A modular concept and a computer language for modeling and simulation of a passive boiling water reactor
A Modular Concept and a Computer Language for Modeling and Simulation of a Passive Boiling Water Reactor

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
for the degree of
Doctor of Technical Sciences

presented by
Frank Möhle
Dipl. Informatik-Ing. ETH
born January 26, 1968
citizen of Switzerland

accepted on the recommendation of
PD Dr. J. Halin, examiner
Prof. Dr. G. Yadigaroglu, co-examiner
Prof. Dr. R. S. Abhari, co-examiner

2005
Contents

1. Introduction ...................................... 1
   1.1. The "passive" Light Water Reactors ............ 1
       1.1.1. The Simplified Boiling Water Reactor ........ 2
   1.2. Model of the SBWR Containment .............. 5
   1.3. The Conceptional Approach .................. 7

2. Physical Description of Modules ................. 9
   2.1. Pressure Vessel .......................... 12
   2.2. Gas Space ................................ 13
       2.2.1. Conservation of Mass .................. 13
       2.2.2. Conservation of Energy ............... 14
       2.2.3. Equations of State .................. 16
       2.2.4. Coupling with Liquid Space .......... 16
       2.2.5. Complete Model of the Gas Space ....... 17
   2.3. Liquid Space ............................ 17
       2.3.1. Conservation of Mass .................. 18
       2.3.2. Conservation of Energy ............... 18
       2.3.3. Change of Volume .................... 20
       2.3.4. Complete Model of the Liquid Space .... 23
   2.4. Interface ................................ 23
       2.4.1. Pure Steam above the Interface ......... 23
       2.4.2. Gas Mixture above the Interface ....... 24
       2.4.3. Two-phase Liquid Space ................ 25
       2.4.4. Heat and Mass Transfer Coefficients .... 26
       2.4.5. Complete Model for the Interface ....... 29
   2.5. The Pipe ................................ 29
   2.6. The PCCS Condenser Unit .................. 31
       2.6.1. PCCS Operation Modes ................ 32
       2.6.2. The Condenser Tube .................. 37
       2.6.3. Heat Transfer Coefficients .......... 41
       2.6.4. Condensation in Suppression Pools ...... 43
       2.6.5. The Condenser Unit .................. 45
   2.7. Vent Line .................................. 47
   2.8. Properties of Steam, Air, and Water .......... 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.6. Output Files and Debugging Information</td>
<td>185</td>
</tr>
<tr>
<td>C.7. Plotting Results using Gnuplot</td>
<td>186</td>
</tr>
<tr>
<td>D. An Example MSDL Program</td>
<td>187</td>
</tr>
<tr>
<td>D.1. MSDL Source</td>
<td>187</td>
</tr>
<tr>
<td>D.2. Generated Fortran Code</td>
<td>188</td>
</tr>
<tr>
<td>E. Third-Party Numerical Software</td>
<td>201</td>
</tr>
<tr>
<td>E.1. Software for Solving Systems of ODEs</td>
<td>201</td>
</tr>
<tr>
<td>E.1.1. Introduction</td>
<td>201</td>
</tr>
<tr>
<td>E.1.2. Summary of the explicit ODEPACK Solvers</td>
<td>203</td>
</tr>
<tr>
<td>E.2. Description and Use of LSODAR</td>
<td>205</td>
</tr>
<tr>
<td>E.2.1. Summary of Usage</td>
<td>206</td>
</tr>
<tr>
<td>E.2.2. Examples for using DLSODAR</td>
<td>207</td>
</tr>
<tr>
<td>E.3. BLAS (Basic Linear Algebra Subsystem)</td>
<td>231</td>
</tr>
<tr>
<td>E.4. LINPACK</td>
<td>233</td>
</tr>
<tr>
<td>F. The GNU General Public License</td>
<td>235</td>
</tr>
<tr>
<td>F.1. How to Apply These Terms to Your New Programs</td>
<td>242</td>
</tr>
<tr>
<td>G. Nomenclature</td>
<td>245</td>
</tr>
<tr>
<td>G.1. Symbols</td>
<td>245</td>
</tr>
<tr>
<td>G.2. Subscripts</td>
<td>246</td>
</tr>
<tr>
<td>H. Abbreviations</td>
<td>247</td>
</tr>
<tr>
<td>List of Figures</td>
<td>251</td>
</tr>
<tr>
<td>List of Tables</td>
<td>253</td>
</tr>
<tr>
<td>Listings</td>
<td>255</td>
</tr>
<tr>
<td>Bibliography</td>
<td>257</td>
</tr>
<tr>
<td>Curriculum Vitae</td>
<td>266</td>
</tr>
</tbody>
</table>
Abstract

A modular concept for the simulation of a general passive boiling water reactor (BWR) pressure vessel and containment system is developed. The model is based on the assumption of slow transients and well mixed gas and liquid in the containments so that ordinary differential equations (ODEs) can be used for the description of the thermal-hydraulic phenomena instead of partial differential equations (PDEs).

Each module represents a special component of a physical system. A module is written in an advanced formula manipulation language (Maple V) which permits an easy programming of the typically implicitly given ODEs describing this subsystem and their automated transformation into an explicit form required by the subsequent numerical integration.

For handling the interaction between different modules a new language (MSDL: Modular System Description Language) and a corresponding compiler have been developed. MSDL is the language to be applied by the user. In his program the user essentially calls the modules written in Maple.

The formula manipulation of the relatively small input code in MSDL results in automatically generated Fortran-code of typically several thousand statements. This intermediate code reflects different physical states of the system, which can change during simulation due to discontinuities.

The advantage of this new concept is the capability of changing parts of the model with a marginal programming effort. For this kind of application the exact tracking of discontinuities and other events is important. Discontinuities may result from simple changes in input-parameters, all the way to complex changes of the governing model equations at a certain time, e.g. the opening or closing of valves.

For the efficient and stable integration of the differential equations, it is an advantage to apply algorithms of variable order and variable step size with automatic stiffness detection and root finding capabilities for the localization of discontinuous events. Either an Adams-Moulton algorithm or a Gear-type integrator are applied, depending on what is most appropriate.

After having tested the feasibility of the concept, the number of MSDL-modules has been increased and collected in a library.

Finally, the results of a complex simulation is discussed and com-
pared against a real test run of the PANDA Facility operated by PSI in Villigen, Switzerland.
Zusammenfassung

Es soll ein modulares Konzept zur Simulation eines generellen passiven Siedewasserreaktors, bestehend aus Druckbehälter und Sicherheitsbehälter entwickelt werden. Das Modell baut auf der Annahme auf, dass nur langsame transienten auftreten und sowohl Gas als auch flüssiges Wasser stets gut gemischt sind, sodass die Modellierung mit gewöhnlichen Differentialgleichungen für die Beschreibung der Thermo-hydraulischen Phänomene auskommt. Aus diesem Grunde kann auf partielle Differentialgleichungen verzichtet werden.

Jede Systemkomponente wird durch ein eigenes Modul repräsentiert. Die Module werden in einer modernen Sprache zur Formelmanipulation (Maple V) formuliert. Dies erlaubt sowohl eine vergleichswise einfache Programmierung der typischerweise impliziten Differentialgleichungen, welche das betreffende Modell beschreiben, als auch eine automatische Transformation der Gleichungen in ihre explizite Form, wie sie typischerweise für die später erfolgende numerische Integration benötigt werden.

Um die Interaktion der einzelnen Module handhaben zu können, wurde eine neue Sprache (MSDL: Modular System Description Language) sowie ein dazugehöriger Compiler entwickelt. Der Benutzer formuliert den Aufbau eines Systemes in MSDL; der Compiler übersetzt MSDL in die entsprechenden Modulauftrufe.


Für die effiziente und stabile Integration der Differentialgleichungen werden mit Vorteil Integrationsalgorithmen variabler Ordnung

Nachdem die prinzipielle Machbarkeit des Konzeptes überprüft wurde, wurden alle zur simulation benötigten Module implementiert und in einer Bibliothek zusammengefasst.

Schliesslich werden die Resultate einer Komplexen Simulation diskutiert und den experimentellen Testresultaten des PANDA Aufbaues des PSI in Villigen, Schweiz gegenübergestellt.
A new generation of nuclear reactors is being designed worldwide. These include both so-called “evolutionary”, as well as “passive” designs. The 600 MWe passive pressurized water reactor AP-600 [Bru94] and the Simplified Boiling Water Reactor (SBWR) [UCS93] are examples of the latter. Various alternatives to the passive designs are being considered in the U.S. [MLB93] [Yed94], Europe (Italy, Germany, etc.) [MBF93], Japan [TYA94] and other Pacific Basin countries, and elsewhere.

In all the passive designs an attempt is made to substitute relatively simple passive systems relying on natural circulation, gravity and stored energy and “heat sinks”, for the relatively complicated emergency cooling systems of the classical present generation of reactors. Thus both, the cooling of the core following the so-called Loss-of-Coolant Accident (LOCA) as well as the removal of the decay heat from the primary system and the containment rely on totally passive means. The elimination of the complex, redundant, safety-grade emergency cooling systems provides simplification in design, operations
and during accidental situations and consequently also cost reductions, without diminishing overall safety, and possibly by even increasing it [MLB93].

1.1.1. The Simplified Boiling Water Reactor

This thesis deals with certain safety aspects and with modeling of the behavior following a LOCA of the SBWR with an emphasis on the modeling of the long-term containment behavior following the LOCA.

Analytical tools for such analyses have been available for several years now, but they have not been fully tested in relation to passive systems. Moreover, simulations examined here exhibit certain characteristics that are not totally in line with the capabilities of the available standard safety analysis tools. (These are system analysis codes like RELAP5, TRAC, etc.)

Description of the SBWR

In contrast to the classical modern BWR designs, circulation inside the SBWR pressure vessel is gravity driven (natural circulation); there are no pumps driving the recirculation flow. To achieve good natural circulation, the SBWR has a tall adiabatic "chimney", above the core [UCS93].

In case of a break in the primary coolant system, a mixture of steam and water is "blown down", from the pressure vessel into the containment.

Emergency core cooling water must first be added to the RPV to keep the core covered. The steam pressurizes the containment, and transports the decay heat from the reactor core to the containment atmosphere. Consequently, this decay heat must be removed from the containment.

The SBWR uses also gravity or natural circulation-driven, passive safety systems to provide emergency coolant to the core, to keep the core cooled and to remove decay heat from both the primary system and/or the containment. The main systems performing these tasks, as shown in figure 1.1, are the Gravity-Driven Cooling System (GDCS), the Isolation Condenser System (ICS), and the Passive Containment Cooling System (PCCS), [UCS93].

Emergency core cooling water is provided to the core by the GDCS. This system consists of three water pools situated above the top of
Figure 1.1: Schematic of the Simplified Boiling Water Reactor and its PANDA model
the core, from which makeup coolant can flow by gravity to replenish the coolant lost from the Reactor Pressure Vessel (RPV). However, the GDCS can operate only after depressurization of the RPV; therefore, the SBWR is equipped with an Automatic Depressurization System that performs this function. The depressurization of BWR primary systems is well understood, since it has been studied extensively in relation to the classical BWR designs. Indeed, the phenomena taking place during the early phase of blow-down inside the RPV have been extensively investigated.

Decay heat removal from the primary system while it is intact or under high pressure is performed by the ICS. The ICS consists of three Isolation Condensers (IC) located in a pool on the top of the reactor building, as shown in figure 1.1. When redundant condensate return valves are opened, steam from the primary system flows into the tubes of the ICs, condenses, and returns to the RPV, removing stored energy. The behavior of the ICs is well understood, since such units have been in operation for many years in older BWRs. Thus, there is no specific need to experimentally verify the high-pressure operation of the SBWR decay heat removal system.

Decay heat is removed from the drywell (DW) by the PCCS, which employs three PCC condensers also located in interconnected IC pool compartments on top of the reactor building. The PCC condenser tubes are permanently connected to the DW. A mixture of steam and non-condensible gases (nitrogen present in the containment during normal operation) may enter the PCC condensers. The steam will condense, while the non-condensible gases must be vented to assure proper operation of the condensers. This is accomplished by conveying and venting the non-condensible gases into the suppression pool (SP) in the Suppression Chamber (SC) (or Wetwell).

Since the DW volume is connected directly to the SP either via the main pressure suppression vents or through the PCC condensers and their vent lines, the path that the steam will follow depends on the pressure differences between the DW volume and the two possible venting points. During the long-term containment cooling period, direct opening of the main vents and condensation of the steam in the SP must be avoided, since the SP is not provided with a safety-grade cooling system; the steam must be condensed in the PCC (or IC) condensers and any non-condensibles vented to the SC. Although the operation of the condensers is understood, experimental verification of their integral, system behavior under a variety of condition was
1.2. Model of the SBWR Containment

As mentioned earlier, the containment transients in question are very slow and the periods of concern last hours or days; therefore, there are no highly transient phenomena to be considered. During such a long-term cooling phase, the reactor system operates in a rather simple "quasi steady-state" way, and the space distribution of various variables becomes of secondary importance; most of the components of the system can be treated as having a zero dimension. This reduces the complexity of the differential equations governing the system behavior: Instead of these being partial equations, the system can now be described by ordinary differential equations. Moreover, one can take advantage of the fact that several phenomena can be described in a quasi steady-state fashion.

The behavior of the containment system of the SBWR can be described by state equations governing the behavior of containment volumes: The continuity and energy equations are used to calculate the state of the fluid in a given containment volume, assumed to be well

deemed necessary. Two experimental facilities were provided for this purpose. The GIRAFFE facility, operated by Toshiba in Japan [YNT91], provided extensive information about system behavior, especially in relation to long-term decay heat removal. The larger-scale PANDA facility at the Paul Scherrer Institute (PSI) in Switzerland [YCdC+95] [VDB+95] is providing additional information and addresses issues such as the effects of the operation of several condenser units in parallel, the distribution of the constituents (steam and non-condensibles) in the large DW volume, and mixing in the containment compartments.

During the low-pressure, long-term decay heat removal period, the decay heat is removed from the core by natural circulation inside the RPV. The steam that is produced is blown into the containment, condenses in the PCCS System, and the condensate returns to the RPV as makeup flow. The operation of the SBWR is a natural circulation reactor raises (again, since the concern had practically disappeared with the introduction of circulation pumps in the BWRs) concern about flow stability. Natural circulation could be particularly unstable during the low-pressure phases of operation of the system. Low-pressure natural circulation takes place during the long-term decay heat removal period from the containment.
mixed, as is most often the case for long-term transients. Typical containment volumes are the Drywell, the Wetwell, etc. From continuity and energy one obtains the time dependence of the state variables: pressure, temperature, etc.

The transfers of mass and energy between containment volumes can be described by integral momentum equations governing flows between containment volumes. The typical forms of these energy, state and momentum equations are given below [Moo90] [Wal69]

\[ \frac{dM}{dt} = \sum_i W_i \]  

\[ \frac{dM_j}{dt} = \sum_i W_{i,j} \]  

\[ M \frac{de}{dt} = -p \frac{dV}{dt} + \dot{Q} + \sum_i W_i (h_{0,i} - h) + \frac{P}{\rho} \sum_i W_i \]  

\[ Vf_2 \frac{dP}{dt} = \sum_i (W_i (h_{0,i} - h)) + \sum_i \frac{W_i P}{\rho} + \dot{Q} - p \frac{dV}{dt} - V \sum_j \left( f_{1,j} \frac{dy_j}{dt} \right) \]  

\[ \left( \frac{L}{a} \right) \frac{dW}{dt} = \Delta P + gL - \rho g H \left( \frac{F}{a^2} \right) - \frac{W^2}{2\rho} \]  

Equations 1.1 and 1.2 describe simply total and constituent \( j \) mass continuity for a volume containing a total mass \( M \) and a constituent \( j \) mass \( M_j \) in terms of the mass flow rates \( W_i \). Equation 1.3 describes the evolution of the specific energy content of the volume \( e \) in terms of heat addition \( \dot{Q} \) and the enthalpy flows \( Mh_i \) entering such a volume. Any changes of the volume with time (e.g. due to changes in the water level) \( \frac{dV}{dt} \) are also considered. \( P \) is the pressure and \( h \) the average enthalpy in the volume. Equation 1.4 is obtained by a combination of equations 1.1 to 1.3 and governs the rate of pressurization \( \frac{dP}{dt} \) of the volume. Finally, equation 1.5 is obtained by integration of the momentum equation for a flow path connecting two volumes. It governs the flow rate \( M_i \) in this flow path due to a pressure difference \( \Delta P \) and includes the effects of friction (the \( \frac{F}{a^2} \) term), gravity heads (the \( \rho g H \) terms), and inertia (the \( \frac{L}{a} \) term). One sees that they are ordinary differential equations (ODEs) in terms of the system variables, which are...
1.3. The Conceptional Approach

The computational concept for simulating complex systems is depicted in figure 1.2. This hierarchical design will be reflected throughout this work. In chapter 2, the physical models which are needed are developed. This section covers all the physics in the simulation concept. Thus all assumptions, simplifications and other physical details are described in this chapter.

The next fundamental chapter is 3 in which all numerical methods and software packages with respect to numerical problems are described. This includes integrators for systems of ordinary differen-
tial equations (ODE)s and algebraic equations (AE)s.

Based on the first two fundamental chapters, chapter 4 on computer algebra introduces the formula manipulation needed to convert the formulas from chapter 2 into a computer code.

Details on how the implementation of the physical modules is done and how this modules can be used in a simulation program is described in chapter A. Input and output variables of the modules are defined as well as implementational detail of interest are described.

As the systems considered throughout this work become rather complex and large, an input language is needed, which describes a physical system built up from modules described in chapter 2. The output of the corresponding compiler should be a compilable computer code which represents the appropriate system. Details and definitions of the language are given in chapter 5.

Finally, some important case studies of actual computer simulations using the presented code are presented in chapter 6. Some are compared with actual experiments or calculations using other simulation systems.

Possible enhancements and needed changes for future development of this code are given chapter 7.
Physical Description of Modules

For the description of a complex thermal hydraulic system, a modularization is needed. This means that a large system will be divided into smaller parts or subsystems. These subsystems are called modules. The modules are independent from each other and can be considered as isolated subsystems.

The word modularization may be understood in different ways. The first interpretation, used by computer scientists for huge computer source codes, may lead to wrong expectations. For computer code, modularization means the subdivision of a program code into several subroutines for different tasks and mathematical methods, often not reflecting the different modules of the physical model under consideration.

The other way to interpret the word modularization is based on the physical system. This also leads to a modularized computer code, but the modularization is done with respect to the physical components rather than to mathematical means. Of course, also a module describ-
ing a physical component can be further subdivided into submodules describing different physical situations and phenomena.

In this work, the modularization is primarily based on the physical components. For this reason, it is possible to describe the physics of each of the modules separately and to transform the governing systems of ODEs later on to a certain source form, e.g. by the computer algebra language MapleV, from where these systems of equations are transformed automatically to software modules. These modules now can be used by a simulation program, which simulates a specified transient of a system under specific conditions.

The concept of modularization is by no means new. New, however, is the way how model equations in the modules get manipulated by a language for symbolic computation and formula manipulation. This will be explained in chapter A.

In this chapter, the physics of each of the modules is presented. These modules are needed to allow the description of phenomena resulting in the kind of transients to be analyzed in this work. An example of such a physical system is given in figure 2.1. There the actual experimental facility PANDA [HDLA] at the Paul Scherrer Institute (PSI) is depicted schematically. A detailed description of the PANDA facility can be found in chapter 6 on page 97. The modules needed for this system are:

- A pressure vessel containing a liquid part and a gas part. In figure 2.1 such volumes are the Reactor Pressure Vessel (RPV), the Gravity Driven Cooling System (GDCS) component, and the wetwells (WW1, WW2). This kind of vessel will be split into two separated modules: one for the gas part and one for the liquid part.

- A pressure vessel which contains a gas mixture of air and steam. Such vessels are used for the drywells (DW1 and DW2) in figure 2.1, as well as for the gas part of the above described pressure vessel containing liquid and gas.

- The liquid part of the pressure vessels such as the RPV.

- The 3 condensers for the Passive Containment Cooling System (PCCS) used in figure 2.1 named PCC1, PCC2, PCC3, and the Isolation Condenser (IC).

- Pipes which connect the volumes. Two types of pipes are needed for either connecting two gas or two liquid volumes.
Figure 2.1: PANDA experimental facility scheme
• Vent lines, which connect the drywells to the wetwells in figure 2.1. The vent lines do not allow the pressure to exceed the pressure in the wetwells by a given amount, depending on the liquid level in the corresponding suppression pool. If the pressure in the drywell exceeds this limit, gas is blown into the suppression pool, where the steam will condense. The non-condensible (air) will rise up to the suppression chamber gas space.

• The last module needed is the vacuum breaker (VB) which prevents the pressure in the wetwell to exceed the pressure in the corresponding drywell.

Common parameters for the modules describing volumes are heat losses through the walls, heating capabilities, and geometrical descriptions. For the pipes, only geometrical descriptions are needed, as it is assumed that the pipes do neither store energy nor mass.

Many papers on modeling thermo-hydraulic systems have been published. For detailed discussions on specific topics, refer to [Wul96], [Moo90], [oNRR97], and others.

2.1. Pressure Vessel

In the sense of modularization, a pressure vessel containing liquid water and a gas mixture above can be split into three independent modules: The water space, the gas space and the interface. The model to be used is based on the following assumptions:

• The pressure vessel is modeled as a cylinder with a constant cross section. The volume is enclosed by essentially flat plates. As the volumes of the water and the gas space, respectively, are rather large and no local effects have to be considered within the scope of the use of this module, there is no need for considering more complexity in the geometries.

• The total pressure \( p \) is space independent.

• Each phase is always well mixed. Injected mass will mix up instantaneously.

• The gas mixture is composed of steam and air. It may happen that one of the components vanishes.
2.2. Gas Space

The following three sections will describe the physics involved in each of the modules: Gas space, liquid space and the interface in between. In figure 2.2 the main variables of the three modules are shown.

2.2. Gas Space

As the gas is a mixture of steam and air, simple relations apply for the temperatures and pressures:

\[ T_g = T_v = T_a \]  \hspace{1cm} (2.1)

and

\[ p_g = p_v + p_a, \]  \hspace{1cm} (2.2)

respectively.

2.2.1. Conservation of Mass

The conservation of mass has to be fulfilled for both, steam and air. The conservation of mass of steam takes the form

\[ \frac{dM_v}{dt} = \dot{M}_{fg} + \dot{M}_b - \dot{M}_{out}x_v + \dot{M}_{v,in} \]  \hspace{1cm} (2.3)
where $M_{fg}$ is the condensation or evaporation rate at the interface, $M_b$ is the mass flow rate due to rising bubbles if the water below is boiling, $M_{out}$ is the leaving mass flow rate, $x_v$ is the steam mass fraction of the leaving gas and $M_{v,in}$ is the mass flow rate of entering steam (from external sources).

The mass conservation of air takes a similar form, but without the terms for evaporation and rising bubbles:

$$\frac{dM_a}{dt} = -(1 - x_v) M_{out} + M_{a,in} \tag{2.4}$$

where

$$x_v = \frac{M_v}{M_g} = \frac{\rho_v}{\rho_g}. \tag{2.5}$$

Considering the gas as a perfect gas mixture, a relation between the steam fraction and the partial pressures can be found:

$$\phi_v = \frac{M_a x_v}{(M_a - M_v) x_v + M_v} = \frac{p_v}{p_g} \tag{2.6}$$

where $M_a$ and $M_v$ are the molar weights of air and steam, respectively. $\phi_v$ is the molar fraction of vapor. Similarly, the molar fraction of air is given by

$$\phi_a = \frac{M_v(1 - x_v)}{(M_a - M_v) x_v + M_v} = \frac{p_a}{p_g}. \tag{2.7}$$

The time derivative of (2.6) takes the form

$$\frac{d\phi_v}{dt} = \frac{M_a M_v d x_v}{(-x_v M_a + x_v M_v - M_v)^2} = \frac{1}{p_g^2} \left( \frac{p_g}{dt} - \frac{d p_g}{dt} \right). \tag{2.8}$$

### 2.2.2. Conservation of Energy

In the previous subsection the conservation of mass was formulated. The conservation of energy has to be expressed in a similar way. Starting from the basic form for the conservation of energy for a control volume,

$$\frac{dE}{dt} = -p \frac{dV}{dt} + \dot{Q} + \sum_i M_i h_i. \tag{2.9}$$

Equation (2.9) can be transformed to an enthalpy conservation equation using $E = H - pV$.

$$\frac{dE}{dt} = \frac{dH}{dt} - \frac{d}{dt}(pV) \tag{2.10}$$
where \( H = hM \). Thus
\[
\frac{dE}{dt} = M \frac{dh}{dt} + h \frac{dM}{dt} - \frac{d}{dt}(pV). \tag{2.11}
\]

Using the continuity equation
\[
\frac{dM}{dt} = \sum_i \dot{M}_i \tag{2.12}
\]
we get
\[
\frac{dE}{dt} = M \frac{dh}{dt} + h \sum_i \dot{M}_i - \frac{d}{dt}(pV). \tag{2.13}
\]

This can be inserted in equation (2.9). Thus we get
\[
M \frac{dh}{dt} + h \sum_i \dot{M}_i - p \frac{dV}{dt} - V \frac{dp}{dt} = -p \frac{dV}{dt} + \dot{Q} + \sum_i \dot{M}_i h_i \tag{2.14}
\]
which can be simplified to
\[
M \frac{dh}{dt} = \dot{Q} + \sum_i \dot{M}_i (h_i - h) + V \frac{dp}{dt}. \tag{2.15}
\]

For the gas volume, the equation becomes
\[
h_v \frac{dM_v}{dt} + h_a \frac{dM_a}{dt} + M_v \frac{dh_v}{dt} + M_a \frac{h_a}{dt} = \sum \dot{Q} - \sum \dot{M}_i h_i + V \frac{dp_g}{dt} \tag{2.16}
\]
where the sum \( \sum \dot{M}_i h_i \) consists of
\[
\sum \dot{M}_i h_i = \dot{M}_f g h_{iv} + \dot{M}_b h_b - \dot{M}_o ut x_v h_v - \dot{M}_o ut (1 - x) h_a + \dot{M}_{a,in} h_{a,in} + \dot{M}_{v,in} h_{v,in} \tag{2.17}
\]
which now can be combined with the equations for the conservation of mass to get
\[
h_v \left( \dot{M}_f g - \dot{M}_o ut x_v + \dot{M}_{v,in} \right) + h_a \left( - (1 - x_v) \dot{M}_o ut + \dot{M}_{a,in} \right)
+ M_v \frac{dh_v}{dt} + M_a \frac{dh_a}{dt} = \\
\sum \dot{Q} + \dot{M}_f g h_{iv} + \dot{M}_b h_b - \dot{M}_o ut x_v h_v - \dot{M}_o ut (1 - x) h_a
+ \dot{M}_{v,in} h_{v,in} + \dot{M}_{a,in} h_{a,in} + V_g \frac{dp_g}{dt}. \tag{2.18}
\]
This can be simplified to arrive at the equation describing the conservation of energy in the gas space of a volume:

\[
M_a \frac{dh_a}{dt} + M_v \frac{dh_v}{dt} = \sum Q_i + \dot{Q}_{vi} + M_{fg} (h_{vi} - h_v) + M_b (h_b - h_v) + M_{v, in} (h_{v, in} - h_v) + M_{a, in} (h_{a, in} - h_a) + V_g \frac{dp_g}{dt}
\]

where \( \sum Q_i = \dot{Q}_{wall} + \dot{Q}_{in} \) denotes the added heat from the wall and/or other sources. \( \dot{Q}_{vi} \) denotes the heat of evaporation due to condensation or evaporation at the interface with the liquid phase.

### 2.2.3. Equations of State

For the solution of the equations for the gas space, state equations are needed. The air is considered as a perfect gas where steam properties are calculated from steam tables.

\[
h_a = h_a(T_v) = c_{pa}(T_v - T_0) = c_{pa}(T_a - T_0)
\]

\[
h_v = h_v(T_v, p_v)
\]

\[
p_a \nu_a = R_a T_v = R_a T_a
\]

\[
p_g = p_a + p_v.
\]

For implementation details and models used for calculating properties of gas, liquid and steam, see section 2.8.

### 2.2.4. Coupling with Liquid Space

For describing a pressure vessel containing gas and liquid, a coupling of the two phases has to be considered. This coupling is due to two variables: first the pressure, since the pressure has to be equal in both phases. This is one of the very basic assumptions. The second variable is the constant total volume of the pressure vessel. Thus evaporating water results in a decrease of volume of the liquid phase. Therefore the volume of the gas phase has to be increased. In differential form, this can be formulated by

\[
\frac{dV_g}{dt} = \frac{dV_a}{dt} + \frac{dV_v}{dt} = -\frac{dV_l}{dt}.
\]
Using the equations of state for steam and air, and treating the latter as a perfect gas, this leads to

\[
\frac{dV_g}{dt} = \frac{1}{p_a^2} \left( \frac{dM_a}{dt} R_a T p_a + \frac{dT}{dt} M_a p_a R_a - \frac{dp_a}{dt} M_a T R_a \right) +
\]

\[
M_v \left( \frac{\partial v_v}{\partial p_v} \left|_T \right. \frac{dp_v}{dt} + \frac{\partial v_v}{\partial T} \left|_p \right. \frac{dT}{dt} \right) + v_v \frac{dM_v}{dt}.
\]

(2.25)

2.2.5. Complete Model of the Gas Space

The gas space can be described by the following system (2.26) of ODEs. The equations describe the conservation of mass of vapor (a), mass conservation of air (b), energy conservation of gas mixture (c), gas pressure ratio as function of masses of steam and air (d), and the conservation of the volume (e), respectively.

\[
\frac{dM_v}{dt} = \dot{M}_{fg} + \dot{M}_b - \dot{M}_{out} x_v + \dot{M}_{v,in}
\]

(2.26a)

\[
\frac{dM_a}{dt} = -(1 - x_v) \dot{M}_{out} + \dot{M}_{a,in}
\]

(2.26b)

\[
M_a \frac{d h_a}{dt} + M_v \frac{d h_v}{dt} = \sum \dot{Q}_i + \dot{Q}_{vi} + \dot{M}_{fg} (h_{vi} - h_v)
\]

(2.26c)

\[
+ \dot{M}_b (h_b - h_v) + \dot{M}_{v,in} (h_{v,in} - h_v) + \dot{M}_{a,in} (h_{a,in} - h_a) + V_g \frac{dp_g}{dt}
\]

(2.26d)

\[
\frac{M_a M_v \frac{dx_v}{dt}}{(-x_v M_a + x_v M_v - M_v)^2} = \frac{1}{p_a^2} \left( \frac{dp_v}{dt} - \frac{dp_g}{dt} \right)
\]

(2.26e)

\[
\frac{dV_g}{dt} = \frac{1}{p_a^2} \left( \frac{dM_a}{dt} R_a T p_a + \frac{dT}{dt} M_a p_a R_a - \frac{dp_a}{dt} M_a T R_a \right) +
\]

\[
M_v \left( \frac{\partial v_v}{\partial p_v} \left|_T \right. \frac{dp_v}{dt} + \frac{\partial v_v}{\partial T} \left|_p \right. \frac{dT}{dt} \right) + v_v \frac{dM_v}{dt}.
\]

2.3. Liquid Space

For the description of the liquid phase, refer to figure 2.2 for the main variables. Using these variables, equations for the conservation of mass and energy can be formulated. The closure of the system is the conservation of volume of a pressure vessel, as described in section 2.2.
As the liquid space may contain boiling water, the mass in the volume is the sum of liquid water and steam. In the following equations, these masses are referred to as $M_{li}$ for the liquid water in the volume and $M_{ls}$ for the steam. Note that the steam describes only the mass within the volume of the liquid space, not the steam transferred to the gas space. Thus the mass of liquid in this volume is

$$M_l = M_{li} + M_{ls}. \quad (2.27)$$

### 2.3.1. Conservation of Mass

Using the variables in figure 2.2, conservation of mass can be directly derived:

$$\frac{dM_l}{dt} = \dot{M}_{in} - \dot{M}_{lg} - \dot{M}_b - \dot{M}_{out} \quad (2.28)$$

where $\dot{M}_{lg}$ denotes the mass transfer rate at the interface, either condensation or evaporation, $\dot{M}_b$ is the transferred mass by steam bubbles leaving the liquid phase through the interface into the gas space. The modeling of $\dot{M}_{lg}$ and $\dot{M}_b$, will be described in section 2.4 on page 23.

### 2.3.2. Conservation of Energy

For the conservation of energy we start from (2.15) derived earlier:

$$M \frac{dh}{dt} = \dot{Q} + \sum_i \dot{M}_i (h_i - h) + V \frac{dp}{dt}. \quad (2.29)$$

With the appropriate terms for $\dot{Q}$ and $\dot{M}$ ($\dot{Q}_{wall}$, $\dot{Q}_{l-i}$, $\dot{Q}_{in}$, $\dot{M}_{lg}$, $\dot{M}_b$ and $\dot{M}_{in}$), we arrive at the conservation of energy expressed as

$$M_l \frac{dh_l}{dt} = \dot{Q}_{in} + \dot{Q}_{wall} - \dot{Q}_{l-i}$$

$$+ \dot{M}_{in} (h_{in} - h_l) - \dot{M}_{lg} (h_{v,i} - h_l) - \dot{M}_b (h_b - h_l) + V_l \frac{dp}{dt} \quad (2.30)$$

where

$$\dot{Q}_{l-i} = \alpha_{l-i} (T_l - T_i) A_{lv} \quad (2.31)$$

denotes the heat which is transferred from the liquid phase to the interface. The specific enthalpy of the liquid $h_l$ is defined later. The heat
transfer coefficient $\alpha_{l-i}$, the mass flow rates $\dot{M}_{lg}$, $\dot{M}_b$, and the temperature of the interface $T_i$ will be described in section 2.4 on page 23. At saturation, the enthalpy of the rising bubbles $h_b$ has to be at the temperature of the liquid $T_l$. The calculation of the volume of the liquid space has to be done in different ways depending on the presence of one or two phases in the liquid volume.

In the subcooled case, where no formation of steam bubbles is allowed, i.e. no subcooled boiling, the volume can be calculated from the amount of water and the specific volume known from the steam tables. For the calculation of the volume of the boiling liquid space, the steam fraction $x_{lv}$ of the liquid phase has to be introduced:

$$x_{lv} := \frac{M_{lv}}{M_l + M_{lv}}. \tag{2.32}$$

At saturation, due to the presence of saturated steam, both the specific enthalpy and the specific volume will be larger than the enthalpy and specific volume of saturated pure liquid. The average specific volume of the mixture is given by

$$v_l = \frac{V_l}{M_l} = \frac{M_l}{M_l + M_{lv}} v_{l, sat} + \frac{M_{lv}}{M_l + M_{lv}} v_{v, sat}$$

which can be expressed by means of (2.32) as

$$v_l = (1 - x_v)v_{l, sat} + x_v v_{v, sat} = v_{l, sat} + x_v \left( v_{v, sat} - v_{l, sat} \right). \tag{2.34}$$

Similarly, the average specific enthalpy of the mixture is calculated from

$$h_l = (1 - x_v)h_{l, sat} + x_v h_{v, sat} = h_{l, sat} + x_v \left( h_{v, sat} - h_{l, sat} \right). \tag{2.35}$$

Combining (2.34) and (2.35), one can express the average specific volume of the liquid space in the presence of steam as

$$v_l = v_{l, sat} + \frac{h_l - h_{l, sat}}{h_{v, sat} - h_{l, sat}} \left( v_{v, sat} - v_{l, sat} \right). \tag{2.36}$$

Now the volume of the liquid space is expressed by

$$V_l = \begin{cases} M_l v_{l, sat} & \text{if } h_l < h_{sat} \\ M_l \left( v_{l, sat} + \frac{h_l - h_{l, sat}}{h_{v, sat} - h_{l, sat}} \left( v_{v, sat} - v_{l, sat} \right) \right) & \text{if } h_l \geq h_{sat} \end{cases}. \tag{2.37}$$
2.3.3. Change of Volume

The specific volume of the liquid space depends on the number of phases inside. When the liquid is subcooled, only liquid water is present. But as soon as the liquid gets saturated, rising steam bubbles form inside the liquid space and will increase the specific volume of the mixture. Thus, the volume of the liquid space must be a function depending on whether only liquid water, or two phases (liquid water and steam) are present. If saturated, the volume of steam bubbles has to be considered. In general, however, the volume of water in the liquid space is a function of mass, enthalpy, and pressure:

\[ V_l = V_l(M_l, h_l, p). \]  

(2.38)

Single Phase Subcooled Water

For the subcooled single-phase liquid water, the volume is a simple expression as

\[ V_l = M_l v_l(p, h_l) \]  

(2.39)

or in differentiated form

\[ \frac{dV_l}{dt} = v_l \frac{dM_l}{dt} + M_l \left( \frac{\partial v_l}{\partial p} \frac{dp}{dt} + \frac{\partial v_l}{\partial h_l} \frac{dh_l}{dt} \right). \]  

(2.40)

Two Phase Mixture of Saturated Water and Steam

Adding more energy to already saturated water, hence increasing the enthalpy further, the volume will be a more complex function, since bubbles of steam are appearing and thus the volume of water will be increased. For starting the analysis of the change of volume of saturated water, we use

\[ \frac{dV_l}{dt} = \frac{d}{dt}(M_l v_l) = v_l \frac{dM_l}{dt} + M_l \frac{dv_l}{dt}. \]  

(2.41)

The two terms on the right hand side will be discussed separately. The first term results from (2.36):

\[ v_l \frac{dM_l}{dt} = \left( v_{l,sat} + \frac{h_l - h_{l,sat}}{h_{l,sat} - h_{l,sat}} \left( v_{l,sat} - v_{l,sat} \right) \right) \frac{dM_l}{dt}. \]  

(2.42)
We get the second term by differentiating (2.36) with respect to $t$:

$$M_l \frac{dv_l}{dt} = M_l \left( \frac{dv_{l,sat}}{dt} + \frac{d}{dt} \left( \frac{h_l - h_{l,sat}}{h_{v,sat} - h_{l,sat}} \left( v_{l,sat} - v_{l,sat} \right) \right) \right). \tag{2.43}$$

This can be expanded to

$$M_l \frac{dv_l}{dt} = M_l \frac{dv_{l,sat}}{dt} + M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_l}{dt}$$

$$- M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_{l,sat}}{dt}$$

$$- M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_{l,sat}}{dt} \left( h_l - h_{l,sat} \right) \frac{d}{dt} \left( h_{v,sat} - h_{l,sat} \right)$$

$$+ M_l \frac{h_l - h_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dv_{l,sat}}{dt} - M_l \frac{h_l - h_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dv_{l,sat}}{dt}. \tag{2.44}$$

Using (2.35) and expanding again we get

$$M_l \frac{dv_l}{dt} = M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_l}{dt} + M_l \frac{dv_{l,sat}}{dt} + M_l \frac{dv_{l,sat}}{dt}$$

$$- M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_{l,sat}}{dt} - M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dv_{l,sat}}{dt}$$

$$- M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dv_{l,sat}}{dt} + M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dv_{l,sat}}{dt}$$

$$+ M_l \frac{dv_{l,sat}}{dt} - M_l \frac{dv_{l,sat}}{dt}. \tag{2.45}$$

This can be simplified to yield

$$M_l \frac{dv_l}{dt} = M_l \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_l}{dt}$$

$$+ M_l \left( \frac{dv_{l,sat}}{dt} - \frac{v_{l,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_{l,sat}}{dt} \right)$$

$$+ M_l \left( \frac{dv_{v,sat}}{dt} - \frac{v_{v,sat} - v_{l,sat}}{h_{v,sat} - h_{l,sat}} \frac{dh_{v,sat}}{dt} \right). \tag{2.46}$$
In this case water and steam are saturated. Thus the specific volumes depend only on one of the state variables: pressure or temperature. Therefore, we formulate them as depending on the pressure only:

\[
\frac{dV_{l,sat}}{dt} = \frac{dV_{l,sat}}{dp} \frac{dp}{dt},
\]

\[
\frac{dh_{l,sat}}{dt} = \frac{dh_{l,sat}}{dp} \frac{dp}{dt}
\]

which leads finally to

\[
M_{l} \frac{dV_{l}}{dt} = M_{l} \left( v_{l,v,sat} - v_{l,sat} \right) \frac{dh_{l}}{dt} + \left( M_{l} - M_{l,v} \right) \left( \frac{dV_{l,sat}}{dp} - \frac{v_{l,v,sat} - v_{l,sat}}{h_{l,v,sat} - h_{l,sat}} \frac{dh_{l,sat}}{dp} \right) \frac{dp}{dt},
\]

\[
+ M_{l,v} \left( \frac{dV_{l,v,sat}}{dp} - \frac{v_{l,v,sat} - v_{l,sat}}{h_{l,v,sat} - h_{l,sat}} \frac{dh_{l,v,sat}}{dp} \right) \frac{dp}{dt}.
\]

With this result, the derivative of volume in the liquid space can finally be expressed in compact form

\[
\frac{dV_{l}}{dt} = A \frac{dp}{dt} + B \frac{dM_{l}}{dt} + C \frac{dh_{l}}{dt}.
\]

The parameters \( A, B \) and \( C \) depend on the actual state of the volume:

\[
A = \begin{cases} 
M_{l} \frac{dV_{l}}{dp} & \text{if } h_{l} < h_{sat} \\
M_{l} \left( \frac{dV_{l,sat}}{dp} - \frac{v_{l,v,sat} - v_{l,sat}}{h_{l,v,sat} - h_{l,sat}} \frac{dh_{l,sat}}{dp} \right) & \text{if } h_{l} \geq h_{sat}
\end{cases}
\]

\[
B = \begin{cases} 
v_{l} & \text{if } h_{l} < h_{sat} \\
v_{l,sat} + \frac{h_{l} - h_{l,sat}}{h_{l,v,sat} - h_{l,sat}} \left( v_{l,v,sat} - v_{l,sat} \right) & \text{if } h_{l} \geq h_{sat}
\end{cases}
\]

\[
C = \begin{cases} 
M_{l} \frac{dV_{l}}{dh_{l}} & \text{if } h_{l} < h_{sat} \\
v_{l,v,sat} - v_{l,sat} & \text{if } h_{l} \geq h_{sat}
\end{cases}
\]
2.3.4. Complete Model of the Liquid Space

The liquid space, filled either with subcooled water or a mixture of saturated steam and saturated water, can be described by the following system (2.53) of three ODEs: (a) the conservation of mass, (b) the conservation of energy and (c) the conservation of volume.

\[
\frac{dM_l}{dt} = M_{in} - M_{lg} - M_b - M_{out} \quad (a)
\]

\[
M_l \frac{dh_l}{dt} = \dot{Q}_{in} + \dot{Q}_{wall,l} - \dot{Q}_{l-i} + \dot{M}_{in} (h_{in} - h_l) - \dot{M}_{lg} (h_{v,i} - h_l) - M_b (h_b - h_l) + V_l \frac{dp}{dt} \quad (b)
\]

\[
\frac{dV_l}{dt} = A \frac{dp}{dt} + B \frac{dM_l}{dt} + C \frac{dh_l}{dt}. \quad (c)
\]

2.4. Interface

In this module, the evaporation and condensation of water and steam have to be formulated. It has to be considered that the gas may be a mixture of air and steam, pure air or pure steam.

2.4.1. Pure Steam above the Interface

Assuming pure steam in the gas space (either saturated or superheated) of a pressure vessel, the evaporation of water from the liquid space will be at the saturation temperature of the interface which corresponds to the pressure on the liquid. On the other hand, the condensation will take place at the saturation temperature of the interface corresponding to the pressure in the gas space. Due to the assumption of a uniform pressure throughout the vessel, we do not have to distinguish between the pressure in the gas or liquid space. For the temperature \( T_i \) of the interface we write

\[
T_i = T_{sat}(p) \quad (2.54)
\]

which leads to

\[
\dot{Q}_{l-i} = \alpha_{l-i} A_{lg} (T_l - T_i) \quad (2.55)
\]

and

\[
\dot{Q}_{g-i} = \alpha_{g-i} A_{lg} (T_g - T_i). \quad (2.56)
\]
The interface stores neither mass nor energy. Thus, from the quantities in figure 2.3 we get the conservation of energy as

$$M_{fg} \dot{h}_{\text{l, sat}} = \dot{Q}_{\text{l, li}} + \dot{Q}_{\text{g, gi}}$$

in which the heat transfer coefficients $\alpha_{\text{l, li}}$ and $\alpha_{\text{g, gi}}$ have to be determined. This is described later in section 2.4.4.

### 2.4.2. Gas Mixture above the Interface

If a gas mixture with air and steam is present in the gas space, the calculation of the interface temperature is no longer as simple as presented before. Again, we consider the control volume of figure 2.3. For heat transfer, we have

$$\dot{Q}_{\text{l, li}} = \alpha_{\text{l, li}} A_{lg} (T_l - T_i)$$

as well as

$$\dot{Q}_{\text{g, gi}} = \alpha_{\text{g, gi}} A_{lg} (T_g - T_i).$$
Again, when assuming no storage of mass and energy of the interface, the conservation of energy at the interface is given by

$$M_{fg}(h_v - h_l) = Q_{i \rightarrow i} + Q_{g \rightarrow i}.$$ (2.60)

Even though all parameters of the gas and liquid are known, the interface temperature still needs to be determined.

The driving force for evaporation or condensation is the gradient of the partial steam pressure in the gas space. Furthermore, the partial steam pressure at the interface has to correspond to the saturation pressure at the interface temperature:

$$p_{v,i} = p_{sat}(T_i).$$ (2.61)

Following [GWWC95] [Moo90], for the evaporation rate we get

$$\dot{M}_{fg} = \theta A_{lg} \frac{\phi_{v,i} - \phi_v}{1 - \phi_{v,i}} \rho_v.$$ (2.62)

Inserting $p_i = \phi_i p$, results in

$$\dot{M}_{fg} = \theta A_{lg} \frac{p_{v,i} - p_v}{p - p_{v,i}} \rho_v$$ (2.63)

where $p_v$ denotes the partial pressure of the steam in the gas space and $\theta$ is the corresponding mass transfer coefficient.

As (2.60) and (2.63) depend both only on the temperature of the interface, this interface temperature can be found by means of an iterative algorithm. Once the temperature $T_i$ has been found, the mass transfer rate $\dot{M}_{fg}$ can be determined easily.

### 2.4.3. Two-phase Liquid Space

In case of a two-phase mixture in the liquid space, rising gas bubbles have to be considered. These pure steam bubbles are formed due to boiling. They rise up within the liquid phase to be transferred to the gas space. The mass flow rate of these bubbles, $\dot{M}_b$, is determined by the void fraction as well as by the rising velocity of the bubbles. Assuming that the bubbles are uniformly distributed and rise at constant speed, this leads to

$$\dot{M}_b = \varepsilon u_\nu A_{lg} \rho_v$$ (2.64)
where the rising velocity \( u_v \) is considered to be constant. The void fraction \( \varepsilon \) is determined by the mass and volume of the liquid phase by

\[
\varepsilon = \frac{V_{l_v}}{V_l}
\]  
(2.65)

where the volumes are given by

\[
\begin{align*}
M_l &= M_{l_v} + M_{l_i} = \rho_{l_v} V_{l_v} + \rho_{l_i} V_{l_i} \\
V_i &= V_{l_v} + V_{l_i}.
\end{align*}
\]  
(2.66)

Thus the void fraction can be expressed in terms of the variables given in figure 2.2:

\[
\varepsilon = \frac{V_{l_v}}{V_i} = \frac{M_l - \rho_{l_i} V_{l_i}}{\rho_{l_v} - \rho_{l_i}} \frac{1}{V_i}.
\]  
(2.67)

### 2.4.4 Heat and Mass Transfer Coefficients

#### Heat Transfer Coefficients

The heat transfer coefficient at the interface is given by

\[
\alpha = \frac{N \mu}{d}.
\]  
(2.68)

The Nusselt number \( \text{Nu}_h \) for heat transfer problems is given by empirical correlations of the Prandtl number \( \text{Pr} \) and the Grashof number \( \text{Gr} \).

\[
\text{Nu}_h = f(\text{Pr}, \text{Gr}).
\]  
(2.69)

The non-dimensional numbers used here are defined as

\[
\begin{align*}
\text{Pr} &= \frac{c_p \eta}{\lambda} \\
\text{Gr} &= \frac{\rho g l^3 |\rho_i - \rho|}{\eta^2} \\
\text{Sc} &= \frac{\eta}{\rho D_{AB}}
\end{align*}
\]  
(2.70-2.72)

In this work, the binary mass diffusion coefficient for steam and air given in \([m^2/s]\) is calculated from the simple correlation[ONR97]

\[
D_{v,a} = 4.7931 \times 10^{-5} \left( \frac{T^{1.9}}{p} \right)
\]  
(2.73)
which fits tabulated values very close. In this correlation the temperature $T$ is given in degrees K and the pressure has to be given in Pa.

