the original gPROMS code with some extensions by one of the developers of gPROMS. It is still an academic code used for research, mainly in the area of dynamic optimization of large, combined continuous and discrete differential-algebraic equation systems.

ABACUSS II and gPROMS are the only simulation environments in this comparison which are based on a declarative modeling language with equations. Some properties of simulation environments are closely tied to the existence of a high-level, abstract model description format that allows symbolic manipulation of the model before execution.

The following discussion will distinguish between a comparison of language features and other features concerning the simulation environment.

Table 5.3 summarizes the important language features of gPROMS and Modelica. Most of the differences can be attributed to the different goals of the languages: Modelica was designed as a general purpose multi-domain language and gPROMS was designed primarily for chemical process engineering systems. Other important properties for library design can not be compared on the language level: Dymola's Modelica implementation handles high index problems by automatic index reduction. This is not the case in gPROMS, but would be possible, as demonstrated by the ABACUSS II implementation of the same language. The original version of gPROMS described in [Barton, 1992] had a keyword INHERIT and single inheritance as an object-oriented feature. However, no references to the keyword or the concept occur in recently published material about gPROMS. Both ABACUSS and gPROMS do not generate code from the equations, but instead they build up a data-structure in computer memory which is evaluated when simulating. This results in faster turn-around times in model change and test cycles, but leads to somewhat longer simulation times. Since gPROMS is interpreted, execution is slow and memory requirements for the interpreter are high. Both properties are a severe

drawback for hardware-in-the-loop simulations.

The first modeling language to include a notation for partial differential equations in the language was gPROMS. The PDE language extensions to gPROMS, the types of discretization schemes, geometries and boundary condition specifications which are currently implemented are presented in [Oh, 1995]. Integration of language constructs for partial differential equations is still under development in Modelica, see [Saldamli *et al.*, 2002].

ABACUSS II uses almost the same language as gPROMS, but the keyword SENSITIVITY was added to tag the variables whose sensitivities have to be calculated. ABACUSS II makes it also possible to use external FORTRAN routines. Automatic differentiation techniques are used to make external routines behave as similar to equations as possible.

Extending from its original purpose, gPROMS has become part of a suite of tools for modeling, simulation, identification and optimization of chemical engineering processes. Currently, a graphical user interface for the system topology is added to the originally text-based modeling envi-

feature	Modelica	gPROMS
Equations	true equations	true equations
Connections	equations generated by connect-statements, using zero-sum equations for flow variables	connections of streams, all variables use equality assignment at the connect.
Partial differential equation support	no	yes, but only for simple geometries: cylinder and cube.
Hybrid models	yes	yes
Object Orientation	yes	no
High-level model parameterization	yes	no
Graphical layout is part of language	yes	no

Table 5.3 Comparison of language features between gPROMS and Modelica. The ABACUSS II language is almost identical to the gPROMS language.

ronment. This is a difficult task because the graphical representation is not integrated with the modeling language.

ABACUSS II uses other numerical solvers than gPROMS. The solver-collection is called DAEPACK and offers particular features for parametric sensitivity analysis and detection of discontinuities, see [Tolsma and Barton, 1999] and [Tolsma and Barton, 2002]. No graphical modeling environment is currently available. ABACUSS II is one of the few tools that integrate automatic differentiation techniques for FORTRAN with the modeling tool. This is useful in process engineering simulation where it is common to include legacy FORTRAN codes for parts of the problem. Many optimization techniques require differentiation of the model and the focus of ABACUSS II on dynamic optimization makes the automatic generation of derivatives a valuable asset.

APROS

APROS is a commercial, closed source simulation tool for simulation of power plants developed by VTT Finland, see [Juslin, 1995]. Large system providers for power plants also have in-house simulators for these plants which are used both for process development and for operator training. APROS is a well known vendor-independent full scale power plant simulator.

APROS has a broad selection of validated models for standard power plant configurations. Due to its closed black-box structure it does not offer the same flexibility as Modelica based tools. Composing an operator training simulator for a standard power plant is a straightforward task with APROS, but the flexibility of the model use that made Dymola and Modelica attractive for the micro turbine system described in Section 5.10 is lacking. The closed source nature of APROS models makes APROS less suited for control system analysis and design purposes. Extracting reduced order models and systematic model reduction are not possible with black box modeling tools.

The structure of the APROS models is similar to that of the ThermoFluid models: "The primary state variables of the thermal hydraulic nodes are pressure, enthalpy, and component mass fractions, and for branches flow velocity. Material property functions are used in calculating various quantities, such as density and viscosity, according to pressure, enthalpy and component mass fractions"[4].

On the other hand, APROS offers models of different levels of fidelity for each component, which can be selected on a component level to get a good compromise between fidelity and simulation speed. APROS has a communication library to connect the simulator to a distributed control

[4]information from the APROS web site at http://www.vtt.fi/aut/tau/ala/apros.htm

system. This makes it easy to test new or changed control laws against the simulated plant model. For many simulation applications, APROS provides good solutions when all required process models are available in the APROS model library.

FlowMaster 2

FlowMaster 2 is a commercial, closed source simulation environment for systems with internal flows and heat transfer. It has a graphical user interface for model composition and a database of commonly used components. Analysis modules are chosen from application specific packages, e. g.,

- Automotive thermal analysis
- Automotive lubrication
- Liquid piping systems
- Gas piping systems
- Aerospace fuel systems
- Air conditioning

The possible application domains of FlowMaster are almost identical to those of ThermoFluid and Dymola, the difference is the degree of specialization of the tools, the scope and philosophy of the model libraries and the modeling approach. The differences compared to ThermoFluid are very similar the ones between APROS and Dymola/ThermoFluid.

The main advantage of specialized domain specific tools are:

- Integration of all tools and utilities needed for the usual engineering tasks in that domain. Equally important is the fact that no unneeded features of a general purpose tool are cluttering the user interface and increasing the learning time.
- The existence of comprehensive model libraries for a very specific domain. A model library for automotive lubrication is much less general than ThermoFluid but gives results much quicker than building a specialized library on top of ThermoFluid.
- Numerical routines which are specifically adapted to a problem domain can be more efficient for that problem. APROS can handle many thousands of states easily while Dymola currently can not.

The main problems with closed source models like in FlowMaster or APROS are twofold:

- It is impossible to find the details of the model equations. Confidence in models and simulation results rest on a questionable foundation, namely the confidence in the tool vendor.
- When no suitable model can be found in the closed, vendor provided libraries, users usually have to ask for expensive consulting services from the tool vendor to provide the missing closed-source models. If the models do not fulfill the expectations, the user ends up in a vicious circle and will be asked to spend further money on consulting.

In conclusion one might say that a combination of the advantages of both tools would be the best solution for users: ease of use and model libraries which are as comprehensive as FlowMaster's and the level of insight and flexibility to develop custom models as in Dymola with Modelica. Obviously the ability to write custom models raises the level for the ease of use for the modeling part. It should however be possible to make the flow-sheeting and simulation part of a general purpose tool like Dymola as intuitive to use as of any special purpose tool. In that case it is probably necessary to have a customizable user interface.

Other Modelica Libraries

The ThermoFluid library can be viewed as a framework of software components that are designed to handle certain problem scenarios well. If the modeling problem in question is outside that range of problems, it may be much better to design or use a simpler but more specific library than to add yet another application to an already complex library. ThermoFluid is made for compressible flows and complete balance equations for mass, energy and momentum. It is possible to choose between dynamic and static momentum balances. Problems that deal only with mass flows or energy flows may be better dealt with using a simpler structure and a more specialized library. A good example of a specialized library focusing on mass balances for quasi steady state, incompressible internal flows which are typical for liquid transport in chemical processes is described in [Fabricius and Badreddin, 2002]. For incompressible flows it is clearly advantageous to have a separate library. Modelica has the option to interface different libraries if the connection is physically reasonable. Even for certain compressible flow problems with a different focus, e. g., no energy balance like in the hydraulics library [Beater, 2002], a separate library is preferable to over-extending the application range of ThermoFluid.

There is an unavoidable compromise between a very general library and a specific one which has to be considered in each particular case:

- A general library will always have extra overhead in terms of additional parameters, variables or interfaces which are not needed in a specialized model library.

- The generality of a library leads to a higher degree of abstraction and a more fine-grained model structure that are obstacles to new or occasional users.
- Specific libraries have limitations in scope. They are not useful for other or mixed domain applications.

5.12 Summary

In the design of the base library, the concepts of object-oriented modeling have been used to make the library flexible and easy to use. Generalization splits a complex problem into subproblems, which are modeled individually (e.g., balance equations, momentum equations, heat transfer) and aggregated to build component models. This separation simplifies library maintenance and promotes flexibility. Model variants can be assembled more easily. The concept proved particularly useful for a base library that covers a broad range of applications.

The Modelica language offers standard object-oriented features, such as composition and inheritance as well as more advanced features like class parameterization. Using these, the basic properties of thermo-fluid modeling are embodied in the library models, but they are still kept flexible and extensible through specialization and class parameterization. The decomposition of models sometimes makes it difficult to get an overview over inherited parts of a model. However, the advantages of a more maintainable structure and reusable classes outweigh this disadvantage. Readability can be remedied by improved tools for code browsing.

Some further conclusions:

- **Ease of use:** Taking the user perspective early in the library design process is important for the final result.
- **No overkill:** There is a risk of over-structuring using object-oriented methods. Overview can get lost if the structuring is too fine grained.
- **Tool support:** Currently no Modelica tool implements a user interface that makes advanced Modelica features like class parameters available to users that do not master the language well.
- **Nomenclature of research field:** Use of known symbols is very important for the usefulness of the library.

It has been demonstrated that it is possible to model complex systems with the base classes and components implemented in the library. The modeling and simulation tool Dymola has been used in the design of the

library. Dymola has a graphical user interface that allows drag and drop model editing, making the modeling process easier. The library serves well for the purpose of a base library, but there is much room for extensions as the projects using ThermoFluid demonstrate.

6

Design of Model Libraries

Abstract

This chapter gives some guidelines on structuring object-oriented model libraries. There are no unique solutions to that problem, but the idea is to capture key issues and proven solutions in a collection of *Design Patterns* which can be applied to other modeling problems. Design patterns are an attempt to describe "good practice" in a semi-formal way. Most of the design patterns are applicable to modeling in general, but a few are specific to Modelica or thermo-fluid systems. Examples using Modelica and the ThermoFluid library are used throughout the chapter to illustrate the ideas.

6.1 Introduction

Designing a user-extensible model library is different from building models for one time use. The extensibility is a feature which is not needed to the same degree in different engineering disciplines. When modeling electrical circuits, the models often consist of a large number of components, but each component is described by a few well defined mathematical models, all of which can be made available in an extensive component library. This means that a user typically composes system models from the library, but has no need to alter existing models or write new ones. The situation is different for thermodynamics and process engineering, particularly when the scope of the models is as broad as with the ThermoFluid library. The variety of heat- and mass-transfer equations and physical property data is broad by itself. The number of possible variants grows exponentially if different combinations of these are taken into account as well. Very often, modeling requires a problem-specific simplification which does not hold for other problems. Even a huge component library can not cover all variants that a modeler needs for routine modeling use. A library for thermodynamics has to be open for user defined extensions. Designing

a user-extensible library in such a way that it is powerful, flexible and easy to understand is a difficult challenge. Object-oriented decomposition of engineering system models into subsystem and component models follows the same decomposition as that of the system itself: the elements found on the blueprint or plant flow sheet should be library models. This decomposition has been discussed in [Nilsson, 1993] for process engineering and in [Mühlthaler, 2000] for thermal power plants. Much more code reuse can be achieved when the decomposition is continued to the level of physical phenomena. Library design for reuse at the level of phenomena is the topic of this chapter. Examples refer to thermo-fluid applications and cover the same models as ThermoFluid. The conceptual structure is emphasized instead of the details of the actual implementation[1].

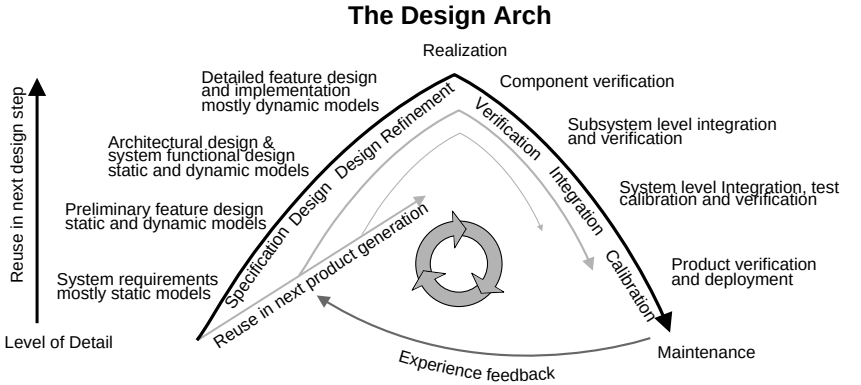


Figure 6.1 Model reuse along the design arch. The development phases are typical for a complex, highly developed product like a car. A similar scheme is sometimes referred to as design-V.

For an engineer, the foremost goal of a system model is to be mathematically correct and to fulfill the requirements in terms of accuracy and simulation speed. Due to the tedious work necessary to verify and validate models, this is a long term process. This model validation is called *calibration* in the automotive industry while the control community refers to the same process as grey-box parameter identification. Some parameters are always uncertain, so for every new system the user has to check again if the simulation gives a "good enough" picture of reality. Parameters have to be adapted to make simulation model output fit to measurement data. Many engineers focus on validating and calibrating their models and tend to neglect structuring and reuse issues. Proper code structuring

[1]A detailed and complete documentation is found at http://www.modelica.org/library.

has proved to increase programming productivity in software engineering. The cyclic nature of modeling in the iterative design of technical products from one generation to the next makes it obvious that reuse will speed up the modeling process substantially. Looking at Figure 6.1 reveals that there are two dimensions of reuse:

1. on the same level of detail for the next product generation and
2. along the path of the "Design Arch" in Figure 6.1, spanning different levels of detail during the same design cycle.

The second dimension of reuse is more difficult to achieve because the mathematical models and the time scales of interest are often different for another level of detail of a system model. The granularity of the system topology can vary along the path of the design process. It is common practice to neglect components with little influence on the system dynamics.

The main incentive for the development of model libraries is the cost savings from code reuse. The estimations for the cost of software development vary widely, in a recent report it was claimed that commercial software goes at a tariff of USD 50 – 200 per line of debugged code. For validated code in a modeling language, the numbers are probably higher. This makes it obvious that validated model libraries of industrial relevance are a valuable resource.

6.2 Means for Library Structuring

Code structuring has been discussed from a language perspective in Chapter 3 and with concrete examples from the ThermoFluid library in Chapter 5. Building on those examples, we will now illustrate how the language tools can be used for model libraries in general. Modeling always offers several ways to solve a problem, but in spite of the many possibilities, similar solutions for the mathematical parts and code structuring are found repeatedly even if the modeling languages are different. Building on power plant library modeling in SMILE and the broader scope of ThermoFluid in Modelica, experiences from designing object-oriented model libraries are summarized.

The advantages of using libraries is to reuse as much well-tested code as possible in models. This minimizes the need for extended testing. Validating and calibrating a model is usually the most time consuming part of the model development process. While there is no way to avoid the so called calibration which consists of adapting the model parameters to plant data via systematic or heuristic methods, testing and internal validation can be substantially reduced when well tested code is reused. For

simulation of standard problems and plants, it is possible to rely exclusively on tested model components. This greatly increases the trust in simulation results.

For the ThermoFluid library, two scenarios of reuse in model development have been considered:

- Use a partial component and complete the model by filling in the missing pieces.
- Start from scratch and build up a model using as many base classes as make sense.

Clearly, using partial components is faster but the partial models may not be available. Building models from base classes is more flexible, but takes longer and needs a thorough understanding of all used classes. When almost complete partial models are used, a developer only needs to know the interfaces and requirements of the missing parts. Readers unfamiliar with object-oriented techniques are recommended to take a look at the glossary in Appendix A in order to get an overview of the vocabulary.

Encapsulation

Information hiding is an important principle to improve the maintainability of programming code. The idea is that all interaction between models occurs via well-defined interfaces. If this principle is neglected, the interdependence between models is likely to increase. That in turn makes it more difficult to change the system model and both flexibility and maintainability are lost. Encapsulation of operations is also a property which makes it easier to debug faulty programs.

At first sight this may not seem like such an important issue. Experience with typical codes for engineering models in industry which evolved over many years shows that these codes tend to mutate to almost unmaintainable spaghetti-code. The main problem in maintainability is undocumented interdependence of different parts of the code, which is difficult to detect. This makes it impossible to find an evolutionary solution to adapt the code to new needs. Many companies depend on the functionality of the code, but when the last of the original developers leaves the company, a complete re-engineering has to be undertaken. Proper encapsulation techniques make it much more likely that an evolutionary solution simplifies the transition to new tools and methods.

Encapsulation in equation based modeling is fundamentally different from encapsulation in object-oriented programs, where interaction is mostly based on message passing between objects. All operations and the data they operate on are encapsulated in objects. In equation based modeling, all components of a system are linked together via a bipartite graph

that connects variables and equations of the differential algebraic equation system. This makes it impossible to speak of encapsulation of operations: an additional equation in one component can be compensated by adding a variable in another component, if the bipartite graph connecting variables and equations allows to match them. As an example, consider a system of a large network of incompressible, internal flow with three boundary conditions defining the interaction with the environment. Two types of boundary conditions can be given: either mass flows or pressures. One of the many possible configurations for boundary conditions is erroneous: when all boundary conditions are mass flows, the pressure inside the network is arbitrary, creating a so called "floating potential" problem.

Thus it is impossible to localize the error to a specific component: any one of the existing boundary conditions could be exchanged against a pressure boundary condition, or an additional pressure boundary condition could be added to remedy the non-physical situation. The equation system glues all components together in a way that the problem could be fixed by providing a pressure anywhere in the system[2].

This does not mean that attempts to encapsulate components or data are useless or impossible. Parameters can be encapsulated in models and access to them can be restricted. Modelica's main tool to achieve encapsulation is to use connectors to couple models. But the main strength of acausal modeling – that the direction of the information flow is not determined in the model, but is derived from the boundary conditions of a complete experiment – makes it difficult to debug models. Variables which are equated in a connection can be calculated in the models on either side of the connection. Therefore it can be useful to impose certain rules to make model debugging and system composition simpler. This has been done in the ThermoFluid library with flow models and control volumes. Flow models never calculate the fluid properties and always compute the flow variables in their connectors, similar rules hold for control volumes.

As in many other programming languages, readable and maintainable program code can not be enforced by the language. Coding style is an important element to achieve safe, maintainable code, as has been pointed out by [Summerville, 2000]. Local parameters and variables in a model can be declared as protected, which means that they can not be accessed by dot-notation from the outside and not be modified, see Chapter 3. This restriction makes debugging easier and prohibits misuse of dot-notation access. Following a design guideline consequently reduces the training time for new users. The Balance-models in ThermoFluid take care of all

[2]Assuming the model has the same number of equations and variables, it has to contain one additional equation for a mass flow as well. It is equally difficult to localize this additional equation.

interaction of a control volume with its environment. It is important that no other functionality is implemented there, this would make it more difficult to get an understanding of the role of each model class. Due to the acausal nature of equations it is impossible to enforce encapsulation of equations in partial components for a library developer who provides partial models. A complement that makes model usage easier is to postulate rules for partial library models and document them extensively.

Inheritance and Aggregation

Inheritance is one of the main tools for achieving reuse both in object-oriented software development and modeling. ▷*Inheritance*[3] ensures that code which is common to several models only appears at one place in the source code, meaning that it only needs to be documented once and maintained once in case of changes. But overuse of inheritance has a few drawbacks. Experience with the design of both software systems and model libraries has shown that the total number of classes in overly structured libraries becomes large. Understanding is difficult and a long learning time is the consequence. Large libraries can not be avoided for complex systems and a broad application scope, but often the large number of classes is caused by too extensive use of inheritance. If all variants of a class of models are derived via inheritance of a base class, many classes are needed. ▷*Aggregation*, on the other hand, makes it possible to add small units of functionality as needed. The difference in complexity becomes obvious when *combinations* of these different units are considered. This will be demonstrated with an example from the ThermoFluid library. The problem is to provide base classes for flow models with one inflow and one outflow e. g., distributed pipe models, but also lumped stirred tank reactors. We consider four optional phenomena which may or may not be required in the model:

- heat transfer interaction,
- dissipative work interaction from a stirrer,
- chemical reactions,
- diffusion through a membrane adjacent to the control volume.