For the Nusselt number, the following correlations [GWWC95] are used:

\[
\text{Nu}_{g \rightarrow i} = 0.21 \text{Gr}^{\frac{1}{6}} \text{Pr}^{\frac{1}{3}} \\
\text{Nu}_{l \rightarrow i} = \max \left( \frac{1}{0.13 \text{Gr}^{\frac{1}{3}} \text{Pr}^{\frac{1}{3}}} \right) \\
\text{Nu}_m = 0.21 \text{Gr}^{\frac{1}{6}} \text{Sc}^{\frac{1}{3}}.
\]

For the gas mixture, the values of the mass properties $\lambda$, $c_p$, $\eta$ and $\rho$ have to be determined. The thermal conductivity is calculated using the Wassiljewa formula [BSL60], [oNRR97]

\[
\lambda_g = \sum_{i=1}^{n} \frac{\lambda_i y_i}{\sum_{j=1}^{n} y_j \Theta_{ij}}
\]

where the molar fractions

\[
y_i = \frac{\dot{m}_i}{M} \frac{M_i}{M}
\]

will be replaced by the partial masses $M_i$:

\[
\lambda_g = \sum_{i=1}^{n} \frac{\lambda_i M_i}{\sum_{j=1}^{n} M_j \frac{\dot{m}_j}{M_j} \Theta_{ij}}.
\]

The dimensionless coefficient $\Theta_{ij}$ is given by

\[
\Theta_{ij} = \frac{1}{\sqrt{8}} \left( 1 + \frac{\dot{m}_i}{\dot{m}_j} \right)^{-\frac{1}{2}} \left( 1 + \left( \frac{\eta_i}{\eta_j} \right)^{\frac{1}{2}} \left( \frac{\dot{m}_j}{\dot{m}_i} \right)^{\frac{1}{4}} \right)^{2}.
\]

The specific heat is given as a linear function weighted by the mass fractions:

\[
c_{pg} = \frac{M_a}{M} c_{pa} + \frac{M_v}{M} c_{pv}.
\]

For the evaluation of the dynamic viscosity of the gas mixture, a similar form to the one used for the heat conductivity is used:

\[
\eta_g = \sum_{i=1}^{n} \frac{\eta_i M_i}{\sum_{j=1}^{n} M_j \frac{\dot{m}_j}{M_j} \Theta_{ij}}
\]
where the dimensionless coefficient $\Theta_{ij}$ remains the same as above.

The density $\rho_g$ of the gas mixture has to be known at the interface ($\rho_{g,i}$) and in the bulk ($\rho_{g,\infty}$) for determining the Grashof number. In the bulk, the density is given by the volume and the mass of the gas space

$$\rho_{g,\infty} = \rho_{v,\infty} + \rho_{a,\infty} = \frac{M_v + M_a}{V_g}.$$  (2.81)

The calculation at the interface is somewhat more complicated. As the interface has to be saturated at the partial steam pressure as mentioned above, the partial steam and air pressures can be expressed as functions of the temperature $T_i$ of the interface.

$$p_{v,i} = p_{sat}(T_i)$$  (2.82)
$$p_{a,i} = p - p_{v,i}.$$  (2.83)

The density of air, considered as a perfect gas, at the interface is given by

$$\rho_{a,i} = \frac{p_{a,i}}{R_m T_i} = \frac{p - p_{v,i}}{R_m T_i}$$  (2.84)

where the density of steam at the interface is obtained from the steam property functions:

$$\rho_{v,i} = \frac{1}{\nu_{v,sat}(p_{v,i})}.$$  (2.85)

Both, the density of air and steam are now functions of temperature only.

**Mass Transfer Coefficients**

The Nusselt number for mass transfer $N_u_m$ is given by an empirical correlation function of the Schmidt number $S_c$ and the Grashof number $G_r$:

$$N_u_m = f(S_c, G_r).$$  (2.86)

As used in [GWWC95] the mass transfer coefficient $\theta$ is expressed by

$$\theta = \frac{N_u_m \phi_g D_{AB}}{l}$$  (2.87)

where the total molar concentration $\phi_g$ is expressed by

$$\phi_g = \frac{\rho_a}{M_a} + \frac{\rho_v}{M_v}.$$  (2.88)
2.4.5. Complete Model for the Interface

The formulation for the temperature of the interface can now be summarized. Once the temperature has been determined, the mass transfer rate can be calculated easily. As mentioned above, the interface temperature cannot be calculated directly, but has to be determined iteratively. (The equation for \( T_i \) cannot be solved for \( T_i \) directly.) The interface temperature can be evaluated from

\[
\frac{\alpha_{l \rightarrow i} A_{lg}(T_i - T_l) + \alpha_{g \rightarrow i} A_{lg}(T_g - T_i)}{h_v - h_l} = \theta A_{lg} \frac{p_{v,i} - p_v}{p - p_{v,i}} \dot{m}_v \tag{2.89}
\]

where the left hand side is derived from equation (2.60) and the right hand side results from equation (2.63). Equation (2.89) usually has exactly one fixed-point for \( T_i \) for which it can be satisfied. See section A.3.4 on page 153 for a detailed discussion about the method used for solving (2.89).

As both sides of equation (2.89) denote the mass transfer rate of the interface, \( \dot{M}_{fg} \) is determined within the same iteration step of the solver algorithm. The heat and mass transfer coefficients have to be evaluated for each iteration with the correlations given above.

In case of two-phase liquid, rising steam bubbles have to be considered which transfer some additional mass from the liquid to the gas space:

\[
\dot{M}_b = \varepsilon_v u_v A_{lg} \rho_v. \tag{2.90}
\]

2.5. The Pipe

A pipe is the most simple connection of two vessels. Within the scope of this work, two different types of pipes are provided: The gas pipe serves for transporting gas, the liquid pipe is used for connecting two liquid spaces, respectively. The mass flow rate through the pipes is modeled the same way. However, the properties of the masses are different. The mass flow rate is driven by the pressure difference from input to output. The geometrical parameters, namely \( L \) and \( D_n \) for the pipe length and its diameter, are represented in figure 2.4. The pressure difference \( \Delta p \) from input to output as the driving force for the mass flow rate \( \dot{M} \) through the pipe can be written as

\[
\Delta p = \sum_j \frac{1}{2} \rho u_j^2 \left( \frac{4 f_j l_j}{D_n j} + k_j \right) + \frac{d\dot{M}}{dt} \sum_j \frac{l_j}{a_j} \tag{2.91}
\]
where \( u \) is the flow velocity, \( L \) and \( D_n \) are the geometrical parameters as mentioned above, \( k \) represents the overall line loss coefficient, \( l \) and \( a \) the length and cross sections of all pipe segments \( j \), and \( f \) is the friction factor given by

\[
f = 0.046 \times Re^{-0.2}
\]  

(2.92)

and \( Re \) represents the Reynold number

\[
Re = \frac{u \rho D_n}{\eta}
\]  

(2.93)

where \( \eta \) is the dynamic viscosity and \( \rho \) the density of the mass flowing through the pipe.

Reducing the model of the pipe to one single pipe segment with constant diameter, and thus with a constant cross section, such that

\[
\sum_j \frac{l_j}{a_j} = \frac{L_i}{A}
\]  

(2.94)

equation 2.91 is simplified to

\[
\Delta p = \frac{1}{2} \rho u^2 \left( \frac{4fL_f}{D_n} + k \right) + \frac{dM}{dt} \frac{L_i}{A}.
\]  

(2.95)

The mass flow rate inside a pipe with a constant cross section \( A \) can be expressed by

\[
\dot{M} = \rho Au
\]  

(2.96)
2.6. The PCCS Condenser Unit

Using equation (2.96) the relation between mass flow rate and flow velocity, equation (2.95) can be written as a function of $\dot{M}$ instead of $u$:

$$\Delta p = \frac{1}{2} \frac{\dot{M}^2}{A^2 \rho} \left( \frac{4 f L f}{D_n} + k \right) + \frac{d \dot{M}}{dt} \frac{L_i}{A}.$$ \hspace{1cm} (2.97)

Equation 2.97 now can be solved for $\frac{d \dot{M}}{dt}$, which represents an ODE for the pipe mass flow rate $\dot{M}$:

$$\frac{d \dot{M}}{dt} = \Delta p \frac{A}{L} - \frac{1}{2} \frac{\dot{M}^2}{L A \rho} \left( \frac{4 f L}{D_n} + k \right).$$ \hspace{1cm} (2.98)

For the gas pipe, the viscosity of the gas mixture has to be calculated again by

$$\eta_g = \sum_{i=1}^{n} \frac{\eta_i M_i}{\sum_{j=1}^{n} M_j \eta_j} \Theta_{ij}$$ \hspace{1cm} (2.99)

as before in equation (2.80) in section 2.4.3 on page 27.

2.6. The PCCS Condenser Unit

The basic schematic of a PCCS condenser unit is shown in figure 2.5. The condenser is connected at the hot side to the drywell as input source. The condensate will fall back to the RPV while the remaining non condensibles (basically air) and non-condensed steam are piped into the liquid space of the suppression chambers, the WW.

As the condensers are passive systems, the mass flow is either pressure-difference driven, or driven by the capacity of the condenser. The driving pressures may vary, depending on the state of the condensers. Basically a condenser can operate in a mode called “open” or “close”. These modes are defined as follows:

- **Open** means that the mass flow rate from the DW to the WW through the condenser is controlled by the pressure difference $\Delta p(DW - WW)$. The water level in the suppression pipe is at the bottom of the pipe. No water column can be built up in that pipe. The remaining non condensed steam will be condensed in the suppression pool of the wetwell. (See left side in figure 2.6.

- **Closed** The internal pressure inside the condenser has dropped below the pressure in the wetwell: The mass flow rate is controlled by the pressure difference $\Delta p(DW - PCC)$. As the pressure inside the condenser is lower than the pressure in the
wetwell a water column will built up in the suppression pipe. No mass can flow from the condenser to the WW, except the condensate which will drain to the RPV. The level of the water column is controlled by the pressures in the PCC and WW.

As no mass can escape from the vent pipe air and non condensed steam will collect inside the condenser tubes. This will lead to a non activated part of the condenser unit.

The switching argument determining the state of the condenser is the pressure difference between the PCC and WW:

\[
\begin{align*}
    p_{PCC} &< p_{WW} \iff \text{state = close} \\
    p_{PCC} &> p_{WW} \iff \text{state = open}
\end{align*}
\]  

(2.100)

2.6.1. PCCS Operation Modes

Open Mode Operation

When the condenser unit operates in open mode, as described above, the mass flow rate is fully controlled by pressure differences. The con-
2.6. The PCCS Condenser Unit

The condensate rate thus can be evaluated solving the initial value problem discussed later.

**Close Mode Operation**

In closed operation, things are more complicated: As gas can no longer exit the suppression pipe in the wetwells, air and non condensed steam collects in the suppression pipe as well as inside the condenser tubes. Thus the active part of the condenser tubes becomes smaller as more non-condensible gas is collected inside the tubes. The mass flow rate will be no longer controlled by the pressure difference of drywell and wetwell, but by the condensation rate of the active part of the condenser. For a detailed view of this mode of operation, refer to figure 2.7.

The suppression pipe from the lower drum to the wetwell is closed due to water rising inside the pipe. The level is controlled by the pressure difference of DW and WW. As the air inside the pipe is assumed to be non-compressible gas and condensation inside the pipe is neglected here, the volume (mass) of the gas controls the lower level of the active condenser part, called *condensation boundary, cb*.

The ODEs to be integrated are now no longer an *initial value problem* but a *boundary value problem*. The boundary conditions are given at the top and at the end of the active part:
• At the top the conditions come from the drywell. Temperature, partial steam, and air pressures are given.

• At the lower end, the temperature of the remaining gas, the temperature of the condensate and the pool temperature are all the same: $T_{pool}$. As all steam has been condensed here, the steam partial pressure is approximately the saturation pressure at the pool temperature.

From these boundary conditions, the mass flow rate will be calculated. In the scope of this work, the closed mode operation is discussed only roughly and the numerical implementation of the closed mode operation is neglected. However, the next section gives a general discussion of the condenser logic, including open and closed mode operation.

**Logic of Condenser Operation**

It is not obviously clear, whether the condenser operates in open mode or in closed mode. Thus some more discussion of the logic of the operation is needed.
First of all, the pressure difference between the drywell and wetwell decides whether the condenser operates in open or in closed mode.

\[
P_{DW} = P_{WW}?
\]

If the pressure in the drywell is larger than the pressure in the wetwell plus the hydrostatic pressure corresponding to the submergence of the vent, the system operates in open mode. This means, no non-condensibles are left in the condenser pipes, thus the condenser mass flow can be calculated in two ways, either by pressure difference between drywell and wetwell, which gives the flow \( M_{\text{feed1}} \), or by the condensing capacity of the condensing pipe \( M_{\text{feed2}} \), respectively. The larger flow rate should be the relevant one for the feed flow rate. In the second case, the length of the condensing tubes is the full tube length.

\[
\begin{align*}
    l_c &= l_{c0} \\
    M_{\text{feed1}} &= f(\Delta p) \\
    M_{\text{feed2}} &= f(l_c)
\end{align*}
\]

In closed mode operation, however, the amount of non-condensibles inside the condensing pipe is very important. The volume of the gas gives the effective active length of the condensing tubes and has to be evaluated. Furthermore, we have to check, that the condensing pipes are not filled with gas, otherwise the feed flow rate has to be set to exactly 0 kilograms per second.
The larger feed flow rate should be taken into account. This leads usually to a complete, i.e. 100% condensation rate in most of the cases, except when the pressure difference is huge and the condensers are under dimensioned.

The effective mass flow rate is the maximum of the two calculated flow rates \( \dot{M}_{\text{feed1}} \) and \( \dot{M}_{\text{feed2}} \), respectively. From this, the condensate flow rate \( \dot{M}_{\text{cond}} \) and the non-condensibles flow rate \( \dot{M}_{\overline{\text{cond}}} \) are calculated.

Finally, the level of the water column inside the pipe that connects
the condenser with the wetwell has to be adjusted depending on the pressure difference between drywell and wetwell if in closed mode operation. Otherwise the amount of non-condensibles inside the condensing tubes is set to 0.

The overall logic of the condenser unit and the calculation of its mass flow feed rate are given in figure 2.8.

2.6.2. The Condenser Tube

The main element of each condenser is one of its several tubes. In this work, all effects of multiple parallel tubes like temperature variations inside the PCCS pool are neglected.

The following derivations are always made for one single tube of the condenser. The driving force for heat transfer is the difference of the coolant temperature $T_c$ and the temperature of the gas $T_h$ flowing through the tube. This heat transfer has to be in equilibrium with the energy conservation of the unit. In the actual PCCS condensers there are 20 tubes in one unit. We can model the heat transfer per unit length $q'(z)$ of one side of a condenser tube (see figure 2.9) by the following three equations. The first equation describes the heat transfer in the liquid film inside the condenser. This film will grow with increasing depth $z$. The second one describes the heat transfer inside the tube wall, while the third equation is the heat transfer from the outside
38 Chapter 2: Physical Description of Modules

Figure 2.8: Overall logic of the condenser unit
Since we assume quasi steady state operation of the condenser, and the fact that the condenser tube can neither store mass nor energy, all the $q'(z)$ must be equal. Therefore, we will combine the above three equations in such a way that the intermediate temperatures $T_{w,c}$ and $T_{w,h}$ will cancel. For this reason, equations (2.102) and (2.103) are solved for $T_{w,h}$ and $T_{w,c}$, respectively:

\begin{align}
T_{w,h}(z) &= q'(z) \frac{\ln \frac{D_0}{D_i}}{2\pi \lambda_w} + T_{w,c}(z) \tag{2.104} \\
T_{w,c}(z) &= q'(z) \frac{1}{\alpha_c(z) D_0 \pi} + T_c(z). \tag{2.105}
\end{align}
Inserting (2.105) in (2.104) gives

\[
T_w(z) = q'(z) - \ln \frac{D_0}{D_i} \frac{1}{2\pi \lambda_{w}} + q'(z) \frac{1}{\alpha_c(z) D_o \pi} + T_c(z) \quad (2.106)
\]

which now can be inserted into (2.101) to obtain an expression in \( T_c \) and \( T_h \) as required:

\[
q'(z) = \frac{T_h(z) - T_c(z)}{\frac{1}{\alpha_h(z) D_o \pi} + \frac{\ln \frac{D_0}{D_i} \frac{1}{2\pi \lambda_{w}}}{} + \frac{1}{\alpha_c(z) D_o \pi}}. \quad (2.107)
\]

The heat transfer coefficients \( \alpha_i \) depend on temperatures, geometry and mass flow rates. From these quantities, the heat transfer coefficients are calculated by means of empirical correlations described later on in this chapter.

The pool temperature \( T_c \) is given from the state of the pool while the temperature \( T_h \) of the gas has to be determined from the assumption of saturated steam at location \( z \):

\[
T_h = T_{sat}(p_v(z)). \quad (2.108)
\]

The local steam partial pressure \( p_v(z) \) is a function of mass flow rates of air and steam, respectively:

\[
p_v(z) = \frac{\frac{M_v(z)}{\partial M_v}}{\frac{M_v(z)}{\partial M_v} + \frac{M_a(z)}{\partial M_a}} p_h \quad (2.109)
\]

thus the gas temperature is given as a function of the local steam flow rate \( M_v \) and the overall constant air flow rate by

\[
T_h = T_{sat} \left( \frac{\frac{M_v(z)}{\partial M_v}}{\frac{M_v(z)}{\partial M_v} + \frac{M_a(z)}{\partial M_a}} p_h \right). \quad (2.110)
\]

The total pressure is assumed to be equal to the drywell pressure, as pressure drops inside the condenser are marginal.

The heating power per unit length \( q' \) as given in (2.107) has to be in equilibrium with the energy balance in a control volume as shown in figure 2.9. Taking the values for mass flow rates and enthalpies at
2.6. The PCCS Condenser Unit

\[ z - \Delta z \text{ and at } z, \text{ the energy balance can be written as follows:} \]
\[
\Delta q = \dot{M}_v(z) h_{v,\text{sat}}(p_v(z)) - \dot{M}_v(z) h_{v,\text{sat}}(p_v(z))
+ \dot{M}_c(z) h_1(T_c(z - \Delta z), p_h) - \dot{M}_c(z) h_1(T_c(z), p_h)
+ \dot{M}_a c_p T_h(z - \Delta z) - \dot{M}_a c_p T_h(z). \quad (2.111)
\]

To get a differential equation which can be integrated with state-of-the-art integrators, the control volume size of \( \Delta z \) has to be reduced to infinitesimal size; the above equation then can be written as
\[
\frac{dq}{dz} = d (\dot{M}_v(z) h_{v,\text{sat}}(p_v(z)))
+ d (\dot{M}_c(z) h_1(T_c(z), p_h))
+ d (\dot{M}_a c_p T_h(z)). \quad (2.112)
\]

Using the mass continuity
\[
\dot{M}_v + \dot{M}_c = \text{const} \quad (2.113)
\]
we get an ODE which can be readily integrated:
\[
dq(z) = q'(z)dz \quad (2.114)
\]
or in normal form
\[
\frac{dq(z)}{dz} = q'(z) \quad (2.115)
\]
which describes an ODE in \( \dot{M}_v(z) \).

2.6.3. Heat Transfer Coefficients

The heat transfer coefficients \( \alpha_h \) and \( \alpha_c \) are to be found using empirical correlations. In this module the condensing heat transfer coefficients are determined using a correlation proposed by Vierow and Schrock [VS91] for the inside condensing heat transfer coefficient, and the Chen correlation [BCD+81] for the outside saturated forced-convective-boiling heat transfer coefficients, respectively.

The Vierow Correlation

The Vierow correlation is the adaption of the Nusselt condensate film theory for the case of a steam-nitrogen mixture flowing down inside a tube. From a series of measurements, a correction factor \( f \) has been
derived which accounts for the influence of the non-condensible gas and for shear forces acting on the liquid film.

Inside the condensing tube a laminar film of down-flowing liquid will be built up. The temperature of the interface from liquid to gas will be at saturation for the given partial steam pressure. The heat transfer is given by the thermal conductivity and the thickness of the liquid film:

$$\alpha_{Nu} = \frac{\lambda_l}{\delta}$$  (2.116)

where the film thickness $\delta$ can be expressed from the momentum conservation equation for a film falling on a vertical wall by

$$\delta = \left(\frac{3\eta_l \Gamma}{8\rho_l (\rho_l - \rho_g)}\right)^\frac{1}{3}$$  (2.117)

where $\Gamma$ denotes the mass flow rate per unit width. Given the condensate mass flow rate, $\dot{M}_c$, and the geometry of the tube, we get

$$\Gamma = \frac{\dot{M}_c}{\pi D_i}$$  (2.118)

where $D_i$ denotes the inner diameter of the tube.

Vierow and Schrock found a ‘degradation factor’ $f$ which should correct for the presence of the non-condensibles and for the interface shear. $f$ is a function of the Reynolds number and the nitrogen mass fraction of the gas mixture. The Reynolds number is given by

$$Re_g = \frac{\rho_g u_g D_i}{\eta_g} = \frac{(\dot{M}_v + \dot{M}_n) D_i}{(D_i - 2\delta)^2 \frac{\pi}{4} \eta_g}$$  (2.119)

where the index $n$ indicates the non-condensibles (nitrogen). The determination of the viscosity of the mixture follows the procedure used at the interface presented in section 2.4.

The non-condensible mass fraction is given by

$$M_a = \frac{\dot{M}_n}{\dot{M}_n + \dot{M}_v}.$$  (2.120)

In [VS91] a term $CM^b_a$ is defined as function of the non condensible
mass fraction

\[
CM_a^b := \begin{cases} 
10M_a & \text{if } M_a \leq 0.063 \\
0.938M_a^{0.13} & \text{if } 0.063 < M_a \leq 0.6 \\
M_a^{0.22} & \text{if } 0.6 < M_a
\end{cases}
\]  
(2.121)

This allows to express a ‘degradation factor’ as

\[
f = f_1 f_2 = \left(1 + 2.88 \times 10^{-5} \text{Re}^{1.18} \right) \left(1 - CM_a^b\right)
\]  
(2.122)

where the first factor \((f_1)\) should account for interfacial shear forces and the second one \((f_2)\) should correct the presence of non-condensibles (nitrogen).

With this factor, the corrected heat transfer coefficient within the tube is given by

\[
\alpha_h = f \alpha_{Nu}.
\]  
(2.123)

The outer heat transfer coefficient \(\alpha_c\) can be calculated using the Chen correlation [BCD^+81], used in [MMLG98]. But due to the nature of the problems calculated within this work, the outer heat transfer coefficient will be assumed as a constant value.

### 2.6.4. Condensation in Suppression Pools

When a steam-air mixture is injected through the suppression pipe below the suppression pool surface a part of the steam will condense in the pool water. A complete disappearance of the bubbles will not occur due to the presence of air. For the largest possible condensation, a saturated mixture of steam and air at \(T_{pool}\) will escape to gas space of the wetwell. The gas escaping to the wetwell will have a partial steam pressure of \(p_v = p_{sat}(T_{pool})\), plus an air partial pressure \(p_a\). The nature of this process is well described in [MMLG98]. In the same paper, the authors present an experimental study of large steam-air bubbles, condensing inside a suppression pool. The experimental view will not be discussed here, but it confirms the underlying theory that follows.

The situation is depicted in figure 2.10. As shown, the aim is to bring the mixture injected at \(T_{in}\) with a volumetric flow rate

\[
\dot{V} = \dot{V}_v + \dot{V}_a
\]  
(2.124)
Figure 2.10: Steam partial pressure and volumetric flow rates vs. temperature in the suppression pool

and the corresponding steam fraction

\[ x_v = \frac{\dot{V}_v}{\dot{V}} = \frac{p_v}{p} \]  \hspace{1cm} (2.125)

down to equilibrium conditions at pool temperature \( T_{pool} \) and saturation pressure \( p_{sat}(T_{pool}) \), with the flow rate

\[ \dot{V}_{eq} = \dot{V}_{v,eq} + \dot{V}_{a,eq} \]  \hspace{1cm} (2.126)

and the steam fraction

\[ x_{v,eq} = \frac{\dot{V}_{v,eq}}{\dot{V}_{eq}} = \frac{p_{sat}(T_{pool})}{p} \]  \hspace{1cm} (2.127)

Again, we assume the pool to be well mixed, and a constant pressure at the suppression pipe exit, and the gas exiting the suppression pipe to be an ideal gas.

Assuming no air absorption in the liquid, the air mass flow rate will remain constant, i.e. all air entering the suppression pool will escape (at lower temperature) to the gas space. The volumetric air flow
rate will change slightly due to cooling:
\[
\hat{V}_{a,eq} = \hat{V}_a \frac{T_{pool}}{T_{in}} = \frac{\dot{M}_a}{v_v(p_u, T_{pool})}
\] (2.128)

Now, starting from equations (2.127) and (2.128), respectively, the volumetric air and steam flow rates can be calculated from the input mass flow rates.

The heat of condensation, calculated from the differences between input gas and output gas and liquid, will be added to the liquid space of a suppression pool.

\[
Q_{out} = Q_{in} - Q_{out}
\] (2.129)

where
\[
Q_{in} = \dot{M}_{v, in} h_{v, in} + \dot{M}_{a, in} h_{a, in}
\] (2.130)

and
\[
Q_{out} = \dot{M}_{v, out} h_{v, out} + \dot{M}_{a, out} h_{a, out} + \dot{M}_{l, out} h_{l, out}
\] (2.131)

respectively.

Thus, during condensation with a mass flow into the suppression pool, the temperature of the corresponding suppression pool is expected to increase.

### 2.6.5. The Condenser Unit

The condenser unit can be considered as a composition of a number of condenser tubes which connect two volumes (see figure 2.11). The upper header volume at the inlet is a gas volume, while the lower header volume is a two phase volume which collects the gas (non-condensed steam as well as the non-condensibles) and the liquid at the outlet of the condenser tubes. This lower volume has two connections: one for the gas outlet and one for the liquid outlet. Usually, air is expected to accumulate in the gas part of this volume.

As the lower header volume is composed of given modules (one gas space, one pressure vessel and a number of parallel condenser pipes), it seems reasonable to represent it using the available standard modules. The same way, a module for the pressure vessel, as used for the RPV, has been composed consisting of a liquid space, a gas space and an interface in between.
When modeling the PCCS in the way described above, some numerical problems appeared due to the enormous difference in volumes. The upper and lower drums of the PCCS condenser are of the order of $10^{-1} \ m^3$ where the drywells and wetwells are of the order of $10^2 \ m^3$. Thus the drums will be neglected in the actual implementation of condensers in the transients dealing with whole systems, but can be included for systems featuring only condensers for test purposes.

The total mass flow through the condenser is calculated using the model of a gas pipe described in section 2.5. This mass flow is the fraction of the total mass flow corresponding to the number of parallel condenser tubes. In this model, a homogeneous distribution over all tubes is considered.

The condensate output of the condenser will be drained to a liquid volume, as it might be the liquid phase of the RPV or the suppression pool, as seen in figure 2.1, while the gas output usually gets connected to a suppression pool of a wetwell. The connection ends well below the water surface of a suppression pool, where most of the remaining steam will be condensed, as stated above.
2.7. Vent Line

The vent line connects the drywell to a wetwell, where the exit of the vent line is well below the water surface of the wetwell. The vent line opens only when the pressure in the drywell exceeds the pressure in the wetwell by the hydrostatic pressure of the amount of water above the vent line outlet. The theory is not very difficult. Basically, it is the combination of a gas pipe for calculating the maximum flow rate, and the condensation in a suppression pool, as discussed earlier.

2.8. Properties of Steam, Air, and Water

The subroutines used for the properties of steam, air and water are based on the 1967 IFC formulation for industrial use [Com67] and other IAPWS releases. The code is distributed in Fortran source code by the American Society of Mechanical Engineers (ASME) [MMSs92].

During the use of these subroutines an unexpected behavior of some of the functions has been observed. Under certain circumstances, the step size of the integrator is influenced in an unfavorable way. Another disadvantage is the fact that partial derivatives are not available from these subroutines; in a first approach, these are calculated using a very simple linear differences quotient.

In some cases, the passing of parameters is not very efficient because of encapsulating routines which only convert units from SI units to US units.

A promising alternative would have been to use the PROST package [Bau98]. This should be evaluated in the future even if the package is written in C and the passing of parameters from the Fortran routines is not straightforward. The disadvantage of this solution is the loss of system and compiler independent source code.
Simulating a continuous system as a thermal hydraulic setup in this work requires robust and fast numerical and analytical methods for a variety of tasks. These include the numerical integration of a system of differential equations, the numerical solution of systems of non linear algebraic equations, function evaluations and table-lookups, etc. On the other hand, a special purpose software tool is required for doing certain laborious formula manipulations for the model equations while translating the resulting equations into a proper computer code. This code must be in a language which can be compiled, linked and executed on the target system.

This chapter describes the basic numerical techniques used during this work. The target system is defined with all hard- and software requirements. The software components are analyzed in detail:

First, *Maple* is used as a computer algebra tool for doing formula manipulation tasks. Then some standard *POSIX*\(^1\) compliant filters and

---

\(^1\)Acronym for portable operating system interface for computer environments. A
Editors are used to modify these Maple solutions and for converting them into proper Fortran routines. Under control of a new compiler, these routines are processed and combined to a Fortran code describing the overall system. This code needs to be integrated by a stable and efficient numerical integrator.

3.1. The Target System

The target system has been defined regarding some important issues of the code being developed. These include good portability from one platform to another one, the availability of a stable, widely accepted operating system, reliable compilers and of application software and languages such as Fortran, Maple, etc. The system must allow possibilities for parallelization as this is getting increasing importance on newer platforms. We found Unix environments as the most attractive way fulfilling the requirements mentioned above and especially while also spanning the way from PCs to workstations up to supercomputers. There, a parallelization can be implemented using PVM\(^2\), the parallel virtual machine on one hand or using a real multiprocessor machine on the other hand. This leads to the following definition of software:

- Unix based operating system,
- Fortran Compiler,
- Maple,
- POSIX filters and editors are included in Unix standard systems,
- Optionally PVM.

Federal Information Processing Standard Publication (FIPS PUB 151-1) for a vendor-independent interface between an operating system and an application program, including operating system interfaces and source code functions. Note: IEEE Standard 1003.1-1988 was adopted by reference and published as FIPS PUB 151-1.

\(^2\)PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. The individual computers may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations, that may be interconnected by a variety of networks, such as Ethernet, FDDI. User programs written in C, C++ or Fortran access PVM through library routines. Daemon programs provide communication and process control between computers.
3.2. Numerical Integration

One alternative is the operating system Linux which got increasingly common since recent years. It turned out that Linux\(^3\), after having performed for years extremely well on Intel based PC platforms and having gained the attention of leading IT companies such as IBM, Oracle, HP, etc. is becoming one of the leading operating systems for commercial servers.

3.2. Numerical Integration

The problem of solving ordinary differential equations can always be reduced to finding the solution of a system of first order ODEs by an adequate solver. These systems can be written in the general form

\[
\frac{dy_i(t)}{dt} = f_i(t, y_1, \ldots, y_n), \quad (i = 1, \ldots, N), \quad (t_0 < t < t_f), \quad (3.1)
\]

where the functions \(f_i\) depend on the independent variable \(t\) and the dependent state variables \(y_1, \ldots, y_n\) must be known.

The system of equations given above does not specify a problem completely. Since differential equations have an infinite set of solutions, one needs to fix the desired solution at some point in the \((N+1)\)-dimensional time-state space. This is done by algebraic conditions called boundary conditions.

Boundary conditions can be divided into two broad categories:

- Initial value problems, where all states \(y_i\) are given at some starting time \(t_0\). The problem is to find the \(y_i\) for some discrete times \(t_j\), or at some final time \(t_f\).

- Boundary value problems, e.g. in their most common form as two-point boundary value problems, on the other hand, specify the given values at more than one instant of time \(t\). Typically, some conditions are given at a starting point \(t_0\) and the other values are given at the final point \(t_f\), so that all in all \(N\) conditions are given for solving the system 3.1.

In the context of this work, only initial value problems are needed, as all state variables are known at the beginning of an experiment.

---

\(^3\)Available on www.linux.org under the GNU Public License (GPL). This includes the right of copy, modification and redistribution of code. Code under GPL include Linux, the g77 Fortran compiler, the C compiler gcc and lots of other software. See appendix F for the full text of the GPL.
or simulation run. As the underlying concepts of all integration algorithms are largely identical, the different numerical packages specialize in a broad way. In general, all solvers provide the solution only for discrete values of the independent variable $t$. In this, the size $h$ of the integration steps depend on the order of the integrator and during integration $h$ is adapted in such a way, that a required accuracy can be achieved.

For this, some methods have been developed, such as the Runge-Kutta[HLR89], the Bulirsch-Stoer[SB02] or the Predictor-Corrector [KMN89] methods.

If a system consists of more than one first-order differential equation the problem of stiffness possibly arises. Stiffness occurs in problems, where physical phenomena are simultaneously taking place on a superimposed way but with different speeds of variation. An example would be a device where the reaction of two chemical substances is taking place. One reaction may go on almost instantaneously whereas the other one is taking a considerable larger amount of time for showing any noticeable changes of concentrations.

A more general example is the system

$$
\begin{align*}
\dot{u} &= 998u + 1998v \quad u(0) = 1 \\
\dot{v} &= -999u - 1999v \quad v(0) = 0
\end{align*}
$$

or in matrix notation

$$
\begin{pmatrix}
\dot{u} \\
\dot{v}
\end{pmatrix} = \mathbf{A}
\begin{pmatrix}
u \\
v
\end{pmatrix}; \quad u(0) = 1 \\
v(0) = 0
$$

where

$$
\mathbf{A} = \begin{bmatrix}
998 & 1998 \\
-999 & -1999
\end{bmatrix}
$$

has the analytical solutions

$$
\begin{align*}
u &= 2e^{-t} - e^{-1000t} \\
v &= e^{-t} + e^{-1000t}
\end{align*}
$$

where the time constants $\lambda_1 = -1$ and $\lambda_2 = -1000$ are the eigenvalues of $\mathbf{A}$. Integrating this system with any ordinary (non stiff) integrator, a step size of $h \leq c/\max(|\lambda_i|) = c/1000$ where the constant $c$ depends on the algorithm, e.g. $c = 2$ for the first order forward Euler method and $c = 2.785$ for 4th order Runge Kutta method, would be required.
to get a stable solution. In order to do a stable integration, this stability requirement imposes a severe limitation of the step size \( h \), even tough the \( e^{-1000t} \) term is completely negligible in determining \( u \) and \( v \) after some distance from the beginning.

In the following sections 3.2.2 and 3.2.3, some detailed discussion about integration of stiff and nonstiff systems can be found. In section 3.2.4 we will discuss the handling of discontinuities in input functions as well as in the results. Finally, in section E.1 we discuss one of the advanced software packages for the integration of ODEs.

### 3.2.1. Stability of Algorithms

A very important feature of every integration algorithm is its stability property. To analyze an algorithms stability, it is applied to a test system

\[
\frac{d\hat{y}}{dt} = -A(t)\hat{y} + b(t) \tag{3.6}
\]

Neglecting the time dependency of the matrix \( A \) and supposing that \( A \) is a \( N \times N \) real matrix and \( b \) is an \( N \)-dimensional real column vector of the special form \( b = 0 \), then the \( l \)-th component of the solution \( \hat{y} \) reads

\[
\hat{y}_l = -\sum_{m=1}^{N} \alpha_{l,m} \exp(-\lambda_m t), \quad l = 1, 2, \ldots, N \tag{3.7}
\]

where \( \lambda_m \) is the \( m \)-th eigenvalue of \( A \) while \( \alpha_{l,m} \) is the \( l \)-th component of the the \( m \)-th eigenvector. Therefore, it is enough to consider the following test equation

\[
\hat{y} = -\lambda y, \quad y(0) = y_0 \tag{3.8}
\]

with the analytical solution \( y(t) = y_0 \exp(-\lambda t) \).

When doing the numerical integration, e.g. with a constant step-size \( h \) it is convenient to write

\[
t_i = ih \quad (i = 0, 1, 2, \ldots)
\]

\[
y_i = y(t_i)
\]

\[
f_i = f(t_i, y(t_i))
\]

An algorithm is called *numerically stable*, if it reflects the behavior of the exact solution

\[
y_{i+1} \rightarrow 0, \quad \text{as} \quad i \rightarrow \infty \tag{3.9}
\]
or in general, if the growth factor \( P(\sigma) = P(h\lambda) \) is limited by

\[
|P(\sigma)| < 1
\]  

(3.10)

where

\[
y_{i+1} = P(\sigma)y_i \quad \text{and} \quad \sigma = \Re(\sigma) + i\Im(\sigma) = h (\Re(\lambda) + i\Im(\lambda)) = h\lambda
\]

The symbols \( \Re(\sigma) \) and \( \Im(\sigma) \) denote the real and the imaginary part of the value \( \sigma \), respectively.

**Stability of Forward Euler Algorithm**

The forward Euler algorithm is well known; applied to the above discussed test equation 3.8 one gets

\[
y_{i+1} = y_i + hf_i \]

\[
= (1 - h\lambda)y_i
\]

\[
= (1 - h\lambda)^{i+1}y_0
\]

(3.11)

To fulfill the stability criterion 3.9, the step size must be adapted, as

\[
|1 - h\lambda| < 1
\]

(3.12)

thus if \( \lambda \) is real and positive, \( h \) is given by

\[
h < \frac{2}{\lambda}
\]

(3.13)

Note that when integrating a system of ODEs by means of the forward Euler algorithm equation 3.13 needs to be replaced by

\[
h < \frac{2}{|\lambda_{\text{max}}|}
\]

(3.14)

where \( \lambda_{\text{max}} \) is the largest eigenvalue. The growth factor for forward Euler is given by

\[
P(\sigma) = 1 - \sigma
\]

(3.15)

and leads to the region of stability shown in figure 3.1.
3.2. Numerical Integration

Stability of Trapezoidal Rule (Heun’s Algorithm)

The Trapezoidal rule is a second order algorithm given by

\[ y_{i+1} = y_i + \frac{1}{2}h(f_i + f_{i+1}) \]  
\[ (3.16) \]

Also note that this algorithm is implicit with respect to \( y_{i+1} \) as

\[ y_{i+1} = y_i + \frac{1}{2} (f(t_i, y_i) + f(t_{i+1}, y_{i+1}) \]  
\[ (3.17) \]

This leads for the test equation to

\[ y_{i+1} = y_i + \frac{1}{2} h (-\lambda y_i - \lambda y_{i+1}) \]
\[ = \frac{1 - \frac{1}{2} h\lambda}{1 + \frac{1}{2} h\lambda} y_i \]  
\[ (3.18) \]
\[ = \left( \frac{1 - \frac{1}{2} h\lambda}{1 + \frac{1}{2} h\lambda} \right)^{i+1} y_0 \]

and the growth factor \( P(\sigma) \) for the trapezoidal rule

\[ P(\sigma) = \frac{1 - \frac{1}{2} \sigma}{1 + \frac{1}{2} \sigma} \]  
\[ (3.19) \]

which turns out to be stable for all eigenvalues \( \lambda \) with a positive real part (see figure 3.2).
Note that the step size $h$ of the trapezoidal rule can be chosen any length, without loosing stability, as long as the real part of the eigenvalues of the system are positive. The step size is then limited by the required accuracy. Thus, do not confuse stability with accuracy!

3.2.2. Integration of Non-stiff ODEs

For integrating linear systems of non-stiff ODEs many methods have been developed. From the most simple Euler algorithm up to advanced multi step methods can be stable for some step size $h$, provided $|P(\sigma)| \leq 1$ holds. From the previous remarks it follows that implicit algorithms are featuring a more suitable stability domain than explicit algorithms.

Therefore, for integrating large systems of ODEs, advanced implicit multi step methods with variable order and variable step size are needed in order to keep computation times within reasonable bounds [SG75], [Gea71].

Single step methods are using only information of the last step during the integration process. In contrast, multi step methods are trying to use some more 'historical' information about the last few steps of the integration. Thus, a somewhat more advanced housekeeping of data is required. In the special case of linear multi step methods, all the mesh points are equidistant. If the information is not available, it must be approximated by some interpolation method.

Basic one step methods for integrating systems of non-stiff ODEs
3.2. Numerical Integration

are:

- Euler
- Runge-Kutta
- Bulirsch-Stoer
- Predictor-Corrector

and as a Multi step method,

- Adams-Moulton
- Gear

should be considered.

As an example for a single step method the well known Runge-Kutta algorithm is presented [HNW87].

For a given system of ODEs,

$$\dot{y}(t) = F(t, y(t)); \quad y(0) = y_0 \quad (3.20)$$

where \( y, \dot{y} \) and \( F \) are \( N \)-dimensional the integration of the system of ODEs over the first time step \( t_0 \leq t \leq t_1 = t_0 + h \) is readily performed by applying

- \( k_1 = h \, F(t_i, y_i) \)
- \( k_2 = h \, F \left( t_i + \frac{h}{2}, y_i + \frac{k_1}{2} \right) \)
- \( k_3 = h \, F \left( t_i + \frac{h}{2}, y_i + \frac{k_2}{2} \right) \)
- \( k_4 = h \, F \left( t_i + h, y_i + k_3 \right) \)

$$y_{i+1} = y_i + \frac{1}{6} \left( k_1 + 2k_2 + 2k_3 + k_4 \right) \quad (3.21)$$

where \( k_1, k_2, k_3 \) and \( k_4 \) are \( N \)-dimensional vectors.

This yields the state vector \( y_1 \) at location \( t_1 \). Note that the algorithm is explicit, i.e. all information on the right-hand side is available from previous computations. Of course this algorithm can be applied using variable step sizes.

The main disadvantage of explicit algorithms is that their stability is only guaranteed if their step size \( h \) is limited such that

$$h \leq \frac{c}{|\lambda_{\text{max}}|} \quad (3.22)$$
as mentioned before. For the presented Runge-Kutta 4,4 the value of the constant \( c = 2.785 \).

Given the system of ODEs

\[
\dot{y} = F(t, y)
\]  (3.23)

as example,

\[
\begin{pmatrix}
\dot{y}_1 \\
\dot{y}_2 \\
\dot{y}_3
\end{pmatrix}
= \begin{pmatrix}
3y_1 - y_2y_3 \\
y_3 t + e^{-y_1} \\
y_1 \sin y_2 - y_3^2
\end{pmatrix}
\]  (3.24)

the Jacobian \( J \) defined by

\[
J = \begin{pmatrix}
\frac{\partial F_1}{\partial y_1} & \cdots & \frac{\partial F_1}{\partial y_N} \\
\vdots & \ddots & \vdots \\
\frac{\partial F_N}{\partial y_1} & \cdots & \frac{\partial F_N}{\partial y_N}
\end{pmatrix}
\]  (3.25)

is calculated as

\[
J = \begin{pmatrix}
3 & -y_3 & -y_2 \\
-e^{-y_1} & 0 & t \\
\sin y_2 & y_1 \cos y_2 & -2y_3
\end{pmatrix}
\]  (3.26)

Now, the eigenvalues can be evaluated by solving

\[
\det (J - \Lambda) = 0
\]  (3.27)

where

\[
\Lambda = \begin{pmatrix}
\lambda_1 & 0 \\
& \ddots \\
0 & \cdots & \lambda_N
\end{pmatrix}
\]  (3.28)

Thus, only terms in \( F \) featuring state variables appearing in linear form with constant coefficients, e.g. the term \( 3y_1 \), contribute to constant elements of the Jacobian. Terms with nonlinear relations of state variables, such as \( -y_2y_3 \) or \( e^{-y_1} \), as well as linear terms with state variables in linear form but time dependent coefficients result in nonlinear entries in the Jacobian.

It is obvious that a single non constant element of the Jacobian causes time dependent eigenvalues, and consequently time dependent integration step sizes so as to perform an integration without violating the stability requirement (3.22). See [Gea71] and [HNW87] for further discussion on integrating ODEs.
3.2.3. Integration of Stiff ODEs

The simplest implicit algorithm suited for integrating stiff systems is known as Heun’s method. The integration of the ODE

\[ \dot{y}(t) = F(t, y(t)); \quad y(0) = y_0 \]  

yields:

\[ \int_{t_i}^{t_{i+1}} \dot{y}(t) dt = \int_{t_i}^{t_{i+1}} F(t, y(t)) dt \]  

While the left-hand side can be integrated analytically the integral on the right-hand side is approximated by the trapezoidal method:

\[ y_{i+1} - y_i = h \frac{F(t_i, y_i) + F(t_{i+1}, y_{i+1})}{2} \]  

where \( h = t_{i+1} - t_i \), \( y_{i+1} = y(t_{i+1}) \), and \( y_i = y(t_i) \). Note that the new state vector \( y_{i+1} \) is showing up in implicit form which in general does not permit solving for \( y_{i+1} \) as \( F \) is most likely nonlinear with respect to some of the state variables.

A common approach for handling this problem consists in applying a so called “predictor - corrector” concept.

Using the explicit Forward-Euler method, the state vector \( y_{i+1} \) at \( t_{i+1} \) is predicted as

\[ y_{i+1, p} = y_i + h F(t_i, y_i) \]  

Inserting this into \( F(t, y(t)) \) at \( t = t_{i+1} \) yields

\[ y_{i+1, c} = y_i + h \frac{F(t_i, y_i) + F(t_{i+1}, y_{i+1, p})}{2} \]  

where \( y_{i+1, c} \) is the “corrected” state vector.

(3.32) and (3.33) are the necessary steps for applying Heun’s method. It can be shown that this method is of second order [HW91]. Of course, the difference between \( y_{i+1, p} \) and \( y_{i+1, c} \) can be made arbitrary small by using \( y_{i+1, c} \) successively as predictor in the right-hand side of (3.33). However, this is not done in practice as additional iterations will not affect the fact that Heun’s method is of second order. A deeper discussion on stiff systems of ODEs can be found in [Aik85].
Gear's Method

Starting from the test system

$$\dot{y}_l(t) = -\lambda_l y_l(t) = -(\alpha_l + I\beta_l) \dot{y}_l(t), \quad l = 1, 2, \ldots, N$$

(3.34)

or in general for a system of ODEs in matrix notation

$$\frac{d}{dt} \dot{\mathbf{y}} = \mathbf{F}(t, \dot{\mathbf{y}})$$

(3.35)

The analytical solution of the test system is given by

$$\dot{y}_l(t) = c_l \exp(-\lambda_l t) = c_l \exp(-\alpha_l t) \sin(\beta_l t)$$

(3.36)

where $\alpha_l$ and $\beta_l$ represent the real and the imaginary part, respectively, of the $l$-th eigenvalue.

According to Gear [Gea71], a method for integrating stiff systems of ODEs should show a domain of stability as given in figure 3.3. The subregions a, b and c, respectively, are discussed as follows:

Case 1 $\Re(\lambda_l) = \alpha_l \geq 0$

Since the step size $h$ is always positive and $h > 0$, we note that $\Re(\sigma) = \Re(h\alpha_l) > 0$, which means the whole right half plane in figure 3.3. We have to discuss two cases:
Region a: Fast component $\hat{y}_i(t)$ of $\hat{y}(t)$ has not yet died out, so an accurate integration is needed in this transition phase. The region is limited by

\[
\begin{align*}
\text{Re} : & 0 \leq h\alpha_i < \delta < 1 \\
\text{Im} : & \frac{\beta_i h}{2\pi} \leq \frac{1}{8} \Rightarrow \theta \leq \frac{\pi}{4}
\end{align*}
\] (3.37)

Region b: Fast component $\hat{y}_i(t)$ of $\hat{y}(t)$ has sufficiently died out. This means that $y_i(t) \approx 0$ and the remaining (slow) components are dominant. For their integration, no high accuracy is required. This region is limited by

\[
\text{Re} : h\alpha_i \geq \delta > 0, \quad h \text{ large}
\] (3.38)

Case 2 $Re(\lambda_i) = \alpha_i < 0$

Since $h$ is positive and $h > 0$ we have a growth factor which leads to growing exponentials which lead to an instable integration on the open left half plane. Thus, limitations are necessary for region c:

\[
\begin{align*}
\text{Re} : & -\mu < h\alpha_i < 0 \\
\text{Im} : & \frac{\beta_i h}{2\pi} \leq \frac{1}{8} \Rightarrow \theta \leq \frac{\pi}{4}
\end{align*}
\] (3.39)

The factor $\frac{1}{8}$ in the equations above has been proposed by Gear to enforce a certain minimum number of integration steps.

An algorithm satisfying the conditions for the three regions a, b, and c, respectively is called stiffly stable. Algorithms like forward Euler or Adams-Bashford are not suitable for solving stiff problems since they do not satisfy the conditions for region b. Thus, algorithms like Adams-Moulton, backward Euler, Heun’s method, trapezoidal rule, etc. should be used. This leads to general backward differentiation formula (BDF) algorithms. A multi-step method of the common form

\[
\sum_{i=0}^{k} a_i y_{n-i} = hb_0 F_n
\] (3.40)

is called a BDF formula. Note that from now on the notation $\hat{y}$ is replaced by $y$. The parameters $a_i$ in the BDF formulas up to the order $k$
Table 3.1: Parameter values for the k-th order Gear algorithm

<table>
<thead>
<tr>
<th>k</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(a_1)</td>
<td>1</td>
<td>(\frac{4}{3})</td>
<td>18</td>
<td>48</td>
<td>300</td>
<td>360</td>
</tr>
<tr>
<td>(a_2)</td>
<td>-(\frac{1}{3})</td>
<td>-(\frac{9}{11})</td>
<td>-(\frac{36}{25})</td>
<td>-300</td>
<td>-450</td>
<td></td>
</tr>
<tr>
<td>(a_3)</td>
<td>-(\frac{2}{11})</td>
<td>16</td>
<td>200</td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a_4)</td>
<td>-(\frac{3}{25})</td>
<td>-(\frac{75}{137})</td>
<td>-(\frac{225}{147})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a_5)</td>
<td>(\frac{12}{137})</td>
<td>(\frac{72}{147})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a_6)</td>
<td>(\frac{10}{147})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b_k)</td>
<td>1</td>
<td>(\frac{2}{3})</td>
<td>(\frac{6}{11})</td>
<td>(\frac{12}{25})</td>
<td>(\frac{60}{137})</td>
<td>(\frac{60}{147})</td>
</tr>
</tbody>
</table>

are derived from a \(k\--th\) degree polynomial, passing through the data points \((x_{n-i}, y_{n-i})\) for \(i = 0, 1, \ldots, k\). This leads to the \(k\--th\) order BDF algorithms, also called \(k\--th\) order Gear algorithms.

For the system of ODEs

\[
\dot{y} = F(t, y)
\]  
(3.41)

the \(k\)-th order Gear algorithm is given by

\[
a_0 y_{i+1} = \sum_{j=1}^{k} a_j y_{i-j+1} + h b_k F(t_{i+1}, y_{i+1})
\]  
(3.42)

where the parameters \(a_j\) and \(b_k\) are given in table 3.1.

As an example, we insert the values from table 3.1 into the algorithm given in formula 3.42 for second order \((k = 2)\):

\[
y_{i+1} = \frac{4}{3} y_i - \frac{1}{3} y_{i-1} + \frac{2}{3} h F(t_{i+1}, y_{i+1})
\]  
(3.43)

The problem with the implicit Gear formulas is the implicitly given new value \(y_{i+1}\). This implicit equation can not be solved using fixed-
point iterations as known from Heun’s algorithm presented previ¬
ously. The fixed point iteration would converge only in a very lim¬
ited number of cases, where $h < \frac{c}{|\lambda_{\text{max}}|}$. For solving the implicit equa¬
tion without this severe constraint on $h$, a Newton-Raphson technique
must be applied when dealing with stiff problems. In this, the solution
vector $y_{i+1}$ must be found such, that the vector $G(y_{i+1})$ of residuals
becomes a zero-vector:

$$G(y_{i+1}) = y_{i+1} - h b_k F(t_{i+1}, y_{i+1}) - \sum_{j=1}^{k} a_j y_{i-j+1} = 0$$  \hspace{1cm} (3.44)

Then the Newton-Raphson algorithm reads:

$$y_{i+1,v+1} = y_{i+1,v} - H(y_{i+1})^{-1} G(y_{i+1})|_{y_{i+1}=y_{i+1,v}}$$  \hspace{1cm} (3.45)

where

$$H = \frac{\partial G(t,y_{i+1})}{\partial y} \bigg|_{t=t_{i+1},y=y_{i+1}} = 1 - h b J$$  \hspace{1cm} (3.46)

In 3.46, 1 is an identity matrix and $J$ is the Jacobian as defined in 3.25.

As the Jacobian is almost never known in analytical form, the par¬
tial derivatives are approximated by either central, forward or back¬
ward differences. In our code, backward differences as

$$\frac{\partial F}{\partial y_j} = \frac{F(y_1, \ldots, y_{j-1}, y_j, y_{j+1}, \ldots, y_N)}{\Delta y_j} - \frac{F(y_1, \ldots, y_{j-1}, y_j - \Delta y_j, y_{j+1}, \ldots, y_N)}{\Delta y_j}$$  \hspace{1cm} (3.47)

are used.

### 3.2.4. Treatment of Discontinuities

A discontinuity occurs when using different forms of the right-hand
side of an ODE within the same integration step. Discontinuities can
be divided into two classes:

**Events at discrete a priori known time points (Time Events)**

An example is the ODE

$$\dot{y}(t) = \begin{cases} y(t)^2 & \text{if } t \leq 0.5 \\ \cos(y(t)) & \text{if } t > 0.5 \end{cases}$$  \hspace{1cm} (3.48)
When integrating from $t < 0.5$ in the positive $t$-direction and hereby using the relation $\dot{y}(t) = y(t)^2$ it is necessary to adopt the step-size $h$ in such a way that the $i$-th integration step gets finished at $t_{i+1} = 0.5$. The next step will start at $t = 0.5$ and finish at some location $t > 0.5$. However for this step and the following ones the relation $\dot{y}(t) = \cos(y(t))$ is valid.

**Events at the roots of a given state dependent function (State Events)**

An Example is the ODE

$$
\dot{y}(t) = \begin{cases} 
    y(t)^2 & \text{if } \sin(y) \geq 0.5 \\
    \cos(y(t)) & \text{if } \sin(y) < 0.5
\end{cases}
$$

When the time of an occurring discontinuity, e.g. $\sin(y(t)) \leq 0.5$ is not known a priori, a so called *root functions* has to be evaluated. This function $g(t,y) = 0$ has to be supplied. During integration, the solver has to *extrapolate* the expected function value $g(y)_{i+1}$ from previous evaluated values of $g$ and from the next time step expected.

Here the next time step given by the integration algorithm will be $h = t_{i+1} - t_i$. Prior to integration the given root function $g(t,y)$ is ex-
3.2. Numerical Integration

...trapolated up to \(t_{i+1}\). If a change of sign in \(g(t, y)\) will be noted, a discontinuity would appear in this step and the integration step can not be performed. The point of the discontinuity where \(g(\hat{t}, y) = 0\) has to be evaluated by extrapolation. This leads to \(\hat{t}\) and the next integration step will be performed with a lightly smaller step size than \((\hat{t} - t_i)\) so as to avoid overshooting. Clearly, a sufficient accurate localization of a state event requires an iterative approach. Note that the given simulation of the SBWR involves several root functions so that the form \(g(t, y)\) is required. As an example, let's heat up water in a RPV. As the temperature of the water exceeds saturation temperature, an other model (steam or superheated water, respectively) has to be applied. The evaluation of water properties above saturation would lead to an error in the steam tables subroutines and therefore terminate the calculation.

Switching from one model to another one can be driven by the sign of some properly defined root functions. In the software used in this project, an arbitrarily number of root functions can be used.

For a detailed description of LSODAR see E.1 on page 201.
As computer algebra is required for automatically doing the tedious job of transforming the fairly large systems of implicit model equations into an explicit system of ODEs some basics of computer algebra need to be presented in this chapter. Examples are given in Maple which is used throughout this work. Other common computer algebra tools are Mathematica [WGKm05], Maxima, [dSFMsY04] or Kalamaris[Lar00] which is available under the GNU Public License, just to name the most important ones.

Maple is a system for symbolic mathematical computation which has been developed at the University of Waterloo[Mol04], Ontario, Canada. Unlike numerical software packages such as Matlab [Mol04], Maple computes with symbols rather than with numbers. However, Maple can evaluate numerical results using its own floating point engine, which can perform computations with any desired accuracy. Besides computer algebra and numerical evaluation Maple has a very advanced, built-in plotting engine for two- and three-dimensional plots with numerous options. All these capabilities are integrated in an interactive user interface. The use of this Graphical User Interface (GUI), however, is optional. Maple can be started in batch mode or in
pure text mode as well.

Maple consists of a system kernel, written in the programming language C [KR88], and a library of mathematical procedures. The kernel supports basic activities such as floating point arithmetic, integer arithmetic, polynomial evaluations as well as the Maple programming and command language. This language is similar to Pascal [JW91] and is used in Maple programs, in mathematical libraries, and in interactive sessions. The library functions which are needed are loaded at runtime. Due to this, Maple memory requirements are adapted dynamically.

4.1. The Role of Maple Within the Scope of this Work

Once the equations are formulated for a given module as presented in chapter 2, they have to be coded in Maple. Maple as a computer algebra tool understands equations in a more natural way than other common programming languages. As an example, the model of the gas space is presented. The other modules are handled similarly. For simplification, the system of ODEs 2.26 on page 17 is given again. Then the whole Maple program is presented and discussed here.

The System of ODEs (given in 2.26) reads:

\[
\begin{align*}
\frac{dM_v}{dt} &= \dot{M}_f + \dot{M}_b - \dot{M}_{out} x_v + \dot{M}_{v,in} \\
\frac{dM_a}{dt} &= - (1 - x_v) \dot{M}_{out} + \dot{M}_{a,in} \\
M_a \frac{dh_a}{dt} + M_v \frac{dh_v}{dt} &= \sum \dot{Q}_i + \dot{Q}_v + \dot{M}_f g (h_{v1} - h_v) \\
+ &\dot{M}_b (h_b - h_v) + \dot{M}_{v,in} (h_{v,in} - h_v) + \dot{M}_{a,in} (h_{a,in} - h_a) + V_g \frac{dp_g}{dt} \\
\frac{dx_v}{dt} &= \frac{p_a}{p_g^2} \left( \left( \frac{dM_a}{dt} R_a T p_a + \frac{dT}{dt} M_a p_a R_a - \frac{dp_a}{dt} M_a TR_a \right) + \right. \\
&\left. M_v \left( \frac{\partial v}{\partial p_v} \right|_T \frac{dp_v}{dt} + \frac{\partial v}{\partial T} \right|_p \frac{dT}{dt} \right) + v_v \frac{dM_v}{dt}.
\end{align*}
\]

In a first step, the equations from 2.26 have to be translated into Maple syntax. Note the syntactic difference between the = and := op-
4.1. The Role of Maple Within the Scope of this Work

Equations from chapter 2, translated into Maple syntax:

```maple
restart;
readlib( unassign );
with( linalg );

Warning, the protected names norm and trace have been redefined and unprotected

Equations from chapter 2, translated into Maple syntax:

\[ \text{eq\_mass\_1} := \frac{dM}{dt} = \text{Mdot\_fg} + \text{Mdot\_b} - \text{Mdot\_out} \cdot x_v + \text{Mdot\_vin}; \]

\[ \text{eq\_mass\_2} := \frac{dM}{dt} = \text{Mdot\_fg} + \text{Mdot\_b} - \text{Mdot\_out} \cdot x_v + \text{Mdot\_vin}; \]

\[ \text{eq\_mass\_2} := \frac{dM}{dt} = -(1-x_v) \cdot \text{Mdot\_out} + \text{Mdot\_ain}; \]

\[ \text{eq\_ener} := M_a \cdot \frac{dh}{dt} + M_v \cdot \frac{dh}{dt} = Qdot_i + \text{Mdot\_fg} \cdot (h_{vi} - h_v) + \text{Mdot\_b} \cdot (h_b - h_v) + \text{Mdot\_vin} \cdot (h_{vin} - h_v) + \text{Mdot\_ain} \cdot (h_{ain} - h_a) + Qdot_v_i + V_g \cdot dp_g / dt; \]

\[ \phi_v := \frac{\text{mol\_a} \cdot \text{mol\_v} \cdot dx_v / dt}{((1-x_v) \cdot \text{BioI\_a} + s_v \cdot \text{mol\_v}) A^2} = \frac{1}{p_e A^2} \cdot CP_v \cdot \frac{dp_v}{dt} - p_w \cdot \frac{dp_g}{dt}; \]

\[ dvol := \frac{dV_g}{dt} = \frac{dM}{dt} \cdot R_a \cdot T \cdot p_a - M_v \cdot \frac{dv_v}{dt} \cdot dp_v + \frac{dV_g}{dt} \cdot p_v \cdot dp_g / dt; \]
```

Figure 4.1: Step 1: Definition of the equations
operators: the first one is the (mathematical) equality operator while the second one is Maple’s assignment operator. Thus, every equation is assigned to a name. As example, the first equation is assigned the name *eq_mass_1*. This step is shown in the Maple worksheet 4.1.

The second step shows how state equations and other physical relations are entered into Maple to complete the system. This is performed in the Maple worksheet 4.2.

In the third step solving the system of ODEs for the derivatives is presented in the worksheet 4.3. Note, that all the output of the solution is suppressed as it would require several hundred lines of code and is therefore neither intended to be human-readable nor to be modified.

Finally these solutions are converted into Fortran code. For better performance this code has to be optimized. The code is stored in a file (here: *autofort.for*), which is not intended to be human readable. This step is shown in the Maple worksheet 4.4.

The resulting Fortran code just for one module counts for as much as 1027 lines! For giving an idea about this automatically generated code, just the first and last lines are given in the listing below. The code is optimized with respect to the number of operations. While in the first part, many user defined known variables are involved in the code, there are almost only internal variables in the end of the listing.

**Listing 4.1: Automatically generated Fortran code**

```fortran
TL = Mdot fg*M_v 
T2 = Mdot fg*M_a 
T3 = Mdot b*M_v 
T4 = Mdot b*M_a 
T5 = Mdot_out*M_v 
T6 = Mdot_vin*M_v 
T7 = Mdot_vin*M_a 
T10 = 1/(M_v+M_a) 

DM_vdt = (t1+t2+t3+t4-t5+t6+t7)*t10 
T11 = Mdot_a*M_a 

#05-t2327*t2374*t2059*t2061+t2352+t2554-t2365*t1964-t2266*t2374+2*t2177*t2654+t2083*t2091-t2266*t2381 
b1 = t2327*t2268+t2076*t2079-2*t2343*t2359-t2365*t1960-t2365*t1987 
#-2*t2627*t1468+2*t2210+t2247+2*t2630*t2772-t2107-t2231-2*t2210*t22 
#54-t2371*t2317-t2630*t1556-2*t2630*t2765+t2552*t2237-t2334*t2768+2 
#*t2627*t1407-t2266*t2340+t2589+t2266*t2328-2*t233*t2630*t1416 
#t2660*t2661+2*t1000*t1153*R_a*t2054*Mdot_out*t2070-t2377*t2357*t2 
#181*t2167*t1125*t2085+t2083*t2085-t2630*t1500-t2266*t2357*t2233*t1 
#561*t2076*t2068-t2630*t1400-t2334*t2770+t2192+t2620+t2227+t2646+t2 
#293*t2323+s2+t2401+t2426+t2700+t2730 

s2 = t915*t1146 
dTdt = s1*s2
```
The five equations have now to be collected as a system of equations. This collection is termed a 'set' and is an unordered sequence of distinct expressions enclosed in braces, representing a set in the mathematical sense. The user should not assume that the expressions will be maintained in any particular order because Maple uses an ordering convenient for its implementation. In this example, the command is completed with a colon (:) instead of a semicolon (;) to suppress the output.

```maple
> equsystem := [eq_mass_1, eq_mass_2, eq_ener, phi_v, dvol]:

State equations

Some state equations and other definitions of physical properties are added, to close the system. These equations are entered in a very similar way:

```maple
> h1 := dh_vdt = cp_v * dTdt - (T*dv_vdT_p - v_v) * dp_vdt;

> h2 := dh_adt = cp_a * dTdt;

> xv := x_v = M_v / (M_v + M_a);

> dxv := dx_vdt = 1/(M_v + M_a)^2 * (M_v*dv_vdt + (M_v + M_a) - M_v*(dM_vdt + dM_adt));

> pg := p_g = p_a + p_v;

> dp_g := dp_gdt = dp_adt + dp_vdt;
```

```maple
> dpg := dp_gdt = dp_adt + dp_vdt
```

Figure 4.2: Step 2: Definition of state equations
Now the state equations are substituted back into the system defined before. Thus we get a system of five equations referred to as `equsystem` below, and five unknowns which can be readily solved:

```plaintext
> equsystem := subs(h1, h2, x1, d1x, p1, dp1, 
        equsystem);

equsystem :=

\[
\begin{align*}
\frac{dV_g}{dt} &= 
\frac{dM_j}{dt} R_a T_p a + \frac{dT}{dt} M_a p_a R_a - M_a T_R a \frac{dp_a}{dt} \\
+ M_v (d\nu dp_T d\nu dt + d\nu d\nu T p d\nu dt) + v_v dM_v dt, \\
\frac{dM_v}{dt} &= M_{dot g} + M_{dot in} - M_v (d\nu d\nu T p d\nu dt) + v_v dM_v dt, \\
\frac{dM_a}{dt} &= M_{dot a} + M_{dot v} (d\nu d\nu T p d\nu dt) - M_a (d\nu d\nu T p d\nu dt), \\
\frac{dM}{dt} &= \frac{M_v mol_a}{M_v + M_a} + \frac{M_v mol_v}{M_v + M_a} - \frac{mol}{M_v + M_a} \\
\frac{dp_a + p_v}{p_a + p_v} dp_v dt &= p_v (dp_a dt + dp_v dt), \\
\end{align*}
\]

\[dM_a = \left(1 - \frac{M_v}{M_v + M_a}\right) M_{dot a} + M_{dot in},\]

\[M_v c_p_a dT dt + M_v (c_p_v dT dt - (T d\nu T p d\nu dt) dp_v dt) = Q_{dot i} + M_{dot g} (h_i - h_v) + M_{dot v} (h_v - h_a) + Q_{dot i} + V_g (dp_a dt + dp_v dt),\]

\[mol_a mol_v (dM_v dt (M_v + M_a) - M_v (dM_v dt + dM_a dt)) =\]

\[\frac{(M_v + M_a)^2}{(p_a + p_v)^2 dp_v dt} p_v (dp_a dt + dp_v dt)\]

> assign(solve(equsystem, \{dM_v dt, dM_a dt, dp_v dt, 
        dp_a dt, dT dt\}));

> dM_a = \left(-M_{dot a} M_a + M_a M_{dot out} - M_{dot in} M_v\right) \left(\frac{1}{M_v + M_a}\right).

Figure 4.3: Step 3: Solving the system of ODEs
4.1. The Role of Maple Within the Scope of this Work

The command `allvalues` is used to compute all possible values of expressions. If the option 'dependent' is given, a particular `RootOf(p)` represents the same root at each occurrence within `expr`. This is the default. In contrast, if the option is 'independent' then each occurrence of `RootOf(p)` is to be treated independently. This option has no effect on `RootOfs` containing a selector which makes them single-valued.

Most output is suppressed in the following commands.

```maple
> soil := allvalues(dM_ydt, 'dependent');
> sol2 := allvalues(dM_adt, 'dependent');
> sol3 := allvalues(dp_vdt, 'dependent');
> sol4 := allvalues(dp_adt, 'dependent');
> sol5 := allvalues(dp_Tdt, 'dependent');
> unassign('dM_ydt', 'dM_adt', 'dp_vdt', 'dp_adt', 'dTdt');
```

Generating Fortran Code

```maple
> with(codegen):
> fortran([dM_ydt = soil, dM_adt = sol2, dp_vdt = sol3, dp_adt = sol4, dTdt = sol5 ], optimized, filename = 'autofort.for');
> cost(sol5);
   417 additions + 4466 multiplications + 2 divisions
> cost(optimize(sol5));
   1243 multiplications + 373 assignments + 417 additions + 2 divisions
```

Figure 4.4: Step 4: Generating Fortran Code
The concept of this way of code generation has some major advantages:

- The user can write these equations in a "more natural" way depending on his preferences. In this the user can largely follow the form how the equations were formulated in 2. The state equations of the overall system and necessary substitutions therefore can be done automatically.

After the implicit system of ODEs, which describe a module, have been solved for the time derivatives, the system is available in explicit form suitable for numerical computation. This allows for a better performance of the used integrator.

- The code can be optimized with respect to the number of floating point operations. As example, the cost of the temperature derivative can be reduced from 4885 floating point operations down to 1662 floating point operations plus 373 assignments(!). See lower part of maple worksheet 4.4.

- The user does not have to worry about internals of the code for modifying physical models. Beside this, the Fortran code is highly optimized with respect to floating point operations. This is a great benefit for computing time needed for function evaluations. The main advantage however, is the fact that transforming the originally implicit structure of the ODEs into an explicit one would be extremely laborious and error prone without the use of a program for symbolic computation, formula manipulation, and code generation.
The Input Language

To create a valuable and usable modular environment for simulating complex thermal hydraulic systems, a language is needed for describing the system. This language has to be simple if it is to be used by an unexperienced user who usually has no computer science background. Thus, this language should hide as much of the overall modules and integration complexity as possible. On the other hand, the output code produced has to be a clear, well structured and commented Fortran code which can be edited by an experienced user.

5.1. Compiler, Translator and Interpreter

5.1.1. Definition of Terms

In order to deal with expressions like compiler, semantics, or language, their definitions are shortly given below (citations from the references in italic font).

Compiler: A compiler, or translator, is a computer program that translates a computer program written in one computer language (called the source language) into a program written in another computer language (called the
output or the target language). [tfe05]

 Interpreter: An interpreter is a computer program that executes other programs. This is in contrast to a compiler which does not execute its input program (the source code) but translates it into executable machine code (also called object code) which is output to a file for later execution. It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced. [tfe05]

 (Formal) Language: Mathematics and computer science use artificial entities called formal languages (including programming languages and markup languages). These often take the form of character strings, produced by some combination of formal grammar and semantics of arbitrary complexity. [tfe05]

 Grammar: In computer science a formal grammar is an abstract structure that describes a formal language precisely: i.e., a set of rules that mathematically delineates a (usually infinite) set of finite-length strings over a (usually finite) alphabet. Formal grammars are so named by analogy to grammar in human languages. [tfe05]

 Semantics: In general, semantics (from the Greek semantikos, or "significant meaning," derived from sema, sign) is the study of meaning, in some sense of that term. Semantics is often opposed to syntax, in which case the former pertains to what something means while the latter pertains to the formal structure/patterns in which something is expressed (e.g. written or spoken). [tfe05]

5.1.2. Formal Description

From an input language, to the output of a simulation run of a complex thermal hydraulic system, several compilers and interpreters are necessary. Beside the compilers, an input language has been developed which will be presented later. This language makes the use of the modules, the setup of a system, and the generation of an usable Fortran code much easier than if all the work were done manually. Thus, before starting with the description of the input language, a formal approach to interpreters and compilers is needed.

Programming Languages

A programming language is the formal definition, how a function is to be written to produce a specific output on a given input. The set of all syntactically correct programs written in the programming language
$L$ is denoted $L$-programs. Thus, the meaning of a program $p \in L$-programs is defined as

$$[p]_L : \text{input} \rightarrow \text{output}$$  \hspace{1cm} (5.1)

This means, that the program $p$, written in $L$, computes output on input.

**Interpreters**

Considering now an implementation language $L$. This means, there exists a machine which can run programs in $L$. Furthermore, another language $S$, the source language is given. An interpreter $\text{int} \in L$-programs for a source language $S$ has two inputs: the source program $p \in S$-programs to be executed, and its input data $d \in D$ which is to be processed by $S$. Finally, running $p$ with $d$ on a $S$-machine must lead to the same output as running the interpreter $\text{int}$ with inputs $d$ and $p$ on a $L$-machine.

Formally, the $L$-program $\text{int}$ is an interpreter for $S$, if for every $p \in S$-programs end every $d \in D$

$$[p]_S d = [\text{int}]_L[p, d]$$  \hspace{1cm} (5.2)

The set of all interpreters for $S$ written in $L$ is usually written with the following symbol:

$$\begin{array}{c}
S \\
L
\end{array} = \{ \text{int} | \forall p, d. [p]_S d = [\text{int}]_L[p, d] \}$$  \hspace{1cm} (5.3)

**Compilers**

Instead of interpreting a given program every time it is executed, it can be translated into the corresponding machine language a priori. Thus, let $T$ be a target language. Then, a compiler $\text{comp} \in L$-programs which transforms programs from a source language $S$ into a target language $T$ has one input, which is the source program $p \in S$-programs which has to be compiled. Running the compiler with input $p$ on a $L$-machine (Remember: $\text{comp} \in L$-programs) produces a target program such that running it on a $T$-machine has the same effect as running $p$ on a $S$-machine.
Formally, the $L$-program $comp$ is a compiler from $S$ to $T$, if for every $p \in S$-programs and for every $d \in D$

$$[P]_{sd} = [[comp]_L P]_{Td} \quad (5.4)$$

The set of all compilers from $S$ to $T$, written in $L$ is usually written with the symbol

$$S \rightarrow^L T = \text{comp}\forall p, d, [P]_{sd} = [[comp]_L P]_{Td} \quad (5.5)$$

**Compilation of programs**

The compilation process assumes some given, correct $S$-programs. The purpose of these programs is not known to the compilation process. This set of correct $S$-programs is denoted by

$$S \quad \quad \quad (5.6)$$

Suppose now a given compiler $comp$ which translates programs from a source language $S$ to a target language $T$. The compiler itself is written in $L$. Then, it is possible to perform translations on a $L$-machine from $S$ to $T$. This situation can be depicted as

$$\text{source program} \in S \rightarrow^L T \Rightarrow \text{target program} \quad (5.7)$$

which means that the compiler $comp$ translates every correct source program written in $S$ into a valid $T$ program.
5.1.3. Compilation in the Context of the Input Language

It is obvious that the code for simulating a complex thermal hydraulic setup can not be written manually for every possible setup from scratch. Thus, a high level language will be defined, which allows the user to choose the compartments, their connections, input parameters, etc.

This compiler has no direct link to the executable simulation code. Furthermore, the computer language, in which the numerical algorithms are implemented (Fortran 77) is not suitable for writing compilers because compilers usually build up a huge intermediate data structure, as discussed in detail later. The handling of these data structures can hardly be made static, but is implemented usually dynamically. This means, that the size of the data tree can grow during compilation time and is (theoretically) unlimited\(^1\). Furthermore, the intermediate data structures, often implemented as trees, are parsed using recursive algorithms. Both, recursive programming as well as dynamic data structures are not supported by Fortran 77. Due to these limitations, the compiler is written in C\cite{KR88}.

Thus, at least two more computer languages beside our own language for describing thermal hydraulic systems are involved in the generation of an executable simulation code.

The development of an interpreter for our own language would be very tricky and difficult, as no target language exists which fulfills all the needs of our simulations, e.g runtime interpretation of numerical integration, manual modification on automatic generated parameters, etc.

Due to this and from the theory given the preceding chapters, we can derive that compilers have to play a key role in the translating process. Given the fact, that C and Fortran 77 compilers are well available on virtually every computing platform, only the compiler for our own language has to be developed and implemented. The Source language of this compiler is given. But which is the target language? It is obvious, that the target language for the final program must be executable on a real existing computer, thus must be some x86, or SPARC binary code, just to name two examples.

The starting points are the following:

\(^1\)limited only by the restrictions of the operating system and the available hardware and processing time
Figure 5.1: Compilers and their interfaces needed to translate an MSDL program written in M into an executable program in machine language T

- Source language of the thermal hydraulic system: MSDL\(^2\).
- Source language of the MSDL Compiler: C.
- Target language of the MSDL Compiler: Fortran 77.
- C compiler is available to produce machine code.\(^3\)
- Fortran compiler is available to produce machine code.

Thus, in summary, the situation in figure 5.1 is to be considered a suitable solution for writing, compiling and running an MSDL program, using existing Fortran and C compilers.

As the compilers are not the whole story, but many external libraries and interpreters, as well as manual modifications of automatic generated code are needed, the overall process is depicted in the overall work flow diagram depicted in figure 5.2.

In the following sections, the relevant basics for compiler construction is presented in section 5.2, including a overview of computer language theory, syntax and semantics. Regular expressions are explained, and compiler compilers are introduced in general, and, of course yacc as one of it’s most known realizations is presented. The section closes with lexical and syntactical analysis of languages.

\(^2\)Modular System Description Language
\(^3\)machine code means here come code which can be linked against system libraries and then be executed on a specific, real digital computer. The theory of linking, relocating, and other needed topics for back-end requirements are not discussed in this work.
5.1. Compiler, Translator and Interpreter

Write MSDL Program file

Compile MSDL file to F77

Edit Fortran File if necessary

Compile Fortran file

Link to numerical and physical modules

Execute resulting program

Plot results

Develop physical Modeling Equations

Compile Modeling Eqns into F77 using Maple and Shell Tools

Evaluate and develop numerical methods

Implement and compile physical Modules

Implement and compile numerical Modules

MSDL Toolbox, to be implemented and compiled only once

Figure 5.2: Workflow from MSDL to simulation output
Section 5.3 deals with the development of MSDL. Its syntax as well as its lexical conventions are presented and discussed using an example of MSDL.

The remaining sections of this chapter are dealing with the implementation of the MSDL compiler for this work. The implementation of the lexical analyzer and of the parser is discussed. The intermediate data structures are presented as well as the overall architecture of the front-end of the MSDL compiler. This chapter closes with the discussion of the code generator and the output file format.

5.2. Compiler Construction Basics

The translation from the input language to a Fortran code is done by a compiler. Simply stated, a compiler is a program that reads a program written in one language and translates it into an equivalent program in another language. These two languages for input and output are called source language and target language, respectively. As an important part, the compiler reports the presence of errors in the source language to the user.

Some years ago, the construction of a compiler was a complicated task. As an example, the first Fortran compiler took 18 man-years for its implementation [Bac57]. However, since then, systematic techniques for handling many of the important tasks have been discovered. Good implementation languages, programming environments, and software tools have been developed. With these advances, a compiler for an input language which describes thermal hydraulic systems under consideration in this work could be implemented within reasonable time.

5.2.1. Principles of Compilers

The translation process essentially consists of the following parts, as shown in figure 5.3:

The lexical analyzer (or on short form the lexer) translates the sequence of characters of the source language into corresponding symbols of the vocabulary of the language. As an example, a sequence of digits, including a dot will be recognized as a real number, where a sequence of characters enclosed in double quotes will be identified as a quoted string. This phase of the compilation is called lexical analysis.

The next step is to transform the sequence of symbols into valid
5.2. Compiler Construction Basics

phrases of the language. This step is called syntax analysis, or parsing. High level languages are classifying variables and functions according to their type. Therefore, in addition to syntactic rules, the language is defined by additional compatibility rules or semantic analysis among the types. Hence, a verification of this compatibility rules is an additional duty of the compiler. This verification is called type checking. Within the presented language, this type checking capability makes sure that only compatible modules are connected to each other, e.g. we do not allow to connect a pipe conducting liquid water to a gas space.

The parser builds up an intermediate data structure which represents the syntactical definition of the language. This data structure is called a syntax tree. The syntax tree is then used by the next instance, the code generator. This code generator creates an output code which the vocabulary and syntax of the target language.

As all the units are decoupled, the syntax tree can be considered as a central instance of the compiler. This syntax tree is build up by a front end part. This front end part can be exchanged without the need of rewriting the code generator. In this work, this leads to the possibility to replace the lexer for the input language with a graphical user interface which builds up the syntax tree interactively.

On the other hand, the code generator can be considered as a back end. This allows to replace the existing Fortran code generator by a code generator which generates any other code like C or others. This structure is depicted in figure 5.4.

5.2.2. Languages and Syntax

Before starting with the description of the implemented input language for the systems under consideration, we have to look at lan-
Every language displays a structure called its grammar or *syntax*. As an example, an English sentence always consists of a subject followed by a predicate. This simple fact can be described in a formal way by

\[
\text{sentence} = \text{subject} \text{ predicate}.
\]

Now, the subject and predicate have to be defined:

\[
\text{subject} = \text{"John" | "Mary"},
\]

\[
\text{predicate} = \text{"eats" | "talks"}.
\]

The above formulas are called *productions* or *syntax rules*. Subject and predicate are called *syntactic classes*. A *language* is now the set of all possible sentences which can be generated by repeated substitution of the left-hand sides by the right-hand sides of the above productions. Note that a language usually has an infinite set of valid sentences.

A language is defined by the following four items:

1. A set of *terminal symbols*. These are the symbols (words) that occur in the language. Terminal symbols can not be substituted by other symbols. The set of terminal symbols is also called the *vocabulary* of the language. Terminals are often called *tokens* in the context of grammars for programming languages.

2. The set of *nonterminal symbols*. The nonterminal symbols are the syntactic classes, such as subject and predicate in the above ex-
ample. Nonterminals also impose a hierarchical structure on the language that is useful for syntax analysis.

3. A set of productions which describe the language. For every non-terminal symbol at least one production is needed.

4. The start symbol. The start symbol is a selected nonterminal symbol. In the above example, the symbol “sentence” is the start symbol.

This notation was introduced in 1960 by J. Backus and P. Naur for the description of the Algol 60 programming language and is therefore called the Backus Naur Form (BNF) [ed69]. The BNF has been extended to the EBNF by N. Wirth [Wir77] in 1977 to allow repetitions of symbols within a single production.

For the construction of a compiler, a language should be context free and regular. In a context free language a substitution defined by any production is always permitted, regardless of the context in which the symbol is embedded within the sentence.

A regular language is a subset of the context free languages. This subclass plays a significant role in the realm of programming languages. In essence, they are the context free languages whose syntax contains no recursion except for the specification of repetition. Thus, a language is regular if its syntax can be specified by a single EBNF expression.

5.2.3. Regular Expressions

The first time we met regular expressions was when we discussed the awk language in chapter A. Regular expressions are widely used within the Unix operating system. Lex, egrep and many other programs and languages use a rich regular expression language.

A regular expression is a pattern description using a metalanguage. That is, a language to describe particular patterns of interest. The characters used in this metalanguage are part of the standard ASCII character set. Due to this, sometimes a regular expression may look a little cryptic or strange to the unexperienced user. The characters which form a regular expression are:

- Matches any single character except the newline character.

* Matches any number (including zero) copies of the immediately preceding expression.
[] Defines a character class. Matches one of the enclosed characters. If the first character within the brackets is a ^, the meaning is negated: match any character which is not given in the brackets. A dash (-) indicates a character range: [A-F] means the same as [ABCDEF]. If a dash or a square bracket is the first character within the brackets, this character is interpreted literally. This allows to include these special characters into character classes.

^ Matches the beginning of a line if used as the first character of a regular expression. Also used for excluding character classes (see above).

$ As the last character of a regular expression, $ matches the end of a line.

{} Says how often the previous expression is allowed to match. For example, the expression A{2,3} matches AA or AAA, but not A or AAAA.

\ Is used to escape the meaning of a meta-character and is part of the usual C language escape sequences (e.g. \t stands for the tab-character).

+ Matches one or more copies of the preceding expression, but not zero copies.

? Matches zero or one occurrences of the preceding expression. With this, -? [0-9]+ matches an integer number with an optional leading minus.

| Matches either the preceding or the following regular expression.

() Groups regular expressions together to new regular expressions. Often used to built up complex regular expressions with |, + and *.

As these definitions are quite hard to understand, some examples should clarify the usage of regular expression. The first one,

[ \t]+ matches any white space in a text. White space in this definition is one or more space character or tab character. This regular expression is later used to ignore white space in the input language.
An important thing for an input language is the possibility for writing comments which are ignored by the compiler. A line beginning with a # character is defined to be a comment. The regular expression to match a comment line in the input language is simple:
#.*

In our language to be defined, a quoted string is needed to define names. It turns out as an important detail that always the longest possible string is matched by a regular expression. This is the problem when writing a regular expression for matching the quoted string, because, in case of a line like
Pipe "pl", Type = Gas, Src="RPV", Dest="DW1"
we want to match every single name and not the longest possible, the regular expression which does the job is
\"[^\"\n]*[^\"\n]\n
which matches a word between two double quotes within the same line.

To get an expression within brackets, e.g. ( Volume ), we have to change the expression above to
\(([^\"\n]*[^\"\n])\n
As a last example I want to show how a regular expression is built up to match a decimal number in any common form. The regular expression for a single digit is
[0-9]
which can be expanded to match an integer value by
[0-9]+ where the plus sign does not allow numbers with zero digits. To allow an optional unary minus, we write
-? [0-9]+ The next step is to expand the regular expression to allow decimal numbers. This is done by [0-9]*\.[0-9]+ This expression does no longer match any integer value without the decimal point. To reintroduce the integer values again, we have to combine the two former definitions:
\([0-9]+\) | (0-9)*\.[0-9]+\)
As both of the expressions, integer or decimal numbers should allow the unary minus, we have to expand the expression to
-?\(([0-9]+\) | (0-9)*\.[0-9]+\))
A decimal number is often represented in exponential form, e.g. 1.4E-6. To allow this, we write a regular expression for the exponent:
[eE][-+]? [0-9]
This exponent expression can be combined with our former regular expression for a number. This leads to the final version of a regular expression representing a real number:

\[-?((0-9)+) | ((0-9) \star . (0-9)+)((eE) [-+] ? (0-9)+)\]

which will be used in the lexer of our input language later. More about regular expression and a detailed discussion can be found in [LMB95], [Dou91], [KP84].

5.2.4. Lexical Analysis

Lex is a tool designed for writers of compilers and interpreters. Some examples of programs which have been implemented using lex for the lexer part are the Unix desktop calculator bc, the Gnu C compiler GCC or the Lex program itself. The compiler for our thermal hydraulic systems also uses the help of lex for lexical analysis.

Lex is used to specify the lexical analysis of our input language. Lex is used in the manner depicted in figure 5.5. First, a specification of a lexical analyzer is prepared by creating a program lex.1 in the lex language. Then, lex.1 is run through the lex compiler to produce a C program lex.yy.c. The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expressions of lex.1, together with a standard routine that uses the table to recognize the symbols. The actions associated with regular expressions in lex.1 are pieces of C code and are carried over directly to yy.lex.c. Finally, yy.lex.c is run through the C compiler to produce an executable program (a.out), which is the lexical analyzer that transforms an input stream into a sequence of tokens defined, as well as error messages, as appropriate.
5.2. Compiler Construction Basics

As an example, the character stream from the example listing (see page 91)

Listing 5.1: Character stream to parse

```
# Created Jan 12, 1999
#
Name "p3g"
Vessel "RPV", Type = PV
```

will be translated in the stream of tokens (note, the lexer will not create any tokens for comment lines):

```
name qstring vessel qstring , type = pv
```

5.2.5. Syntactical Analysis

For the parsing several tools are available as well as for the construction of the lexer. When parsing a language, some strategies are commonly used: Either we are beginning at the top of the parse tree and built up the tree until we reach the leaves (top-down approach) or we do the task vice versa: We start at the leaves of the parse tree and built up the tree up to the root. According to Knuth [Knu65], bottom-up parsing is called LR Parsing and top-down method is called LL parsing, respectively. Yacc, our parser generator, uses a parser technique called LALR (Lookahead LR). Yacc stand for “yet another compiler compiler” and is available on a variety of Unix systems, but also on many other platforms including windows and mac like PCs. It works according to figure 5.6: The yacc specification is converted into a C code y.tab.c. The Program y.tab.c represents a classical implementation of the LALR method [ASU88]. The program y.tab.c the is compiled using

![Diagram of yacc process](image-url)
a standard C compiler to create an executable program, called a.out in this example. Using this program a source program can be translated into an output program. This mechanism will be outlined in the discussion of the specification of the compiler for thermal hydraulic systems presented later.

As the productions analyzed by the parser are used to build up the intermediate data structure, we might emphasize this fact by the following example. The following line of code (from the example listing on page 91)

\[
\text{Vessel "RPV", Type = PV}
\]

will result in the fully initialized internal data structure

<table>
<thead>
<tr>
<th>17: *system-&gt;v</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = 0x80724e8 &quot;RPV&quot;</td>
</tr>
<tr>
<td>numofpipe = 2</td>
</tr>
<tr>
<td>init_gasspace =</td>
</tr>
<tr>
<td>p_tot = 350000</td>
</tr>
<tr>
<td>p_a = 1000</td>
</tr>
<tr>
<td>init_pv =</td>
</tr>
<tr>
<td>p_tot = 350000</td>
</tr>
<tr>
<td>p_a = 1000</td>
</tr>
<tr>
<td>LiqLevel = 6.1299999999999999</td>
</tr>
<tr>
<td>T_1 = 104.8</td>
</tr>
<tr>
<td>next = 0x8072578</td>
</tr>
</tbody>
</table>

Note the numerical behavior of floating point numbers: It is not possible to represent 6.13 in the binary floating point format; it has to be rounded down to 6.1299999999999999, which is the nearest representable floating point number.

### 5.3. Source Language: MSDL

To get a proper language which can be translated by a machine, it has to follow certain rules which are well known from compiler designs for other computer languages such as C or Pascal [Wir77]. A regular language can be described by a grammar in EBNF [Wir96], like on page 92. This grammar has to be parsed and interpreted by a compiler.

Furthermore, this developed language must be extensible in a simple way for future enhancements or changes.

The specification led to a new language, called Modular System Description Language, or MSDL, respectively. As the name supplies, the language is not a new language for numerical computer simulations, but is a language for describing modular (thermal hydraulic) systems. In this sense, MSDL is designed to generate Fortran (or other
computer code) rather than to run complete computer simulations.

An example Program in MSDL is given in the next subsection. Next, the syntactical and lexical part of the language have to be discussed. The compiler for the MSDL Program must accept a program like the one given below. In order to break up the complexity of the source languages, two parts are given: one for the syntactical definitions and for the lexical conventions, respectively.

5.3.1. Example listing

The Example listing and its generated output program in Fortran can be found in Appendix D on page 187.

Listing 5.2: MSDL Example setup for P3 Simulation

```
# Description file for simulation setup P3G equivalent
# Created Jan 12, 1999
#
Name "p3g"

Vessel "RPV", Type = PV
Vessel "WW2", Type=PV
Vessel "WW1", Type = PV

Volume (RPV) = 22.8
Crosssection(RPV) = 1.2271846

Volume(WW1) = 117.0
Crosssection(WW1) = 7.161972

Volume(WW2) = 117.0
Crosssection(WW2) = 7.161972

Pipe "p1", Type = Liquid, Src="WW1", Dest="RPV"
Pipe "p2", Type=Liquid, Src="RPV", Dest = "WW2"

Length(p1) = 5.15
Diameter(p1) = 0.93
k (p1) = 1.5
Depth_in(p1) = 3.2
Depth_out(p1) = 3.2

Length(p2) = 10.0
Diameter(p2) = 0.16
k (p2) = 10.0
Depth_in(p2) = 1.2
Depth_out(p2) = 2.1

InitPg(RPV) = 350000
InitPa(RPV) = 1000.0
InitLiqLevel(RPV) = 6.13
InitTLiq(RPV) = 104.8
InitPg(WW1) = 350001
```
5.3.2. Syntax of MSDL

In this section, a LALR(1) grammar of MSDL is given. If writing an appropriate parser for it, note the left-recursion of some of its productions. As long as a parser generator which is able to handle left recursion, nothing has to be changed.

As commonly used in literature, non terminal symbols are printed italic, terminal symbols are printed boldface.

\[
\begin{align*}
\text{system} & \rightarrow \\
& \quad \text{sname vessels v-geos pipes p-geos inits \_end} \\
& \quad | \epsilon \\
\text{sname} & \rightarrow \\
& \quad \text{name qstring} \\
\text{vessels} & \rightarrow \\
& \quad \text{vessels vessel} \\
& \quad \epsilon \\
\text{vessel} & \rightarrow \\
& \quad \text{vessel qstring, type pv} \\
& \quad \quad \text{| vessel qstring, type gas} \\
\text{v-geos} & \rightarrow \\
& \quad \text{v-geos v-geo} \\
& \quad \epsilon \\
\text{v-geo} & \rightarrow \\
& \quad \text{volume bstring = val} \\
& \quad \quad \text{| crosssection bstring = val} \\
\text{pipes} & \rightarrow \\
& \quad \text{pipes pipe} \\
& \quad \epsilon \\
\text{pipe} & \rightarrow \\
& \quad \text{pipe qstring, type gas, src qstring, dest qstring} \\
& \quad \text{pipe qstring, type liquid, src qstring, dest qstring} \\
& \quad \text{pipe qstring, type event, src qstring, dest qstring} \\
& \quad \text{pipe qstring, type condenser, src qstring, dest qstring}
\end{align*}
\]
pipe qstring, type vent, src qstring, dest qstring

\[ p_{\text{geos}} \rightarrow \]
\[ p_{\text{geos}} p_{\text{geo}} \]
\[ | \epsilon \]

\[ p_{\text{geo}} \rightarrow \]
\[ \text{length bstring} = \text{val} \]
\[ | \text{diameter bstring} = \text{val} \]
\[ | \text{k bstring} = \text{val} \]
\[ | \text{depth.in bstring} = \text{val} \]
\[ | \text{depth.out bstring} = \text{val} \]

\[ \text{inits} \rightarrow \]
\[ \text{inits} \text{ init} \]
\[ | \epsilon \]

\[ \text{init} \rightarrow \]
\[ \text{initpg bstring} = \text{val} \]
\[ | \text{initpa bstring} = \text{val} \]
\[ | \text{initliqulevel bstring} = \text{val} \]
\[ | \text{initliq bstring} = \text{val} \]

\[ s_{\text{end}} \rightarrow \]
\[ \text{end qstring} \]

The terminal symbols are explained in the next section on page 94. The non terminal symbols of the above grammar are mainly self describing. Although a short description is given below:

1. **system**: An overall system can be either the empty system or it can be the sequence of a system name, vessels, vessel geometry definitions, pipes, pipe geometry definitions, initial definitions and a final end statement.

2. **sname**: Every system has a defined, symbolic name which is of the type ‘qstring’.

3. **vessels**: The vessels of a system are either an empty list, or vessels followed by an additional vessel. With this production it is possible to define as many vessels as needed, without any limitations by the compiler.

4. **vessel**: This production defines a vessel either as a gas vessel or a pressure vessel containing both, liquid and gas.

5. **v.geos**: This is the geometrical description production for the vessels. Similar to the vessels production above, it defines just the
list of geometrical descriptions; the actual description follows in
the next production.

6. \textit{v.geo}: The actual geometrical description. Defines volumes and
cross sections of both, gas and pv vessel types.

7. \textit{pipes}: Similar to the vessels production, the pipes in the system
have to be defined.

8. \textit{pipe}: The actual definition for each pipe. Here, the different types
of pipes allowed in the system are defined. Valid types are gas,
liquid, event, condenser or vent pipes. Each pipe has a name
(qstring), a declared type, a source and a destination vessel.
Note that a grammar can not provide consistency checks. These
checks have to be performed in the code generator data structure
described later.

9. \textit{p.geos}: Same as \textit{v.geos}.

10. \textit{p.geo}: For the geometry of a pipe, the following characteristics of
a pipe can be defined: Length, diameter, overall loss coefficient,
and the levels of connection of the vessels.

11. \textit{inits}: Same as \textit{v.geos}.

12. \textit{init}: This productions need to be used for the initialization of a
system. Here, the initial conditions are specified. These are: Ini¬
tial (total) gas pressure, initial (partial) air pressure, initial liquid
level and initial liquid temperature.

13. \textit{s.end}: The end mark of a system description.

\textbf{5.3.3. Lexical Conventions}

MSDL operates on the very common ASCII alphabet. In this context,
a \textit{string} is a finite sequence of symbols from the ASCII character set. In
literature, the expressions \textit{word} or \textit{sentence} are often used as synonyms
for strings. The empty string is denoted as \( \epsilon \).

The tokens used in MSDL are defined as regular definitions. A regu-
lar definition has the form

\[
\begin{align*}
d_1 & \rightarrow r_1 \\
d_2 & \rightarrow r_2 \\
& \quad \vdots \\
d_n & \rightarrow r_n
\end{align*}
\]
where each $d_i$ is a (distinct) symbolic name and the $r_i$ are regular expressions over the previously defined symbols $d_1, \ldots, d_{i-1}$ and the basic symbols of the alphabet. As a convention, we print the names in the regular definitions **boldface** and the basic symbols of the alphabet in **typewriter fonts**. The meaning of the special characters of regular expressions are explained earlier on page 85. The lexical symbols used in MSDL are given in table 5.1. In addition to those, a few more lexical rules have to be defined:

- Comment lines are lines beginning with #.
- White space is either a space character or a tab character.
- The number of blanks is up to the user.
- Lines are ended with a newline character.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>bstring</td>
<td>([^\n]*[^\n]\n)</td>
</tr>
<tr>
<td>condenser</td>
<td>Condenser</td>
</tr>
<tr>
<td>crosssection</td>
<td>Crosssection</td>
</tr>
<tr>
<td>cvent</td>
<td>PCCS</td>
</tr>
<tr>
<td>depth_in</td>
<td>Depth_in</td>
</tr>
<tr>
<td>depth_out</td>
<td>Depth_out</td>
</tr>
<tr>
<td>dest</td>
<td>Dest\ *=</td>
</tr>
<tr>
<td>diameter</td>
<td>Diameter</td>
</tr>
<tr>
<td>end</td>
<td>End</td>
</tr>
<tr>
<td>gas</td>
<td>Gas</td>
</tr>
<tr>
<td>initTliq</td>
<td>InitTliq</td>
</tr>
<tr>
<td>initliqulivel</td>
<td>InitLiqLevel</td>
</tr>
<tr>
<td>initpa</td>
<td>InitPa</td>
</tr>
<tr>
<td>initpg</td>
<td>InitPg</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>length</td>
<td>Length</td>
</tr>
<tr>
<td>liquid</td>
<td>Liquid</td>
</tr>
<tr>
<td>name</td>
<td>Name</td>
</tr>
<tr>
<td>pipe</td>
<td>Pipe</td>
</tr>
<tr>
<td>pv</td>
<td>PV</td>
</tr>
<tr>
<td>qstring</td>
<td>&quot;[^\n]*[^\n]\n)</td>
</tr>
<tr>
<td>src</td>
<td>Src\ *=</td>
</tr>
<tr>
<td>type</td>
<td>Type\ *=</td>
</tr>
<tr>
<td>val</td>
<td>([0-9]+</td>
</tr>
<tr>
<td>vent</td>
<td>Vent</td>
</tr>
<tr>
<td>vessel</td>
<td>Vessel</td>
</tr>
<tr>
<td>volume</td>
<td>Volume</td>
</tr>
</tbody>
</table>

Table 5.1: Lexical symbols and their regular expression representation in MSDL
This chapter describes how the presented simulation language can be used to build up real thermal hydraulic systems. The system described here was actually built at the PSI test site in Villigen (Switzerland) as the Panda system. The Panda facility is well documented and many papers have been published on the topic. The expert reader may wish to consult the original publications [HAB+98b] [HAB+98c] [HDLA] [Hug97] [Hug95] [HAB+98a] [AD99] [LAA+99] [FDA+97] [YDH+97] [DHA+96] [AF99] [DHA+95] [BDH+99] for getting a more detailed understanding.