Two alternative designs are considered, one which only uses inheritance and another option which uses both inheritance and aggregation. The class structure of both alternatives is illustrated in Figure 6.2 and Figure 6.3. The usage of the resulting library models is slightly different, so the figures do not cover the same ranges of model behavior.

[3] First occurrences of important terms defined in the glossary are marked with a triangle and typeset in ▷*slanted*.

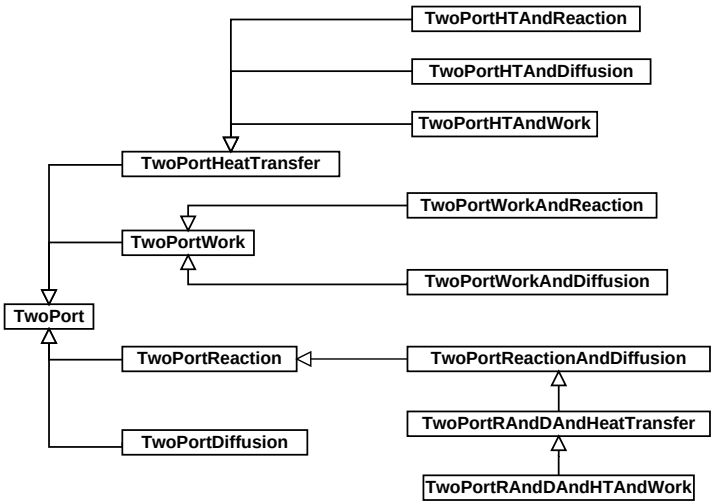


Figure 6.2 Design alternative for TwoPort bases classes using only inheritance. Not all possible combinations are included. The picture makes it obvious that inheritance does not lead to a simple library structure.

Figure 6.2 demonstrates what happens when only single inheritance is used to provide base classes of many model variants which in principle can be combined in an arbitrary fashion. In spite of the large number of classes in the graph, not all possible combinations are present, other combinations are included even if that particular combination is very unlikely to occur in practice. For example, a control volume with both dissipative work and membrane diffusion is very unusual in practice, but from a systematical point of view it should be part of the class structure. It should be kept in mind that none of the classes in Figure 6.2 implements any specific heat transfer or reaction mechanism, they just provide the interfaces.

Figure 6.3 shows the structure of the actual implementation in ThermoFluid, see Section 5.4 for more details. It uses a combination of single inheritance, aggregation and class parameters to achieve a structure which is both powerful and simple. Inheritance is used to specialize a general TwoPort to a TwoPort with heat- and mass transfer interaction. A TwoPort can be used as a base class for models with one inflow and one outflow, but also for more complex subsystems with one inflow and one outflow, e. g., a drum boiler composed of several simple models. TwoPortWithInteraction does not make sense as a base class for a subsystem, but it can be used for control volumes of different types. The instance of a HeatAndMassInteraction model contained in TwoPortWithInteraction

is replaceable, so that simple cases (only heat transfer) don't need to have parts which complicate matters, like reactions. Because membrane diffusion is the least common type of interaction, the diffusion-connector is not present in the default case. The example illuminates different features of

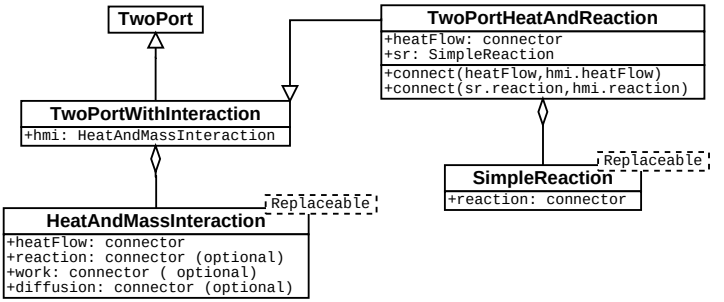


Figure 6.3 Design alternative for TwoPort bases classes using a combination of aggregation, inheritance and class parameters. The graphical notation is explained in Appendix A

.

code reuse using inheritance and aggregation and it also shows that the border between these cases is floating. Generalizing, it can be claimed that

- Inheritance fits very well the paradigm of starting with a very general model which is then specialized step by step.
- Aggregation is useful to cope with optional features which can occur in many combinations.
- Class parameters, discussed in more detail in Section 6.2 can help to keep the simple cases simple while keeping the option for more complex models.

The usage of multiple inheritance (MI) is haunted by the rumor that it adds more complexity than benefits. Multiple inheritance always adds complexity and some possible semantic pitfalls demand more coding discipline. There are however situations when a solution using multiple inheritance is simpler than other alternatives. As many tools for code structuring, multiple inheritance has to be used with care. There are also some fundamental differences between multiple inheritance in programming languages compared to an equation based modeling language. *Repeated* inheritance of the same base class via two inheritance paths, see Figure 6.4 is problematic in some object-oriented programming languages,

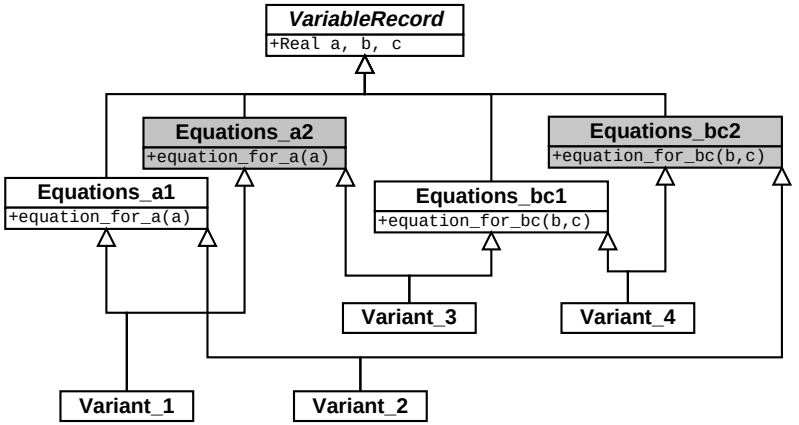


Figure 6.4 Repeated inheritance of the same base class VariableRecord by model variants one through four. Numerical and other modeling issues can be a reason for having different implementations of some equations. When several possible implementations exist and there are good reasons to combine them in a "mix and match" manner, multiple inheritance gives a compact solution.

but does not pose problems in Modelica. Two issues have to be kept in mind with MI in Modelica:

- When declarations with identical type and variable name are found in two base classes, these have to be identical in all components, including modifications. This is a consequence of merging repeated declarations into one without preferring any of them.
- Repeated inheritance works only for classes that do not have any equations (except the definition equations in modifications, which have to be identical and are merged into one equation), because including the same equation twice in a model is *always* a mistake.

Many of the problematic sides of multiple inheritance do not exist in Modelica due to different semantics, others are easy to avoid. In summary the reasons for using multiple inheritance in Modelica are:

- Ease of combination of ▷*polymorphic* implementations. For equation based modeling this means different equation implementations for the same set of variables. These might have different ranges of validity or numerical properties.
- For so called ▷*mixin classes*: behavior which is not always needed can be added by inheriting from one additional base class.

- Separate the graphical representation from the implementation. This can be used to customize graphical plant schematics, recreate the visual appearance of other simulation programs and similar goals without affecting the model behavior.

EXAMPLE 1—MULTIPLE INHERITANCE

As an example we compare the use of MI with other design options which fulfill the same requirements. Alternative designs will be discussed for a situation similar to Figure 6.4, but with more variants. It is assumed that three alternatives each exist for four equation parts which all operate on the same set of twenty variables, giving a total of twelve partial models. The parts implement different physical features. Some of the features are optional, some can be implemented in different ways. Each of the partial models implements a feature with a few equations using a subset of the common variables. For simplicity it is assumed that all possible combinations make sense from a modeling point of view, giving a total of $3^4 = 81$ possible combinations. The following design alternatives are considered:

Multiple inheritance. With multiple inheritance, twelve base classes are needed, giving an inheritance structure similar to Figure 6.4. The more common of the 81 cases can be provided as ready-to-use models, the others can easily be programmed when needed in a "some-assembly-required" fashion.

Single inheritance. Providing 81 classes using only single inheritance results in much redundant code and many classes, so this alternative can be ruled out.

Component aggregation with connectors. An alternative is to model the partial behavior in twelve components. If all information propagation between the components uses connectors, six connector types are needed and overlapping parts of the twenty variables have to be present in each component. In many cases this gives a lot of overhead which hinders readability. All interaction between components is made explicit with connections.

Component aggregation and modifications. Modifications are used in Modelica to propagate variables from a main model into its component models. Interaction between the container model and the components is achieved via using the propagated variables in equations. Compared to multiple inheritance, the modification code is additional overhead.

□

The actual design of a similar case in ThermoFluid are the control volume models which make use of a mix of all four structuring alternatives, taking advantage of their respective strengths and weaknesses. A few guidelines can be deduced from the experiences with ThermoFluid:

- It is practical to have optional parts as components because they can be added later on at any time.
- Multiple inheritance is advantageous for parts with a variety of implementations which can be mixed and matched in many combinations. This means also that multiple inheritance is only used to split the implementation of complex physical phenomena inside a single piece of equipment into more manageable parts, but not on the level of system composition.
- Mix-in behavior is a good case for multiple inheritance. In ThermoFluid, the initialization can be regarded as mix-in behavior and is added to the main model with multiple inheritance.
- System composition is always done by aggregation of engineering components using connections for the information exchange.
- Model parts which should be encapsulated can be put into a component. The component can be part of a model which is then used in multiple inheritance.
- Single inheritance and specialization of the child class should be used to finalize a partial model. A control volume class is complete, but some important high level parameters have to be specified for the final model: the type of fluid, the geometry, the heat transfer equation and similar details are defined in a child class using modifications.

Splitting up the implementation of the equations into different submodels does neither contradict nor enhance encapsulation, because the graph structure of the equation system is largely independent from the component structure anyway.

A known problem of multiple inheritance, name clashes and unintentional merging of variables with equal names, is easier to avoid in Modelica than in traditional programming languages. The Modelica typesystem combined with coding discipline make such errors unlikely: two variables typed as `SIunits.Pressure` and `Real` but both named p will cause an error. When both variables are of type `Real` this results in an unwanted merge of the definitions. When all physical variables make use of Modelica's fine-grained typing, such errors are very unlikely to occur.

From a structuring viewpoint, multiple inheritance is closer to aggregation than to single inheritance because it makes it possible to treat

parts of the model behavior as optional. The parts can then be assembled as needed. In object-oriented programming this use of multiple inheritance is called "mixin" class. A detailed example of the use of multiple inheritance and aggregation in ThermoFluid is found in Section 5.4. Similar structural designs can in principle be achieved with aggregation from components and multiple inheritance. The difference is the way the parts interact:

- When model parts are assembled using multiple inheritance, all interaction is implicit in the equations. Interaction is hidden in the bipartite graph that connects variables and equations. Some of the variables have to be present in more than one base class.
- For aggregation, there are three options of interaction:
  - equations in the container model that access variables in component models using dot-notation,
  - propagation of variables from the surrounding model to the components using modifications.
  - use of connectors and connections, either between components or from the surrounding model to a component.

  The last option is the most explicit way of interaction. Connectors result in a lot of overhead for small components with only one or two equations. Components with dot-notation can make equations difficult to read.

In Modelica multiple inheritance often increases the readability of the models because it results in compact code. As [Abelson *et al.*, 1985] put it: "programs must be written for people to read, and only incidentally for machines to execute". This holds equally for modeling languages. A disadvantage shared by both methods of aggregation and inheritance is that it can be difficult to get an overview over the complete set of equations that form the model. An editor that has the possibility to show the "flattened" code and merges all declarations and equations from base classes and components would overcome this drawback.

Class Parameters

Mathematical models evolve partially before and partially in parallel to building prototypes of the real system. This parallelism requires models which are flexible to quickly answer questions that come up during the design process. The most important feature to adapt models to changing needs is flexibility of the model development process. The responsibility

for achieving this flexibility is shared between the modeling tool, the modeling language and the libraries[4]. A Modelica feature that promotes flexibility is the concept of generic classes, usually called ▷*type parameters* or ▷*class parameters*. Using type parameters, models become *polymorphic*, meaning that they can represent different behavior depending on the value of the type parameter.

Type parameters are different from aggregation and inheritance because they do not only provide flexibility during the model development, but they also keep a model flexible all the way to the model user. A complete model ready for ▷*instantiation* can represent vastly differing behavior depending on the chosen type parameters. This illustrates the close connection between language issues and tool issues with respect to model flexibility: a type parameter can select a linear model instead of a nonlinear one, but a tool can equally well automate that process. From a user perspective it may not make a difference whether the linearized model is generated by the tool or built into the library.

For model library design, the first task is to identify the model parts or subsystems which should be kept exchangeable. In the ThermoFluid library, three types of submodels are kept as replaceable models:

- the fluid property calculation in the `Medium` type parameter,
- equations for heat transfer and
- friction pressure drop equations

Replaceable functions are a special case of generic classes. They are approximately equivalent to virtual methods in object-oriented programming languages. Replaceable functions are useful to keep the implementation of functional computations with given input-output relations exchangeable. A good example for a replaceable function is the computation of the isentropic change of enthalpy for turbines, valves, pumps and compressors. No matter in what equipment it is used, it always takes the inflow specific entropy and the outflow pressure as input arguments and it returns the corresponding specific enthalpy.

Type parameters are used in component modifications for propagating a type into hierarchical submodels in the same way as ordinary parameters are propagated. This is a very powerful feature that makes complete system models polymorphic. It is also the safest way to make sure that a type change is introduced consistently at all places where it has to be introduced. An example is a refrigeration system which can be used with different types of refrigerant, e. g., R134a and R22. A user can change the refrigerant type at the system level and the changes are propagated into all components and subcomponents, as illustrated in Figure 6.5.

[4] A more detailed look at modeling tools is outside the scope of this thesis.

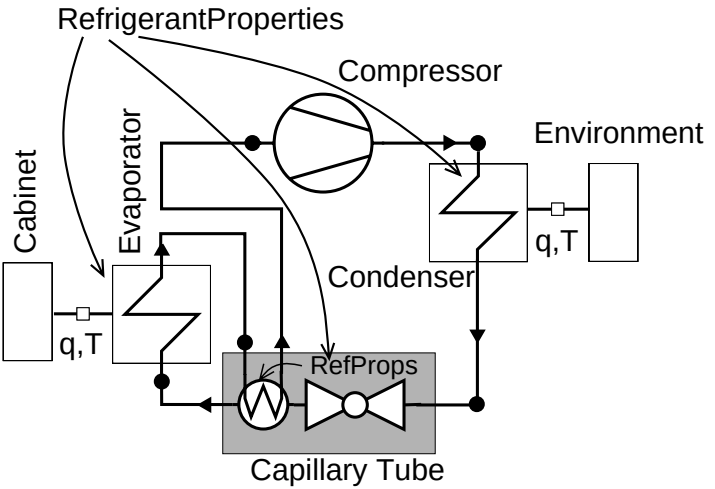


Figure 6.5 A refrigeration system is a prototype case where type parameter propagation makes the model very general. The type parameter for the refrigerant type is propagated down to all levels of the component hierarchy where a fluid property model is needed.

A requirement for building such systems is that the components or types which are going to be redeclared have been declared as replaceable to begin with.

6.3 Design Patterns for Modeling

Software design borrowed the notion of *design patterns* from architecture: there it has been in use for a long time to transfer knowledge and proven solutions to new generations of architects. Design patterns in mathematical modeling address recurring modeling situations by using a library design or modeling language idiom that helps to solve that modeling problem efficiently. The idea is to capture a structuring concept in a catch-phrase that is easy to remember. A design pattern should be sufficiently abstract to apply to many different situations, yet concrete enough to make its application to a particular problem situation obvious. Design patterns for software have been characterized into three categories: Creational Patterns, Structural Patterns and Behavioral Patterns. Mathematical models have rich dynamics, but the run-time code structure of dynamic models is completely static. Creational patterns are not yet implemented for this type of engineering modeling. Some simulation environments with a focus

on discrete event systems permit to create and destroy objects with continuous states during simulation runtime. Usually these are very simple models with few states which are integrated with their own instance of an explicit Euler or Runge-Kutta integrators, e. g., a car on a highway section.

Assuming that the scope and equations of the mathematical models in the library are clear, the task of library design is to divide the models into building blocks with well-defined interfaces, similar to Figure 6.6. Coding guidelines for structuring system models can be classified into two categories:

- **Structural Patterns** for code reuse. These can be classified as physical patterns that abstract physical behavior and topology patterns reflecting system structure.
- **Numerical Patterns** that make sure that solution methods can deal with the models as effectively as possible.

The possibilities for design patterns are closely tied to the features of the modeling language. The existence of equations as independent entities in the language can be seen as a pattern for modeling. The `flow`-prefix in Modelica is a kind of design pattern, derived from a generalization of Kirchhoff's law for electrical circuits to all modeling domains where flows of force, torque, mass etc. follow the same semantics. Consequently, some of the following patterns are specific to Modelica, but others are completely independent of the modeling language and apply equally to FORTRAN subroutines used in a legacy simulator. This holds mostly for the numerical patterns.

6.4 Structural Design Patterns

In mathematical modeling of systems there are two structures that design patterns can refer to: the inheritance based class structure and the mathematical structure of the equation system. The class structure is responsible for achieving code reuse and the mathematical structure is important for computational performance. Most people with experience in mathematical modeling do not at the same time have a background in software engineering. The software design motivated design patterns should therefore be suitable for non-programmers and straightforward to use. The simplicity for the model user is not so much a question of the complexity of the underlying implementation but of how well the simulation tool wraps the concept into an intuitive user interface. Some of the following simple patterns are well known since years and used by many

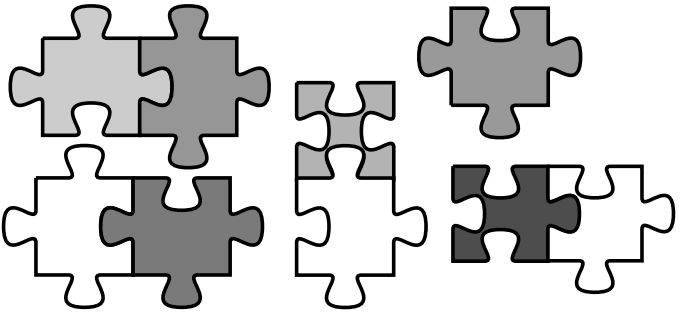


Figure 6.6 Design patterns: Finding abstractions in a class of technical products which are useful jigsaw pieces in many contexts.

modelers, but usually patterns which are well known in one engineering domain are ignored in other domains where they are equally useful.

Unfortunately, many of the design patterns depend both on the expressiveness of the modeling language and the capabilities of the modeling tool. The following design pattern assume Modelica 2.0 as the modeling language and Dymola as the simulation tool. The patterns are extracted from the experiences of developing the ThermoFluid library and not meant to be complete for other engineering domains. Especially the numerical patterns represent the most common pitfalls for non-experts in simulation. Our experience is that 80 % of the questions and problems arising from the use of ThermoFluid would have been avoided if all users understood these numerical pitfalls. The remaining 20 of support requests were caused by "chattering" of discrete modes, an as of yet unsolved problem of combined continuous and discrete simulation, see Section 2.1.

Physical Patterns

Many attempts have been made to make modeling a more systematic activity. All of these attempts emphasize the importance of identifying the *driving forces* or potentials and *flows*. If models are split into submodels and the connections between these submodels are abstracted to have zero volume, then the driving forces on both sides of the connection are equal and the flows are equal in magnitude but opposite in sign. This *flow semantics* is found in all areas of physical modeling, they have among others been used in Bond Graphs and the many extensions to Bond Graphs that try to extend Bond Graphs beyond energy flow, see e. g., [Cellier, 1991] and [Gawthrop and Smith, 1995]. A recent attempt to develop systematic rules for physical modeling which can be seen as physical design patterns

is elaborated in [Weiss and Preisig, 2000].