The code developed in this work is tested against the Panda Test P3. The following section on the Panda P3 physical test is mainly quoted from [HAB+98b]. The following chapter describes the setup for the computational approach, which then is compared to the physical test. Finally, the differences are discussed.
6.1. The PANDA Facility

PANDA is a large-scale facility for the investigation of passive ALWR containment phenomena and simulation of system response. It was originally designed and used as a large-scale integrated containment test facility located at PSI in Switzerland. The facility was first used for SBWR related tests and later for the follow-up ESBWR design.

PANDA has a modular structure of cylindrical vessels interconnected by piping (as shown in the schematic in figure 6.1). The various mixing and natural circulation phenomena taking place in the containment are complex, depend on the particular geometry and dimensions of the containment building, and simple linear geometric scaling would have produced serious scaling distortions. The general philosophy adopted for PANDA was to allow multidimensional effects to take place by dividing the main containment compartments (DW and WW) in two. This allows spatial distribution effects to develop. A variety of controlled boundary conditions can be imposed during the experiments to study mixing phenomena under well-established conditions. Parametric or sensitivity experiments conducted under well-controlled boundary conditions can provide more valuable data for code qualification than experiments where the mixing phenomena are distorted by the scale of the facility or other reasons.

The facility is an approximately \( \frac{1}{25} \) volumetric scale, full-scale height simulation of a SBWR containment system. Following this general experimental design philosophy, the various containment volumes are simulated by six cylindrical pressure vessels. One of these represents the Reactor Pressure Vessel (RPV) and is equipped with programmable 1.5 MW electric heaters simulating the history of core decay heat generation. Two other vessels may represent the Drywell (DW), another two the Wetwell (WW), and one a Gravity Driven Cooling System (GDCS) pool or another containment volume. Four rectangular pools open to the atmosphere are located on top of the facility. For the SBWR and ESBWR tests, these contained three Passive Containment Coolers (PCC) connected to the DW and one Isolation Condenser (IC) connected to the RPV. The modular facility arrangement provides the flexibility needed to investigate a variety of containment designs.

The three PCCS condensers used for the SBWR tests are connected to the two DW vessels. The fact that the three PCC units are connected to two DW vessels allows asymmetric behavior and creates flows be-
Figure 6.1: PANDA test facility schematic
tween the two DW vessels. Such an asymmetry occurs also with equal flow resistance from the RPV to the two DW volumes when all three PCCS condensers are in operation.

The SBWR and ESBWR containment concepts rely on vacuum breakers (VB) installed between the DW and WW. Their function is not to allow the WW pressure to exceed DW pressure by a certain margin. There are two vacuum breakers connecting the two DWs to the two WWs in PANDA. The operation of the actual vacuum breakers is simulated in PANDA by control valves. These are opened and closed by the facility control system when the measured differential pressure between the DWs and the WWs exceeds an upper and a lower limit, respectively.

6.2. The PANDA P3 Physical Test

6.2.1. General Objectives

The main goal of the P-Series integral system tests was to make a significant contribution to the technology enhancement of passive safety systems of the ESBWR. Several new containment features were experimentally and analytically investigated. The general objectives of the experimental part were to

- Investigate the effect of new design features on the long-term containment response after a LOCA
- Demonstrate that the decay heat removal systems operate under different accident scenarios
- Provide a data base for computer code assessment and analysis of particular phenomena

The approach for reaching these objectives was to perform a series of eight transient system tests in the existing and accordingly modified thermal-hydraulic large-scale test facility PANDA.

6.2.2. General Description and Purpose of the Test

The test P3 was the "PCCS Start-up" PANDA transient system test. The test investigated the start-up behavior of the passive decay heat removal system under bounding non-condensible gas concentrations. The drywell and the PCC units were initially filled with air. Only the
two PCCs connected to DW2 were operable and the entire break flow was directed to DW2. Consequently, DW1 represented a "dead" volume. The initial pressure was approximately 1.3 bar. The purpose of this 8-hour test was to provide data on the PCCS start-up under most challenging conditions.

6.2.3. Test Facility Configuration

For the test P3 the PANDA facility was configured for simulating the ESBWR post-LOCA configuration as follows:

1. The RPV supplies steam into DW2.
2. The RPV heater power was held constant throughout test at 0.85 MW.
3. The IC unit was isolated from the system.
4. Only PCC2 and PCC3, which both were connected to DW2, were in operation, i.e. took feed flow from the DW, vented non-condensible (and steam) to the suppression pool and drained condensate back to the RPV. PCC1 was isolated from the system.
5. During preconditioning the PCC Pools 2 and 3 (henceforth: PCC2/3 Pool) were filled with saturated water and then isolated from each other. During the test the PCC2/3 Pools remained isolated from each other and no water was added to or drained from the pools. (The IP Pool and PCC Pool 1 were not involved in Test P3.)
6. The only lines of communication between the DW and WW were through the Vacuum Breakers (VB) (when the WW pressure exceeded DW pressure sufficiently to open the VB) and the Main Vent Lines (MVL) (which were submerged in the SC Pool).
7. The GDCS gas space and the WW gas spaces were interconnected by using auxiliary steam system lines as pressure equalization lines.
8. The GDCS Drain Line was closed. (As the GDCS Pool was already empty at the begin of Test P3. There was a check valve preventing back-flow from the RPV, the GDCS Drain Line had no function and therefore remained closed during the test.)
9. The equalization lines between the RPV and suppression pools were closed.

6.2.4. Test Specific Information

Test Time

| Start of Test P3: | Jan 21, 1998 / 15:28:48 | 0 sec |
| End of Test P3:   | Jan 21, 1998 / 23:30:13 | 28885 sec |
| Test Duration     | 8 hrs 1 min 25 sec     | 28885 sec |
| Test Period       | 0 to 28885 sec         |        |

6.2.5. RPV Power Curve Analysis

The analysis of the RPV heating power consisted in comparing the measured power with the theoretical value defined in Test Specification ALPHA-703 [Hug97]. The power during the test P3 remained constant (0.85 MW).

- Power analysis period: 20 : 28883 sec
- Maximum negative deviation: -1.4 kW
- Maximum positive deviation: 13.4 kW
- Standard deviation: 0.9 kW
- Power curve tolerance: ±25 kW

6.2.6. Test Initial Conditions

The system pressure for Test P3 was approximately 1.3 bar. The DW contained humid air at room temperature. The initial conditions for Test P3 are presented in table 6.1. The “Average Value” represents the arithmetic mean over the one minute measurement period before connecting the vessels to the required test configuration. The column “Process ID” references the actual probe within the test facility. The instruments and their location can be found in the original PANDA Documentation [HDLA].
### PANDA P3 Initial Conditions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Process ID</th>
<th>Unit</th>
<th>Average Value</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Environment Parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atmospheric Pressure</td>
<td>MP.EN</td>
<td>bar</td>
<td>0.99</td>
<td>0.000</td>
</tr>
<tr>
<td>Temperature of saturation</td>
<td>(T_{sat,EN})</td>
<td>C</td>
<td>99.1</td>
<td></td>
</tr>
<tr>
<td><strong>RPV Parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Pressure</td>
<td>MP.RP1</td>
<td>bar</td>
<td>1.29</td>
<td>0.003</td>
</tr>
<tr>
<td>Fluid Temperatures:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial Average</td>
<td>T_{F,mean} (RP)</td>
<td>C</td>
<td>104.8</td>
<td>1.375</td>
</tr>
<tr>
<td>Local</td>
<td>MTF.RP.1</td>
<td>C</td>
<td>105.0</td>
<td>0.045</td>
</tr>
<tr>
<td></td>
<td>MTF.RP.2</td>
<td>C</td>
<td>103.2</td>
<td>0.022</td>
</tr>
<tr>
<td></td>
<td>MTF.RP.3</td>
<td>C</td>
<td>103.6</td>
<td>0.021</td>
</tr>
<tr>
<td></td>
<td>MTF.RP.4</td>
<td>C</td>
<td>106.6</td>
<td>0.037</td>
</tr>
<tr>
<td></td>
<td>MTF.RP.5</td>
<td>C</td>
<td>105.5</td>
<td>0.034</td>
</tr>
<tr>
<td>Water level</td>
<td>ML.RP.1</td>
<td>m</td>
<td>6.13</td>
<td>0.003</td>
</tr>
<tr>
<td><strong>Drywell Parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Pressure</td>
<td>MP.D1</td>
<td>bar</td>
<td>1.29</td>
<td>0.001</td>
</tr>
<tr>
<td>Air Partial Pressure</td>
<td>MPG.D1.1</td>
<td>bar</td>
<td>1.24</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>MPG.D1.3</td>
<td>bar</td>
<td>1.24</td>
<td>0.002</td>
</tr>
<tr>
<td>Air Partial Pressure</td>
<td>MPG.D2.1</td>
<td>bar</td>
<td>1.25</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>MPG.D2.2</td>
<td>bar</td>
<td>1.41</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>MPG.D2.3</td>
<td>bar</td>
<td>1.24</td>
<td>0.004</td>
</tr>
<tr>
<td>Gas Temperatures:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial Average</td>
<td>T_{G,mean} (D1)</td>
<td>C</td>
<td>27.6</td>
<td>0.057</td>
</tr>
<tr>
<td>Local</td>
<td>MTG.D1.1</td>
<td>C</td>
<td>27.6</td>
<td>0.024</td>
</tr>
<tr>
<td></td>
<td>MTG.D1.2</td>
<td>C</td>
<td>27.6</td>
<td>0.026</td>
</tr>
<tr>
<td></td>
<td>MTG.D1.3</td>
<td>C</td>
<td>27.5</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>MTG.D1.4</td>
<td>C</td>
<td>27.6</td>
<td>0.023</td>
</tr>
<tr>
<td></td>
<td>MTG.D1.5</td>
<td>C</td>
<td>27.6</td>
<td>0.031</td>
</tr>
<tr>
<td></td>
<td>MTG.D1.6</td>
<td>C</td>
<td>27.6</td>
<td>0.039</td>
</tr>
<tr>
<td>Spatial Average</td>
<td>T_{G,mean} (D2)</td>
<td>C</td>
<td>27.4</td>
<td>0.084</td>
</tr>
<tr>
<td>Local</td>
<td>MTG.D2.1</td>
<td>C</td>
<td>27.5</td>
<td>0.056</td>
</tr>
<tr>
<td></td>
<td>MTG.D2.2</td>
<td>C</td>
<td>27.5</td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>MTG.D2.3</td>
<td>C</td>
<td>27.3</td>
<td>0.020</td>
</tr>
<tr>
<td></td>
<td>MTG.D2.4</td>
<td>C</td>
<td>27.4</td>
<td>0.022</td>
</tr>
<tr>
<td></td>
<td>MTG.D2.5</td>
<td>C</td>
<td>27.4</td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>MTG.D2.6</td>
<td>C</td>
<td>27.5</td>
<td>0.023</td>
</tr>
<tr>
<td>Water Level</td>
<td>ML.D1</td>
<td>m</td>
<td>0.03</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>ML.D2</td>
<td>m</td>
<td>0.04</td>
<td>0.000</td>
</tr>
<tr>
<td><strong>Wetwell Parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Pressure</td>
<td>MPS1</td>
<td>bar</td>
<td>1.30</td>
<td>0.002</td>
</tr>
<tr>
<td>Air Partial Pressure</td>
<td>MPG.S1</td>
<td>bar</td>
<td>0.79</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>MPG.S2</td>
<td>bar</td>
<td>0.82</td>
<td>0.001</td>
</tr>
<tr>
<td>Liquid Temperatures:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial Average</td>
<td>T_{L,mean} (S1)</td>
<td>C</td>
<td>80.9</td>
<td>0.127</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Variable</th>
<th>Process ID</th>
<th>Unit</th>
<th>Average Value</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>MTL.S1.1</td>
<td>C</td>
<td>81.1</td>
<td>0.039</td>
</tr>
<tr>
<td></td>
<td>MTL.S1.2</td>
<td>C</td>
<td>81.0</td>
<td>0.040</td>
</tr>
<tr>
<td></td>
<td>MTL.S1.3</td>
<td>C</td>
<td>81.1</td>
<td>0.028</td>
</tr>
<tr>
<td></td>
<td>MTL.S1.4</td>
<td>C</td>
<td>80.8</td>
<td>0.026</td>
</tr>
<tr>
<td></td>
<td>MTL.S1.5</td>
<td>C</td>
<td>80.8</td>
<td>0.036</td>
</tr>
<tr>
<td></td>
<td>MTL.S1.6</td>
<td>C</td>
<td>80.7</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td>Spatial Average</td>
<td>$T_{L,mean}(S2)$</td>
<td>C</td>
<td>80.8</td>
</tr>
<tr>
<td></td>
<td>MTL.S2.1</td>
<td>C</td>
<td>80.7</td>
<td>0.035</td>
</tr>
<tr>
<td></td>
<td>MTL.S2.2</td>
<td>C</td>
<td>80.8</td>
<td>0.031</td>
</tr>
<tr>
<td></td>
<td>MTL.S2.3</td>
<td>C</td>
<td>80.8</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td>MTL.S2.4</td>
<td>C</td>
<td>80.8</td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>MTL.S2.5</td>
<td>C</td>
<td>80.8</td>
<td>0.031</td>
</tr>
<tr>
<td></td>
<td>MTL.S2.6</td>
<td>C</td>
<td>80.7</td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>Spatial Average</td>
<td>$T_{G,mean}(S1)$</td>
<td>C</td>
<td>79.3</td>
</tr>
<tr>
<td></td>
<td>MTG.S1.1</td>
<td>C</td>
<td>79.1</td>
<td>0.033</td>
</tr>
<tr>
<td></td>
<td>MTG.S1.2</td>
<td>C</td>
<td>79.2</td>
<td>0.032</td>
</tr>
<tr>
<td></td>
<td>MTG.S1.3</td>
<td>C</td>
<td>79.2</td>
<td>0.036</td>
</tr>
<tr>
<td></td>
<td>MTG.S1.4</td>
<td>C</td>
<td>79.3</td>
<td>0.065</td>
</tr>
<tr>
<td></td>
<td>MTG.S1.5</td>
<td>C</td>
<td>79.3</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td>MTG.S1.6</td>
<td>C</td>
<td>80.0</td>
<td>0.045</td>
</tr>
<tr>
<td></td>
<td>Spatial Average</td>
<td>$T_{G,mean}(S2)$</td>
<td>C</td>
<td>79.1</td>
</tr>
<tr>
<td></td>
<td>MTG.S2.1</td>
<td>C</td>
<td>79.1</td>
<td>0.044</td>
</tr>
<tr>
<td></td>
<td>MTG.S2.2</td>
<td>C</td>
<td>79.2</td>
<td>0.030</td>
</tr>
<tr>
<td></td>
<td>MTG.S2.3</td>
<td>C</td>
<td>79.1</td>
<td>0.076</td>
</tr>
<tr>
<td></td>
<td>MTG.S2.4</td>
<td>C</td>
<td>79.1</td>
<td>0.074</td>
</tr>
<tr>
<td></td>
<td>MTG.S2.5</td>
<td>C</td>
<td>79.1</td>
<td>0.053</td>
</tr>
<tr>
<td></td>
<td>MTG.S2.6</td>
<td>C</td>
<td>79.3</td>
<td>0.049</td>
</tr>
<tr>
<td></td>
<td>Water Level</td>
<td>ML.S1</td>
<td>m</td>
<td>3.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ML.S2</td>
<td>m</td>
<td>3.99</td>
</tr>
</tbody>
</table>

**GDCS Parameters**

| Total Pressure | MP.GD | bar | 1.30 | 0.001 |

**Fluid Temperatures:**

| Spatial Average | $T_{F,mean}(GD)$ | C | 56.1 | 3.363 |
| Local           | MTF.GD.1        | C | 56.1 | 0.064 |
|                 | MTF.GD.2        | C | 55.8 | 0.202 |
|                 | MTF.GD.3        | C | 56.0 | 0.081 |
|                 | MTF.GD.4        | C | 56.7 | 0.074 |
|                 | MTF.GD.5        | C | 55.8 | 0.124 |
|                 | MTF.GD.6        | C | 56.3 | 0.102 |
|                 | MTF.GD.7        | C | 55.7 | 0.260 |
### Table 6.1: Test P3 initial conditions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Process ID</th>
<th>Unit</th>
<th>Average Value</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Water Level</td>
<td>ML.GD</td>
<td>m</td>
<td>0.07</td>
<td>0.001</td>
</tr>
<tr>
<td>PCC2 Parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Water Temperatures:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial Average</td>
<td>$T_{R,\text{mean}}(U2)$</td>
<td>C</td>
<td>95.0</td>
<td>0.047</td>
</tr>
<tr>
<td>Local</td>
<td>MTF.U2.1</td>
<td>C</td>
<td>95.0</td>
<td>0.035</td>
</tr>
<tr>
<td></td>
<td>MTF.U2.2</td>
<td>C</td>
<td>94.9</td>
<td>0.049</td>
</tr>
<tr>
<td></td>
<td>MTF.U2.3</td>
<td>C</td>
<td>95.1</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>MTF.U2.4</td>
<td>C</td>
<td>95.0</td>
<td>0.036</td>
</tr>
<tr>
<td></td>
<td>MTF.U2.5</td>
<td>C</td>
<td>95.0</td>
<td>0.045</td>
</tr>
<tr>
<td></td>
<td>MTF.U2.6</td>
<td>C</td>
<td>95.0</td>
<td>0.023</td>
</tr>
<tr>
<td></td>
<td>MTF.U2.7</td>
<td>C</td>
<td>94.9</td>
<td>0.026</td>
</tr>
<tr>
<td>Water Level</td>
<td>ML.U2</td>
<td>m</td>
<td>4.41</td>
<td>0.001</td>
</tr>
<tr>
<td>PCC3 Parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Water Temperatures:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spatial Average</td>
<td>$T_{R,\text{mean}}(U3)$</td>
<td>C</td>
<td>95.1</td>
<td>0.058</td>
</tr>
<tr>
<td>Local</td>
<td>MTF.U3.1</td>
<td>C</td>
<td>95.0</td>
<td>0.062</td>
</tr>
<tr>
<td></td>
<td>MTF.U3.2</td>
<td>C</td>
<td>95.0</td>
<td>0.037</td>
</tr>
<tr>
<td></td>
<td>MTF.U3.3</td>
<td>C</td>
<td>95.0</td>
<td>0.086</td>
</tr>
<tr>
<td></td>
<td>MTF.U3.4</td>
<td>C</td>
<td>95.1</td>
<td>0.076</td>
</tr>
<tr>
<td></td>
<td>MTF.U3.5</td>
<td>C</td>
<td>95.0</td>
<td>0.098</td>
</tr>
<tr>
<td>Water Level</td>
<td>ML.U3</td>
<td>m</td>
<td>4.40</td>
<td>0.001</td>
</tr>
</tbody>
</table>

### 6.2.7. Test results: Main Observations

In test P3, the most challenging conditions for PCCS start-up were investigated. The drywell and the passive containment coolers were initially filled with air and only the two PCC units connected to DW2 were operable. Immediately after test initiation, the PCCs started to vent the air from the drywell to the wetwell. This is confirmed by the measured air partial pressure in DW2 (figure 6.9) and in WW1 and WW2 (figure 6.10). Also the PCC vent line phase indicators showed frequent venting right from the beginning of the test. During the first 5000 seconds into the test the system was heated up and pressurized. The system pressure increased sharply from 1.3 bar to about 2.7 bar during the heat-up phase (figure 6.2). As can be seen from figures 6.5 and 6.6 the gas temperatures in DW2 and in the upper region of DW1 showed a uniform increase. The temperatures in the lower region of DW1 (below the DW1 to DW2 connecting line) remained at low level. This was because DW1 represented a “dead” volume (no steam injec-
tion to DW1, no PCC operable) and therefore, air was accumulated in the lower region of DW1. This behavior was also confirmed by the measured DW air partial pressures shown in figure 6.9. It can also be seen from this figure that approximately 4 hours after the start of the test, almost all the air from DW2 and from the upper part of DW1 was purged to the wetwell.

After the heat-up phase was completed, only two PCC units operable were not sufficient to condense all the steam from the drywell. Therefore, the DW to WW pressure difference increased until the main vents began to clear. Because the experiment was run with constant heating power instead of following the decay heat curve, the main vents periodically cleared throughout the test. This is confirmed by the main vent phase indicators and is also reflected in the main vent line gas temperatures. Consequently, there was a continuous energy transfer from drywell to wetwell. The heat-up of the upper part of the WW water volume and the WW gas space was documented by the WW temperatures (figures 6.7 and 6.8). The heat-up in the two WW vessels was not symmetric. This can be explained by the asymmetric test conditions (total steam flow to DW2 and only PCCs connected to DW2 operable). To the end of the test the temperatures in WW2 were roughly 2°C higher than in WW1. From the measured gas temperatures in the main vent lines it was concluded that there was a higher flow rate through vent line 2 than through vent line 1. The vent lines of the operating PCC units were both connected to WW2. These lines also represented a path for energy transfer to the wetwell.

In accordance with the continuous temperature increase in the wetwell also the WW pressure and therefore also the system pressure increased slowly by about 0.02 bar/hr to the end of the test.

During the heat-up phase steam from the RPV condensed mainly on the cold drywell walls. The condensate which was collected at the bottom of the drywell caused a loss of inventory in the RPV during the early phase of test (figure 6.11. Later, when the main portion of the steam was condensed in the PCCs and the condensate was returned to the RPV the drop in RPV level was much less. Figure 6.4 shows the total PCC condensate flow rate which remained constant after the heat-up phase. The two PCC units took the same heat load. As shown in figure 6.3 both PCCs got the same feed flow and the drop in water level was also the same for both PCC pools.
6.2. The PANDA P3 Physical Test

Figure 6.2: DW1, WW1 and RPV pressures

Figure 6.3: PCC feed flow rates
Figure 6.4: Total condensate flow rate

Figure 6.5: DW1 gas temperature
Figure 6.6: DW2 gas temperature

Figure 6.7: WW1 gas, pool surface and liquid temperature
Figure 6.8: WW2 gas, pool surface and liquid temperature

Figure 6.9: DW air partial pressure
6.2. The PANDA P3 Physical Test

**Figure 6.10: WW air partial pressure**

![Graph showing WW air partial pressures over time](image)

**Figure 6.11: RPV collapsed level**

![Graph showing RPV collapsed level over time](image)
6.3. The PANDA P3 Simulation Setup

6.3.1. Program preparation and setup

The set up of a simulation program for the presented configuration consists mainly of a high level description of the system, using MSDL. The MSDL program entails five parts for declaring vessels and pipes, for the appropriate geometrical characteristics, and for the initial conditions, respectively.

In the first part given below, the vessels called RPV, DW1, DW2, WW1, WW2, and GDCS, respectively are declared with their appropriate types. A ”Gas” type describes a vessel which consists of a gas space only, where a PV Type consists of a gas space and a liquid space, with a virtual interface in between. In the second part of the listing, the Vessels can be referenced by their names, which makes the reading of such a description relatively easy.

```
Listing 6.1: Part 1: Declaration of Vessels
# Description file for simulation setup of P3 test
# Created Jun 14, 2004
#
Name "p3"
#
# Names and geometrical description of vessels
#
Vessel "RPV", Type = PV
Vessel "DW1", Type = Gas
Vessel "DW2", Type = Gas
Vessel "WW1", Type = PV
Vessel "WW2", Type = PV
Vessel "GDCS", Type = Gas
```

In the second part, the volumes and the cross-sections of the vessels are defined. The values were taken from the PANDA documentation. In the listing below, for example, the Drywell 1 is declared to have a volume of 90 $m^3$, with a cross section of 7.161972 $m^2$. The models used in this work assume perfect cylindrical geometry for all vessels.

```
Listing 6.2: Part 2: Geometrical Properties of Vessels
Volume (RPV) = 22.8
Crosssection (RPV) = 1.2271846
Volume (DW1) = 90.0
Crosssection (DW1) = 7.161972
Volume (DW2) = 90.0
```
Crosssection(DW2) = 7.161972
Volume(WW1) = 117.0
Crosssection(WW1) = 12.5663711
Volume(WW2) = 117.0
Crosssection(WW2) = 12.5663711
Volume(GDCS) = 17.6
Crosssection(GDCS) = 3.14159

The third part is more or less identical with the first part. Here, all pipes and condensers, vacuum breakers and other elements are declared which are characterized by a mass flow. Again, the naming of the elements are according to the ALPHA documentation. Beside the name of the element, its source and destination vessel(s) must be specified. Note, that a PCC element has two destination vessels: one for the condensate, and one for the remaining non condensed steam and the non-condensibles like air. As seen in the listing below, it is possible to declare parallel pipes or condensers. This is done with the PCCS2 and PCCS3 in this setup.

Listing 6.3: Part 3: Declaration of Pipes

```bash
# Name of pipes and pccs according to alpha report

# Pipe "MSL2", Type = Gas, Src="RPV", Dest="DW2"
Pipe "PCCS2", Type = PCCS, Src="DW2", Dest="WW2", Dest_L = "RPV"
Pipe "PCCS3", Type = PCCS, Src="DW2", Dest="WW2", Dest_L = "RPV"
Pipe "VENT1", Type = Vent, Src="DW1", Dest="WW1"
Pipe "VENT2", Type = Vent, Src="DW2", Dest="WW2"
Pipe "P1", Type = Gas, Src="DW1", Dest="DW2"
Pipe "P2", Type = Gas, Src="WW1", Dest="WW2"
Pipe "P3", Type = Liquid, Src="WW1", Dest="WW2"
Pipe "AUX1", Type = Gas, Src="GDCS", Dest="WW2"
```

In the next section, the characteristics of the lines are defined. Pure gas lines are only characterized by their hydraulic diameter $D_n$, their length $L$ and their overall friction factor $k$. For the liquid pipes, the depth of the connecter below the surface in the liquid becomes important. Last, but not least, the depth of the vent line into the wet well becomes important for the condensers.

Listing 6.4: Part 4: Geometrical Properties of Pipes

```bash
# Geometrical characteristics of the lines
```
In the last section finally, the initial conditions for all vessels must be specified. These include the pressures and the partial air pressures of all vessels, the level of the liquids and the initial temperature of the liquids in the PVs. These values are taken from the P3 Test documentation.

### Listing 6.5: Part 5: Initial Conditions

```python
# Initial conditions for vessels

InitPg(RPV) = 129000.0
```
6.3. The PANDA P3 Simulation Setup

6.3.2. Generation of Fortran Code

The file is stored in the file p3.sim. The .sim file now can be translated into Fortran code using the compiler discussed earlier in this work. The very compact p3.sim file (111 lines of code) is converted into a complex Fortran code of 1387 lines of code. This code, however, does not include the code of the modules described in the previous chapters. The program p3.f only contains the logic of how to call these modules and how to use the output of one module as input of another module.

The compilation time for this task is basically negligible, i.e. 0.01 seconds of user time on a modern laptop computer at a clock speed of 1.2 GHz.

6.3.3. Compilation and Execution

Compilation

The code generated by the MSDL compiler is now compiled using the g77 Fortran compiler which is based on gcc version 3.3.1. There is a makefile provided with the code which keeps all modules and libraries up-to-date such that no obsolete code is linked or compiled. In its standard configuration, the profiler option is activated which causes the compiler to generate extra code for writing profile information suitable for the analysis program gprof. This analysis is presented below. Gprof calculates the amount of time spent in each routine. Next,
these times are propagated along the edges of the call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle. This information may be very useful for resolving performance problems of the resulting program.

**Run-Time Profile**

The table 6.2 shows a flat profile of the top functions of the simulation run. The table shows all functions which are consuming more than 1% of the CPU time for the overall run. The total run time of this program was 82.86 seconds on a modern laptop computer.

The meaning of the columns is as follows:

- **% time**: the percentage of the total running time of the program used by this function.
- **total seconds**: a running sum of the number of seconds accounted by this function and those listed above it.
- **self seconds** the number of seconds accounted for by this function alone. This is the major sort for this listing.
- **self ms/call**: the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
- **name** the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

It is seen clearly, that the main CPU time is used to calculate steam and mass properties. Due to this, a highly optimized library of steam table functions could reduce the overall runtime considerably.

The overall runtime was 82.86 seconds, as seen in the table 6.2. The overall program is represented by the init_ function, which is called only once.

**6.3.4. Result files**

The results of the simulation run are stored in text files of the form

---

1Fujitsu-Siemens Lifebook E2010, Mobile Intel Pentium Processor running at 2.20 GHz, cache size 512 kB, 512 MB of RAM
### Table 6.2: Top cpu time consuming functions in P3 Simulation

<table>
<thead>
<tr>
<th>% time</th>
<th>total seconds</th>
<th>self seconds</th>
<th>self calls</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.66</td>
<td>10.49</td>
<td>10.49</td>
<td>33396799</td>
<td>comt2_</td>
</tr>
<tr>
<td>12.47</td>
<td>20.82</td>
<td>10.33</td>
<td>40961254</td>
<td>comt1_</td>
</tr>
<tr>
<td>10.58</td>
<td>29.59</td>
<td>8.77</td>
<td>48553088</td>
<td>__g77_masterfun_psatt</td>
</tr>
<tr>
<td>8.81</td>
<td>36.89</td>
<td>7.30</td>
<td>27967218</td>
<td>__g77_masterfun_tsatp</td>
</tr>
<tr>
<td>5.68</td>
<td>41.60</td>
<td>4.71</td>
<td>10209295</td>
<td>__g77_masterfun_visl</td>
</tr>
<tr>
<td>5.09</td>
<td>45.81</td>
<td>4.21</td>
<td>3308766</td>
<td>func_</td>
</tr>
<tr>
<td>4.15</td>
<td>49.26</td>
<td>3.44</td>
<td>9926298</td>
<td>__g77_masterfun_condv</td>
</tr>
<tr>
<td>4.04</td>
<td>52.60</td>
<td>3.35</td>
<td>19259274</td>
<td>hpt1_</td>
</tr>
<tr>
<td>3.57</td>
<td>55.56</td>
<td>2.96</td>
<td>18393214</td>
<td>vpt1_</td>
</tr>
<tr>
<td>3.27</td>
<td>58.27</td>
<td>2.71</td>
<td>13127562</td>
<td>hpt2_</td>
</tr>
<tr>
<td>2.82</td>
<td>60.62</td>
<td>2.34</td>
<td>16465915</td>
<td>vpt2_</td>
</tr>
<tr>
<td>2.68</td>
<td>62.84</td>
<td>2.22</td>
<td>494556</td>
<td>dots_</td>
</tr>
<tr>
<td>2.12</td>
<td>64.59</td>
<td>1.76</td>
<td>3803322</td>
<td>cpp2_</td>
</tr>
<tr>
<td>2.08</td>
<td>66.31</td>
<td>1.72</td>
<td>3308766</td>
<td>cpp1_</td>
</tr>
<tr>
<td>1.76</td>
<td>67.78</td>
<td>1.46</td>
<td>2732102</td>
<td>fex_</td>
</tr>
<tr>
<td>1.16</td>
<td>68.73</td>
<td>0.96</td>
<td>21303754</td>
<td>stoda_</td>
</tr>
<tr>
<td>1.05</td>
<td>69.61</td>
<td>0.87</td>
<td>18393214</td>
<td>vptL</td>
</tr>
<tr>
<td>0.00</td>
<td>82.86</td>
<td>0.00</td>
<td>1</td>
<td>init_</td>
</tr>
</tbody>
</table>

Table 6.2: Top cpu time consuming functions in P3 Simulation
The example above shows the mass of steam in the RPV, which initially is about 13.5 kg and increases up to 56.7 kg after 29000 seconds.

Beside the data files, which represent all the dependent variables of the model, a comprehensive log file is being written during the execution of the program. Thus, some very interesting information of the behavior of the system can be explored. These variables include the condensation rate of the PCCS, the quality of the mass flow in the vent lines, the step size of the actual integration step, and much more.

6.3.5. Discussion of results

The main observations of the general system behavior of the physical tests have been confirmed in two ways. Firstly by comparing the numerical results with experimental data from one of the PANDA-experiments [HAB+98b] carried out at PSI. Secondly, the results are compared against another, independent computer simulation carried out using RELAP5. Numerical and experimental results can be depicted for the total pressure evolution in DW2 and WW2 from figure 6.12.

Obviously there are some noticeable differences between the simulation and the experimental data. During the first part of the test, the simulated pressures rise at a faster rate than in the experiment. This can be explained by two major facts as follows:

Firstly, in the pressure vessel modules, no condensation of steam at the containment walls has been modeled. This is in contrast to the observation in the actual P3 test, as during the heat-up phase (up to about 2'000 seconds in the test) steam from the RPV condenses mainly on the cold drywell walls. Thus, basically no steam is forwarded to the condensers and therefore no condensate flow can be observed in the early startup phase of the test. In the experiment, however, the
condensate from the cold DW walls was collected on the bottom of the drywell. This results in a less steep increase of the pressure in the DW.

In the simulation, however, the condensers took up operation as soon as some steam entered the DWs, just after about 60 seconds in the test. This is due to the fact that the models of the condensers are based on the assumption, that the mass flow is purely pressure driven. Therefore as soon as some minimal amount of steam enters the DW, where it is not condensed on the walls, will start the operation of the condensers immediately. As a consequence, there will be an instantaneous increase of the pressure. In contrast in the experiment condensation starts slowly, mainly caused by the condensation on the containment walls.

This startup behavior can be simulated by either reducing the reactor power, or by artificially removing some steam out the DW during the early startup phase. Both possibilities are intended to remove some energy and steam out of the system. For this numerical experiment, the mass of liquid collected on the bottom of the DW can be neglected.

Figure 6.13 shows clearly the adjustment of the increase of the pressure in the DWs. For this experiment, the reactor power was reduced to \(0.20 \times 10^6 W + 75.0 \text{ W/s}\) for the first 4000 seconds. This slow increasing of RPV heating power matches the increasing pressure in the DW in the experiment and in the numerical simulation.

On the other hand, due to the reduced RPV power, the pressurization of the system starts a little later, as it obviously takes longer for the liquid inside the RPV to saturate.

Secondly, the differences are partly also due to the fact, that a well mixed gas in all vessels is assumed in the simulations that delays the vapor entrance into the condensers. In the long term, the simulated pressures are always well below the experimental values. This considerably reduced condensation rate is due to the large amount of air that remains trapped mainly in DW1, which acts as a dead-end volume. In the test, however, air is only accumulated in the lower part of DW1. The remainder of the air was purged to the wetwell resulting in an increase of the system pressure.

As mentioned before, the numerical models are based on the assumption of a well mixed gas inside the vessels. Experimental data from PANDA [HAB+98b], however, reveal, that the partial air pressures remain even during longer runs very much space dependent.
As can be depicted from figure 6.9 the partial air pressures of DW1 differ noticeably from top to bottom.

The experimentally verified vertical space dependence of the partial air pressures within DW1 will cause more air to be purged to the WW as in the numerical simulation. As a consequence the pressure in the numerical simulation will always lack behind the measured pressure and will therefore be lower by a certain amount. As a consequence, depending on the amount of air in the WW, the calculated pressure will differ from the experimentally measured values by a certain amount.

The very same comparison with the PANDA P3-experiment was also carried out by L. Batet et al. [BCnCR99] in another simulation study done with the code RELAP5/Mod3.2. The containment pressure of this comparison are visualized in figure 6.14.

It has to be noted how well the results obtained with RELAP5 match the ones presented in this work.

Batet and his co-authors find similar reasons for explaining the discrepancy between their numerical values and the experimental PANDA-results.

Immediately after the beginning the PCCs started to vent air form DW2 to WW2. A few 100 seconds after start no more air is left in the RPV. About 2500 seconds after initiation, all air is vented out of DW2. The total condensate flow rate from the condensers back to the RPV can be seen from figure 6.15

The gas temperature of DW1 and DW2 show a higher increase in DW2, as hot gas is fed from the RPV only to DW2, and there after vented from DW2 to the PCCs. Thus, DW1 is a dead end and its temperature will basically not increase due to hot gas injections, but only due to the pressure increase. On the other hand, the gas in DW2 is vented down to WW1, which causes the liquid temperature in WW1 to rise. The temperature behavior of the DWs is shown in figures 6.16 (DW1) and 6.17 (DW2), respectively.

As above, the results of figure 6.17 have to be compared to the ones obtained by L. Batet et al. [BCnCR99] in figure 6.18. Again, it has to be noted how well and consistent the results of L. Batet et al. match the results of the ones presented in this thesis.

It should be noted, than in the work of Batet the results based on TRAC (models based on PDEs) came by far closer to experimental results than the ones the authors obtained by applying RELAP5 (models based on ODEs as also used in the models of this thesis).
6.3. The PANDA P3 Simulation Setup

Figure 6.12: Total pressure evolution calculated and measured

Figure 6.13: Total pressure evolution calculated and measured with reduced RPV heating power for the first 4,000 seconds into the test
Figure 6.14: Total pressure evolution calculated and measured by L. Batet et al.

Figure 6.15: Condensate flow rate calculated and measured
Figure 6.16: Gas temperature in DW1 calculated and measured

Figure 6.17: Gas temperature in DW2 calculated and measured
This leads to the conclusion, that one of the reasons for the deviation of experimental results from simulated results is probably due to the fact that the underlying assumption for all the models of all components was based on almost steady-state conditions whereas the experiments to be simulated where showing severe transients at least in some initial phases in some modules.

Therefore, the assumption of using ODEs instead of PDEs was probably not completely adequate for describing transport phenomena as they would be important to describe the dynamics within several models. In particular, the modeling of phenomena going on within the condensers need much more detailed descriptions which necessitates the use of PDEs.

Probably for the same reasons, the results of the presented work should be seen in the same way as they are also based on ODEs, which do not allow for a sufficiently realistic description of transport phenomena.

As indicated earlier, the kind of the models are useful for almost steady-state situations, but this assumption comes to a limit in some of
the test cases where also faster transients play an important role. This is also confirmed by the fact that the measurements in the experiments show a strong space dependency at all times of air and steam which does not comply with the assumption of well-mixture used for the models.

Therefore, as a result, the application of the program in its present form should be limited to cases with much slower transients. However, it might be a challenging study in the future to do numerical experiments with models involving both, partly ODEs and partly PDEs, the latter option for at least some “critical” components e.g. for the condensers, an item where the inadequate modeling by using only ODEs was not known when starting the presented work.
Conclusions

MSDL is a new computer language, for describing a class of thermodynamic systems made of vessels receiving mixtures of steam and gas, connected with pipes, such as boiling water reactor systems, as the "Simplified Boiling Water Reactor" (SBWR) and others. The MSDL Compiler translates the MSDL Code to a Fortran code, which finally can be compiled and run on any standard computer. During compilation, a collection of models, a so-called library, is linked to the code, which implements the physical details of each module. This new approach for describing thermal-hydraulic facilities in a highly modular way can be splitted into two main parts:

The mathematical kernel of the simulation code is based on LSODAR, a numerical integrator from ODEPACK. LSODAR allows a reasonably fast and accurate integration of ODEs and permits stiffness and event handling.

The physical descriptions of the modules are developed using Maple. Maple allows transforming implicit differential equations into ODEs in explicit form, using Fortran as the target language of output.

The core of the programming goals in this work was the development of the "glue", i.e. the interface, between physical module de-
criptions together with the mathematical kernel of the simulation environment. The highly automated code generation from a descriptive form to executable Fortran code has been implemented successfully. The simulation code is generated by a code generator, which is part of the MSDL compiler.

Comparisons between numerical computations using MSDL and real experiments where showing that the code is not perfectly suitable for a detailed thermal hydraulic analysis of integral experimental facilities of passive reactors. However, considering the underlying assumptions of the physical model this is not surprising. Differences are mainly due to the (too simple) assumption, that air and steam are always well mixed in the large compartments, and that even the liquid is well mixed with respect to temperature profiles. Referencing the experimental data, the distribution of air is far from being “well mixed” with steam in the Dry Wells and the Wet Wells. On the other hand, the code is very useful for predicting the general trends of a scenario. The software concept applied has shown to be very efficient for passing the steps between the implementation of a model of a system or a component to getting numerical results. Moreover it was possible to demonstrate that rather complex systems can be run efficiently even on a personal computer.

In a future development the code could be expanded by an additional module for the gas spaces, which takes care of the distribution of air within the volume. Such a model then could allow a better approach of the simulated data to the experimental facts.
Implementation of Modules

Systems have sub-systems and sub-systems have sub-systems and so on ad infinitum — which is why we’re always starting over.

— Epigrams in Programming, ACM SIGPLAN Sept. 1982

Without the extensive use of computer algebra tools and stream editors the implementation of the modules for a specific system would be extremely cumbersome and error prone. Therefore, the automated implementation of modules is one of the main objectives and achievements of this work. For this reason, a detailed description will be outlined in this chapter.

The modules are all implemented following the same scheme: From the physical description, differential equation systems are derived. These systems are solved analytically for their time derivatives. This analytical solution is converted into Fortran code.

The resulting code from the Maple code generator engine is just
a sequence of Fortran statements for a subroutine body. However, no code is generated for variable declarations, subroutine or function headers, etc. Thus some additional Fortran code has to be added to the Maple code which completes the code to a subroutine. These are the code parts for subroutine declaration, variable declaration, common blocks needed for input and output variables, and a proper return code at the end of the subroutine. This additional code generation beyond Maple code generator capabilities can be automated as well. For this task, some filters\(^1\) and stream editors\(^2\) are used. The filters and editors used will be described in some detail in section A.1.

To use this automatically generated code, a uniform user interface has to be implemented. This user interface provides the capability to choose the appropriate subroutine for the given physical state of the module. This state is determined by the main integrator program. Thus, the user does not have to worry about the physical state of the system, but needs only to provide a state vector which needs to be updated by the integrating system.

The modules used are explained in mainly tabular form. For one module, the whole development for the code generation is given.

A.1. Filters and Editors

The program for transforming the Maple code into a Fortran subroutine needs some filters and editors. These programs are part of standard Unix environments and are well available for other operating system environments as well.

A.1.1. Grep and Egrep

Grep stands for global regular expression print; that is, search through an entire file for a given regular expression (a pattern specified) and print the line or the lines that contain this pattern. The synopsis of grep is

\[
\text{grep [options] PATTERN [FILE...]} \\
\text{grep [options] [-e PATTERN | -f FILE] [FILE...]} 
\]

\(^1\)A filter is a program which selects data from a file by matching requested patterns. 
\(^2\)A stream editor is a program for editing files by means of applying a series of edits to a number of files automatically rather than editing each of the files interactively and manually.
where the most common options are i (ignore case), v (invert the sense of matching), and e (use PATTERN as pattern; useful to protect patterns beginning with −), respectively. A full documentation of grep and egrep can be found in the Unix man pages\(^3\)

A regular expression is a sequence of ordinary characters combined with special operators. Using regular expressions it is possible to search for general strings in files. Regular expressions are discussed in some detail later in section 5.2.3. Grep and egrep do not alter the original files. The output goes to the standard output device, usually the screen. To save the results, the output must be redirected to a file.

As an example, all the assignment statements of a Fortran code have to be filtered out of a source file. The assignment statements are characterized by the assignment operator “=”.

To find out all the assignment statements form the function \(vnorm\)^4:  

```
double precision function vnorm (n, v, w)
c--- optimize
c this function routine computes the weighted root-mean-square norm
of the vector of length n contained in the array v, with weights
in the array w of length n..
c vnorm = sqrt( (1/n) * sum( v(i)*w(i) )**2 )
c
integer n, i
double precision v, w, sum
dimension v(n), w(n)
sum = 0.0d0
do 10 i = 1,n
  sum = sum + (v(i)*w(i))**2
10 vnorm = dsqrt(sum/dfloat(n))
return
c------------------- end of function vnorm -------------------
end
```

In order to get only the assignment statements, we say

grep = vnorm.f

and get

c vnorm = sqrt( (1/n) * sum( v(i)*w(i) )**2 )
  sum = 0.0d0
  do 10 i = 1,n
```

\(^3\)The common Unix man pages should be installed on every Unix system. The Unix tools used for this word should be mostly system independent and work in the manner described here on any Unix system. If there is no running Unix system available, the man pages can be found on the web, e.g. at http://www.rt.com/man/

\(^4\)vnorm.f is a file describing the function vnorm, coming with the ode-pack software package
It is obvious, the line defining the do-loop is not an assignment. Do loops are characterized by a the key word “do”, followed by a numeric label, and more, which is not important at this point:

\[ \text{grep } = \text{vnorm.f | grep -v -e do.*[0-9]*.*} \]

And we get:

\[ \text{sum = sum + (v(i)*w(i))**2} \]
\[ \text{vnorm = dsqrt(sum/dfloat(n))} \]

As the last step, we have to ignore comments:

\[ \text{grep } = \text{vnorm.f | grep -v -e do.*[0-9]*.* | grep -v -e ^[cC]} \]

And we finally find all assignments in the function as:

\[ \text{sum = 0.0d0} \]
\[ \text{sum = sum + (v(i)*w(i))**2} \]
\[ \text{vnorm = dsqrt(sum/dfloat(n))} \]

A.1.2. Awk

The awk utility is a powerful pattern matching and processing language. It searches for one or more specified files and checks for patterns that match a specified expression. If awk finds a match, a corresponding action is performed. The concept is quite simple, but very powerful, which will turn out later on the description of the implemented awk programs for processing code generated by Maple. Awk uses the syntactical structure from the C programming language. However, awk is much more flexible than C, has some predefined variables and automations and a good output formatting engine. Built-in automations are, for example, the automatic retrieval of a record (usually a line of text from the input file(s)), the separation of this record into fields (usually words) and type conversions of variables where needed.

Awk was written by Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan in 1977[AKW79]. Since then, various extensions and new versions have been created. Finally a POSIX Standard (1003.2) [CIEE94] has been defined which is used in the freely available GNU-awk program gawk. This version in turn is based on the description in The AWK Programming Language, by Aho, Kernighan, and Weinberger, with the additional features found in the System V Release 4 version of
Unix awk. Gawk also provides more recent Bell Labs awk extensions, and some GNU-specific extensions.

As an example, a valid Fortran variable declaration has to be created out of a list of names of variables. The input file named input reads as

```
Alg
Gammastar
M1
Mdotb
Mdotin
Mdotout
Qdotin
Qdotwall1
V1
dpdt
dvildh11
dvildp
h1
hb
hin
hvi
v11
```

and has to be converted into the output file input.inc as

```
Listing A.2: Fortran variable declaration

```

```
DOUBLE PRECISION  Alg, Gammastar, M1, Mdotb,
#    Mdotin, Mdotout, Qdotin, Qdotwall1,
#    V1, dpdt, dvildh11, dvildp,
#    h1, hb, hin, hvi,
#    v11
```

The code producing the Fortran declaration reads as given in the following listing:

```
Listing A.3: AWK Program for generating declarations

```

```
BEGIN {
  start=1
  max=4
  s=0
}

{ if (s == 0)
  { if (start == 1)
    { printf "\t\t\t"DOUBLE\tPRECISION\t\t\t$1",$1
      start=0
    } else
      printf \t\t\t#\t\t\t\t\t$1",$1
  }
```
if (s >= 1 && s < max)
  printf ",\$s",$1
s++
if (s == max)
  s=0
}

END {  
  printf "\n\n"
}

As AWK works line oriented on the input file, the sections named BEGIN and END are executed only in the beginning on at the end of the processing, however. The main body of the program, beginning on line eight, is executed for every line of the data input file.

A.1.3. Sed

As mentioned above, sed is a non-interactive stream editor [PP02], thus can be used to modify information traveling through a Unix pipeline. It interprets a script that controls the actions to be performed by sed. Sed input typically comes from files. The output is written to the standard output device, usually the screen, but can be redirected to other files or programs. Sed offers capabilities as known from standard (interactive) editors: Searching and replacing words, deleting lines, inserting text at specified positions, etc. The difference is the fact that all edits to be applied have to be known in advance. Then this modifications of files can be done in a stream rather than manually. For example, in the Maple source code, all words consisting of 5 space characters followed by a # have to be replaced by a single space character. If done interactively in several files with 1000 lines each, this is an annoying and time consuming task. Sed can do that in a fraction of a second. The possibilities of sed can be summarized in three major points:

- Automated editing of multiple files
- Simplicity of performing the same edits on multiple files
- Write conversation programs.

As a simple example, let's substitute all ** by a single ^ in a file named sourcefile:
That’s it! It is obvious, the above sed-command can be applied from within a script, so many files can be edited in a very efficient and reliable way.

A.1.4. Sort

Sort sorts, merges, or compares all the lines from the given files, or standard input if none are given. By default, sort writes the results to standard output. Some options are available, such to force sort not to print multiple entries, to ignore leading blanks, to ignore case sensitivity or to sort in reverse order, to give some examples.

The program for converting automatically generated maple code into a valid Fortran subroutine makes extensive use of the programs described above. See A.2.2 for the full listing.

A.2. From Mathematical Description to Library Modules

As an example for the conversation from the physical description into a library subroutine, the development of the module describing a liquid space is reconstructed here step by step. The needed sed edits and awk scripts as well as the Maple programs and intermediate results are presented.

A.2.1. Step 1: From Mathematical Description to Fortran Code Body

The first step is to translate the physical equations into a system of ODEs in a form Maple can handle. The three equations of a liquid phase system of ODEs is presented in chapter 2 on page 23 in equation (2.53). For convenience, this system is given as reference below again:

\begin{equation}
\frac{dM_i}{dt} = \dot{M}_{in} - \dot{M}_{ig} - \dot{M}_b - \dot{M}_{out} \tag{a}
\end{equation}

\begin{equation}
M_i \frac{dh_i}{dt} = \dot{Q}_{in} + \dot{Q}_{wall,i} - \dot{Q}_{l\rightarrow i} + \dot{M}_{in} (h_{in} - h_l) - \dot{M}_{ig} (h_{v,i} - h_l) - \dot{M}_b (h_b - h_l) + V_l \frac{dp}{dt} \tag{b}
\end{equation}

\begin{equation}
\frac{dV_l}{dt} = A \frac{dp}{dt} + B \frac{dM_i}{dt} + C \frac{dh_i}{dt} \tag{c}
\end{equation}

(A.1)
This set of equations is formulated in Maple as

\begin{align*}
eq_	ext{mass} &:= \frac{dM}{dt} = M_{\text{dot in}} - \Gamma_{\text{star}} \cdot \text{Alg} \\
eq_	ext{ener} &:= M_{\text{l}} \cdot \frac{dh}{dt} = Q_{\text{dot in}} + Q_{\text{dot wall l}} - Q_{\text{dot l}} \\&+ M_{\text{dot in}} \cdot (h_{\text{i}} - h_{\text{l}}) \\
eq_	ext{Vol} &:= \frac{dV}{dt} = A \cdot \frac{dp}{dt} + B \cdot \frac{dM}{dt} + C \cdot \frac{dh}{dt};
\end{align*}

These three equations have to be collected to a system of equations which is solved for the unknown, implicit given time derivatives \( \frac{dM}{dt} \), \( \frac{dh}{dt} \) and \( \frac{dV}{dt} \), respectively:

\begin{align*}
\text{eqsystem} &:= \{\text{eq_mass, eq_ener, eq_Vol}\}; \\
\text{unkns} &:= \{\frac{dM}{dt}, \frac{dh}{dt}, \frac{dV}{dt}\};
\end{align*}

The state equations as well as the equations for the parameters \( A, B, \) and \( C \) have to be defined. This is shown for the subcooled case only. The saturated case is done in a similar way. The expressions for the parameters \( A, B, \) and \( C \) have to be substituted into the system of equations.

\begin{align*}
\text{specvolsub} &:= v_{\text{l}} = v_{\text{l sat}}; \\
\text{Asub} &:= A = M_{\text{l}} \cdot \text{dvlldp}; \\
\text{Bsub} &:= B = vl; \\
\text{Csub} &:= C = M_{\text{l}} \cdot \text{dvlldhll};
\end{align*}

\begin{align*}
\text{parasub} &:= \{\text{Asub, Bsub, Csub}\}; \\
\text{syssub} &:= \text{subs(parasub, specvolsub, eqsystem)};
\end{align*}

This resulting system of equations now can be solved for the unknowns. After the system has been solved (analytically!), the solution is converted into optimized Fortran code.

\begin{align*}
\text{solve}(\text{syssub, unkns}); \\
\text{assign("}); \\
\text{sol1} := \text{allvalues}(\frac{dM}{dt}, \text{\'dependent\');} \\
\text{sol2} := \text{allvalues}(\frac{dV}{dt}, \text{\'dependent\');} \\
\text{sol3} := \text{allvalues}(\frac{dh}{dt}, \text{\'dependent\');} \\
\text{fortran([}\text{\{dM/dt=\ sol1, dV/dt=\ sol2, dh/dt=\ sol3\},} \\
\text{ optimized, \text{\'filename=\'autofort_sub.for\');}
\end{align*}

Optimized means that a common subexpression optimization is performed. The result is a sequence of assignment statements in which temporary values are stored in local variables beginning with the letter \( t \). The global names \( t0, t1, t2, \ldots \) are reserved for use by For-
tran for this purpose. The resulting Fortran code is stored in the file
autofort_sub.for:

Listing A.4: Automatically generated, native Fortran Code

\[
\begin{align*}
t1 &= \text{Gammastar}\times\text{Alg} \\
dMldt &= \text{Mdotin}-t1-\text{Mdotb}-\text{Mdotout} \\
t12 &= \text{dvildhdl1}\times\text{Mdotin} \\
t15 &= \text{dvildhdl1}\times\text{Gammastar} \\
t20 &= \text{dvildhdl1}\times\text{Mdotb} \\
dVldt &= M1\times\text{dvildp}\times\text{dpdt}+vll\times\text{Mdotin}-vll\times\text{Gammastar}\times\text{Alg}-vll\times\text{Mdotb}-vll\times Mdotout + dvildhll \times Qdotin + dvildhll \times Qdotwalll - dvildhll \times Qdot1i + t12 \times hin \\
#- t12 \times hl - t15 \times Alg - hl - t20 \times hb + t20 \times hl + dvildhll \times Vl \times dpdt \\
dhldt &= (Qdotin + Qdotwalll - Qdot1i + Mdotin \times hin - Mdotin \times hl - t1 \times hv1 + t1 \times hl \\
#- Mdotb \times hb + Mdotb \times hl + Vl \times dpdt) / M1
\end{align*}
\]

A.2.2. Step 2: From Fortran Code Body to Fortran Subroutine

The resulting Fortran code from Maple (seen above) obviously is not
very useful to the end user of the code. First of all, it has to be con¬
verted into a real subroutine with input and output capabilities. This
 task can be performed automatically by a small Unix shell script,
which will be explained line by line below.

Listing A.5: Translation from Maple to Fortran code

```
#!/bin/sh
grep = autofort.for | awk '{print $1}' | egrep 't[1-9][0-9]*' | \  
sort -u > declare 
grep = autofort.for | awk '{print $1}' | egrep -v 't[1-9][0-9]*' | \  
sort -u > output 
awk -f $HOME/diss/bin/oneline.awk autofort.for | sed 's/ /#/g' | \
    awk '{print $3}' > ccc 
sed 's/DOUBLE PRECISION/COMMON /input/g' ccc > ddd 
sed 's/DOUBLE PRECISION/COMMON /output/g' output.inc >> output.inc 
awk -f $HOME/diss/bin/dec_t.awk declare > autofort.f 
cat autofort.for >> autofort.f 
echo 'END' >> autofort.f 
echo 'Automatic generated code ends.' >> autofort.f 
awk -f $HOME/diss/bin/include.awk input >> input.inc 
sed 's/DOUBLE PRECISION/COMMON /input//g' input.inc >> input.inc 
awk -f $HOME/diss/bin/include.awk output >> output.inc 
sed 's/DOUBLE PRECISION/COMMON /output//g' output.inc >> output.inc 
rm input output declare
```

The first line introduces the script as a shell script run in the sh
environment. For this purpose, a new shell is forked with the line
The next two lines are Unix pipelines with the four commands grep, awk, egrep and sort, respectively. Note, that a Unix command line is continued with the \ character at the end of a text line. The command

```
grep = autofort.for | awk '{print $l}' | egrep 't[l-9][0-9]*' | sort -u > declare
```

finds all variables beginning with a t and writes the sorted list to the file declare. This list looks like

- t1
- t12
- t15
- t20

For this task, every variable has to appear at least once on the left hand side of an assignment statement. Thus, all lines containing a = character are piped by grep, the awk command which prints out the first word. The first word on a line containing an assignment is the variable name itself. Using this mechanism we find all variables which need to be declared in our program. Besides the variables to be declared we have to find the output variables. These are the variables to which a value is assigned but whose names do not match the regular expression t [1-9] [0-9]*. Regular expressions are discussed in section 5.2.3. This is done in the command line

```
grep = autofort.for | awk '{print $l}' | egrep -v 't[l-9][0-9]*' | sort -u > output
```

The output file contains the entries

- dMldt
- dVl.dt
- dhldt

The next task is to find out the input variables. This turns out to be a little more difficult because of the continuation line syntax of Fortran. Since Maple breaks a Fortran line after exactly 72 characters, it may break within a variable name. For this, the Fortran code is modified to an intermediate code without continuation lines but with very long lines. From this code, all operators are replaced by newline characters which leads to a list of all variables. An additional grep filter eliminates the intermediate and output variables. The code for this task is
A.2. From Mathematical Description to Library Modules

The awk program dec_t.awk creates the declaration section of variables and writes the subroutine header:

Listing A.6: AWK Program for generating Subroutine Headers

```
BEGIN {
    s=0
    max=6
    printf "C-:-----------------------------C\n"
    printf "C_Automatically_generated_code_starts...C\n"
    printf "C-:----------------------------------C\n"
    printf "C_DO_NOT_EDIT_THIS_FILE!!!C\n"
    printf "C-:----------------------------------C\n"
    printf "C-:----------------------------------C\n"
    printf "SUBROUTINE_dots\n"
    printf "implicit\n"
    printf "include\"input.inc\"\n"
    printf "include\"output.inc\"
}
```

which leads to the file input:

```
Alg
Gammastar
ML
Mdotb
Mdotin
Mdotout
Qdotin
Qdotwall
V1
dpdt
dv1ldh11
dv1ldp
hl
hb
hin
hvi
vll
```

The next four lines

```
awk -f $HOME/diss/bin/dec_t.awk declare > autofort.f
cat autofort.for >> autofort.f
echo 'C Automatically generated code ends.' >> autofort.f
```

finally complete the Maple generated Fortran body to a subroutine.

```
awk -f $HOME/diss/bin/oneline.awk autofort.for | sed 's/ #/\n' | \nawk '{print $3}' > ccc
sed 's/\*\*/\*/g' ccc > ddd
sed 's/[^a-zA-Z0-9 \]/g' ddd | sed 's/ //g' | grep -v 't[l--9][0-9]*/' | \ngrep '[a-zA-Z].*' sort -u > input
rm ccc ddd
```

```
which leads to the file input:
```
```
```
```
```
Thereafter the Fortran subroutine looks like

Listing A.7: Automatically generated Fortran Code

```fortran
SUBROUTINE dotS_Sub
    implicit none
    include "input.inc"
    include "output.inc"

    DOUBLE PRECISION t1, t12, t15, t20
    t1 = Gammastar*Alg
dM1dt = Mdotin-t1-Mdotb-Mdotout
t12 = dv11dh11*Mdotin
t15 = dv11dh11*Gammastar
t20 = dv11dh11*Mdotb
dV1dt = M1*dv1ldp*dpdt+v11*Mdotin-v11*Gammastar*Alg-v11*Mdotb-v11*
#Mdotout+dv11dh11*Qdotin+dv11dh11*Qdotwall1-dv11dh11*Qdot1i+t12*hin
#-t12*h1-t15*Alg+hvi+t15*Alg*h1-t20*hb+t20*h1+dv1ldh11*V1*dpdt

dh1dt = (Qdotin+Qdotwall1-Qdot1i+Mdotin*hin-Mdotin*h1-t1*hvi+t1*h1
#-Mdotb*hb+Mdotb*h1+V1*dpdt)/M1
END
```

which contains all required additional lines for subroutine header, variable declaration and input/output capabilities using include files which are generated in the last few lines of the script:

```
awk -f $HOME/diss/bin/include.awk input > input.inc
```
sed 's/DOUBLE PRECISION/COMMON \"input\"/g' input.inc >> input.inc
awk -f $HOME/diss/bin/include.awk output > output.inc
sed 's/DOUBLE PRECISION/COMMON \"output\"/g' output.inc >> output.inc

The corresponding awk scripts are based on the files containing the lists of variables, created above. Thereafter, the files input.inc and output.inc, respectively look as follows:

Listing A.8: Generated Fortran Input File

```fortran
DOUBLE PRECISION  Alg, Gammastar, Ml, Mdotb,
# Mdotin, Mdotout, Qdotin, Qdotwalll, Qdotli, Vl,
# dh1lsatdp, dh1vsatdp, dpdt, dv1lsatdp,
# dv1vsatdp, h1, hilsat, hvsat,
# hb, hin, hvi, vllsat,
# vllsat,
# dv1ldhll, dv1ldp, v1

COMMON /liquid_in/  Alg, Gammastar, Ml, Mdotb,
# Mdotin, Mdotout, Qdotin, Qdotwalll, Qdotli, Vl,
# dh1lsatdp, dh1vsatdp, dpdt, dv1lsatdp,
# dv1vsatdp, h1, hilsat, hvsat,
# hb, hin, hvi, vllsat,
# vllsat,
# dv1ldhll, dv1ldp, v1
```

for the input include file and for the output file

Listing A.9: Generated Fortran Output File

```fortran
DOUBLE PRECISION  dMldt, dvldt, dhldt

COMMON /liquid_out/  dMldt, dvldt, dhldt
```

as discussed. These subroutines are neither very comfortable nor do they save enough for the end user of the simulation environment presented in this work. Thus an additional library routine has to be developed which uses the automatically generated subroutine above.

A.2.3. Step 3: From Fortran Subroutine to Library Routine

The main task of the library routine is to distinguish between the cases where different physical models can be used. This routine has to provide the capability for selecting the right model according to a state vector and has to provide the input variables for the automatically generated subroutines. Due to this, the user does not have to know anything about neither the implementation of mass property functions used by the module, nor the parameter passing mechanism. The only thing transparent to the user is the calling interface. This calling interface in case of the liquid phase is given by
where all variables except the state variable are of Fortran type double precision. State is a state vector which describes the physical state of the module. Thanks to this, no if statements are required inside the module beside the differentiation of the appropriate physical model given by the state vector. This allows for a stable numerical integration over the switching of modules.

Depending on the numerical values of some of the state variables, different model descriptions may become valid during the transient. As an example, when heating up water to the boiling point, different models need to be applied during the warmup phase and afterwards when boiling has started. Times of the occurrence of events which cause the change of the model description need to be determined accurately during integration. These events may be formulated as functions of time and/or one or more state variables. Events occur at those points where such functions show a zero crossing. Therefore special integrators need to be applied which feature root finding capabilities for localizing such zero crossings.

The formulation of these functions is relatively simple and is added to the code automatically at compile time. As an example the function for localizing the event when the saturation of water in the RPV happens, is given by

$$R_i = h_l - h_{sat}(P)$$

(A.2)

where $R_i$ is the $i$-th element of the function vector $R$ which represents all the event functions in the system.

The event functions may differ in complexity. The simplest event function depends only on time. Hence the occurrence of the associated event is known in advance, e.g. the opening of a valve at a given time.

Once an event is detected a status vector is updated which takes care of all the actual models in use at any time. Within each module this status vector is checked to insure the usage of the appropriate model description.

The state vector of this module is described later in table A.3. The implementation of the library routine is given as

Listing A.11: Library Routine Body
A.3. The Modules Library

This implementation allows for a save and uniform calling interface for the physical modules provided. For debugging purposes, it is very easy to implement a very verbose output to control the input and output conditions for a vessel. Besides that, all input parameters might be tested easily for a valid physical range and corresponding warning messages might be printed out.

All the compiled library modules are collected in a library archive file. This file is created using the ar utility available in Unix. This is not absolutely needed but simplifies the linking step in the main compilation routine of simulation programs.
tion of its modules. He only needs to know the interface of the callable subroutine. Currently, only Fortran interfaces are supported; the migration of the interfaces to C/C++ will be a straightforward task and may be needed in the future.

In this chapter, the interfaces of all the modules callable by the user are described. A brief description of the input and output parameters is provided in tabular form beside the user interface which is mandatory for the programmer. Other details, as the physical background of the modules or numerical methods used are described in chapter 2 on modularization and 3 on numerical methods, respectively.

A.3.1. Properties of Water, Air and Steam

The properties of water, air and steam are implemented as functions. The implementation is based on the ASME steam tables [MMSs]92 with an SI compatible interface. Functions for water and air are based on [MS00]. The available functions are given in table A.1. Properties of gas mixtures are calculated in the modules where needed. They are based on the pure steam or pure air properties.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_{v,\text{sat}}(p)$</td>
<td>Saturated vapor enthalpy as function of $P$</td>
<td>hgp_si(PIN)</td>
</tr>
<tr>
<td>$h_{l,\text{sat}}(p)$</td>
<td>Saturated liquid enthalpy as function of $P$</td>
<td>hfp_si(PIN)</td>
</tr>
<tr>
<td>$h_{v}(p,T)$</td>
<td>Enthalpy of superheated steam as function of $P$ and $T$</td>
<td>hptd_si(PIN,TIN)</td>
</tr>
<tr>
<td>$h_{l}(p,T)$</td>
<td>Enthalpy of compressed liquid as function of $P$ and $T$</td>
<td>hptl_si(PIN, TIN)</td>
</tr>
</tbody>
</table>

**Enthalpies of water and steam**

**Temperatures of water and steam**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{sat}}(p)$</td>
<td>Saturation temperature as a function of $P$</td>
<td>tsatp_si(PIN)</td>
</tr>
<tr>
<td>$T_{l}(p,h)$</td>
<td>Temperature of compressed liquid as a function of $P$ and $H$</td>
<td>tphl_si(PIN, HIN)</td>
</tr>
<tr>
<td>$T_{v}(p,h)$</td>
<td>Superheated steam temperature as a function of $P$ and $H$</td>
<td>tphd_si(PIN, HIN)</td>
</tr>
</tbody>
</table>

**Pressure of saturation**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{\text{sat}}(T)$</td>
<td>Saturation pressure as a function of $T$</td>
<td>psatt_si(TIN)</td>
</tr>
</tbody>
</table>

**Specific heat of water and steam**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{p,v}(p,T)$</td>
<td>Specific Heat of superheated steam as a function of $P$ and $T$</td>
<td>cpptd_si(PIN,TIN)</td>
</tr>
<tr>
<td>$c_{p,l}(p,T)$</td>
<td>Specific Heat $(C_p)$ of compressed liquid as a function of $P$ and $T$</td>
<td>cpptl_si(PIN,TIN)</td>
</tr>
</tbody>
</table>

**Specific volume of water and steam**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_{v}(p,T)$</td>
<td>Volume of superheated steam as a function of $P$ and $T$</td>
<td>vptd_si(PIN, TIN)</td>
</tr>
<tr>
<td>$v_{l}(p,T)$</td>
<td>Volume of compressed liquid as a function of $P$ and $T$</td>
<td>vptl_si(PIN, TIN)</td>
</tr>
<tr>
<td>$v_{l,\text{sat}}(p)$</td>
<td>Saturated liquid volume as a function of $P$</td>
<td>vfp_si(PIN)</td>
</tr>
<tr>
<td>$v_{v,\text{sat}}(p)$</td>
<td>Saturated vapor volume as a function of $P$</td>
<td>vgp_si(PIN)</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\eta(v, T)$</td>
<td>Dynamic viscosity of superheated steam and compressed liquid as a function of $V$ and $T$</td>
<td>viscos_si(VIN, TIN)</td>
</tr>
<tr>
<td>$\eta_v(p, T)$</td>
<td>Dynamic viscosity of superheated steam as a function of $P$ and $T$</td>
<td>visv_si(PIN, TIN)</td>
</tr>
<tr>
<td>$\eta_l(p, T)$</td>
<td>Dynamic viscosity of compressed liquid as a function of $P$ and $T$</td>
<td>visl_si(PIN, TIN)</td>
</tr>
</tbody>
</table>

**Viscosity of water and steam**

**Thermal conductivity of water and steam**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda(v, T)$</td>
<td>Thermal conductivity of superheated steam and compressed liquid as a function of $V$ and $T$</td>
<td>convt_si(VIN, TIN)</td>
</tr>
<tr>
<td>$\lambda_v(p, T)$</td>
<td>Thermal conductivity of superheated steam as a function of $P$ and $T$</td>
<td>condv_si(PIN, TIN)</td>
</tr>
<tr>
<td>$\lambda_l(p, T)$</td>
<td>Thermal conductivity of compressed liquid as a function of $P$ and $T$</td>
<td>condl_si(PIN, TIN)</td>
</tr>
</tbody>
</table>

**Prandtl numbers of water and steam**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Pr(p, T)$</td>
<td>Prandtl number of compressed liquid or superheated steam as function of $P$ and $T$</td>
<td>prandl_si(PIN, TIN)</td>
</tr>
<tr>
<td>$Pr_l(p, T)$</td>
<td>Prandtl number of compressed liquid as a function of $P$ and $T$</td>
<td>prlig_si(PIN, TIN)</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{\partial v_y}{\partial T}$</td>
<td>$p$</td>
<td>dv_vdT_p_si(PIN, TIN)</td>
</tr>
<tr>
<td>$\frac{\partial v_y}{\partial p}$</td>
<td>$T$</td>
<td>dv_vdp_T_si(PIN, TIN)</td>
</tr>
<tr>
<td>$\frac{\partial v_y}{\partial T}$</td>
<td>$T$</td>
<td>dv_vdp_T_p(TIN)</td>
</tr>
<tr>
<td>$\frac{\partial h_l}{\partial p}$</td>
<td>$sat$</td>
<td>dhlsatdp_si(PIN)</td>
</tr>
<tr>
<td>$\frac{\partial h_l}{\partial p}$</td>
<td>$sat$</td>
<td>dhvsatdp_si(PIN)</td>
</tr>
<tr>
<td>$\frac{\partial h_l}{\partial p}$</td>
<td>$sat$</td>
<td>dvlsatdp_si(PIN)</td>
</tr>
<tr>
<td>$\frac{\partial h_l}{\partial p}$</td>
<td>$sat$</td>
<td>dvvsatdp_si(PIN)</td>
</tr>
</tbody>
</table>

### Properties of air

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Fortran call</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{pa}(T)$</td>
<td>Specific heat of air</td>
<td>cp_air_si(TIN)</td>
</tr>
<tr>
<td>$h_{a}(T)$</td>
<td>Specific enthalpy of air</td>
<td>h_air_si(TIN)</td>
</tr>
<tr>
<td>$\eta_{a}(T)$</td>
<td>Dynamic viscosity of air</td>
<td>vis_air_si(TIN)</td>
</tr>
<tr>
<td>$\lambda_{a}(T)$</td>
<td>Thermal conductivity of air</td>
<td>cond_air_si(TIN)</td>
</tr>
<tr>
<td>$v_{a}(p, T)$</td>
<td>Specific volume of air</td>
<td>vpt_air_si(PIN, TIN)</td>
</tr>
</tbody>
</table>

*Table A.1: Properties of mass*
A.3.2. The Gas Space

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_v )</td>
<td>( \text{kg} )</td>
<td>( M_v )</td>
<td>Mass of steam</td>
</tr>
<tr>
<td>( M_a )</td>
<td>( \text{kg} )</td>
<td>( M_a )</td>
<td>Mass of air</td>
</tr>
<tr>
<td>( p_v )</td>
<td>( \text{Pa} )</td>
<td>( p_v )</td>
<td>Partial steam pressure</td>
</tr>
<tr>
<td>( p_a )</td>
<td>( \text{Pa} )</td>
<td>( p_a )</td>
<td>Partial air pressure</td>
</tr>
<tr>
<td>( T )</td>
<td>( \text{K} )</td>
<td>( T )</td>
<td>Gas temperature</td>
</tr>
<tr>
<td>( \dot{M}_{fg} )</td>
<td>( \text{kg} \text{s}^{-1} )</td>
<td>( \dot{M}_{fg} )</td>
<td>Mass transfer rate through interface</td>
</tr>
<tr>
<td>( M_b )</td>
<td>( \text{kg} )</td>
<td>( M_b )</td>
<td>Mass transfer in steam bubbles</td>
</tr>
<tr>
<td>( h_{vi} )</td>
<td>( \text{kg} )</td>
<td>( h_{vi} )</td>
<td>Enthalpy of the transferring steam</td>
</tr>
<tr>
<td>( h_b )</td>
<td>( \text{kg} )</td>
<td>( h_b )</td>
<td>Enthalpy of steam bubbles</td>
</tr>
<tr>
<td>( M_{out} )</td>
<td>( \text{kg} \text{s}^{-1} )</td>
<td>( M_{out} )</td>
<td>Gas exit rate</td>
</tr>
<tr>
<td>( \dot{M}_{v,\text{in}} )</td>
<td>( \text{kg} \text{s}^{-1} )</td>
<td>( \dot{M}_{v,\text{in}} )</td>
<td>Steam injection rate</td>
</tr>
<tr>
<td>( \dot{M}_{a,\text{in}} )</td>
<td>( \text{kg} \text{s}^{-1} )</td>
<td>( \dot{M}_{a,\text{in}} )</td>
<td>Air injection rate</td>
</tr>
<tr>
<td>( h_{v,\text{in}} )</td>
<td>( \text{kg} )</td>
<td>( h_{v,\text{in}} )</td>
<td>Enthalpy of injected steam</td>
</tr>
<tr>
<td>( h_{a,\text{in}} )</td>
<td>( \text{kg} )</td>
<td>( h_{a,\text{in}} )</td>
<td>Enthalpy of injected air</td>
</tr>
<tr>
<td>( \dot{Q}_{vi} )</td>
<td>( \text{W} )</td>
<td>( \dot{Q}_{vi} )</td>
<td>Heat from interface to gas</td>
</tr>
<tr>
<td>( \dot{Q}_t )</td>
<td>( \text{W} )</td>
<td>( \dot{Q}_t )</td>
<td>Added heat</td>
</tr>
<tr>
<td>( A_{lp} )</td>
<td>( \text{m}^2 )</td>
<td>( A_{lp} )</td>
<td>Area of vessel</td>
</tr>
<tr>
<td>( V_g )</td>
<td>( \text{m}^3 )</td>
<td>( V_g )</td>
<td>Volume of vessel</td>
</tr>
<tr>
<td>( \frac{dV_g}{dt} )</td>
<td>( \text{m}^3 \text{s}^{-1} )</td>
<td>( \frac{dV_g}{dt} )</td>
<td>Change of gas volume</td>
</tr>
<tr>
<td>state</td>
<td>-</td>
<td>state(5)</td>
<td>Physical state vector</td>
</tr>
</tbody>
</table>

**Output parameters**

| \( \frac{dM_v}{dt} \) | \( \text{kg} \text{s}^{-1} \) | \( \frac{dM_v}{dt} \) | Derivative of mass of steam |
| \( \frac{dM_a}{dt} \) | \( \text{kg} \text{s}^{-1} \) | \( \frac{dM_a}{dt} \) | Derivative of mass of air |
| \( \frac{dp_v}{dt} \) | \( \text{N} \text{m}^{-2} \text{s}^{-1} \) | \( \frac{dp_v}{dt} \) | Derivative of steam partial pressure |
| \( \frac{dp_a}{dt} \) | \( \text{N} \text{m}^{-2} \text{s}^{-1} \) | \( \frac{dp_a}{dt} \) | Derivative of air partial pressure |
| \( \frac{dT}{dt} \) | \( \text{K} \text{s}^{-1} \) | \( \frac{dT}{dt} \) | Derivative of gas temperature |

**State vector**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>state(1)</td>
<td>( &gt; 0 )</td>
<td>Mix of steam and air in gas space</td>
</tr>
<tr>
<td>state(1)</td>
<td>( &lt; 0 )</td>
<td>Pure steam in gas space</td>
</tr>
<tr>
<td>state(2)</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

*continued on next page*
Test run of the Gas Space Module

For the test run of the gas space, its volume is kept constant at 10 m$^3$. There is no mass flow due to any liquid gas interfaces. Thus the gas space is something like a perfect drywell in the PANDA facility. The mass inside the gas space is forced at a constant level, but some input and output of masses is enforced. The output is at a constant rate of 0.05 kg/s. The output mixture corresponds to the actual mixture inside the gas volume, as we assume a well mixed gas all over the time.

The input flow rates are set to 0.02 kg/s for steam and 0.03 kg/s for air, respectively. The enthalpies correspond to saturation at 4.0 x 10$^5$ Pa for the steam, and for the corresponding temperature for air.

As seen in figure A.1, the steam mass and partial pressure are decreasing due to the lower steam input rates, compared to air input. The overall mass in the system remains constant at 28.57 kg, and the total pressure changes little from 4.00000 x 10$^5$ Pa to 4.12175 x 10$^5$ Pa due to the higher enthalpies, and the different composition of the gas mixture.
Figure A.1: Test run of the gas space module

A.3.3. The Liquid Space

<table>
<thead>
<tr>
<th>Module: liquid space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>Input parameters</td>
</tr>
<tr>
<td>$A_{lg}$</td>
</tr>
<tr>
<td>$\dot{M}_{fg}$</td>
</tr>
<tr>
<td>$M_1$</td>
</tr>
<tr>
<td>$\dot{M}_b$</td>
</tr>
<tr>
<td>$\dot{M}_{in}$</td>
</tr>
<tr>
<td>$\dot{Q}_{1-in}$</td>
</tr>
<tr>
<td>$\dot{Q}_{in}$</td>
</tr>
<tr>
<td>$\dot{Q}_{wall}$</td>
</tr>
<tr>
<td>$V_1$</td>
</tr>
<tr>
<td>$p$</td>
</tr>
<tr>
<td>$\frac{dp}{dt}$</td>
</tr>
<tr>
<td>$h_1$</td>
</tr>
<tr>
<td>$h_{in}$</td>
</tr>
</tbody>
</table>
Test run of the liquid space Module

No mass is added or removed to the liquid at the interface. No mass flow is allowed to or from the module. Additionally, a constant heat source of $5.0 \times 10^6 W$ is applied. Initially, the liquid is well subcooled at 293.15K. As clearly seen in figure A.2, after 841.3s, the water gets saturated. From this moment on, steam bubbles start to form inside the liquid. As we still assume a well mixed liquid phase, its volume has to increase dramatically. This case is not realistic in means of a SBWR, or PANDA, as the liquid phase modeled in this test run is held inside a flexible containment, like a balloon.

Nevertheless, this run shows clearly the tracking of the discontinuities, and thus the change of physical model as soon as saturation is reached. Besides this, the increasing volume of the liquid with increasing temperature (or enthalpy, respectively) is verified. From the data plotted in figure A.2, the rise of liquid volume in the time from 0 to 841.3s can be observed as well, starting at $10.016 m^3$ and increasing up to $10.608 m^3$ just before boiling.
The end-point of this transient can be verified analytically. Given the energy and mass balances, the total volume of the liquid module can be calculated as outlined below:

Starting from the energy and mass balances

\[ m_i h_i + m_v h_v = Q_{\text{tot}} \]
\[ m_i + m_v = m_0 \]  \hspace{1cm} (A.3)

we get the final volume of the steam as

\[ m_v = \frac{Q_{\text{tot}} - m_0 h_{\text{sat}}}{h_{v_{\text{sat}}} - h_{l_{\text{sat}}}} \]  \hspace{1cm} (A.4)

with

\[ Q_{\text{tot}} = m_0 h_{l_0} + \dot{Q} t. \]  \hspace{1cm} (A.5)

As no mass is added or removed from the module, and the pressure is kept constant, the mass of vapor can be calculated. With the initial values \( T_0 = 293.15K, p = 2.0 \times 10^5 Pa, M_0 = 1.0 \times 10^4 kg \) and \( \dot{Q} = 5.0 \times 10^6 W \) we get \( m_v = 814.6 kg \) which leads to the total volume of the liquid space after 1200s as 731.05 m\(^3\), which fits the value calculated by the test run as 731.046 m\(^3\), as expected.
A.3. The Modules Library

A.3.4. The Interface

This module implements the numerical solution of the non-linear equation 2.89

\[
\frac{\alpha_{l \rightarrow i} A_{lg}(T_l - T_i) + \alpha_{g \rightarrow i} A_{lg}(T_g - T_i)}{h_v - h_l} = \theta A_{lg} \frac{p_{v,i} - p_v}{p - p_{v,i}} \quad \text{(A.6)}
\]

from chapter 2.4.5 on page 29. This equation depends on the interface temperature \( T_i \). As the interface has to be evaluated very often during a system integration, the numerical method needs to be very efficient. The actual implementation uses the implementation known as the Van Wijngaarden-Dekker-Brent method as proposed by Brent. [Bre73]. The method is guaranteed to converge, if properly applied. In the actual implementation for solving the interface equation, it converges with an accuracy of about \( 10^{-16} \) after 5 to 9 iteration steps. The calling interface of the interface module is presented in table A.4.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_v )</td>
<td>( kg )</td>
<td>( M_v )</td>
<td>Mass of steam in gas space</td>
</tr>
<tr>
<td>( M_a )</td>
<td>( kg )</td>
<td>( M_a )</td>
<td>Mass of air in gas space</td>
</tr>
<tr>
<td>( M_l )</td>
<td>( kg )</td>
<td>( M_l )</td>
<td>Mass of water in liquid space</td>
</tr>
<tr>
<td>( p_v )</td>
<td>( N )</td>
<td>( p_v )</td>
<td>Partial steam pressure</td>
</tr>
<tr>
<td>( p_a )</td>
<td>( N )</td>
<td>( p_a )</td>
<td>Partial air pressure</td>
</tr>
<tr>
<td>( h_l )</td>
<td></td>
<td>( h_l )</td>
<td>Enthalpy of water</td>
</tr>
<tr>
<td>( T_g )</td>
<td>( K )</td>
<td>( T_g )</td>
<td>Temperature of gas</td>
</tr>
<tr>
<td>( V_g )</td>
<td>( m^3 )</td>
<td>( V_g )</td>
<td>Volume of gas space</td>
</tr>
<tr>
<td>( V_{tot} )</td>
<td>( m^3 )</td>
<td>( V_{tot} )</td>
<td>Total volume of pressure vessel</td>
</tr>
<tr>
<td>( A_{lg} )</td>
<td>( m^2 )</td>
<td>( A_{lg} )</td>
<td>Area of vessel</td>
</tr>
<tr>
<td>( \text{state} )</td>
<td></td>
<td>( \text{state(5)} )</td>
<td>Physical state</td>
</tr>
</tbody>
</table>

Output parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \dot{M}_{fg} )</td>
<td>( kg )</td>
<td>( \dot{M}_{fg} )</td>
<td>Mass transfer liquid → gas</td>
</tr>
<tr>
<td>( \dot{M}_b )</td>
<td>( kg )</td>
<td>( \dot{M}_b )</td>
<td>Mass transfer flashing</td>
</tr>
<tr>
<td>( h_{oi} )</td>
<td>( kg )</td>
<td>( h_{oi} )</td>
<td>Enthalpy of evaporation</td>
</tr>
<tr>
<td>( h_b )</td>
<td>( kg )</td>
<td>( h_b )</td>
<td>Enthalpy of raising bubbles</td>
</tr>
<tr>
<td>( \dot{Q}_{l \rightarrow i} )</td>
<td>( W )</td>
<td>( \alpha_{li} )</td>
<td>Heat transfer liquid→interface</td>
</tr>
<tr>
<td>( \dot{Q}_{g \rightarrow i} )</td>
<td>( W )</td>
<td>( \alpha_{ig} )</td>
<td>Heat transfer gas → Interface</td>
</tr>
</tbody>
</table>

State vector

continued on next page
Appendix A: Implementation of Modules

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state(1)</td>
<td></td>
<td>&lt;0</td>
<td>Water saturated ($h_l = h_{lsat}$)</td>
</tr>
<tr>
<td>state(2)</td>
<td></td>
<td>&gt;0</td>
<td>Mix of steam and air in gas space</td>
</tr>
<tr>
<td>state(5)</td>
<td></td>
<td>-</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Test run of the Interface

To verify the implementation of the interface, two test cases are presented. In both cases, the interface is supposed to be located in a vessel with a total volume of 20m$^3$, having a cross section of 2m$^2$. The volumes of the liquid and gas space are equal (10m$^3$ each).

In the first test run, the partial steam and air pressures are kept constant at $1.0 \times 10^5 Pa$. The gas temperature is kept constant as well at 450K. The enthalpy of the liquid is varied, starting at $322\frac{kJ}{kg}$ and increased up to $505\frac{kJ}{kg}$. Thus, initially the liquid is subcooled and gets superheated for the tests.

The results of the first test run are shown in figure A.3. The figure shows the mass transfer rates due to rising bubbles form the liquid phase ($\dot{M}_b$) which gets the dominant mass transfer mechanism in the case with superheated liquid. On the other hand, the mass transfer due to evaporation at the interface is still present. The positive mass flow is defined from the liquid space to the gas space. Thus, some mass from the gas space is condensing and flowing back to the liquid, due to equilibrium conditions at the interface, as the gas space temperature is always much higher than the liquid temperature.

As the temperature of the liquid is limited by the saturation temperature at the given (constant) pressure, and the gas temperature is...
Figure A.3: Test run of the interface with constant pressure

Figure A.4: Test run of the interface with constant temperature
forced at a constant value as well, the condensation at the interface remains constant at $0.035 \frac{kg}{s}$, as the gas temperature is much higher than the (saturated) liquid. On the other hand, the amount of steam escaping to the gas space due to rising bubbles increases after reaching saturation.

Compare to the next test case, where the mass flow at the interface is shown as a function of the partial steam and air pressure.

In the second test case, the temperature is kept constant at $300K$ for the gas and at $370K$ for the liquid. The total pressure is constant at $3.0 \times 10^5 Pa$. However, the partial steam pressure is increased linearly starting at almost zero, and the air partial pressure is decreased, starting at almost $3.0 \times 10^5 Pa$. As the liquid phase is always subcooled, no mass transfer due to rising bubbles takes place. All the mass transfer is due to evaporation or condensation at the interface. As the gas space is not saturated, some liquid water may evaporate into the gas space to reach equilibrium. Vice-versa, as the steam partial pressure is higher than saturation, an equilibrium can only be reached due to condensation at the interface. This behavior is clearly seen in figure A.4.

Note the point Eq when no mass flow at the interface takes place. This is at the point where the steam partial pressure in the gas space is at saturation pressure of the liquid space. As the liquid is at $370K$, the saturation pressure is $90448 Pa$.

A.3.5. The Pressure Vessel

The pressure vessel is implemented as a combination of a liquid space, a gas space and an interface in between. The respective calling interfaces to the Fortran code have been discussed in the previous sections. Thus, the implementation of the pressure vessel, e.g. a reactor pressure vessel, or a wetwell, contains no new equations or numerical solutions to any physical problems, but is a sequential calling to the appropriate modules.

The Fortran calling interface is given in table A.5. The state vector needs to be set up carefully, as the physical model used in the simulation run depends on these values. A bad set up of the state vector may result in unpredictable and preposterous results.
### Module: Pressure Vessel

#### Input parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_v$</td>
<td>kg</td>
<td>$M_v$</td>
<td>Mass of steam</td>
</tr>
<tr>
<td>$M_a$</td>
<td>kg</td>
<td>$M_a$</td>
<td>Mass of air</td>
</tr>
<tr>
<td>$p_v$</td>
<td>N/m²</td>
<td>$p_v$</td>
<td>Partial steam pressure</td>
</tr>
<tr>
<td>$p_a$</td>
<td>N/m²</td>
<td>$p_a$</td>
<td>Partial air pressure</td>
</tr>
<tr>
<td>$T_g$</td>
<td>K</td>
<td>$T_g$</td>
<td>Gas temperature</td>
</tr>
<tr>
<td>$V_l$</td>
<td>m³</td>
<td>$V_l$</td>
<td>Volume of liquid</td>
</tr>
<tr>
<td>$h_l$</td>
<td>J/kg</td>
<td>$h_l$</td>
<td>Enthalpy of liquid</td>
</tr>
<tr>
<td>$Q_{in,v}$</td>
<td>W</td>
<td>$Q_{in,v}$</td>
<td>Added heat in liquid</td>
</tr>
<tr>
<td>$Q_{in,a}$</td>
<td>W</td>
<td>$Q_{in,a}$</td>
<td>Added heat in gas</td>
</tr>
<tr>
<td>$Q_{wall,v}$</td>
<td>W</td>
<td>$Q_{wall,v}$</td>
<td>Heat loss through wall (liquid)</td>
</tr>
<tr>
<td>$Q_{wall,a}$</td>
<td>W</td>
<td>$Q_{wall,a}$</td>
<td>Heat loss through wall (gas)</td>
</tr>
<tr>
<td>$M_{out,v}$</td>
<td>kg</td>
<td>$M_{out,v}$</td>
<td>Liquid exit rate</td>
</tr>
<tr>
<td>$M_{in,v}$</td>
<td>kg</td>
<td>$M_{in,v}$</td>
<td>Liquid injection rate</td>
</tr>
<tr>
<td>$h_{in,v}$</td>
<td>J/kg</td>
<td>$h_{in,v}$</td>
<td>Enthalpy of injected liquid</td>
</tr>
<tr>
<td>$M_{in,a}$</td>
<td>kg</td>
<td>$M_{in,a}$</td>
<td>Air injection rate</td>
</tr>
<tr>
<td>$h_{in,a}$</td>
<td>J/kg</td>
<td>$h_{in,a}$</td>
<td>Enthalpy of injected air</td>
</tr>
<tr>
<td>$M_{out,a}$</td>
<td>kg</td>
<td>$M_{out,a}$</td>
<td>Gas exit rate</td>
</tr>
<tr>
<td>$A_{lg}$</td>
<td>m²</td>
<td>$A_{lg}$</td>
<td>Area of the vessel</td>
</tr>
<tr>
<td>$V_{tot}$</td>
<td>m³</td>
<td>$V_{tot}$</td>
<td>Volume of the vessel</td>
</tr>
</tbody>
</table>

#### Output parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{dM_v}{dt}$</td>
<td>kg/s</td>
<td>$dM_v$</td>
<td>Derivative of mass of steam</td>
</tr>
<tr>
<td>$\frac{dM_a}{dt}$</td>
<td>kg/s</td>
<td>$dM_a$</td>
<td>Derivative of mass of air</td>
</tr>
<tr>
<td>$\frac{dp_v}{dt}$</td>
<td>N/m²/s</td>
<td>dp_v</td>
<td>Derivative of steam partial pressure</td>
</tr>
<tr>
<td>$\frac{dp_a}{dt}$</td>
<td>N/m²/s</td>
<td>dp_a</td>
<td>Derivative of air partial pressure</td>
</tr>
<tr>
<td>$\frac{dT_g}{dt}$</td>
<td>K/s</td>
<td>$dT_g$</td>
<td>Derivative of gas temperature</td>
</tr>
<tr>
<td>$\frac{dV_l}{dt}$</td>
<td>m³/s</td>
<td>$dV_l$</td>
<td>Derivative of volume of liquid</td>
</tr>
<tr>
<td>$\frac{dh_l}{dt}$</td>
<td>J/kg/s</td>
<td>$dh_l$</td>
<td>Derivative of enthalpy of liquid</td>
</tr>
</tbody>
</table>

#### State vector

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>state(1)</td>
<td>&gt; 0</td>
<td>Water subcooled ($h_l &lt; h_{l,at}$)</td>
</tr>
</tbody>
</table>

*continued on next page*
## Table A.5: The pressure vessel module

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state(1)</td>
<td>&lt;0</td>
<td></td>
<td>Water saturated ( h_l = h_{l_{\text{sat}}} )</td>
</tr>
<tr>
<td>state(2)</td>
<td>&gt;0</td>
<td></td>
<td>Mix of steam and air in gas space</td>
</tr>
<tr>
<td>state(2)</td>
<td>&lt;0</td>
<td></td>
<td>Pure steam in gas space</td>
</tr>
<tr>
<td>state(3)</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>state(10)</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### Calling interface

```c
CALL Vessel_deriv_evt(M_v, M_a, p_v, p_a, T_g, M_l, 
# V_l, h_l, Qdotin_l, Qdotin_3, Qdotwall_1, 
# Qdotwall_3, Mdotlout, Mdotlin, hlin, Mdot_vin, 
# Mdot_vin, h_vin, h_air, Mdot_gout, A_lg, V_tot, 
# state, dM_v, dM_a, dp_v, dp_a, dT_g, dM_l, dV_l, 
# dh_l)
```

### Test run of the Pressure Vessel

The pressure vessel has been validated against a well known computer code. In figure A.5 the test results against the corresponding Gothic code [GWWC95] are shown [MH92]. In this test case a pressure vessel with a total volume of 20m\(^3\) and a cross section of 1m\(^2\) is filled to the half with water. At the beginning the water is subcooled by 10K and the steam is superheated by 10K. The initial ratio of partial air pressure to partial steam pressure is 1. The system is left untouched for 50'000s so as to approach steady state. After 50'000s the gas pressure is reduced by opening a valve for getting a flow rate of gas of 0.02 kg/s. This pressure release will cause boiling and hence an increased production of vapor. After 80'000s the valve is closed again and the system will reach another steady state after about 100'000s.

The difference of the results after 50'000s are due to a different physical model of the interface between the liquid and the gaseous phase in the vessel.

In the example presented, different events occurred. The simple cases are the a priori known points in time, when the mass flow is started and stopped, respectively. The other events are due to saturation of the liquid space inside the pressure vessel. These events are not known a priori and have to be tracked during integration of the
Figure A.5: Comparison of tests on MSDL against Gothic
system. The events during this run are summarized in table A.6. Each event causes a discontinuity and therefore a multi step integrator like LSODAR (Gear, Adams-Moulton) needs to be restarted. The discontinuities are clearly visible in the plots. The total run time of this example was 3.2 seconds\(^5\). The discrepancies are caused due to different modeling in MSDL and Gothic. In MSDL, in contrast to Gothic, only very simple lumped parameter models are used, and no condensation on the containment walls or inside the gas space are taken into account.

### A.3.6. The Gas Pipe

The model for the gas pipe is implemented following the physical description in section 2.5. Internally, the mass flow direction is always positive from the high pressure end to the low pressure end of the pipe. Thus, the implementation becomes quite simple. The calling interface is given in table A.7.

<table>
<thead>
<tr>
<th>Module: Gas Pipe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symbol</strong></td>
</tr>
<tr>
<td>( M_g )</td>
</tr>
<tr>
<td>( p_{v,1} )</td>
</tr>
<tr>
<td>( p_{a,1} )</td>
</tr>
<tr>
<td>( T_1 )</td>
</tr>
<tr>
<td>( p_{v,2} )</td>
</tr>
<tr>
<td>( p_{a,2} )</td>
</tr>
<tr>
<td>( T_2 )</td>
</tr>
<tr>
<td>( L )</td>
</tr>
<tr>
<td>( D_n )</td>
</tr>
</tbody>
</table>

\(^5\)On a standard Linux laptop, running at 2.2 GHz clock speed
Test run of the Gas Pipe Module

The gas pipe module has been tested isolated. For test purpose, a controlled pressure at the gas pipe inlet and its outlet is applied. The pressure is either constant or a simple linear function. The diameter of the pipe is kept constant at 16.0 cm which corresponds more or less to the main steam line of the PANDA test facility. The length of the pipe is set to 10 m. The initial values for the pipe inlets and outlets are as follows:

\[
\begin{align*}
  p_{V1} & = 5.0 \times 10^5 Pa \\
  p_{a1} & = 3.0 \times 10^4 Pa \\
  T_1 & = 4.7 \times 10^2 K \\
  p_{V2} & = 5.0 \times 10^5 Pa \\
  p_{a2} & = 3.0 \times 10^4 Pa \\
  T_2 & = 4.7 \times 10^2 K
\end{align*}
\]

Figure A.6 shows the summary of the test run. It is clearly shown that the mass flow rate is not a linear function of the pressure difference. With increasing pressure, the mass flow rate will flatten out to a maximum due to the limited capacity of the pipe. The test shows a run with variant overall line loss coefficients \( k \), ranging from \( k = 15.0 \) to \( k = 45.0 \), which corresponds to realistic PANDA values [Yad94].

During the tests, the partial steam pressure is varied linearly, and the other values are left constant. Thus, as seen in figure A.6, a linear variation of the pressure difference occurs.

The values in the plot can be verified by hand, applying the static mass flow equation

\[
\Delta p = \frac{1}{2} \frac{\dot{M}^2}{A^2 \rho} \left( \frac{4fL}{D_n} + k \right). \tag{A.7}
\]
Inserting the values from figure A.6 where Δp is constant at 1000Pa, e.g.\(^6\)

\[
\begin{align*}
\dot{M} & = 0.2411773420 \quad \text{[kg/s]} \\
A & = 0.0201061935 \quad \text{[m}^2]\] \\
L & = 10.0 \quad \text{[m]} \\
\rho & = 2.59821885 \quad \text{[kg/m}^3]\] \\
k & = 35.0 \\
Re & = 116502.49 \\
\bar{f} & = 0.00446160148 \\
D_n & = 0.16 \quad \text{[m}^2]\]
\]
we get \(\Delta p = 1000Pa\), as expected.

A.3.7. The Liquid Pipe

The implementation of the liquid pipe is very similar to the gas pipe. The main difference is the calling interface due to different parameters.

\(^6\)The values presented here were dumped from a test run, which generated the plot in figure A.6. The figure was calculated using the differential form of the mass flow equation, as presented in chapter 2.
needed. The calculation is more or less the same as in the gas pipe, with different mass property functions, of course. The calling interface is given in table A.8.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_l )</td>
<td>( \text{kg} )</td>
<td>( Y )</td>
<td>Actual mass flow</td>
</tr>
<tr>
<td>( p_{\text{in}} )</td>
<td>( \text{Pa} )</td>
<td>( p_{\text{in}} )</td>
<td>Pressure in source vessel</td>
</tr>
<tr>
<td>( h_{\text{in}} )</td>
<td>( \text{J} )</td>
<td>( h_{\text{in}} )</td>
<td>Liquid enthalpy in source vessel</td>
</tr>
<tr>
<td>( p_{\text{out}} )</td>
<td>( \text{Pa} )</td>
<td>( p_{\text{out}} )</td>
<td>Pressure in destination vessel</td>
</tr>
<tr>
<td>( h_{\text{out}} )</td>
<td>( \text{J} )</td>
<td>( h_{\text{out}} )</td>
<td>Liquid enthalpy in destination vessel</td>
</tr>
<tr>
<td>( L )</td>
<td>( \text{m} )</td>
<td>( L )</td>
<td>Length of pipe</td>
</tr>
<tr>
<td>( D_n )</td>
<td>( \text{m} )</td>
<td>( D_n )</td>
<td>Diameter of pipe</td>
</tr>
<tr>
<td>( k )</td>
<td>—</td>
<td>( k )</td>
<td>Overall loss coefficient</td>
</tr>
</tbody>
</table>

### Output parameter

| \( \frac{dM_l}{dt} \) | \( \text{kg} \) | \( Y_{\text{dot}} \) | Derivative of mass flow rate |

### Calling interface