Flow semantics are not common in other thermo-hydraulic simulation tools, but using flow semantics makes a significant difference for model reuse. For the ThermoFluid library it simply means that a 1-to-5 flow splitter can be realized by attaching 5 flow models to a control volume model. No extra model is needed and due to the flow semantics the mass and energy balances are fulfilled.

DESIGN PATTERN 6.1—FLOWCONNECTOR
Use Modelica flow semantics for transport of conserved quantities for all physical connectors between subsystems. △

EXAMPLE 2—FLOW SEMANTICS
The ThermoFluid library uses flow semantics for three flow types: vector of component masses, flow of enthalpy and flow of momentum. The usage of flow semantics makes it in most cases unnecessary to have models for flow junctions. Splitting a flow into many smaller ones is simply done by using a one-to-many connection. □

Using flow semantics for mass and energy flows avoids errors and reduces the number of classes needed. It also works for momentum flows for simple cases, but due to the simplification of the three-dimensional momentum vector to a scalar, connections with an angle different from 180°C have to be modeled with a detailed model instead of just connecting the flow channels.

Topology Patterns

DESIGN PATTERN 6.2—CONNECTORSET
Provide models with typical connector configurations as base classes. Implement the physics inside them in derived classes. △

This design pattern is very basic and has been used in all engineering Modelica libraries, for example:

- TwoPin is a base class for electrical models such as resistors, capacitances, diodes and many other models,
- TwoFlanges are base classes to all rotational, one dimensional mechanical models with two flanges,
- TwoPorts are the base classes in ThermoFluid with two flow connectors like pipes, valves or pumps.

Similar base classes exist also for other libraries. These base classes are reused using single or multiple inheritance in the base classes.

DESIGN PATTERN 6.3—TYPERECORD
Collect the set of variables and record-components that define type compatibility in a record class. Classes which belong to this type compatible set shall inherit from this class. △

TypeRecord is a simple means to ensure *type compatibility* among a group of models where a basic, simple model is designed to be replaceable by a more complex one if needed. The TypeRecord is used as the *constraining class* in the declaration of the replaceable component or class.

EXAMPLE 3—TYPE RECORDS
Many classes in the package CommonRecords are designed as TypeRecords: collections of variables that characterize a group of models. The class StateVariables_ph defines type compatibility for all fluid property models using pressure p and specific enthalpy h as inputs to the medium property calculations. The TypeRecord makes it easy for other developers to write fluid property models which can replace an existing property model in ThermoFluid without any adapters or interface code. TypeRecord is a fundamental pattern for polymorphic model implementations in Modelica. □

DESIGN PATTERN 6.4—CONSISTENCYMODEL
Collect sets of data, functions and equations that have to stay together for consistency reasons in one replaceable model. △

This pattern is similar to class design in object-oriented programming. In mathematical modeling there are also cases where several functions operate on the same set of data. If these functions should be replaceable, they should by designed in such a way that it is not possible to replace parts that render the whole model inconsistent.

EXAMPLE 4—CONSISTENT REDECLARATION
High accuracy property functions consist of a large number of parameters describing a non-linear thermodynamic surface. Many functions make use of this data: if e. g., a function for the speed of sound and one for the specific heat capacity are needed within the same model, the unit of redeclaration should comprise both the functions and the parameters. □

DESIGN PATTERN 6.5—PARAMETERLIFTING
This pattern ensures that consistency constraints between parameters are enforced when propagating parameters into submodels. △

Geometrical parameters at the interface between two components obviously have to be consistent in both models. This can be achieved with

parameters in connectors – an assertion is generated to make sure that both parameters are identical on both sides of the connect. However, this would lead to a large number of connectors. A better solution is to make the container model responsible for consistency of the parameters. This is best demonstrated with an example.

EXAMPLE 5—PARAMETER PROPAGATION

A heat exchanger in ThermoFluid is composed of a discretized control volume on the hot side, one on the cold side and a solid wall separating them. We assume a tube-and-shell configuration with a cylindrical shell and ten straight tubes. For simplicity, the heat capacity of the outer cylinder is neglected. The given data is:

component	dimension	symbol
tube bundle	number of tubes	N
tube bundle	inner diameter	d_i
tube bundle	outer diameter	d_o
tube bundle	length	l
tube bundle	density	ρ
tube bundle	specific heat capacity	c_p
shell cylinder	length	$L = l$
shell cylinder	inner diameter	D_i

The heat capacity of the simplified single tube wall is computed as

$$C = N\ l\ 0.25\pi(d_o^2 - d_i^2)\rho\ c_p.$$

The volume of the shell side of the heat exchanger is computed as

$$V_{shell} = L(\pi D_i^2 - N\ \pi d_o^2)/4$$

similar expressions hold for other parameters. This re-parameterization has to make sure that:

- All low-level parameters are assigned in modifications to avoid possible sources of errors when setting such values by hand.
- Parameters shared by several components are consistent.
- The top level parameters that a user has to specify are easily accessible from component blueprints.

This may seem to be a trivial issue. However, it is very easy to overlook such parameter dependencies and it is a common source of errors in component based modeling. □

6.5 Numerical Design Patterns

Numerical design patterns address typical numerical pitfalls when textbook equations are used in modeling code. There are many books on numerical mathematics, e. g., [Press *et al.*, 1986] and [Hairer and Wanner, 1996], but they usually do not address the numerical problems in system simulation. The task of a model library designer is to identify typical pitfalls in a domain and provide library models that avoid them. For non-obvious pitfalls the library designer should go as far as to discourage the user to run into them when building models.

In order to be numerically robust, library models should not contain code that may cause problems. In practice, this can not be achieved completely, because many problems are not manifested before a complete model is assembled. Many of the following problems could automatically be detected by tools and warnings could be issued

DESIGN PATTERN 6.6—SINGULARITYCHECK
Make sure that functions with singular points or singular derivatives which are non-physical due to simplifications are regularized properly.
△

Many empirical models return physically meaningless values or have singularities outside their region of validity. For simple system models it is often not justified to model the regions in detail, but it increases the robustness and usefulness of the model if the results are qualitatively correct and the numerical singularities are taken care of. This is sometimes a hack for physical models, but often an unavoidable compromise to keep system models simple. Physical correctness outside the region of validity of the model is sacrificed as long as the qualitative behavior is still correct and larger robustness of the model is obtained.

EXAMPLE 6—SQUARE ROOT FUNCTIONS
A notorious problem in modeling of flow resistances is the use of empirical or semi-empirical flow resistance formulas involving the square root function. An example is a formula derived for turbulent flow with high Reynolds numbers, for which the behavior is often extrapolated to low flow speeds when the exact behavior at low speeds is not irrelevant. They trigger a standard problem when Newton-Raphson algorithms are used for solving non-linear equations. In its simplest form, pressure loss formulas can be written in the following form:

$$f(\Delta p) = \dot{m} - k\ sign(\Delta p)\sqrt{\rho\ abs(\Delta p)} = 0 \quad \Delta p = p_1 - p_2$$

Assume that the Δp has to be calculated from the above equation, e. g., in a zero-volume T-junction, see Section 5.8. Successive approximations

to the solution of $f(\Delta p)$ are obtained from the following iteration:

$$\Delta p^{j+1} = \Delta p^j + \frac{f(\Delta p^j)}{\frac{\partial f(\Delta p^j)}{\partial \Delta p^j}} \approx \Delta p^j + \frac{f(\Delta p^j)}{\frac{\Delta f(\Delta p^j)}{\Delta(\Delta p^j)}}$$

The step sizes of the Newton method depend on the approximated or analytically computed derivative of $f(\Delta p)$ with respect to Δp. For Δp close to zero the derivative goes to infinity and the step size goes to zero. This means that the iteration progresses very slowly. This phenomenon is sometimes called *inflection* because it occurs at inflection points of a curve with an infinite derivative at that point.

The singularity of the pressure drop function near the origin has no physical significance. Therefore it is perfectly reasonable to replace the singular formula with an approximation that does not cause numerical problems. The approximation should of course be correct qualitatively and it should not influence the system behavior more than necessary. In the ThermoFluid library, third order polynomials are used in a neighborhood around zero flow. The polynomial coefficients are chosen such that the overall function is continuous with continuous derivatives. □

EXAMPLE 7—THE LOG-MEAN TEMPERATURE

Another example where a careful implementation is needed is the *log-mean temperature difference*, ΔT_{lm} which has a statically correct behavior for heat transfer in heat exchangers. This means that heat transfer is calculated based on:

$$\Delta T_{lm} = \frac{\Delta T_1 - \Delta T_2}{ln(\Delta T_1 / \Delta T_2)}$$

where ΔT_1 is the temperature difference at one end of the heat exchanger and ΔT_2 is the temperature difference at the other end of the heat exchanger. Dynamically and under start-up conditions, the temperature gradients can be reversed for short times. It does not make sense to use the log-mean temperature difference when the signs of ΔT_1 and ΔT_2 are different. A numerically robust implementation has to take care of this case, too. Singularities or numerical ill-conditioning occur when:

- $\Delta T_1 \approx \Delta T_2$ and
- either $\Delta T_1 \approx 0$ or $\Delta T_2 \approx 0$.

The singularities near zero can be treated in the same way as the flow singularity above, the case of $\Delta T_1 \approx \Delta T_2$ can according to [Mattsson, 1997] be treated as follows. When $|\Delta T_1 - \Delta T_2| < max(|\Delta T_1|, |\Delta T_2|)$ it is better

to use the Taylor expansion

$$\Delta T_{lm} = 0.5(\Delta T_1 + \Delta T_2) \times \left(1 - \frac{1}{12}\frac{(\Delta T_1 - \Delta T_2)^2}{\Delta T_1 \Delta T_2}\left[1 - \frac{1}{2}\frac{(\Delta T_1 - \Delta T_2)^2}{\Delta T_1 \Delta T_2}\right]\right)$$

□

Scaling and normalization are important techniques which traditionally are used to improve numerical calculations. It is often advantageous to use dimension free variables and parameters. Many design patterns for model derivation are based on scaling and normalization. Many books on modeling elaborate on scaling and dimension-free quantities as fundamental modeling techniques, see e. g., [Lin and Segel, 1988]. Different engineering domains have different traditions regarding these techniques: in some domains it is common to always normalize models, in others unscaled values are used. Normalization is not common in thermo-fluid systems and process engineering and therefore ThermoFluid uses non-normalized quantities.

Some types of numerically motivated scaling have to be done by the model developer, for example the following:

DESIGN PATTERN 6.7—SCALING
Scale extremely nonlinear functions to improve numerics. △

EXAMPLE 8—SCALING OF EXPONENTIAL FUNCTIONS
Chemical equilibrium reactions often contain exponentials of temperature functions as the equilibrium constant. The dissociation of hydrogen at high temperatures, $\frac{1}{2}\ H_2 \leftrightarrow H$, can be described by the following equations:

$$k = 2.6727 - \frac{11.247}{T} - 0.0743\ T + 0.43170\ log(T) + 0.002407\ T^2$$

$$x_H = \frac{\sqrt{x_{H_2}}}{\sqrt{p}}\ e^k$$

where x_H is the mole fraction of atomic hydrogen, x_{H_2} is the mole fraction of molecular hydrogen, p is the pressure in atmospheres and T is the temperature in Kelvin. At low temperatures, the mole fraction of atomic hydrogen is extremely small. The second equation is scaled by taking logarithms. This can be achieved by introducing scaled variables, e. g., $logx_H = log(x_H)$. At 1 atmosphere and 280 K, the ratio of the left- and right hand side of the equations is 1.3×10^{75} in the non-scaled variables. This ratio reduces to 172 when logarithmic scales are used. □

Scaling of variables and equations can also be done by the tool, but the tool needs to have information about the ranges and the nominal values of the variables.

DESIGN PATTERN 6.8—VARIABLERANGES
Set tight minimum and maximum and accurate nominal values for all physical variables. △

Accurate minimum, maximum and nominal values can help numerical routines to find the solution and improve the numerical conditioning. Making sure that ranges are set as accurately as possible is thus a part of careful modeling. Many models have mathematically correct solutions which are physically meaningless. Equations for chemical equilibrium always permit solutions with negative concentrations which do not make sense physically. Limiting the search for solutions of nonlinear solvers to the physically meaningful ranges reduces the number of failures and the need for users to provide good initial guess values.

Nominal values are for example important to determine how perturbations should be chosen for numerical linearization of a model.

DESIGN PATTERN 6.9—SMOOTHING
Piece-wise and discontinuous function approximations which should be continuous for physical reasons shall be smoothened. △

In physical modeling it is common to have empirical or semi-empirical formulae that approximate measured data. Non-dimensional numbers are used to describe a given problem with a few parameters. In fluid flow there are many relations for turbulent or laminar flow with considerable uncertainty in the transition.

EXAMPLE 9—HEAT TRANSFER EQUATIONS
Convective heat transfer with outer flow perpendicular to a cylinder is characterized by the following empirical equations:

$$Nu_{lam} = 0.664 Re^{1/2} Pr^{1/3}$$
$$Nu_{turb} = \frac{0.037 Re^{0.8} Pr}{1 + 2.443 Re^{-0.1}(Pr^{2/3} - 1)}$$

These formulas are combined as

$$Nu = 0.3 + \sqrt{Nu_{lam}^2 + Nu_{turb}^2}$$
$$10 < Re < 10^7, \quad 0.6 < Pr < 1000.0$$

with an offset for low Reynolds numbers. The weighted mean is continuous and continously differentiable except at the origin. If this formula is to be used for plant startup, $Re = 0$, design pattern 6.6 SingularityCheck

should be used to make the formula robust at zero flow speed. Note that in this case smoothing acts like a weighted summation because the total Nusselt number is always larger than any of the parts. □

The following two design patterns deal with the *stiffness* of the resulting equation system, see Chapter 2. Stiffness is often the result of composing a system from subsystems. Therefore it is not possible to avoid all stiffness problems with library models. Choosing a solver that can deal with stiff equations may not necessarily be the best solution. Making the equation non-stiff speeds up the solution considerably and broadens the range of applicable solvers.

DESIGN PATTERN 6.10—TIMECONSTANTSELECTION
Make a problem non-stiff by using a steady-state assumption (singular perturbation technique) when the fast dynamics of the system are not of interest. Make a problem non-stiff by approximating very slow dynamics with constants. △

Typical examples of this technique are given in Chapter 4, the assumption of chemical equilibrium for fast reactions in Section 4.9 and using the quasi steady-state assumption for the momentum balance of fluids in Section 4.3. The assumption of taking the volume of a control volume to the limit of zero, used in Section 5.8 for T-junctions, is based on the same considerations. This example demonstrates that the decision to use a control volume model with or without dynamics has to be done on the *system* level by the model user. A related technique is to replace very slow dynamics like fouling with a constant. Modelers should be aware of the fact that it only makes sense to look at a limited range of system time constants at a time. This is reflected in the name to this design pattern.

DESIGN PATTERN 6.11—EASINGSTIFFNESS
Render a problem less stiff by making the time constants of the fast dynamics slower. △

This design pattern has been used for modeling of turbines in power plants, see [Thumm, 1989]. This particular way of dealing with stiffness is better than removing the fast states is explained in Section 5.8. The disadvantage of a singular value perturbation in the steam turbine example is a non-linear equation system involving all tap-off mass flows and pressures in the turbine. This is a purely numerical motivation for knowingly altering the dynamics of the real system. Under certain circumstances, this is a reasonable solution:

- Removing the fast states by a singular perturbation leads to numerical difficulties, typically non-linear equations which can be difficult to solve, especially at initialization time.

- The dynamics after changing the time constants are still much faster than the dynamics of interest.
- The change of the time constants makes the equation system solvable by solvers for stiff differential equations.

This technique can also be used to decrease model stiffness in order to use explicit solvers for hardware-in-the-loop simulation. In the case of the steam turbine discussed in Section 4.7, the ratio of the largest to smallest eigenvalue was changed from 10^6 to 100 without a noticeable influence on the dynamics of interest. From a control perspective this means that the model is rendered numerically tractable by changing it outside the frequency range of interest.

The design patterns EasingStiffness and TimeConstantSelection are addressing the same problem but suggesting different solutions. This shows that experience is needed to choose the best solution. In many cases both of the above methods will work well with similar performance, in other cases one of the methods is clearly superior to the other.

DESIGN PATTERN 6.12—FULLYSYMBOLICCODE
Make sure that symbolic derivatives are available for all model parts, including external functions. This improves the solution of non-linear equation systems and enables automatic index reduction. △

This design pattern is specific to tools which support symbolic manipulation of the model code like automatic index reduction and automatic differentiation, but require that all model equations and functions are differentiable. This is the case in Dymola, MathModelica, ABACUSS II and to some extent gPROMS. High index problems, compare Section 2.1, arise when algebraic equations constrain states to a manifold defined by an algebraic equation. Object-oriented model composition goes hand-in-hand with the need for automatic index reduction, because the constraint equations are the equations generated from a connect statement. An algorithmic way to reformulate the model to an equivalent problem with index one is based on the symbolic differentiation of the algebraic equation. Due to the complexity of object-oriented modeling, a large part of the algebraic equations or functions of a model can become a constraint. Consequently, modelers should provide derivatives for all model functions. In Modelica, this is done with the help of derivative annotations. Compared to using equations, this is additional work for the modeler. However, it can not be avoided for external functions or when functions have other significant advantages over equations.

DESIGN PATTERN 6.13—DISCONTINUITIES
Avoid discontinuities in user defined functions whenever possible. △

A Modelica compiler can easily detect discontinuities in equations, but this is not the case with functions. A function returning discontinuous outputs for continuous inputs will cause serious trouble for numerical integrators. Instead of a function with one discontinuity, two functions should be provided. An event is generated automatically when the functions are called in the branches of an if-expression.

It may be impossible to avoid discontinuities when reusing external functions. The next design pattern applies to external functions with discontinuities or discontinuous derivatives.

DESIGN PATTERN 6.14—EVENTDETECTION
Provide explicit crossing functions for non-smooth external functions. △

The Modelica language and Modelica implementations handle the numerical requirements of hybrid models automatically. Crossing functions are needed by numerical integrators for models with discontinuities in the right-hand side of the differential equations, as outlined in Section 2.1. The automatic generation of the crossing functions does not work for external functions, where the discontinuities are hidden from the Modelica translator. The only way to obtain a numerically robust treatment of such cases is to add a crossing function to the model with the discontinuous external function. The function has the following properties:

- The function needs to have one output which changes its sign at the same location as the discontinuity of the original function. This usually means that it has the same inputs as the original function.
- The crossing function has to be used in such a way in the Modelica code that it triggers an event, see the following listing.

```
model ExternalCrossing "a model using a discontinuous external function"
  ... // other model parts omitted
  function externalCF "Modelica declaration of external function"
    input Real a,b,c "sample input";
    output Real zero_xing "value changes sign at discontinuity";
    external "C" myCCode(a,b,c,zero_xing); // name of the external function
  end externalCF
  Real zero_xing "value changes sign at discontinuity";
  Boolean externalEvent(start=true) "a boolean variable";
equation
  zero_xing = externalCF(a,b,c);
  // the next line causes an event when zero_xing changes its sign.
  externalEvent = if zero_xing > 0 then true else false;
  ... // further equations and functions omitted
end ExternalCrossing;
```

Listing 6.1 Usage of an external crossing function.

This type of problem is typical for interfacing Modelica to traditional C– or FORTRAN codes. A non-trivial example which required to develop the external crossing function and the Modelica interface for multi-phase property calculations is described in [Tummescheit and Eborn, 2002]. Crossing functions for external functions could be avoided with similar techniques as derivatives for external functions. Automatic differentiation techniques, described in more detail in Chapter 7, can not only analyze code to compute the derivatives of that code, they can also be used to detect discontinuities. This has been demonstrated in [Tolsma and Barton, 2002].

6.6 Conclusions

This chapter discussed two issues in development of object-oriented model libraries which are independent of the application domain: code structuring for reuse and numerics. Both issues are important to obtain a flexible and robust library. The numerical design patterns and examples in this chapter are proposals for avoiding difficulties but they do not cover all problems that users of ThermoFluid have experienced. The structural design patterns are also only a few patterns for structuring the model code, but they summarize patterns that were successfully used in ThermoFluid. Experiences from model libraries in other domains indicate that there are many similarities between different domains.