```plaintext
CALL waterpipe_diff(Y, p_{\text{in}}, h_{\text{in}}, p_{\text{out}}, h_{\text{out}}, 
# L, D_n, k, Y_{\text{dot}})
```

**Table A.8: The liquid pipe module**

As the gas pipe, the liquid pipe has been tested isolated. For the test run presented here, a constant length of 15m is applied to the pipe module. The diameter of the pipe is set to 40mm, again realistic parameters for the PANDA facility as described in [Yad94].

The result of the test run is shown in figure A.7. Again, the overall line loss coefficient is varied from \( k = 5 \) up to \( k = 45 \) [Yad94].

The results can be verified again by applying the static mass flow equation A.7 on page 161. The values from the test run are as follows:

\[
\begin{align*}
\dot{M} & = 0.249669613 \, \text{kg/s} \\
A & = 0.0012566371 \, \text{m}^2 \\
L & = 15 \, \text{m} \\
\rho & = 979.62215 \, \text{kg/m}^3 \\
k & = 40 \, [-] \\
Re & = 18848.2772 \, [-] \\
f & = 0.00642249076 \, [-] \\
D_n & = 0.04 \, \text{m}
\end{align*}
\]
Figure A.7: Liquid pipe test run

and inserted into equation A.7 gives again a pressure difference of 1000Pa, as expected.

A.3.8. The Condenser

The implementation of the condenser module is divided into two parts: First, the pipe module is called to determine the overall mass flow rate through the condenser. The gas pipe module gets all parameters from the condenser feed line. As in the scope of this work, the passive behavior of the condenser, i.e. the mass flow driven by the condensation rate rather than by the pressure difference, is not covered, this simplified model can be well applied.

After the mass flow and the condenser output parameters have been calculated, the condensation of the remaining steam injected into the wetwell is calculated. As with all modules, the whole complexity of this calculations are encapsulated inside the module. The user interface of the condenser module is summarized in table A.9.
### Table A.9: The condenser module

#### Module: PCCS Condenser

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_{v, DW}$</td>
<td>Pa, DW</td>
<td>$p_{v, DW}$</td>
<td>partial steam pressure in DW</td>
</tr>
<tr>
<td>$P_{a, DW}$</td>
<td>Pa, DW</td>
<td>$p_{a, DW}$</td>
<td>partial air pressure in DW</td>
</tr>
<tr>
<td>$T_{DW}$</td>
<td>K</td>
<td>$T_{DW}$</td>
<td>Temperature in DW</td>
</tr>
<tr>
<td>$p_{v, WW}$</td>
<td>Pa, WW</td>
<td>$p_{v, WW}$</td>
<td>Partial steam pressure in WW</td>
</tr>
<tr>
<td>$P_{a, WW}$</td>
<td>Pa, WW</td>
<td>$p_{a, WW}$</td>
<td>Partial air pressure in WW</td>
</tr>
<tr>
<td>$z$</td>
<td>m</td>
<td>$z_{liq}$</td>
<td>height of water above pipe outlet</td>
</tr>
<tr>
<td>$h_{l, WW}$</td>
<td>J/kg</td>
<td>$h_{l, WW}$</td>
<td>WW liquid enthalpy</td>
</tr>
<tr>
<td>$L$</td>
<td>m</td>
<td>$L$</td>
<td>Condenser feed line length</td>
</tr>
<tr>
<td>$D_n$</td>
<td>m</td>
<td>$D_n$</td>
<td>Condenser feed line diameter</td>
</tr>
<tr>
<td>$k$</td>
<td>—</td>
<td>$k$</td>
<td>Cond. feed line overall loss koeff.</td>
</tr>
</tbody>
</table>

#### Output parameter

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{cond}$</td>
<td>kg/s</td>
<td>$M_{cond}$</td>
<td>Condensate mass flow rate to RPV</td>
</tr>
<tr>
<td>$h_{cond}$</td>
<td>J/kg</td>
<td>$h_{cond}$</td>
<td>Condensate enthalpy to RPV</td>
</tr>
<tr>
<td>$M_{l, WW}$</td>
<td>kg/s</td>
<td>$M_{l, WW}$</td>
<td>Liquid mass flow rate to WW</td>
</tr>
<tr>
<td>$h_{l, WW}$</td>
<td>J/kg</td>
<td>$h_{l, WW}$</td>
<td>Enthalpy of liquid entering WW</td>
</tr>
<tr>
<td>$M_{v, WW}$</td>
<td>kg/s</td>
<td>$M_{v, WW}$</td>
<td>Steam mass flow rate to WW</td>
</tr>
<tr>
<td>$h_{v, WW}$</td>
<td>J/kg</td>
<td>$h_{v, WW}$</td>
<td>Enthalpy of steam entering WW</td>
</tr>
<tr>
<td>$M_{a, WW}$</td>
<td>kg/s</td>
<td>$M_{a, WW}$</td>
<td>Air mass flow rate to WW</td>
</tr>
<tr>
<td>$h_{a, WW}$</td>
<td>J/kg</td>
<td>$h_{a, WW}$</td>
<td>Enthalpy of air entering WW</td>
</tr>
<tr>
<td>$Q_{WW}$</td>
<td>W</td>
<td>$Q_{WW}$</td>
<td>Heat added to WW liquid</td>
</tr>
</tbody>
</table>

#### Calling interface

```plaintext
CALL pccs(pv_DW, pa_DW, T_DW, pv_WW, pa_WW, z_liq,
#   hl_WW, L, Dn, k,
#   M_cond, h_cond,
#   Mlout_WW, hlout_WW, Mvout_WW, hvout_WW,
#   Maout_WW, haout_WW, Qout_WW)
```

---

**Test run of the PCCS Module**

The setup for the condenser test is simple. An increasing difference in pressures form condenser input (a drywell) and its exit (the end of the suppression pipe well below the surface of a wetwell suppression pool) is the driving force. As soon as the pressure difference becomes positive, the condensation starts. The geometrical parameters of the
The condenser opens only when the pressure in the DW becomes higher than the pressure in the WW. As clearly seen in figure A.8, the condensation rate may become relatively low with increasing pressure difference. This means that there is ‘not enough time’ for the condensation of all the steam with very high mass flow rates. Around the point of its design, the condensation rate is very high, however.
Figure B.1: The role of the Lexer inside the MSDL compiler
B.1. Lexical Analysis Implementation

As discussed before, the lexer has to translate an input stream into symbols which belong to the vocabulary of the input language. The role of the lexer is depicted in figure B.1. For this task, lex, or flex\(^1\) as in the actual implementation has been used. Lex (flex) is based on regular expressions. After lex matched a given symbol, standard C code is executed. The most of the lex program is given below:

Listing B.1: MSDL Lexical Analysis

```c
#include "y.tab.h"
#include <string.h>
extern int lineno;

ws  [ \t]+
comment #.*
qstring "[\"\n]*[\"\n]"r\n\]*[\"]n
bstring (r)\n\]*[\)])n
val ( [0-9]+ | ([0-9]*\. [0-9]+) ( [eE] [-+]? [0-9]+)?
nl

{Name { return NAME; }
{ws} ;
{comment} ;
{qstring} { yyval.string = strdup(yytext+1); /* skip open quote */
   if(yyval.string[yyleng-2] != '\''')
      warning("Unterminated character string",(char *)0);
   else
      yyval.string[yyleng-2] = '\0'; /* remove close quote */
      return QSTRING;
}
{bstring} { yyval.string = strdup(yytext+1); /* skip open brace */
   if(yyval.string[yyleng-2] != '\'}
      warning("Brace not closed character string",(char *)0);
   else
      yyval.string[yyleng-2] = '\0'; /* remove close brace*/
      return BSTRING;
}
{val} { yyval.value = atof(yytext); return VAL;
}
```

\(^1\)flex is a rewrite of the AT&T Unix lex tool (the two implementations do not share any code, though), with some extensions and incompatibilities, both of which are of concern to those who wish to write scanners acceptable to either implementation. Flex is fully compliant with the POSIX lex specification, except that when using (the default), a call to unput() destroys the contents of yytext, which is counter to the POSIX specification. See the manual page for flex(1) for details.
The first six lines define white space, comment lines, quoted strings, strings in braces, numerical values and the newline character as regular expressions.

The next section, after the %% symbol, defines the actions to be executed (in standard C code), when a certain symbol is matched. Thus the actions for white space and comments is blank.

When a quoted string or a string in braces is matched, the braces and quotes at the beginning and the end are deleted and the token QSTRING or BSTRING, respectively, is returned. If the quotes or braces do not match, an error message is printed out and the process will be terminated in a proper way.

When the keyword “Name” or “Vessel” is matched, the corresponding token is returned by the lexer program. This way, all possible key words of the source text are converted into a stream of tokens. This stream of tokens then is processed by the parser which checks for the syntactical correctness of the language.

In the first lines of the code, two header files called y.tab.h and string.h are included. The first, y.tab.h is generated by yacc (see next section on page 169). It contains the definition of all tokens needed by the parser. The second file, string.h is part of the GNU C Library and contains subroutines and functions for String handling2.

B.2. Parser Implementation

The parser is implemented using yacc. Yacc is a parser generator3 and provides a way to associate meanings with the components of the grammar in such a way that as the parsing takes place, the meaning can be “evaluated” as well. First, yacc needs a grammar in a form like discussed in section 5.3.2, but in a more systematic way. This defines

---

2ISO C99 Standard: 7.21 String handling <string.h>.
3yacc stands for “yet another compiler compiler”, a comment by its creator, Steve Johnson. It reflects the popularity of such programs around 1972, when a whole number of parser generators were developed. yacc is one of the few that have flourished.
— as discussed before — the syntax of the language. At this stage, yacc is able to report some syntactic errors or ambiguities in the grammar to the developer.

Then, for each production of the grammar, an action can be supplied. An action in this sense is a statement in C which tells what to do when the this production has been matched. These actions define the semantics of the language.

The grammar, written in the form for yacc, is given below. It reflects very much the form of the grammar presented on page 92. For enhancing the grammar part only, the actions, or semantic rules, have been dropped.

Listing B.2: MSDL Grammar Outline

```plaintext
class: /* nothing */
  | sname vessels v_geos pipes p_geos inits s_end
sname: NAME QSTRING
vessels: /* nothing */
  | vessels vessel
vessel: VESSEL QSTRING ',' TYPE PV
  | VESSEL QSTRING ',' TYPE GAS
v_geos: /* nothing */
  | v_geos v_geo
v_geo: VOLUME BSTRING '=' VAL
  | CROSSSECTION BSTRING '=' VAL
pipes: /* nothing */
  | pipes pipe
pipe: PIPE QSTRING ',' TYPE GAS ',' SRC QSTRING ',' DEST QSTRING
  | PIPE QSTRING ',' TYPE LIQUID ',' SRC QSTRING ',' DEST QSTRING
  | PIPE QSTRING ',' TYPE CVENT ',' SRC QSTRING ',' DEST QSTRING
  | PIPE QSTRING ',' TYPE CONDENSER ',' SRC QSTRING ',' DEST QSTRING
  | PIPE QSTRING ',' TYPE VENT ',' SRC QSTRING ',' DEST QSTRING
p_geos: /* nothing */
  | p_geos p_geo
p_geo: LENGTH BSTRING '=' VAL
  | DIAMETER BSTRING '=' VAL
  | K BSTRING '=' VAL
  | DEPTH_IN BSTRING '=' VAL
  | DEPTH_OUT BSTRING '=' VAL
inits: /* nothing */
  | inits init
init: INITPG BSTRING '=' VAL
  | INITPA BSTRING '=' VAL
  | INITLIQLEVEL BSTRING '=' VAL
```
All the terminal symbols used in the above language definition, have to be defined. They are exported to lexer in a header file, which is generated automatically.

After a production has been detected, and resolved down to terminal symbols, the corresponding actions (a set of C instructions) will be executed. This is used to build up the intermediate data structure, the syntax tree.

To complete this section, the semantic rules for the productions involving the vessel, are given below. Note, in contrast to the upper code, this code including the actions is a part of the “real” yacc code.

Listing B.3: MSDL Grammar implementation

```
vessels: /* nothing */
    | vessels vessel

vessel: VESSEL QSTRING ',' TYPE PV
    { vp = (VesselPtr) malloc(sizeof(Vessel));
        vp->name = $2;
        vp->type = PV;
        vp->connect[0] = NULL;
        vp->numofpipe = 0;
        vp->next = NULL;
        if ((VesselPtr) LookupVessel(sys, $2) != NULL)
        {
            fprintf(stderr, "Warning: Duplicate vessel name %s.\n", $2);
        }
        i = InsertVessel(vp, sys);
    }

    | VESSEL QSTRING ',' TYPE GAS
    { vp = (VesselPtr) malloc(sizeof(Vessel));
        fprintf(stderr, "inserting GAS...\n");
        vp->name = $2;
        vp->type = GAS;
        vp->connect[0] = NULL;
        vp->numofpipe = 0;
        vp->next = NULL;
        if ((VesselPtr) LookupVessel(sys, $2) != NULL)
        {
            fprintf(stderr, "Warning: Duplicate vessel name %s.\n", $2);
        }
        i = InsertVessel(vp, sys);
    }
```

As an example, a simple system to be simulated is described by the
program code below. The system consists of a reactor pressure vessel (RPV), a drywell and a wetwell. The reactor is connected to the drywell by a gas pipe, and the drywell is connected to a PCCS condenser, which transfers the condensate back to the reactor pressure vessel, and the remaining gas (air and steam) is vented to the wetwell where most of the remaining steam condenses in the suppression pool. This system is depicted in figure B.2. The initial conditions are to be read directly from the code.

Listing B.4: MSDL P2 Setup code example

```plaintext
# Description file for simulation setup of PCCS test
# Created Feb 11, 1999

Name "p2_1"

Vessel "RPV", Type = PV
Vessel "DWl", Type = Gas
Vessel "WWl", Type = PV

Volume (RPV) = 22.8
Crosssection(RPV) = 1.2271846

Volume(DWl) = 90.0
Crosssection(DWl) = 7.161972

Volume(WWl) = 117.0
Crosssection(WWl) = 12.5663711

Pipe "p1", Type = Gas, Src="RPV", Dest="DWl"
Pipe "p2", Type = PCCS, Src="DWl", Dest_G = "WWl", Dest_L = "RPV"

Length(p1) = 11.271
Diameter(p1) = 0.1631
k (p1) = 10.13

Length(p2) = 7.413
Diameter(p2) = 0.0843
k (p2) = 9.08
Depth_out(p2) = 2.876

InitPg(RPV) = 130000
InitPa(RPV) = 1000.0
InitLiqLevel(RPV) = 6.13
InitTLiq(RPV) = 104.8

InitPg(DWl) = 130000
InitPa(DWl) = 126436

InitPg(WWl) = 130000
InitPa(WWl) = 80000
InitLiqLevel(WWl) = 4.0
InitTLiq(WWl) = 82.0

End "p2_1"
```
Figure B.2: Example setup for code p2.1
B.3. Intermediate Data Structures

In classical compiler construction, a data structure has to be defined which allows the parsing of the code on one hand and code generation on the other hand. This data structure can be edited in various ways: using a parser which interprets a program text, an interactive command line interface, or a graphical interface. If more instances are allowed simultaneously, a control flow has to be defined. This classical approach, known as MVC (Model-View-Control) has been used for the implementation of a compiler. But since the data structure is designed in this way, a graphical user interface (GUI) can be added without changing any intermediate data structure or program code for the code generator module.

As the data structures needed to consist of more complex types than just integers and floating point values, a more sophisticated language such as C was used. Furthermore, C allows dynamic allocation of memory which is needed if the complexity of the system is not known in advance. (In Fortran for example, all data structures and variables have to be declared in advance, which forces the implementation of some upper limits.)

The data structures have to reflect the modeling of the thermal hydraulic system we want to simulate. Thus, abstract data types for vessels and pipes are needed. A vessel may consist of pressure vessels or of gas spaces and various pipes are allowed, so a number of data types has been defined. As an example, the definition of the vessel structure is given below:

Listing B.5: Compiler internal vessel data structure

```c
typedef struct _vessel {
    char *name;        /* Name of vessel */
    int type;          /* 0 Gas, 1: pv */
    PipePtr connect[10]; /* Connected Pipes */
    int numofpipe;     /* How many connections */
    union vessel_geom { /* Geometry of Vessels */
        Geom_Gasspace geom__gasspace;
        Geom_PV geom_pv;
    } Geometry;
    union vessel_init { /* Initial values */
        Init_Gasspace init__gasspace;
        Init_PV init_pv;
    } Init;
    VesselPtr next;     /* Next vessel in list */
} Vessel;
```

This data type allows to create a list of all vessels in the system which is linked and thus allows a very simple parsing of the vessels as
well as a simple insertion and deletion of vessels dynamically at runtime. This data structure makes use of union constructs which allow to "multiplex" two types and use one of more possibilities, as appropriate. The data types for pipes or for the whole system are implemented in the same way.

To see the linking of the data structures after parsing a program text, figure B.3 shows the built-up dynamic data structure after parsing the program p2.l above. For clarity, all the cross references are not displayed as arrows, but can be followed by the data addresses displayed in the fields 'connect'. On these addresses, a simple consistency check can be applied: A gas or liquid pipe needs two connect entries which point to a vessel, whereas a condenser needs three connecting entries pointing to vessels. One more check consists in checking the connections in all vessels which must be the same as all the connecting entries in the pipes.

**B.4. Overall Architecture of the Front end**

To build up the intermediate data structure, a program is needed which parses the program text for a thermal hydraulic system. This part of the compiler is called parser. The parser can be built to parse an ASCII file, a graphical user interface, or in some way an interactive system which might ask the user about the system to be built up. The simplest way is to build an ASCII parser which goes through a data file, as the one given in the example above which describes the system p2.l. This parser is based on Lex and Yacc, as mentioned before.

The parser, generated automatically from a syntactical definition described above, is based on a subroutine which performs a lexicographical analysis of the code. Its task is to read the input file, to recognize all the needed key words and to convert them into numerical tokens. This is not a trivial task, as this subroutine should remain expandable, human readable and should handle things like subexpressions and recursions.

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching most of the text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined, the text corresponding to the match
Figure B.3: The actual intermediate data structure of example p2_1
### B.4. Overall Architecture of the Front end

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>InsertVessel</td>
<td>(VesselPtr vessel, SystemPtr system)</td>
</tr>
<tr>
<td>VesselPtr</td>
<td>LookupVessel</td>
<td>(SystemPtr system, char *name)</td>
</tr>
<tr>
<td>PipePtr</td>
<td>LookupPipe</td>
<td>(SystemPtr system, char *name)</td>
</tr>
<tr>
<td>int</td>
<td>InsertPipe</td>
<td>(PipePtr pipe, SystemPtr system)</td>
</tr>
<tr>
<td>SystemPtr</td>
<td>CreateSystem</td>
<td>(char* name)</td>
</tr>
</tbody>
</table>

*Table B.1: Support Routines for MSDL Parser.*

(called the token) is made available in the global character pointer yytext, and its length in the global integer yyleng. The action corresponding to the matched pattern is then executed, and then the remaining input is scanned for another match.

Each pattern in a rule has a corresponding action, which can be any arbitrary C statement. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. If the action is empty, then when the pattern is matched the input token is simply discarded.

Actions can include arbitrary C code, including return statements to return a value to whatever routine called yylex(). Each time yylex() is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return.

### B.4.1. Compiling the MSDL Compiler

First, the lexer has to be defined in the lex source file compiler.1. This file then can be compiled in a C code with the command lex -l compiler.1. The -l flag tells lex, or flex, respectively, to turn on maximum compatibility with the original AT&T lex implementation. The output is a C program which implements the MSDL lexer.

The parser has to be created next. It is defined, as discussed, in the yacc language. The definition is left in the file compiler.y. The yacc command is read yacc -d compiler.y. The -d option causes the header file y.tab.h to be written automatically by yacc. The output of this command is the C source of the MSDL parser in the file y.tab.c.

The support routines are written in separate files. compiler.c contains the support routines given in table B.1 for creating the intermediate data structure.

The last file needed is codegen.c. This is discussed in detail in the next section.
Building the compiler is finally done by compiling and linking all source files:
```
cc -g codegen.c compiler.c lex.yy.c y.tab.c
```
The flag `-d` enables the C compiler to write additional symbol information needed for debuggers. The purpose of a debugger such as GDB\(^4\) or DDD\(^5\) is to allow you to see what is going on “inside” another program while it executes – or what another program was doing at the moment it crashed.

### B.5. Code Generator

By means of compiler constructions, a classical code generator generates a so-called object code in the assembly language of the appropriate target system. The code generator implemented here has to create another sort of output: Fortran code. This code has to be well documented, well structured and must be easily understandable and editable.

The code makes use of the modules library described in chapter A and of the integrator package described in chapter 3. The resulting code can be compiled and linked to get an executable which runs the actual simulation of the system.

The code generator can be divided into several sections according to the usage of the LSODAR integrator:

- **FinishStructure**(system); Finishes the data structure from figure B.3 and performs some consistency checks.

- **CodeGen_MAIN**(system); Creates the main program including the integrator loop.

- **CodeGen_F1**(system); Generates the subroutine calculating the derivatives of the whole system. This routine in fact calls the modules from the library described in chapter A.

- **CodeGen_GR1**(system); Provides the root functions and other events we would like to track.

- **CodeGen_JAC**(system); Is the Jacobian subroutine to calculate the Jacobian, if needed.

\(^4\)The GNU Debugger, Copyright (C) 1988-2000 Free Software Foundation, Inc.

\(^5\)DDD is a graphical front-end for GDB and other command-line debuggers.
• **CodeGen_INIT(system)**; Initialization of the whole system.

• **CodeGen_PRINT(system)**; Printout of all the state variables and some consistency check printouts, such as the overall mass of the system to check the conservation of mass during integration.

Each of this sections has to complete the same scheme on the main data structure, given in figure B.3. The first step is to print out the appropriate program or subroutine header, as appropriate. Then, the system is parsed for all the pipes, then for the vessels. During this scan, some declarations of variables are printed out to the output file, as well as some internal data is evaluated, such as the actual number of vessels, number of pipes, etc.

During the next scan of the data structure, the base addresses of the state variables for each of the modules involved is generated.

Then, depending on which part of the code generator is running, the appropriate program text is printed to the output file.

**B.6. Output File Format**

The output file is a plain ASCII text containing standard Fortran 77 program text with some Fortran 90 extensions. The used extensions are only for the simplification of the text, thus are making the program text more readable. As this code is generated and not user-written, a change to C or any other programming language is possible by rewriting only the code generator part of the compiler.

A full printout of an example, containing the source MSDL File and the automatically generated Fortran Code, can be found in appendix D on page 187.
This appendix provides an user’s guide to the presented concepts, introducing the Modular System Description Language (MSDL). In section C the MSDL language is introduced. Developing a simple system it will be shown how to use the language, how to modify the generated simulation code and where the results can be found and plotted.

The design issue of the MSDL language was to provide an easy-to-use description language for modular systems. The language is compiled into an intermediate Fortran code (may be later into a corresponding C Code), providing all necessary routines to simulate the system described. This includes subroutines for the time integration, finding of events, and output routines. Besides that, the source code must be in a well readable form for human readers. Thus some comments describing the routines and variables have to be provided. This comments allow for a relatively easy manipulation of the source code. This manipulation may include some more sophisticated input conditions, not yet implemented in the compiler or the printout of some
variables for debugging purpose.

C.1. Syntax of MSDL

The Syntax of the existing MSDL language is given in the BNF description below. A BNF is a set of rules, whose left side is the name of a syntactical construct. The right side of each rule contains a possible form for that syntactic construct. The right sides may contain names of other syntactic constructs or character sequences that must occur literally.

If the right hand side contains forms separated by a pipe character (\(|\)) means that one of the forms can occur. Square brackets ([]) indicate that a part is optional, i.e. the form might be present or missing. A form in curly braces ({}) means that the form may occur arbitrarily often, including zero times.

1. system = sname vessels v_geos pipes p_geos inits s_end.
2. sname = ‘Name’ qstring.
3. qstring = ” {A-Za-z} [A-Za-z0-9] ”.
4. bstring = ‘( {A-Za-z} [A-Za-z0-9] )’.
5. vessels = {vessel}.
6. vessel = ‘Vessel’ qstring ’ Type’ vesseltype.
7. vesseltype = ‘PV’ | ‘Gas’.
8. v_geos = {v_geo}
9. v_geo = geo_v bstring ’ =’ val.
10. geo_v = ‘Volume’ | ‘Crosssection’.
11. val =
12. pipes = {pipe}.
13. pipe = g_pipe | l_pipe | cvent | vent.
14. gpipe = ‘Pipe’ qstring ’ type’ ‘gas’ ’ src’ qstring ’ dest’ qstring.
To describe a model in MSDL the code must follow the above given grammar. Thus, a MSDL source file can be split up into seven sections according to rule 1 in the above given grammar. The needed language elements will be described at this point not in a syntactical but in a semantical way. To follow the development of code, we will construct a simple system composed of one reactor pressure vessel (RPV) and a dry well (DW), connected with one pipe from RPV gas space to the DW. Please note that all blank lines or lines beginning with a '#' are comments and will be ignored by the compiler.

First, the system name has to be identified:

```
# Section 1: Name of the system.
Name "Example"
```

The next section declares all the vessels in the system:

```
# section 2: Declaration of the vessels:
Vessel "RPV", Type = PV
Vessel "DW", Type = Gas
```
Then the vessel’s geometrical characteristics have to be defined:

```
# section 3: Geometrical parameters of vessels:
Volume (RPV) = 22.8
Crosssection (RPV) = 1.2271846

Volume(DW) = 90.0
Crosssection(DW) = 7.161972
```

### C.3. Compilation of MSDL Files

The compilation of the MSDL Source is invoked by the command `compiler p3.sim`. The command gives some informal output on the terminal while parsing the source code:

```
Inserting RPV in system p3.
inserting GAS...
Inserting DW1 in system p3.
Appending DW1 to RPV.
inserting GAS...
Inserting DW2 in system p3.
Appending DW2 to DW1.
Inserting WW1 in system p3.
Appending WW1 to DW2.
Inserting WW2 in system p3.
Appending WW2 to WW1.
inserting GAS...
Inserting GDCS in system p3.
Appending GDCS to WW2.
Inserting MSL2 in system p3.
Inserting PCS22 in system p3.
Appending PCS22 to MSL2.
Inserting PCS3 in system p3.
Appending PCS3 to PCS22.
Inserting VENT1 in system p3.
Appending VENT1 to PCS3.
Inserting VENT2 in system p3.
Appending VENT2 to VENT1.
Inserting P1 in system p3.
Appending P1 to VENT2.
Inserting P2 in system p3.
Appending P2 to P1.
Inserting P3 in system p3.
Appending P3 to P2.
Inserting AUX1 in system p3.
Appending AUX1 to P3.
Processing p3 successful.
```

```
Code Generator for MAIN of p3
There are 18 roots and 44 ODEs in the system
```

```
Code Generator for F1 of p3
Code Generator for GR1 of p3
Code Generator for JAC of p3
Code Generator for INIT of p3
Code Generator for PRINTOUT of p3
```
The output shows the activity of the compiler: First the system is analyzed, the vessels are parsed, the pipes are connected to the vessels. After processing the system, the code generator for the main program gets activated. As the output shows, there are 18 roots (discontinuities) in the system, and the system is described by 44 ODEs.

Some diagnostic output, such as syntax errors or warnings (as for open pipes) in the MSDL file are also reported together with the line number where the error occurred.

The output of the compiler is written to a file `screen.out` by default. That file is a Fortran file which might be edited before compiled and executed.

### C.4. Edit Generated Fortran Files

The generated Fortran code may be edited for some reasons. As an example, MSDL does not yet provide a language element for describing heat curves. Thus, the input heat has to be added to the program. The correct place for this can easily be found as the code is well commented.

The place where heat flows into the vessels is coded is located

```fortran
C--- Calculate heat flows into vessels.
C
C RPV
Qdotin_1_RPV = 0.0D+0
Qdotin_3_RPV = 0.0D+0
Qdotwall_1_RPV = 0.0D+0
Qdotwall_3_RPV = 0.0D+0
```

and can be modified in standard Fortran 77 language syntax. The resulting file should be saved as a Fortran source code file, e.g. `p3.f`

### C.5. Compile Fortran Model

The Fortran File is compiled using any standard Fortran 77 compiler. In the presented work, the GNU Fortran (GCC) 3.3.1 has been used.

### C.6. Output Files and Debugging Information

The compiled Fortran file may now be executed on the target system. At the simulation run produces lots of output, it is recommended to
redirect its output to a file, e.g. `a.out > runlog`. The runlog contains all valuable information about roots, all warnings and lots of diagnostic output.

The state variables are written to data files `fort.*`. These files contain on each line the time as well as the actual value of the corresponding state variable.

### C.7. Plotting Results using Gnuplot

The results presented in this work have been plotted using gnuplot. As an example, the following program has been used to plot the calculated and measured pressure values in DW2 and WW2:

#### Listing C.1: Gnuplot program for plotting Data

```
set xrange [0:30000]
set xlabel 'time [s]'
set key below
set grid
set terminal postscript eps enhanced dashed "AvantGarde-Demi" 20

# Total Pressures in Vessels
set output 'totalpres.eps'
set title 'Total Pressure DW2 and WW2'
set ylabel 'Total Pressure [Pa]
plot 'fort.17' title 'DW1 calculated' with lines lt 2 lw 2, 
    'fort.32' title 'WW1 calculated' with lines lt 3 lw 2, 
    'P3 file9.txt' using 1:($4*100000) ti tie 'DW1 measured' 
    with lines lt 4 lw 2, 
    'P3 file9.txt' using 1:($5*100000) title 'WW1 measured' 
    with lines lt 5 lw 2
```
D.1. MSDL Source

Hallo das ist Text. omplex thermal hydraulic systems, a language is needed with which a system can be described. This language has to be simple if it is to be used by an unexperienced user who usually has no computer science background. Thus, this language should hide as much of the overall modules and integration complexity as possible. On the other hand, the output code produced has to be a

Listing D.1: Example MSDL Listing for p3g-Setup

```
# Description file for simulation setup P3G equivalent
# Created Jan 12, 1999
#
Name "p3g"

Vessel "RPV", Type = PV
Vessel "WW2", Type = PV
Vessel "WW1", Type = PV
```
Appendix D: An Example MSDL Program

Volume (RPV) = 22.8
Volume(WW1) = 117.0
Volume(WW2) = 117.0
Crosssection(RPV) = 1.2271846
Crosssection(WW1) = 7.161972
Crosssection(WW2) = 7.161972

Pipe "p1", Type = Liquid, Src="WW1", Dest="RPV"
Pipe "p2", Type=Liquid, Src="RPV", Dest = "WW2"

Length(p1) = 5.15
Diameter(p1) = 0.93
k (p1) = 1.5
Depth_in(p1) = 3.2
Depth_out(p1) = 3.2

Length(p2) = 10.0
Diameter(p2) = 0.16
k (p2) = 10.0
Depth_in(p2) = 1.2
Depth_out(p2) = 2.1

InitPg(RPV) = 350000
InitPa(RPV) = 1000.0
InitLiqLevel(RPV) = 6.13
InitTLiq(RPV) = 104.8

InitPg(WW1) = 350001
InitPa(WW1) = 100000.0
InitLiqLevel(WW1) = 4.0
InitTLiq(WW1) = 20.5

InitPg(WW2) = 350002
InitPa(WW2) = 200000.0
InitLiqLevel(WW2) = 4
InitTLiq(WW2) = 30.8

End "p3g"

D.2. Generated Fortran Code

Listing D.2: Fortran Output Code for p3g-Setup

C This file is automatically generated by
C the system compiler(fm).
C DO NOT EDIT THIS FILE!

PROGRAM MAIN
implicit none
EXTERNAL Fl, GR1, JAC
INTEGER base_RPV, base_WW2, base_WW1
INTEGER base_pl, base_p2
PARAMETER (base_RPV = 1, base_WW2 = 9, base_WW1 = 17)
PARAMETER (base_pl = 25, base_p2 = 26)

INTEGER IOPT, IOUT, ISTATE, ITASK, ITOL, IWORK, JROOT, JT,
1 LENW, LENRW, LIW, LRW, NEQ, NG,
2 NFE, NFRA, NGE, NJE, NST
INTEGER i
DOUBLE PRECISION ATOL, RTOL, RWORK, T, TOUT, Tlast, Y
DIMENSION Y(26), RWORK(5000), IWORK(400), JROOT(8)
DIMENSION RTOL(26), ATOL(26)
include 'lsodar.com'

C SET ALL INPUT PARAMETERS AND PRINT HEADING.

CALL init(Y)
CALL printout(0.0D+0, Y)

NEQ = 26
T = 0.0D+0
TOUT = 1.0D+0
ITOL = 4
RTOL(base_RPV + 0) = 1.0D-3
RTOL(base_RPV + 1) = 1.0D-3
RTOL(base_RPV + 2) = 1.0D-3
RTOL(base_RPV + 3) = 1.0D-3
RTOL(base_RPV + 4) = 1.0D-3
RTOL(base_RPV + 5) = 1.0D-3
RTOL(base_RPV + 6) = 1.0D-3
RTOL(base_RPV + 7) = 1.0D-3
RTOL(base_WW2 + 0) = 1.0D-3
RTOL(base_WW2 + 1) = 1.0D-3
RTOL(base_WW2 + 2) = 1.0D-3
RTOL(base_WW2 + 3) = 1.0D-3
RTOL(base_WW2 + 4) = 1.0D-3
RTOL(base_WW2 + 5) = 1.0D-3
RTOL(base_WW2 + 6) = 1.0D-3
RTOL(base_WW2 + 7) = 1.0D-3
RTOL(base_WW1 + 0) = 1.0D-3
RTOL(base_WW1 + 1) = 1.0D-3
RTOL(base_WW1 + 2) = 1.0D-3
RTOL(base_WW1 + 3) = 1.0D-3
RTOL(base_WW1 + 4) = 1.0D-3
RTOL(base_WW1 + 5) = 1.0D-3
RTOL(base_WW1 + 6) = 1.0D-3
RTOL(base_WW1 + 7) = 1.0D-3
RTOL(base_pl + 0) = 1.0D-1
RTOL(base_p2 + 0) = 1.0D-1

ATOL(base_RPV + 0) = 1.0D-4
ATOL(base_RPV + 1) = 1.0D-4
ATOL(base_RPV + 2) = 1.0D-4
ATOL(base_RPV + 3) = 1.0D-4
ATOL(base_RPV + 4) = 1.0D-4
ATOL(base_RPV + 5) = 1.0D-4
ATOL(base_RPV + 6) = 1.0D-4
ATOL(base_RPV + 7) = 1.0D-4
ATOL(base_WW2 + 0) = 1.0D-4
ATOL(base_WW2 + 1) = 1.0D-4
ATOL(base_WW2 + 2) = 1.0D-4
ATOL(base_WW2 + 3) = 1.0D-4
ATOL(base_WW2 + 4) = 1.0D-4
ATOL(base_WW2 + 5) = 1.0D-4
ATOL(base_WW2 + 6) = 1.0D-4
ATOL(base_WW2 + 7) = 1.0D-4
ATOL(base_WW1 + 0) = 1.0D-4
ATOL(base_WW1 + 1) = 1.0D-4
ATOL(base_WW1 + 2) = 1.0D-4
ATOL(base_WW1 + 3) = 1.0D-4
ATOL(base_WW1 + 4) = 1.0D-4
ATOL(base_WW1 + 5) = 1.0D-4
ATOL(base_WW1 + 6) = 1.0D-4
ATOL(base_WW1 + 7) = 1.0D-4
ATOL(base_p1 + 0) = 1.0D-2
ATOL(base_p2 + 0) = 1.0D-2

ITASK = 1
ISTATE = 1
IOPT = 1
LRW = 5000
LIW = 100
JT = 2
NG = 8

C-- Set all the optional Inputs to lsodar
rwork(5) = 0.0D+0
rwork(6) = 0.0D+0
rwork(7) = 0.0D+0
rwork(8) = 0.0D+0
rwork(9) = 0.0D+0
rwork(10) = 0.0D+0
iwork(5) = 0
iwork(6) = 0
iwork(7) = 0
iwork(8) = 0
iwork(9) = 0
iwork(10) = 0
Tlast = T

C CALL LSODAR IN LOOP OVER TOUT VALUES 1 .. 40'000

DO IOUT = 1,1000000
   CALL LSODAR(F1,NEQ,Y,T,TOUT,ITOL,RTOL,ATOL,ITASK,ISTATE,1
   IOPT,RWORK,LRW,IWORK,LIW,JAC,JT,GR1,NG,JROOT)
   print *, 'Time is', t, 'Stepsize: ', rwork(11), ' s,'
   # next Stepsize will be', rwork(12), ' s.'
   IF (Tlast .le. t) THEN
      CALL printout(t, Y)
      Tlast = Tlast + 1.0D+1
   ENDIF
   IF ((ISTATE .LT. 0) .OR. (TOUT .GE. 18000)) EXIT
191

D.2. Generated Fortran Code

(ISTATE

IF

i

DO

3)

.EQ.

(*,*)

WRITE

found

(JROOT(i)

IF

ISTATE
ITASK

PROBLEM

1

3)

.NE.

TOUT

=

THEN

1.OD+2

+

IF

COMPLETE.

PRINT

=

IWORK(11)

NFE

=

IWORK(12)

NJE

=

IWORK(13)

NGE

=

STATISTICS.

FINAL

IWORK(10)

LENRW

=

LENIW

=

NFEA

IWORK(17)
IWORK(18)
NFE

=

(JT

WRITE

2)

.EQ.

(*,190)

NFEA

NFE

=

NEQ*NJE

-

LENRW,LENIW,NST,NFE,NFEA,NJE,NGE

FORMAT(//,IX,'

190

2

=

=

NST

IF

THEN

-roots(i)

=

DO

END

C

i=l,NG)

IF

(ISTATE

IF

TOUT
END

T,(JROOT(i),

',

DO

END

ELSE

1)

.EQ.

roots(i)
END

at

NG

1,

=

THEN

'Root

STATISTICS

FINAL

1

IX,'

RWORK

2

IX,'

NUMBER

OF

STEPS

3

IX,'

NUMBER

OF

F-S

SIZE

=

IX,'

5

IX,'

NUMBER

OF

J-S

6

IX,'

NUMBER

OF

G-S

.

.

(EXCLUDING J-S)

4

FOR

IWORK

=',14,

=

.

.

.

17

/,

17

/,

17

/,
/,

17

THIS

RUN..',/,
,14,/,

SIZE

17

END

C

FI

SUBROUTINE

implicit
INTEGER
DOUBLE

Local

(NEQ,

PROGRAM.

MAIN

YDOT)

T,

Y,

Y,

YDOT

none

NEQ
PRECISION

DIMENSION
C

OF

END

Y(*)

Variables

,

for

DOUBLE

PRECISION

DOUBLE

PRECISION

DOUBLE

PRECISION

DOUBLE

PRECISION

DOUBLE

PRECISION

T,

YDOT(*)

liquid pipe pi,

connecting

WWl

to

RPV.

p_in_pl,

h_in_pl
p_OUt_pl, h_OUt_pl
L_pl, Dn_pl, k_pl
Mdot_pl, h_pl
L_sup_in_pl, L_Sup_OUt_pl

dir_pl
5.15, Dn_pl
(L_pl
PARAMETER
3.2
(L_sup_in_pl
PARAMETER
3.2
(L_Sup_OUt_pl
INTEGER

PARAMETER

=

=

=

C

Local

Variables

for

DOUBLE

PRECISION

DOUBLE

PRECISION

DOUBLE

PRECISION

DOUBLE

PRECISION

DOUBLE

PRECISION

INTEGER

dir_p2
(L_p2

PARAMETER

=

0.93,

-

liquid pipe p2,

k_pl

=

1.5)

/ 2.0)

Dn_pl

-

Dn_pl

/ 2.0)

connecting

RPV

p_in_p2,

h_in_p2
p_OUt_p2, h_OUt_p2
L_p2, Dn_p2, k_p2
Mdot_p2, h_p2
L_sup_in_p2, L_Sup_OUt_p2
=

10,

Dn_p2

=

0.16,

k_p2

=

10)

to

WW2.


PARAMETER (L_sup_in_p2 = 1.2 - Dn_p2 / 2.0)
PARAMETER (L_sup_out_p2 = 2.1 - Dn_p2 / 2.0)

C Local Variables for vessel RPV.
DOUBLE PRECISION Qdotin_1_RPV, Qdotin_3_RPV, Qdotwall_1_RPV
DOUBLE PRECISION Qdotwall_3_RPV, Mdotlout_RPV, Mdotlin_RPV
DOUBLE PRECISION hlin_RPV, Mdot_vin_RPV, Mdot_ain_RPV, h_vin_RPV
DOUBLE PRECISION h_ain_RPV, Mdot_gout_RPV, A_lg_RPV
DOUBLE PRECISION V_tot_RPV
INTEGER state_RPV
DIMENSION state_RPV(10)
PARAMETER (A_lg_RPV = 1.22718, V_tot_RPV = 22.8)

C Local Variables for vessel WW2.
DOUBLE PRECISION Qdotin_1_WW2, Qdotin_3_WW2, Qdotwall_1_WW2
DOUBLE PRECISION Qdotwall_3_WW2, Mdotlout_WW2, Mdotlin_WW2
DOUBLE PRECISION hlin_WW2, Mdot_vin_WW2, Mdot_ain_WW2, h_vin_WW2
DOUBLE PRECISION h_ain_WW2, Mdot_gout_WW2, A_lg_WW2
DOUBLE PRECISION V_tot_WW2
INTEGER state_WW2
DIMENSION state_WW2(10)
PARAMETER (A_lg_WW2 = 7.16197, V_tot_WW2 = 117)

C Local Variables for vessel WW1.
DOUBLE PRECISION Qdotin_1_WW1, Qdotin_3_WW1, Qdotwall_1_WW1
DOUBLE PRECISION Qdotwall_3_WW1, Mdotlout_WW1, Mdotlin_WW1
DOUBLE PRECISION hlin_WW1, Mdot_vin_WW1, Mdot_ain_WW1, h_vin_WW1
DOUBLE PRECISION h_ain_WW1, Mdot_gout_WW1, A_lg_WW1
DOUBLE PRECISION V_tot_WW1
INTEGER state_WW1
DIMENSION state_WW1(10)
PARAMETER (A_lg_WW1 = 7.16197, V_tot_WW1 = 117)

C Base indexes for modules (Equations).
INTEGER base_RPV, base_WW2, base_WW1
INTEGER base_pl, base_p2

PARAMETER (base_RPV = 1, base_WW2 = 9, base_WW1 = 17)
PARAMETER (base_pl = 25, base_p2 = 26)

C Other local variables needed.
DOUBLE PRECISION g
PARAMETER (g = 9.81)

C Steam table functions.
DOUBLE PRECISION hptd_si, h_air_si

include "lsodar.com"

C Calculate input conditions for/of pipes
C-----------------------------------------------

C Pipe p1, liquid, WW1 --> RPV:
p_in_pl = Y(base_WW1 + 2) + Y(base_WW1 + 3) + 
# Y(base_WW1 + 5)*g / Y(base_WW1 + 6) * 
# (Y(base_WW1 + 6) / A_lg_WW1 - L_sup_in_pl) 

h_in_pl = Y(base_WW1 + 7) 

p_out_pl= Y(base_RPV + 2) + Y(base_RPV + 3) + 
# Y(base_RPV + 5)*g / Y(base_RPV + 6) * 
# (Y(base_RPV + 6) / A_lg_RPV - L_sup_out_pl) 

h_out_pl= Y(base_RPV + 7) 

IF (Y(base_pl + 0) .GT. 0.0D+0) THEN 
C Positive flow from WW1 to RPV 

dir_pl = 1 
h_pl = h_in_pl 
Mdot_pl = Y(base_pl + 0); 
ELSE 

dir_pl = -1 
h_pl = h_out_pl 
Mdot_pl = -Y(base_pl + 0); 
END IF 

C print *, 'p1...', t, Y(base_pl + 0), Mdot_pl 

C Pipe p2, liquid, RPV --> WW2: 

p_in_p2 = Y(base_RPV + 2) + Y(base_RPV + 3) + 
# Y(base_RPV + 5)*g / Y(base_RPV + 6) * 
# (Y(base_RPV + 6) / A_lg_RPV - L_sup_in_p2) 

h_in_p2 = Y(base_RPV + 7) 

p_out_p2= Y(base_WW2 + 2) + Y(base_WW2 + 3) + 
# Y(base_WW2 + 5)*g / Y(base_WW2 + 6) * 
# (Y(base_WW2 + 6) / A_lg_WW2 - L_sup_out_p2) 

h_out_p2= Y(base_WW2 + 7) 

IF (Y(base_p2 + 0) .GT. 0.0D+0) THEN 
C Positive flow from RPV to WW2 

dir_p2 = 1 
h_p2 = h_in_p2 
Mdot_p2 = Y(base_p2 + 0); 
ELSE 

dir_p2 = -1 
h_p2 = h_out_p2 
Mdot_p2 = -Y(base_p2 + 0); 
END IF 

C print *, 'p2...', t, Y(base_p2 + 0), Mdot_p2 

C Calculate flows and Enthalpies into vessels. 

C RPV 

Mdot_gout_RPV = 0.0D+0 
Mdot_vin_RPV = 0.0D+0 
Mdot_ain_RPV = 0.0D+0 
h_vin_RPV = 0.0D+0 
h_ain_RPV = 0.0D+0 
Mdotlout_RPV = 0.0D+0 
Mdotlin_RPV = 0.0D+0 
hlin_RPV = 0.0D+0 

IF (dir_pl .gt. 0 .AND. Mdot_pl .GT. 1.0D-12) THEN
\[
\text{hlin}_{\text{RPV}} = \left( h_{\text{p1}} \cdot M\text{dot}_{\text{p1}} + h\text{lin}_{\text{RPV}} \cdot M\text{dot}_{\text{lin}_{\text{RPV}}} \right) / (M\text{dot}_{\text{lin}_{\text{RPV}}} + M\text{dot}_{\text{p1}}) \\
\]

\[
\text{Mdotlin}_{\text{RPV}} = M\text{dot}_{\text{lin}_{\text{RPV}}} + M\text{dot}_{\text{p1}}
\]

\begin{verbatim}
END IF
\end{verbatim}

\begin{verbatim}
IF (dir_{p1} .lt. 0) THEN
    Mdotout_{RPV} = Mdotout_{RPV} + M\text{dot}_{\text{p1}}
END IF
\end{verbatim}

\begin{verbatim}
IF (dir_{p2} .lt. 0 .AND. Mdot_{p2} .GT. 1.0D-12) THEN
    hlin_{RPV} = \left( h_{\text{p2}} \cdot M\text{dot}_{\text{p2}} + h\text{lin}_{\text{RPV}} \cdot M\text{dot}_{\text{lin}_{\text{RPV}}} \right) / (M\text{dot}_{\text{lin}_{\text{RPV}}} + M\text{dot}_{\text{p2}}) \\
    \text{Mdotlin}_{\text{RPV}} = M\text{dot}_{\text{lin}_{\text{RPV}}} + M\text{dot}_{\text{p2}}
END IF
\end{verbatim}

\begin{verbatim}
IF (dir_{p2} .gt. 0) THEN
    Mdotlout_{RPV} = Mdotlout_{RPV} + M\text{dot}_{\text{p2}}
END IF
\end{verbatim}

\begin{verbatim}
C WW2
Mdot_{gout_{WW2}} = 0.0D+0
Mdot_{vin_{WW2}} = 0.0D+0
Mdot_{ain_{WW2}} = 0.0D+0
h_{vin_{WW2}} = 0.0D+0
h_{ain_{WW2}} = 0.0D+0
Mdotout_{WW2} = 0.0D+0
hlin_{WW2} = 0.0D+0
\end{verbatim}

\begin{verbatim}
IF (dir_{p2} .gt. 0 .AND. Mdot_{p2} .GT. 1.0D-12) THEN
    hlin_{WW2} = \left( h_{\text{p2}} \cdot M\text{dot}_{\text{p2}} + h\text{lin}_{\text{WW2}} \cdot M\text{dot}_{\text{lin}_{\text{WW2}}} \right) / (M\text{dot}_{\text{lin}_{\text{WW2}}} + M\text{dot}_{\text{p2}}) \\
    \text{Mdotlin}_{\text{WW2}} = M\text{dot}_{\text{lin}_{\text{WW2}}} + M\text{dot}_{\text{p2}}
END IF
\end{verbatim}

\begin{verbatim}
IF (dir_{p2} .lt. 0) THEN
    Mdotlout_{WW2} = Mdotlout_{WW2} + M\text{dot}_{\text{p2}}
END IF
\end{verbatim}

\begin{verbatim}
C WW1
Mdot_{gout_{WW1}} = 0.0D+0
Mdot_{vin_{WW1}} = 0.0D+0
Mdot_{ain_{WW1}} = 0.0D+0
h_{vin_{WW1}} = 0.0D+0
h_{ain_{WW1}} = 0.0D+0
Mdotout_{WW1} = 0.0D+0
hlin_{WW1} = 0.0D+0
\end{verbatim}

\begin{verbatim}
IF (dir_{p1} .lt. 0 .AND. Mdot_{p1} .GT. 1.0D-12) THEN
    hlin_{WW1} = \left( h_{\text{p1}} \cdot M\text{dot}_{\text{p1}} + h\text{lin}_{\text{WW1}} \cdot M\text{dot}_{\text{lin}_{\text{WW1}}} \right) / (M\text{dot}_{\text{lin}_{\text{WW1}}} + M\text{dot}_{\text{p1}}) \\
    \text{Mdotlin}_{\text{WW1}} = M\text{dot}_{\text{lin}_{\text{WW1}}} + M\text{dot}_{\text{p1}}
END IF
\end{verbatim}

\begin{verbatim}
IF (dir_{p1} .gt. 0) THEN
    Mdotlout_{WW1} = Mdotlout_{WW1} + M\text{dot}_{\text{p1}}
END IF
\end{verbatim}

\begin{verbatim}
C Calculate heats into vessels. 
\end{verbatim}

\begin{verbatim}
C RPV 
\end{verbatim}
Qdotin_l_RPV = 0.OD+0
Qdotin_3_RPV = 0.OD+0
Qdotwall_l_RPV = 0.OD+0
Qdotwall_3_RPV = 0.OD+0

C WW2
Qdotin_l_WW2 = 0.OD+0
Qdotin_3_WW2 = 0.OD+0
Qdotwall_l_WW2 = 0.OD+0
Qdotwall_3_WW2 = 0.OD+0

C WW1
Qdotin_l_WW1 = 0.OD+0
Qdotin_3_WW1 = 0.OD+0
Qdotwall_l_WW1 = 0.OD+0
Qdotwall_3_WW1 = 0.OD+0

C Set up state vectors.

C RPV
state_RPV(1) = roots(1)
state_RPV(2) = roots(2)

C WW2
state_WW2(1) = roots(3)
state_WW2(2) = roots(4)

C WW1
state_WW1(1) = roots(5)
state_WW1(2) = roots(6)

C Finally calculate derivatives.

C RPV
CALL Vessel_deriv_evt(Y(base_RPV + 0), Y(base_RPV + 1),
# Y(base_RPV + 2), Y(base_RPV + 3), Y(base_RPV + 4),
# Y(base_RPV + 5), Y(base_RPV + 6), Y(base_RPV + 7),
# Qdotin_l_RPV, Qdotin_3_RPV, Qdotwall_l_RPV,
# Qdotwall_3_RPV, Mdotlout_RPV, Mdotlin_RPV,
# hlin_RPV, Mdot_vin_RPV, Mdot_ain_RPV,
# h_vin_RPV, h_ain_RPV, Mdot_gout_RPV,
# A_lg_RPV, V_tot_RPV, state_RPV,
# YDOT(base_RPV + 0), YDOT(base_RPV + 1), YDOT(base_RPV + 2),
# YDOT(base_RPV + 3), YDOT(base_RPV + 4), YDOT(base_RPV + 5),
# YDOT(base_RPV + 6), YDOT(base_RPV + 7))

C WW2
CALL Vessel_deriv_evt(Y(base_WW2 + 0), Y(base_WW2 + 1),
# Y(base_WW2 + 2), Y(base_WW2 + 3), Y(base_WW2 + 4),
# Y(base_WW2 + 5), Y(base_WW2 + 6), Y(base_WW2 + 7),
# Qdotin_l_WW2, Qdotin_3_WW2, Qdotwall_l_WW2,
# Qdotwall_3_WW2, Mdotlout_WW2, Mdotlin_WW2,
# hlin_WW2, Mdot_vin_WW2, Mdot_ain_WW2,
# h_vin_WW2, h_ain_WW2, Mdot_gout_WW2,
# A_lg_WW2, V_tot_WW2, state_WW2,
# Appendix D: An Example MSDL Program

```fortran
# YDOT(base WW2 + 0), YDOT(base WW2 + 1), YDOT(base WW2 + 2),
# YDOT(base WW2 + 3), YDOT(base WW2 + 4), YDOT(base WW2 + 5),
# YDOT(base WW2 + 6), YDOT(base WW2 + 7))

C WW1
CALL Vessel_deriv_evt(Y(base WW1 + 0), Y(base WW1 + 1),
# Y(base WW1 + 2), Y(base WW1 + 3), Y(base WW1 + 4),
# Y(base WW1 + 5), Y(base WW1 + 6), Y(base WW1 + 7),
# Qdotin_1_WW1, Qdotin_3_WW1, Qdotwall_1_WW1,
# Qdotwall_3_WW1, Mdotlout_WW1, Mdotlin_WW1,
# hlin_WW1, Mdot_vin_WW1, Mdot_ain_WW1,
# h_vin_WW1, h_ain_WW1, Mdot_gout_WW1,
# A_lg_WW1, V_tot_WW1, state_WW1,
# YDOT(base WW1 + 0), YDOT(base WW1 + 1), YDOT(base WW1 + 2),
# YDOT(base WW1 + 3), YDOT(base WW1 + 4), YDOT(base WW1 + 5),
# YDOT(base WW1 + 6), YDOT(base WW1 + 7))

C p1
CALL waterpipe_diff(Y(base pl + 0), p_in_pl, h_in_pl,
# p_out_pl, h_out_pl, L_pl, Dn_pl, k_pl,
# YDOT(base pl + 0))

C p2
CALL waterpipe_diff(Y(base p2 + 0), p_in_p2, h_in_p2,
# p_out_p2, h_out_p2, L_p2, Dn_p2, k_p2,
# YDOT(base p2 + 0))

C------------------------ End of subroutine F1. ------------------------
RETURN
END

SUBROUTINE GRl (NEQ, T, Y, NG, GROOT)
implicit none
INTEGER NEQ, NG
DOUBLE PRECISION T, Y, GROOT
DIMENSION Y(*), GROOT(*)
INTEGER base_RPV, base_WW2, base_wwi
INTEGER base_pl, base_p2
PARAMETER (base_RPV = 1, base_WW2 = 9, base_WW1 = 17)
PARAMETER (base_pl = 25, base_p2 = 26)
DOUBLE PRECISION g, tol_air_min
PARAMETER (g = 9.81, tol_air_min = 1.0D-12)
DOUBLE PRECISION pres
DOUBLE PRECISION hfp_si

C RPV
pres = Y(base_RPV + 2) + Y(base_RPV + 3)
GROOT(1) = Y(base_RPV + 7) - hfp_si(pres)
GROOT(2) = Y(base_RPV + 1) - tol_air_min

C WW2
pres = Y(base WW2 + 2) + Y(base WW2 + 3)
GROOT(3) = Y(base WW2 + 7) - hfp_si(pres)
GROOT(4) = Y(base WW2 + 1) - tol_air_min
```
D.2. Generated Fortran Code

```fortran
C WW1
pres = Y(base_WW1 + 2) + Y(base_WW1 + 3)
GROOT(5) = Y(base_WW1 + 7) - hfp_si(pres)
GROOT(6) = Y(base_WW1 + 1) - tol_air_min
C p1
GROOT(7) = Y(base_p1 + 0) - 1.0D-9
C p2
GROOT(8) = Y(base_p2 + 0) - 1.0D-9
C------------------------ End of subroutine GR1. ------------------------
RETURN
END

SUBROUTINE JAC (NEQ, T, Y, ML, MU, J, NROWPD)
implicit none
INTEGER NEQ, ML, MU, NROWPD
DOUBLE PRECISION T, Y, J
DIMENSION Y(*), J(*)
SUBROUTINE JAC.
RETURN
END

SUBROUTINE INIT (Y)
implicit none
DOUBLE PRECISION Y
DIMENSION Y(*)
include 'lsodar.com'
INTEGER base_RPV, base_WW2, base_WW1
INTEGER base_p1, base_p2
PARAMETER (base_RPV = 1, base_WW2 = 9, base_WW1 = 17)
PARAMETER (base_p1 = 25, base_p2 = 26)
DOUBLE PRECISION tsatp_si, vptl_si, hptl_si, vptd_si
C RPV
Y(base_RPV + 3) = 1000
Y(base_RPV + 2) = 350000 - Y(base_RPV + 3)
Y(base_RPV + 4) = tsatp_si(Y(base_RPV + 2))
Y(base_RPV + 6) = 1.22718 * 6.13
Y(base_RPV + 5) = Y(base_RPV + 6) / vptl_si(Y(base_RPV + 2) +
# Y(base_RPV + 3), 104.8 + 2.7315D+2)
Y(base_RPV + 7) = hptl_si(Y(base_RPV + 2) +
# Y(base_RPV + 3), 104.8 + 2.7315D+2)
Y(base_RPV + 0) = (22.8 - Y(base_RPV + 6)) /
# vptd_si(Y(base_RPV + 2), Y(base_RPV + 4))
Y(base_RPV + 1) = ((22.8 - Y(base_RPV + 6)) * 0.028966 *
# Y(base_RPV + 3)) / (8.31441 * Y(base_RPV + 4))

print *, '--- RPV initial values ---'
print *, 'Mass of vapor in RPV is ', Y(base_RPV + 0), ' kg'
print *, 'Mass of air in RPV is ', Y(base_RPV + 1), ' kg'
print *, 'Pressure of vapor in RPV is ', Y(base_RPV + 2), ' Pa'
print *, 'Pressure of air in RPV is ', Y(base_RPV + 3), ' Pa'
print *, 'Temperature of gas in RPV is ', Y(base_RPV + 4), ' K'
```
print *, 'Mass of liquid in RPV is ', Y(base_RPV + 5), ' kg'
print *, 'Volume of liquid in RPV is ', Y(base_RPV + 6), ' m^3'
print *, 'Enthalpy of liquid in RPV is ', Y(base_RPV + 7), ' J/kg'
print *

C WW2
Y(base_WW2 + 3) = 200000
Y(base_WW2 + 2) = 350002 - Y(base_WW2 + 3)
Y(base_WW2 + 4) = tstatp_si(Y(base_WW2 + 2))
Y(base_WW2 + 6) = 7.16197 * 4
Y(base_WW2 + 5) = Y(base_WW2 + 6) / vptl_si(Y(base_WW2 + 2) +
# Y(base_WW2 + 3), 30.8 + 2.7315D+2)
Y(base_WW2 + 7) = hptl_si(Y(base_WW2 + 2) +
# Y(base_WW2 + 3), 30.8 + 2.7315D+2)
Y(base_WW2 + 0) = (117 - Y(base_WW2 + 6)) /
# vptd_si(Y(base_WW2 + 2), Y(base_WW2 + 4))
Y(base_WW2 + 1) = (((117 - Y(base_WW2 + 6)) * 0.028966 *
# Y(base_WW2 + 3)) / (8.31441 * Y(base_WW2 + 4))

print *, '--- WW2 initial values ---'
print *, 'Mass of vapor in WW2 is ', Y(base_WW2 + 0), ' kg'
print *, 'Mass of air in WW2 is ', Y(base_WW2 + 1), ' kg'
print *, 'Pressure of vapor in WW2 is ', Y(base_WW2 + 2), ' Pa'
print *, 'Pressure of air in WW2 is ', Y(base_WW2 + 3), ' Pa'
print *, 'Temperature of gas in WW2 is ', Y(base_WW2 + 4), ' K'
print *, 'Mass of liquid in WW2 is ', Y(base_WW2 + 5), ' kg'
print *, 'Volume of liquid in WW2 is ', Y(base_WW2 + 6), ' m^3'
print *, 'Enthalpy of liquid in WW2 is ', Y(base_WW2 + 7), ' J/kg'
print *

C WW1
Y(base_WW1 + 3) = 100000
Y(base_WW1 + 2) = 350001 - Y(base_WW1 + 3)
Y(base_WW1 + 4) = tstatp_si(Y(base_WW1 + 2))
Y(base_WW1 + 6) = 7.16197 * 4
Y(base_WW1 + 5) = Y(base_WW1 + 6) / vptl_si(Y(base_WW1 + 2) +
# Y(base_WW1 + 3), 20.5 + 2.7315D+2)
Y(base_WW1 + 7) = hptl_si(Y(base_WW1 + 2) +
# Y(base_WW1 + 3), 20.5 + 2.7315D+2)
Y(base_WW1 + 0) = (117 - Y(base_WW1 + 6)) /
# vptd_si(Y(base_WW1 + 2), Y(base_WW1 + 4))
Y(base_WW1 + 1) = (((117 - Y(base_WW1 + 6)) * 0.028966 *
# Y(base_WW1 + 3)) / (8.31441 * Y(base_WW1 + 4))

print *, '--- WW1 initial values ---'
print *, 'Mass of vapor in WW1 is ', Y(base_WW1 + 0), ' kg'
print *, 'Mass of air in WW1 is ', Y(base_WW1 + 1), ' kg'
print *, 'Pressure of vapor in WW1 is ', Y(base_WW1 + 2), ' Pa'
print *, 'Pressure of air in WW1 is ', Y(base_WW1 + 3), ' Pa'
print *, 'Temperature of gas in WW1 is ', Y(base_WW1 + 4), ' K'
print *, 'Mass of liquid in WW1 is ', Y(base_WW1 + 5), ' kg'
print *, 'Volume of liquid in WW1 is ', Y(base_WW1 + 6), ' m^3'
print *, 'Enthalpy of liquid in WW1 is ', Y(base_WW1 + 7), ' J/kg'
print *

C pl
Y(base_pl + 0) = 1.0D-50
C p2
Y(base_p2 + 0) = 1.0D-50
C RPV
roots(1) = 1  
roots(2) = 1  
C WW2  
roots(3) = 1  
roots(4) = 1  
C WW1  
roots(5) = 1  
roots(6) = 1  
C------------------------ End of subroutine INIT. ------------------------
RETURN
END

SUBROUTINE PRINTOUT (t, Y)
implicit none
DOUBLE PRECISION t, Y
DIMENSION Y(*)

INTEGER base_RPV, base_WW2, base_WW1
INTEGER base_p1, base_p2

PARAMETER (base_RPV = 1, base_WW2 = 9, base_WW1 = 17)
PARAMETER (base_p1 = 25, base_p2 = 26)

C Print T and Y
C RPV
write (1 , 100), t, Y(base_RPV + 0)
write (2 , 100), t, Y(base_RPV + 1)
write (3 , 100), t, Y(base_RPV + 2)
write (4 , 100), t, Y(base_RPV + 3)
write (7 , 100), t, Y(base_RPV + 4)
write (8 , 100), t, Y(base_RPV + 5)
write (9 , 100), t, Y(base_RPV + 6)
write (10 , 100), t, Y(base_RPV + 7)
write (11 , 100), t, Y(base_RPV + 2) + Y(base_RPV + 3)

C WW2
write (12 , 100), t, Y(base_WW2 + 0)
write (13 , 100), t, Y(base_WW2 + 1)
write (14 , 100), t, Y(base_WW2 + 2)
write (15 , 100), t, Y(base_WW2 + 3)
write (16 , 100), t, Y(base_WW2 + 4)
write (17 , 100), t, Y(base_WW2 + 5)
write (18 , 100), t, Y(base_WW2 + 6)
write (19 , 100), t, Y(base_WW2 + 7)
write (20 , 100), t, Y(base_WW2 + 2) + Y(base_WW2 + 3)

C WW1
write (21 , 100), t, Y(base_WW1 + 0)
write (22 , 100), t, Y(base_WW1 + 1)
write (23 , 100), t, Y(base_WW1 + 2)
write (24 , 100), t, Y(base_WW1 + 3)
write (25 , 100), t, Y(base_WW1 + 4)
write (26 , 100), t, Y(base_WW1 + 5)
write (27 , 100), t, Y(base_WW1 + 6)
write (28 , 100), t, Y(base_WW1 + 7)
write (29 , 100), t, Y(base_WW1 + 2) + Y(base_WW1 + 3)

C p1
write (30 , 100), t, Y(base_p1 + 0)
C p2
   write (31, 100), t, Y(base_p2 + 0)
C Total Mass of system
   write (32, 100), t, 0.0D+0
     #   + Y(base_RPV + 0) + Y(base_RPV + 1) + Y(base_RPV + 5)
     #   + Y(base_WW2 + 0) + Y(base_WW2 + 1) + Y(base_WW2 + 5)
     #   + Y(base_WW1 + 0) + Y(base_WW1 + 1) + Y(base_WW1 + 5)
100  format (F22.11, ' ', F29.11)

C-------------------- End of subroutine PRINTOUT. ------------------
RETURN
END
E.1. Software for Solving Systems of ODEs

For solving systems of ODEs the code collection *ODEPACK* by Alan C. Hindmarsh is used.

The description in this chapter is intended to give a short summary of the capabilities of the software and is extracted from the original literature [Hin83]. The given examples are modified in order to give a better description to discontinuities and the method for changing models during the integration.

E.1.1. Introduction

*ODEPACK* is a collection of Fortran solvers for the initial value problem for ordinary differential equation systems. It consists of nine solvers, namely a basic solver called LSODE and eight variants of it — LSODES, LSODA, LSODAR, LSODPK, LSODKR, LSODI, LSOIBT,
and LSODIS. The collection is suitable for both stiff and non-stiff systems. It includes solvers for systems given in explicit form,

$$\frac{dy}{dt} = f(t, y)$$  \hspace{1cm} (E.1)

and also solvers for systems given in linearly implicit form,

$$A(t, y)\frac{dy}{dt} = g(t, y)$$  \hspace{1cm} (E.2)

Two of the solvers use general sparse matrix techniques for the linear systems that arise and which are also particularly suitable when solving parabolic hand hyperbolic PDEs with banded matrix structures. Two others use iterative (preconditioned Krylov [HN88]) methods instead of direct methods for these linear systems. The most recent addition is LSODIS, which solves implicit problems with general sparse treatment of all matrices involved.

The ODEPACK solvers are written in standard Fortran 77, with a few exceptions, and with minimal machine dependencies. There are separate double and single precision versions of ODEPACK. The actual solver names are those given above with a prefix of D- or S- for the double or single precision version, respectively, i.e. DL-SODE/SLSODE, etc. Each solver consists of a main driver subroutine having the same name as the solver and some number of subordinate routines. For each solver, there is also a demonstration program, which solves one or two simple problems and provides the resulting output.

Recently, the ODEPACK solvers were upgraded to improve their portability in numerous ways. Among the improvements are (a) renaming of routines and Common blocks to distinguish double and single precision versions, (b) use of generic intrinsic function names, (c) elimination of the Block Data subprogram, (d) use of a portable routine to set the unit roundoff, and (e) passing of quoted strings to the error message handler. In addition, the prologue and internal comments were reformatted, and use mixed upper/lower case. Numerous minor corrections and improvements were also made.

The above upgrade operations were applied to the LSODE solver earlier than they were implemented in the rest of codes in ODEPACK, and the two upgrades were done somewhat independently. As a result, some differences will be apparent in the source files of LSODE and the other solvers — primarily in the formatting of the comment line prologue of the main driver routine. In Subroutines
DLSODE/SLSODE and their subordinate routines, the prologue was written in “SLATEC format”, while for the other solvers a more relaxed style was used. The differences are entirely cosmetic, however, and do not affect performance.

Documentation on the usage of each solver is provided in the initial block of comment lines in the source file, which (in most cases) includes a simple example. A demonstration program (in separate double/single precision versions) is also available.

What follows is a summary of the capabilities of ODEPACK, comments about usage documentation, and notes about installing the collection. For additional documentation on ODEPACK, see also the papers [Hin83], [RH93] (for LSODE), and [BH89] (for LSODPK and LSODKR), and in the references cited there. (However, the document [RH93] does not reflect the upgrade operations described above.)

E.1.2. Summary of the explicit ODEPACK Solvers

For each of the following solvers, it is assumed that the ODEs are given explicitly, so that the system can be written in the form

$$\frac{dy}{dt} = f(t, y) \quad (E.3)$$

where $y$ is the vector of dependent variables, and $t$ is the independent variable.

The ODEPACK source files are complete except for needed routines from the LINPACK\(^1\) and BLAS\(^2\) collections and machine constant routines, described later in this chapter on page 231 and 233, respectively.

**LSODE**

(Livermore Solver for Ordinary Differential Equations) is the basic solver of the collection. It solves stiff and non-stiff systems of the form $\frac{dy}{dt} = f$. In the stiff case, it treats the Jacobian matrix $\frac{df}{dy}$ as either a full or a banded matrix, and as either user-supplied or internally approximated by difference quotients. It uses the Adams-Moulton method in the non-stiff case, and a Gear type backward differentiation formula

---

\(^1\)Code collection of direct linear routines for solving systems of linear algebraic equations

\(^2\)Basic linear algebraic subsystem.
(BDF) in the stiff case. Both methods are of variable order and provide automatic step-size changes. In either case, both algorithms necessitate the solution of a typically non-linear algebraic system of equations. For this, in the non-stiff case a predictor-corrector method gets applied consisting of an Adams-Bashforth step for providing a predictor for a subsequent Adams-Moulton corrector step. In the stiff case, the Newton-Raphson method using the Jacobian of the system has to be applied so as to find the solution even for large integration steps and large differences in the magnitude of the eigenvalues.

**LSODES**

Written jointly with A. H. Sherman, LSODES solves systems \( \frac{dy}{dt} = f \) and in the stiff case treats the Jacobian matrix in general sparse form. It determines the sparsity structure on its own (or optionally accepts this information from the user), and uses parts of the Yale Sparse Matrix Package (YSMP)\[EGSS82\] to solve the linear systems that arise.

**LSODA**

Written jointly with L. R. Petzold, LSODA solves systems \( \frac{dy}{dt} = f \) with a full or banded Jacobian when the problem is stiff, but it automatically selects between the non-stiff Adams-Moulton Method and the stiff Gear type BDF methods. It uses the non-stiff method initially, and dynamically monitors the amount of work and the size for performing an integration step in order to decide which method to use.

**LSODAR**

Also written jointly with L. R. Petzold, LSODAR is a variant of LSODA with a root-finding capability added. Thus it solves problems \( \frac{dy}{dt} = f \) with full or banded Jacobian and automatic method selection, and at the same time, it finds the roots of any of a set of given functions of the form \( g(t, y) \). This is often useful for finding points where components of the vector \( f \) change discontinuously.

**LSODPK**

Written jointly with Peter N. Brown, LSODPK is a variant of LSODE in which the direct solvers for the linear systems have been replaced by a selection of four preconditioned Krylov (iterative) solvers. The
user must supply a pair of routines to evaluate, preprocess, and solve
the (left and/or right) preconditioner matrices. LSODPK also includes
an option for a user-supplied linear system solver to be used without
Krylov iteration.

**LSODKR**

LSODKR is a variant of LSODPK with the addition of the same
root-finding capability as in LSODAR, and also of automatic switch¬
ing between functional and Newton iteration. The nonlinear iteration
method-switching differs from the method-switching in LSODA and
LSODAR, but provides similar savings by using the cheaper method
in the non-stiff regions of the problem. LSODKR also improves on the
Krylov methods in LSODPK by offering the option to save and reuse
the approximate Jacobian data underlying the preconditioner.

**E.2. Description and Use of LSODAR**

In this work, the 8 August 2001 version in double precision of DLSO-
DAR (Livermore Solver for Ordinary Differential Equations, with Au-
tomatic method switching for stiff and non-stiff problems, and with
root-finding) is used.

It has not been checked if some of the other codes mentioned
would result in higher accuracy or an increased computational effi¬
ciency when applied to the sample problems treated in this thesis.

DLSODAR solves the initial value problem for stiff or non-stiff sys¬
tems of first order ODEs,

\[
\frac{dy}{dt} = f(t, y)
\]  

(E.4)

or, in component form,

\[
\frac{dy_i}{dt} = f_i(t, y_1, y_2, \ldots, y_N), \quad i = 1, \ldots, N
\]  

(E.5)

At the same time, it locates the roots of any of a set of functions

\[
g_j(t, y_1, y_2, \ldots, y_N) = 0 \quad j = 1, \ldots, ng
\]  

(E.6)

where ng denotes the number of root functions.

DLSODAR differs from DLSODE in two ways:
1. It switches automatically between the methods provided for the stiff and for the non-stiff case. This means that the user does not have to determine whether the problem is stiff or not (a property that can change during the integration of a problem). The solver will automatically choose the appropriate method. It always starts with the non-stiff method.

2. It finds the root of at least one of a set of constraint functions $g_j$ of the independent and dependent variables. It finds only those roots for which some $g_j$, as a function of $t$, changes sign in the interval of integration. It then returns the solution at the positive root at the location closest to the beginning of the step, if that occurs sooner than the specified stop condition, and otherwise returns the solution according the specified stop condition.

DLSODAR has been written by Alan C. Hindmarsh and Linda R. Petzold. Details can be found in [Hin83] [Pet83] and [HS80].

**E.2.1. Summary of Usage**

Communication between the user and the DLSODAR package, for normal situations, is summarized here. This summary describes only a subset of the full set of options available. See the full description for details, including alternative treatment of the Jacobian matrix, optional inputs and outputs, nonstandard options, and instructions for special situations in appendix E.2.2 on page 213. See also the example problems (with program and output) on page 207.

1. First provide a subroutine of the form:

   **Listing E.1: Main function for DLSODAR**
   
   ```fortran
   SUBROUTINE F (NEQ, T, Y, YDOT)
   DOUBLE PRECISION T, Y(*), YDOT(*)
   
   which supplies the vector YDOT (*) by evaluating $f_i$ and assigning the result to YDOT (I).
   
   2. Provide a subroutine of the form:

   **Listing E.2: Root functions for DLSODAR**
   
   ```fortran
   SUBROUTINE G (NEQ, T, Y, NG, GOUT)
   DOUBLE PRECISION T, Y(*), GOUT(NG)
   ```
which supplies the vector function $g$ by loading the elements $g_j$ of vector $\text{GOUT}_j$, i.e. the $j$-th constraint function whose zero crossing have to be found.

3. Write a main program which calls Subroutine DLSODAR once for each point at which answers are desired. This should also provide for possible use of logical unit 6 for output of error messages by DLSODAR. On the first call to DLSODAR, supply arguments as given in table E.1

4. The output from the first call (or any call) is given in table E.2.

5. To continue the integration after a successful return, proceed as follows:
   - If $\text{ISTATE} = 2$ on return, reset $\text{TOUT}$ and call DLSODAR again.
   - If $\text{ISTATE} = 3$ on return, reset $\text{ISTATE}$ to 2, call DLSODAR again.

In either case, no other parameters need be reset.

6. Note: If and when DLSODAR regards the problem as stiff, and switches methods accordingly, it must make use of the $(\text{NEQ} \times \text{NEQ})$-dimensional Jacobian matrix, $J = \frac{df}{dy}$. For the sake of simplicity, the inputs to DLSODAR recommended above cause DLSODAR to treat $J$ as a full matrix, and to approximate it internally by difference quotients. Alternatively, $J$ can be treated as a banded matrix (with great potential reduction in the size of the RWORK array). Also, in either the full or banded case, the user can supply $J$ in closed form, with a routine whose name is passed as the JAC argument. These alternatives are described in the paragraphs on RWORK, JAC, and JT in the full description of the call sequence below.

E.2.2. Examples for using DLSODAR

The following is a simple sample problem, with the coding needed for its solution by DLSODAR. The problem is from chemical kinetics, and consists of the first three rate equations given below. The fourth equation, however, has no physical or chemical relevance and is only added
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F</strong></td>
<td>name of subroutine for right-hand side vector f. This name must be declared as external in the calling program.</td>
</tr>
<tr>
<td><strong>NEQ</strong></td>
<td>number of first order ODEs.</td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td>array of initial values, of length NEQ.</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>the initial value of the independent variable.</td>
</tr>
<tr>
<td><strong>TOUT</strong></td>
<td>first point where output is desired (.ne. T).</td>
</tr>
<tr>
<td><strong>ITOL</strong></td>
<td>1 or 2 according as ATOL (below) is a scalar or array.</td>
</tr>
<tr>
<td><strong>RTOL</strong></td>
<td>relative tolerance parameter (scalar).</td>
</tr>
<tr>
<td><strong>ATOL</strong></td>
<td>absolute tolerance parameter (scalar or array). the estimated local error in y(i) will be controlled so as to be less than EWT(i) = RTOL<em>ABS(Y(i)) + ATOL if ITOL = 1, or EWT(i) = RTOL</em>ABS(Y(i)) + ATOL(i) if ITOL = 2. Thus the local error test passes if, in each component, either the absolute error is less than ATOL (or ATOL(i)), or the relative error is less than RTOL. Use RTOL = 0.0 for pure absolute error control, and use ATOL = 0.0 (or ATOL(i) = 0.0) for pure relative error control. Caution: actual (global) errors may exceed these local tolerances, so choose them conservatively.</td>
</tr>
<tr>
<td><strong>ITASK</strong></td>
<td>1 for normal computation of output values of y at t = TOUT.</td>
</tr>
<tr>
<td><strong>ISTATE</strong></td>
<td>integer flag (input and output). Set ISTATE = 1.</td>
</tr>
<tr>
<td><strong>IOPT</strong></td>
<td>0 to indicate no optional inputs used.</td>
</tr>
<tr>
<td><strong>RWORK</strong></td>
<td>real work array of length at least: 22 + NEQ * MAX(16, NEQ + 9) + 3*NG. See also Paragraph F below.</td>
</tr>
<tr>
<td><strong>LRW</strong></td>
<td>declared length of RWORK (in user's dimension).</td>
</tr>
<tr>
<td><strong>IWORK</strong></td>
<td>integer work array of length at least 20 + NEQ.</td>
</tr>
<tr>
<td><strong>LIW</strong></td>
<td>declared length of IWORK (in user's dimension).</td>
</tr>
<tr>
<td><strong>JAC</strong></td>
<td>name of subroutine for Jacobian matrix. Use a dummy name. See also Paragraph F below.</td>
</tr>
<tr>
<td><strong>JT</strong></td>
<td>Jacobian type indicator. Set JT = 2. See also Paragraph F below.</td>
</tr>
<tr>
<td><strong>G</strong></td>
<td>name of subroutine for constraint functions, whose roots are desired during the integration. This name must be declared External in calling program.</td>
</tr>
<tr>
<td><strong>NG</strong></td>
<td>number of constraint functions g(j). If there are none, set NG = 0, and pass a dummy name for G.</td>
</tr>
<tr>
<td><strong>JROOT</strong></td>
<td>integer array of length NG for output of root information. See next paragraph.</td>
</tr>
</tbody>
</table>

**Table E.1: Description of parameters used in the DLSODAR package**
E.2. Description and Use of LSODAR

**Table E.2: Output parameters in the DLSODAR package**

<table>
<thead>
<tr>
<th>Y</th>
<th>array of computed values of state vector ( y(t) ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>corresponding value of independent variable. This is TOUT if ISTATE = 2, or the root location if ISTATE = 3, or the farthest point reached if DLSODAR was unsuccessful.</td>
</tr>
<tr>
<td>ISTATE</td>
<td>2 or 3 if DLSODAR was successful, negative otherwise.</td>
</tr>
<tr>
<td></td>
<td>2 means no root was found, and TOUT was reached as desired.</td>
</tr>
<tr>
<td></td>
<td>3 means a root was found prior to reaching TOUT.</td>
</tr>
<tr>
<td></td>
<td>-1 means excess work done on this call (perhaps wrong JT).</td>
</tr>
<tr>
<td></td>
<td>-2 means excess accuracy requested (tolerances too small).</td>
</tr>
<tr>
<td></td>
<td>-3 means illegal input detected (see printed message).</td>
</tr>
<tr>
<td></td>
<td>-4 means repeated error test failures (check all inputs).</td>
</tr>
<tr>
<td></td>
<td>-5 means repeated convergence failures (perhaps bad Jacobian supplied or wrong choice of JT or tolerances).</td>
</tr>
<tr>
<td></td>
<td>-6 means error weight became zero during problem. (Solution component i vanished, and ATOL or ATOL(i) = 0.)</td>
</tr>
<tr>
<td></td>
<td>-7 means workspace insufficient to finish (see messages).</td>
</tr>
<tr>
<td>JROOT</td>
<td>array showing roots found if ISTATE = 3 on return. JROOT(i) = 1 if g(i) has a root at t, or 0 otherwise.</td>
</tr>
</tbody>
</table>

for showing the impact of its implementation on the programming of the problem. This latter ODE requires a special programming of the main program and of the subroutines FEX and GEX for doing the function evaluations and the evaluation of the constraint functions, respectively. Note that in case of an analytic computation of the Jacobian the code for this subroutine would have to be adapted accordingly, featuring all the necessary logic to distinguish between the various cases for calculating the partials of \( y_4 \) with respect to the other states.
\[
\frac{dy_1}{dt} = -0.04y_1 + 10^4y_2y_3
\]
\[
\frac{dy_2}{dt} = 0.04y_1 - 10^4y_2y_3 - 3 \times 10^7y_2^2
\]
\[
\frac{dy_3}{dt} = 3 \times 10^7y_2^2
\]
\[
\frac{dy_4}{dt} = \begin{cases} 
    \frac{dy_3}{dt} & \text{if } y_1 > 10^{-3} \land y_1 > y_3 \land y_3 < 100y_2 \\
    0 & \text{if } y_1 > 10^{-3} \land y_1 > y_3 \land y_3 \geq 100y_2 \\
    30y_4 \frac{dy_1}{dt} & \text{if } y_1 > 10^{-3} \land y_1 \leq y_3 \land y_3 \geq 100y_2 \\
    0 & \text{if } y_1 \leq 10^{-3} \land y_1 \leq y_3 \land y_3 \geq 100y_2
\end{cases}
\]  

(E.7)

with the initial conditions
\[
y_1(0) = 1 \\
y_2(0) = 0 \\
y_3(0) = 0 \\
y_4(0) = 0.001
\]  

(E.8)

Listing E.3: Example Program for using DLSODAR: Chem.f

```
EXTERNAL FEX, GEX
DOUBLE PRECISION ATOL, RTOL, RWORK, T, TOUT, Y
DIMENSION Y(4), ATOL(4), RWORK(93), IWORK(24), JROOT(3)
INTEGER I, J, FLAG(3)
COMMON /FLAGS/ FLAG
NEQ = 4
Y(1) = 1.
Y(2) = 0.
Y(3) = 0.
Y(4) = 0.001
FLAG(1) = 1
FLAG(2) = 1
FLAG(3) = 1
T = 0.
TOUT = .01
ITOL = 2
RTOL = 1.D-4
ATOL(1) = 1.D-6
ATOL(2) = 1.D-10
ATOL(3) = 1.D-6
ATOL(4) = 1.D-6
ITASK = 1
ISTATE = 1
IOPT = 0
```
E.2. Description and Use of LSODAR

LRW = 93
LIW = 24
JT = 2
NG = 3

DO 40 IOUT = 1,35
10 CALL DLSODAR(FEX, NEQ, Y, T, TOUT, ITOL, RTOL, ATOL, ITASK, ISTATE,
WRITE(6,20)T, Y(1), Y(2), Y(3), Y(4)
C WRITE(6,101)T, FLAG(1), FLAG(2), FLAG(3)
20 FORMAT(' At t = ', D12.4, ' Flag values = ', 3I5)
IF (ISTATE .LT. 0) GO TO 80
IF (ISTATE .EQ. 2) GO TO 40
WRITE(6,30)JROOT(1), JROOT(2), JROOT(3)
30 FORMAT(5X, 'The above line is a root. JROOT = ', 3I5)
C A Root got detected requiring a re-initialization. Set ISTATE=1.
ISTATE = 1
DO 35 J = 1, NG
C The nested braces where introduced on a explicit comment by J. Halin
IF (JROOT(J) .EQ. 1) FLAG(J) = FLAG(J) * ((-1))
35 CONTINUE
GO TO 10
40 TOUT = TOUT*2.
WRITE(6,60)IWORK(11), IWORK(12), IWORK(13), IWORK(10),
1 IWORK(19), RWORK(15)
60 FORMAT(' No. steps = ', I4, ' No. f-s = ', I4, ' No. J-s = ', I4,
1 No. g-s = ', I4/
2 ' Method last used = ', I2,' Last switch was at t = ', D12.4)
STOP
80 WRITE(6,90)ISTATE
90 FORMAT(///' Error halt.. ISTATE = ', I3)
STOP
END
SUBROUTINE FEX (NEQ, T, Y, YDOT)
DOUBLE PRECISION T, Y, YDOT
DIMENSION Y(4), YDOT(4)
INTEGER FLAG(3)
COMMON /FLAGS/ FLAG
YDOT(1) = -.04*Y(1) + 1.04*Y(2)*Y(3)
YDOT(2) = -YDOT(1) - YDOT(3)
YDOT(3) = 3.07*Y(2)**2
YDOT(4) = 0.
IF ((FLAG(1).EQ.1).AND.(FLAG(2).EQ.1).AND.(FLAG(3).EQ.1)) THEN
YDOT(4) = ydot(3)
ELSE IF
1 ((FLAG(1).EQ.1).AND.(FLAG(2).EQ.1).AND.(FLAG(3).EQ.-1)) THEN
YDOT(4) = 0.
ELSE IF
1 ((FLAG(1).EQ.1).AND.(FLAG(2).EQ.-1).AND.(FLAG(3).EQ.-1)) THEN
YDOT(4) = YDOT(1) * 30.*Y(4)
ELSE IF
1 ((FLAG(1).EQ.-1).AND.(FLAG(2).EQ.-1).AND.(FLAG(3).EQ.-1)) THEN
YDOT(4) = 0.
ELSE
YDOT(4) = 0.
ENDIF
C WRITE(6,101)FLAG(1), FLAG(2), FLAG(3)
101 FORMAT( 'Flag values = ', 3I5)
RETURN
The coding of the example E.7 is straightforward. In the lines 1 to 28, all variables and parameters needed for DLSODAR and the communication between the function evaluation subroutine FEX, are defined and initialized. The communication and the state of the system is stored in the array flag, which is passed to the function evaluation subroutine by the common block flags, declared on lines four and five. The initial conditions of the system are initialized on lines seven to ten.

The main loop of the integrator starts on line 29 and is closed on line 45. Within this loop, DLSODAR is called as often as needed and its output variables, especially istate is checked (see also table E.2. If a root has been found, the new state of the system is stored in the flags array. This additional array is needed, as an access to JROOT, or a direct test to the root conditions, is not allowed during the integration step.

Finally, collected statistical information is written on lines 46 through 50. This gives some information on the number of steps needed for the problem, the number of Jacobian evaluations, the number of root function evaluations and, of course, the number of function evaluations.

The evaluation of the system of ODEs from equation E.7 is coded on lines 57 through 83 (subroutine FEX). The first three equations from equation E.7 are straightforward with a little modification due to performance (number of floating point operations). The fourth equation needs a little more attention. An IF statement is needed to distinguish the four cases as required by equation E.7. As a direct access to the state variables of the form IF (Y(1) .GT. Y(3)) is not allowed, the array flag is needed. This array is passed by its corresponding common block on line 61. The flags in this array are only modified from outside the integrator when a root was found and, however, the integrator has been restarted as required.

Last but not least, on lines 85 through 92, the root functions are described to catch the events, when the model of equation E.7 needs to be changed.
The output produced by the program above looks like

For the interpretation of the output, a plot of all the data is required. This is done using gnuplot\[^3\][WK98]. The Plot from the chem.f example is given in figure E.1.

Note the location of the roots due to the following conditions:

<table>
<thead>
<tr>
<th>Time t</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.9223 \times 10^{-1}$</td>
<td>$y_1 = 10^{-3}$</td>
</tr>
<tr>
<td>$0.2683 \times 10^3$</td>
<td>$y_1 = y_3$</td>
</tr>
<tr>
<td>$0.2053 \times 10^7$</td>
<td>$y_3 = 100y_2$</td>
</tr>
</tbody>
</table>

The user interface to DLSODAR consists of the following parts:

- The call sequence to the subroutine DLSODAR, which is a driver routine for the solver. This includes descriptions of both the call sequence arguments and of user-supplied routines. Following these descriptions is a description of optional inputs available through the call sequence, and then a description of optional outputs (in the work arrays).

- Descriptions of other routines in the DLSODAR package that may be (optionally) called by the user. These provide the ability to alter error message handling, save and restore the internal common, and obtain specified derivatives of the solution $y(t)$.

- Descriptions of common blocks to be declared in overlay or similar environments, or to be saved when doing an interrupt of the problem and continued solution later.

- Description of a subroutine in the DLSODAR package, which the user may replace with his/her own version, if desired. This relates to the measurement of errors.

The following section describes the call sequence to DLSODAR with its parameters. As this work does not make relevant use of the other routines which might be called optionally, nor does it use the internal common blocks, these sections are not described here in detail; see the original documentation included in the software package\[^4\] for details.

\[^3\]gnuplot is command-driven interactive function plotting program. If files are given, gnuplot loads each file with the load command, in the order specified. Gnuplot exits after the last file is processed.

\[^4\]This work is based on the ODEPACK version dated 20 June 2001, available from netlib (www.netlib.org).
### Listing E.4: Output produced by the Example chem.f

<table>
<thead>
<tr>
<th>At t</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1000E-01</td>
<td>0.999601E+00</td>
<td>0.364505E-04</td>
<td>0.362867E-03</td>
<td>0.136287E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2000E-01</td>
<td>0.999204E+00</td>
<td>0.363773E-04</td>
<td>0.759448E-03</td>
<td>0.175945E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4000E-01</td>
<td>0.998413E+00</td>
<td>0.362317E-04</td>
<td>0.155027E-02</td>
<td>0.255027E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8000E-01</td>
<td>0.996851E+00</td>
<td>0.359454E-04</td>
<td>0.311316E-02</td>
<td>0.411316E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9223E-01</td>
<td>0.996378E+00</td>
<td>0.358591E-04</td>
<td>0.358591E-02</td>
<td>0.458591E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above line is a root, $JROOT = 0$ 0 1

<table>
<thead>
<tr>
<th>At t</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1600E+00</td>
<td>0.993798E+00</td>
<td>0.353913E-04</td>
<td>0.616630E-02</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.3200E+00</td>
<td>0.987964E+00</td>
<td>0.343522E-04</td>
<td>0.120012E-01</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6400E+00</td>
<td>0.977249E+00</td>
<td>0.325112E-04</td>
<td>0.227181E-01</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1280E+01</td>
<td>0.958308E+00</td>
<td>0.295519E-04</td>
<td>0.411400E-01</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2560E+01</td>
<td>0.930068E+00</td>
<td>0.254423E-04</td>
<td>0.699069E-01</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5120E+01</td>
<td>0.889958E+00</td>
<td>0.206874E-04</td>
<td>0.110021E+00</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1024E+02</td>
<td>0.839479E+00</td>
<td>0.160852E-04</td>
<td>0.160505E+00</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2048E+02</td>
<td>0.780250E+00</td>
<td>0.121784E-04</td>
<td>0.219738E+00</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4096E+02</td>
<td>0.713404E+00</td>
<td>0.909194E-05</td>
<td>0.286587E+00</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8192E+02</td>
<td>0.639623E+00</td>
<td>0.672329E-05</td>
<td>0.360370E+00</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1638E+03</td>
<td>0.559737E+00</td>
<td>0.492050E-05</td>
<td>0.440258E+00</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2683E+03</td>
<td>0.499998E+00</td>
<td>0.390833E-05</td>
<td>0.499998E+00</td>
<td>0.470238E-02</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above line is a root, $JROOT = 0$ 1 0
At $t = 0.2684E+07$ $Y = 0.766914E-03$ $0.306998E-08$ $0.999233E+00$ $0.157316E-08$
At $t = 0.5369E+07$ $Y = 0.385144E-03$ $0.154116E-08$ $0.999615E+00$ $0.157185E-08$
At $t = 0.1074E+08$ $Y = 0.193032E-03$ $0.772274E-09$ $0.999807E+00$ $0.157187E-08$
At $t = 0.2147E+08$ $Y = 0.968241E-04$ $0.367333E-09$ $0.999903E+00$ $0.157186E-08$
At $t = 0.4295E+08$ $Y = 0.484202E-04$ $0.193690E-09$ $0.999952E+00$ $0.157188E-08$
At $t = 0.8590E+08$ $Y = 0.245967E-04$ $0.983893E-10$ $0.999975E+00$ $0.157188E-08$
At $t = 0.1718E+09$ $Y = 0.129136E-04$ $0.516552E-10$ $0.999987E+00$ $0.157188E-08$

No. steps = 358  No. f-s = 784  No. J-s = 68  No. g-s = 423
Method last used = 2  Last switch was at $t = 0.6009E-02$
The call sequence parameters used for input only are F, NEQ, TOUT, ITOL, RTOL, ATOL, ITASK, IOPT, LRW, LIW, JAC, JT, G, and NG, that used only for output is JROOT, and those used for both input and output are Y, T, ISTATE. The work arrays RWORK and IWORK are also used for conditional and optional inputs and optional outputs. (The term output here refers to the return from Subroutine DLSODAR to the user's calling program.)

The legality of input parameters will be thoroughly checked on the initial call for the problem, but not checked thereafter unless a change in input parameters is flagged by ISTATE = 3 on input.

The descriptions of the call arguments are as follows.

**F:** the name of the user-supplied subroutine defining the ODE system. The system must be put in the first-order form \( \frac{dy}{dt} = f(t,y) \), where \( f \) is a vector-valued function of the scalar \( t \) and the vector \( y \). Subroutine F is to compute the function \( f \). It is to have the form

```
SUBROUTINE F (NEQ, T, Y, YDOT)
```
DOUBLE PRECISION T, Y(*), YDOT(*)

where NEQ, T, and Y are input, and the array YDOT = f(t,y)
is output. Y and YDOT are arrays of length NEQ.

Subroutine F should not alter Y(1), ..., Y(NEQ). F must be declared External in the calling program. Subroutine F may access user-defined quantities in NEQ(2), ... and/or in Y(NEQ(1)+1), ... if NEQ is an array (dimensioned in F) and/or Y has length exceeding NEQ(1). See the descriptions of NEQ and Y below.

If quantities computed in the F routine are needed externally to DLSODAR, an extra call to F should be made for this purpose, for consistent and accurate results. If only the derivative \( \frac{dy}{dt} \) is needed, use DINTDY instead.

**NEQ**: the size of the ODE system (number of first order ordinary differential equations). Used only for input. NEQ may be decreased, but not increased, during the problem. If NEQ is decreased (with ISTATE = 3 on input), the remaining components of Y should be left undisturbed, if these are to be accessed in F and/or JAC.