Object-oriented modeling or software techniques are not like a silver bullet that ensures well structured, well documented and flexible code. Discipline in documenting code and adequate use of the object-oriented features are necessary in order to take full advantage of the benefits of object orientation.

7

Recommendations for Future Work

Abstract

The Modelica language has been under active development during the time of the development of the ThermoFluid library. In this chapter, a few recommendation for future work based on the experiences with the development of ThermoFluid will be given. The ThermoFluid library has been under continuous development for several years. Even though it is useful in its current state, there is a lot of room for extensions and improvements.

The declared goal of the Modelica effort is to become a de facto standard for storing modeling knowledge and for model exchange. Trying to establish a standard is costly and time-consuming. The most difficult issue in standardization efforts is that competing companies have to agree on technical details. Another difficulty lies in the implicit dilemma common to emerging standards: as the evolving language definition of Modelica gets powerful and encompasses a large variety of modeling formalisms, it will become complex to implement. When the language gets too complex, the entry hurdle for companies planning to implement Modelica-based simulation environments gets too high. A programming language which suffered from the fate of being to complicated for a complete implementation was Algol 68[1]. On the other hand, if the language definition is too simplistic, it can not cover a large enough share of relevant modeling problems and is not worth implementing for that reason. This dilemma is related to a second one: Some desirable modeling features can either be expressed with *language extensions* or with *tool-specific extensions*. Which of the two possibilities is better is not always easy to decide. In addition, all language extensions need first to be supported by a tool before users

[1]A language that introduced expressions on the left-hand side of assignment statements

can benefit. A grey area between tool and language extensions is the Modelica version of pragmas called annotations.

The following recommendations are partly related to simulation tools, partly to a combination of Modelica language and simulation tool extensions. They focus on topics which are particularly relevant for thermo-fluid systems. Many of these topics have been discussed repeatedly at Modelica design meetings[2]. The Modelica language has reached a state where it is applicable to modeling of complex multi-domain engineering systems, but there are many dynamic system models which are difficult or impossible to express in Modelica. The current Modelica tools focus on dynamic simulation. Model descriptions in Modelica are equally well suited for other interesting uses of models like steady-state design and optimization. One of the ideas of a standardized language is inter-operability between different tools, but there are many other possible levels for integration of tools. Tool extensions for particular engineering domains can also be achieved with standardized Application Programming Interfaces (APIs). For example, a standardized interface definition for numerical solvers would make it possible to customize the numerical back-end of a tool. These issues have been discussed in the Modelica Association, but finalized standards have not yet been produced.

Modeling, simulation and analysis of dynamical systems is a broad field with active research. Everyone who works with dynamical systems on a regular basis knows many open problems and has recommendations for improvements. The following is an incomplete list of interesting topics:

- Language features
  - A formal approach to equation manipulation.
  - Support for partial differential equations.
  - Improved support for derivatives and sensitivities.
  - Description of model uncertainties.
  - Better support for describing a group of variants in a “meta-model”.
  - Better support for graphical discrete event modeling formalisms like Grafcet [David and Alla, 1992] or State Charts [Harel, 1987].
- Programming interfaces
  - An interface definition to numerical solvers for the integration of the complete set of equations.

[2]Minutes of the meetings are published on the Modelica site http://www.modelica.org.

  - Interface definitions to permit different numerical solvers for different subsystems, e.g., PDE.
  - Interfaces to optimization codes. Many types of optimization methods exist, each with different requirements for interfacing.
  - Interfaces to graphical front ends which facilitate implementation of operator training simulators.
  - Facilities for co-simulation with other simulation environments.

- Simulation environment features
  - Automatic differentiation of Modelica functions and in the second step also external functions.
  - Diagnosis and practical handling of notorious numerical pitfalls like chattering.
  - Improved debugging of acausal models.
  - Support for further symbolic equation manipulation methods, e.g., symbolic integration.
  - Improved tool support for Modelica class parameters.
  - Support for saving the manipulated equation system in symbolic form to permit further manipulation and analysis by computer algebra tools and for editing and printing.

- Analysis of dynamical systems
  - Eigenvalue analysis to relate parameter uncertainties to uncertainties in the dynamics.
  - Analysis of the effects of parameter uncertainties in the model.
  - Integration of continuation techniques to detect bifurcations and map the manifolds in parameter space where system behavior changes drastically.

Many of these items illustrate that it is difficult to separate language and tool issues in a clear way. Partial differential equations require language extensions, but also interfaces to special purpose solvers and other simulation tool support. The issues discussed in more detail in this chapter are restricted to those related to language and library design. Developing and testing model libraries is a time consuming effort. Improvements in simulation languages and tools motivate library development. In some cases a new modeling language feature makes an important difference: when the academic tool gPROMS, see Section 5.11, was able to describe partial differential equation problems for process engineering, it

was commercialized and challenged the established simulation tools in that domain.

The modeling process consists of the activities a modeler has to perform during model development. Beginning with the derivation of the equations, continuing with model implementation and finally validating the result are standard steps, the exact procedure varies from case to case. Modeling techniques and tools support this process. The availability of such techniques determines how well and efficient a modeling task can be performed. The issues in the following section treat techniques which were particularly missed during the development of ThermoFluid.

7.1 Writing Models

Assuming that we have already obtained a good insight into the physical phenomena involved, we are faced with the problem of entering the model into the computer. For large systems, this can be a substantial, time consuming effort. In industrial projects the time and effort needed to complete a modeling task are the decisive factors which suggest or rule out simulation as a means of solving the problem. Even in PhD-projects and in model library development the effort-to-result ratio separates feasible from infeasible tasks. The ease of writing a model compared to the utility of the model thus make the difference if implementation in a library is worthwhile. The ease of model writing is influenced by the expressiveness of the model language, the symbolic and numeric capabilities of the simulation tool and by possible tool support in deriving, assembling and testing the model.

Derivatives and Inverses of Functions

During the development of the ThermoFluid library a lot of effort was spent on providing a model formulation which is numerically efficient. The choice of dynamic state variables is a non-linear transformation of the mass- and energy balance into suitable states for simulation, see Section 4.6. The practical result of the transformation is a substantial gain in performance because non-linear equation systems are replaced by non-iterative function evaluation. Such a transformation could also be done automatically by a tool that only has a text-book form of the balance and constitutive equations and understands two additional properties of the model:

- the input-output causality of the physical property calculation routines defining the thermodynamic equation of state (the simple part) and

- the partial derivatives of the property routines with respect to the inputs of the property calculations (the difficult part).

The current status of both the Modelica language and the Dymola tool do not offer the option of performing this automatically. It is possible to use the state selection algorithm in Dymola in such a way that the numerically favorable states are chosen provided that the user knows the numerically preferable form. This requires that the property function is written in a very specific form. In addition, this form may be very inefficient due to the definition of the Modelica **annotations** for derivative functions, as discussed in Section 3.4. Two essential components are needed before such derivations can be done automatically:

1. Automatic differentiation, short AD, [Griewank and Corliss, 1991] and [**?**]is a relatively mature technique to generate computer code that calculates derivatives for given computer code of the function. Several tools are available for FORTRAN [Bischof *et al.*, 1995a], C [Bischof *et al.*, 1995b] or C++ code [Griewank *et al.*, June 1996]. These tools still lack the ease of use and integration into a modeling environment which would be needed in the case of Modelica, but the algorithms for automatic differentiation are applicable to all programming languages.

2. The possibility to specify partial derivatives in a different way than currently possible in Modelica. For functions which are expensive to calculate it should also be possible to return the derivative values and the function value in one function call, because the function value can always be obtained as a by-product of the derivative calculation, see [Griewank, 1991].

The alternative solution of having annotations for partial derivatives would actually not be required with a tool for automatic differentiation that could handle external and Modelica functions efficiently without any user interaction. Most hand written optimized derivative functions would become obsolete.

EXAMPLE 1—A STEAM SUPERHEATER

As an example consider the following simple model of a steam superheater, adapted from [Elmqvist, 1978]. It is a simplified version of the

more general model in Section 4.6. The following equations[3] are given:

$$\begin{aligned}
\frac{dM}{dt} &= \dot{m}_{in} - \dot{m}_{out} && \text{mass balance}\\
\frac{dE}{dt} &= Q_i n + h_{in}\dot{m}_{in} - h\dot{m}_{out} && \text{energy balance}\\
M &= V\rho\\
E &= V\rho h\\
\rho &= g(p,h) && \text{equation of state (EOS)}
\end{aligned}$$

Assume that partial derivatives of the state equation can be generated or their existence can be made known to the symbolic transformation algorithm. Choosing p and h as states, it is straightforward to eliminate the non-linear equation for ρ by differentiating the EOS and the definition equation for the energy. Assuming that the volume is constant we get:

$$\begin{aligned}
\frac{dE}{dt} &= V\left(\frac{d\rho}{dt}h + \rho\frac{dh}{dt}\right)\\
\frac{d\rho}{dt} &= g_h(p,h)\frac{dh}{dt} + g_p(p,h)\frac{dp}{dt}
\end{aligned}$$

This equation can be solved to give explicit differential equations in p and h because the volume V, density ρ and the partial derivative $g_p(p,h)$ are different from zero. This is equivalent to saying that the Jacobian

$$\mathsf{J} = \begin{pmatrix} \frac{\partial M}{\partial p}\Big|_h & \frac{\partial M}{\partial h}\Big|_p \\ \frac{\partial E}{\partial p}\Big|_h & \frac{\partial E}{\partial h}\Big|_p \end{pmatrix}$$

of the partial derivatives for transforming the problem from the states M and E to p and h is always invertible. Introducing, $E'(p,h) = dE/dt$ and $M'(p,h) = dM/dt$ we find that

$$\begin{aligned}
\frac{dp}{dt} &= \frac{-E'(p,h)g_h(p,h) + (\rho + g_h(p,h)h)M'(p,h)}{g_p(p,h)\rho V}\\
\frac{dh}{dt} &= \frac{E'(p,h) - hM'(p,h)}{\rho V}
\end{aligned}$$

An implementation of partial derivatives, e. g., via automatic differentiation, would calculate the function value ρ and the derivatives ρ_h and

[3] In the given example, no distinction was made between inner energy and enthalpy. This is a common simplifying assumption that is justified if the changes in the pressure level of the superheater are small.

ρ_p in one function call. Interestingly, the computational expense of calculating ρ, ρ_h and ρ_p in one call for the IF97 steam tables is about 1 % higher than calculating only ρ_p due to common subexpressions. Automatic differentiation for obtaining partial derivatives could decrease the implementation burden for EOS based physical properties substantially. Even the current implementation could not have been done within the scope of a PhD-project without using computer algebra tools for differentiating the complex multi-parameter EOS for water and refrigerant properties and generating Modelica code automatically. □

This proposed feature in the Modelica language would eliminate the current need for different classes for different dynamic states in the ThermoFluid library. The number of base classes for single fluid control volumes would be reduced by a factor of three (currently three pairs of states are implemented) and for mixtures of fluids the reduction would be a factor of two. The current Modelica annotations for functions only allows specification of *time derivatives* which are needed in index reduction algorithms. The discussed annotations for partial derivatives are prepared as a proposal for adaptation into the Modelica language.

Partial Inverses When manipulating models we are often faced with the following problem. Given the equation

$$z = f(x, y)$$

we are interested in finding functions f_x^- and f_y^- such that

$$x = f_x^-(y, z)$$
$$y = f_y^-(x, z)$$

In analogy with partial derivatives we shall call them partial inverses. Partial inverses are an alternative to differentiation for manipulation of equations.

In an equation based language like Modelica, inverses are not a major issue. They are mainly needed for external functions, because invertible relations are normally written as equations. Example 1 could also have been solved using an analytical inverse of $\rho = g(p, h)$. Assuming the existence of a unique inverse to $g(p, h)$ with respect to the first input variable, the model translator could use the inverse $p = g_\rho^-(\rho, h)$ and choose ρ and h as dynamic states with forward evaluation of the inverse instead of iterating the original equation $\rho = g(p, h)$. This was actually proposed in [Elmqvist, 1978].

Scalar functions is one case where inverses are advantageous. Inverses and derivatives are related, because both help to improve efficiency when

dealing with non-linear equation systems. Inverses supply an explicit solution and thus avoid numerical inversion of an equation system. Symbolic derivatives improve the speed and robustness of the numerical solution. If the non-linear equation system is the right-hand side of an ODE, partial derivatives combined with the "dummy derivative" algorithm [Mattsson and Söderlind, 1993] make it possible to reduce the equation system, under the condition that the original unknowns in the equations are candidates to become dynamic states. This has been illustrated with the Example 1. An inverse with respect to some variables can be calculated symbolically if the partial derivatives in the Jacobian are available and the new input variables of the non-linear function are good candidates for state variables. The second condition is fulfilled more often than expected. The problems of inverses and partial derivatives are thus closely related in two ways:

- Access to partial derivatives improves the numerical reliability to solve for the inverse.
- In special cases partial derivatives and the dummy derivative algorithm avoid inverses by choosing different state variables.

A drawback of object-oriented modeling is that a large number of unnecessary variables and equations are created. The early work on development of Dymola [Elmqvist, 1978] and Omola [Andersson, 1994] has demonstrated the necessity of applying symbolic calculation to reduce the number of variables and to remove trivial equations. The development of ThermoFluid has clearly demonstrated the usefulness of transformations of variables and equations. It has also indicated that it may be very useful to introduce additional formal semantics for equations. Combining the two ideas indicate a research direction that would be very interesting, namely to develop a safe machinery for transformation of variables and equations. Such a machinery should contain the facilities for dealing with changes of variables, substitutions, computation of partial inverses and symbolic differentiation. A starting point could be the Lambda Calculus of Church, [Church, 1941], that influenced functional programming.

7.2 Model Debugging

The acausal model description by means of equations is one of the strong features of Modelica. Unfortunately this feature makes it quite difficult to find errors in models. A library like ThermoFluid which is designed to be open for user-written model extensions requires more debugging features than a library which is only used for graphical assembly. Clear

and comprehensive model documentation reduces the need for debugging, but does not eliminate it.

The problem with debugging equation systems has briefly been mentioned in Section 6.2. Recently [Bunus and Fritzson, 2002] presented results from a prototype implementation of a debugging environment for equation based, object-oriented modeling. The bipartite graph connecting equations and variables can according to [Dulmage and Mendelsohn, 1958] be decomposed into three parts: an over-constrained part, a well-constrained part and an under-constrained one. The over-constrained part has more equations than variables, the under-constrained one has more variables than equations. This decomposition is straightforward, but the information from the decomposition is often not sufficient to be helpful: there are too many possible choices for the additional equation or variable. The paper [Bunus and Fritzson, 2002] uses additional information about the model structure and knowledge of the Modelica semantics to narrow down the number of choices and annotates the equations with this information. One such annotation adds the attribute *connector-generated* to equations. When there is one extra equation, it is clearly infeasible to remove an equation which is generated from a Modelica `connect` statement that generated several equations in total. Debugging of models derived from extensible model libraries like ThermoFluid could benefit a lot from an extension to this idea: annotations to partial models providing extra debugging information.

Degrees of Freedom

The physically motivated structure of the ThermoFluid library makes it possible to draw a number of conclusions about equations which necessarily have to be part of a model. Such extra information would be very useful for debugging but currently it can not be expressed in Modelica. The additional information is a consequence of the chosen abstractions of control volumes and flow models, see Chapter 5. These abstractions do not define computational causality but they define which physical phenomena have to be part of a flow or control volume model. The difference between the total number of variables and the total number of equations can be defined as the "degree of freedom" of the model:

$$\text{degrees of freedom} = \#\text{ of variables} - \#\text{ of equations} \quad or$$
$$dof = nvar - neq$$

The degrees of freedom of control volume models in ThermoFluid depend on the number of flow and heat transfer connectors. Three assertions can be formulated based on the physical model structure:

- The degrees of freedom are calculated as:

$$2 \times \#\text{ of flow connectors} + \#\text{ of heat transfer connectors}$$

- None of the flow variables is assigned by an equation in the model.
- All states and fluid properties are assigned by an equation in the model.

It would easily be possible to add an annotation for the degrees of freedom of a model to Modelica. The possible mapping between variables and equations is usually not unique when $dof \neq 0$, but the above assertions must hold. These rules have to hold also in all cases where a class parameter is changed in the control volume. Such physically motivated annotations would make it possible to localize errors due to additional or missing equations.

Similar calculations of the local degrees of freedom due to the physical constraints can be made for all models in ThermoFluid. Using that extra information it would be possible to give precise diagnostics for errors in user-defined models.

Named Equations

In the current version of the Modelica language (2.0), the type system is completely independent of the equations in the model. The type of a model is entirely based on the hierarchical declaration of models using the five basic types: Real, Integer, Boolean, String and Enumeration. This provides a large amount of safety against misuse of class parameters, but prevents library designers from using equations for correctness checks. This can be compared to the situation in object-oriented programming where the methods which implement the behavior are declared in the class interface. Equations are a core entity in equation based DAE system modeling, but they are anonymous in Modelica. We illustrate this by an example.

EXAMPLE 2—THE VALVE EQUATION

As an illustrative case an example from the ThermoFluid library will be used that has caused users trouble several times. A wrong assumption about existence of equations in a base class causes errors which are very difficult to debug by non-expert users. A base class for valves contains two flow connectors a and b, variables internal to the valve which take care of always using the upstream variables for reversing flow and a few parameters. For the valve equation, a lot of variations exist for gases, liquids and different types of valves. From a modeling point of view it is clear that the valve equation has to provide a relation between the

pressures on both sides of the valve and the mass flow. One simple model is:

$$\dot{m} = C_v \sqrt{\rho_{in}(p_a - p_b)} \tag{7.1}$$

where C_v is a parameter. All of the variables can be declared in a base class and used in the type checking, but nothing is said about the requirement to have an equation that relates $\dot{m}, p_a$ and p_b. For constitutive equations, it is almost always possible to list the variables that have to appear. Connecting variables to an equation which should be declared in a subclass is possible only if the equations are labeled and hence not anonymous. A possibility would be to introduce *named equations* which provide a handle to add type checking for some aspects of the equation system as well. Named equations can be made part of the existing type system quite easily. A named equation would have to be declared in the declaration section to belong to the new built-in type *Equation* and be implemented with a reference to the name in the equation section.

```
declaredEquations
Equation valve(a.p,b.p,mdot,rho) "simple valve equation";
equation
valve:: mdot = Cv*sqrt(rho*(a.p-b.p);
```

□

Equations can also be declared in a partial model, where they do not need to be implemented. This makes it possible to provide restricted but safe partial models in libraries. With the above suggestion, there would be two types of equations: anonymous ones and named equations. A similar type-safety as with named equations can be achieved with local functions. The disadvantage of functions is their reduced flexibility in symbolic computations. The disadvantage of named equations is the added complexity of a new language feature. Named equations would bring three advantages:

- improved debugging,
- increased type-safety for partial models and
- more similarity with object-oriented programming.

Named equations have been discussed before in the Modelica design group. A definition of formal semantics for equations would require some label for equations, they can not be anonymous entities.

7.3 User Interface Issues

An important factor for the acceptance of an engineering tool is the "learning curve": how long time does it take to get acquainted with the tool and use most of the available functionality. With the Modelica language as a freely available standard, distinction has to be made between three factors determining the ease of use

- the language definition
- the quality of a particular implementation and
- fundamental properties of equation based modeling.

The Modelica language is still evolving and therefore it is natural that not all language features are fully supported in the existing simulation tools. However, intuitive graphical support makes a substantial difference in acceptance for users wanting to use advanced modeling features. There is also a chicken-and-egg problem with developers who want to write libraries using advanced language features and tool developers supporting those features in a graphical user environment.

When selecting class parameters for example, a drop-down menu for selecting between the choices "flue gas", "steam" and "nitrogen" is much easier for users than writing out the equivalent Modelica syntax at the right place:

```
redeclare model Medium = MediumModels.IdealGas.Nitrogen;
```

Mainstream acceptance for powerful concepts like class parameters and multiple inheritance requires support tools to make them easy to use. Good diagnostics are needed which lead to the cause of the problem when something goes wrong. Graphical modeling support for class parameters is just beginning to emerge in the Dymola tool. Unfortunately it was not available during the development of ThermoFluid. The development of ThermoFluid actually had a significant impact on the development of the concept of class parameters in Modelica. Some parts of the library do not use class parameters even though they would be useful. Without user-friendly means for redeclaring models, an implementation based on class parameters did not make sense in those cases. Therefore, many distinct classes exist in ThermoFluid which could easily be folded into one more general class with switches for the appropriate choice of class parameters. The large number of model variants in thermo-fluid modeling can be structured nicely with the help of class parameters, but for normal simulation users this flexibility comes at the price of too much complexity. User interfaces have to translate this complexity to intuitive graphical selection mechanisms. The Modelica language has everything in place,

but the tools have not yet reached a sufficient level of user-friendliness to confront simulation users with class parameters.

7.4 Partial Differential Equations

Some parts of the ThermoFluid library are based on partial differential equations. For these models it would be highly desirable to have direct support for partial differential equations in the modeling language. In ThermoFluid, the partial differential equations (PDE) were manually transformed into ordinary differential equations using the finite volume method. The main difficulty with PDE is the large variety of solution methods which reflects the widely different properties of the solution. Many PDE problems require solvers which are adapted to a particular problem class. In contrast, a large class of differential algebraic equations with discontinuities can be handled well by existing DAE-solvers. When the partial differential equations are an integral part of a system model, the other system parts may need solution techniques with requirements that are difficult to reconcile with the PDE solver. Coordination of numerical solvers where one of the solvers tries to control the integration error is a non-trivial, unsolved problem. The main challenge of PDEs in system simulation is not only the solution of isolated, homogeneous PDE problems as in traditional software, but the integrated solution of multi-domain PDE with actuation and control. Modeling and control of flow-induced vibrations is a typical and challenging multi-domain problem. Currently, such work requires three different modeling tools: a finite element based solver for the vibrations, a finite volume or finite differences solver for the fluid flow and a system simulation tool for actuation and control. Integration of such tools is not impossible but difficult. There are many open questions with respect to the numerical stability and the reliability of the interaction between the different solvers. However, with an evolutionary approach it should be possible to get practical solutions for many problems involving spatially distributed models in Modelica. Modelica extensions for partial differential equations are currently under development, see [Saldamli *et al.*, 2002].

PDE Solution Methods

A brief overview over solution methods for differential algebraic and partial differential equations has been given in Section 2.1. The main conclusion was that the most promising methods to integrate the solution of distributed and lumped models are method of lines approaches which apply a fixed spatial discretization to a PDE problem. The spatial discretization transforms the PDE problem into a DAE problem which can

be solved with standard DAE solvers for large, sparse problems. The spatial discretization can be based on one of the three common methods:

- Finite Difference Methods (FDM) find the solution on a grid of discrete points. Derivatives are approximated by differences between these points.
- The Finite Element Methods (FEM) are based on weighted residual methods applied to sub-domains, "elements", of the solution domain with certain continuity conditions at the element boundaries. Weighted Residual Methods assume that the solution to a PDE can be represented as a weighted combination of polynomial functions with unknown weighting factors.
- Finite Volume Methods (FVM) are based on the idea of integrating the dependent variables over a finite control volume and applying the conservation principle to the integrated variables.

Each of the methods is useful for certain types of problems. A large number of software packages is available for each of these solution methods. In the following, a closer look will be taken at those software packages that intend to model multi-domain PDE-problems and focus on system modeling aspects.

Software Packages for PDE

Among the existing special purpose languages for modeling, gPROMS is the only one that currently has language elements to describe PDE, see [Oh, 1995]. This is not so surprising, because PDE support is both essential for process engineering applications and relatively straightforward to implement because typical process equipment has simple geometries. The gPROMS implementation uses several variants of the method of lines with orthogonal collocation FEM and some choices of finite difference approximations as discretizations. The gPROMS language has a syntax that describes PDEs symbolically over simple geometries in a given coordinate system, e. g., in rectangular or cylindrical coordinates. When the model is used, a discretization method with the spatial discretization order as parameter, is applied to the partial differential equation and transforms it into a system of DAE. During simulation, the spatial discretization is fixed and the time discretization is controlled by the DAE solver. The discretization method and grid spacing are not regarded as part of the model, but instead as part of the simulation experiment. Experiences of using this method for chemical process engineering problems with mixed lumped and distributed parts has demonstrated that the method of lines works well for this type of problems.

It should be noted that gPROMS does not implement the finite volume method as a discretization method. The reason is probably that it is not trivial to formalize the concept of control volumes and a staggered grid, see Chapter 5. Extra model information which is not needed in the other methods has to be supplied. For example, which variables are calculated on which grid? Reversing flows is another feature of ThermoFluid that is not handled in the current PDE implementation of gPROMS. The boundary condition in a piece of equipment changes conditionally depending on the flow direction. Surprisingly, conditional boundary conditions have not been considered in gPROMS, even if conditional equations exist.

FEMLAB [Comsol AB, 1998] is a product compatible with Matlab for modeling of multi-domain PDE in a graphical user environment. However, FEMLAB does not describe the problem using a formal modeling language. One generalized PDE formulation is used in all applications and the user specifies coefficients and geometry. The general PDE covers many classical distributed problems. The Finite Element Method (FEM) is used as discretization method in all cases. FEMLAB is a powerful tool for those PDE problems which fall into the (rather large) class of problems that can be solved by the generalized PDE used by FEMLAB[4] and the FEM method. FEMLAB's strong side is the user interface for specifying the domain and boundary geometry graphically. Other finite element packages are usually less general than FEMLAB, but execute faster. FEMLAB is integrated into Matlab and Simulink and makes it possible to investigate control of PDE models.

Modelica is designed for multi-domain problems. Relevant problems involving partial differential equations in mechanical systems have complex geometries. The limitation to simple geometries as in gPROMS is not sufficient for mechanical systems. A Modelica PDE implementation would need a combination of the main features of gPROMS and FEMLAB:

- A formal language for the equations, boundary conditions and the solution domain geometry as well as
- the flexibility to define the problem geometry graphically.

7.5 Extensions to ThermoFluid

The scope of the ThermoFluid library has grown considerably during its development. Chemical reactions were not part of the original design goals and many of the uses of ThermoFluid were not considered from the beginning. Building application specific component libraries based on ThermoFluid base classes would be a valuable extension of the current effort.

[4] Compressible flow with heat transfer is not among the solvable problems

Some extensions would be of academical interest and could be done at universities, other ones could be commercial, domain specific libraries. System models are an integral part of engineering research. Interesting research is being done in areas where the use of ThermoFluid could speed up the development of dynamic models: fuel cells, combustion engines, refrigeration systems, integrated combined heat and power, reactive distillation columns and other areas. Other extensions are of little academical interest, but useful in industrial model development. Industrial requests for extensions to ThermoFluid have initiated the formation of a company by the authors of ThermoFluid, Scynamics HB. Scynamics HB developed an interface to a fluid property calculation package which extends the range of fluids for which ThermoFluid models are applicable by a large number. The ThermoFluid library itself is freely down-loadable from the Internet[5]. A broader industrial acceptance would need work on

- more documentation,
- a tutorial with many examples and
- quality assurance in the form of simulatable, well documented feature tests.

ThermoFluid has been a testbed for some of Modelica's language features, e. g., class parameters. The thermo-fluid domain covers many complex modeling phenomena and well structured code helps to handle this complexity. ThermoFluid could also in the future be used for this purpose. Parts of the current implementation of ThermoFluid would benefit from an update to new features in Modelica 2.0, for example the fluid property functions. Arrays of interconnected model components are defined in Modelica but have not yet been used in model libraries which demonstrate the usefulness, e. g., distillation column trays.

7.6 Summary

Physical modeling of systems is a complex endeavor. It is challenging even for experts and easily gets frustrating for novice users stumbling into one of the many possible pitfalls. While power users request more features on the "high end" and do not care so much for beginner-friendly diagnostics, new users profit much more from better diagnostics and improved handling of numerical pitfalls like chattering. Modelica is in a good position to gain a become a widespread modeling standard. In order to attain this goal, extensions to Modelica have to keep the balance between extensions for power users and improvements for new users.

[5]For example from the Modelica site at http://www.modelica.org/libraries.html

8

Conclusions

This thesis has described the experiences of the development of a library for thermo-fluid systems. The development of the ThermoFluid library and its application in a wide range of industrial and academic problems have stretched over several years and many researchers have been involved. The key contributions are: the library itself and some principles for constructing object-oriented libraries expressed as design patterns.

Detailed systems modeling requires substantial resources. ThermoFluid has been used successfully in many projects. These projects have demonstrated that object-oriented libraries can help to speed up and increase quality of modeling by providing reusable building blocks for mathematical models.

The ThermoFluid library is a new type of object-oriented model library that provides reuse not only at the level of engineering equipment but also at the level of physical phenomena. This fine-grained reusability makes it applicable to a broad range of modeling tasks. Building blocks representing physical phenomena make it simple to assemble specialized models and still benefit from well-tested library code. This combination of rapid model development and high flexibility is a unique feature of phenomena-based object-oriented libraries. Structuring of the building blocks is a crucial issue for achieving the combination of flexibility and safety. The structure has to reflect fundamental physical principles, e. g., the conservation of mass and energy and constitutive relations. The **replaceable** HeatAndMass -object is designed to accommodate any type of heat- and mass transfer. For equipment-oriented system decomposition, single inheritance provides all a modeler needs to handle code reuse. However, decomposition at the level of physical concepts benefits from multiple inheritance, because it simplifies the description of interaction between tightly coupled phenomena. Multiple inheritance should be used carefully, but it offers valuable flexibility if used in appropriate situations.

ThermoFluid can be regarded as a framework for thermo-fluid applica-

tions. It is flexible to apply to a wide variety of tasks, but some effort has to be devoted to adapt it to a particular application in order to make it as easy to use as black-box simulation programs. ThermoFluid has been used for models in process engineering, gas turbine systems, refrigeration systems and steam power plants.

ThermoFluid models and Modelica tools strike a good balance between black box simulation tools – easy to use but unflexible – and detailed C- and FORTRAN codes, which provide full flexibility at the cost of very long development times. Models written in Modelica have significant advantages over closed source models:

- The flexibility is much higher with open source models. This is particularly important for rapid model development in emerging technologies.
- Model verification and validation is much easier with full access to the model code.
- Equation based models are to a large degree self-documenting. The encoding of modeling knowledge is much closer to the original source of the knowledge, equations in text books, than general purpose programming code. Furthermore, closed source models often do not provide any source code to verify the model implementation and are rarely documented in all details.

During the development of the ThermoFluid library and its predecessors in OMOLA and SMILE, certain problems in the implementation and use of model libraries turned up repeatedly. Library users without extensive modeling experience were puzzled again and again by typical pitfalls. Library developers needed several iterations until their models worked as expected. This suggested the use of an idea from object-oriented programming, namely to collect experiences in the form of *design patterns*. Design patterns use a catch phrase to describe a problem and suggest a way how to deal with it. The fact that design patterns are useful is illustrated by the experience that 80 % of the support requests by users of the ThermoFluid library could be covered by two design patterns.

The tight feedback between the Modelica language design and the development of libraries, which use the newly designed language features has been mutually beneficial. The Modelica language and libraries continue to be actively developed. A continued close cooperation between users and developers will be beneficial for both sides. The open discussion between users and developers is one of the advantages of a language design process which is open to all interested parties.

Since the first definition of Modelica 1.0 in 1997, object-oriented modeling techniques are steadily gaining acceptance in industry. At the begin-

ning equation-based, object-oriented modeling was only used for technology assessment projects. Now the acceptance increases and the technique is used on a larger scale for product development and control design. The technical feasibility has been proven in many projects. The remaining obstacles for this technology to gain mainstream acceptance are economical reasons due to the enormous inertia of industry when introducing new technologies. More simulation environments using Modelica would certainly accelerate that process. Whether Modelica can reach its declared goal to become a de-facto or even formal standard for physical modeling remains to be seen.

9

References

Abadi, M. and L. Cardelli (1996): *A Theory of Objects*. Springer-Verlag, New York, Berlin.

Abelson, H., J. G. J. Sussman, and J. Sussman (1985): *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA.

Adams (2002): "http://www.adams.com."

Alhir, S. S. (1998): *UML in a Nutshell*. O'Reilly.

Andersson, M. (1994): *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis ISRN LUTFD2/TFRT–1043–SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Årzén, K.-E. (1994): "Grafcet for Intelligent Supervisory Control Applications." *Automatica*, **30:10**, pp. 1513–1526.

Årzén, K.-E. (1996): "Grafchart: A graphical language for sequential supervisory control applications." In *IFAC'96, Preprints 13th World Congress of IFAC*. San Francisco, California.

Åström, K. J. (2002): "Modeling of Complex Systems." In Gong and Shi, Eds., *Modeling, Control and Optimization of Complex Systems – In Honor of Professor Yu-Chi Ho*. Kluwer, Boston, MA. to appear.

Åström, K. J. and R. D. Bell (2000): "Drum boiler dynamics." *Automatica*, **36**, pp. 363–378.

Åström, K. J., H. Elmqvist, and S. E. Mattsson (1998): "Evolution of continuous-time modeling and simulation." In Zobel and Moeller, Eds., *Proceedings of the 12th European Simulation Multiconference, ESM'98*, pp. 9–18. Society for Computer Simulation International, Manchester, UK.

Åström, K. J. and B. Wittenmark (1990): *Computer Controlled Systems—Theory and Design*. Prentice-Hall, Englewood Cliffs, New Jersey.

Barton, P. and C. Pantelides (1994): "Modeling of combined discrete/continuous processes." *AIChE J.*, **40**, pp. 966–979.

Barton, P. I. (1992): *The Modelling and Simulation of Combined Discrete/-Continuous Processes*. PhD thesis, Imperial College of Science, Technology and Medicine, London.

Bauer, O. (1999): "Modelling of two-phase flows with Modelica." Technical Report Masters thesis ISRN LUTFD2/TFRT–5629–SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Bauer, O. and H. Tummescheit (2000): "Modeling of two-phase flows in modelica." In *Proceedings of the 3rd MATHMOD Conference, MATHMOD 2000*. Vienna.

Beater, P. (2002): "http://www.modelica.org/library/HyLib/docu/HyLib.html."

Beck, B. T. and G. L. Wedekind (1981): "A generalization of the system mean void fraction model for transient two-phase evaporation flows." *Int. J. of Heat Transfer*, **103**, pp. 81 – 85.

Bejan, A. (1997): *Advanced Engineering Thermodynamics*. John Wiley & Sons Inc., New York.

Biersack, M. (1994): "Entwurf und Implementierung einer Simulationssprache für dynamische Systeme.". Master's thesis, Fachbereich Informatik der TU Berlin.

Bischof, C., A. Carle, P. Khademi, A. Mauer, and P. Hovland (1995a): "Adifor 2.0 user's guide (revision c)." Technical Report. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

Bischof, C., L. Roh, and A. Mauer (1995b): "ADIC — An Extensible Automatic Differentiation Tool for ANSI-C." Technical Report. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

Bittanti, S., M. Bottinelli, A. D. Marco, M. Facchetti, and W. Prandoni (2001): "Performance Assessment of the Control System of Once-Through Boilers." In *Proceedings of the 13th Conference on Process Control '01*.

Bohlin, T. (1991): *Interactive System Identification: Prospects and Pitfalls*. Springer-Verlag, Berlin, Germany.

Bohlin, T. (1998): "Process Model Calibrator and Validator." In *Preprints of Reglermöte*, pp. 58–62. Lund Institute of Technology, Lund, Sweden.

Breunese, A. P. and J. F. Broenink (1997): "Modeling mechatronic systems using the SIDOPS+ language." In *Proceedings of ICBGM'97, 3rd International Conference on Bond Graph Modeling and Simulation*, Simulation Series, Vol.29, No.1, pp. 301–306. The Society for Computer Simulation International.

Bronstein, I. N. and K. A. Semendjajew (1989): *Taschenbuch der Mathematik*. Teubner Verlagsgesellschaft, Leipzig.

Brück, D., , H. Elmqvist, S. E. Mattsson, and H. Olsson (2002): "Dymola for Multi-Engineering Modeling and Simulation." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference*. Modelica Association, Oberpfaffenhofen.

Bunus, P. and P. Fritzson (2002): "Methods for Structural Analysis and Debugging of Modelica Code." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference 2002*, pp. 157–165. Modelica Association and DLR, Oberpfaffenhofen.

Buse, S. (2001): *Objektorientierte Modellierung und dynamische Simulation druckaufgeladenener Wirbelschicht-Dampferzugeranlagen*. PhD thesis, Technical University Hamburg Harburg, Hamburg.

Cellier, F. E. (1991): *Continuous System Modeling*. Springer-Verlag, Berlin Heidelberg New York.

Char, B. W., K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monogan, and S. M. Watt (1992): *Maple V Library Reference Manual*. Springer-Verlag, New York.

Church, A. (1941): *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ.

Comsol AB (1998): *FEMLAB Reference Manual*. Comsol AB.

Cordes, G. (1963): *Strömungstechnik der gasbeaufschlagten Axialturbine*. Springer-Verlag, Berlin.

Curtis, C. F. and J. O. Hirschfelder (1952): "Integration of Stiff Equations." *Proc. Nat. Acad. Sci.*, **38**, pp. 235–243.

David, R. and H. Alla (1992): *Petri Nets and Grafcet*. Prentice Hall, New York.

Doležal, R. (1957): *Hochdruck-Heissdampf*. Vulkan-Verlag, Essen, Germany.

Dulmage, A. L. and N. S. Mendelsohn (1958): "Coverings of Bipartite Graphs." *Canadian J. Math.*, **10**, pp. 517–534.

Eborn, J. (1998): "Modelling and simulation of thermal power plants." Technical Report Licentiate thesis ISRN LUTFD2/TFRT–3219–SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Eborn, J. (2001): *On Model Libraries for Thermo-hydraulic Applications*. PhD thesis ISRN LUTFD2/TFRT–1061–SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Eborn, J. and B. Nilsson (1996): "Simulation of a thermal power plant using an object-oriented model database." In *IFAC'96, Preprints 13th World Congress of IFAC*, vol. O, pp. 121–126. San Francisco, California.

Eborn, J., H. Tummescheit, and K. J. Åström (1999): "Physical system modeling with Modelica." In *14th World Congress of IFAC*, vol. N.

Elmqvist, H. (1978): *A Structured Model Language for Large Continuous Systems*. PhD thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Fabricius, S. M. O. and E. Badreddin (2002): "Modelica Library for Hybrid Simulation of Mass Flow in Process Plants." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference 2002*. Modelica Association and DLR, Oberpfaffenhofen.

Franke, R. (2002): "Formulation of Dynamic Optimization Problems Using Modelica." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference 2002*. Modelica Association and DLR, Oberpfaffenhofen.

Fritzson, P., J. Gunnarsson, and M. Jirstrand (2002): "MathModelica – an Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference*. Modelica Association, Oberpfaffenhofen.

Gašparović, N. and D. Stapersma (1973): *Berechnung der Kennfelder mehrstufiger axialer Turbomaschinen*, vol. 39 of *Forschung im Ingenieurwesen*. VDI-Verlag.

Gawthrop, P. and L. Smith (1995): *Metamodelling: Bond Graphs and Dynamic Systems*. Prentice Hall, New York.

Gensym (1992): *G2 Reference Manual, version 3.0*. Gensym Corp., Cambrigde, MA.

GómezP̃érez, A. A. (2001): "Modelling of a Gas Turbine with Modelica." Master's thesis ISRN LUTFD2/TFRT--5668--SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Gordon, S. and B. J. McBride (1994): "Computer Program for Calculation of Complex Chemical Equilibrium Compositions and Applications. Part 1: Analysis." NASA Technical Reports Document ID: 19950013764 N (95N20180), NASA-RP-1311 E-8017 NAS 1.61:1311. NASA.

Griewank, A. (1991): "The chain rule revisited in scientific computing." *SIAM News*, **24:3 & 4**, pp. 20 – 21 & 8 ff. Also appeared as Preprint MCS–P227–0491, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439–4801.