Normally, NEQ is a scalar, and it is generally referred to as a scalar in this user interface description. However, NEQ may be an array, with NEQ(1) set to the system size. (The DL-SODAR package accesses only NEQ(1).) In either case, this parameter is passed as the NEQ argument in all calls to F, JAC, and G. Hence, if it is an array, locations NEQ(2), ... may be used to store other integer data and pass it to F, JAC, and G. Each such subroutine must include NEQ in a Dimension statement in that case.

**Y**: a real array for the vector of dependent variables, of length NEQ or more. Used for both input and output on the first call (ISTATE = 1), and only for output on other calls. On the first call, Y must contain the vector of initial values. On output, Y contains the computed solution vector, evaluated at T. If desired, the Y array may be used for other purposes between calls to the solver.

This array is passed as the Y argument in all calls to F, JAC, and G. Hence its length may exceed NEQ, and loca-
tions \( Y(\text{NEQ}+1), \ldots \) may be used to store other real data and pass it to \( F, JAC, \) and \( G. \) (The DLSODAR package accesses only \( Y(1), \ldots, Y(\text{NEQ}). \))

\( T: \) the independent variable. On input, \( T \) is used only on the first call, as the initial point of the integration. On output, after each call, \( T \) is the value at which a computed solution \( y \) is evaluated (usually the same as \( \text{TOUT} \)). If a root was found, \( T \) is the computed location of the root reached first, on output. On an error return, \( T \) is the farthest point reached.

\( \text{TOUT}: \) the next value of \( t \) at which a computed solution is desired. Used only for input.

When starting the problem (\( \text{ISTATE} = 1 \)), \( \text{TOUT} \) may be equal to \( T \) for one call, then should \( \text{ne.} \) \( T \) for the next call. For the initial \( T, \) an input value of \( \text{TOUT} \) \( \text{ne.} \) \( T \) is used in order to determine the direction of the integration (i.e. the algebraic sign of the step sizes) and the rough scale of the problem. Integration in either direction (forward or backward in \( t \)) is permitted.

If \( \text{ITASK} = 2 \) or 5 (one-step modes), \( \text{TOUT} \) is ignored after the first call (i.e. the first call with \( \text{TOUT} \) \( \text{ne.} \) \( T \)). Otherwise, \( \text{TOUT} \) is required on every call.

If \( \text{ITASK} = 1, 3, \) or 4, the values of \( \text{TOUT} \) need not be monotone, but a value of \( \text{TOUT} \) which backs up is limited to the current internal \( T \) interval, whose endpoints are \( \text{TCUR} - \text{HU} \) and \( \text{TCUR} \) (see optional outputs, below, for \( \text{TCUR} \) and \( \text{HU} \)).

\( \text{ITOL}: \) an indicator for the type of error control. See description below under \( \text{ATOL}. \) Used only for input.

\( \text{RTOL}: \) a relative error tolerance parameter, either a scalar or an array of length \( \text{NEQ}. \) See description below under \( \text{ATOL}. \) Input only.

\( \text{ATOL}: \) an absolute error tolerance parameter, either a scalar or an array of length \( \text{NEQ}. \) Input only.

The input parameters \( \text{ITOL}, \text{RTOL}, \) and \( \text{ATOL} \) determine the error control performed by the solver. The solver will control the vector \( E = (E(i)) \) of estimated local errors in \( y, \)
according to an inequality of the form
\[
\left\| \frac{E_i}{\omega_i} \right\|_{\text{max}} \leq 1
\]  
(E.9)

where \( \omega = (EWT(i)) \) is a vector of positive error weights. The values of \( \text{RTOL} \) and \( \text{ATOL} \) should all be non-negative. The following table gives the types (scalar/array) of \( \text{RTOL} \) and \( \text{ATOL} \), and the corresponding form of \( \text{EWT}(i) \):

<table>
<thead>
<tr>
<th>ITOL</th>
<th>RTOL</th>
<th>ATOL</th>
<th>EWT(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>scalar</td>
<td>scalar</td>
<td>( \text{RTOL} \cdot \text{ABS}(Y(i)) + \text{ATOL} )</td>
</tr>
<tr>
<td>2</td>
<td>scalar</td>
<td>array</td>
<td>( \text{RTOL} \cdot \text{ABS}(Y(i)) + \text{ATOL}(i) )</td>
</tr>
<tr>
<td>3</td>
<td>array</td>
<td>scalar</td>
<td>( \text{RTOL}(i) \cdot \text{ABS}(Y(i)) + \text{ATOL} )</td>
</tr>
<tr>
<td>4</td>
<td>array</td>
<td>array</td>
<td>( \text{RTOL}(i) \cdot \text{ABS}(Y(i)) + \text{ATOL}(i) )</td>
</tr>
</tbody>
</table>

When either of these parameters is a scalar, it need not be dimensioned in the user's calling program.

If none of the above choices (with \( \text{ITOL} \), \( \text{RTOL} \), and \( \text{ATOL} \) fixed throughout the problem) is suitable, more general error controls can be obtained by substituting a user-supplied routine for the setting of \( \text{EWT} \). See below.

If global errors are to be estimated by making a repeated run on the same problem with smaller tolerances, then all components of \( \text{RTOL} \) and \( \text{ATOL} \) (i.e. of \( \text{EWT} \)) should be scaled down uniformly.

**ITASK**: an index specifying the task to be performed. input only.

**ITASK** has the following values and meanings.

- **ITASK = 1** means normal computation of output values of \( y(t) \) at \( t = \text{TOUT} \) (by overshooting and interpolating).
- **ITASK = 2** means take one step only and return.
- **ITASK = 3** means stop at the first internal mesh point at or beyond \( t = \text{TOUT} \) and return.
- **ITASK = 4** means normal computation of output values of \( y(t) \) at \( t = \text{TOUT} \) but without overshooting \( t = \text{TCRIT} \). \( \text{TCRIT} \) must be input as \( \text{RWORK}(1) \). \( \text{TCRIT} \) may be equal to or beyond \( \text{TOUT} \), but not behind it in the direction of integration. This option is useful if the problem has a singularity at or beyond \( t = \text{TCRIT} \).
\textbf{ITASK = 5} means take one step, without passing TCRIT, and return. TCRIT must be input as RWORK(1).

Note: If \textbf{ITASK = 4} or 5 and the solver reaches TCRIT (within roundoff), it will return T = TCRIT (exactly) to indicate this (unless \textbf{ITASK = 4} and TOUT comes before TCRIT, in which case answers at t = TOUT are returned first).

\textbf{ISTATE}: an index used for input and output to specify the the state of the calculation.

On input, the values of ISTATE are as follows.

\textbf{ISTATE = 1} means this is the first call for the problem (initializations will be done). See note below.

\textbf{ISTATE = 2} means this is not the first call, and the calculation is to continue normally, with no change in any input parameters except possibly TOUT and ITASK. (If ITOL, RTOL, and/or ATOL are changed between calls with ISTATE = 2, the new values will be used but not tested for legality.)

\textbf{ISTATE = 3} means this is not the first call, and the calculation is to continue normally, but with a change in input parameters other than TOUT and ITASK. Changes are allowed in NEQ, ITOL, RTOL, ATOL, IOPT, LRW, LIW, JT, ML, MU, and any optional inputs except HO, MXORDN, and MXORDS. (See IWORK description for ML and MU.) In addition, immediately following a return with ISTATE = 3 (root found), NG and G may be changed. (But changing NG from 0 to > 0 is not allowed.)

Note: A preliminary call with TOUT = T is not counted as a first call here, as no initialization or checking of input is done. (Such a call is sometimes useful for the purpose of outputting the initial conditions.) Thus the first call for which TOUT neq T requires ISTATE = 1 on input.

On output, ISTATE has the following values and meanings:
1 means nothing was done; TOUT = t and ISTATE = 1 on input.
2 means the integration was performed successfully, and no roots were found.
3 means the integration was successful, and one or more roots were found before satisfying the stop condition specified by ITASK. See JROOT.
-1 means an excessive amount of work (more than MXSTEP steps) was done on this call, before completing the requested task, but the integration was otherwise successful as far as T. (MXSTEP is an optional input and is normally 500.) To continue, the user may simply reset ISTATE to a value > 1 and call again (the excess work step counter will be reset to 0). In addition, the user may increase MXSTEP to avoid this error return (see below on optional inputs).
-2 means too much accuracy was requested for the precision of the machine being used. This was detected before completing the requested task, but the integration was successful as far as T. To continue, the tolerance parameters must be reset, and ISTATE must be set to 3. The optional output TOLSF may be used for this purpose. (Note: If this condition is detected before taking any steps, then an illegal input return (ISTATE = -3) occurs instead.)
-3 means illegal input was detected, before taking any integration steps. See written message for details. Note: If the solver detects an infinite loop of calls to the solver with illegal input, it will cause the run to stop.
-4 means there were repeated error test failures on one attempted step, before completing the requested task, but the integration was successful as far as T. The problem may have a singularity, or the input may be inappropriate.
-5 means there were repeated convergence test failures on one attempted step, before completing the requested task, but the integration was successful as far as T. This may be caused by an inaccurate Jacobian matrix, if one is being used.
-6 means EWT(i) became zero for some i during the integration. Pure relative error control (ATOL(i) = 0.0) was requested on a variable which has now vanished. The integration was successful as far as T.
means the length of \texttt{RWORK} and/or \texttt{IWORK} was too small to proceed, but the integration was successful as far as \texttt{T}. This happens when \texttt{DLSODAR} chooses to switch methods but \texttt{LRW} and/or \texttt{LIW} is too small for the new method.

Note: Since the normal output value of \texttt{ISTATE} is 2, it does not need to be reset for normal continuation. Also, since a negative input value of \texttt{ISTATE} will be regarded as illegal, a negative output value requires the user to change it, and possibly other inputs, before calling the solver again.

\textbf{IOPT}: an integer flag to specify whether or not any optional inputs are being used on this call. Input only. The optional inputs are as follows:

\begin{itemize}
  \item \texttt{IOPT = 0} means no optional inputs are being used. Default values will be used in all cases.
  \item \texttt{IOPT = 1} means one or more optional inputs are being used.
\end{itemize}

\textbf{RWORK}: a real array (double precision) for work space, and (in the first 20 words) for conditional and optional inputs and optional outputs. As \texttt{DLSODAR} switches automatically between stiff and non-stiff methods, the required length of \texttt{RWORK} can change during the problem. Thus the \texttt{RWORK} array passed to \texttt{DLSODAR} can either have a static (fixed) length large enough for both methods, or have a dynamic (changing) length altered by the calling program in response to output from \texttt{DLSODAR}.

\begin{itemize}
  \item \textbf{fixed Length Case} If the \texttt{RWORK} length is to be fixed, it should be at least \texttt{max (LRN, LRS)}, where \texttt{LRN} and \texttt{LRS} are the \texttt{RWORK} lengths required when the current method is non-stiff or stiff, respectively. The separate \texttt{RWORK} length requirements \texttt{LRN} and \texttt{LRS} are as follows:
    \begin{itemize}
      \item If \texttt{NEQ} is constant and the maximum method orders have their default values, then
        \begin{align*}
          \texttt{LRN} &= 20 + 16 \times \texttt{NEQ} + 3 \times \texttt{NG}, \\
          \texttt{LRS} &= 22 + 9 \times \texttt{NEQ} + \texttt{NEQ} \times 2 + 3 \times \texttt{NG}
        \end{align*}
      \text{if } \texttt{JT} = 1 \text{ or 2:}
      \text{if } \texttt{JT} = 4 \text{ or 5:}
    \end{itemize}
\end{itemize}
LRS = 22 + 10*NEQ + (2*ML+MU)*NEQ + 3*NG

- Under any other conditions, LRN and LRS are given by:
  
  LRN = 20 + NYH*(MXORDN+1) + 3*NEQ + 3*NG,
  LRS = 20 + NYH*(MXORDS+1) + 3*NEQ + LMAT + 3*NG,

where

NYH = the initial value of NEQ,
MXORDN = 12, unless a smaller value is given as an optional input,
MXORDS = 5, unless a smaller value is given as an optional input,
LMAT = length of matrix work space:
LMAT = NEQ**2 + 2 if JT = 1 or 2,
LMAT = (2*ML + MU + 1)*NEQ + 2 if JT = 4 or 5.

- **Dynamic Length Case** If the length of RWORK is to be dynamic, then it should be at least LRN or LRS, as defined above, depending on the current method. Initially, it must be at least LRN (since DLSODAR starts with the non-stiff method). On any return from DLSODAR, the optional output MCUR indicates the current method. If MCUR differs from the value it had on the previous return, or if there has only been one call to DLSODAR and MCUR is now 2, then DLSODAR has switched methods during the last call, and the length of RWORK should be reset (to LRN if MCUR = 1, or to LRS if MCUR = 2). (An increase in the RWORK length is required if DLSODAR returned ISTATE = -7, but not otherwise.) After resetting the length, call DLSODAR with ISTATE = 3 to signal that change.

LRW: the length of the array RWORK, as declared by the user. (This will be checked by the solver.)

IWORK: an integer array for work space. DLSODAR switches automatically between stiff and non-stiff methods. Thus, the required length of IWORK can change during problem, between LIS = 20 + NEQ and LIN = 20, respectively. Thus the IWORK array passed to DLSODAR can either have a fixed length of at least 20 + NEQ, or have a dynamic length of at least LIN or LIS, depending on the current
method. The comments on dynamic length under RWORK above apply here. Initially, this length need only be at least LIN = 20.

The first few words of IWORK are used for conditional and optional inputs and optional outputs.

The following two words in IWORK are conditional inputs:
- IWORK(1) = ML
- IWORK(2) = MU

These are the lower and upper half-bandwidths, respectively, of the banded Jacobian, excluding the main diagonal. The band is defined by the matrix locations \((i, j)\) with \(i - ML \leq j \leq i + MU\). ML and MU must satisfy \(0 < ML, MU \leq NEQ - 1\). These are required if JT is 4 or 5, and ignored otherwise. ML and MU may in fact be the band parameters for a matrix to which \(\frac{df}{dy}\) is only approximately equal.

**LIW:** the length of the array IWORK, as declared by the user. (This will be checked by the solver.)

Note: The base addresses of the work arrays must not be altered between calls to DLSODAR for the same problem. The contents of the work arrays must not be altered between calls, except possibly for the conditional and optional inputs, and except for the last \(3 \times NEQ\) words of RWORK. The latter space is used for internal scratch space, and so is available for use by the user outside DLSODAR between calls, if desired (but not for use by \(F, JAC, \) or \(G\)).

**JAC:** the name of the user-supplied routine to compute the Jacobian matrix, \(\frac{df}{dy}\), if JT = 1 or 4. The JAC routine is optional, but if the problem is expected to be stiff much of the time, you are encouraged to supply JAC, for the sake of efficiency. (Alternatively, set JT = 2 or 5 to have DLSODAR compute \(\frac{df}{dy}\) internally by difference quotients.) If and when DLSODAR uses \(\frac{df}{dy}\), it treats this \(NEQ \times NEQ\) matrix either as full (JT = 1 or 2), or as banded (JT = 4 or 5) with half-bandwidths ML and MU (discussed under IWORK above). In either case, if JT = 1 or 4, the JAC routine must compute \(\frac{df}{dy}\)
as a function of the scalar $t$ and the vector $y$. It is to have the form

$$\text{SUBROUTINE JAC (NEQ, T, Y, ML, MU, PD, NROWPD)}$$
$$\text{DOUBLE PRECISION T, Y(*)}, \text{PD(NROWPD,*)}$$

where $\text{NEQ}$, $T$, $Y$, $\text{ML}$, $\text{MU}$, and $\text{NROWPD}$ are input and the array $\text{PD}$ is to be loaded with partial derivatives (elements of the Jacobian matrix) on output. $\text{PD}$ must be given a first dimension of $\text{NROWPD}$. $T$ and $Y$ have the same meaning as in Subroutine $F$.

In the full matrix case ($JT = 1$), $\text{ML}$ and $\text{MU}$ are ignored, and the Jacobian is to be loaded into $\text{PD}$ in column-wise manner, with $\frac{df_i}{dy_j}$ loaded into $pd(i, j)$.

In the band matrix case ($JT = 4$), the elements within the band are to be loaded into $\text{PD}$ in column-wise manner, with diagonal lines of $\frac{df_i}{dy_j}$ loaded into the rows of $\text{PD}$. Thus $\frac{df_i}{dy_j}$ is to be loaded into $\text{PD} (i-j+\text{MU}+1, j)$. $\text{ML}$ and $\text{MU}$ are the half-bandwidth parameters (see $\text{IWORK}$). The locations in $\text{PD}$ in the two triangular areas which correspond to nonexistent matrix elements can be ignored or loaded arbitrarily, as they are overwritten by DLSODAR.

$\text{JAC}$ need not provide $\frac{df}{dy}$ exactly. A crude approximation (possibly with a smaller bandwidth) will do. In either case, $\text{PD}$ is preset to zero by the solver, so that only the nonzero elements need be loaded by $\text{JAC}$. Each call to $\text{JAC}$ is preceded by a call to $\text{F}$ with the same arguments $\text{NEQ}$, $T$, and $Y$. Thus to gain some efficiency, intermediate quantities shared by both calculations may be saved in a user Common block by $\text{F}$ and not recomputed by $\text{JAC}$, if desired. Also, $\text{JAC}$ may alter the $Y$ array, if desired. $\text{JAC}$ must be declared External in the calling program.

The subroutine $\text{JAC}$ may access user-defined quantities in $\text{NEQ}(2), \ldots$ and/or in $Y(\text{NEQ}(1)+1), \ldots$ if $\text{NEQ}$ is an array (dimensioned in $\text{JAC}$) and/or $Y$ has length exceeding $\text{NEQ}(1)$. See the descriptions of $\text{NEQ}$ and $Y$ above.

$JT$: Jacobian type indicator. Used only for input. $JT$ specifies how the Jacobian matrix $\frac{df}{dy}$ will be treated, if and when
DLSODAR requires this matrix. JT has the following values and meanings:

1. means a user-supplied full (NEQ × NEQ) Jacobian.
2. means an internally generated (difference quotient) full Jacobian (using NEQ extra calls to $\frac{df}{dy}$ value).
4. means a user-supplied banded Jacobian.
5. means an internally generated banded Jacobian (using ML+MU+1 extra calls to $\frac{df}{dy}$ evaluation).

If JT = 1 or 4, the user must supply a Subroutine JAC (the name is arbitrary) as described above under JAC. If JT = 2 or 5, a dummy argument can be used.

G: the name of subroutine for constraint functions, whose roots are desired during the integration. It is to have the form

```fortran
SUBROUTINE G (NEQ, T, Y, NG, GOUT)
DOUBLE PRECISION T, Y(*), GOUT(NG)
```

where NEQ, T, Y, and NG are input, and the array GOUT is output. NEQ, T, and Y have the same meaning as in the F routine, and GOUT is an array of length NG. For $j = 1, \ldots, NG$, this routine is to load into GOUT($j$) the value at (T, Y) of the $j$-th constraint function $g_j$. DLSODAR will find roots of the $g_j$ of odd multiplicity (i.e. sign changes) as they occur during the integration. G must be declared External in the calling program.

Caution: Because of numerical errors in the functions $g_j$ due to roundoff and integration error, DLSODAR may return false roots, or return the same root at two or more nearly equal values of $t$. If such false roots are suspected, the user should consider smaller error tolerances and/or higher precision in the evaluation of the $g_j$.

If a root of some $g_j$ defines the end of the problem, the input to DLSODAR should nevertheless allow integration to a point slightly past that root, so that DLSODAR can locate the root by interpolation.

The subroutine G may access user-defined quantities in $\text{NEQ}(2), \ldots$ and $\text{Y}(\text{NEQ}(1)+1), \ldots$ if NEQ is an array (dimensioned in G) and/or Y has length exceeding NEQ(1). See the descriptions of NEQ and Y above.
NG: number of constraint functions $g_j$. If there are none, set NG = 0, and pass a dummy name for G.

JROOT: integer array of length NG. Used only for output. On a return with ISTATE = 3 (one or more roots found), JROOT(j) = 1 if $g_j$ has a root at T, or JROOT(j) = 0 if not.

Optional Inputs

The optional inputs for the call sequence is given in table E.2.2. For each such input variable, this table lists its name as used in this documentation, its location in the call sequence, its meaning, and the default value. The use of any of these inputs requires IOPT = 1, and in that case all of these inputs are examined. A value of zero for any of these optional inputs will cause the default value to be used. Thus to use a subset of the optional inputs, simply preload locations 5 to 10 in RWORK and IWORK to 0.0 and 0 respectively, and then set those of interest to nonzero values.

Optional Outputs

As optional additional output from DLSODAR, the variables listed in table E.2.2 are quantities related to the performance of DLSODAR which are available to the user. These are communicated by way of the work arrays, but also have internal mnemonic names as shown. Except where stated otherwise, all of these outputs are defined on any successful return from DLSODAR, and on any return with ISTATE = -1, -2, -4, -5, or -6. On an illegal input return (ISTATE = -3), they will be unchanged from their existing values (if any), except possibly for TOLSF, LENRW, and LENIW. On any error return, outputs relevant to the error will be defined, as noted below.

The following two arrays are segments of the RWORK array which may also be of interest to the user as optional outputs. For each array, the table below gives its internal name, its base address in RWORK, and its description.
### LSODAR optional input parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Meaning and Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>RWORK(5)</td>
<td>the step size to be attempted on the first step. The default value is determined by the solver.</td>
</tr>
<tr>
<td>HMAX</td>
<td>RWORK(6)</td>
<td>the maximum absolute step size allowed. The default value is infinite.</td>
</tr>
<tr>
<td>HMIN</td>
<td>RWORK(7)</td>
<td>the minimum absolute step size allowed. The default value is 0. (This lower bound is not enforced on the final step before reaching TCRIT when ITASK = 4 or 5.)</td>
</tr>
<tr>
<td>IXPR</td>
<td>IWORK(5)</td>
<td>flag to generate extra printing at method switches. IXPR = 0 means no extra printing (the default). IXPR = 1 means print data on each switch. T, H, and NST will be printed on the same logical unit as used for error messages.</td>
</tr>
<tr>
<td>MXSTEP</td>
<td>IWORK(6)</td>
<td>maximum number of (internally defined) steps allowed during one call to the solver. The default value is 500.</td>
</tr>
<tr>
<td>MXHNIL</td>
<td>IWORK(7)</td>
<td>maximum number of messages printed (per problem) warning that T + H = T on a step (H = step size). This must be positive to result in a non-default value. The default value is 10.</td>
</tr>
<tr>
<td>MXORDN</td>
<td>IWORK(8)</td>
<td>the maximum order to be allowed for the non-stiff (Adams) method. The default value is 12. If MXORDN exceeds the default value, it will be reduced to the default value. MXORDN is held constant during the problem.</td>
</tr>
<tr>
<td>MXORDS</td>
<td>IWORK(9)</td>
<td>the maximum order to be allowed for the stiff (BDF) method. The default value is 5. If MXORDS exceeds the default value, it will be reduced to the default value. MXORDS is held constant during the problem.</td>
</tr>
</tbody>
</table>

*Table E.3: LSODAR optional input parameters*
**LSODAR optional output parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HU</td>
<td>RWORK(11)</td>
<td>the step size in t last used (successfully).</td>
</tr>
<tr>
<td>HCUR</td>
<td>RWORK(12)</td>
<td>the step size to be attempted on the next step.</td>
</tr>
<tr>
<td>TCUR</td>
<td>RWORK(13)</td>
<td>the current value of the independent variable which the solver has actually reached, i.e. the current internal meshpoint in t. On output, TCUR will always be at least as far as the argument T, but may be farther (if interpolation was done).</td>
</tr>
<tr>
<td>TOLSF</td>
<td>RWORK(14)</td>
<td>a tolerance scale factor, greater than 1.0, computed when a request for too much accuracy was detected (ISTATE = -3 if detected at the start of the problem, ISTATE = -2 otherwise). If ITOL is left unaltered but RTOL and ATOL are uniformly scaled up by a factor of TOLSF for the next call, then the solver is deemed likely to succeed. (The user may also ignore TOLSF and alter the tolerance parameters in any other way appropriate.)</td>
</tr>
<tr>
<td>TSW</td>
<td>RWORK(15)</td>
<td>the value of t at the time of the last method switch, if any.</td>
</tr>
<tr>
<td>NGE</td>
<td>IWORK(10)</td>
<td>the number of g evaluations for the problem so far.</td>
</tr>
<tr>
<td>NST</td>
<td>IWORK(11)</td>
<td>the number of steps taken for the problem so far.</td>
</tr>
<tr>
<td>NFE</td>
<td>IWORK(12)</td>
<td>the number of f evaluations for the problem so far.</td>
</tr>
<tr>
<td>NJE</td>
<td>IWORK(13)</td>
<td>the number of Jacobian evaluations (and of matrix LU decompositions) for the problem so far.</td>
</tr>
<tr>
<td>NQU</td>
<td>IWORK(14)</td>
<td>the method order last used (successfully).</td>
</tr>
<tr>
<td>NQCUR</td>
<td>IWORK(15)</td>
<td>the order to be attempted on the next step.</td>
</tr>
<tr>
<td>IMXER</td>
<td>IWORK(16)</td>
<td>the index of the component of largest magnitude in the weighted local error vector (E(i)/EWT(i)), on an error return with ISTATE = -4 or -5.</td>
</tr>
<tr>
<td>Name</td>
<td>Location</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LENRW</td>
<td>IWORK(17)</td>
<td>the length of RWORK actually required, assuming that the length of RWORK is to be fixed for the rest of the problem, and that switching may occur. This is defined on normal returns and on an illegal input return for insufficient storage.</td>
</tr>
<tr>
<td>LENIW</td>
<td>IWORK(18)</td>
<td>the length of IWORK actually required, assuming that the length of IWORK is to be fixed for the rest of the problem, and that switching may occur. This is defined on normal returns and on an illegal input return for insufficient storage.</td>
</tr>
<tr>
<td>MUSED</td>
<td>IWORK(19)</td>
<td>the method indicator for the last successful step: 1 means Adams (non-stiff), 2 means BDF (stiff).</td>
</tr>
<tr>
<td>MCUR</td>
<td>IWORK(20)</td>
<td>the current method indicator: 1 means Adams (non-stiff), 2 means BDF (stiff). This is the method to be attempted on the next step. Thus it differs from MUSED only if a method switch has just been made.</td>
</tr>
</tbody>
</table>

*Table E.4: LSODAR optional output parameters*
E.3. BLAS (Basic Linear Algebra Subsystem)

BLAS is a library of routines that perform basic operations involving matrices and vectors developed by Jack J. Dongarra et al [LHKK79], [DCHH88b], [DCHH88a], [DCHD90b], [DCHD90a]. They were designed as a way of achieving efficiency in the solution of linear algebra problems. The BLAS, as they are now commonly called, have been very successful and have been used in a wide range of software, including LINPACK, EISPACK, LAPACK and many of the algorithms published by the ACM Transactions on Mathematical Software. They are an aid to clarity, portability, modularity and maintenance of software, and have become the de facto standard for elementary vector and matrix operations.

The BLAS promote modularity by identifying frequently occurring operations of linear algebra and by specifying a standard interface to these operations. Efficiency is achieved through optimization within the BLAS without altering the higher-level code that has referenced them.

There are three levels of BLAS. The original set of BLAS, commonly referred as the Level 1 BLAS, perform low-level operations such as dot-product and the adding of a multiple of one vector to another. Typically these operations involve $O(N)$ floating point oper-
232 Appendix E: Third-Party Numerical Software

ations and $O(N)$ data items moved (loaded or stored), where $N$ is the length of the vectors. The Level 1 BLAS permit efficient implementation on scalar machines, but the ratio of floating-point operations to data movement is too low to achieve effective use of most vector or parallel hardware.

The Level 2 BLAS perform Matrix-Vector operations that occur frequently in the implementation of many of the most common linear algebra algorithms. They involve $O(N^2)$ floating point operations. Algorithms that use Level 2 BLAS can be very efficient on vector computers, but are not well suited to computers with a hierarchy of memory (such as cache memory).

The Level 3 BLAS are targeted at matrix-matrix operations. These operations generally involve $O(N^3)$ floating point operations, while only creating $O(N^2)$ data movement. These operations permit efficient reuse of data that resides in cache and create what is often called the surface-to-volume effect for the ratio of computations to data movement. In addition, matrices can be partitioned into blocks, and operations on distinct blocks can be performed in parallel, and within the operations on each block, scalar or vector operations may be performed in parallel.

Machine-specific optimized BLAS libraries are available for a variety of computer architectures. These optimized BLAS libraries are provided by the computer vendor or by independent software vendors.

Alternatively, the user can use ATLAS [WPD01] [WD99] to automatically generate an optimized BLAS library for his architecture. ATLAS is an approach for the automatic generation and optimization of numerical software for processors with deep memory hierarchies and pipelined functional units. The production of such software for machines ranging from desktop workstations to embedded processors can be a tedious and time consuming task. ATLAS has been designed to automate much of this process. ATLAS can be downloaded from netlib\textsuperscript{5}.

Beside the optimized and compiled code, the user can download the Fortran77 reference implementation of the BLAS from netlib. However, this reference implementation is not optimized.

\textsuperscript{5}www.netlib.org
E.4. LINPACK

LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square by applying direct methods. In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems. LINPACK uses column-oriented algorithms to increase efficiency by preserving locality of reference.

LINPACK has been Developed by Jack Dongarra, Jim Bunch, Cleve Moler and Pete Stewart.[DMBS79]

An excellent manual for the linpack routines is provided by Sun Microsystems[SM].
The GNU General Public License

Version 2, June 1991
Copyright © 1989, 1991 Free Software Foundation, Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the
GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

**GNU General Public License**

**Terms and Conditions For Copying, Distribution and Modification**

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a
"work based on the Program" means either the Program or any
derivative work under copyright law: that is to say, a work con-
taining the Program or a portion of it, either verbatim or with
modifications and/or translated into another language. (Here-
inafter, translation is included without limitation in the term
"modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are
not covered by this License; they are outside its scope. The act of
running the Program is not restricted, and the output from the
Program is covered only if its contents constitute a work based
on the Program (independent of having been made by running
the Program). Whether that is true depends on what the Program
does.

1. You may copy and distribute verbatim copies of the Program’s
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an ap-
propriate copyright notice and disclaimer of warranty; keep in-
tact all the notices that refer to this License and to the absence
of any warranty; and give any other recipients of the Program a
copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy,
and you may at your option offer warranty protection in ex-
change for a fee.

2. You may modify your copy or copies of the Program or any por-
tion of it, thus forming a work based on the Program, and copy
and distribute such modifications or work under the terms of
Section 1 above, provided that you also meet all of these condi-
tions:

(a) You must cause the modified files to carry prominent no-
tices stating that you changed the files and the date of any
change.

(b) You must cause any work that you distribute or publish,
that in whole or in part contains or is derived from the Pro-
gram or any part thereof, to be licensed as a whole at no
charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interac-
tively when run, you must cause it, when started running
for such interactive use in the most ordinary way, to print or
display an announcement including an appropriate copy-
right notice and a notice that there is no warranty (or else,
saying that you provide a warranty) and that users may re-
distribute the program under these conditions, and telling
the user how to view a copy of this License. (Exception: if
the Program itself is interactive but does not normally print
such an announcement, your work based on the Program is
not required to print an announcement.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Pro-
gram, and can be reasonably considered independent and sepa-
rate works in themselves, then this License, and its terms, do
not apply to those sections when you distribute them as sepa-
rate works. But when you distribute the same sections as part of
a whole which is a work based on the Program, the distribution
of the whole must be on the terms of this License, whose permi-
sions for other licensees extend to the entire whole, and thus to
each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is
to exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the
Program with the Program (or with a work based on the Pro-
gram) on a volume of a storage or distribution medium does not
bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on
it, under Section 2) in object code or executable form under the
terms of Sections 1 and 2 above provided that you also do one of
the following:

(a) Accompany it with the complete corresponding machine-
readable source code, which must be distributed under the
terms of Sections 1 and 2 above on a medium customarily
used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three
years, to give any third party, for a charge no more than
your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License.
Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system
and a licensee cannot impose that choice.
This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN
writing the copyright holders and/or other parties provide the program “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

F.1. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program’s name and a brief idea of what it does. Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General
Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) <year> <name of author> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands show w and show c should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than show w and show c; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine
library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.
## G.1. Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>Area</td>
<td>$m^2$</td>
</tr>
<tr>
<td>$a$</td>
<td>Heat Transfer Coefficient</td>
<td>$\frac{W}{m^2K}$</td>
</tr>
<tr>
<td>$D_n$</td>
<td>Hydraulic Diameter</td>
<td>$m$</td>
</tr>
<tr>
<td>$\dot{M}$</td>
<td>Mass Flow Rate</td>
<td>$\frac{kg}{s}$</td>
</tr>
<tr>
<td>$\dot{Q}$</td>
<td>Heat Flow Rate</td>
<td>$W$</td>
</tr>
<tr>
<td>$\eta$</td>
<td>Dynamic Viscosity</td>
<td>$\frac{Ns}{m^2}$</td>
</tr>
<tr>
<td>$g$</td>
<td>Acceleration of Gravity</td>
<td>$\frac{m}{s^2}$</td>
</tr>
<tr>
<td>$h$</td>
<td>Specific Enthalpy</td>
<td>$\frac{J}{kg}$</td>
</tr>
<tr>
<td>$l$</td>
<td>Lenght</td>
<td>$m$</td>
</tr>
<tr>
<td>$M$</td>
<td>Mass</td>
<td>$kg$</td>
</tr>
<tr>
<td>$p$</td>
<td>Pressure</td>
<td>$Pa$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>Molar Fraction</td>
<td>–</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Specific Mass</td>
<td>$\frac{kg}{m^3}$</td>
</tr>
<tr>
<td>Subscript</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>Temperature</td>
<td></td>
</tr>
<tr>
<td>(\theta)</td>
<td>Mass Transfer Coefficient</td>
<td>(\text{mol , m}^{-2} , \text{mol}^{-1} , \text{m}^{-3})</td>
</tr>
<tr>
<td>V</td>
<td>Volume</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>Mass Fraction</td>
<td></td>
</tr>
</tbody>
</table>

G.2. Subscripts

<table>
<thead>
<tr>
<th>Subscript</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>air</td>
</tr>
<tr>
<td>b</td>
<td>bubbles</td>
</tr>
<tr>
<td>c</td>
<td>cold side</td>
</tr>
<tr>
<td>cond</td>
<td>condensate</td>
</tr>
<tr>
<td>feed</td>
<td>feed flow</td>
</tr>
<tr>
<td>fg</td>
<td>fluid to gas</td>
</tr>
<tr>
<td>g</td>
<td>gas</td>
</tr>
<tr>
<td>h</td>
<td>hot side</td>
</tr>
<tr>
<td>i</td>
<td>interface</td>
</tr>
<tr>
<td>ig</td>
<td>interface to gas</td>
</tr>
<tr>
<td>in</td>
<td>input, inlet</td>
</tr>
<tr>
<td>l</td>
<td>liquid</td>
</tr>
<tr>
<td>lg</td>
<td>liquid to gas</td>
</tr>
<tr>
<td>li</td>
<td>liquid to interface</td>
</tr>
<tr>
<td>out</td>
<td>output, outlet</td>
</tr>
<tr>
<td>pool</td>
<td>pool</td>
</tr>
<tr>
<td>sat</td>
<td>saturated</td>
</tr>
<tr>
<td>tot</td>
<td>total</td>
</tr>
<tr>
<td>v</td>
<td>vapor</td>
</tr>
<tr>
<td>vi</td>
<td>vapor to interface</td>
</tr>
<tr>
<td>wall</td>
<td>wall</td>
</tr>
</tbody>
</table>
### Abbreviations

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery</td>
</tr>
<tr>
<td>ALPHA</td>
<td>Project Name for “Advanced Light Water Reactor Decay Heat Removal and Aerosol Retention”</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ASME</td>
<td>American Society of Mechanical Engineers</td>
</tr>
<tr>
<td>ATLAS</td>
<td>Automatically Tuned Linear Algebra Software: Numerical Programs and Routines</td>
</tr>
<tr>
<td>AT&amp;T</td>
<td>American Telephone and Telegraph Company</td>
</tr>
<tr>
<td>AUX</td>
<td>Auxiliary System Lines</td>
</tr>
<tr>
<td>AWK</td>
<td>pattern scanning and processing language</td>
</tr>
<tr>
<td>BDF</td>
<td>Backward differentiation formula</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subsystem</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus Naur Form</td>
</tr>
<tr>
<td>BWR</td>
<td>Boiling Water Reactor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DLSODAR</td>
<td>Livermore Solver for Ordinary Differential Equations, with Automatic method switching for stiff and non-stiff problems, and with root-finding</td>
</tr>
<tr>
<td>DLSODE</td>
<td>Livermore Solver for Ordinary Differential Equations</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW</td>
<td>Dry Well</td>
</tr>
<tr>
<td>EBNF</td>
<td>Enhanced Backus Naur Form</td>
</tr>
<tr>
<td>EISPACK</td>
<td>A library of FORTRAN90 routines for calculating the eigenvalues and eigenvectors of a matrix</td>
</tr>
<tr>
<td>ESBWR</td>
<td>Economic Simplified Boiling Water Reactor</td>
</tr>
<tr>
<td>GDCS</td>
<td>Gravity Driven Cooling System</td>
</tr>
<tr>
<td>GE</td>
<td>General Electric</td>
</tr>
<tr>
<td>GHz</td>
<td>Giga Hertz ($10^9$ Hz)</td>
</tr>
<tr>
<td>GPL</td>
<td>GNU General Public License</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines, Inc.</td>
</tr>
<tr>
<td>IC</td>
<td>Isolation Condenser</td>
</tr>
<tr>
<td>ICS</td>
<td>Isolation Condenser System</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers, Inc.</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>LALR</td>
<td>Look-Ahead Left to Right (parser)</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Linear Algebra Package</td>
</tr>
<tr>
<td>LINPACK</td>
<td>A library of Fortran 77 subroutines that analyze and solve linear equations and linear least-squares problems</td>
</tr>
<tr>
<td>LOCA</td>
<td>Loss Of Coolant Accident</td>
</tr>
<tr>
<td>MSDL</td>
<td>Modular System Description Language</td>
</tr>
<tr>
<td>MSL</td>
<td>Main Steam Line</td>
</tr>
<tr>
<td>MVC</td>
<td>Model – View – Control</td>
</tr>
<tr>
<td>MVL</td>
<td>Main Vent Line</td>
</tr>
<tr>
<td>MW</td>
<td>Mega Watts ($10^6$ W)</td>
</tr>
<tr>
<td>ODE</td>
<td>Ordinary Differential Equation</td>
</tr>
<tr>
<td>ODEPACK</td>
<td>A Systematized Collection of ODE Solvers</td>
</tr>
<tr>
<td>PANDA</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PCC</td>
<td>Passive Cooling Condenser</td>
</tr>
<tr>
<td>PCCS</td>
<td>Passive Containment Cooling System</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial Differential Equation</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
<tr>
<td>PSI</td>
<td>Paul Scherrer Institute, Villigen, Switzerland</td>
</tr>
<tr>
<td>PV</td>
<td>Pressure Vessel</td>
</tr>
<tr>
<td>PVM</td>
<td>Parallel Virtual Machine</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RPV</td>
<td>Reactor Pressure Vessel</td>
</tr>
<tr>
<td>SBWR</td>
<td>Simplified Boiling Water Reactor</td>
</tr>
<tr>
<td>SC</td>
<td>Suppression Chamber</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>International System of Units</td>
</tr>
<tr>
<td>SIAM</td>
<td>Society for Industrial and Applied Mathematics</td>
</tr>
<tr>
<td>SIGPLAN</td>
<td>ACM Special Interest Group for Programming Languages</td>
</tr>
<tr>
<td>VB</td>
<td>Vacuum Breaker</td>
</tr>
<tr>
<td>WW</td>
<td>Wet Well</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Schematic of the SBWR and its PANDA model ........ 3  
1.2 The computational concept for the simulation .......... 7  

2.1 PANDA experimental facility scheme ..................... 11  
2.2 Main variables in pressure vessel ....................... 13  
2.3 Main variables at the interface .......................... 24  
2.4 Geometrical parameters of the pipe ........................ 30  
2.5 Basic schematic of a PCCS condenser ................. 32  
2.6 The condenser operation modes .......................... 33  
2.7 The condenser in closed operation mode .................. 34  
2.8 Overall logic of the condenser unit ...................... 38  
2.9 Variables and control volumes for a condenser tube ....... 39  
2.10 Pressure and flow rate vs. temperature ................... 44  
2.11 Schematic representation of a condenser unit .......... 46  

3.1 Stability domain of the forward Euler method ........... 55  
3.2 Stability domain of the trapezoidal rule .................. 56  
3.3 Ideal stability domain of Gear's method .................. 60  
3.4 Localization of a Time Event ............................ 64  

4.1 Step 1: Definition of the equations ...................... 69  
4.2 Step 2: Definition of state equations .................... 71  
4.3 Step 3: Solving the system of ODEs ...................... 72  
4.4 Step 4: Generating Fortran Code .......................... 73  

5.1 Compilers for translating an MSDL program ............... 80  
5.2 Workflow from MSDL to simulation output ................. 81  
5.3 Structure of a compiler ................................. 83  
5.4 Front end and back end of a compiler ...................... 84  
5.5 Data streams of a lex program .......................... 88  
5.6 The parser generator yacc ................................ 89  

6.1 PANDA test facility schematic .......................... 99  
6.2 DW1, WW1 and RPV pressures ............................ 107  
6.3 PCC feed flow rates ................................... 107
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Total condensate flow rate</td>
<td>108</td>
</tr>
<tr>
<td>6.5</td>
<td>DW1 gas temperature</td>
<td>108</td>
</tr>
<tr>
<td>6.6</td>
<td>DW2 gas temperature</td>
<td>109</td>
</tr>
<tr>
<td>6.7</td>
<td>WW1 gas, pool surface and liquid temperature</td>
<td>109</td>
</tr>
<tr>
<td>6.8</td>
<td>WW2 gas, pool surface and liquid temperature</td>
<td>110</td>
</tr>
<tr>
<td>6.9</td>
<td>DW air partial pressure</td>
<td>110</td>
</tr>
<tr>
<td>6.10</td>
<td>WW air partial pressure</td>
<td>111</td>
</tr>
<tr>
<td>6.11</td>
<td>RPV collapsed level</td>
<td>111</td>
</tr>
<tr>
<td>6.12</td>
<td>Total pressure evolution calculated and measured</td>
<td>121</td>
</tr>
<tr>
<td>6.13</td>
<td>Reduced heat total pressure evolution</td>
<td>121</td>
</tr>
<tr>
<td>6.14</td>
<td>Total pressure evolution by L. Batet et al.</td>
<td>122</td>
</tr>
<tr>
<td>6.15</td>
<td>Condensate flow rate calculated and measured</td>
<td>122</td>
</tr>
<tr>
<td>6.16</td>
<td>Gas temperature in DW1 calculated and measured</td>
<td>123</td>
</tr>
<tr>
<td>6.17</td>
<td>Gas temperature in DW2 calculated and measured</td>
<td>123</td>
</tr>
<tr>
<td>6.18</td>
<td>Gas temperature in DW2 by L. Batet et al.</td>
<td>124</td>
</tr>
<tr>
<td>A.1</td>
<td>Test run of the gas space module</td>
<td>150</td>
</tr>
<tr>
<td>A.2</td>
<td>Test run of the liquid module</td>
<td>152</td>
</tr>
<tr>
<td>A.3</td>
<td>Test run of the interface with constant pressure</td>
<td>155</td>
</tr>
<tr>
<td>A.4</td>
<td>Test run of the interface with constant temperature</td>
<td>155</td>
</tr>
<tr>
<td>A.5</td>
<td>Comparison of tests on MSDL against Gothic</td>
<td>159</td>
</tr>
<tr>
<td>A.6</td>
<td>Gas pipe test run</td>
<td>162</td>
</tr>
<tr>
<td>A.7</td>
<td>Liquid pipe test run</td>
<td>164</td>
</tr>
<tr>
<td>A.8</td>
<td>PCCS test run</td>
<td>166</td>
</tr>
<tr>
<td>B.1</td>
<td>The role of the Lexer inside the MSDL compiler</td>
<td>167</td>
</tr>
<tr>
<td>B.2</td>
<td>Example setup for code p2.1</td>
<td>173</td>
</tr>
<tr>
<td>B.3</td>
<td>Intermediate data structure of p2.1</td>
<td>176</td>
</tr>
<tr>
<td>E.1</td>
<td>Example plot produced by Program chem.f.</td>
<td>216</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Parameter values for the k-th order Gear algorithm . . 62
5.1 Lexical symbols in MSDL . . . . . . . . . . . . . . . . . . . 96
6.1 Test P3 initial conditions . . . . . . . . . . . . . . . . . . . 105
6.2 Top cpu time consuming functions in P3 Simulation . . 117
A.1 Properties of mass . . . . . . . . . . . . . . . . . . . . . . . 147
A.2 The gas space module . . . . . . . . . . . . . . . . . . . . . 149
A.3 The liquid space module . . . . . . . . . . . . . . . . . . . . 151
A.4 The interface module . . . . . . . . . . . . . . . . . . . . . . 154
A.5 The pressure vessel module . . . . . . . . . . . . . . . . . . 158
A.6 Events during pressure vessel test run . . . . . . . . . . . 160
A.7 The gas pipe module calling interface . . . . . . . . . . . 161
A.8 The liquid pipe module . . . . . . . . . . . . . . . . . . . . . 163
A.9 The condenser module . . . . . . . . . . . . . . . . . . . . . 165
B.1 Support Routines for MSDL Parser . . . . . . . . . . . . . 177
E.1 Parameters used in DLSODAR . . . . . . . . . . . . . . . . 208
E.2 Output parameters in the DLSODAR package . . . . . . . 209
E.3 LSODAR optional input parameters . . . . . . . . . . . . . 228
E.4 LSODAR optional output parameters . . . . . . . . . . . . 230
Listings

4.1 Automatically generated Fortran code ............ 70
5.1 Character stream to parse .......................... 89
5.2 MSDL Example setup for P3 Simulation ............ 91
6.1 Part 1: Declaration of Vessels ..................... 112
6.2 Part 2: Geometrical Properties of Vessels .......... 112
6.3 Part 3: Declaration of Pipes ........................ 113
6.4 Part 4: Geometrical Properties of Pipes .......... 113
6.5 Part 5: Initial Conditions ........................... 114
A.1 vnorm.f (from ODE Pack) ......................... 131
A.2 Fortran variable declaration ....................... 133
A.3 AWK Program for generating declarations ........ 133
A.4 Automatically generated, native Fortran Code ...... 137
A.5 Translation from Maple to Fortran code .......... 137
A.6 AWK Program for generating Subroutine Headers .. 139
A.7 Automatically generated Fortran Code ............ 140
A.8 Generated Fortran Input File ....................... 141
A.9 Generated Fortran Output File ..................... 141
A.10 Library Routine Header ............................ 142
A.11 Library Routine Body .............................. 142
B.1 MSDL Lexical Analysis ............................ 168
B.2 MSDL Grammar Outline ............................ 170
B.3 MSDL Grammar implementation .................... 171
B.4 MSDL P2 Setup code example ..................... 172
B.5 Compiler internal vessel data structure .......... 174
C.1 Gnuplot program for plotting Data ................. 186
D.1 Example MSDL Listing for p3g-Setup .............. 187
D.2 Fortran Output Code for p3g-Setup ............... 188
E.1 Main function for DLSODAR ....................... 206
E.2 Root functions for DLSODAR ....................... 206
E.3 Example Program for using DLSODAR: Chem.f ...... 210
E.4 Output produced by the Example chem.f .......... 214
Bibliography


[BCnCR99] Lluís Batet, Sergio Chiva, José Luis Mu noz Cobo, and Francesc Reventós. Modelling activities of experimental facilities related to advanced reactors. considerations on
1d/3d issues. In *Proceedings of the 7th International Conference on Nuclear Engineering*, Tokyo, Japan, Apr 1999. ICONE, JSME.


[DCHD90b] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Sub-


[HDLA] M. Huggenberger, J. Dreier, S. Lomperski, and C. Aubert. PANDA Facility, Test Program and Data Base General Description. Paul Scherrer Institut, Villigen, Switzerland. ALPHA-606.


Curriculum Vitae

Frank Möhle

January 26, 1968  Born in Schwerte, Germany. Citizen of Switzerland

1974 – 1983  Primary / Secondary School, Peist and Chur, Switzerland

1983 – 1989  State College, Kantonsschule Chur, Chur, Switzerland
1989  Matura/Baccalaureate Typus C Kantonsschule Chur

1989 – 1995  Studies in Computer Science at the Swiss Federal Institute of Technology ETH Zurich, Switzerland

1995  Masters Diploma in Computer Science, ETH Zurich, Switzerland

1995 – 1999  Research and teaching assistant, Nuclear Engineering Laboratory, ETH Zurich, Switzerland

1999 – 2002  Head of the Computer Science Department of the University of Applied Sciences, Zurich

2003 – 2005  Scientific Collaborator and Project Manager at the Swiss Federal Archives, Berne

2005  Founder of cerrom engineering gmbh