Griewank, A. and G. F. Corliss, Eds. (1991): *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA.

Griewank, A., D. Juedes, H. Mitev, J. Utke, O. Vogel, , and A. Walther (June 1996): "ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++." *ACM TOMS*, **22:2**, pp. 131–167.

Gustafsson, K. (1992): *Control of Error and Convergence in ODE Solvers*. PhD thesis ISRN LUTFD2/TFRT–1036–SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Guyon, E., J.-P. Hulin, and L. Petit (1997): *Hydrodynamik*. Verlag Vieweg, Braunschweig/Wiesbaden.

Hairer, E. and G. Wanner (1996): *Solving Ordinary Differential Equations II*, vol. 17 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin Heidelberg New-York.

Harel, D. (1987): "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, **8**, pp. 231–274.

Harlow, F. H. and J. E. Welch (1965): "Numerical Calculation of Time-dependent Viscous Incompressible Flow of Fluid with Free Surface." *Phys. Fluids*, **8**, pp. 2182 – 2189.

Haugwitz, S. (2002): "Modelling of Microturbine Systems.". Master's thesis ISRN LUTFD2/TFRT--5687--SE.

He, X. and S. Liu (1998): "Multivariable Control of Vapor Compression Systems." *HVAC Research*, **4**, pp. 205 – 230.

He, X., S. Liu, and H. Asada (1994): "A Moving-Interface Model of Two-Phase Flow Heat Exchanger Dynamics for Control of Vapor Compression Cycle." *ASME, Heat Pump and Refrigeration Systems Design, Analysis and Applications*, **32**, pp. 69 – 75.

He, X., S. Liu, and H. Asada (1995): "Modeling of Vapor Compression Cycles for Advanced Controls in HVAC Systems." In *Proceedings of the American Control Conference 1995*, vol. 5, pp. 3664 – 3668.

He, X.-D., S. Liu, and H. H. Asada (1997): "Modeling of Vapor Compression Cycles for Multivariable Feedback Control of HVAC Systems." *Journal of Dynamic Systems, Measurement, and Control*, **119**, pp. 183 – 191.

Herzke, K. (1983): *Beschreibung des instationären Verhaltens von Gasturbinen durch Simulationsmodelle*. PhD thesis, TU Hannover.

Hetsroni, G. (1982): *Handbook of Multiphase Systems*. Hemisphere Publishing Corporation.

Heusser, P. A. (1996): *Modelling and Simulation of Boiling Channels with a General Front Tracking Approach*. SCS.

Honeywell (2002): "http://www.htc.honeywell.com/dome (dome)."

ITI GmbH (2002): "http://www.iti.de."

Jeandel, A., F. Boudaud, P. Ravier, and A. Buhsing (1996): "U.L.M: Un Langage de Modélisation, a modelling language." In *Proceedings of the CESA'96 IMACS Multiconference*. IMACS, Lille, France.

Jensen, J. M. and H. Tummescheit (2002): "Moving Boundary Models for Dynamic Simulation of Two-Phase Flows." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference 2002*. Modelica Assoc. and DLR, Oberpfaffenhofen.

Jochum, P. and M. Kloas (1994): "The Dynamic Simulation Environment Smile." In Tsatsaronis, Ed., *Second Biennial European Conference on System Design & Analysis*, pp. 53–56. The American Society of Mechanical Engineers.

Johansson, K. H., M. Egerstedt, J. Lygeros, and S. Sastry (1999): "On the Regularization of Zeno Hybrid Automata." *System & Control Letters*, **38**, pp. 141–150.

Johnsson, C. (1999): *A Graphical Language for Batch Control*. PhD thesis ISRN LUTFD2/TFRT–1051–SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Johnsson, C. and K.-E. Årzén (1999): "Grafchart and Grafcet: A Comparison between Two Graphical Languages Aimed for Sequential Control Applications." In *Preprints 14th World Congress of IFAC*, vol. A, pp. 19–24. Beijing, P.R. China.

Juslin, K. (1995): "Experience on mechanistic modelling of industrial processes with apros." *Mathametics and Computers in Simulation*, **39**, pp. 505–511.

Kohavi, Z. (1978): *Swichting and Finite Automata Theory*. McGraw-Hill, New York.

Kolev, N. I. (1986): *Transiente Zweiphasen-Strömung*. Springer-Verlag, Berlin.

Levy, S. Y. and D. Abdollahian (1982): "Homogeneous non-equilibrium Critical Flow Model." *Int. J. Heat and Mass Transfer*, **25:6**, pp. 247–252.

Lin, C. C. and L. A. Segel (1988): *Mathematics Applied to Deterministic Problems in the Natural Sciences*. SIAM Classics in Applied Mathematics, New York.

Lindstrand, N. (2002): "Noll problem med simulering." *Svensk Papperstidning*, **1**, pp. 28–29.

Linnecken, H. (1957): "Die Mengendruckgleichung für eine Turbinen-Stufengruppe." *BWK*, **9:2**, pp. 53–56.

Logan, J. D. (1994): *An Introduction to Nonlinear Partial Differential Equations*. John Wiles & Sons, Inc., New York.

Malmborg, J. (1998): *Analysis and Design of Hybrid Control Systems*. PhD thesis ISRN LUTFD2/TFRT–1050–SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Massobrio, G. and P. Antognietti (1993): *Semiconductor Device Modeling with SPICE*. McGraw Hill.

MathWorks (2001a): *Matlab User's Guide*. The Math Works Inc., Cochituate Place, 24 Prime Park Way, Natick, MA 01760.

MathWorks (2001b): *SIMULINK, a Program for Simulating Dynamic Systems, User's Guide*. The Math Works Inc., Cochituate Place, 24 Prime Park Way, Natick, MA 01760.

Mattsson, S. E. (1996): "On object-oriented modeling of relays and sliding mode behaviour." In *IFAC'96, Preprints 13th World Congress of IFAC*, vol. F, pp. 259–264. San Francisco, California.

Mattsson, S. E. (1997): "On modeling of heat exchangers in Modelica." In Winfried and Lehmann, Eds., *Proceedings of the 1997 European Simulation Symposium (ESS'97)*, pp. 127–133. SCS, The Society for Computer Simulation International, Passau, Germany.

Mattsson, S. E., M. Andersson, and K. J. Åström (1993): "Modeling and simulation of behavioral systems." In *Proceedings of the 32nd IEEE Conference on Decision and Control*, vol. 4, pp. 3636–3641. San Antonio, Texas.

Mattsson, S. E., H. Olsson, and H. Elmqvist (2000): "Dynamic Selection of States in Dymola." In *Modelica 2000 Workshop Proceedings*, pp. 61–67. Modelica Association, Lund.

Mattsson, S. E. and G. Söderlind (1993): "Index reduction in differential-algebraic equations using dummy derivatives." *SIAM Journal of Scientific and Statistical Computing*, **14:3**, pp. 677–692.

McLinden, M. O., S. A. Klein, E. W. Lemmon, and A. P. Peskin (1998): *NIST Thermodynamic and Transport Properties of Refrigerants and Refrigerant Mixtures—REFPROP*. U. S. Department of Commerce, 6th edition.

Minsky, M. (1965): "Models, Minds, Machines." In *Proceedings, IFIP Congress*, pp. 45–49.

Modelica Association (2000a): "Modelica Language Specification, Version 1.4." http://www.Modelica.org/documents.shtml.

Modelica Association (2000b): "Modelica Language Tutorial, Version 1.4." http://www.Modelica.org/documents.shtml.

Modelica Association (2002a): "http://www.Modelica.org."

Modelica Association (2002b): "Modelica Language Specification, Version 2.0." http://www.Modelica.org/documents.shtml.

Mostermann, P. J. and H. Vangheluwe (2000): "Computer Automated Multi-Paradigm Modeling in Control System Design." In *Proceedings of the IEE International Sympsium on Computer Aided Control System Design*, pp. 65–70.

Mühlthaler, G. (2000): *Anwendung objektorientierter Simulationssprachen zur Modellierung von Kraftwerkskomponenten*. PhD thesis, Technische Universität Hamburg Harburg.

Murata, T. (1989): "Petri Nets: Properties Analysis and Application." *Proceedings of the IEEE*, **77:4**, pp. 541–580.

Nilsson, B. (1993): *Object-Oriented Modeling of Chemical Processes*. PhD thesis ISRN LUTFD2/TFRT–1041–SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Nilsson, B. and J. Eborn (1994): "K2 model database - tutorial and reference manual." Technical Report TFRT-7528. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Oh, M. (1995): *Modelling and Simulation of Combined Lumped and Distributed Processes*. PhD thesis, Imperial College of Science, Technology and Medicine.

Öhman, M. (1998): "Trajectory-based model reduction of nonlinear systems." Technical Report Licentiate thesis ISRN LUTFD2/TFRT–3223–SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Pantelides, C. (1988): "The Consistent Initialization of Differential-Algebraic Systems." *SIAM Journal of Scientific and Statistical Computing*, **9**, pp. 213–231.

Pantelides, C. C. (2000): "The Mathematical Modelling of the Dynamic Behaviour of Process Systems." Centre for Process Systems Engineering, Imperial College of Science, Technology and Medicine.

Patankar, S. V. (1980): *Numerical Heat Transfer and Fluid Flow*. Hemisphere Publishing Corporation, Taylor & Francis Group, New York.

Petzold, L. (1982): "A Description of DASSL: a Differential-Algebraic Equation Solver." In *Proceedings of IMACS World Congress*. Montreal, Canada.

Pfafferot, T. and G. Schmitz (2002): "Modeling and Simulation of Refrigeration Systems with the Natural Refrigerant Carbon Dioxid." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference 2002*. Modelica Association and DLR, Oberpfaffenhofen.

Pfafferott, T. and G. Schmitz (2001): "Numerische Simulation von CO2-Kühlprozessen mit Modelica." In *DKV-Tagungsbericht 2001*, vol. IV 28. Jahrgang. DKV, Stuttgart.

Pfleiderer, C. and H. Petermann (1991): *Strömungsmaschinen, 6. Auflage*. Springer-Verlag, Berlin.

Poling, B. E., J. M. Prausnitz, and J. P. O'Connell (2001): *The Properties of Gases and Liquids*, fifth edition. Mc Graw Hill, Boston, Massachusetts.

Popper, K. R. (1935): *Logik der Forschung*. Julius Springer Verlag, Vienna, Austria.

Preisig, H. (2001): “Modeling of Process Systems.” Lecture Notes, Systems and Control Group, TU Eindhoven.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling (1986): *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, New York.

Profos, P. (1962): *Die Regelung von Dampfanlagen*. Springer-Verlag, Berlin.

PSEnterprise (2002): “http://www.psenterprise.uk (gEST).”

Reid, R. C., J. M. Prausnitz, and B. E. Poling (1987): *The Properties of Gases and Liquids*. Mc Graw Hill, Boston, Massachusetts.

Sahlin, P., A. Bring, and E.F.Sowell (1996): “The Neutral Model Format for Building Simulation, Version 3.02.” Technical Report. Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden.

Saldamli, L., P. Fritzson, and B. Bachmann (2002): “Extending Modelica for Partial Differential Equations.” In Otter, Ed., *Proceedings of the 2nd International Modelica Conference 2002*, pp. 157–165. Modelica Association and DLR, Oberpfaffenhofen.

Sørlie, J. and J. Eborn (1997): “A grey-box identification case study: The åström–Bell drum-boiler model.” Technical Report ISRN LUTFD2/TFRT–7563–SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Sørlie, J. and J. Eborn (1998): “Parameter optimization results for a family of thermo-physical drum boiler models.” In *Preprints of Reglermöte ’98*, pp. 131–136. Lund Institute of Technology, Sweden.

Span, R. (2000): *Multi-Parameter Equations of State*. Springer-Verlag, Berlin.

Span, R. and W. Wagner (1996): “A New Equation of State for Carbon Dioxide from the Triple-Point Temperature to to 1100 K at Pressures of up to 800 MPa.” *Journal of Physical and Chemical Reference Data*, **25:6**, pp. 1509–1596.

Summerville, I. (2000): *Software Engineering*. Addison Wesley Publishing Company, Reading, Massachusetts.

Telnes, K. (1992): *Computer Aided Modeling of Dynamic Processes based on Elementary Physics*. PhD thesis 47, the Norwegian Institute of Technology.

Thumm, C. T. (1989): *Wirkleistungs-Sekundenreserve-Maßnahmen, untersucht am Beispiel eines Dampfkraftwerksblockes*. PhD thesis, Universität Stuttgart.

Tiller, M., C. Davis, H. Tummescheit, and N. Trigui (2000): "Powertrain modeling with modelica." In *Prooceedings of IMECE2000, 2000 ASME International Mechanical Engineering Congress and Exposition*.

Tiller, M. M. (2001): *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers.

Tillner-Roth, R. and H. D. Baehr (1994): "An international standard formulation of the thermodynamic properties of 1,1,1,2-tetrafluoroethane (HFC-134a) covering temperatures from 170 K to 455 K at pressures up to 70 MPa." *J. Phys. Chem. Ref. Data*, **23**, pp. 657–729.

Tolsma, J. and P. Barton (1999): "DAEPACK: An Open Modeling Environment for Legacy Models." *Ind. Eng. Chem. Res.*, September.

Tolsma, J. and P. Barton (2002): "Hidden Discontinuities and Parametetric Sensitivity Analysis." *SIAM Journal on Scientific Computing*, **23:6**, pp. 1862–1875.

Traupel, W. (1977): *Thermische Turbomaschinen*, third edition. Springer-Verlag, Berlin, Germany.

Tummescheit, H. (2000a): "Object-oriented Modeling of Physical Systems, Part 11." *Automatisierungstechnik*, **48:2**. In german.

Tummescheit, H. (2000b): "Object-oriented Modeling of Physical Systems, Part 12." *Automatisierungstechnik*, **48:4**. In german.

Tummescheit, H. and J. Eborn (1998): "Design of a thermo-hydraulic model library in Modelica." In Zobel and Moeller, Eds., *Proc. of the 12th European Simulation Multiconference, ESM'98*, pp. 132–136. Manchester, UK.

Tummescheit, H. and J. Eborn (2002): "Flexible Handling of Reactions and Diffusion in ThermoFluid." In Otter, Ed., *Proceedings of the 2nd International Modelica Conference 2002*. Modelica Association and DLR, Oberpfaffenhofen.

Tummescheit, H., J. Eborn, and F. Wagner (2000): "Development of a Modelica base library for modeling of thermo-hydraulic systems." In *Modelica 2000 Workshop Proceedings*, pp. 41–51. Lund.

Tummescheit, H., M. Klose, and T. Ernst (1997): "Modelica and Smile – a Case Study Applying Object-Oriented Concepts to Multi-Facet Modeling." In Hahn and Lehmann, Eds., *Simulation in Industry – Proceedings of the 9th European Simulation Symposium ESS97*, pp. 122–126. Society for Computer Simulation International, Budapest, Hungary.

Tummescheit, H. and R. Pitz-Paal (1997): "Simulation of a Solar Thermal Central Receiver Power Plant." In Sydow, Ed., *Proceedings of the 15th IMACS world congress on Scientific Computation, Modelling and Applied Mathematics*, vol. 6, pp. 671–676. Wissenschaft und Technik Verlag, Berlin, Germany.

Tummescheit, H. and M. Tiller (2000): "Object-oriented modeling of physical systems, part 17." *Automatisierungstechnik*, **48:12**.

Turns, S. R. (1993): *An Introduction to Combustion*. McGraw Hill International Editions, New York.

Versteeg, H. K. and W. Malalasekera (1995): *An Introduction to Computational Fluid Dynamics*. Addison Wesley Longman Limited.

Viklund, L. and P. Fritzson (1995): "ObjectMath •– An object-oriented language and environment for symbolic and numerical processing in scientific computing." *Scientific Programming*, **4**, pp. 229–250.

Wagner, F. J. (2000): *Object-Oriented Modeling of Energy Systems*. PhD thesis, Technical University of Denmark, Lyngby, Denmark.

Wagner, W. and A. Kruse (1998): *Properties of water and steam*. Springer-Verlag, Berlin.

Wang, H. (1991): *Modelling of a Refrigeration System together with a Refrigerated Room*. PhD thesis, Delft University of Technology.

Weiss, M. and H. A. Preisig (2000): "Structural Analysis in the Dynamical Modelling of Chemical Engineering Systems." *Mathematical and Computer Modelling of Dynamical Systems*, **6:4**, pp. 325–364.

Westerweele, M. R. and H. A. Preisig (2001): "Minimal Representation of First Principle Models." In *Proceedings of the 11th European Symposium of Computer Aided Process Engineering*. CAPEC.

Whalley, P. (1987): *Boiling, Condensation and Gas-Liquid Flow*. Clarendon, Oxford.

Wolfram, S. (1990): *Mathematica: A System for Doing Mathematics by Computer*, 2nd edition. Addison-Wesley, Reading, Mass.

Zhang, J., K. H. Johansson, J. Lygeros, and S. Sastry (2000): *Hybrid Systems: Computation and Control*, vol. 1790, chapter Dynamical Systems revisited: Hybrid Systems with Zeno Executions. Springer-Verlag, Berlin.

Zivi, S. M. (1964): "Estimation of Steady-State Steam Void Fraction by means of the Principle of Minimum Entropy Production." *ASME Journal of Heat Transfer*, **86**, pp. 759–770.

A

Glossary

This glossary is for readers with a background in physical modeling, but not in object-oriented programming. Object-oriented modeling is not the same as object-oriented programming, as has been discussed in Chapter 6. Where appropriate, the differences are highlighted. Simple, informal explanations for terms from object-oriented programming and some Modelica-specific definitions are given below. Terms defined in other entries in the glossary are typeset in *italics*.

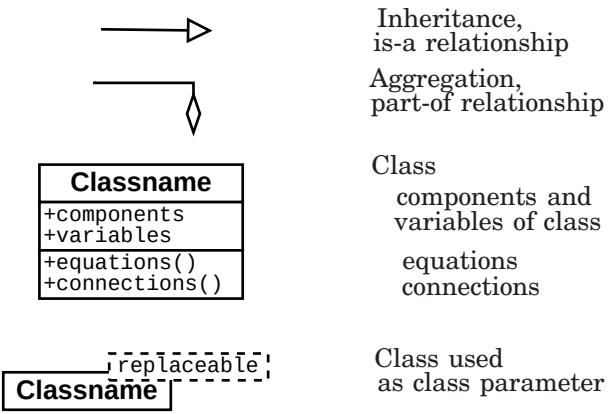


Figure A.1 Graphical notation for object-oriented model structuring, adapted from the UML (Universal Modeling Language) notation, see [Alhir, 1998].

abstract class An abstract class is a *class* that can not be *instantiated* because it is incomplete. In Modelica this means that there are more variables than equations or that a *component* is an *instance* of an abstract class that has to be *redeclared*.

aggregation According to Webster's Dictionary, aggregation is defined as a group or mass of distinct or varied things. In object-oriented programming it is the technical term for putting together more complex entities from simpler ones. This can be done by declaring *instances* of other *classes* (models) in a model. Instead of *aggregation*, the term **has-a** relation is often used: A car has-a motor. In many cases, a similar functional decomposition can be achieved with *multiple inheritance*.

algorithm In Modelica, an algorithm is used to describe behavior by assignment statements. Algorithms may contain loops and while statements and allow multiple assignments to the same variable. *Equations*, in contrast, allow only a single assignment to a variable.

annotation Annotations in Modelica are a part of the language with a formal grammar that has no influence on the model semantics. Annotations are mostly used for graphical information, but also for pragmas. Some Modelica annotations for the graphical layout of *component* declarations are standardized in order to keep graphical system information consistent even when different tools are used.

base class A base class is a *class* that is a generalization of a set of classes. Base classes are often *abstract classes*, which means that they can not be *instantiated*. Derived classes or child classes inherit from base classes.

built-in type Modelica has five built-in data types: Real, Integer, Boolean, String and Enumeration.

class A class is the programming abstraction for real-world objects with similar properties. This coincides well with the common notion of a model. In Modelica, there are several synonyms for class: model, type, record, block, connector, package and function. Some of the synonyms are restricted classes and may not contain all declarations which are allowed in general classes.

class parameter A class parameter is a high-level *parameter* that makes the *type* of class replaceable by other, *type-compatible classes*. See also *parameter*.

child class or short child. A class which inherits from another class is a child with respect to the class it inherits from.

component A component is an *instance* of a model inside another model. Components are used to *aggregate* complex systems from simpler parts. A component in object-oriented modeling is analogous to an object in object-oriented programming.

declarative language A language that describes *what-is* as opposed to *how-to-compute* something. This is a very useful abstraction for modeling knowledge. For equation based modeling it means that the equations do not have to be in the same sequence as computation requires and the computed variable does not have to be on the left side of an assignment statement.

derived class A derived class *inherits* from a *base class* and adds variables or behavior to it, see child class.

design pattern A design pattern motivates and explains a general solution for a recurring design problem in object-oriented modeling. It describes the problem, the solution, trade-offs and when the solution should be applied. A design pattern has to be customized and implemented in order to solve the problem in a particular context.

dot-notation Modelica offers the possibility to access public components inside a model using an access-operator, a dot. For Example, `pump.p` refers to the variable `p` inside the component `pump`.

encapsulation Encapsulation is a central technique to keep complex software systems maintainable. Details of a model or class should be hidden and unaccessible from the outside. Access to internal variables of a class should only be possible via well-specified interfaces. In Modelica, *protected* variables can not be accessed from the outside.

equation Modelica code describes model behavior with equations. This is different from object-oriented programming languages, where methods or functions are used to program how an object behaves.

inheritance Inheritance is the technique for reusing model or programming code from a more general *class*. Semantically, it is identical to including the inherited code at the place where the inheritance declaration is found in the code. The inheritance relation is also called **is-a** relation: a Volkswagen is-a car. The simplest way of understanding inheritance is to view it as *inclusion* of all declarations and equations of the base class. The Modelica keyword for inheritance is **`extends`**. In object-oriented programming, methods can be overridden by subclasses. Equations can not be overridden, but **`replaceable`** declarations can be **`redeclared`**.

instance An instance is an object created from a class. When a *component* is declared to be of a certain *class*, an instance is created. The declaration `Resistor r1(R=1000)` declares the component `r1` as an instance of the class Resistor and changes the parameter value of R to 1000.

instance parameter An instance parameter is a *replaceable component* in a model.

instantiate The process of finding the declarations of all *components* of a model and creating a simulatable *instance*.

mixin The term mixin is used in object-oriented programming to denote implementation parts which are not always needed. A mixin class adds additional features to a class that is functional even without the mixin.

model Model is one of the Modelica synonyms for class.

modification In Modelica, the values of *parameters*, *instance* parameters and *class parameters* can be changed in the declaration of a component. Such changes to an *instance* are called modifications. For example, in the declaration `Volume v1(V=1.5e-6, redeclare model Medium=hydrogen)`, the expressions in parentheses are modifications to the Volume class.

multiple inheritance When *classes* are allowed to inherit declarations from more than one *base class*, this is called multiple inheritance.

object An object is an *instance* of a *class*. In Modelica objects are mostly called *components*.

parameter Parameter has a very broad meaning in engineering. In Modelica there are three usages of the term parameter. All of them refer to properties of a model that can be changed at *instantiation* time or before simulations, but are constant during simulation. The ordinary parameter is a prefix to a built-in type or class, meaning that the values of the variable(s) in that type or class are constant during simulation. Instance parameters denote that the class of a component is *replaceable* in a *modification*. *Class parameters* are similar to instance parameters, but they refer to the type instead. With a class parameter, the class of several components can be changed at once or the type of the *base class* can be redeclared.

package A package is a collection of Modelica models. Usually the models are meant to be used together.

partial model A partial model is an incomplete definition of a model. This is the Modelica equivalent to an *abstract class* in object-oriented programming.

parent class or short parent. Parent class is a synonym for *base class*. The class from which other classes *inherit*.

polymorphism In object-oriented programming, polymorphism means the ability to substitute *objects* with matching interfaces for one another at run-time. The meaning of polymorphism in object oriented modeling is slightly more static, if simulation time is seen as the equivalent of run-time. The type substitution for objects has to take place at instantiation time in Modelica.

prefix A Modelica keyword which is put in front of a variable or class declaration and alters its semantics. A common prefix is parameter. When e. g., a record is declared with the prefix parameter, all elements of the record have to be constant during simulation.

protected Access to protected variables from the outside is forbidden.

public Public variables can be accessed from the outside via *dot-notation*.

redeclaration Declaring the type of an *instance parameter* or class parameter in a modification is a redeclaration. In the component declaration `Volume v1(V=1.5e-6, redeclare Medium = hydrogen)`, the second expression redeclares the *class parameter* Medium to be of type hydrogen in the component `v1`.

replaceable In Modelica, both *components* and *classes* can be declared to be replaceable. This means that the class of a component or the placeholder class can be exchanged against another, compatible class, see *type compatibility*.

pragma Pragmas are hints to compilers to make them aware of possible optimizations. Some Modelica annotations can be regarded as pragmas. Certain built-in Modelica operators can also be regarded as pragmas, e. g., `smooth(x,2)` is a pragma to numerical routines which tells them that the expression x is two times continuously differentiable.

specialization A specialization is often used as a synonym to *inheritance*. A class that inherits another classes behavior and adds components or equations to the parent class is more specialized and less general than the parent.

type Type is one of the Modelica synonyms to class. It is usually used for variants and arrays of one of the built-in types, e. g., `type Force = Real[3](Unit="N")`. Type is often used instead of *class* when referring to the set of properties and classification aspects of a class.

type compatibility Type compatibility is a compatibility condition on models which makes them exchangeable in a *redeclare* statement. A type compatible model has to contain at least the same *components* as the model it is compared to.

B

Thermodynamic Derivatives

Abstract

This appendix has been extended to cover multi-component mixtures and several fundamental equations from a similar form which has been published before as an Appendix to the Master's thesis [Bauer, 1999]. It describes the background knowledge and basic procedures for calculating thermodynamic derivatives from fundamental equations of state. The methods for rearranging thermodynamic derivatives have been implemented in a **Maple**-package, see [Char *et al.*, 1992]. This appendix documents the implementation of fluid property calculations in the ThermoFluid library.

B.1 Fundamental Equations

A short overview over fundamental equations of intensive thermodynamic fluid properties is given. The equations derived here will be used later to derive formulas for transforming derivatives of fundamental equations into familiar thermodynamic properties.

Basic Form

Comparison of the first law of thermodynamics in differential form

$$du = Tds - pdv \tag{B.1}$$

with the total differential of the function $u = u(s, v)$

$$du = \left.\frac{\partial u}{\partial s}\right|_v ds + \left.\frac{\partial u}{\partial v}\right|_s dv \tag{B.2}$$

shows

$$T = \left.\frac{\partial u}{\partial s}\right|_v \qquad p = -\left.\frac{\partial u}{\partial v}\right|_s \tag{B.3}$$

Therefore, an equation of state $u = u(s, v)$ does not only serve to compute the internal energy, but also yields temperature end pressure when being differentiated symbolically. The enthalpy is then obtained from $h = u + pv$. An equation of state, that contains the complete information for calculating all pure component thermodynamic variables from two input variables is called *fundamental* equation.

Secondary Forms

The independent variables in $u(s, v)$ may be changed and still the thermodynamic state is completely determined, but the related equation may not contain sufficient information to allow computation of all properties by differentiation. Loss of information is avoided when applying the Legendre transformation [Bejan, 1997]. The Legendre transformation serves

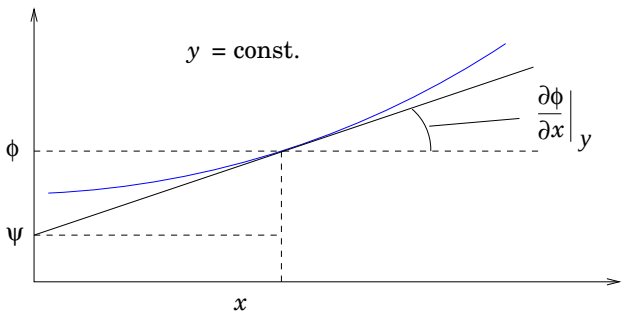


Figure B.1 Legendre transformation

to create secondary forms of a fundamental equation $\phi(x, y)$. In its simplest form, one independent variable is replaced by the partial derivative with respect to that variable.

$$\phi(x, y) \rightarrow \psi \left(\left. \frac{\partial \psi}{\partial x} \right|_y , y \right) \tag{B.4}$$

The derivative represents a tangent on a curve $y = \text{const}$ in a $\phi(x)$ diagram, see Figure B.1. To determine that tangent, its crossing point with the ordinate is required, which gives the definition of the new function ψ. This definition makes sure that the new function ψ contains the same information as the original function ϕ. This transformation is possible as long as $\partial\psi/\partial x \neq 0$, in other words $\psi(x, y = const)$ is strictly increasing or decreasing $\forall y$.

$$\phi(x, y) \rightarrow \psi = \phi - \left. \frac{\partial \phi}{\partial x} \right|_y x \tag{B.5}$$

Free Energy The function obtained from replacing the entropy in $u(s,v)$ with the related derivative is called the free energy f. Together with (B.3) the Legendre transformation yields

$$u(s,v) \to f\left(\left.\frac{\partial u}{\partial s}\right|_v, v\right) = f(T,v) \tag{B.6}$$

And the definition of f is found to be

$$u(s,v) \to f = u - \left.\frac{\partial u}{\partial s}\right|_v s = u - Ts \tag{B.7}$$

Differentiation of the last expression gives, after replacing du with (B.1),

$$df = du - Tds - sdT = -pdv - sdT \tag{B.8}$$

Comparison with the total differential of $f(T,v)$, similar to (B.2), shows

$$p = -\left.\frac{\partial f}{\partial v}\right|_T \qquad\qquad s = -\left.\frac{\partial f}{\partial T}\right|_v \tag{B.9}$$

Internal energy and enthalpy are obtained from $u = f + Ts$ and $h = u + pv$. Often v is replaced by $\rho = 1/v$ to give a fundamental function $f(T,\rho)$. In that case the pressure is obtained from

$$p = -\frac{d\rho}{dv}\left.\frac{\partial f}{\partial \rho}\right|_T = \rho^2 \left.\frac{\partial f}{\partial \rho}\right|_T \tag{B.10}$$

Free Enthalpy The fundamental equation for the free enthalpy or Gibbs' function g is obtained from replacing the specific volume in $f(T,v)$ with the related derivative

$$f(T,v) \to g\left(T, \left.\frac{\partial f}{\partial v}\right|_T\right) = g(T,p) \tag{B.11}$$

The definition of g is found to be

$$f(T,v) \to g = f - \left.\frac{\partial f}{\partial v}\right|_T v = f + pv \tag{B.12}$$

Differentiation of the last expression yields with df from (B.8)

$$dg = df + pdv + vdp = -sdT + vdp \tag{B.13}$$

Comparison with the total differential of $g(T,p)$ shows

$$s = -\left.\frac{\partial g}{\partial T}\right|_p \qquad\qquad v = \left.\frac{\partial g}{\partial p}\right|_T \tag{B.14}$$

Enthalpy and internal energy are obtained from and $h = g + Ts$ and $u = h - pv$

Enthalpy If the temperature in $g(T,p)$ is replaced by the derivative of g with respect to T we obtain the fundamental equation for the enthalpy h

$$g(T,p) \to h\left(\left.\frac{\partial g}{\partial T}\right|_p, p\right) = h(s,p) \tag{B.15}$$

defined by

$$g(T,p) \to h = g - \left.\frac{\partial g}{\partial T}\right|_p T = g + sT \tag{B.16}$$

The last term is differentiated and rearranged by use of (B.13)

$$dh = dg + sdT + Tds = vdp + Tds \tag{B.17}$$

Comparison with the total differential of $h(s,p)$ shows

$$v = \left.\frac{\partial h}{\partial p}\right|_s \qquad\qquad T = \left.\frac{\partial h}{\partial s}\right|_p \tag{B.18}$$

The internal energy is obtained from $u = h - pv$. Replacing the pressure in $h(s,p)$ with $\partial h/\partial p|_s$ yields $u(s,v)$, which brings us back to the starting point of (B.1).

From the four fundamental equations, only $f(T,v)$ and $g(T,p)$ are used for multi-parameter equations of state. The Helmholtz equation is often used with the density as $f'(T,\rho = 1/v)$.

B.2 Transformation of Partial Derivatives

From a fundamental equation $\phi(x,y)$, derivatives of thermodynamic functions with respect to x and y can easily be obtained from symbolic differentiation, but a derivative

$$\left.\frac{\partial a}{\partial b}\right|_c \tag{B.19}$$

where b and/or c differs from x and/or y appears to require numerical approaches. As will be shown in this section, it can be reduced to derivatives with respect to x and y.

The tool for rewriting partial derivatives are some simple relations and *functional determinants* or Jacobians. The simpler rules given later in this section are derived as special cases from the Jacobians. The examples are all for the case of single component systems, but the Jacobians below show that they hold in the same way if an arbitrary additional number

of variables is hold constant instead of just one, so that the examples generalize to multi-component systems in an obvious way.

$$\frac{\partial(x, y, ..., z)}{\partial(\alpha, \beta, ..., \gamma)} = \begin{vmatrix} \frac{\partial x}{\partial \alpha} & \frac{\partial x}{\partial \beta} & \cdots & \frac{\partial x}{\partial \gamma} \\ \frac{\partial y}{\partial \alpha} & \frac{\partial y}{\partial \beta} & \cdots & \frac{\partial y}{\partial \gamma} \\ \vdots & & \ddots & \vdots \\ \frac{\partial z}{\partial \alpha} & \frac{\partial z}{\partial \beta} & \cdots & \frac{\partial z}{\partial \gamma} \end{vmatrix}$$

$$\frac{\partial(x, y, ..., z)}{\partial(\alpha, \beta, ..., \gamma)} = -\frac{\partial(y, x, ..., z)}{\partial(\alpha, \beta, ..., \gamma)} \qquad \text{(exchanging positions of x and y)}$$

$$\frac{\partial(x, y, ..., z)}{\partial(\alpha, \beta, ..., \gamma)} = \left[\frac{\partial(\alpha, \beta, ..., \gamma)}{\partial(x, y, ..., z)}\right]^{-1} \qquad \text{(reciprocal relation)}$$

$$\frac{\partial(x, y, ..., z)}{\partial(\alpha, \beta, ..., \gamma)} = \frac{\partial(x, y, ..., z)}{\partial(A, B, ...C)} \frac{\partial(A, B, ..., C)}{\partial(\alpha, \beta, ..., \gamma)} \qquad \text{(chain rule)}$$

From the calculation of determinants with

$$\left.\frac{\partial x}{\partial x}\right|_\alpha = \left.\frac{\partial \alpha}{\partial \alpha}\right|_x = 1 \qquad \left.\frac{\partial x}{\partial \alpha}\right|_x = \left.\frac{\partial \alpha}{\partial x}\right|_\alpha = 0$$

it follows that

$$\left.\frac{\partial x}{\partial \alpha}\right|_{\beta,....,\gamma} = \frac{\partial(x, \beta, ..., \gamma)}{\partial(\alpha, \beta, ..., \gamma)}$$

First Derivatives

Let a and another variable β be functions of b and c.

$$a = a(b, c)$$
$$\beta = \beta(b, c)$$

The related Jacobian matrix contains the partial derivatives of a and β

$$\mathsf{J} = \begin{pmatrix} \left.\frac{\partial a}{\partial b}\right|_c & \left.\frac{\partial a}{\partial c}\right|_b \\ \left.\frac{\partial \beta}{\partial b}\right|_c & \left.\frac{\partial \beta}{\partial c}\right|_b \end{pmatrix}$$

Its determinant is computed as follows

$$\det \mathsf{J} = \frac{\partial(a, \beta)}{\partial(b, c)} = \left.\frac{\partial a}{\partial b}\right|_c \left.\frac{\partial \beta}{\partial c}\right|_b - \left.\frac{\partial a}{\partial c}\right|_b \left.\frac{\partial \beta}{\partial b}\right|_c$$

In the special case β=c we obtain, since $\partial c/\partial c|_b = 1$ and $\partial c/\partial b|_c = 0$,

$$\left.\frac{\partial a}{\partial b}\right|_c = \frac{\partial(a,c)}{\partial(b,c)}$$

which is the derivative to be determined. The right side is expanded by application of the multiplication theorem for functional determinants [Bronstein and Semendjajew, 1989]

$$\left.\frac{\partial a}{\partial b}\right|_c = \frac{\partial(a,c)}{\partial(x,y)}\frac{\partial(x,y)}{\partial(b,c)} = \frac{\partial(a,c)/\partial(x,y)}{\partial(b,c)/\partial(x,y)}$$

which is equivalent to

$$\left.\frac{\partial a}{\partial b}\right|_c = \frac{\partial a/\partial x|_y\,\partial c/\partial y|_x - \partial a/\partial y|_x\,\partial c/\partial x|_y}{\partial b/\partial x|_y\,\partial c/\partial y|_x - \partial b/\partial y|_x\,\partial c/\partial x|_y}$$

This equation allows the derivative on the left side to be written in terms of derivatives with respect to x and y. Further simplification is possible, if a,b and/or c agrees with x and/or y, because

$$\left.\frac{\partial x}{\partial x}\right|_y = \left.\frac{\partial y}{\partial y}\right|_x = 1 \qquad \left.\frac{\partial x}{\partial y}\right|_x = \left.\frac{\partial y}{\partial x}\right|_y = 0$$

which gives

$$\left.\frac{\partial a}{\partial b}\right|_x = \frac{\partial a/\partial y|_x}{\partial b/\partial y|_x} \tag{B.20}$$

$$\left.\frac{\partial x}{\partial y}\right|_a = -\frac{\partial a/\partial y|_x}{\partial a/\partial x|_y} \tag{B.21}$$

$$\left.\frac{\partial a}{\partial x}\right|_b = \left.\frac{\partial a}{\partial x}\right|_y - \left.\frac{\partial b}{\partial x}\right|_y \frac{\partial a/\partial y|_x}{\partial b/\partial y|_x} \tag{B.22}$$

EXAMPLE 1—VELOCITY OF SOUND

As an example a relation for the velocity of sound will be derived: The velocity of sound a is

$$a = \sqrt{\left.\frac{\partial p}{\partial \rho}\right|_s} \tag{B.23}$$

therefore

$$\frac{1}{a^2} = \left.\frac{\partial \rho}{\partial p}\right|_s \tag{B.24}$$

Application of (B.22) to the right side yields with $(x, y) = (p, h)$

$$\left.\frac{\partial \rho}{\partial p}\right|_s = \left.\frac{\partial \rho}{\partial p}\right|_h - \frac{\partial s/\partial p|_h}{\partial s/\partial h|_p} \left.\frac{\partial \rho}{\partial h}\right|_p \tag{B.25}$$

From (B.21) and (B.18) we find

$$\frac{\partial s/\partial p|_h}{\partial s/\partial h|_p} = -\left.\frac{\partial h}{\partial p}\right|_s = -v = -\frac{1}{\rho} \tag{B.26}$$

Therefore

$$\frac{1}{a^2} = \left.\frac{\partial \rho}{\partial p}\right|_h + \frac{1}{\rho} \left.\frac{\partial \rho}{\partial h}\right|_p . \tag{B.27}$$

□

Second Derivatives

The equations above can easily be used to form second derivatives, which may be written as follows

$$\left.\frac{\partial A}{\partial B}\right|_C \qquad \text{with} \qquad A = \left.\frac{\partial a}{\partial b}\right|_c \tag{B.28}$$

The derivative $\partial A/\partial B|_C$ can be reduced to derivatives of A, B, C with respect to x or y. Therein, the derivatives $\partial A/\partial x|_y$ and/or $\partial A/\partial y|_x$ will appear. To compute these, A is reduced to derivatives of a, b, c with respect to x or y. Then A can be differentiated by x and/or y, which gives the required derivatives. In the same way derivatives of any desired order can be formed.

The above relations were implemented in a **Maple**-package. The resulting program **deriv**, developed for the work presented in [Bauer, 1999], reduces any first or second derivative of the thermodynamic functions T, p, v, h, u, s, f, g, x to derivatives of the fundamental equations $f(T, v)$, $g(T, p)$ or $h(s, p)$. The fundamental equation for the free energy is also included in the form of $f(T, \rho)$.

For every fundamental equation several basic properties and derivatives are used as an option to substitute for the derivatives of the fundamental equation. A shorthand notation is used in the following: subscript denotes derivatives, repeated subscripts higher order derivatives, variables that do not appear are held constant, e. g.,

$$f_{T\rho} = \frac{\partial}{\partial \rho} \left(\left.\frac{\partial f}{\partial T}\right|_\rho \right) \Bigg|_T$$

In the fundamental equation $f(T,\rho)$ these are

$$\begin{aligned}
p &= \rho^2 f_\rho \\
s &= -f_T \\
p_T &= \rho^2 f_{T\rho} \\
p_\rho &= 2\rho f_\rho + \rho^2 f_{\rho\rho} \\
c_v &= -T f_{TT} \\
p_{TT} &= \rho^2 f_{TT\rho} \\
p_{\rho\rho} &= 2f_\rho + 4\rho f_{\rho\rho} + \rho^2 f_{\rho\rho\rho} \\
p_{T\rho} &= 2\rho f_{T\rho} + \rho^2 f_{T\rho\rho} \\
c_{vT} &= -f_{TT} - T f_{TTT}
\end{aligned}$$

and for the fundamental equation $g(T,p)$ the basic derivatives are:

$$\begin{aligned}
s &= -g_T \\
v &= g_p \\
v_T &= g_{pT} \\
v_p &= g_{pp} \\
c_p &= -T * g_{pp} \\
v_{TT} &= g_{pTT} \\
v_{pp} &= g_{ppp} \\
v_{Tp} &= g_{ppT} \\
c_{pT} &== -g_{TT} - T g_{TTT}
\end{aligned}$$

where c_v is the specific isochoric and c_p is the specific isobaric heat capacity. Except for these two quantity, subscripts denote derivatives.

Examples for the output of **deriv**:

$$\left.\frac{\partial\rho}{\partial p}\right|_h = \frac{\rho\,(c_v\rho + p_T)}{\rho^2 p_\rho c_v + T p_T^2} \qquad \left.\frac{\partial\rho}{\partial h}\right|_p = -\frac{\rho^2 p_T}{\rho^2 p_\rho c_v + T p_T^2}$$

B.3 Derivatives in the Two-Phase Region

In the two-phase region the derivatives cannot be obtained from differentiation of the fundamental equation. The two-phase equilibrium conditions have to be taken into account. Some derivatives are undefined in the two-phase region because of the fixed coupling of p and T: $\partial h/\partial T|_p =$

$c_p = 1/\left.\partial T/\partial h\right|_p = 1/0$. In summary: all derivatives $\left.\partial T/\partial x\right|_p = 0$ and $\left.\partial p/\partial x\right|_T = 0$ with an arbitrary thermodynamic variable x and their inverses are undefined.

Density Derivatives

With $(x, y) = (v, T)$ (B.22) gives

$$\left.\frac{\partial v}{\partial h}\right|_p = \left[\left.\frac{\partial h}{\partial v}\right|_p\right]^{-1} = \frac{\left.\partial p/\partial T\right|_v}{\left.\partial h/\partial v\right|_T \left.\partial p/\partial T\right|_v - \left.\partial h/\partial T\right|_v \left.\partial p/\partial v\right|_T} \tag{B.29}$$

$$\left.\frac{\partial v}{\partial p}\right|_h = \left[\left.\frac{\partial p}{\partial v}\right|_h\right]^{-1} = \frac{\left.\partial h/\partial T\right|_v}{\left.\partial p/\partial v\right|_T \left.\partial h/\partial T\right|_v - \left.\partial p/\partial T\right|_v \left.\partial h/\partial v\right|_T} \tag{B.30}$$

Since $h = u + pv$ the enthalpy derivatives are

$$\left.\frac{\partial h}{\partial T}\right|_v = \left.\frac{\partial u}{\partial T}\right|_v + v \left.\frac{\partial p}{\partial T}\right|_v$$
$$\left.\frac{\partial h}{\partial v}\right|_T = \left.\frac{\partial u}{\partial v}\right|_T + v \left.\frac{\partial p}{\partial v}\right|_T + p$$

where

$$\left.\frac{\partial u}{\partial T}\right|_v =: c_v$$

is the specific isochoric heat capacity and $\left.\partial u/\partial v\right|_T$ can be obtained from differentiation of $u = f + Ts$. Employing (B.9) yields

$$\left.\frac{\partial u}{\partial v}\right|_T = \frac{\partial f}{\partial v} + T \left.\frac{\partial s}{\partial v}\right|_T = -p - T\frac{\partial^2 f}{\partial T \partial v} = -p + T \left.\frac{\partial p}{\partial T}\right|_v$$

Therefore

$$\left.\frac{\partial h}{\partial T}\right|_v = c_v + v \left.\frac{\partial p}{\partial T}\right|_v \tag{B.31}$$

$$\left.\frac{\partial h}{\partial v}\right|_T = T \left.\frac{\partial p}{\partial T}\right|_v + v \left.\frac{\partial p}{\partial v}\right|_T \tag{B.32}$$

Now (B.29)and (B.30) get

$$\left.\frac{\partial v}{\partial h}\right|_p = \frac{\left.\partial p/\partial T\right|_v}{T(\left.\partial p/\partial T\right|_v)^2 - c_v \left.\partial p/\partial v\right|_T}$$
$$\left.\frac{\partial v}{\partial p}\right|_h = \frac{c_v + v \left.\partial p/\partial T\right|_v}{c_v \left.\partial p/\partial v\right|_T - T(\left.\partial p/\partial T\right|_v)^2}$$

In case of two-phase equilibrium and constant composition the pressure is a function of the temperature only, thus

$$\left.\frac{\partial p}{\partial v}\right|_T = 0 \qquad \left.\frac{\partial p}{\partial T}\right|_v = \frac{dp}{dT}$$

The Clausius-Clapeyron relation defines the gradient of the saturation pressure:

$$\frac{dp}{dT} = \frac{s'' - s'}{v'' - v'} = \frac{1}{T}\frac{h'' - h'}{v'' - v'}$$

The equations simplify to

$$\left.\frac{\partial v}{\partial h}\right|_p = \frac{1}{T}\frac{dT}{dp} = \frac{v'' - v'}{s'' - s'}$$

$$\left.\frac{\partial v}{\partial p}\right|_h = -\frac{c_v + v(dp/dT)}{T(dp/dT)^2} = -\frac{c_v}{T}\left(\frac{dT}{dp}\right)^2 - \frac{v}{T}\frac{dT}{dp}$$

The last equation is easily transformed into density derivatives as $d = 1/v$:

$$\left.\frac{\partial \rho}{\partial h}\right|_p = -\rho^2 \left.\frac{\partial v}{\partial h}\right|_p = -\frac{\rho^2}{T}\frac{dT}{dp}$$

$$\left.\frac{\partial \rho}{\partial p}\right|_h = -\rho^2 \left.\frac{\partial v}{\partial p}\right|_h = \frac{\rho^2 c_v}{T}\left(\frac{dT}{dp}\right)^2 + \frac{\rho}{T}\frac{dT}{dp}$$

Heat Capacity

To compute the isochoric heat capacity in the two-phase region (B.22) is applied with (T, x) as independent parameters

$$c_v = \left.\frac{\partial u}{\partial T}\right|_x - \left.\frac{\partial v}{\partial T}\right|_x \frac{\partial u/\partial x|_T}{\partial v/\partial x|_T}$$

In the last term, x can be canceled down:

$$c_v = \left.\frac{\partial u}{\partial T}\right|_x - \left.\frac{\partial v}{\partial T}\right|_x \left.\frac{\partial u}{\partial v}\right|_T$$

From (B.32) and (B.3) we find

$$\left.\frac{\partial u}{\partial v}\right|_T = T\frac{dp}{dT} - p$$

Differentiation of $u = xu'' + (1-x)u'$ and $v = xv'' + (1-x)v'$ yields, since the liquid and vapor properties are a function of T only,

$$\left.\frac{\partial u}{\partial T}\right|_x = x\frac{du}{dT}^{''} + (1-x)\frac{du}{dT}^{'}$$
$$\left.\frac{\partial v}{\partial T}\right|_x = x\frac{dv}{dT}^{''} + (1-x)\frac{dv}{dT}^{'}$$

The total differentials of u and v can be written as

$$\frac{du}{dT} = \left.\frac{\partial u}{\partial T}\right|_p + \left.\frac{\partial u}{\partial p}\right|_T \frac{dp}{dT}$$
$$\frac{dv}{dT} = \left.\frac{\partial v}{\partial T}\right|_p + \left.\frac{\partial v}{\partial p}\right|_T \frac{dp}{dT}$$

Application of (B.20) yields

$$\left.\frac{\partial u}{\partial T}\right|_p = -\frac{\partial u/\partial T|_v}{\partial u/\partial v|_T} \qquad \left.\frac{\partial v}{\partial T}\right|_p = -\frac{\partial p/\partial T|_v}{\partial p/\partial v|_T}$$

and from (B.22) we find

$$\left.\frac{\partial u}{\partial p}\right|_T = \left.\frac{\partial u}{\partial v}\right|_T - \left.\frac{\partial u}{\partial T}\right|_v \frac{\partial p/\partial T|_v}{\partial p/\partial v|_T}$$

Therein $\partial u/\partial T|_v$ and $\partial u/\partial v|_T$ are known from (B.3) and (B.3). Altogether we obtain

$$c_v = x\tilde{c}_v'' + (1-x)\tilde{c}_v'$$

where $\tilde{c}_v''$ and $\tilde{c}_v'$ are the limiting isochoric heat capacities on the dew and boiling point when approached from the two-phase region

$$\tilde{c}_v'' = c_v'' - \frac{T}{\partial p/\partial v|_T''}\left(\frac{dp}{dT} - \left.\frac{\partial p}{\partial T}\right|_\rho''\right)^2$$
$$\tilde{c}_v' = c_v' - \frac{T}{\partial p/\partial v|_T'}\left(\frac{dp}{dT} - \left.\frac{\partial p}{\partial T}\right|_\rho'\right)^2$$

C

Moving Boundary Models

This appendix details some of the derivations of the moving boundary model equations. For the sake of a concise treatment of the key ideas in Chapter 4, it has been moved here. The nomenclature is the same as in Section 4.11.

Mass- and Energy Balances

The detailed derivation of the mass- and energy balances of the superheated and two phase zones of the three region moving boundary model is done as follows.

Mass Balance for the Two-Phase Region

The mass balance (4.6) is integrated over the two-phase region from L_1 to $L_1 + L_2$. Applying Leibnitz's rule to a constant area pipe yields

$$A\frac{d}{dt}\int_{L_1}^{L_1+L_2}\rho dz + A\rho_l\frac{dL_1}{dt} - A\rho_g\frac{d(L_1+L_2)}{dt} + \dot{m}_{23} - \dot{m}_{12} = 0.$$

The flow is assumed to be homogeneous at equilibrium conditions with a mean density of $\rho = \bar{\gamma}\rho_g + (1-\bar{\gamma})\rho_l$. The mass balance for the two-phase region becomes

$$A\left[\frac{d}{dt}(\rho_2 L_2) + (\rho_l - \rho_g)\frac{dL_1}{dt} - \rho_g\frac{dL_2}{dt}\right] = \dot{m}_{12} - \dot{m}_{23} \tag{C.1}$$

where $\rho_2 = \bar{\gamma}\rho_g + (1-\bar{\gamma})\rho_l$. Assuming that $d\bar{\gamma}/dt = 0$, the time derivative of ρ_2 can be expanded to

$$\frac{d\rho_2}{dt} = \left(\bar{\gamma}\frac{d\rho_g}{dp} + (1-\bar{\gamma})\frac{d\rho_l}{dp}\right)\frac{dp}{dt}.$$

which inserted into the mass balance (C.1) gives the final mass balance for the two-phase region as stated in (4.62).

Energy Balance for the Two-Phase Region

The energy balance (4.48) is integrated over the two-phase region from L_1 to $L_1 + L_2$. For a constant area pipe this gives

$$\begin{aligned} &A\frac{d}{dt}\int_{L_1}^{L_1+L_2}\rho h dz + A\rho_l h_l \frac{dL_1}{dt} - A(L_1+L_2)\frac{dp}{dt} \\ &- A\rho_g h_g \frac{d(L_1+L_2)}{dt} + \dot{m}_{23}h_g - \dot{m}_{12}h_l = q'_{w22}. \end{aligned} \tag{C.2}$$

Note that ρ_l, h_l are the fluid conditions at L_1 and ρ_g, h_g are the fluid conditions at $(L_1 + L_2)$. The first term is evaluated as

$$\begin{aligned} \frac{d}{dt}\int_{L_1}^{L_1+L_2}\rho h dz =& \frac{d}{dt}\int_{L_1}^{L_1+L_2}\left(\gamma\rho_g h_g + (1-\gamma)\rho_l h_l\right) dz \\ =& \frac{d}{dt}\left[\left(\bar{\gamma}\rho_g h_g + (1-\bar{\gamma})\rho_l h_l\right) L_2\right] \\ =& L_2\left[\bar{\gamma}\frac{d(\rho_g h_g)}{dp} + (1-\bar{\gamma})\frac{d(\rho_l h_l)}{dp}\right]\frac{dp}{dt} \\ &+ \left[\bar{\gamma}\rho_g h_g + (1-\bar{\gamma})\rho_l h_l\right]\frac{dL_2}{dt} \end{aligned} \tag{C.3}$$

Inserting (C.3) into (C.2) gives the final energy balance for the two-phase region as state in (4.63).

Mass Balance for the Superheated Region

The mass balance (4.6) is integrated over the superheated region from $L_1 + L_2$ to L. This yields

$$\int_{L_1+L_2}^{L}\frac{\partial A\rho}{\partial t}dz + \int_{L_1+L_2}^{L}\frac{\partial \dot{m}}{\partial z}dz = 0.$$

Integrating the first term and integrating the second term give for a constant area pipe

$$A\frac{d}{dt}\int_{L_1+L_2}^{L}\rho dz + A\rho(L_1+L_2)\frac{d(L_1+L_2)}{dt} + \dot{m}_{out} - \dot{m}_{23} = 0. \tag{C.4}$$

The mean density in the superheated region is $\rho_3 = \frac{1}{L_3}\int_{L_1+L_2}^{L}\rho dz \approx \rho(p, h_3)$, which inserted in the mass balance (C.4) gives

$$A\left[L_3\frac{d\rho_3}{dt} + (\rho_g - \rho_3)\frac{dL_1}{dt} + (\rho_g - \rho_3)\frac{dL_2}{dt}\right] = \dot{m}_{23} - \dot{m}_{out}. \tag{C.5}$$

The derivative of ρ_3 is calculated as

$$\begin{aligned}\frac{d\rho_3}{dt} =& \frac{\partial \rho_3}{\partial p}\Big|_h \frac{dp}{dt} + \frac{\partial \rho_3}{\partial h}\Big|_p \frac{dh}{dt} \\ =& \Big(\frac{1}{2}\frac{\partial \rho_3}{\partial h_3}\Big|_p \frac{dh_g}{dp} + \frac{\partial \rho_3}{\partial p}\Big|_h\Big)\frac{dp}{dt} + \frac{1}{2}\frac{\partial \rho_3}{\partial h_3}\Big|_p \frac{dh_{out}}{dt}\end{aligned} \tag{C.6}$$

The expression for $\frac{d\rho_3}{dt}$ is inserted into (C.5), which gives the final mass balance for the superheated region as stated in (4.58).

Energy Balance for the Superheated Region

The energy Equation (4.48) is integrated over the superheated region from $L_1 + L_2$ to L. The integral evaluates to

$$\begin{aligned}A\frac{d}{dt}\int_{L_1+L_2}^{L} \rho h dz + A\rho(L_1+L_2)h(L_1+L_2)\frac{d(L_2)}{dt} \\ - AL_3\frac{dp}{dt} + \dot{m}_{out}h_o ut - \dot{m}_{23}h_g = q'_{w33}.\end{aligned} \tag{C.7}$$

The first term is calculated as

$$\begin{aligned}\frac{d}{dt}\int_{L_1+L_2}^{L} \rho h dz = \frac{d}{dt}(\bar{\rho}_3\bar{h}_3 L_3) = -\frac{1}{2}\bar{\rho}_3(h_g + h_{out})\Big(\frac{d(L_1+L_2)}{dt}\Big) \\ + \frac{1}{2}L_3(h_g + h_{out})\frac{d\bar{\rho}_3}{dt} + \frac{1}{2}\bar{\rho}_3 L_3\Big(\frac{dh_g}{dp}\frac{dp}{dt} + \frac{dh_{out}}{dt}\Big)\end{aligned} \tag{C.8}$$

where $\bar{h}_3 = \frac{1}{2}(h_g + h_{out})$ and $\bar{\rho}_3 = \rho(p, \bar{h}_3)$. Equation (C.8) and the expression for $\frac{d\bar{\rho}_3}{dt}$ from equation (C.6) is inserted into (C.7), which after some rearranging gives the final energy balance for the superheated region, Equation (4.59).

Energy Balance for the Pipe Walls

For the wall region adjacent to the two-phase region $\alpha = L_1$ and $\beta = L_1 + L_2$, which inserted in (4.65) gives

$$\begin{aligned}C_w\rho_w A_w\Bigg[L_2\frac{dT_{w2}}{dt} + (T_w(L_1) - T_{w2})\frac{dL_1}{dt} \\ + (T_{w2} - T_w(L_1+L_2))\frac{dL_2}{dt}\Bigg] = -q'_{w22} + q'_{ambw2}\end{aligned} \tag{C.9}$$

$T_w(L_1)$ is given by (4.67), and $T_w(L_1 + L_2)$ is given by

$$\begin{aligned} T_w(L_1 + L_2) &= T_{w3} \text{ for } \frac{dL_2}{dt} > 0 \\ T_w(L_1 + L_2) &= T_{w2} \text{ for } \frac{dL_2}{dt} \leq 0 \end{aligned} \tag{C.10}$$

For the wall region adjacent to the superheated region $\alpha = L_1 + L_2$ and $\beta = L$, which inserted in (4.65) gives

$$\begin{aligned} C_w \rho_w A_w \Bigg[& L_3 \frac{dT_{w3}}{dt} + (T_w(L_1) - T_{w2}) \frac{dL_1}{dt} \\ & + (T_w(L_1 + L_2) - T_{w3}) \left(\frac{dL_1}{dt} + \frac{dL_2}{dt} \right) \Bigg] = -q'_{w33} + q'_{ambw3}. \end{aligned} \tag{C.11}$$

D

Modelica Language Constructs

The Modelica language specification[1] defines the notions of subtype and type equivalence for classes as follows:

For any classes $\mathcal{S}$ and C, $\mathcal{S}$ is a supertype of C and C is a subtype of $\mathcal{S}$ if they are equivalent or if:

- every public declaration element of $\mathcal{S}$ also exists in C (according to their names)
- those element types in $\mathcal{S}$ are supertypes of the corresponding element types in C.

A base class is the class referred to in an extends clause. The class containing the extends clause is called the derived class. Example: Base classes of C are typically supertypes of C, but other classes not related by inheritance can also be supertypes of C.

Two types $\mathcal{T}$ and $\mathcal{U}$ are equivalent if:

- $\mathcal{T}$ and $\mathcal{U}$ denote the same built-in type (one of RealType, IntegerType, StringType or BooleanType), or
- $\mathcal{T}$ and $\mathcal{U}$ are classes, $\mathcal{T}$ and $\mathcal{U}$ contain the same public declaration elements (according to their names), and the elements types in $\mathcal{T}$ are equivalent to the corresponding element types in $\mathcal{U}$.

The specification defines further details about type relations (*type identity* and *ordered type identity*, but the above definitions suffice for the present discussion. The distinguishing feature of the subtype relations is that it is not based on inheritance but on the above definitions instead. This is a necessary condition for using multi-facet models in a safe way and in combination with validated model libraries.

[1] Here quoted from version 1.4 at http://www.modelica.org